Amortizing Maliciously Secure Two-party Computation

Roberto Trifiletti

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Amortizing Maliciously Secure Two-party Computation

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Roberto Trifiletti
October 4, 2017

Abstract

In this PhD dissertation we present new and improved techniques for secure two-party computation (2PC) with malicious security. Starting from the asymptotically efficient LEGO paradigm introduced by Nielsen & Orlandi (TCC 2009), we propose several optimizations and for the first time investigate its practical efficiency in detail. Our findings show that, in contrast to earlier beliefs, LEGO can be among the most practical techniques to date for 2PC using garbled circuits. In more detail

- We present a new UC-secure XOR-homomorphic commitment scheme specifically tailored to the LEGO context that is very close to being communication optimal. Our scheme is based on the non-homomorphic version of the commitment scheme of Cascudo *et al.* (PKC 2015). However, we manage to add the additive homomorphic property, while at the same time reducing the concrete communication overhead. As an example, when considering commitments to 128-bit strings with statistical security $s = 40$ our protocol sends 2.6x less data than the non-homomorphic version of Cascudo *et al.* and 75x less than their homomorphic version.
- Next, we present various optimizations of the recent TinyLEGO 2PC protocol of Frederiksen *et al.* (ePrint 2015), including a new efficient solution to the so-called selective OT-Attack. The resulting protocol is optimized for the offline/online setting, concretely efficient, runs in a constant number of rounds and supports the notion of function-independent preprocessing. Armed with the above commitment scheme we implement this LEGO protocol and find that it is highly competitive with all recent 2PC protocols based on garbled circuits.
- Finally, we generalize the idea of LEGO to any subcircuit size (LEGO considers only boolean gates). We show that this new DUPLO technique for 2PC can be superior to all previous techniques in the malicious setting, depending on the structure of the function to securely compute. Our experiments show that in various settings DUPLO is between 2-7x faster than any other 2PC protocol. In this way, we contribute with a highly versatile approach that is more efficient than any multi-execution protocol to date, while at the same time being superior in the single-execution setting for a rich class of functions.

Resumé

I denne PhD afhandling præsenterer vi nye og forbedrede teknikker til sikre to-parts beregninger (2PC) med aktiv sikkerhed. Med udgangspunkt i det asymptotisk effektive LEGO paradigme introduceret af Nielsen & Orlandi (TCC 2009), foreslår vi en række forbedringer og undersøger for første gang paradigmets praktiske effektivitet i detaljer. Vores resultater viser, i modsætning til den gængse opfattelse, at LEGO kan være blandt de mest praktiske teknikker til dato inden for 2PC der bruger krypterede kredsløb.

- Vi præsenterer et nyt UC-sikkert XOR-homomorft commitment system, der er specifikt designet til LEGO-konteksten og er meget tæt på at være optimalt i forhold til kommunikation. Vores system er baseret på det ikke-homomorfe commitment system af Cascudo *et al.* (PKC 2015). Vi lykkes dog at tilføje homomorfi og på samme tid reducere det konkrete kommunikationsoverhead. Som et eksempel, for commitments til 128-bit strenge med statistisk sikkerhed $s = 40$ sender vores protocol 2.6x mindre data end den ikke-homomorfe version af Cascudo *et al.*, og 75x mindre end deres homomorfe version.
- Vi fremlægger dernæst en række forbedringer af den nylige TinyLEGO 2PC protokol af Frederiksen *et al.* (ePrint 2015), inklusiv en ny effektiv løsning til det såkaldte selektive OT-Angreb. Dette medfører en protokol der er fintunet til offline/online modellen, effektiv i praksis, kører i et konstant antal runder og understøtter funktionsuafhængig præprocessering. Udstyret med ovenstående commitment system implementerer vi denne LEGO protokol og finder at den er konkurrencedygtig med alle nylige 2PC protokoller baseret på krypterede kredsløb.
- Slutteligt generaliserer vi LEGO-ideen til enhver sub-kredsløb størrelse (LEGO bruger kun boolske gates). Vi viser at denne nye DUPLO teknik til 2PC kan være alle tidligere teknikker i den aktive model overlegen, afhængig af strukturen af den funktion der skal beregnes sikkert. Vores eksperimenter viser at i nogle sammenhænge er DUPLO mellem 2-7x hurtigere end alle eksisterende 2PC protokoller. Vi bidrager dermed med en meget versatil teknik, der er mere effektiv end nogen anden multi-udførsel protokol, mens den på samme tid er overlegen i single-udførsel modellen for en rig klasse af funktioner.

Acknowledgement

Firstly, I would like to thank my main supervisor *Jesper Buus Nielsen* for the support and guidance provided throughout my PhD years and for taking me on as a student in the first place. I thank my co-supervisor *Ivan Bjerre Damgård* for his guidance and intriguing teaching which initially got me exited about cryptography. I am truly grateful to my former and current colleagues at the Aarhus University crypto group for making my research years so giving and fulfilling. I am proud to say I have worked alongside giants, both personally and professionally.

Further, I would like to thank *Thomas Schneider* for accepting my research proposal and hosting me on several occasions at TU Darmstadt. It has been a very fruitful experience, which has broadened my horizon on practical secure computation. To this end, I additionally credit *Michael Zohner* for our many discussions on the topic and his help in getting me started on implementing secure computation protocols.

This dissertation is indeed the result of a lot of hard work on my part, but it would never have been possible without the great contributions of my brilliant co-authors *Ignacio Cascudo, Ivan Bjerre Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, Tore Kasper Frederiksen, Thomas P. Jakobsen, Thomas Schneider, Vladimir Kolesnikov, Mike Rosulek, Ni Trieu,* and *Peter Rindal*. I am honored to have worked alongside each of you.

My greatest thanks however goes to my family, *Daniela* and *Sofia* (and *Emilio*-to-be), for putting up with me in the times of heavy workloads, high frequency traveling and with me being distant due to work-related matters – know that you are my greatest strength and reason for everything I do.

Roberto Trifiletti,
Aarhus, October 4, 2017.

Contents

Abstract **i**

Resumé **iii**

Acknowledgement **v**

Contents **vii**

I Overview **1**

1 Introduction **3**
1.1 Approaches to Secure Computation 4
1.2 Contributions and Timeline of the PhD Project 7

2 Preliminaries **11**
2.1 Notation 11
2.2 Universal Composability 12

II Publications **19**

3 On the Complexity of Additively Homomorphic UC Commitments **21**
3.1 Introduction 21
3.2 The Protocol 25
3.3 Security 32
3.4 Comparison with Recent Schemes 39
3.5 Protocol Extension 42

4 Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO **45**
4.1 Introduction 45
4.2 Preliminaries 52
4.3 A New Approach to Eliminate the Selective-OT Attack 54

4.4 The Protocol 57
4.5 Implementation 65
4.6 Performance 68

5 DUPLO: Unifying Cut-and-Choose for Garbled Circuits 77
5.1 Introduction 77
5.2 Preliminaries 83
5.3 Overview of the LEGO Paradigm 84
5.4 Overview of Our Construction 85
5.5 DUPLO Protocol Details 87
5.6 System Framework 91
5.7 Performance 95
5.8 Protocol Details 105
5.9 Analysis 125

Bibliography 133

Part I

Overview

Chapter 1

Introduction

Secure Multiparty Computation (MPC), originally introduced in 1982 by Andrew Yao [Yao82, Yao86], is a cryptographic technique allowing arbitrary computations on distributed data without revealing anything but the output to the participating parties. In more detail, the technology allows for n parties P_i for $i = 1, \ldots, n$, each holding some private data x_i, to perform any joint computation $f(x_1, x_2, \ldots, x_n)$ without revealing their private data to one another.

To concretise the above abstract description, consider todays recurring incidents of online large-scale data breaches (5,428 public incidents since 2005 [Cle17]). When it comes to keeping people's private data safe from intruders, MPC offers an elegant solution without sacrificing functionality for security. It is well-known that an effective counter-measure to data theft is to make the data unintelligible to begin with, for instance through encryption. If combined with the cryptographic key being stored in a separate secure location, this solution is highly effective as an attacker must then break into (at least) two distinct systems. These systems could potentially be situated at different physical locations running different hardware/software to further enhance security. Unfortunately, this approach has the major downside of disabling processing of the data while in encrypted form. Therefore, in order to decrypt and process the data, knowledge of the key is required by the processing system at some point in time, creating an opportunity for the attacker to target that system directly. To alleviate this issue, MPC can be applied and can achieve the best of both worlds: the data is stored encrypted at all times, but can still be processed by a system that has no knowledge of the decryption key. In brief, the MPC computation would be comprised of two parties, the processing system and the key-holder system. The computation f can then be defined to first decrypt the data, secondly compute the prescribed processing and optionally re-encrypt the data under a new key. As only the output of the processing is revealed to the parties, the data at rest is kept confidential, potentially towards both participating systems.

The above scenario is just one example of the applicability of secure computation, but illustrates the power and potential of MPC in todays' data-driven society. Although already introduced in the early 80s, it was only in 2008 that the first real world application of MPC was presented [BCD+09]. The project successfully ran a sealed bid auction using MPC among Danish sugar beet farmers and the only Danish sugar beet processor, Danisco, to securely compute the market clearing price. Since 2008, MPC has had several real world applications, *e.g.* in managing cryptographic keys [Sep, Dya], privacy preserving data analysis [Sha] and computing aggregate pay equity metrics by gender and ethnicity [BLJ+17].

The primary goal of this PhD research project has been to develop more efficient techniques for secure computation as many of todays' problems and data-set sizes are still out of reach for secure processing. It is therefore the hope that the techniques presented here will contribute to the joint effort of the secure computation community in making secure computation more applicable at a large scale. Examples could be in emerging data-intensive areas such as DNA processing, privacy-preserving machine learning and cloud computing. Before going into details of the contributions of the project we briefly survey and discuss the existing fundamental paradigms and techniques for secure computation.

1.1 Approaches to Secure Computation

Secure computation was initially proposed by Andrew Yao in the early 80s, introducing what is today known as Yao's garbled circuits for general-purpose secure two-party computation [Yao86]. The protocol offers semi-honest security, meaning that as long as no party deviates from the protocol specification, no information about the inputs are exposed (see Section 2.2 for more details on semi-honest security). Later, in 1987 Goldreich, Micali and Wigderson [GMW87, Gol04] presented the first protocol supporting an arbitrary number of parties and at the same time guaranteeing the stronger notion of malicious security. Essentially this guarantees that any protocol deviation is detected by the honest parties, and we again refer to Section 2.2 for more details on these adversarial models.

It is worth mentioning that both the protocol of [Yao86] and [GMW87] require complexity-theoretic assumptions to guarantee security, such as the existence of trapdoor permutations (see *e.g.* [KL14, p. 488] for a formal definition of this primitive). In contrast, the later proposed information-theoretic protocols of [BGW88, CCD88] does not require any such assumptions and still guarantee malicious security as long as at most one third of the parties are corrupt. An important distinction between these information-theoretic and the computational protocols of [Yao86, GMW87] is the domain of the computation, as the latter support computations over $\mathbb{F}_2$ only, whereas the

former support any (sufficiently large) finite field $\mathbb{F}$. Theoretically both of these representations can compute any function, but in practice the difference in performance can be significant. However as neither representation dominates the other, it is highly function-dependent which is best suited. As we will discuss in more detail in the following, there are many such trade-offs in the landscape of secure computation and one needs to carefully pick the protocol to match the application for maximal performance.

Since the initial theoretical discoveries of the 80s, significant research effort has gone into improving and innovating protocols and techniques for MPC. However, even after 30 years of active research, the most efficient protocols for *general-purpose* MPC are still based on the above fundamental protocols, indeed a testament to the deep insights of the pioneers of the field.

Paradigm Trade-offs

As can be seen from the above-mentioned protocols, there are many axes of properties to consider when choosing a MPC protocol for an application, including

Function Domain: Boolean or Arithmetic representation.
Security Guarantees: Semi-honest or Malicious.
Number of Supported Parties: Fixed or Arbitrary number.
Required Assumptions: General, specific or no assumption.

In this PhD dissertation, we present new techniques and insights improving practical efficiency of secure computation focusing on *maliciously* secure *two*-party computation of *boolean* functions under *complexity-theoretic* assumptions. With these restrictions in mind, the main paradigms for secure computation of boolean functions are GMW and Yao. We therefore briefly survey both approaches in terms of efficiency trade-offs, and highlight todays' state-of-the-art protocols that fall into these categories and summarize these in Table 1.1.

GMW. The GMW protocol shares many similarities to the previously mentioned arithmetic protocols of [BGW88, CCD88], as it also follows a secret-sharing based approach. In more detail, the parties initially secret-share their inputs and then *interactively* evaluate the function in order to obtain a secret-shared output. By interactively we mean that the protocol requires $\mathcal{O}(\mathsf{depth}(f))$ rounds of communication. Skipping many details, the highlights of the GMW approach is that it scales naturally to any number of parties, only uneven computation gates require interaction and it can be based on the general assumption of oblivious transfer (OT, see Figure 2.2 for more details on this primitive). Recent protocols following the GMW paradigm includes [CHK$^+$12, SZ13] for semi-honest security and [NNOB12, LOS14, BLN$^+$15] for malicious security.

	GMW		Yao	
	Semi-honest	Malicious	Semi-honest	Malicious
Overhead	$\mathcal{O}(\kappa)$	$\mathcal{O}(s\kappa/\log\lvert f\rvert)$	$\mathcal{O}(\kappa)$	$\mathcal{O}(s\kappa/\log\lvert f\rvert)$
# Rounds	$\mathcal{O}(\mathsf{depth}(f))$		$\mathcal{O}(1)$	
# Parties	n		n	
Free-XOR	✓		✓	
Preprocessing	✓		✓	
Assumption	OT		OT	

Table 1.1: Summarizing the main properties of the GMW and Yao paradigms for secure computation of boolean functions.

Yao. Unlike GMW, Yao's garbled circuits protocol is highly asymmetric, meaning that the two parties play distinct roles during execution. On a high level, one party (the constructor) initially "garbles" the function f, and through OT it offers the other party (the evaluator) to learn "encrypted" inputs for the computation. Finally, it sends its own encrypted inputs along with the garbling of f, which the evaluator can then use to evaluate the garbled circuit and recover the plaintext output. This approach also differs from GMW in that it only requires a constant amount of communication rounds, as opposed to increasing with the depth of f. Therefore, when it comes to high-latency networks, such as WANs, Yao's approach almost always outperforms GMW due to the overhead induced by the network latency. Although traditionally bounded to only two parties, the seminal result of [BMR90] presents a way of extending Yao's garbled circuits to an arbitrary number of parties as well. Until very recently, the [BMR90] protocol has mainly been considered a theoretical result, but the recent works of [LPSY15, LSS16] have presented several concrete optimizations to [BMR90]. Additionally, they have shown run-time experiments which indicate that for some settings, multi-party Yao is more efficient than multi-party GMW. We note, however, that the GMW approach seems to scale better than BMR in the number of parties and in most situations, multi-party GMW is still superior to multi-party Yao.

Several optimizations have been proposed over the years for Yao's garbled circuits, including the celebrated Free-XOR technique [KS08], Point-and-Permute [BMR90] and the garbled row reduction techniques of [NPS99, KMR14, ZRE15]. Recent protocols with semi-honest security based on Yao includes [SZ13, DSZ15]. For the case of malicious security, the canonical approach to making Yao actively secure is based on Cut-and-Choose (C&C) where the constructor sends $\mathcal{O}(s)$ garbled circuits to the evaluator instead of one. A fraction of these are then opened and verified to be correct, after which the remaining are used for evaluation, guaranteeing that no cheating can occur. In the past decade, improving concrete efficiency of malicious Yao has received a lot of attention from the secure computation community

and tremendous advances has been made, both asymptotically and concretely [LP07, NO09, PSSW09, LP11, sS11, HEKM11, KsS12, FJN+13, Bra13, FN13, HKE13, Lin13, MR13, sS13, HMsG13, FJN14, AMPR14, WMK17, KRW17b].

1.2 Contributions and Timeline of the PhD Project

The main contributions of this PhD dissertation are in the realm of general-purpose secure two-party computation based on Yao's garbled circuits with malicious security. However, several of the proposed techniques and optimizations are general and have already seen applications beyond two-party Yao.

At the onset of this research project the asymptotically best approach in the above context, using a constant amount of rounds, was the LEGO paradigm introduced by Nielsen & Orlandi in [NO09] and further optimized in [FJN+13]. This approach considers a gate-level approach to C&C which reduces the overhead from $\mathcal{O}(s\kappa)$ to $\mathcal{O}(s\kappa/\log|f|)$ where κ and s is the computational and statistical security parameter, respectively, and $|f|$ is the number of AND gates considered in the computation. Further, the LEGO paradigm naturally offers additional useful properties such as supporting function-independent processing, allowing reactive computations [NR16] and having a highly efficient online-phase. To this end, one would therefore expect LEGO to become superior to the other $\mathcal{O}(s\kappa)$ approaches as $|f|$ increases. However, back-of-the-envelope calculations indicated that the total overhead induced by this gate approach, albeit constant, was still orders of magnitude away from being competitive with the contemporary state-of-the-art protocols.

The high concrete overhead of LEGO stems from the usage of UC-secure XOR-homomorphic commitments on all circuit wires, which are needed to securely combine the garbled AND gates into a circuit computing f. Although [FJN+13] had made significant progress in reducing the commitment overhead over [NO09] it was still far from making LEGO competitive in practice. This was the main motivation for our work on improving practical efficiency of UC XOR-homomorphic commitments [CDD+15, FJNT16] appearing at PKC 2015 and TCC 2016-A, respectively. As [FJNT16] is a direct successor to [CDD+15], we here focus solely on the more efficient [FJNT16] which also constitute the third chapter of this dissertation. In short we present an extremely efficient UC-secure XOR-homomorphic commitment scheme, and furthermore present concrete optimizations directly applicable to the LEGO setting. In more detail, the following results are achieved

Chapter 3 We present the first construction of a UC-secure additively homomorphic commitment scheme that achieves close to optimal communication complexity in both the commitment and decommitment phase.

In particular, when (de)committing to binary strings of length k, our scheme requires $k+o(1)$ bits of communication.[1] The results are achieved by augmenting the non-homomorphic scheme of [CDD+15] with an efficient consistency check that relaxes the requirement on the minimum distance of underlying error correcting code, while at the same time achieving additive homomorphism. In addition, we consider a systematic representation of the constructed codewords which allows us to save k bits of communication per message when committing to random values. Concretely, when considering commitments to 128 bits with statistical security $s = 40$, our augmented scheme reduced communication by 2.6x over the non-homomorphic version of [CDD+15] and 75x compared to the homomorphic version of their scheme.

With the advent of an almost communication optimal XOR-homomorphic commitment scheme it was natural to investigate the LEGO paradigm as an immediate application. Starting from the MiniLEGO protocol [FJN+13] we made several optimizations to increase concrete efficiency, resulting in the protocol TinyLEGO [FJNT15]. Instantiated with the commitment scheme of [FJNT16], and based on an analysis of the required communication of the protocol it was conjectured that TinyLEGO could potentially be more efficient than standard C&C protocols for practical and realistic circuit sizes. To verify this claim, an implementation project was commenced with the goal of determining the exact performance of the TinyLEGO protocol. The results of that project is described in [NST17] which appeared at NDSS 2017 and forms the basis for the fourth chapter of this dissertation.

Chapter 4 We further optimize and implement the TinyLEGO protocol to investigate the practicality of LEGO. In addition to slightly modifying the original protocol to support several levels of preprocessing, we also propose a new efficient solution to the so-called selective OT-attack taking advantage of the homomorphic commitments at our disposal. This new technique requires a special OT primitive called $\mathcal{F}_{\Delta\text{-ROT}}$, or globally correlated OT which is similar to the $\mathcal{F}_{\mathsf{ROT}}$ functionality described in Figure 2.2, but with the distinction that all produced OT-strings are correlated with a global difference Δ.
The resulting implementation of TinyLEGO confirms the previous conjecture that the LEGO paradigm can indeed compete with, and in some cases out-perform state-of-the-art protocols for the same setting. It is also highlighted that one of the main strengths of LEGO is in the power to preprocess most of the communication and computation, even without knowledge of the function(s) to be computed.

[1] For any *UC-secure* commitment scheme it is known that the lower bound for both phases is k.

As an additional contribution we perform a tightened security analysis of the best known construction of $\mathcal{F}_{\Delta\text{-ROT}}$ [NNOB12, BLN+15], reducing the concrete communication overhead of $\mathcal{F}_{\Delta\text{-ROT}}$ by 5-6x. As a side-effect, this has a direct impact on the performance of the GMW-like protocol of [NNOB12, BLN+15] as $\mathcal{F}_{\Delta\text{-ROT}}$ constitute one of the main building blocks in that approach. Indeed, recent work has already taken advantage of this optimization (and applied several others) in an effort to merge the approaches of [NNOB12, BLN+15] with Yao's garbled circuits to further push the performance of maliciously secure computation forward [KRW17b, KRW17a, HSSV17].

Having investigated the practical efficiency of TinyLEGO in detail, it was concluded that LEGO was much more competitive than previously thought. However, in terms of *total* running time the LEGO paradigm could still not compare with the state-of-the-art protocols as it required ∼3x more communication for typical parameters and circuits. From the experiences obtained in the engineering and development of the [NST17] implementation it was apparent that the predominant cause of this overhead was due to the sheer amount of commitments required, *e.g.* more than 400 million commitments for 1024 AES-128 circuits. The above realization was the catalyst for the final and culminating project of this PhD dissertation, entitled *"DUPLO: Unifying Cut-and-Choose for Garbled Circuits"* [KNR+17] and is presented in our last chapter.

Chapter 5 In this work we present a new approach to C&C of garbled circuits inspired by the LEGO paradigm. Previous approaches have either considered C&C on the circuit-level or gate-level, while our work can be seen as a generalization that spans the entire continuum between these two extremes. Specifically, we lift the idea of LEGO to any number of distinct subcomponents of varying size. This has the major advantage of retaining most of the efficiency of LEGO, while at the same time requiring far less commitments. This is the case as there are now less overall input/output wires to stitch together. Our experimental evaluations show that this new approach scales better than any previous approach, yielding between 4-7x improvement on computations requiring several millions of gates. For certain functions we therefore see that DUPLO outperforms both multi-execution protocols and single-execution ones from moderate to large-scaled computations, while still retaining the ability to perform reactive computations.

In addition to the new protocol, we provide an extension to the Frigate circuit compiler [MGC+16] to easily allow programmers to express their functionality in a high-level C-style language. This is then compiled into a subcomponent circuit representation that is suitable for our new C&C technique.

Full List of Publications and Manuscripts

During my PhD studies I have been involved in several research projects involving maliciously secure two-party computation. The following chapters are based on the works of [FJNT16], [NST17] and [KNR+17]. For completeness, I here present a full list of the publications and unpublished manuscripts produced during my studies.

[CDD+15] Ignacio Cascudo, Ivan Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, and Roberto Trifiletti. Additively homomorphic UC commitments with optimal amortized overhead. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 495–515. Springer, March / April 2015

[FJNT15] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. Cryptology ePrint Archive, Report 2015/309, 2015. `https://eprint.iacr.org/2015/309`

[FJNT16] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 542–565. Springer, January 2016

[NST17] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *NDSS 2017*. The Internet Society, February 2017

[KNR+17] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. Cryptology ePrint Archive, Report 2017/344, 2017. `https://eprint.iacr.org/2017/344`

[RT17] Peter Rindal and Roberto Trifiletti. SplitCommit: Implementing and analyzing homomorphic UC commitments. Cryptology ePrint Archive, Report 2017/407, 2017. `https://eprint.iacr.org/2017/407`

Chapter 2

Preliminaries

The goal of secure two-party computation can informally be described as enabling two parties to compute a function in a black-box manner, meaning each party only provides their respective input and receives back the output. Another way of formulating this property is that the only information any party should learn is the output of the computation (and its own input). In order to formally argue about this property for a given cryptographic protocol Π the *simulation paradigm* is typically used:

We say a two-party protocol Π is simulatable for a party P that inputs x if all received data can be efficiently computed given the pair (x, z), where $z = f(x, y)$ and y is the input of the other party.

The above informal statement captures that the only thing learned from participating in a protocol Π is the input and output (x, z), since everything else in the protocol can be computed by knowledge of only this pair. The concept of simulation was first introduced by Goldwasser and Micali in [GM84] specifically for encryption schemes and was later extended to the more general interactive setting in [GMW87]. In the following, we set up some useful notation which enables us to formally describe how we prove security for secure two-party computation protocols.

2.1 Notation

The following section is comprised of the notation sections of [FJNT16, NST17] and has been copied nearly verbatim from those works.

We use κ and s to denote the computational and statistical security parameter, respectively. This means that for any fixed s and any polynomial time bounded adversary, the advantage of the adversary is $2^{-s} + \mathrm{negl}(\kappa)$ for a negligible function negl. In words, the advantage of any polynomial time bounded adversary goes to 2^{-s} faster than any inverse polynomial in the computational security parameter. For two ensembles $X = \{X_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$

and $Y = \{Y_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ of binary random variables we say these are *computationally indistinguishable*, denoted by $X \overset{c}{\approx} Y$, if for all z it holds that $|\Pr[X_{\kappa,z} = 1] - \Pr[Y_{\kappa,z} = 1]| \leq \text{negl}(\kappa)$. We write $\approx$ when the type of the indistinguishability notion is arbitrary. In all our specific experiments throughout the dissertation we set $\kappa = 128$ and $s = 40$.

We will use as shorthand $[n] = \{1, 2, \ldots, n\}$ and $[i; j] = \{i, i+1, i+2, \ldots, j\}$. We write $e \in_R S$ to mean: sample an element e uniformly at random from the finite set S. We will interchangeably use subscript and bracket notation to denote an index of a vector, *i.e.* x_i and $\boldsymbol{x}[i]$ denotes the i'th entry of a vector $\boldsymbol{x}$ which we always write in bold. When $\boldsymbol{r}$ and $\boldsymbol{m}$ are vectors, we write $\boldsymbol{r}\|\boldsymbol{m}$ to mean the vector that is the concatenation of $\boldsymbol{r}$ and $\boldsymbol{m}$. We write $z \leftarrow P(x)$ to mean: perform the (potentially randomized) procedure P on input x and store the output in variable z. We use $x := y$ to denote an assignment of x to y. Furthermore we use $\pi_{i,j}$ to denote a projection of a vector that extracts the entries from index i to index j, *i.e.* $\pi_{i,j}(\boldsymbol{x}) = (x_i, x_{i+1}, \ldots, x_j)$. Additionally, we use $\pi_l(\boldsymbol{x}) = \pi_{1,l}(\boldsymbol{x})$ as shorthand notation to denote the first l entries of $\boldsymbol{x}$.

2.2 Universal Composability

The de facto standard for proving security of secure computation protocols is the Universal Composability (UC) framework of [Can01] which in turn is based on the previous works of [GL91, Bea92, MR92, Can96, Can00]. It is defined using the Real World/Ideal World paradigm, which is based on the concept of simulation. The ideal world IDEAL is characterized by the presence of a non-corruptible ideal functionality $\mathcal{F}$ that carries out the task at hand for the participating parties. The real world REAL does not allow for such an ideal entity so instead the parties interact by following a cryptographic protocol Π. Security is achieved by showing that $\mathsf{REAL} \approx \mathsf{IDEAL}$ and we then say that Π UC-realizes $\mathcal{F}$. In this case, using the protocol Π is as secure as using the non-corruptible ideal functionality $\mathcal{F}$. One informal way of seeing this type of security definition, is to view $\mathcal{F}$ as a security specification that Π then satisfies.

The UC framework supports a powerful notion of composability. Indeed, the composability theorem of [Can01] states that if a protocol Π UC-realizes an ideal functionality $\mathcal{F}$, it can replace the role of $\mathcal{F}$ in any system while retaining all security properties. This is a tremendously important quality for deployment in todays' vast and interconnected computer systems, such as for instance the Internet. Another benefit of composability is that of modularity, as it allows us to split our often complex cryptographic protocols into smaller logical parts that are easier to analyze separately. Once we have proven security of such sub-functionalities in isolation we can combine them to solve the original problem with confidence that the composed system is secure as well. Specifically, this is done by modifying REAL to allow for the existence of (sub) ideal functionalities $\mathcal{G}$ that the parties can make use of in order to

realize the actual functionality $\mathcal{F}$. When this is the case we say we are in the $\mathcal{G}$-Hybrid model which we denote $\mathsf{HYBRID}^{\mathcal{G}}$.

Adversarial Behavior

We also need to address how we model adversarial behavior. This is done through the introduction of a new entity, the adversary $\mathcal{A}$. Depending on the security model the adversary is allowed a number of options of how to attack either one of the parties running the protocol.[1] Once a party has been influenced by $\mathcal{A}$ we say it is *corrupt* and its past and future state becomes known to $\mathcal{A}$. Classically we consider two main types of adversaries:

Semi-Honest In this model, when the adversary corrupts a party it obtains the current and future state of that entity. Specifically, the adversary is not allowed to influence the corrupt party to deviate from the protocol specification in any way. Thus the only "attack" possible in this model is to try and learn more information than the allowed input/output pair (x, z). This is a fairly weak type of security, but it still captures meaningful scenarios. For instance, say a secure computation has been carried out honestly, and at some later point in time one of the parties gets corrupted by $\mathcal{A}$. By semi-honest security it is now guaranteed that this breach reveals no information to $\mathcal{A}$ except for the input/output pair (x, z), as there is no way of influencing a protocol that has already terminated.

Malicious In the stronger malicious model, in addition to learning the state of the corrupt party, $\mathcal{A}$ is also allowed to "take control" and instruct it to behave in any arbitrary way throughout the protocol execution. Therefore when running a maliciously secure protocol the parties are guaranteed that such an "online" attack cannot cause any extra leakage beyond the allowed (x, z). As one would expect, malicious security is typically harder to achieve than semi-honest security. This is because extra work is usually needed from each party, in order to verify that no one is deviating from the protocol specification.

More recently, the *covert* model [AL07] has also been introduced and can be seen as offering security at a level in-between semi-honest and malicious. In this setting, the adversary is allowed to instruct the corrupt party to cheat, but it is then detected by the honest parties with probability at least ϵ (called the deterrent factor) for $0 \leq \epsilon \leq 1$. The motivation for such a security model is to capture applications where semi-honest security is too weak and malicious security is unnecessarily strong. In order to achieve better performance, one

[1] Note that we only allow $\mathcal{A}$ to attack a single party since if both parties ever get compromised security is always lost.

can therefore construct a covertly secure protocol that satisfies the security requirement but with much lower overhead than a maliciously secure one.

While the above notions model the capabilities of $\mathcal{A}$ once it has corrupted a party, the question of *when* this corruption can take place has, up until this point, not been addressed. In secure computation we typically allow for two types of such corruption patterns:

Static In this case, the adversary is only allowed to pick which party to corrupt *before* the protocol execution starts.

Adaptive In this stronger adaptive setting the adversary is allowed to wait arbitrarily until choosing which party to corrupt. Therefore, the choice can depend on the messages sent between the parties. Due to this potential dependence, and other more technical reasons, adaptive security is generally much harder to achieve than static security.

All protocols presented in this PhD dissertation are proven secure against a static and malicious adversary.

UC Environment

Recall that the notion of UC security is based on $\mathsf{REAL} \approx \mathsf{IDEAL}$. In order for this to be well-defined we also need to address for whom these executions look indistinguishable. In UC this role is played by the environment $\mathcal{Z}$ which interacts with either an execution of REAL or IDEAL in a black-box manner by giving input and receiving output. The idea is that $\mathcal{Z}$ models everything going on outside the protocol execution and because we put no restrictions on $\mathcal{Z}$ it can model any interconnected system. After $\mathcal{Z}$ has finished this interaction it is tasked with deciding which world it actually interacted with. Security is shown by proving that no $\mathcal{Z}$ can do much better than to guess, or more specifically that it only succeeds in this experiment with probability at most $1/2 + \mathrm{negl}(\kappa)$.

The above distinguisher role of $\mathcal{Z}$ is a common way of defining security in most areas of cryptology. What is particular about the UC setting however, and the reason for the strong security that it provides, is that the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ are allowed to communicate continuously throughout the experiment. This makes the security proof significantly harder as the run of REAL and IDEAL need to look identical at all times as else this would be detected by $\mathcal{Z}$. In contrast, security in the weaker *stand-alone* model only requires that the executions look indistinguishable at the end of the experiment. This is technically enforced by $\mathcal{A}$ not being allowed to communicate with the distinguisher. See *e.g.* [HL10] for a thorough treatment of stand-alone two-party computation.

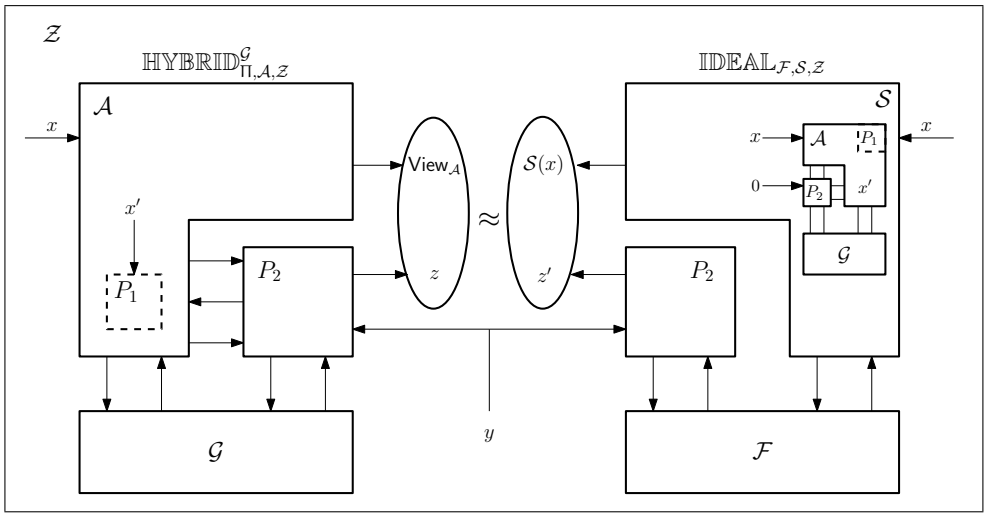

Figure 2.1: Illustration of the UC framework with corrupt P_1 for a protocol Π UC-realizing $\mathcal{F}$ in the $\mathcal{G}$-hybrid model.

Simulation

The simulator $\mathcal{S}$ constitutes the last entity in the UC framework. As part of the proof of security (that Π securely realizes $\mathcal{F}$), a concrete simulator is described that plays the role of any corrupt party in IDEAL. It is then the job of the simulator to ensure that $\mathsf{IDEAL} \approx \mathsf{REAL}$ from $\mathcal{Z}$'s point of view, no matter the instructions given by $\mathcal{A}$. In doing so, $\mathcal{S}$ communicates with the ideal functionality $\mathcal{F}$ on behalf of the corrupted party, say P_1 for concreteness. To keep $\mathcal{A}$ (and thus $\mathcal{Z}$) from noticing the simulation, $\mathcal{S}$ emulates an execution of the protocol Π playing the role of a "dummy" P_2 which interacts with the corrupted P_1. In order to prove UC security this emulated protocol execution must look indistinguishable to REAL given the prescribed input x for P_1, even though $\mathcal{A}$ might be instructing P_1 to use a different input x'. If the protocol in question is described in a hybrid world $\mathsf{HYBRID}^{\mathcal{G}}$, then $\mathcal{S}$ has full control of the (sub) ideal functionality $\mathcal{G}$ available to the parties running Π. This advantage is typically the reason why the simulator can succeed in its task, as from controlling $\mathcal{G}$ it is often possible to *extract* the effective input x'. After this extraction, $\mathcal{S}$ can forward x' to $\mathcal{F}$ in order to learn z. In addition, a correct extraction of x' ensures that the non-corrupt party, P_2, outputs the same value in both $\mathsf{HYBRID}^{\mathcal{G}}$ and IDEAL (*i.e.* that $z = z'$ in the specific example). In Figure 2.1 this scenario is pictorially described for a corrupt P_1.

Setup Assumptions. It is known that most "interesting" ideal functionalities, such as UC-secure commitments or OT cannot be obtained in the plain model [CF01]. In order to overcome this impossibility, UC-secure protocols require some kind of *setup assumption* that models a resource that is available

$\mathcal{F}_{\mathsf{ROT}}$ interacts with a sender P_s, a receiver P_r and an adversary $\mathcal{S}$ and it proceeds as follows:

Transfer: Upon receiving $(\texttt{transfer}, \texttt{sid}, \texttt{otid}, k)$ from both P_s and P_r, forward this message to $\mathcal{S}$ and wait for a reply. If $\mathcal{S}$ sends back $(\texttt{no-corrupt}, \texttt{sid}, \texttt{otid})$, sample $l^0, l^1 \in_R \{0,1\}^k$ and $b \in_R \{0,1\}$ and output $(\texttt{deliver}, \texttt{sid}, \texttt{otid}, (l^0, l^1))$ to P_s and $(\texttt{deliver}, \texttt{sid}, \texttt{otid}, (l^b, b))$ to P_r.
If $\mathcal{S}$ instead sends back $(\texttt{corrupt-sender}, \texttt{sid}, \texttt{otid}, (\tilde{l}^0, \tilde{l}^1))$ or $(\texttt{corrupt-receiver}, \texttt{sid}, \texttt{otid}, (\tilde{l}^{\tilde{b}}, \tilde{b}))$ and the sender, respectively the receiver is corrupted, proceed as above, but instead of sampling all values at random, use the values provided by $\mathcal{S}$.

Figure 2.2: Ideal Functionality $\mathcal{F}_{\mathsf{ROT}}$. Originally appearing in [FJNT16].

to the parties before the execution starts. Therefore the starting point for a UC protocol is usually in some form of $\mathcal{G}$-hybrid model. In this dissertation our starting point is in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model upon which we build a UC secure commitment scheme and several UC secure 2PC protocols. We describe the behavior of $\mathcal{F}_{\mathsf{ROT}}$ in Figure 2.2. For completeness the $\mathcal{F}_{\mathsf{ROT}}$ functionality can be efficiently UC-realized by *e.g.* the protocols of [PVW08] (relying on the setup assumption of a common reference string) and by using the OT-extension techniques of [Bea96, IKNP03, Nie07, NNOB12, Lar15, ALSZ15, KOS15].

Formal Definition

Having introduced all the entities in the UC framework we now summarize the above concepts in more formal and precise terms. In the following, we consider only the case of two parties and *computational* security. However we stress that the full framework supports an arbitrary amount of participating parties and other notions of security such as *statistical* and *perfect*. It was shown in [Cle86] that fairness and guaranteed output delivery is in general impossible to achieve for two-party protocols. Therefore, since all protocols in this dissertation fall into this category we allow $\mathcal{A}$ the ability to abort the execution at any time in the below definition, potentially, even before the non-corrupt party receives the output.

For a given protocol Π, we start by defining the random variables

$$\mathsf{HYBRID}^{\mathcal{G}}_{\Pi,\mathcal{A},i,\mathcal{Z}}(x,y,\kappa), \quad \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},i,\mathcal{Z}}(x,y,\kappa)$$

denoting the hybrid and ideal execution experiments. The output of these variables is the guess bit of $\mathcal{Z}$ where $i \in \{1,2\}$ specifies the party $\mathcal{A}$ corrupts, $x, y \in \{0,1\}^*$ are the inputs provided to P_1 and P_2, respectively, and $\kappa \in \mathbb{N}$ is the security parameter. Randomness is taken over the random coins of the

participating parties. This leads to the definition of the ensembles

$$\begin{aligned} \mathsf{HYBRID}^{\mathcal{G}}_{\Pi,\mathcal{A},i,\mathcal{Z}} &= \left\{\mathsf{HYBRID}^{\mathcal{G}}_{\Pi,\mathcal{A},i,\mathcal{Z}}(x,y,\kappa)\right\}_{x,y\in\{0,1\}^*,\kappa\in\mathbb{N}} \\ \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},i,\mathcal{Z}} &= \left\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},i,\mathcal{Z}}(x,y,\kappa)\right\}_{x,y\in\{0,1\}^*,\kappa\in\mathbb{N}} \end{aligned}$$

Finally, we assume that P_1 and P_2 always communicate over an authenticated channel, meaning that $\mathcal{A}$ and $\mathcal{S}$ can see the exchanged messages, but is unable to modify them (unless it is sent by a corrupt party). Further, it is implicitly the case that the parties enhance the communication channel to also guarantee confidentiality by adding symmetric-key encryption.[2] This implies that if there are no active corruptions, $\mathcal{A}$ cannot read or change any message, making simulation trivial for this case. With the above terminology in place we define UC-security for our particular setting as follows:

Definition 1 (Static, Malicious 2-party UC-Security)**.** A protocol Π securely realizes a functionality $\mathcal{F}$ with abort in the $\mathcal{G}$-hybrid world if for every probabilistic polynomial-time malicious and static adversary $\mathcal{A}$ in the hybrid world, there exists a probabilistic polynomial-time simulator $\mathcal{S}$ for the ideal world such that for $i \in \{0,1\}$:

$$\mathsf{HYBRID}^{\mathcal{G}}_{\Pi,\mathcal{A},i,\mathcal{Z}} \overset{c}{\approx} \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},i,\mathcal{Z}}$$

For a more detailed treatment of the UC-framework we refer the reader to [Can01].

[2] As we are in the computational setting, this is straight-forward to achieve.

Part II

Publications

Chapter 3

On the Complexity of Additively Homomorphic UC Commitments

The following chapter is based on the work of [FJNT16] and is therefore identical (except for minor layout modifications) to the current full version available at `https://eprint.iacr.org/2015/694`.

3.1 Introduction

Commitment schemes are the digital equivalent of a securely locked box: it allows a sender P_s to hide a secret from a receiver P_r by putting the secret inside the box, sealing it, and sending the box to P_r. As the receiver cannot look inside we say that the commitment is *hiding*. As the sender is unable to change his mind as he has given the box away we say the commitment is also *binding*. These simple, yet powerful properties are needed in countless cryptographic protocols, especially when guaranteeing security against a *malicious* adversary who can arbitrarily deviate from the protocol at hand. In the stand-alone model, commitment schemes can be made very efficient, both in terms of communication and computation and can be based entirely on the existence of one-way functions. These can *e.g.* be constructed from cheap symmetric cryptography such as pseudorandom generators [Nao90].

In this work we give an additively homomorphic commitment scheme secure in the UC-framework of [Can01], a model considering protocols running in a concurrent and asynchronous setting. The first UC-secure commitment schemes were given in [CF01, CLOS02] as feasibility results, while in [CF01] it was also shown that UC-commitments cannot be instantiated in the standard model and therefore require some form of setup assumption, such as a CRS. Moreover a construction for UC-commitments in such a model implies public-key cryptography [DG03]. Also, in the UC setting the previously mentioned

hiding and binding properties are augmented with the notions of *equivocality* and *extractability*, respectively. These properties are needed to realize the commitment functionality we introduce later on. Loosely speaking, a scheme is equivocal if a single commitment can be opened to any message using special trapdoor information. Likewise a scheme is extractable if from a commitment the underlying message can be extracted efficiently using again some special trapdoor information.

Based on the above it is not surprising that UC-commitments are significantly less efficient than constructions in the stand-alone model. Nevertheless a plethora of improvements have been proposed in the literature, *e.g.* [DN02, NFT09, Lin11, BCPV13, Fuj14, CJS14] considering different number theoretic hardness assumptions, types of setup assumption and adversarial models. Until recently, the most efficient schemes for the adversarial model considered in this work were that of [Lin11, BCPV13] in the CRS model and [HMQ04, CJS14] in different variations of the random oracle model [BR93].

Related Work. In [GIKW14] and independently in [DDGN14] it was considered to construct UC-commitments in the OT-hybrid model and at the same time confining the use of the OT primitive to a once-and-for-all setup phase. After the setup phase, the idea is to only use cheap symmetric primitives for each commitment thus amortizing away the cost of the initial OTs. Both approaches strongly resembles the "MPC-in-the-head" line of work of [IKOS07, HIKN08, IPS08] in that the receiver is watching a number of communication channels not disclosed to the sender. In order to cheat meaningfully in this paradigm the sender needs to cheat in many channels, but since he is unaware where the receiver is watching he will get caught with high probability. Concretely these schemes build on VSS and allow the receiver to learn an unqualified set of shares for a secret s. However the setup is such that the sender does not know which unqualified set is being "watched", so when opening he is forced to open to enough positions with consistent shares to avoid getting caught. The scheme of [GIKW14] focused primarily on the rate of the commitments in an asymptotic setting while [DDGN14] focused on the computational complexity. Furthermore the secret sharing scheme of the latter is based on Reed-Solomon codes and the scheme achieved both additive and multiplicative homomorphisms.

The idea of using OTs and error correction codes to realize commitments was also considered in [FJN+13] in the setting of two-party secure computation using garbled circuits. Their scheme also allowed for additively homomorphic operations on commitments, but requires a code with a specific privacy property. The authors pointed to [CC06] for an example of such a code, but it turns out this achieves quite low constant rate due to the privacy restriction. Care also has to be taken when using this scheme, as binding is not guaranteed for all committed messages. The authors capture this by allowing some message to

be "wildcards". However, in their application this is acceptable and properly dealt with.

In [CDD+15] a new approach to the above OT watch channel paradigm was proposed. Instead of basing the underlying secret sharing scheme on a threshold scheme the authors proposed a scheme for a particular access structure. This allowed realization of the scheme using additive secret sharing and any linear code, which achieved very good concrete efficiency. The only requirement of the code is that it is linear and the minimum distance is at least $2s + 1$ for statistical security s. To commit to a message m it is first encoded into a codeword c. Then each field element c_i of c is additively shared into two field elements c_i^0 and c_i^1 and the receiver learns one of these shares via an oblivious transfer. This in done in the watch-list paradigm where the same shares c_i^0 are learned for all the commitments, by using the OTs only to transfer short seeds and then masking the share c_i^0 and c_i^1 for all commitments from these pairs of seeds. This can be seen as reusing an idea ultimately going back to [Kil88, CvT95]. Even if the adversary commits to a string c' which is not a codeword, to open to another message, it would have to guess at least s of the random choice bits of the receiver. Furthermore the authors propose an additively homomorphic version of their scheme, however at the cost of using VSS which imposes higher constants than their basic non-homomorphic construction.

Finally in the very recent work of [CDD+16] the asymptotics of our proposed commitment scheme are improved as the authors give a protocol for additively homomorphic commitments that achieves linear time and close to rate 1. This work does not achieve the linear time property. Their protocol is however very similar in design to both [CDD+15] and the one presented here, but constructs and uses a binary linear time encodable code with non-trivial distance and rate close to 1 to achieve the mentioned results.

Motivation. As already mentioned, commitment schemes are extremely useful when security against a malicious adversary is required. With the added support for additively homomorphic operations on committed values even more applications become possible. One is that of maliciously secure two-party computation using the LEGO protocols of [NO09, FJN+13, FJNT15]. These protocols are based on cut-and-choose of garbled circuits and require a large amount of homomorphic commitments, in particular one commitment for each wire of all garbled gates. In a similar fashion the scheme of [AHMR15] for secure evaluation of RAM programs also make use of homomorphic commitments to transform garbled wire labels of one garbled circuit to another. Thus any improvement in the efficiency of homomorphic commitments is directly transferred to the above settings as well.

Our Contribution. We introduce a new, very efficient, additively homomorphic UC-secure commitment scheme in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model. Our scheme shows that:

1. The asymptotic complexity of additively homomorphic UC commitment is the same as the asymptotic complexity of non-homomorphic UC commitment, *i.e.*, the achievable rate is $1 - o(1)$. In particular, the homomorphic property comes for free.
2. In addition to being asymptotically optimal, our scheme is also more practical (smaller hidden constants) than any other existing UC commitment scheme, even non-homomorphic schemes and even schemes in the random oracle model.

In more detail our main contributions are as follows:

– We improve on the basic non-homomorphic commitment scheme of [CDD+15] by reducing the requirement of the minimum distance of the underlying linear code from $2s + 1$ to s for statistical security s. At the same time our scheme becomes additively homomorphic, a property not shared with the above scheme. This is achieved by introducing an efficient consistency check at the end of the commit phase, as described now. Assume that the corrupted sender commits to a string c' which has Hamming distance 1 to some codeword c_0 encoding message m_0 and has Hamming distance $s - 1$ to some other codeword c_1 encoding message m_1. For both the scheme in [CDD+15] and our scheme this means the adversary can later open to m_0 with probability $1/2$ and to m_1 with probability 2^{-s+1}. Both of these probabilities are considered too high as we want statistical security 2^{-s}. So, even if we could decode c' to for instance m_0, this might not be the message that the adversary will open to later. It is, however, the case that the adversary cannot later open to both m_0 and m_1, except with probability 2^{-s} as this would require guessing s of the random choice bits. The UC simulator, however, needs to extract which of m_0 and m_1 will be opened to already at commitment time. We introduce a new consistency check where we after the commitment phase ask the adversary to open $2s$ random linear combinations of the committed purported codewords. These linear combinations will with overwhelming probability in a well defined manner "contain" information about every dirty codeword c' and will force the adversary to guess some of the choice bits to successfully open them to some close codeword c. The trick is then that the simulator can extract which of the choice bits the adversary had to guess and that if we puncture the code and the committed strings at the positions at which the adversary guessed the choice bits, then the remaining strings can be proven to be codewords in the punctured code. Since the adversary guesses at most $s-1$ choice bits, except with negligible probability 2^{-s} we only need to puncture $s - 1$

positions, so the punctured code still has distance 1. We can therefore erasure decode and thus extract the committed message. If the adversary later open to another message he will have to guess additional choice bits, bringing him up to having guessed at least s choice bits. With the minimal distance lowered the required code length is also reduced and therefore also the amount of required initial OTs. As an example, for committing to messages of size $k = 256$ with statistical security $s = 40$ this amounts to roughly 33% less initial OTs than required by [CDD+15].

– We furthermore propose a number of optimizations that reduce the communication complexity by a factor of 2 for each commitment compared to [CDD+15] (without taking into account the smaller code length required). We give a detailed comparison to the schemes of [Lin11, BCPV13, CJS14] and [CDD+15] in Section 3.4 and show that for the above setting with $k = 256$ and $s = 40$ our new construction outperforms all existing schemes in terms of communication if committing to 319 messages or more while retaining the computational efficiency of [CDD+15]. This comparison includes the cost of the initial OTs. If committing to 10,000 messages or more we see the total communication is around 1/3 of [BCPV13], around 1/2 of the basic scheme of [CDD+15] and around 1/21 of the homomorphic version.
– Finally we give an extension of any additively homomorphic commitment scheme that achieves an amortized rate close to 1 in the opening phase. Put together with our proposed scheme and breaking a long message into many smaller blocks we achieve rate close to 1 in both the commitment and open phase of our protocol. This extension is interactive and is very similar in nature to the introduced consistency check for decreasing the required minimum distance. Although based on folklore techniques this extension allows for very efficiently homomorphic commitment to long messages without requiring correspondingly many OTs.

3.2 The Protocol

In Figure 3.1 we present the ideal functionality $\mathcal{F}_{\mathsf{HCOM}}$ that we UC-realize in this work. The functionality differs from other commitment functionalities in the literature by only allowing the sender P_s to decide the number of values he wants to commit to. The functionality then commits him to *random* values towards a receiver P_r and reveals the values to P_s. The reason for having the functionality commit to several values at a time is to reflect the batched nature of our protocol. That the values committed to are random is a design choice to offer flexibility for possible applications. In Section 3.5 we show an efficient black-box extension of $\mathcal{F}_{\mathsf{HCOM}}$ to chosen-message commitments.

$\mathcal{F}_{\mathsf{HCOM}}$ interacts with a sender P_s, a receiver P_r and an adversary $\mathcal{S}$, working over a finite field $\mathbb{F}$.

Init: Upon receiving a message $(\texttt{init}, \texttt{sid}, k)$ from both parties P_s and P_r, store the message length k.

Commit: Upon receiving a message $(\texttt{commit}, \texttt{sid}, \gamma)$ from P_s, forward this message to $\mathcal{S}$ and wait for a reply. If $\mathcal{S}$ sends back $(\texttt{no-corrupt}, \texttt{sid})$ proceed as follows:
Sample γ uniformly random values $\boldsymbol{r}_j \in \mathbb{F}^k$ and associate to each of these a unique unused identifier j and store the tuple $(\texttt{random}, \texttt{sid}, j, \boldsymbol{r}_j)$. We let $\mathcal{J}$ denote the set of these identifiers. Finally send $(\texttt{committed}, \texttt{sid}, \mathcal{J}, \{\boldsymbol{r}_j\}_{j\in\mathcal{J}})$ to P_s and $(\texttt{receipt}, \texttt{sid}, \mathcal{J})$ to P_r and $\mathcal{S}$.
If P_s is corrupted and $\mathcal{S}$ instead sends back $(\texttt{corrupt-commit}, \texttt{sid}, \{\tilde{\boldsymbol{r}}_j\}_{j\in\mathcal{J}})$, proceed as above, but instead of sampling the values at random, use the values provided by $\mathcal{S}$.

Open: Upon receiving a message $(\texttt{open}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C})$ from P_s, if for all $c \in C$, a tuple $(\texttt{random}, \texttt{sid}, c, \boldsymbol{r}_c)$ was previously recorded and $\alpha_c \in \mathbb{F}$, send $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C}, \sum_{c\in C} \alpha_c \cdot \boldsymbol{r}_c)$ to P_r and $\mathcal{S}$. Otherwise, ignore.

Figure 3.1: Ideal Functionality $\mathcal{F}_{\mathsf{HCOM}}$.

Protocol Π_{HCOM}

Our protocol Π_{HCOM} is cast in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model, meaning the parties are assumed access to the ideal functionality $\mathcal{F}_{\mathsf{ROT}}$ in Figure 2.2. The protocol UC-realizes the functionality $\mathcal{F}_{\mathsf{HCOM}}$ and is presented in full in Figure 3.3 and Figure 3.4. At the start of the protocol a once-and-for-all **Init** step is performed where P_s and P_r only need to know the size of the committed values k and the security parameters. We furthermore assume that the parties agree on a $[n, k, d]$ linear code $\mathcal{C}$ in systematic form over the finite field $\mathbb{F}$ and require that the minimum distance $d \geq s$ for statistical security parameter s. The parties then invoke n copies of the ideal functionality $\mathcal{F}_{\mathsf{ROT}}$ with the computational security parameter κ as input, such that P_s learns n pairs of κ-bit strings l_i^0, l_i^1 for $i \in [n]$, while P_r only learns one string of each pair. In addition to the above the parties also introduce a commitment counter $\mathcal{T}$ which simply stores the number of values committed to. Our protocol is phrased such that multiple commitment phases are possible after the initial ROTs have been performed, and the counter is simply incremented accordingly.

Next a **Commit** phase is described where at the end, P_s is committed to γ pseudorandom values. The protocol instructs the parties to expand the previously learned κ-bit strings, using a pseudorandom generator PRG, into row-vectors $\bar{\boldsymbol{s}}_i^b \in \mathbb{F}^{\mathcal{T}+\gamma+2s}$ for $b \in \{0, 1\}$ and $i \in [n]$. The reason for the extra $2s$ commitments will be apparent later. We denote by $\mathcal{J} = \{\mathcal{T}+1, \ldots, \mathcal{T}+\gamma+2s\}$ the set of indices of the $\gamma + 2s$ commitments being setup in this invocation of **Commit**. After the expansion P_s knows all of the above $2n$ row-vectors, while P_r only knows half. The parties then view these row-vectors as matrices

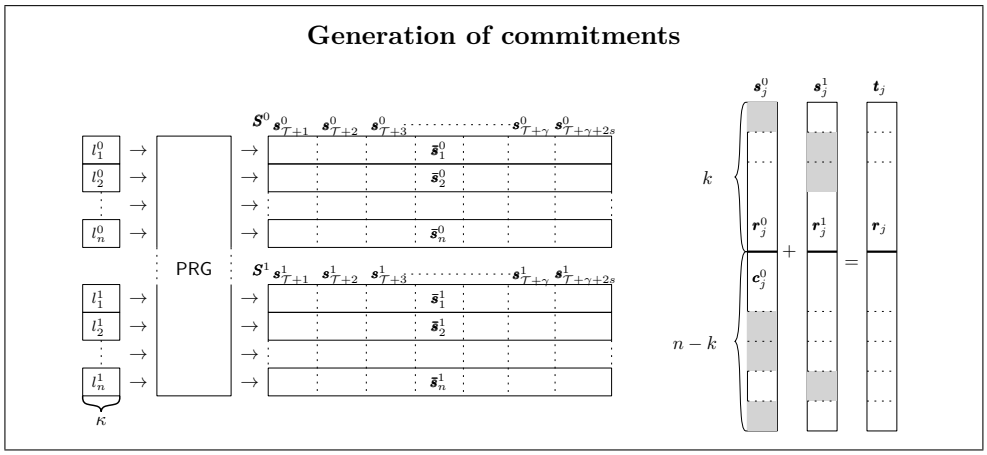

Figure 3.2: On the left hand side we see how the initial part of the **Commit** phase of Π_{HCOM} is performed by P_s when committing to γ messages. On the right hand side we look at a single column of the two matrices $\boldsymbol{S}^0, \boldsymbol{S}^1$ and how they define the codeword $\boldsymbol{t}_j$ for column $j \in \mathcal{J}$, where $\mathcal{J} = \{\mathcal{T}+1, \ldots, \mathcal{T}+\gamma+2s\}$.

$\boldsymbol{S}^0$ and $\boldsymbol{S}^1$ where row i of $\boldsymbol{S}^b$ consists of the vector $\bar{\boldsymbol{s}}_i^b$. We let $\boldsymbol{s}_j^b \in \mathbb{F}^n$ denote the j'th column vector of the matrix $\boldsymbol{S}^b$ for $j \in \mathcal{J}$. These column vectors now determine the committed pseudorandom values, which we define as $\boldsymbol{r}_j = \boldsymbol{r}_j^0 + \boldsymbol{r}_j^1$ where $\boldsymbol{r}_j^b = \pi_k(\boldsymbol{s}_j^b)$ for $j \in \mathcal{J}$. The above steps are also pictorially described in Figure 3.2.

The goal of the commit phase is for P_r to hold one out of two shares of each entry of a codeword of $\mathcal{C}$ that encodes the vector $\boldsymbol{r}_j$ for all $j \in \mathcal{J}$. At this point of the protocol, what P_r holds is however not of the above form. Though, because the code is in systematic form we have by definition that P_r holds such a sharing for the first k entries of each of these codewords. To ensure the same for the rest of the entries, for all $j \in \mathcal{J}$, P_s computes $\boldsymbol{t}_j \leftarrow \mathcal{C}(\boldsymbol{r}_j)$ and lets $\boldsymbol{c}_j^0 = \pi_{k+1,n}(\boldsymbol{s}_j^0)$. It then computes the correction value $\bar{\boldsymbol{c}}_j = \pi_{k+1,n}(\boldsymbol{t}_j) - \boldsymbol{c}_j^0 - \pi_{k+1,n}(\boldsymbol{s}_j^1)$ and sends this to P_r. Figure 3.2 also gives a quick overview of how these vectors are related.

When receiving the correction value $\bar{\boldsymbol{c}}_j$, we notice that for the columns $\boldsymbol{s}_j^0$ and $\boldsymbol{s}_j^1$, P_r knows only the entries $w_j^i = \boldsymbol{s}_j^{b_i}[i]$ where b_i is the choice-bit it received from $\mathcal{F}_{\mathsf{ROT}}$ in the i'th invocation. For all $l \in [n-k]$, if $b_{k+l} = 1$ it is instructed to update its entry as follows:

$$w_j^{k+l} := \bar{\boldsymbol{c}}_j[l] + w_j^{k+l} = \boldsymbol{t}_j[k+l] - \boldsymbol{c}_j^0[l] - \boldsymbol{s}_j^1[k+l] + w_j^{k+l} = \boldsymbol{t}_j[k+l] - \boldsymbol{c}_j^0[l] \ .$$

Due to the above corrections, it is now the case that for all $l \in [n-k]$ if $b_{k+l} = 0$, then $w_j^{k+l} = \boldsymbol{c}_j^0[l]$ and if $b_{k+l} = 1$, $w_j^{k+l} = \boldsymbol{t}_j[k+l] - \boldsymbol{c}_j^0[l]$. This means that at this point, for all $j \in \mathcal{J}$ and all $i \in [n]$, P_r holds exactly one out of two shares for each entry of the codeword $\boldsymbol{t}_j$ that encodes the vector $\boldsymbol{r}_j$.

The **Open** procedure describes how P_s can open to linear combinations of previously committed values. We let C be the indices to be opened and α_c for $c \in C$ be the corresponding coefficients. The sender then computes $\boldsymbol{r}^0 = \sum_{c \in C} \alpha_c \cdot \boldsymbol{r}_c^0$, $\boldsymbol{r}^1 = \sum_{c \in C} \alpha_c \cdot \boldsymbol{r}_c^1$, and $\boldsymbol{c}^0 = \sum_{c \in C} \alpha_c \cdot \boldsymbol{c}_c^0$ and sends these to P_r. When receiving the three values, the receiver computes the codeword $\boldsymbol{t} \leftarrow \mathcal{C}(\boldsymbol{r}^0 + \boldsymbol{r}^1)$ and from $\boldsymbol{c}^0$ and $\boldsymbol{t}$ it computes $\boldsymbol{c}^1$. It also computes $\boldsymbol{w} = \sum_{c \in C} \alpha_c \cdot \boldsymbol{w}_c$ and verifies that $\boldsymbol{r}^0, \boldsymbol{r}^1, \boldsymbol{c}^0$, and $\boldsymbol{c}^1$ are consistent with these. If everything matches it accepts $\boldsymbol{r}^0 + \boldsymbol{r}^1$ as the value opened to.

If the sender P_s behaves honestly in **Commit** of Π_{HCOM}, then the scheme is UC-secure as it is presented until now. In fact it is also additively homomorphic due to the linearity of the code $\mathcal{C}$ and the linearity of additive secret sharing. However, this only holds because P_r holds shares of valid codewords. If we consider a malicious corruption of P_s, then the shares held by P_r might not be of valid codewords, and then it is undefined at commitment time what the value committed to is. To see this consider a corrupt P_s that sends shares so that P_r holds shares of something that has *e.g.* distance $d-1$ to one codeword and distance 1 to another. Then it can open to any one of these values at a later time (although not both) with probability $1 - 2^{-d+1}$ or $1 - 2^{-1}$, neither of which are negligible when $d \geq s$. To achieve UC-security for a commitment scheme, the committed values need to be well-defined and *extractable* for a polynomial time simulator $\mathcal{S}$ at commitment time. It is therefore crucial for guaranteeing extractability that the shares P_r holds are in fact shares of a codeword, or ensure that he can only open successfully to a single well-defined codeword. In the following section we explain how our protocol ensures this.

Optimizations over [CDD+15]

The work of [CDD+15] describes two commitment schemes, a basic and a homomorphic version. For both schemes therein the above issue of sending correct shares is handled by requiring the underlying code $\overline{\mathcal{C}}$ with parameters $[\overline{n}, k, \overline{d}]$ to have minimum distance $\overline{d} \geq 2s + 1$, as then the committed values are always defined to be the closest valid codewords of the receivers shares. This is however not enough to guarantee binding when allowing homomorphic operations. To support this, the authors propose a version of the scheme that involves the sender P_s running a "MPC-in-the-head" protocol based on a verifiable secret sharing scheme of which the views of the simulated parties must be sent to P_r.

Up until now the scheme we have described is very similar to the basic scheme of [CDD+15]. The main difference is the use of $\mathcal{F}_{\mathsf{ROT}}$ as a starting assumption instead of $\mathcal{F}_{\mathsf{OT}}$ and the way we define and send the committed value corrections. In [CDD+15] the corrections sent are for both the 0 and the 1 share. This means they send $2\overline{n}$ field elements for each commitment in total. Having the code in systematic form implies that for all $j \in \mathcal{J}$ and $i \in [k]$ the entries $\boldsymbol{w}_j^i$ are already defined for P_r as part of the output of the PRG, thus

Π_{HCOM} describes a protocol between a sender P_s and a receiver P_r. We let $\mathsf{PRG} : \{0,1\}^\kappa \to \mathbb{F}^{\mathrm{poly}(\kappa)}$ be a pseudorandom generator with arbitrary polynomial stretch taking a κ-bit seed as input and outputting elements of a predetermined finite field $\mathbb{F}$.

Init:

1. On common input $(\texttt{init}, \texttt{sid}, k)$ we assume the parties agree on a linear code $\mathcal{C}$ in systematic form over $\mathbb{F}$ with parameters $[n, k, d]$. The parties also initialize an internal commitment counter $\mathcal{T} = 0$.
2. For $i \in [n]$, both parties send $(\texttt{transfer}, \texttt{sid}, i, \kappa)$ to $\mathcal{F}_{\mathsf{ROT}}$. It replies with $(\texttt{deliver}, \texttt{sid}, i, (l_i^0, l_i^1))$ to P_s and $(\texttt{deliver}, \texttt{sid}, i, (l_i^{b_i}, b_i))$ to P_r.

Commit:

1. On common input $(\texttt{commit}, \texttt{sid}, \gamma)$, for $i \in [n]$, both parties use PRG to extend their received seeds into vectors of length $\mathcal{T} + \gamma + 2s$. These are denoted $\bar{\boldsymbol{s}}_i^0, \bar{\boldsymbol{s}}_i^1 \in \mathbb{F}^{\mathcal{T}+\gamma+2s}$ where P_s knows both and P_r knows $\bar{\boldsymbol{s}}_i^{b_i}$. Next define the matrices $\boldsymbol{S}^0, \boldsymbol{S}^1 \in \mathbb{F}^{n \times (\mathcal{T}+\gamma+2s)}$ such that for $i \in [n]$ the i'th row of $\boldsymbol{S}^b$ is $\bar{\boldsymbol{s}}_i^b$ for $b \in \{0,1\}$.
2. Let $\mathcal{J} = \{\mathcal{T}+1, \ldots, \mathcal{T}+\gamma+2s\}$. For $j \in \mathcal{J}$ let the column vector of these matrices be $\boldsymbol{s}_j^0$, respectively $\boldsymbol{s}_j^1$. For $b \in \{0,1\}$, P_s lets $\boldsymbol{r}_j^b = \pi_k(\boldsymbol{s}_j^b)$ and lets $\boldsymbol{r}_j = \boldsymbol{r}_j^0 + \boldsymbol{r}_j^1$. Also P_r lets $\boldsymbol{w}_j = (w_j^1, \ldots, w_j^n)$ and $(b_1, \ldots, b_n) \leftarrow \boldsymbol{b}$ where $w_j^i = \boldsymbol{s}_j^{b_i}[i]$ for $i \in [n]$.
3. For $j \in \mathcal{J}$, P_s computes $\boldsymbol{t}_j \leftarrow \mathcal{C}\left(\boldsymbol{r}_j\right)$ and lets $\boldsymbol{c}_j^0 = \pi_{k+1,n}(\boldsymbol{s}_j^0)$. It then computes the correction value $\bar{\boldsymbol{c}}_j = \pi_{k+1,n}(\boldsymbol{t}_j) - \boldsymbol{c}_j^0 - \pi_{k+1,n}(\boldsymbol{s}_j^1)$.
4. Finally P_s sends the set $\left\{\bar{\boldsymbol{c}}_j\right\}_{j\in\mathcal{J}}$ to P_r. For $l \in [n-k]$ if $b_{k+l} = 1$, P_r updates $w_j^{k+l} := \bar{\boldsymbol{c}}_j[l] + w_j^{k+l}$.

Consistency Check

5. For $g \in [2s]$ P_r samples $x_1^g, \ldots, x_\gamma^g \in_R \mathbb{F}$ and sends these to P_s.
6. P_s then computes

$$\tilde{\boldsymbol{r}}_g^0 = \boldsymbol{r}_{\mathcal{T}+\gamma+g}^0 + \sum_{j=1}^{\gamma} x_j^g \boldsymbol{r}_{\mathcal{T}+j}^0, \quad \tilde{\boldsymbol{r}}_g^1 = \boldsymbol{r}_{\mathcal{T}+\gamma+g}^1 + \sum_{j=1}^{\gamma} x_j^g \boldsymbol{r}_{\mathcal{T}+j}^1,$$

$$\tilde{\boldsymbol{c}}_g^0 = \boldsymbol{c}_{\mathcal{T}+\gamma+g}^0 + \sum_{j=1}^{\gamma} x_j^g \boldsymbol{c}_{\mathcal{T}+j}^0$$

and sends the $2s$ tuples $(\tilde{\boldsymbol{r}}_g^0, \tilde{\boldsymbol{r}}_g^1, \tilde{\boldsymbol{c}}_g^0)$ to P_r.

Figure 3.3: Protocol Π_{HCOM} UC-realizing $\mathcal{F}_{\mathsf{HCOM}}$ in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model – part 1.

Commit (continued):

7. For $g \in [2s]$ P_r computes $\tilde{\boldsymbol{w}}_g = \boldsymbol{w}_{\mathcal{T}+\gamma+g} + \sum_{j=1}^{\gamma} x_j^g \boldsymbol{w}_{\mathcal{T}+j}$ and $\tilde{\boldsymbol{t}}_g \leftarrow \mathcal{C}\left(\tilde{\boldsymbol{r}}_g^0 + \tilde{\boldsymbol{r}}_g^1\right)$. It lets $\tilde{\boldsymbol{c}}_g \leftarrow \pi_{k+1,n}(\tilde{\boldsymbol{t}}_g)$ and lets $\tilde{\boldsymbol{c}}_g^1 = \tilde{\boldsymbol{c}}_g - \tilde{\boldsymbol{c}}_g^0$. Finally for $u \in [k]$ and $v \in [n-k]$, P_r verifies that

$$\tilde{\boldsymbol{r}}_g^{b_u}[u] = \tilde{\boldsymbol{w}}_g[u] \ , \ \tilde{\boldsymbol{c}}_g^{b_{k+v}}[v] = \tilde{\boldsymbol{w}}_g[k+v] \ .$$

   If any of the $2s$ checks fail P_r outputs abort and halts.

Output

8. Both parties increment their local counter $\mathcal{T} := \mathcal{T} + \gamma$. P_s now holds opening information $\left\{(\boldsymbol{r}_j^0, \boldsymbol{r}_j^1, \boldsymbol{c}_j^0)\right\}_{j\in[\mathcal{T}]}$ and P_r holds the verifying information $\left\{\boldsymbol{w}_j\right\}_{j\in[\mathcal{T}]}$. Let $\overline{\mathcal{J}} = \mathcal{J} \setminus \{\mathcal{T}+\gamma+2s\}$. P_s outputs $(\texttt{committed}, \texttt{sid}, \overline{\mathcal{J}}, \{\boldsymbol{r}_j\}_{j\in\overline{\mathcal{J}}})$ and P_r outputs $(\texttt{receipt}, \texttt{sid}, \overline{\mathcal{J}})$.[a]

Open:

1. On input $(\texttt{open}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C})$ where each $\alpha_c \in \mathbb{F}$, if for all $c \in C$, P_s holds $(\boldsymbol{r}_c^0, \boldsymbol{r}_c^1, \boldsymbol{c}_c^0)$ it computes

$$\boldsymbol{r}^0 = \sum_{c\in C} \alpha_c \cdot \boldsymbol{r}_c^0, \quad \boldsymbol{r}^1 = \sum_{c\in C} \alpha_c \cdot \boldsymbol{r}_c^1, \quad \boldsymbol{c}^0 = \sum_{c\in C} \alpha_c \cdot \boldsymbol{c}_c^0$$

   and sends $(\texttt{opening}, \{c, \alpha_c\}_{c\in C}, (\boldsymbol{r}^0, \boldsymbol{r}^1, \boldsymbol{c}^0))$ to P_r. Else it ignores the input message.
2. Upon receiving the message $(\texttt{opening}, \{c, \alpha_c\}_{c\in C}, (\boldsymbol{r}^0, \boldsymbol{r}^1, \boldsymbol{c}^0))$ from P_s, if for all $c \in C$, P_r holds $\boldsymbol{w}_c$ it lets $\boldsymbol{r} = \boldsymbol{r}^0 + \boldsymbol{r}^1$ and computes

$$\boldsymbol{w} = \sum_{c\in C} \alpha_c \cdot \boldsymbol{w}_c, \quad \boldsymbol{t} \leftarrow \mathcal{C}(\boldsymbol{r}) \ .$$

   It lets $\boldsymbol{c} = \pi_{k+1,n}(\boldsymbol{t})$ and computes $\boldsymbol{c}^1 = \boldsymbol{c} - \boldsymbol{c}^0$.
   Finally for $i \in [k]$ and $l \in [n-k]$, P_r verifies that

$$\boldsymbol{r}^{b_i}[i] = \boldsymbol{w}[i] \ , \ \boldsymbol{c}^{b_{k+l}}[l] = \boldsymbol{w}[k+l] \ .$$

   If all checks are valid P_r outputs $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C}, \boldsymbol{r})$. Else it aborts and halts.

[a] For clarity we assume that P_s and P_r locally discard the $2s$ extra commitments used for blinding. This includes the bookkeeping of the index offset this creates when generating multiple batches.

Figure 3.4: Protocol Π_{HCOM} UC-realizing $\mathcal{F}_{\mathsf{HCOM}}$ in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model – part 2.

saving $2k$ field elements of communication per commitment. Together with only sending corrections to the 1-share, we only need to send $n - k$ field elements as corrections. Meanwhile this only commits the sender to a pseudorandom value, so to commit to a chosen value another correction of k elements needs to be sent. In total we therefore save a factor 2 of communication from these optimizations.

However the main advantage of our approach comes from ensuring that the shares held by P_r binds the sender P_s to his committed value, while only requiring a minimum distance of s. On top of that our approach is also additively homomorphic. The idea is that P_r will challenge P_s to open $2s$ random linear combinations of all the committed values and check that these are valid according to $\mathcal{C}$. Recall that $\gamma + 2s$ commitments are produced in total. The reason for this is to guarantee hiding for the commitments, even when P_r learns a random linear combinations of these. Therefore, each linear combination is "blinded" by a pseudorandom value only used once and thus it appears pseudorandom to P_r as well. This is the reason committing to $2s$ additional values for each invocation of **Commit**.

The intuition why the above approach works is that if the sender P_s sends inconsistent corrections, it will get challenged on these positions with high probability. In order to pass the check, P_s must therefore guess which choice-bit P_r holds for each position for which it sent inconsistent values. The random linear combinations therefore force P_s to make a decision at commitment time which underlying value to send consistent openings to, and after that it can only open to that value successfully. In fact, the above approach also guarantees that the scheme is homomorphic. This is because all the freedom P_s might have had by sending maliciously constructed corrections is removed already at commitment time *for all values*, so after this phase commitments and shares can be added together without issue.

To extract *all* committed values when receiving the openings to the linear combinations the simulator identifies which rows of $\boldsymbol{S}^0$ and $\boldsymbol{S}^1$ P_s is sending inconsistent shares for. For these positions it inserts erasures in all positions of $\boldsymbol{t}_j$ (as defined by $\boldsymbol{S}^0, \boldsymbol{S}^1, \tilde{\boldsymbol{c}}_j$ and $\mathcal{C}$). As there are at most $s - 1$ positions where P_s could have cheated and the distance of the linear code is $d \geq s$ the simulator can erasure decode all columns to a unique value, and this is the only value P_s can successfully open to.[1]

Protocol Extension

The protocol Π_{HCOM} implements a commitment scheme where the sender commits to pseudorandom values. In many applications however it is needed to commit to chosen values instead. It is know that for any UC-secure commitment scheme one can easily turn a commitment from a random value

[1] All linear codes can be efficiently erasure decoded if the number of erasures is $\leq d - 1$.

into a commitment of a chosen one using the random value as a one-time pad encryption of the chosen value. For completeness, in Section 3.5, we show this extension for any protocol implementing $\mathcal{F}_{\mathsf{HCOM}}$.

In addition we also highlight that all additively homomorphic commitment schemes support the notion of batch-opening. For applications where a large amount of messages need to be opened at the same time this has great implications on efficiency. The technique is roughly that P_s sends the values he wants to open directly to P_r. To verify correctness the receiver then challenges the sender to open to $\hat{s} = s/\log_2(|\mathbb{F}|)$ random linear combinations of the received messages. It is easy to see that if for any commitment the sender sent a message different from the committed one, then a random linear combination of the commitments will commit to a message different from the same random linear combination of the claimed message except with probability $1/|\mathbb{F}|$. Therefore the sender is caught except with probability $(1/|\mathbb{F}|)^{\hat{s}} = 2^{-s}$. Using this method the overhead of opening the commitments is independent of the number of messages opened to and therefore amortizes away in the same manner as the consistency check and the initial OTs. However this way of opening messages has the downside of making the opening phase interactive, which is not optimal for all applications. See Section 3.5 for details.

The abovementioned batch-opening technique also has applicability when committing to large messages. Say we want to commit to a message m of length M. The naive approach would be to instantiate our scheme using a $[n_M, M, s]$ code. However this would require $n_M \geq M$ initial OTs and in addition only achieve rate $M/(M+n_M) \geq 1/2$ in the opening phase. Instead of the above, the idea is to break the large message of length M into blocks of length l for $l << M$. There will now be $N = \lceil M/l \rceil$ of these blocks in total. We then instantiate our scheme with a $[n_s, l, s]$ code and commit to m in blocks of size l. When required to open we use the above-mentioned batch-opening to open all N blocks of m. It is clear that the above technique remains additively homomorphic for commitments to the large messages. In [GIKW14] they show an example for messages of size 2^{30} where they achieve rate $1.046^{-1} \approx 0.95$ in both the commit and open phase. In Section 3.5 we apply our above approach to the same setting and conclude that in the commit phase we achieve rate ≈ 0.974 and even higher in the opening phase. This is including the cost of the initial OTs.

3.3 Security

In this section we prove the following theorem.

Theorem 1. The protocol Π_{HCOM} in Figure 3.3 and Figure 3.4 UC-realizes the $\mathcal{F}_{\mathsf{HCOM}}$ functionality of Figure 3.1 in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model against any number of static corruptions.

In the proof we will need a technical lemma, which we state and prove first. Let $\mathbb{F}$ be a finite field and let $\mathcal{C}$ be a $\mathbb{F}$-linear code with parameters $[n, k, d]$. Let $\mathcal{C}^{\odot m} \subset \mathbb{F}^{n\times m}$ consist of the set of matrices for which each column is from $\mathcal{C}$. We can think of $\mathcal{C}^{\odot m}$ as a linear code of length n with symbols from $\mathbb{F}^m$. For a matrix $M \in \mathbb{F}^{n\times m}$ we use $\|M\|_0$ to denote the number of rows of M which are not all-zero. This is also the Hamming weight of M when viewed as a n long vector of m-bit symbols. The minimum distance of $\mathcal{C}^{\odot m}$ is then $d' = \min\{\|M\|_0 \mid M \in \mathcal{C}^{\odot m}\}$. It is easy to see that $d' \geq d$.

We can view each matrix $H \in \mathbb{F}^{m\times \ell}$ as specifying a linear function $H : \mathbb{F}^{n\times \ell} \to \mathbb{F}^{n\times m}$ by $M \mapsto M \circ H^\top$, where $\circ$ denotes matrix product and $\top$ denotes transposition of a matrix; We write $M' = H(M)$. Notice that if $M \in \mathcal{C}^{\odot \ell}$ and $M' = H(M)$, then $M' \in \mathcal{C}^{\odot m}$.

For a matrix $M \in \mathbb{F}^{n\times m}$ we let $D_{\mathcal{C}}(M) = \{D \in \mathbb{F}^{n\times m} \backslash \mathcal{C}^{\odot m} \mid M+D \in \mathcal{C}^{\odot m}\}$ denote the set of possible errors D that would explain M as a codeword plus the error D. For a matrix $D \in \mathbb{F}^{n\times m}$ we let $\delta(D) \subseteq [n]$ be the set which contains i iff row i in D is not all-zero. We let $\Delta_{\mathcal{C}}(M) = \{\delta(D) \mid D \in D_{\mathcal{C}}(M)\}$. This is the set of possible error positions E that would explain M as a codeword plus errors in the rows $i \in E$. We let $d_{\mathcal{C}}(M) = \min\{|E| \mid E \in \Delta_{\mathcal{C}}(M)\}$ with $\delta_{\mathcal{C}}(M) = 0$ if $M \in \mathcal{C}^{\odot m}$. This is the Hamming distance of M to the code $\mathcal{C}^{\odot m}$.

Lemma 1. There exist a property $P : \mathbb{F}^{2d\times \ell} \times \mathbb{F}^{n\times \ell} \to \{\bot, \top\}$ such that for all $H \in \mathbb{F}^{2d\times \ell}$ and all $M \in \mathbb{F}^{n\times \ell}$ one of the following conditions are true:

1. $P(H, M) = \top$,
2. $\Delta_{\mathcal{C}}(H(M)) \subseteq \Delta_{\mathcal{C}}(M)$,
3. $\delta_{\mathcal{C}}(H(M)) \geq d$.

Furthermore, for a fixed $M \in \mathbb{F}^{n\times \ell}$ and a uniformly random $H \in \mathbb{F}^{2d\times \ell}$ it holds that $P(H, M) = \top$ with probability at most 2^{-d}.

In words the lemma says that if you fix a matrix $M \in \mathbb{F}^{n\times \ell}$ and pick a uniformly random matrix $H \in \mathbb{F}^{2d\times \ell}$ and compute $M' = H(M) \in \mathbb{F}^{n\times 2d}$, then except with probability 2^{-d} the result M' will have maximal distance d to $\mathcal{C}^{\odot 2d}$ or any subset of rows $E \subseteq [n]$ which allows to see all the columns of M' as codewords plus errors only in rows $i \in E$ will have the property that we can also see all the columns of M as codewords plus errors only in rows $i \in E$. If we have $\ell > 2d$ this therefore gives us a way to boil down all the errors in M into the much shorter M'.

Proof of Lemma 1. First note that each $H \in \mathbb{F}^{2d\times \ell}$ can be seen as a function $H : \mathbb{F}^\ell \to \mathbb{F}^{2d}$ by $x \mapsto Hx$. It is well known and easy to check that if we sample a uniformly random $H \in \mathbb{F}^{2d\times \ell}$, then we have a 2-universal hash function family, which means that it in particular is a 0-almost 2-universal hash function family. Then use Theorem 1 from [CDD+16] with $t = 0$. □

Proof of Theorem 1. We prove security for the case with a dummy adversary, so that the simulator is outputting simulated values directly to the environment and is receiving inputs directly from the environment. We focus on the case with one call to **Commit**. The proof trivially lifts to the case with multiple invocations. The case with two static corruptions is trivial. The case with no corruptions follows from the case with a corrupted receiver, as in the ideal functionality $\mathcal{F}_{\mathsf{HCOM}}$ the adversary is given all values which are given to the receiver, so one can just simulate the corrupted receiver and then output only the public transcript of the communication to the environment. We now first prove the case with a corrupted receiver and then the case with a corrupted sender.

Assume that P_r is corrupted. We use $\breve{P}_r$ to denote the corrupted receiver. This is just a mnemonic pseudonym for the environment $\mathcal{Z}$. The main idea behind the simulation is to simply run honestly until the opening phase. In the opening phase we then equivocate the commitment to the value received from the ideal functionality $\mathcal{F}_{\mathsf{HCOM}}$ by adjusting the bits $\bar{\boldsymbol{s}}_j^{1-b_i}$ not being watched by the receiver. This will be indistinguishable from the real world as the vectors $\bar{\boldsymbol{s}}_i^{1-b_i}$ are indistinguishable from uniform in the view of $\breve{P}_r$ and if all the vectors $\bar{\boldsymbol{s}}_i^{1-b_i}$ were uniform, then adjusting the bits not watched by $\breve{P}_r$ would be perfectly indistinguishable.

We first describe how to simulate the protocol without the step *Consistency Check*. We then discuss how to extend the simulation to this case.

The simulator $\mathcal{S}$ will run **Init** honestly, simulating $\mathcal{F}_{\mathsf{ROT}}$ to $\breve{P}_r$. It then runs **Commit** honestly. On input $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C}, \boldsymbol{r})$ it must simulate an opening.

In the simulation we use the fact that in the real protocol P_r can recompute all the values received from P_s given just the value $\boldsymbol{r}$ and the values $\boldsymbol{w}_c$, which it already knows, and assuming that the checks

$$\boldsymbol{r}^{b_i}[i] = \boldsymbol{w}[i] \ , \ \boldsymbol{c}^{b_{k+l}}[l] = \boldsymbol{w}[k+l]$$

at the end of Figure 3.4 are true. This goes as follows: First compute

$$\boldsymbol{w} = \sum_{c\in C} \alpha_c \cdot \boldsymbol{w}_c, \quad \boldsymbol{t} = \mathcal{C}(\boldsymbol{r}) \ , \ \boldsymbol{c} = \pi_{k+1,n}(\boldsymbol{t}) \ , \tag{3.1}$$

as in the protocol. Then for $i \in [k]$ and $l \in [n-k]$ define

$$\boldsymbol{r}^{b_i}[i] = \boldsymbol{w}[i] \ , \ \boldsymbol{c}^{b_{k+l}}[l] = \boldsymbol{w}[k+l] \ . \tag{3.2}$$

$$\boldsymbol{r}^{1-b_i}[i] = \boldsymbol{r}[i] - \boldsymbol{r}^{b_i}[i] \ , \ \boldsymbol{c}^{1-b_{k+l}}[l] = \boldsymbol{c}[l] - \boldsymbol{c}^{b_{k+l}}[l] \ . \tag{3.3}$$

In (3.2) we use that the checks are true. In (3.3) we use that $\boldsymbol{r} = \boldsymbol{r}^0 + \boldsymbol{r}^1$ and $\boldsymbol{c}^1 = \boldsymbol{c} - \boldsymbol{c}^0$ by construction of P_r. This clearly correctly recomputes $(\boldsymbol{r}^0, \boldsymbol{r}^1, \boldsymbol{c}^0)$.

On input $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c \in C}, \boldsymbol{r})$ from $\mathcal{F}_{\mathsf{HCOM}}$, the simulator will compute $(\boldsymbol{r}^0, \boldsymbol{r}^1, \boldsymbol{c}^0)$ from $\boldsymbol{r}$ and the values $\boldsymbol{w}_c$ known by $\breve{P}_r$ as above and send $(\texttt{opening}, \{c, \alpha_c\}_{c \in C}, (\boldsymbol{r}^0, \boldsymbol{r}^1, \boldsymbol{c}^0))$ to $\breve{P}_r$.

We now argue that the simulation is computationally indistinguishable from the real protocol. We go via two hybrids.

We define *Hybrid I* as follows. Instead of computing the rows $\bar{\boldsymbol{s}}_i^{1-b_i}$ from the seeds $l_i^{1-b_i}$ the simulator samples $\bar{\boldsymbol{s}}_i^{1-b_i}$ uniformly at random of the same length. Since $\breve{P}_r$ never sees the seeds $l_i^{1-b_i}$ and P_s only uses them as input to PRG, we can show that the view of $\breve{P}_r$ in the simulation and Hybrid I are computationally indistinguishable by a black box reduction to the security of PRG.

We define *Hybrid II* as follows. We start from the real protocol, but instead of computing the rows $\bar{\boldsymbol{s}}_i^{1-b_i}$ from the seeds $l_i^{1-b_i}$ we again sample $\bar{\boldsymbol{s}}_i^{1-b_i}$ uniformly at random of the same length. As above, we can show that the view of $\breve{P}_r$ in the protocol and Hybrid II are computationally indistinguishable.

The proof then concludes by transitivity of computational indistinguishability and by observing that the views of $\breve{P}_r$ in Hybrid I and Hybrid II are perfectly indistinguishable. The main observation needed for seeing this is that in Hybrid I all the bits $\boldsymbol{r}_j[i]$ are chosen uniformly at random and independently by $\mathcal{F}_{\mathsf{HCOM}}$, whereas in Hybrid II they are defined by $\boldsymbol{r}_j[i] = \boldsymbol{r}_j^0[i] + \boldsymbol{r}_j^1[i] = \boldsymbol{r}_j^{b_i}[i] + \boldsymbol{r}_j^{1-b_i}[i]$, where all the bits $\boldsymbol{r}_j^{1-b_i}[i]$ are chosen uniformly at random and independently by $\mathcal{S}$. This yields the same distributions of the values $\boldsymbol{r}_j$. All other value clearly have the same distribution.

We now address the step *Consistency Check*. The simulation of this step follows the same pattern as above. For $g \in [2s]$ define $\tilde{\boldsymbol{r}}_g = \tilde{\boldsymbol{r}}_g^0 + \tilde{\boldsymbol{r}}_g^1$. This is the value from which $\tilde{\boldsymbol{t}}_g$ is computed in Step 7 in Figure 3.4. In the simulation and Hybrid I, instead pick $\tilde{\boldsymbol{r}}_g$ uniformly at random and then recompute the values sent to $\breve{P}_r$ as above. In Hybrid II compute $\tilde{\boldsymbol{r}}_g$ as in the protocol (but still starting from the uniformly random $\bar{\boldsymbol{s}}_i^{1-b_i}$). Then simply observe that $\tilde{\boldsymbol{r}}_g$ has the same distribution in Hybrid I and Hybrid II. In Hybrid I it is uniformly random. In Hybrid II it is computed as

$$\tilde{\boldsymbol{r}}_g^0 + \tilde{\boldsymbol{r}}_g^1 = (\boldsymbol{r}_{\mathcal{T}+\gamma+g}^0 + \boldsymbol{r}_{\mathcal{T}+\gamma+g}^1) + \sum_{j=1}^{\gamma} x_j^g \boldsymbol{r}_{\mathcal{T}+j} \ ,$$

and it is easy to see that $\boldsymbol{r}_{\mathcal{T}+\gamma+g}^0 + \boldsymbol{r}_{\mathcal{T}+\gamma+g}^1$ is uniformly random and independent of all other values in the view of $\breve{P}_r$.

We now consider the case where the sender is corrupted who we denote $\breve{P}_s$. The simulator will run the code of P_s honestly, simulating also $\mathcal{F}_{\mathsf{ROT}}$ honestly. It will record the values (b_i, l_i^0, l_i^1) from **Init**. The remaining job of the simulator is then to extract the values $\tilde{\boldsymbol{r}}_j$ to send to $\mathcal{F}_{\mathsf{HCOM}}$ in the command $(\texttt{corrupt-commit}, \texttt{sid}, \{\tilde{\boldsymbol{r}}_j\}_{j \in \mathcal{J}})$. This should be done such that the probability that the receiver later outputs $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c \in C}, \boldsymbol{r})$

for $\boldsymbol{r} \neq \sum_{c \in C} \alpha_c \tilde{\boldsymbol{r}}_c$ is at most 2^{-s}. We first describe how to extract the values $\tilde{\boldsymbol{r}}_j$ and then show that the commitments are binding to these values.

We use the *Consistency Check* performed in the second half of Figure 3.3 to define a set $E \subseteq \{1, \ldots, n\}$. We call this the erasure set. This name will make sense later, but for now think of E as the set of indices for which the corrupted sender $\breve{P}_s$ after the consistency checks knows the choice bits b_i for $i \in E$ and for which the bits b_i for $i \notin E$ are still uniform in the view of $\breve{P}_s$.

We first describe how to compute the erasure set for a single of the $2s$ consistency checks and then discuss how to compute the joint set. We therefore omit the consistency check index g in the following and use the first of the $2s$ extra commitments as the blinding value.

Define the column vectors $\boldsymbol{s}_j^0$ and $\boldsymbol{s}_j^1$ as in the protocol. This is possible as the seeds from $\mathcal{F}_{\mathsf{ROT}}$ are well defined. Following the protocol, and adding a few more definitions, define

$$\begin{gathered}
\boldsymbol{r}_j^0 = \pi_k(\boldsymbol{s}_j^0)\ ,\ \boldsymbol{r}_j^1 = \pi_k(\boldsymbol{s}_j^1)\ ,\ \boldsymbol{r}_j = \boldsymbol{r}_j^0 + \boldsymbol{r}_j^1\ ,\ \boldsymbol{u}_j^0 = \pi_{k+1,n}(\boldsymbol{s}_j^0)\ , \\
\boldsymbol{u}_j^1 = \pi_{k+1,n}(\boldsymbol{s}_j^1)\ , \boldsymbol{u}_j = \boldsymbol{u}_j^0 + \boldsymbol{u}_j^1\ ,\ \boldsymbol{t}_j = \mathcal{C}(\boldsymbol{r}_j)\ ,\ \boldsymbol{c}_j = \pi_{k+1,n}(\boldsymbol{t}_j)\ , \\
\boldsymbol{c}_j^0 = \boldsymbol{u}_j^0\ ,\ \boldsymbol{c}_j^1 = \boldsymbol{c}_j - \boldsymbol{c}_j^0\ , \boldsymbol{d}_j^0 = \boldsymbol{u}_j^0\ ,\ \boldsymbol{d}_j^1 = \boldsymbol{u}_j^1 + \bar{\boldsymbol{c}}_j\ ,\ \boldsymbol{d}_j = \boldsymbol{d}_j^0 + \boldsymbol{d}_j^1 = \boldsymbol{u}_j + \bar{\boldsymbol{c}}_j\ , \\
\boldsymbol{w}_j^0 = \boldsymbol{r}_j^0 \| \boldsymbol{d}_j^0\ ,\ \boldsymbol{w}_j^1 = \boldsymbol{r}_j^1 \| \boldsymbol{d}_j^1\ .
\end{gathered}$$

Notice that if P_s is honest, then

$$\bar{\boldsymbol{c}}_j = \boldsymbol{c}_j - \boldsymbol{u}_j$$

and therefore

$$\boldsymbol{d}_j = \boldsymbol{d}_j^0 + \boldsymbol{d}_j^1 = \boldsymbol{u}_j^0 + \boldsymbol{u}_j^1 + \bar{\boldsymbol{c}}_j = \boldsymbol{c}_j\ .$$

Hence $\boldsymbol{d}_j^0$ and $\boldsymbol{d}_j^1$ are the two shares of the non-systematic part $\boldsymbol{c}_j$ the same way that $\boldsymbol{r}_j^0$ and $\boldsymbol{r}_j^1$ are the two shares of the systematic part $\boldsymbol{r}_j$. If the sender was honest we would in particular have that

$$\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1 = \boldsymbol{r}_j \| \boldsymbol{d}_j = \boldsymbol{r}_j \| \boldsymbol{c}_j = \mathcal{C}(\boldsymbol{r}_j)\ ,$$

i.e., $\boldsymbol{w}_j^0$ and $\boldsymbol{w}_j^1$ would be the two shares of the whole codeword.

We can define the values that an honest P_s *should* send as

$$\tilde{\boldsymbol{r}}^0 = \boldsymbol{r}^0_{\mathcal{T}+\gamma+1} + \sum_j x_j \boldsymbol{r}_j^0\ ,\ \tilde{\boldsymbol{r}}^1 = \boldsymbol{r}^1_{\mathcal{T}+\gamma+1} + \sum_j x_j \boldsymbol{r}_j^1\ ,\ \tilde{\boldsymbol{c}}^0 = \boldsymbol{c}^0_{\mathcal{T}+\gamma+1} + \sum_j x_j \boldsymbol{c}_j^0\ .$$

These values can be used to define values

$$\begin{gathered}
\tilde{\boldsymbol{r}} = \tilde{\boldsymbol{r}}^0 + \tilde{\boldsymbol{r}}^1\ ,\ \tilde{\boldsymbol{t}} = \mathcal{C}(\tilde{\boldsymbol{r}})\ ,\ \tilde{\boldsymbol{c}} = \pi_{k+1,n}(\tilde{\boldsymbol{t}})\ , \\
\tilde{\boldsymbol{c}}^1 = \tilde{\boldsymbol{c}} - \tilde{\boldsymbol{c}}^0\ ,\ \tilde{\boldsymbol{w}}^0 = \tilde{\boldsymbol{r}}^0 \| \tilde{\boldsymbol{c}}^0\ ,\ \tilde{\boldsymbol{w}}^1 = \tilde{\boldsymbol{r}}^1 \| \tilde{\boldsymbol{c}}^1\ .
\end{gathered}$$

We use $(\breve{\boldsymbol{r}}^0, \breve{\boldsymbol{r}}^1, \breve{\boldsymbol{c}}^0)$ to denote the values actually sent by $\breve{P}_s$ and we let the following denote the values computed by P_r (plus some extra definitions).

$$\breve{\boldsymbol{r}} = \breve{\boldsymbol{r}}^0 + \breve{\boldsymbol{r}}^1 \ , \ \breve{\boldsymbol{t}} = \mathcal{C}(\breve{\boldsymbol{r}}) \ , \ \breve{\boldsymbol{c}} = \pi_{k+1,n}(\breve{\boldsymbol{t}}) \ ,$$
$$\breve{\boldsymbol{c}}^1 = \breve{\boldsymbol{c}} - \breve{\boldsymbol{c}}^0 \ , \ \breve{\boldsymbol{w}}^0 = \breve{\boldsymbol{r}}^0 \| \breve{\boldsymbol{c}}^0 \ , \ \breve{\boldsymbol{w}}^1 = \breve{\boldsymbol{r}}^1 \| \breve{\boldsymbol{c}}^1 \ , \ \breve{\boldsymbol{w}} = \breve{\boldsymbol{w}}^0 + \breve{\boldsymbol{w}}^1 \ .$$

The simulator computes

$$\tilde{\boldsymbol{w}} = \boldsymbol{w}_{\mathcal{T}+\gamma+1} + \sum_j x_j \boldsymbol{w}_{\mathcal{T}+j} \tag{3.4}$$

as P_r in the protocol. For later use, define $\tilde{\boldsymbol{w}}^0 = \boldsymbol{w}^0_{\mathcal{T}+\gamma+1} + \sum_j x_j \boldsymbol{w}^0_{\mathcal{T}+j}$ and $\tilde{\boldsymbol{w}}^1 = \boldsymbol{w}^1_{\mathcal{T}+\gamma+1} + \sum_j x_j \boldsymbol{w}^1_{\mathcal{T}+j}$.

The check performed by P_r is then simply to check for $u = 1, \ldots, n$ that

$$\breve{\boldsymbol{w}}^{b_u}[u] = \tilde{\boldsymbol{w}}[u] \ . \tag{3.5}$$

Notice that in the protocol we have that

$$\boldsymbol{w}_j = \boldsymbol{b} * (\boldsymbol{w}_j^1 - \boldsymbol{w}_j^0) + \boldsymbol{w}_j^0 \ ,$$

where $*$ denotes the Schur product also known as the positionwise product of vectors. To see this notice that $(\boldsymbol{b} * (\boldsymbol{w}_j^1 - \boldsymbol{w}_j^0) + \boldsymbol{w}_j^0)[i] = b_i(\boldsymbol{w}_j^1[i] - \boldsymbol{w}_j^0[i]) + \boldsymbol{w}_j^0[i] = \boldsymbol{w}_j^{b_i}[i]$. In other words,

$$\boldsymbol{w}_j[i] = \boldsymbol{w}_j^{b_i}[i] \ .$$

It then follows from (3.4) that

$$\tilde{\boldsymbol{w}} = \boldsymbol{b} * (\tilde{\boldsymbol{w}}^1 - \tilde{\boldsymbol{w}}^0) + \tilde{\boldsymbol{w}}^0 \ ,$$

from which it follows that

$$\tilde{\boldsymbol{w}}[u] = \tilde{\boldsymbol{w}}^{b_u}[u] \ .$$

From (3.5) it then follows that $\breve{P}_s$ passes the consistency check if and only if for $u = 1, \ldots, n$ it holds that

$$\breve{\boldsymbol{w}}^{b_u}[u] = \tilde{\boldsymbol{w}}^{b_u}[u] \ . \tag{3.6}$$

We make some definitions related to the check in (3.6). We say that a position $u \in [n]$ is *silly* if $\breve{\boldsymbol{w}}^0[u] \neq \tilde{\boldsymbol{w}}^0[u]$ and $\breve{\boldsymbol{w}}^1[u] \neq \tilde{\boldsymbol{w}}^1[u]$. We say that a position $u \in [n]$ is *clean* if $\breve{\boldsymbol{w}}^0[u] = \tilde{\boldsymbol{w}}^0[u]$ and $\breve{\boldsymbol{w}}^1[u] = \tilde{\boldsymbol{w}}^1[u]$. We say that a position $u \in [n]$ is *probing* if it is not silly or clean. Let E denote the set of probing positions u. Notice that if there is a silly position u, then $\breve{\boldsymbol{w}}^{b_u}[u] \neq \tilde{\boldsymbol{w}}^{b_u}[u]$ so $\breve{P}_s$ gets caught. We can therefore assume without loss of generality that there are no silly positions. For the probing positions $u \in E$, there is by definition a bit c_u such that $\breve{\boldsymbol{w}}^{1-c_u}[u] \neq \tilde{\boldsymbol{w}}^{1-c_u}[u]$ and such that $\breve{\boldsymbol{w}}^{c_u}[u] = \tilde{\boldsymbol{w}}^{c_u}[u]$. This means that $\breve{P}_s$ passes the test only if $c_u = b_u$ for all $u \in E$. Since $\breve{P}_s$ knows c_u

it follows that if $\breve{P}_s$ does not get caught, then it can guess b_u for $u \in E$ with probability 1.

We compute a set E like above for each of the $2s$ checks and let E be the union of these. Clearly $\breve{P}_s$ passes the $2s$ tests only if it can guess b_u for $u \in E$ with probability 1.

Before we proceed to describe the extractor, we are now going to show two facts about E. First we will show that $|E| < s$, except with probability 2^{-s}. This follows from the simple observation that each b_u for $u \in E$ is uniformly random and $\breve{P}_s$ passes the consistency test if and only if $c_u = b_u$ for $u \in E$ and the only information that $\breve{P}_s$ has on the bits b_u is via the probing positions. Hence $\breve{P}_s$ passes the consistency test with probability at most $2^{-|E|}$. We can therefore assume for the rest of the proof that $|E| < s$.

Second, let $\mathcal{C}_{-E}$ be the code obtained from $\mathcal{C}$ by puncturing at the positions $u \in E$, *i.e.*, a codeword of $\mathcal{C}_{-E}$ can be computed as $\boldsymbol{t} = \mathcal{C}(\boldsymbol{r})$ and then outputting $\boldsymbol{t}_{-E}$, *i.e.*, the vector $\boldsymbol{t}$ where we remove the positions $u \in E$. We show that for all $j = \mathcal{T}+1, \ldots, \mathcal{T}+\gamma$ it holds that

$$(\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{-E} \in \mathcal{C}_{-E}(\mathbb{F}^k) \ ,$$

except with probability 2^{-s}. This is equivalent to proving that

$$E \in \Delta_{\mathcal{C}}((\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{j=\mathcal{T}+1}^{\mathcal{T}+\gamma}) \ .$$

Since the vectors $(\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{j=\mathcal{T}+1}^{\mathcal{T}+\gamma}$ are fixed at the point where the receiver samples the linear combinations we can use Lemma 1 to conclude that except with probability 2^{-s} either $|E| \geq s$ or

$$\Delta_{\mathcal{C}}((\tilde{\boldsymbol{w}}_g^0 + \tilde{\boldsymbol{w}}_g^1)_{g=1}^{2s}) \subseteq \Delta_{\mathcal{C}}((\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{j=\mathcal{T}+1}^{\mathcal{T}+\gamma}) \ , \tag{3.7}$$

where $\tilde{\boldsymbol{w}}_g^0$ and $\tilde{\boldsymbol{w}}_g^1$ are the values of $\tilde{\boldsymbol{w}}^0$ and $\tilde{\boldsymbol{w}}^1$ in test number g. Since we have assumed that $|E| < s$, we can assume that (3.7) holds except with probability 2^{-s} (as $d \geq s$). It is therefore sufficient to prove that

$$E \in \Delta_{\mathcal{C}}((\tilde{\boldsymbol{w}}_g^0 + \tilde{\boldsymbol{w}}_g^1)_{g=1}^{2s}) \ .$$

To see that $E \in \Delta_{\mathcal{C}}((\tilde{\boldsymbol{w}}_g^0 + \tilde{\boldsymbol{w}}_g^1)_{g=1}^{2s})$, observe that by construction we have that $(\breve{\boldsymbol{w}}^0 + \breve{\boldsymbol{w}}^1)_{-E} \in \mathcal{C}_{-E}(\mathbb{F}^k)$, so if a $(\tilde{\boldsymbol{w}}_g^0 + \tilde{\boldsymbol{w}}_g^1)_{-E} \notin \mathcal{C}_{-E}(\mathbb{F}^k)$ we either have that $(\tilde{\boldsymbol{w}}_g^0)_{-E} \neq (\breve{\boldsymbol{w}}^0)_{-E}$ or $(\tilde{\boldsymbol{w}}_g^1)_{-E} \neq (\breve{\boldsymbol{w}}^1)_{-E}$. Since there are no silly positions, this implies that we have a new probing position $u \notin E$, a contradiction to the definition of E.

We can now assume without loss of generality that $|E| < s$ and that $(\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{-E} \in \mathcal{C}_{-E}(\mathbb{F}^k)$. From $|E| < s$ and $\mathcal{C}$ having minimal distance $d \geq s$ we have that $\mathcal{C}_{-E}$ has minimal distance ≥ 1. Hence we can from each j and each $(\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{-E} \in \mathcal{C}_{-E}(\mathbb{F}^k)$ compute $\tilde{\boldsymbol{r}}_j \in \mathbb{F}^k$ such that

$$(\boldsymbol{w}_j^0 + \boldsymbol{w}_j^1)_{-E} = \mathcal{C}_{-E}(\tilde{\boldsymbol{r}}_j) \ .$$

These are the values that $\mathcal{S}$ will send to $\mathcal{F}_{\mathsf{HCOM}}$.

We then proceed to show that for all $\{(c, \alpha_c)\}_{c\in C}$ the environment can open to $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C}, \tilde{\boldsymbol{r}})$ for $\tilde{\boldsymbol{r}} = \sum_{c\in C} \alpha_c \tilde{\boldsymbol{r}}_c$ with probability 1. The reason for this is that if $\breve{P}_s$ computes the values in the opening correctly, then clearly $(\breve{\boldsymbol{w}}^0)_{-E} = (\tilde{\boldsymbol{w}}^0)_{-E}$ and $(\breve{\boldsymbol{w}}^1)_{-E} = (\tilde{\boldsymbol{w}}^1)_{-E}$. Furthermore, for the positions $u \in E$ it can open to any value as it knows b_u. It therefore follows that if $\breve{P}_s$ can open to $(\texttt{opened}, \texttt{sid}, \{(c, \alpha_c)\}_{c\in C}, \boldsymbol{r})$ for $\boldsymbol{r} \neq \sum_{c\in C} \alpha_c \tilde{\boldsymbol{r}}_c$, then it can open $\{(c, \alpha_c)\}_{c\in C}$ to two different values. Since the code has distance $d \geq s$, it is easy to see that after opening some $\{(c, \alpha_c)\}_{c\in C}$ to two different values, the environment can compute with probability 1 at least s of the choice bits b_u, which it can do with probability at most 2^{-s}, which is negligible. □

3.4 Comparison with Recent Schemes

In this section we compare the efficiency of our scheme to the most efficient schemes in the literature realizing UC-secure commitments with security against a static and malicious adversary. In particular, we compare our construction to the schemes of [Lin11], [BCPV13], [CJS14] and [CDD+15]. We omit the scheme of [CDD+16] in the following as in terms of communication it is equivalent to ours and our concrete comparison does not reflect the asymptotic differences in computation time.

The scheme of [BCPV13] (Fig. 6) is a slightly optimized version of [Lin11] (Protocol 2) which implement a multi-commitment ideal functionality. Along with [CJS14] these schemes support commitments between multiple parties natively, a property not shared with the rest of the protocols in this comparison. We therefore only consider the two party case where a sender commits to a receiver. The schemes of [Lin11, BCPV13] are in the CRS-model and their security relies on the DDH assumption. As the messages to be committed to are encoded as group elements the message size and the level of security are coupled in these schemes. For large messages this is not a big issue as the group size would just increase as well, or one can break the message into smaller blocks and commit to each block. However, for shorter messages, it is not possible to decrease the group size, as this would weaken security. The authors propose instantiating their scheme over an elliptic curve group over a field size of 256-bits so later in our comparison we also consider committing to values of this length. This is optimal for these schemes as the overhead of working with group elements of 256-bits would become more apparent if committing to smaller values.

The scheme of [CJS14] in the global random oracle model can be based on any stand-alone secure trapdoor commitment scheme, but for concreteness we compare the scheme instantiated with the commitment scheme of [Ped92] as also proposed by the authors. As [Ped92] is also based on the DDH assumption we use the same setting and parameters for [CJS14] as for the former two

Scheme	Hom.	OTs		Communication		Rounds		Computation			
		$\binom{2}{1}$	$\binom{3}{2}$	Commit	Open	Commit	Open	Commit		Open	
								Exp.	Enc.	Exp.	Enc.
[Lin11]	✗	0	0	$4g$	$6g + 4l + k$	1	5	5	0	$18^1/_3$	0
[BCPV13]	✗	0	0	$4g$	$5g + 3l + k$	1	3	10	0	12	0
[CJS14]	✗	0	0	$4g + 2l + h$	$3l + 2h + 3\kappa + k$	2	3	5	0	5	0
[CDD+15], basic	✗	$\overline{n}$	0	$2\overline{n}f$	$(k + \overline{n} + 1)f$	1	1	0	1	0	1
[CDD+15], homo	✓	0	$\overline{n}$	$^{6(k+2\overline{n})\overline{n}f}/_k$	$(k + 2\overline{n} + 1)f$	1	1	0	$8\overline{n}/k + 2$	0	1
This Work	✓	n	0	$^{(2s\cdot 2nf+\kappa)}/_\gamma + nf$	$(k + n + 1)f$	3	1	0	$^{2s\cdot 2}/_\gamma + 1$	0	1

Table 3.1: Comparison of the most efficient UC-secure schemes for committing to γ messages of k components. Sizes are in bits. Legend: g is size of a group element, l is size of a scalar in the exponent, h is the output length of the random oracle, f is the size of a finite field element, Exp. denotes the number of modular exponentiations, Enc. denotes the number of encoding procedures of the corresponding codes which have length $\overline{n}$ and n. The schemes of [CDD+15] are presented with the sharing parameter t set to 2 for the basic and 3 for the homomorphic.

schemes.

We present our detailed comparison in Table 3.1. The table shows the costs of all the previously mentioned schemes in terms of OTs required, communication, number of rounds and computation. For the schemes of [CDD+15] we have fixed the sharing parameter t to 2 and 3 for the basic and homomorphic version, respectively. To the best of our knowledge this is also the optimal choice in all settings. Also for the scheme of [CJS14] we do not list the queries to the random oracle in the table, but remark that their scheme requires 6 queries per commitment. For our scheme, instead of counting the cost of sending the challenges $(x_1^g, x_2^g, \ldots, x_\gamma^g) \in \mathbb{F}$ for $g \in [2s]$, we assume the receiver sends a random seed of size κ instead. This is then used as input to a PRG whose output is used to determine the challenges.

To give a flavor of the actual numbers we compute Table 3.1 for specific parameters in Table 3.2. We fix the field to $\mathbb{F}_2$ and look at computational security $\kappa = 128$, statistical security $s = 40$ and instantiate the random oracle required by [CJS14] with SHA-256. As the schemes of [Lin11, BCPV13, CJS14] rely on the hardness of the DDH assumption, a 256-bit EC group is assumed sufficient for 128-bit security [SRG+14]. As already mentioned we look at message length $k = 256$ as this is well suited for these schemes.[2] The best code we could find for the schemes of [CDD+15] in this setting has parameters $[631, 256, 81]$ and is a shortened BCH code. For our scheme, the best code we have identified for the above parameters is a $[419, 256, 40]$ expurgated BCH code [SS06]. Also, we recall the experiments performed in [CDD+15] showing that exponentiations in a EC-DDH group of the above size require roughly

[2] We here assume a perfect efficient encoding of 256-bit values to group elements of a 256-bit EC group.

Scheme	Homo	OTs		Communication		Rounds		Computation			
		$\binom{2}{1}$	$\binom{3}{2}$	Commit	Open	Commit	Open	Commit		Open	
								Exp.	Enc.	Exp.	Enc.
[Lin11]	✗	0	0	1,024	2,816	1	5	5	0	$18^{1}/_{3}$	0
[BCPV13]	✗	0	0	1,024	2,304	1	3	10	0	12	0
[CJS14]	✗	0	0	1,792	1,920	2	3	5	0	5	0
[CDD+15], basic, $\gamma = 319$	✗	631	0	4,301	888	1	1	22	1	0	1
[CDD+15], homo, $\gamma = 319$	✓	0	631	35,615	1,519	1	1	88	22	0	1
This Work, $\gamma = 319$	✓	419	0	2,648	676	3	1	15	1.5	0	1
[CDD+15], basic, $\gamma = 1,000$	✗	631	0	2,232	888	1	1	7	1	0	1
[CDD+15], homo, $\gamma = 1,000$	✓	0	631	26,649	1,519	1	1	28	22	0	1
This Work, $\gamma = 1,000$	✓	419	0	1,130	676	3	1	5	1	0	1
[CDD+15], basic, $\gamma = 10,000$	✗	631	0	1,359	888	1	1	0	1	0	1
[CDD+15], homo, $\gamma = 10,000$	✓	0	631	22,869	1,519	1	1	3	22	0	1
This Work, $\gamma = 10,000$	✓	419	0	491	676	3	1	0	1	0	1
[CDD+15], basic, $\gamma = 100,000$	✗	631	0	1,272	888	1	1	0	1	0	1
[CDD+15], homo, $\gamma = 100,000$	✓	0	631	22,491	1,519	1	1	0	22	0	1
This Work, $\gamma = 100,000$	✓	419	0	427	676	3	1	0	1	0	1

Table 3.2: Concrete efficiency comparison of the most efficient UC-secure schemes for committing to messages of size $k = 256$, $\kappa = 128$, $h = 256$ and $s = 40$ where the field is $\mathbb{F}_2$. In the table γ represents the number of commitments the parties perform. These numbers include the cost of performing the initial OTs, both in terms of communication and computation.

500 times more computation time compared to encoding using a BCH code for parameters of the above type.[3] In their brief comparison with [HMQ04], another commitment scheme in the random oracle model, the experiments showed that one of the above BCH encodings is roughly 1.6 times faster than 4 SHA-256 invocations, which is the number of random oracle queries required by [HMQ04]. This therefore suggests that one BCH encoding is also faster than the 6 random oracle queries required by [CJS14] if indeed instantiated with SHA-256.

To give as meaningful comparisons as possible we also instantiate the initial OTs and include the cost of these in Table 3.2. As the homomorphic version of [CDD+15] require 2-out-of-3 OTs in the setup phase, using techniques described in [LOP11, LP11], we have calculated that these require communicating 26 group elements and 44 exponentiations per invocation. The standard 1-out-of-2 OTs we instantiate with [PVW08] which require communicating 6 group elements and computing 11 exponentiations per invocation.

In Table 3.2 we do not take into consideration OT extension techniques [Bea96, IKNP03, Nie07, NNOB12, Lar15, ALSZ15, KOS15], as we do so few OTs that even the most efficient of these schemes *might* not improve the efficiency in practice. We note however that if in a setting where OT extension is already used, this would have a very positive impact on our scheme as the OTs in the setup phase would be much less costly. On a technical note some

[3] They run the experiments with a shortened BCH code with parameters $[796, 256, 121]$, which therefore suggests their observations are also valid for our choice of parameters.

of the ideas used in this work are very related to the OT extension techniques introduced in [IKNP03] (and used in all follow-up work that make black-box use of a PRG). However an important and interesting difference is that in our work we do not "swap" the roles of the sender and receiver for the initial OTs as otherwise the case for current OT extension protocols. This observation means that the related work of [GIKW14], which makes use of OT extension, would look inherently different from our protocol, if instantiated with one of the OT extension protocols that follow the [IKNP03] blueprint.

As can be seen in Table 3.2, our scheme improves as the number of committed values γ grows. In particular we see that at around 319 commitments, for the above message sizes and security parameters, our scheme outperforms all previous schemes in total communication, while at the same time offering additive homomorphism.

3.5 Protocol Extension

As the scheme presented in Section 3.2 only implements commitments to random values we here describe an efficient extension to chosen message commitments. Our extension Π_{EHCOM} is phrased in the $\mathcal{F}_{\mathsf{HCOM}}$-hybrid model and it is presented in Figure 3.5. The techniques presented therein are folklore and are known to work for any UC-secure commitment scheme, but we include them as a protocol extension for completeness. The **Chosen-Commit** step shows how one can turn a commitment of a random value into a commitment of a chosen value. This is done by simply using the committed random value as a one-time pad on the chosen value and sending this to P_r. The **Extended-Open** step describes how to open to linear combinations of either random commitments, chosen commitments or both. It works by using $\mathcal{F}_{\mathsf{HCOM}}$ to open to the random commitments and the commitments used to one-time pad the chosen commitments. Together with the previously sent one-time pad the receiver can then learn the designated linear combination.

Finally we present a **Batch-Open** step that achieves very close to optimal amortized communication complexity for opening to a set of messages. The technique is similar to the consistency check of Π_{HCOM}. When required to open to a set of messages, the sender P_s will start by sending the messages directly to the receiver P_r. Next, the receiver challenges the sender to open to $\hat{s} = {}^{s}/_{\log_2(|\mathbb{F}|)}$ random linear combinations of all the received messages. Notice that unlike the initial commit step where $2s$ linear combinations are required, only $\hat{s} \leq s$ combinations are required for batch opening. See Section 3.2 for an explanation why $\hat{s}$ suffices for **Batch-Open**. When receiving the opening from $\mathcal{F}_{\mathsf{HCOM}}$, P_r verifies that it is consistent with the previously received messages and if this is the case it accepts these. For the exact same reasons as covered in the proof of Theorem 1 it follows that this approach of opening values is secure. For clarity and ease of presentation the description of batch-opening

Π_{EHCOM} describes a protocol between a sender P_s and a receiver P_r.

Chosen-Commit:

1. On input $(\texttt{chosen-commit}, \texttt{sid}, \texttt{cid}, \boldsymbol{m})$, P_s picks an already committed to value $\boldsymbol{r}_j$ and computes $\widetilde{\boldsymbol{m}} = \boldsymbol{m} - \boldsymbol{r}_j$. It then sends $(\texttt{chosen}, \texttt{sid}, \texttt{cid}, j, \widetilde{\boldsymbol{m}})$ to P_r. Else it ignores the message.
2. P_r stores $(\texttt{chosen}, \texttt{sid}, \texttt{cid}, j, \widetilde{\boldsymbol{m}})$ and outputs $(\texttt{chosen-receipt}, \texttt{sid}, \texttt{cid})$.

Extended-Open:

1. On input $(\texttt{extended-open}, \texttt{sid}, \{(j, \alpha_j)\}_{j \in C_r}, \{(l, \beta_l)\}_{l \in C_c})$ with $\beta_l \in \mathbb{F}$ for $l \in C_c$ and $\alpha_j \in \mathbb{F}$, for $j \in C_r$, P_s verifies that and it has previously committed to a value $\boldsymbol{r}_j$ using $\mathcal{F}_{\mathsf{HCOM}}$ for $j \in C_r$. Else it ignores the message. For all $l \in C_c$ P_s verifies that it previously sent the message $(\texttt{chosen}, \texttt{sid}, l, \overline{j}, \widetilde{\boldsymbol{m}}_l)$ to P_r. Let $\overline{\mathcal{J}}$ be the set of the corresponding indices $\overline{j}$, similarly let $\overline{\beta_{\overline{j}}} = \beta_l$ for the corresponding ID l. P_s then sends $(\texttt{open}, \texttt{sid}, \{(j, \alpha_j)\}_{j \in C_r} \cup \{(\overline{j}, \overline{\beta_{\overline{j}}})\}_{\overline{j} \in \overline{\mathcal{J}}})$ to $\mathcal{F}_{\mathsf{HCOM}}$.
2. Upon receiving $(\texttt{open}, \texttt{sid}, \{(j, \alpha_j)\}_{j \in C_r} \cup \{(\overline{j}, \overline{\beta_{\overline{j}}})\}_{\overline{j} \in \overline{\mathcal{J}}}, \boldsymbol{r})$ from $\mathcal{F}_{\mathsf{HCOM}}$, P_r identifies the previously received messages $(\texttt{chosen}, \texttt{sid}, l, \overline{j}, \widetilde{\boldsymbol{m}}_l)$ and outputs $(\texttt{extended-opened}, \texttt{sid}, \{(j, \alpha_j)\}_{j \in C_r}, \{(l, \beta_l)\}_{l \in C_c}, \boldsymbol{r} + \sum_{l \in C_c} \beta_l \cdot \widetilde{\boldsymbol{m}}_l)$.

Batch-Open:

1. On input $(\texttt{batch-open}, \texttt{sid}, C_r, C_c)$. For all $j \in C_r$ P_s verifies that it has previously committed to a value $\boldsymbol{r}_j$ using $\mathcal{F}_{\mathsf{HCOM}}$. Else it ignores the message. For all $l \in C_c$ P_s verifies that it previously sent the message $(\texttt{chosen}, \texttt{sid}, l, \overline{j}, \widetilde{\boldsymbol{m}}_l)$ to P_r. P_s then sends $(\texttt{batch-open}, \texttt{sid}, \{(j, \boldsymbol{r}_j)\}_{j \in C_r}, \{(l, \boldsymbol{m}_l)\}_{l \in C_c})$ to P_r, where $\boldsymbol{r}_j$ and $\boldsymbol{m}_l$ are random and chosen messages, respectively, previously committed to.
2. Let $t_r = |C_r|, t_c = |C_c|$ and $\hat{s} = s/\log_2(|\mathbb{F}|)$. For $g \in [\hat{s}]$ P_r then samples random values $x_1^g, \ldots, x_{t_r}^g, y_1^g, \ldots, y_{t_c}^g \in_R \mathbb{F}$ and sends these to P_s.
3. Then for $g \in [\hat{s}]$ P_s and P_r run **Extended-Open** with input

$$(\texttt{extended-open}, \texttt{sid}, \{(j_u, x_u^g)\}_{u \in [t_r]}, \{(l_v, y_v^g)\}_{v \in [t_c]})$$

   where j_u and l_v are the u'th and v'th element of C_r and C_c respectively, under an arbitrary ordering.
4. P_r lets $(\texttt{extended-opened}, \texttt{sid}, \{(j_u, x_u^g)\}_{u \in [t_r]}, \{(l_v, y_v^g)\}_{v \in [t_c]}, \boldsymbol{n}_g)$ be the output of running **Extended-Open**. Finally for $g \in [\hat{s}]$ P_r now verifies that

$$\boldsymbol{n}_g = \sum_{u \in [t_r]} x_u^g \cdot \boldsymbol{r}_{j_u} + \sum_{v \in [t_c]} y_v^g \cdot \boldsymbol{m}_{l_v} \ .$$

   If true then P_r outputs $(\texttt{batch-opened}, \texttt{sid}, \{(j, \boldsymbol{r}_j)\}_{j \in C_r} \cup \{(l, \boldsymbol{m}_l)\}_{l \in C_c})$. Else it aborts and halts.

Figure 3.5: Protocol Π_{EHCOM} in the $\mathcal{F}_{\mathsf{HCOM}}$-hybrid model.

does not take into account opening to linear combinations of random and chosen commitments. However the procedure can easily be extended to this setting using the same approach as in **Extended-Open**.

In terms of efficiency, to open N commitments with message-size l, the sender needs to send lN field elements along with the verification overhead $\hat{s}\hat{O} + \kappa$ where $\hat{O}$ is the cost of opening to a commitment using $\mathcal{F}_{\mathsf{HCOM}}$. Therefore if the functionality is instantiated with the scheme Π_{HCOM}, the total communication for batch-opening is $\hat{s}(k+n)f + \kappa + kNf$ bits where k is the length of the message, n is the length of the code used, f is the size of a field element.

We now elaborate on the applicability of batch-opening for committing to large messages as mentioned in Section 3.2. Recall that there we split the large message m of size M into N blocks of size l and the idea is to instantiate Π_{HCOM} with a $[n_s, l, s]$ code and commit to m in blocks of size l. This requires n_s initial OTs to setup and requires sending $\hat{s} \cdot 2n_s f + \kappa + n_s N f$ bits to commit to all blocks. For a fixed $\hat{s}$ this has rate close to 1 for large enough l. In the opening phase we can then use the above batch-opening technique to open to all the blocks of the original message, and thus achieve a rate of $Mf/(\hat{s}(l+n_s)f+\kappa+lNf) \approx 1$ in the opening phase as well.

In [GIKW14] the authors present an example of committing to strings of length 2^{30} with statistical security $s = 30$ achieving rate $1.046^{-1} \approx 0.95$ in both the commit and open phase. To achieve these numbers the field size is required to be very large as well. The authors propose techniques to reduce the field size, however at the cost of reducing the rate. We will instantiate the approach described above using a binary BCH code over the field $\mathbb{F}_2$ and recall that these have parameters $[n-1, n-\lceil d-1/2 \rceil \log(n+1), \geq d]$. Using a block length of 2^{13} and $s = 30$ therefore gives us a code with parameters $[8191, 7996, 30]$. Thus we split the message into $134,285 = \lceil 2^{30}/7996 \rceil$ blocks. In the commitment phase we therefore achieve rate $2^{30}/(30 \cdot 2 \cdot 8191 + 128 + 8191 \cdot 134,285) \approx 0.976$. Using the batch-opening technique the rate in the opening phase is even higher than in the commit phase, as this does not require any "blinding" values. In the above calculations we do not take into account the 8191 initial OTs required to setup our scheme. However using the OT-extension techniques of [KOS15], each OT for κ-bit strings can be run using only κ initial "seed" OTs and each extended OT then requires only κ bits of communication. Instantiating the seed OTs with the protocol of [PVW08] for $\kappa = 128$ results in $6 \cdot 256 \cdot 128 + 8191 \cdot 128 = 1,245,056$ extra bits of communication which lowers the rate to 0.974.

Finally, based on local experiments with BCH codes with the above parameters, we observe that the running time of an encoding operation using the above larger parameters is roughly 2.5 times slower than an encoding using a BCH code with parameters $[796, 256, 121]$. This suggests that the above approach remains practical for implementations as well.

Chapter 4

Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO

The following chapter is based on the work of [NST17] and is therefore identical (except for minor layout modifications) to the current full version available at `https://eprint.iacr.org/2016/1069`.

4.1 Introduction

Secure two-party computation is the area of cryptology dealing with two mutually distrusting parties wishing to compute an arbitrary function f on private inputs. Say A has input x and B has input y. The guarantee offered from securely computing f is that the only thing learned from the computation is the output $z = f(x, y)$, in particular nothing is revealed about the other party's input that cannot be inferred from the output z. This seemingly simple guarantee turns out to be extremely powerful and several real-world applications and companies using secure computation have arisen in recent years [BCD+09, BLW08, Dya, Par, Sep]. The idea of secure computation was initially conceived in 1982 by Andrew Yao [Yao82, Yao86], particularly for the *semi-honest* setting, in which all parties are assumed to follow the protocol specification but can try to extract as much information as possible from the protocol execution. Yao gave an approach for preventing any such extraction using a technique referred to as the *garbled circuit technique*. At a very high level, using the abstraction of [BHR12b], A starts by garbling or "encrypting" the circuit f using the garbling algorithm $(F, e, d) = \mathsf{Gb}(f)$ obtaining a garbled circuit F, an input encoding function e and an output decoding function d. It

then encodes its input as $X = \mathsf{En}(x, e)$ and sends (F, X, d) to B. Then, using oblivious transfer (OT), A blindly transfers a garbled version Y of B's input which enables B to compute a garbled output $Z = \mathsf{Ev}(F, X\|Y)$ which it can then decode to obtain $z = \mathsf{De}(Z, d)$. Finally B returns z to A.

As already mentioned, the above sketched approach can be proven secure in the semi-honest setting [LP09]. In the stronger *malicious* setting however it completely breaks down as in this model the parties are allowed to deviate arbitrarily from the protocol specification. One of the most obvious attacks is for A to garble a different function $f' \neq f$ which could enable A to learn y from the resulting value z' without B even noticing this. To combat this type of attack the cut-and-choose technique can be applied: instead of garbling a single circuit, A garbles several copies and sends these to B. A random subset of them is now selected for checking by B and if everything is correct, some fraction of the remaining circuits must be correct with overwhelming probability. Leaving out many details these can now be evaluated to obtain a single final output. While this approach thwarts the above attack it unfortunately opens up for several new ones that also need to be dealt with, for instance ensuring *input consistency* for all the remaining evaluation garbled circuits. Another more subtle issue in the malicious setting is the *Selective-OT Attack* (also called Selective Failure Attack) as pointed out in [MF06, KS06]. A corrupt A can cheat when offering B the garbled inputs in the OT step by using a bogus value for either the 0 or the 1 input. This will either result in B aborting as it cannot evaluate the garbled circuit or it will go through undetected and B will return the output to A. Either way the input bit of B is revealed to A which is a direct breach of security. It is easy to see that using this attack A can learn any l bits of B's input with probability 2^{-l} if not properly dealt with.

Over the last decade several solutions to the above issues have been proposed, along with dramatic efficiency improvements for secure 2PC protocols based on the cut-and-choose approach of garbled circuits [LP07, PSSW09, LP11, sS11, HEKM11, KsS12, Bra13, FN13, HKE13, Lin13, MR13, sS13, HMsG13, FJN14, AMPR14, WMK17]. Finally we note for completeness that secure computation has also been studied in great detail for many other settings, including the more general multi-party case (MPC). Several different adversarial models such as honest majority [GMW87, BGW88, CCD88], dishonest majority [DPSZ12] and covert security [AL07] have also been proposed in the literature. In this work we focus solely on the special case of two parties with malicious security and in the next section we discuss the reported concrete efficiency of state-of-the-art protocols in this setting.

Related Work

In the less than 10 years since the first reported implementation of maliciously secure 2PC based on garbled circuits [LPS08], the performance advancements have been enormous [PSSW09, sS11, KsS12, sS13, FN13, FJN14, AMPR14,

LR15, RR16, WMK17]. Furthermore different settings and hardware configurations have been explored, notably using commodity grade GPUs in [FN13, FJN14] and large-scale CPU clusters [sS13] to parallelize the bulk of the computation. In the single-execution setting based solely on standard hardware the best reported performance time is that of [WMK17] which evaluates an AES-128 circuit in total time 65 ms. In addition, the works of [LR15] and [RR16] explore the more restricted setting of amortizing secure 2PC based on the cut-and-choose and the dual execution approach, respectively. By amortizing we mean that the protocols exploit constructing multiple secure evaluations of the same function f yielding impressive performance benefits over the more general single execution setting. Furthermore these protocols are in the offline/online setting where the bulk of the computation and communication can be done before the inputs are determined. We highlight that for both protocols, the offline computation depends on the function to be computed and we will refer to this as *dependent* preprocessing. However both protocols allow for the inputs to be chosen sequentially when securely evaluating f. This allows for a low latency online phase which is desirable for many applications. For securely computing 1024 AES-128 circuits, [LR15] reports 74 ms offline and 7 ms online per evaluation, while the more recent [RR16] reports 5.1 ms offline and 1.3 ms online for the same setting. Furthermore [RR16] achieves a 0.26 ms online phase when considering throughput alone, *i.e.* batched evaluation.

Another direction in secure computation is the secret sharing approach where the parties initially secret share their inputs and then interactively compute the function f in a secure manner. A particularly nice property of these protocols is that when considering the offline/online setting the offline phase can usually be done independently of the circuit f which we call *independent* preprocessing. This allows for naively utilizing parallelism in the preprocessing phase and also adds more flexibility as the offline material produced is universal. Another benefit is that in general this secret-sharing technique works for any number of parties and over any field, which depending on the desired functionality f can significantly increase performance. We note however that these protocols usually employ expensive public-key cryptography in the preprocessing phase and are therefore much slower than the offline phases of *e.g.* [LR15, RR16]. Finally the inherent interactiveness of the online phase, which has $\mathcal{O}(\mathsf{depth}(f))$ rounds of interaction, makes these protocols ill-suited for high latency networks such as WANs. There are many variations of the secret sharing approach but they typically enjoy the same overall pros and cons in terms of independent preprocessing and required interactivity. Examples of recent protocols following this paradigm are: TinyOT [NNOB12, LOS14, BLN+15], SPDZ [DPSZ12, DKL+13, KSS13, KOS16], MiniMac [DZ13, DLT14, DZ16], and TinyTables [DNNR17]. The fastest reported online time for computing an AES-128 circuit in this setting is 1.05 ms by [DNNR17] using dependent preprocessing. The work of [KSS13] reports 12 ms online time using independent preprocessing, but the evaluation exploits the algebraic structure

of AES-128. Furthermore [DNNR17] has an impressive throughput of 0.45 µs per AES-128 while [KSS13] and [DZ16] have throughput ∼1 ms and 0.4 ms, respectively.

In Table 4.1 we give an overview of the properties of the mentioned protocols and the reported timings for securely evaluating AES-128 on LAN in the offline/online setting. As the secret sharing-based, non-constant round, protocols are ill-suited for high latency networks we omit this from Table 4.1 since no AES-128 timings are published for these schemes in a WAN setting (however see Section 4.6 for a WAN comparison of the garbled circuit protocols). The timings reported for [DNNR17, LR15, RR16] and **This Work** are all measured on the same hardware (Amazon Web Services, c4.8xlarge instances on 10 Gbit LAN), while the timings for [WMK17] are on a less powerful instance (Amazon Web Services, c4.2xlarge instances on 2.5 Gbit LAN). Finally the results of [KSS13, DZ16] have been obtained on high-end Desktop machines with 1 Gbit LAN. The timings of [LR15, RR16] and **This Work** are all for 1024 AES-128 evaluations, while those of [WMK17] are for a single-execution. We believe the difference in performance between the offline/online (62 ms + 21 ms) and total latency (65 ms) settings for [WMK17] can be explained by the inability to interleave the sending/checking and evaluation of garbled circuits in the offline/online setting. In summary, as can be seen in the table our work is the first implementation of a protocol combining the advantages of independent (and dependent) preprocessing using only a constant number of rounds.

	Secret Sharing-based			Garbled Circuit-based			
Protocol	[KSS13]*	[DZ16]*	[DNNR17]*	[WMK17]	[LR15]	[RR16]	**This Work**
Constant round	✗	✗	✗	✓	✓	✓	✓
Independent preprocessing	N/A	✗	✗	✗	✗	✗	**13.84 ms**
Dependent preprocessing	✗	N/A	N/A	62 ms	74 ms	5.1 ms	**0.74 ms**
Best online latency	12 ms	6 ms	**1.05 ms**	21 ms	7 ms	1.3 ms	1.13 ms
Best online throughput	∼1 ms	0.4 ms	**0.45 µs**	N/A	N/A	0.26 ms	0.08 ms

*Uses dedicated techniques for evaluating AES.

Table 4.1: Overview of state-of-the-art protocols and the best reported timings for securely evaluating AES-128 with malicious security on a LAN in the offline/online setting. All timings are per AES-128. N/A stands for Not Available.

LEGO

The Large Efficient Garbled-circuit Optimization (LEGO) was first introduced by Nielsen and Orlandi in [NO09] which showed a new approach for maliciously secure 2PC based on cut-and-choose of garbled *gates*. This gave an asymptotic complexity improvement to $\mathcal{O}(s/\log(|f|))$ as opposed to $\mathcal{O}(s)$ for the standard *circuit* cut-and-choose approach for statistical security s. However the construction of [NO09] was heavily based on expensive public-key cryptography and was mainly considered an asymptotic advancement. This was later improved in the two follow-up works of MiniLEGO [FJN+13] and TinyLEGO [FJNT15] yielding incrementing asymptotic and concrete efficiency improvements. In a nutshell, the LEGO technique works by the generator A first garbling multiple individual AND gates (as opposed to garbling entire circuits) and sending these to the evaluator B. Then a cut-and-choose step on a random subset of these gates is carried out and finally the remaining unchecked gates are combined (or soldered) into a garbled fault tolerant circuit computing f. A crucial ingredient for securely soldering the garbled gates into circuits are XOR-homomorphic commitments which in [NO09] were realized using expensive Pedersen commitments [Ped92]. In the follow-up construction of [FJN+13] these were replaced by an asymptotically more efficient construction, however the concrete communication overhead of the proposed commitment scheme was inadequate for the protocol to be competitive for realistic circuit sizes and parameters. In the recent works of [FJNT16, CDD+16] this overhead has been improved to an optimal rate-1 and the resulting UC-secure XOR-homomorphic commitment scheme is both asymptotically and concretely very efficient. Finally the work of [HZ15] introduced a different primitive for LEGO soldering called XOR-Homomorphic Interactive Hash, which has some advantages over the commitment approach. However, the best instantiation of XOR-Homomorphic Interactive Hash still induces higher overall overhead than the commitment approach when using the schemes of [FJNT16, CDD+16].

Although the original LEGO protocol, and the above-mentioned follow-up works, asymptotically are very efficient, the overall consensus in the secure computation community has been that the reliance of XOR-homomorphic commitments for all circuit wires hinders actual practical efficiency. In this work we thoroughly investigate the practical efficiency of the LEGO approach and, in contrast to earlier beliefs, we demonstrate that it is indeed among the most practical protocols to date for general secure 2PC using garbled circuits.

Our Contributions

We implement the TinyLEGO protocol with added support for both independent and dependent preprocessing. Furthermore, our protocol supports fully reactive computation, meaning that when a function result has been obtained, another function depending on this result can be evaluated. Also,

Setting	Ind. Preprocessing*	Dep. Preprocessing	Online Latency	Online Throughput
Single Execution				
1 x AES-128	89.61 ms	13.23 ms	1.46 ms	1.46 ms
1 x SHA-256	478.54 ms	164.40 ms	11.19 ms	11.19 ms
Amortized				
128 x AES-128	14.85 ms	0.68 ms	1.15 ms	0.09 ms
128 x SHA-256	173.05 ms	12.13 ms	9.35 ms	1.09 ms

*Not including the time to compute the initial BaseOTs.

Table 4.2: Performance summary of our protocol on a high bandwidth (10Gbit) LAN network.

the independent preprocessing phase can be rerun at any time if additional garbling material is necessary. As part of our prototype we also implement the XOR-homomorphic commitment scheme of [FJNT16] and report on its efficiency separately as we believe our findings can be of independent interest. This is to our knowledge the first implementation of a protocol based directly on the LEGO paradigm and of the mentioned commitment scheme. The support for independent preprocessing is achieved from the fact that the bulk of the computation using the LEGO approach is based on cut-and-choose of independently garbled gates and hence only depends on the security parameter and the number of AND gates one wishes to preprocess. The subsequent soldering phase can then be seen as a dependent preprocessing phase where knowledge of the circuit f is required. This multi-level preprocessing is in contrast to previous non-LEGO protocols based on cut-and-choose of garbled circuits in the offline/online setting where the entire offline phase depends on the circuit to be evaluated. In more detail our main contributions are as follows:

1. We propose a new technique for dealing with the selective-OT attack on 2PC protocols based on garbled circuits. Our technique makes use of a globally correlated OT functionality ($\mathcal{F}_{\Delta\text{-ROT}}$) combined with XOR-homomorphic commitments and a Free-XOR garbling scheme [KS08]. Using the well-known fact of Beaver [Bea95] that OTs can be precomputed, we can mitigate the selective-OT attack by having the circuit constructor decommit to a single value per input bit of the evaluator in the online phase. This ensures that if the constructor tries to cheat, the evaluator aborts regardless of the value of his input. The technique is general and we believe that it can be used in other 2PC protocols based on garbled circuits as well. We also provide a more efficient instantiation

of $\mathcal{F}_{\Delta\text{-ROT}}$ than previously appearing in the literature by tightening the analysis of the construction presented in [BLN+15].

2. As part of our 2PC prototype we also implement the XOR-homomorphic commitment scheme of [FJNT16]. It is already known that this scheme is asymptotically very efficient, but this is to our knowledge the first time its practical efficiency has been thoroughly investigated. The result is a very efficient scheme achieving an amortized cost of less than a microsecond for both committing and decommitting to a random value. To maximize performance we utilize cache efficient matrix-transposition and inspired by the construction of [CDD+16] we use the Intel Streaming SIMD Extension (SSE) instruction PCLMULQDQ to efficiently compute the required linear combinations.
3. We build our LEGO prototype on top of the above-mentioned implementation which results in a very efficient and flexible protocol for maliciously secure reactive 2PC. As our online phase consists of an optimal two rounds, we can securely evaluate an AES-128 with latency down to 1.13 ms. When considering throughput we can do each AES-128 block in amortized online time 0.08 ms (considering 1024 blocks). In applications where independent preprocessing can be utilized our offline phase is superior to all previous 2PC protocols, in particular based on our experiments we see a 6-54x gain over [RR16] depending on network and number of circuits considered. If preprocessing is not applicable, for most settings we cannot compete with the offline phase of [RR16], but note that the difference is within a factor 1.2-3x. See Table 4.2 for an overview of our performance in different settings and Section 4.6 for a more detailed presentation and comparison of our results.

4.2 Preliminaries

In this section we give some of the technical background for LEGO garbling, adopting the notation and conventions of the original TinyLEGO protocol [FJNT15] for ease of exposition.

Circuit Conventions

We assume A is the party constructing the garbled gates and call it the *constructor*. Likewise, we assume B is the party evaluating the garbled gates and call it the *evaluator*. Furthermore, we say that the functionality they wish to compute is $z = f(x, y)$, where A gives input x and B gives input y. We assume that f is described using only NOT, XOR and AND gates. The XOR gates are allowed to have unlimited fan-in, while the AND gates are restricted to fan-in 2, and NOT gates have fan-in 1. All gates are allowed to have unlimited fan-out. We denote the bit-length of x by $|x| = n_{\mathsf{A}}$, the bit-length of y by $|y| = n_{\mathsf{B}}$ and let $n = n_{\mathsf{A}} + n_{\mathsf{B}}$. We will denote the bit-length

of the output z by $|z| = m$. Furthermore, we assume that the first n_{A} input wires are for A's input and the following n_{B} input wires are for B's input.

We define the *semantic* value of a wire-key of a garbled gate to be the bit it represents. We will use K_j^b to denote the j'th wire key representing bit b. Sometimes, when the context allows it, we will let $L_g^{b_l}$, $R_g^{b_r}$, and $O_g^{b_o}$ denote the left input, right input, and output key respectively for garbled gate g representing the bits b_l, b_r and b_o. When the bit represented by a key is unknown we simply omit the superscript, *e.g.* K_j.

Free-XOR and Soldering

The LEGO protocols [NO09, FJN+13] and [FJNT15] all assume that the underlying garbling scheme supports the notion of Free-XOR [KS08], meaning that the XOR of the 0- and 1-key on any wire of any garbled gate yields the same value, Δ, which we call the *global difference*. In addition to making garbling and evaluating XOR gates virtually free, this optimization also allows for easily soldering wires together. A soldering of two wires is a way of transforming a key representing bit b on one wire into a key representing bit b on the other wire. As we will see in more detail below, with Free-XOR, a soldering is simply the XOR of the 0-keys on the two wires. Furthermore, in order to avoid any cheating all wires of all garbled gates are committed to using a XOR-homomorphic commitment functionality $\mathcal{F}_{\mathsf{HCOM}}$ and the solderings are then always decomitted when needed.

As an example, assume we wish to solder the output wire of gate g onto the left input wire of gate $g+1$. In doing so we decommit the value $S_{g+1}^L = O_g^0 \oplus L_{g+1}^0$ using $\mathcal{F}_{\mathsf{HCOM}}$. When gate g outputs the key representing the bit b one can now learn the left b-key for gate $g+1$. Specifically it can be computed as $O_g^b \oplus S_{g+1}^L = \left(O_g^0 \oplus (b \cdot \Delta)\right) \oplus O_g^0 \oplus L_{g+1}^0 = L_{g+1}^0 \oplus (b \cdot \Delta) = L_{g+1}^b$. This obviously generalizes when one wishes to solder together several wires, *e.g.* if we wish to solder the output wire of gate g to the left input wire of gate $g+1$, $g+2$ and $g+3$, then it is enough to decommit the values $S_{g+1}^L = O_g^0 \oplus L_{g+1}^0, S_{g+2}^L = O_g^0 \oplus L_{g+2}^0, S_{g+3}^L = O_g^0 \oplus L_{g+3}^0$.

It is also straightforward to evaluate XOR gates as part of the soldering: To compute the XOR of g and $g+1$ and then use this result as the left input to gate $g+2$ we decommit the value $S_{g+2}^L = \left(O_g^0 \oplus O_{g+1}^0\right) \oplus L_{g+2}^0$. We see now that $O_g^a \oplus O_{g+1}^b \oplus S_{g+2}^L = a \cdot \Delta \oplus b \cdot \Delta \oplus L_{g+2}^0 = L_{g+2}^{a \oplus b}$ as desired. In conclusion a soldering is therefore always the XOR of the 0-keys of the wires going into an XOR gate and the 0-key of the wire we wish to solder the result onto.

4.3 A New Approach to Eliminate the Selective-OT Attack

As already mentioned in Section 4.1 the selective-OT attack enables a corrupt A to learn any input bit of B with success probability $1/2$ for each bit. Prior work has dealt with this attack in different ways, but the off-the-shelf black-box solution has typically been the s-probe resistant matrix approach of [LP07, sS13, LR15]. These approaches augment the evaluation circuit $f \to f'$ so that learning any $s-1$ input bits of B in f' leaks nothing about the actual input used in the original f. The downside of this approach is that the input length of B needs to be increased to $n'_{\mathsf{B}} > n_{\mathsf{B}}$, which in turn results in more communication, computation and OTs. For the approach of [LP07] and [LR15] the increase is to $n'_{\mathsf{B}} = n_{\mathsf{B}} + \max(4n_{\mathsf{B}}, {}^{20s}/_{3})$ while for the approach of [sS13] we have $n'_{\mathsf{B}} \leq n_{\mathsf{B}} + \lg(n_{\mathsf{B}}) + n_{\mathsf{B}} + s + s \cdot \max(\lg(4n_{\mathsf{B}}), \lg(4s))$. In addition to extending the input size, experiments of [LR15] show that producing the s-probe resistant version of f can be a computationally expensive task (up to several seconds for 1000-bit input).

Our New Approach

We propose a new approach that combines the use of 1-out-of-2 Δ-ROTs (also called globally correlated OTs), XOR-homomorphic commitments and the Free-XOR technique that sidesteps the need of expanding the input size of B as described above. We recall that Δ-ROTs are similar to Random OTs (ROT), except that all OTs produced are correlated with a global difference Δ. In other words, for each Δ-ROT i produced, the following relation holds for a fixed Δ: $r_i^1 = r_i^0 \oplus \Delta$ where $r_i^0, \Delta \in \{0,1\}^\kappa$ are uniformly random strings known to the sender and $b_i \in \{0,1\}$ is the uniformly random choice-bit of the receiver who learns $r_i^{b_i}$ as part of the OT protocol. Our approach is described below and is inspired by the protocol of Beaver [Bea95] for precomputing OT.

1. The parties precompute $(n_{\mathsf{B}} + s)$ Δ-ROTs such that the sender learns (Δ, r_i^0) and the receiver learns $(r_i^{b_i}, b_i)$ for $i \in [n_{\mathsf{B}} + s]$. The sender will now commit, using the XOR-homomorphic commitment scheme, to Δ and each r_i^0. In order to verify that the sender indeed committed to the Δ used in the OTs, the parties run a simple check in the following way: B sends $\left\{(r_j^{b_j}, b_j)\right\}_{j \in [n_{\mathsf{B}}; n_{\mathsf{B}}+s]}$ to A which in turn needs to successfully decommit to the received values. The s OTs used for the check are hereafter discarded. The reason why B needs to send the values to A in the first place is that it needs to prove knowledge of the value $r_j^{b_j}$ before it is safe for A to open it. For each of the s tests, if A did not commit to the Δ used in the OTs, then it can only pass the test with probability at most $1/2$ as the choice-bits of B are uniformly random. Because these

are also chosen independently we see that the check therefore catches a cheating A with overwhelming probability $1 - 2^{-s}$.

2. If the check succeeds, A uses the Δ learned from the OTs above as the global difference in the Free-XOR garbling scheme. Recall that this means that all garbling keys will be correlated in the same way as the Δ-ROTs, *i.e.* $K_l^1 = K_l^0 \oplus \Delta$ for all l. In particular this is the case for the keys associated to the input of B which need to be obliviously transferred in the online phase. In addition, in our 2PC protocol all 0-keys K_l^0 have been committed to using the same XOR-homomorphic commitment scheme as used for the OT strings r_i^0 and Δ.
3. Finally when B learns its real input y, it computes $e = y \oplus b$ and sends this to A where b are the choice-bits used in the precomputed Δ-ROTs. A will respond by decommitting the values $\{D_i = K_i^0 \oplus r_i^{e_i}\}_{i \in [n_\mathsf{B}]}$. B can now compute its actual input keys $K_i^{y_i} = D_i \oplus r_i^{b_i} = K_i^0 \oplus r_i^{e_i} \oplus r_i^{b_i} = K_i^0 \oplus y_i \cdot \Delta$.

The above approach eliminates the selective-OT attack as the only way a corrupt A can cheat is by committing to different values $r'^0_i \neq r_i^0$ where r_i^0 is the value sent using the Δ-ROTs. However if this is the case then $D'_i \oplus r_i^{b_i} \notin \{K_i^0, K_i^1\}$ and B will abort regardless of the value of his input y_i. One caveat of the above approach is that it allows a corrupt A to flip an input bit i of B without getting caught by committing to $r_i^0 \oplus \Delta$ instead of r_i^0. In our 2PC protocol we eliminate this issue by ensuring that $\mathsf{lsb}(\Delta) = 1$ and by securely leaking $\mathsf{lsb}(r_i^0)$ to B. This allows B to check that the resulting key $K_i = D_i \oplus r_i^{b_i}$ indeed carries the correct value y_i, by verifying that

$$\begin{aligned} y_i &= \mathsf{lsb}(K_i) \oplus \mathsf{lsb}(D_i) \oplus \mathsf{lsb}(r_i^0) \oplus e_i \\ &= \mathsf{lsb}(K_i) \oplus \mathsf{lsb}(K_i^0) \oplus \mathsf{lsb}(r_i^{e_i}) \oplus \mathsf{lsb}(r_i^0) \oplus e_i \\ &= \mathsf{lsb}(K_i) \oplus \mathsf{lsb}(K_i^0) \ . \end{aligned}$$

This secure leaking is described in the *VerLeak* step of Figure 4.1 and the check is carried out as part of the **Eval** step of Figure 4.2, both of which are presented in Section 4.4.

On Constructing Δ-ROTs

The above technique requires $(n_\mathsf{B} + s)$ Δ-ROTs to obliviously transfer the input keys of B. However, current state-of-the-art protocols for OT extension [NNOB12, BLN+15, ALSZ15, KOS15] all produce ROTs. It can be seen by inspecting the above OT extension protocols that they all produce a weaker variant of Δ-ROT called *leaky* Δ-ROT as an intermediate step. The leaky Δ-ROT is identical to Δ-ROT in that all OT pairs are correlated with a global Δ, however a corrupt receiver can cheat and learn some bits of Δ with non-negligible probability. In fact, for each bit learned of Δ, the receiver gets caught with probability $1/2$, which means it can learn up to $s - 1$ bits

of Δ, while the other bits remain uniformly random in its view. The work of [BLN+15] gives a construction for Δ-ROTs of string length v from leaky Δ-ROTs of string length $22v/3 \sim 7.33v$ using linear randomness extraction. Concretely, they propose multiplying all the strings learned from the leaky Δ-ROT protocol with a random matrix $A \in \{0,1\}^{22v/3 \times v}$. In this work we observe that the factor $22/3$ is not tight and by applying Theorem 2 below we can reduce the number of rows in A down to $v + s$, going from a multiplicative to an additive factor.

Theorem 2 ([ZB11], Theorem 7)**.** Let $X = x_1, x_2, \ldots, x_u$ be a binary sequence generated from a bit fixing source in which l bits are unbiased and independent, the other $u - l$ bits are fixed or copies of the l independent random bits. Let A be a $u \times v$ random matrix such that each entry of A is 0 or 1 with probability $1/2$. Given $Y = XA$, then we have that

$$\Pr_A[\rho(Y) \neq 0] \leq 2^{v-l}$$

where $\rho(Y)$ is defined as the statistical distance to the uniform distribution over $\{0,1\}^v$, *i.e.* $\rho(Y) =$
$\frac{1}{2}\sum_{y \in \{0,1\}^v} |\Pr[Y = y] - 2^{-v}|$.

Now let $u = v + s$ and let Δ have length u and let ΔA have length v. Consider an adversary B who is allowed to try to learn some of the bits of Δ to make ΔA non-uniform. In our setting, if an adversary B tries to learn λ bits of Δ it is caught except with probability $2^{-\lambda}$. If B is not caught then it learns λ bit positions and the remaining bits are independent and uniform. Since we have $u = v + s$, the l in the above theorem equals $u - \lambda = v + (s - \lambda)$ when B learns λ bits, *i.e.*, $2^{v-l} = 2^{\lambda - s}$. This implies that for all $0 \leq \lambda < s$ it holds that the probability that B is not caught and at the same time ΔA is not uniform is at most $2^{-\lambda}2^{\lambda-s} = 2^{-s}$. Clearly, for all $\lambda \geq s$, then the probability that B is not caught and at the same time ΔA is not uniform is at most the probability B is not caught, which is at most 2^{-s}. This shows that when $u = v + s$ then for all B, the probability that B is not caught and at the same time ΔA is not uniform is at most 2^{-s}, which is negligible.

The consequence of our new analysis is that we can choose the random matrix as $A \in \{0,1\}^{(v+s)\times v}$ and thus we only have to produce leaky Δ-ROTs of length $v + s$ instead of length $22v/3$, a substantial optimization. As we ultimately require (non-leaky) Δ-ROTs of length κ, we can utilize any of the mentioned OT extension protocols to produce leaky Δ-ROTs of length $\kappa + s$ and then apply the linear randomness extraction on the resulting OT-strings. For the parameters $s = 40$ and $\kappa = 128$ considered in this work our refined analysis ultimately yields an improvement of around a factor 5.6x compared to the previous best known result of [BLN+15].

4.4 The Protocol

As already mentioned in Section 4.1, our protocol is based on TinyLEGO [FJNT15], but modified to support preprocessing of all garbled components along with our new approach for dealing with the selective-OT attack of Section 4.3. This includes removing the restriction of B choosing input and committing to the cut-and-choose challenges before obtaining the garbling material and solderings. As a consequence, our modifications allow for multi-leveled preprocessing and a very efficient online phase. We give a description of our resulting protocol in Figure 4.1 and Figure 4.2. See [FJNT15] for a more detailed specification of the original TinyLEGO protocol. At a high level our protocol can be broken down into four main steps.

1. The **Setup** phase initializes the commitment scheme. All public-key operations of our protocol can be carried out in this initial step, including the BaseOTs required for bootstrapping OT extension.
2. The **Generate** step takes as input the number of gates q, number of inputs n and number of outputs m the parties wish to preprocess. After sending the garbled gates and wire authenticators and committing to all associated wires, a cut-and-choose step is run between the parties. The wire authenticators is a gadget that either accepts or rejects a given key (without revealing the value of the key) and it was shown in [FJNT15] that constructing AND buckets from both garbled gates and wire authenticators can significantly reduce the overall communication compared to using garbled gates alone. After the cut-and-choose step, using the XOR-homomorphic commitments, the parties solder the remaining garbled gates and wire authenticators randomly into independent fault tolerant AND buckets.
3. The **Build** step takes as input the circuit description f and through the XOR-homomorphic commitments, A sends the required solderings to glue together a subset of previously produced AND buckets so that they compute f.
4. Finally, the **Eval** step depends on the parties' inputs to f. It consists of two rounds, first B sends a correction value e which depends on his input y, and as a response A decommits to B's masked input keys as well as sending it's own input keys directly. Finally A also decommitments to the lsb of all output 0-keys. This allows B to evaluate the garbled circuit and decode the final output.

We highlight that our modified protocol also naturally supports the notion of *streaming* or *pipelining* of garbled circuit evaluation [HEKM11] which was not the case in [FJNT15]. This can be seen by the fact that one can evaluate the circuit f in a layered approach and using the XOR-homomorphic commitments to glue the output of one layer onto the input of the next layer. Each of these

The Setup step is only run once, regardless of the number of calls to Generate.
Setup($\mathtt{pp}$):

1. On input $(\kappa, s, p_g, p_a, \beta, \alpha, \lambda_g, \lambda_a) \leftarrow \mathtt{pp}$, A and B initialize the functionality $\mathcal{F}_{\mathsf{HCOM}}$ by sending $(\mathtt{init}, \mathtt{sid}, \mathsf{A}, \mathsf{B}, \kappa)$ to it, where κ is the key-length of the garbling scheme.

The Generate step produces q garbled AND gates which can be soldered into circuits that in total can have n inputs and m outputs. It is possible to do multiple calls to Generate in order to produce more garbling material.
Generate(q, n, m):

1. Let Q and A be chosen such that after running the below cut-and-choose step, with overwhelming probability $(q\beta + n\lambda_g)$ garbled gates and $(q\alpha + n\lambda_a)$ wire authenticators survive.
2. A and B invoke $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$ $(n + s)$ times from which A learns Δ and random strings r_i^0 and B learns the choice-bits b_i and $r_i^{b_i}$ for $i \in [n+s]$. Furthermore, A instructs $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$ to ensure that $\mathsf{lsb}(\Delta) = 1$.
3. Next, A garbles Q AND gates and constructs A wire authenticators using Δ and sends these to B.
4. A then commits to each wire of the garbled AND gates, each authenticated wire produced, Δ, the 0-strings received from $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$, and $m + s$ random values $\{v_j\}_{[m+s]}$. Thus it sends $3Q + A + 1 + n + s + m + s$ values to $\mathcal{F}_{\mathsf{HCOM}}$.

VerLeak:

5. For $i \in [n]$ and $j \in [m + s]$, A sends $\mathsf{lsb}(r_i^0)$ and $\mathsf{lsb}(v_j)$ to B.
6. B challenges A to send, using $\mathcal{F}_{\mathsf{HCOM}}$, s random linear combinations of r_i^0, v_j and Δ for $i \in [n]$ and $j \in [m]$. Also, the l'th combination is set to include a one-time blinding value v_{m+l} for $l \in [s]$.
7. B verifies that lsb of the received s values correspond to the same linear combinations of the initial lsb values sent by A in step 5. In addition, if Δ is included in a linear combination B flips the value. This ensures that indeed $\mathsf{lsb}(\Delta) = 1$.

Cut-and-Choose:

8. After receiving the garbled gates (wire authenticators), B chooses to check any gate (wire authenticator) with probability p_g (p_a). B then challenges A to send, using $\mathcal{F}_{\mathsf{HCOM}}$, two random inputs and the corresponding AND output of the selected garbled gates and a random input of the selected wire authenticators.
9. B evaluates the selected garbled gates and wire authenticators and checks that they output the received output key and that they verify the values received from $\mathcal{F}_{\mathsf{HCOM}}$, respectively.
10. In addition, for $i \in [n; n + s]$ B sends $r_i^{b_i} = r_i^0 \oplus (b_i \cdot \Delta)$ to A which in turn instructs $\mathcal{F}_{\mathsf{HCOM}}$ to send back the same value. This is to ensure that the committed Δ is the one used in $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$.

Figure 4.1: The modified TinyLEGO protocol with support for preprocessing in the ($\mathcal{F}_{\mathsf{HCOM}}$, $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$)-hybrid model (Part 1).

Generate(q, n, m) **(continued):**
Bucketing:

11. For the remaining garbled gates (wire authenticators), B samples and sends a random permutation that fully describes how these are to be combined into q AND buckets of size $\beta + \alpha$, n input buckets of size λ_g and n input authenticators of size λ_a.
12. A then sends, using $\mathcal{F}_{\mathsf{HCOM}}$, all the required solderings such that for all the specified bucket gadgets, each component is defined with the same input/output keys.

The Build step uses the garbling material created in Generate to construct a fault tolerant garbled circuit computing f.
Build(f)**:**

1. A instructs $\mathcal{F}_{\mathsf{HCOM}}$ to send the required solderings such that the first $|f|$ unused AND buckets correctly compute f. This includes the solderings to attach n_{A} input buckets and n input authenticators onto the final garbled circuit.

In the Eval step the parties transfer to B all input keys in an oblivious manner which then allows B to evaluate and decode the garbled circuit previously constructed using the Build step.
Eval(x, y)**:**

1. For input $y \in \{0,1\}^{n_{\mathsf{B}}}$, B sends $e = b \oplus y$ to A, where b is the first n_{B} unused choice-bits of $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$.
2. A then instructs $\mathcal{F}_{\mathsf{HCOM}}$ to send to the values $\left\{D_i = r_i^0 \oplus K_i^0 \oplus e_i \cdot \Delta\right\}_{i \in [n_{\mathsf{B}}]}$ where K_i^0 is the 0-key on the i'th input wire of B and r_i^0 is the first unused Δ-ROT string. Also, for the input $x \in \{0,1\}^{n_{\mathsf{A}}}$, it sends the corresponding input keys $\{K_i^{x_i}\}_{i \in [n_{\mathsf{A}}]}$ directly to B.
3. Finally, A instructs $\mathcal{F}_{\mathsf{HCOM}}$ to send the output decoding values $\left\{D_j = v_j^0 \oplus K_j^0\right\}_{j \in [m]}$ to B where K_j^0 is the j'th output 0-key of the garbled circuit and $\left\{v_j\right\}_{j \in [m]}$ are the first m unused blinding values setup in the *VerLeak* step.
4. Upon receiving the above, for $i \in [n_{\mathsf{B}}]$ B computes $K_{n_{\mathsf{A}}+i} = r_i^{b_i} \oplus D_i$ and $\mathsf{lsb}(K_i^0) = \mathsf{lsb}(D_i) \oplus \mathsf{lsb}(r_i^0) \oplus e_i$ and verifies that $\mathsf{lsb}(K_{n_{\mathsf{A}}+i}) \oplus \mathsf{lsb}(K_i^0) = y_i$. Then using the input authenticators, B also verifies that the keys $\{K_i\}_{i \in [n_{\mathsf{A}}]}$ of A are valid input keys to the garbled circuit.
5. If everything checks out, B evaluates the previously constructed garbled circuit on the input keys $(K_1, K_2, \ldots, K_n)$ to obtain the output keys $(Z_1, Z_2, \ldots, Z_m)$. For $j \in [m]$ it then computes $d_j = \mathsf{lsb}(v_j) \oplus \mathsf{lsb}(D_j)$ and decodes $z_j = \mathsf{lsb}(Z_j) \oplus d_j$. Finally, B outputs $z = (z_1, z_2, \ldots, z_m)$.

Figure 4.2: The modified TinyLEGO protocol with support for preprocessing in the ($\mathcal{F}_{\mathsf{HCOM}}$, $\mathcal{F}_{\Delta\text{-}\mathsf{ROT}}$)-hybrid model (Part 2).

layers can be processed on the fly with our protocol and in this way the circuit never needs to be stored entirely at any given time. This approach is similar to that proposed in [MGBF14] for reusing garbled values, however in this setting everything works out-of-the-box due to the XOR-homormophic commitments on all circuit wires.

The LEGO approach also has the advantage compared to traditional cut-and-choose protocols that only a *single fault tolerant* garbled circuit is produced and evaluated. This removes the necessity of ensuring input consistency for all the evaluation circuits. It also sidesteps the overhead of transferring multiple sets of input keys in the online phase, one set for each evaluation circuit.

Bucketing

With the bucketing approach of [FJNT15] each AND bucket consists of β garbled gates and α wire authenticators. For any garbled gate (wire authenticator) the probability p_g (p_a) is used to determine if it is checked in the cut-and-choose or not. The value of p_g and p_a therefore induces a certain sense of "quality" level of the remaining non-checked garbled components which affects the required bucketing size. In addition there are also the special cases of input buckets and input authenticators, which are buckets that consist of garbled gates only (size λ_g) and wire authenticators only (size λ_a), respectively. These are attached to the input wires of the final garbled circuit and serve to guarantee validity of the input keys, along with guaranteeing that B can always learn the final output $f(x, y)$, even if A is cheating. This is so since the input buckets can be seen as a trapdoor that together with the global difference Δ allows B to extract the input x of A. It is then clear that it can compute $f(x, y)$ directly. These special buckets are necessary as our regular AND buckets do not rule out outputting both the 0 and 1-key (say if one of the garbled gates in the bucket is in fact a NAND gate). However, if a bucket outputs two distinct keys it is guaranteed that they are both valid and hence their XOR is Δ and B can extract x. If no cheating is detected then the input buckets are simply ignored by B.

As already established in the original LEGO paper [NO09], the number of AND gates q directly affects the required size of the buckets, meaning that as q grows the required bucket size can be decreased while still retaining the same level of security. Theorem 3 below gives a direct way of computing the success probability of a corrupt A given the parameters $q, n, \beta, \alpha, \lambda_g, \lambda_a$.

Theorem 3 ([FJNT15], Lemma 9)**.** Given the bucketing parameters q, n, β, α, λ_g, λ_a for the case where $\alpha = \beta - 1$ we can bound the probability of the bad bucketing events occurring as:

$$\Pr[\text{Any bad bucket}] \leq$$
$$q \cdot \Bigg(\prod_{i=\beta}^{1} \Big(\frac{(1-p_g)4i}{p_g(q\beta + n\lambda_g) + (1-p_g)4i} \Big) + \sum_{l=2}^{\beta} \prod_{i=\beta}^{l} \Big(\frac{(1-p_g)4i}{p_g(q\beta + n\lambda_g) + (1-p_g)4i} \Big) \cdot \prod_{j=\alpha}^{\alpha+2-l} \Big(\frac{(1-p_a)2j}{p_a(q\alpha + n\lambda_a) + (1-p_a)2j} \Big) \Bigg)$$

$$\Pr[\text{Any bad input authenticator}] \leq$$
$$n \cdot \sum_{v=1}^{\lceil \frac{\lambda_a}{2} \rceil} \prod_{l=\lambda_a}^{v} \Big(\frac{(1-p_a)2l}{p_a(q\alpha + n\lambda_a) + (1-p_a)2l} \Big)$$

$$\Pr[\text{Any bad input bucket}] \leq$$
$$n \cdot \sum_{l=1}^{\lceil \frac{\lambda_g}{2} \rceil} \prod_{i=\lambda_g}^{l} \Big(\frac{(1-p_g)4i}{p_g(q\beta + n\lambda_g) + (1-p_g)4i} \Big)$$

Based on Theorem 3, given the number of AND gates q and the number of inputs n we directly compute the optimal choices of $\beta, \alpha, \lambda_g, \lambda_a$ for minimizing the overall communication of the protocol while still guaranteeing a negligible upper bound on the success probability of a corrupt A. This is a once and for all computation so for our implementation we have precomputed a table of secure choices using a simple script which is looked up on runtime when q and n have been decided. We note that it is also possible to minimize for lowest possible bucket size if desired. This has the effect of reducing the computational overhead in the online phase at the price of increasing both communication and computational overhead in the independent preprocessing phase. In our experiments in Section 4.6 we solely minimize for overall communication.

Security

Our protocol is similar to the TinyLEGO protocol in [FJNT15] and the proof follows the same general outline. We will therefore only give a very brief sketch of the overall proof strategy and then describe how to deal with the changes we made relative to TinyLEGO.

Consider first the case where the garbler A is corrupted. As is typically the case it is easy to see that the communication of the protocol does not leak any information on the input of B as long as the protocol does not abort. The garbler A might however give wrong input to some of the OTs used by B to choose its input keys, giving rise to selective errors where the abort probability depends on the input of B. The garbler A might also create some bad garbled gates which could *a priori* result in an abort or a wrong output, which might both leak information on the input of B. The problem with bad gates is handled exactly as in [FJNT15], by setting the cut-and-choose parameters and bucket sizes appropriately. We however handle the case with bad inputs to the OTs differently, as described below. In the universal composability (UC) framework [Can01], when A is corrupt, we also need to be able to extract the input of the corrupted A from its communication and input to ideal functionalities (OT and commitment). We handle this exactly as [FJNT15]: the cut-and-choose ensures that most key authenticators only accept their two corresponding committed keys. For a good key authenticator the accepted key can then be compared to the committed values to compute its semantic value. The bucket size has been set such that there is a majority of good key authenticators on all input wires. This allows to compute the semantic of any accepted key by taking majority.

Consider then the case where the evaluator B is corrupted. As is typically the case, the communication clearly does not leak information to B about the input of A. All that is left is therefore to describe how to handle two technical requirements imposed by the UC framework. First, we have to describe how to extract the input y of a corrupted B. Second, after learning y and $z = f(x, y)$ we must enforce that the simulated protocol constructs a circuit that evaluates to z. This must be done without knowing x. Extracting y is handled exactly as in [FJNT15]. We simply inspect which choice bits B uses in the OTs for selecting its input. Hitting z in the simulation is also handled exactly as in [FJNT15]. We simply construct the circuit correctly and run with input 0 for A. This gives a potentially wrong output $z' = f(0, y)$. We patch this by giving appropriately chosen wrong output decoding information by opening a wrong least significant bit of the output key for the output wires i where $z'_i \neq z_i$. This is possible as the simulator controls the ideal functionality for commitment in the simulation.

We now focus on the changes we made to [FJNT15].

Change 1 In both protocols the output decoding information consists of the least significant bit of the output keys, securely leaked via the commitment scheme. However, the implementation differs. We use a non-interactive implementation which is slightly heavier on communication. In [FJNT15] they use an interactive protocol with less communication.

Change 2 In [FJNT15] they let B commit to the cut-and-choose challenges

and choose his input via OT before obtaining the garbled gates, wire authenticators and solderings. We have removed this step. Now B picks his input after the circuit is constructed and does not commit to his challenges.

Change 3 In our protocol we take the global Δ-value output by the OT extension and reuse it as the global Δ in the Free-XOR garbling scheme. In [FJNT15] they use two independent values.

Change 4 We protect against selective error on the input of B by using the same Δ in OT extension and garbling and using a Δ-ROT to offer the input keys to B. We also leak the least significant bit of the Δ-OT 0-strings to B for all his input wires. In [FJNT15] a different technique was used.

Change 1 does not affect security. It was introduced to give better execution time for typical circuits.

We now address Change 2. The reason why B commits to the cut-and-choose challenges and chooses his input via OT before obtaining the garbled material in [FJNT15] is that security is proven via a reduction to a standard (non-adaptive) selective garbling scheme (*e.g.* [BHR12b]), where the adversary in the security game must supply its input before it gets the garbled circuit. Therefore they need to be able to extract the input and cut-and-choose challenges of B before assembling the garbled circuit in the simulation. We have skipped this step as it would prevent independent preprocessing. Now that we assemble the circuit before B picks its input, the hope would be that we could do a reduction to an adaptive garbling scheme. However, due to the soldering approach of LEGO where the XOR of 0-keys are sent to the evaluator before the input is determined, it is unclear how to reduce security to the standard notion of adaptive garbling, as some of these 0-keys are not known to the simulator. To overcome this we instead identify the defining property we need from the underlying garbling function which is that *the outputs of the hash function appear random as long as the inputs are all unknown and have sufficient entropy, even if the inputs are related.* A non-extractable, non-programmable random oracle clearly satisfies this property [BR93]. The type of garbling scheme considered in this work [ZRE15] could in principle be made adaptively secure by using a programmable random oracle using techniques described in [BHR12a, LR14].[1] We avoid using the programability of the random oracle by changing the usual approach a little. Normally security proofs need to program the circuit to hit the right output. We instead garble the circuits correctly and then program or equivocate the output decoding information to decode to the value we need to hit. Specifically we use equivocation of the UC commitment scheme to incorrectly open the least significant bit of the output

[1] It is also possible to build an adaptively secure garbling scheme (with short input keys) using a non-programmable random oracle [BHK13], but this particular scheme is not as efficient as the one of [ZRE15].

keys when we need to hit a different value. Returning to the simulation, we therefore garble all gates and answer all cut-and-choose challenges honestly. As we now know all garbling keys we can also open consistently to differences between 0-keys for the remaining evaluation gates. Finally, in order to make the complete soldered garbled circuit "hit" the output z we equivocate the openings of the least significant bits of the output keys such that this becomes the decoded value. This is possible as the decoding information is only opened after we extract the input of the corrupt receiver. If the garbling is done using a random oracle this will have the same distribution as in the protocol.

For Change 3 we again use that we are in the random oracle model. The first step in the proof will be to go from the case where Δ is reused in the garbling scheme to the case where an independent Δ' is used for garbling as in [FJNT15]. In this hybrid we also let A commit to Δ'. We then use equivocation of the commitment scheme to make the cut-and-choose proof that $\Delta' = \Delta$ go through. Since B only sees one key for each wire and has high entropy on Δ and Δ', this change will be indistinguishable to B if the output of the hash function appears random as long as the inputs are all unknown and have sufficient entropy, even if inputs are related. As above, a non-extractable, non-programmable random oracle clearly satisfies this property.

We finally address Change 4. Using a Δ-ROT to offer the input keys to B ensures that when A inputs the keys to the OT, either both are correct or both are incorrect. If both are incorrect, it will be detected by a key authenticator independently of the input of B. This means that the only remaining attack vector is for A to swap the two correct keys. This is detected by B as B knows the least significant bit of the Δ-ROT 0-strings and the two correct keys have different least significant bits. Again the detection is independent of the input of B. Notice that the output decoding information is sent to B using $\mathcal{F}_{\mathsf{HCOM}}$ *after* he sends his input correction value e, so we can equivocate it to hit the correct output z. This, together with the fact that the outputs of the hash function appear random, is why it is secure to perform the *VerLeak* step before learning the input of B.

As argued above, our modified protocol can be proven secure in the non-extractable, non-programmable random oracle model following the proof of [FJNT15]. As we do not require programmability of the random oracle we conjecture that our protocol can be proven secure in the standard (OT hybrid) model using the recently proposed ICE framework of [FM16], an extension of the UCE framework of [BHK13]. However, we note that it does not seem like our scheme can be proven secure using the UCE framework as in our setting all the garbled gates are related (all garbled with the same Δ) and therefore a single leakage phase as prescribed in the UCE framework seems insufficient.

4.5 Implementation

In this section we highlight some of the more technical details of our implementations of the XOR-homomorphic commitment scheme and our final 2PC protocol supporting independent preprocessing. The source code of the project can be found at `https://github.com/AarhusCrypto/TinyLEGO`.

UC-Secure XOR-Homomorphic Commitments

As part of our full 2PC prototype we implement the XOR-homomorphic commitment scheme of [FJNT16] as a separate subprotocol. This is to our knowledge the first time a scheme following this OT + PRG blueprint has been implemented and we believe our experimental findings are of independent interest. At a high level, the scheme works by the parties initially doing n BaseOTs of security parameter κ-bit strings, where n is the code-length for some linear error correcting code $\mathcal{C}$ with parameters $[n, \kappa, s]_{\mathbb{F}_2}$ with κ being the bit-length of the committed messages. The parties then expand the received κ-bit strings into bit-strings of length γ using a PRG which then define the γ random commitments the sender is committing to. Next, the sender sends a correctional value for each produced commitment to turn these into codewords of $\mathcal{C}$. Finally, to ensure that the sender sent valid corrections, the receiver challenges the sender to decommit to $2s$ random linear combinations of all produced commitments. This is done in a way such that no information is leaked about the γ committed values. Additively homomorphism then follows from the fact that the code $\mathcal{C}$ is linear and all operations on the expanded PRG strings are linear as well. We highlight the fact that any XOR homomorphic commitment scheme supports the notion of batch opening/decommitment which is similar in nature to the above consistency check. The idea is that the sender initially sends the decommited values directly to the receiver, who hereafter challenges the sender to decommit to s linear combinations of the postulated values where s is the statistical security parameter. Notice that it is only in the initial commit step that $2s$ combinations are necessary. If the decommitted values match the linear combinations of the postulated values, the receiver accepts. As now only s values are decommitted this approach has the benefit of making the communication overhead independent of the number of values decommitted to. For the full details we refer to [FJNT16].

We implement the above scheme in C++14 taking advantage of multi-core capabilities and Intel SSE instructions. We can therefore base the PRG on AES-NI in counter mode and for the error correcting code we use a modified version of the linux kernel implementation of the BCH code family [Hoc59, BRC60]. As part of the commitment step the parties are required to transpose a binary matrix $S \in \{0,1\}^{n \times \gamma}$ in order to efficiently address the committed values in column-major order. As γ in our case can be huge ($>$ 220 Mio. for 2000 AES-128 computations) we use the efficient implementation of Ekhlund's

cache-efficient algorithm for binary matrix transposition [Ekl72] presented in [ALSZ13, ALSZ15].[2] As a side note we also augment the OT extension code to support the randomness extraction technique described in Section 4.3 to implement the $\mathcal{F}_{\Delta\text{-ROT}}$ functionality needed in our 2PC protocol.

During the development of our implementation we identified the main computational bottleneck of the scheme to be the computation of the random linear combinations. Even if these operations are based on mere XORs, when implemented naively, the number of required instructions is still γs in expectation. Therefore, inspired by [CDD+16], we use a different approach for computing the consistency checks using Galois field multiplication. Combined with efficient matrix transposition the effect of using $\mathrm{GF}(2^l)$ multiplication can be seen as computing l linear combinations in parallel. For our particular setting we set $l = 128$ as this is the smallest power of 2 greater than the required $2s$ for $s = 40$. We can then use the Intel SSE instruction PCLMULQDQ to very efficiently compute the $\mathrm{GF}(2^{128})$ multiplications. In detail, our approach to compute the checks is as follows:

1. Given a random challenge element $\alpha \in_R \mathrm{GF}(2^l)$ the matrix S of committed values in column-major order is split into $u = \lceil S/l \rceil$ blocks $B_i \in \{0,1\}^{n \times l}$. Each block is then transposed into row-major order.
2. For $i \in [u]$ and $j \in [n]$, the new matrix $B'_i = B_i^j \cdot \alpha^i$ is computed where $B_i^j \in \{0,1\}^l$ is the j'th row of B_i interpreted as an element of $\mathrm{GF}(2^l)$.
3. Finally, the combined matrix $B' = \sum_{i=1}^{u} B'_i$ is produced and transposed back into column-major order.

Each column of B' can now be seen as a random linear combination of all values of S. As a further optimization we see that most GF elements are only multiplied a single time in the above and we can therefore postpone the expensive degree reduction step of the multiplication until B' has been fully computed. This is different for computing α^i which we therefore reduce at each iteration. In total the required number of degree reductions becomes $n + u$ as opposed to $(n + 1)u$. Our experiments show that using the above method of computing 128 linear combinations compared to the naive approach is between 10-13x faster starting at a moderate number of random commitments $\gamma > 8000$.

As our implementation of the XOR-homomorphic commitment scheme might be of independent interest we here present our observed timings for committing and decommitting to γ random bit-strings of length 128 with $\kappa = 128$ and $s = 40$. We instantiate the binary BCH code with parameters $[312, 128, 41]$ and for convenience we use the implementation of [ALSZ15] augmented with our randomness extraction technique to compute the required 312 Random OTs. In total this takes about 1400 ms with our implementation, where 1392 ms are due to the BaseOTs (using PVW [PVW08]). From the timings reported in [CO15] we predict that this initial setup step can be done

[2] Available at `https://github.com/encryptogroup/OTExtension`

γ	#Threads	Commit [µs]		Decommit [µs]
500	1	7.21	(2815.28)	2.20
1000	2	3.85	(1401.83)	1.40
15 000	4	0.64	(93.99)	0.34
50 000	8	0.57	(28.49)	0.22
500 000	20	0.45	(3.25)	0.17
10 000 000	200	0.21	(0.35)	0.14
200 000 000	400	0.20	(0.21)	0.14

Table 4.3: Timings for committing to γ strings of length 128 bit with $s = 40$. All timings are µs per commitment. The commit time in parentheses includes the cost of the initial BaseOTs.

much faster (around 20 ms) using their implementation, but since this requires a programmable random oracle assumption and this cost amortizes away as γ grows we did not pursue this. Also, if the commitment scheme is used in an application that already relies on oblivious transfer, OT extension can be used to produce the starting BaseOTs at very low cost. We report our findings in Table 4.3. As the scheme requires n BaseOTs to setup we include this cost in the commit timings in parentheses. It can thus be seen by comparing the commitment numbers how the initial OT cost amortizes away as γ increases. As there is no initial cost associated with decommitment these timings are only affected by the number of worker-threads we spawn. Furthermore, these experiments were performed on the local LAN setup described in Section 4.6 and *not* on the Amazon Web Services (AWS) architecture.

2PC with preprocessing using LEGO

We implement the TinyLEGO protocol with our modifications on top of the previously described commitment scheme. The code is also written in C++14 and makes heavy use of parallelism and Intel SSE instructions for garbling and evaluation of the garbled gates. At a high level, the **Generate** step is implemented by first partitioning the inputs (q, n, m) into t equally sized subsets for some parameter t. The main thread then starts t parallel executions of the generate step with two synchronization points, one where the commitment to Δ is sent (which only one execution is charged with), and one after the cut-and-choose step. The latter is necessary as the random permutation that describes the initial bucketing must only be revealed after all garbled components have been sent to B. We emphasize that it is due to our preprocessing being independent of the structure of f that we can trivially

parallelize the above step using any number of threads t. Due to the above design we also run t executions of the commitment scheme, however for the PRG expansion we use the same seed OT values in all executions. As the PRG is based on a block cipher in counter mode this is not an issue as execution $i+1$ sets it's counter sufficiently high compared to the i'th execution so there is no overlap. Since they all use the same seed OTs the choice-bits are also the same across all executions and they can therefore be combined in the same way as for a single execution.

The **Build** and **Eval** phases follow roughly the same design pattern as above. We note however that in these phases each thread is responsible for soldering and evaluating an entire circuit. The garbling and evaluation of garbled gates and wire authenticators are implemented purely as 128-bit SSE instructions to maximize performance. We base the hash function for garbling gates and producing wire authenticators on Fixed-Key AES-NI as advocated in [BHKR13]. This choice is mainly motivated by producing as comparable results as possible to previous works that are also based on Fixed-Key AES-NI.

The **Eval** phase consists of two rounds, one where B specifies the input mask and one where A replies with its keys and decommitments. A's reply has communication complexity κn_{A} for A's input keys and $(n+\kappa)(n_{\mathsf{B}}+m)$ for the decommitments of B's input keys and the lsb masks of the output keys, where n is the code-length of the BCH code. We note that the communication cost of the decommitments can be reduced to $(n+\kappa)s+\kappa(n_{\mathsf{B}}+m)$ using the batch decommit approach mentioned in Section 4.5, but at the cost of adding an additional round. For the circuits used in our experiments (AES-128, SHA-256) we observed a loss of around a factor 1.25 in the LAN setting and much more in the WAN setting with this approach. Still, for other circuits where the ratio $(n_{\mathsf{B}}+m)/|f|$ is substantial and both network latency and bandwidth are low we suspect that adding this extra round can pay off. Finally if one is willing to assume a programmable random oracle the online cost for the output bits can be eliminated entirely as the simulator then can program the oracle to output matching output keys for a preprocessed decommitment lsb-bit once it learns the final output.

4.6 Performance

To give a broad view of the performance of our prototype we run experiments in a local LAN setting and on both a LAN and WAN on the Amazon Web Services (AWS). In more detail:

Local LAN with two machines, one acting as A and the other acting as B. We measured a total bandwidth of 942 Mbits/sec with round trip time (rtt) 0.12 ms. Both machines run Ubuntu 16.04 with an Intel Ivy Bridge i7 3.5 GHz quad-core processor and 32 GB DDR3 RAM.

AWS LAN with two c4.8xlarge instances located in the Virginia region connected via a high performance LAN. We measured a bandwidth of 9.52 Gbits/sec with rtt 0.16 ms. Both machines run Amazon Linux AMI 2016.03.2 with an Intel Xeon E5-2666 v3 (Haswell) processor with 36 vCPUs and 60 GB RAM.

AWS WAN with two c4.8xlarge instances, one in Virginia and one in Ireland. We measured a bandwidth of 214 Mbits/sec on average for a single TCP connection and up to 3.17 Gbit/sec when running many parallel connections. The rtt measured was 81.32 ms. Both machines run Amazon Linux AMI 2016.03.2 with an Intel Xeon E5-2666 v3 (Haswell) processor with 36 vCPUs and 60 GB RAM.

For all settings the code was compiled using GCC-5.4 with the -O3 optimization flag set. As mentioned in Section 4.5 the implementation used for the BaseOTs are based on [ALSZ15] using PVW [PVW08]. If one is willing to assume a programmable random oracle, these can be replaced with the fast protocol and implementation of [CO15] and we would expect a total cost around 20 ms as opposed to 850 ms (AWS LAN) with the current implementation.

Our Performance Results

We summarize our measured results in Table 4.4 for the three above-mentioned settings. All numbers reported are averages of 10 executions. Not surprisingly we see the best performance on the AWS machines in the LAN setting where we can evaluate an AES-128 circuit with latency 1.13 ms or 0.08 ms throughput per AES-128 in the online phase. We also see that when considering 1024 AES-128 evaluations the dependent preprocessing + the online phase is below 2 ms. When including the cost of the independent preprocessing each AES-128 can be done in total time less than 16 ms. Similarly when considering 256 SHA-256 evaluations the online phase can be done with latency 9.14 ms or 1.05 ms of throughput per SHA-256. Also the dependent preprocessing + online phase and total cost is below 22 ms and 205 ms, respectively (when preprocessing material enough for 256 SHA-256 evaluations).

For the single execution setting we see a significant increase in execution time for the dependent preprocessing compared to above. This is due to the design of our prototype which only uses multiple execution threads in the dependent preprocessing and online phases if several, possibly different, circuits are processed at the same time. We also note that our prototype requires a large amount of RAM as we store all garbling material and commitments in-memory. This design choice is due to convenience, but for a deployed system based on the LEGO approach this should be addressed using external memory sources with support for pipelined evaluation as described in Section 4.4.

For the AWS WAN setting we see that a single execution of AES-128 takes around 83 ms online time where almost all of the runtime is spent

Setting	Circuit	Number	BaseOTs	Ind. Preprocessing	Dep. Preprocessing	Online (latency)	Online (throughput)
Local LAN	AES-128	1	1400.89	220.44	15.12	2.52	2.52
		32	44.90	87.63	3.35	2.25	0.35
		128	11.23	70.89	2.95	1.86	0.26
		1024	1.40	61.33	2.85	1.60	0.25
	SHA-256	1	1400.39	1381.05	208.00	22.65	22.65
		32	44.94	812.12	44.59	11.77	3.12
		128	11.22	771.27	37.72	10.43	3.02
AWS LAN	AES-128	1	850.42	89.61	13.23	1.46	1.46
		32	26.61	27.91	0.85	1.23	0.18
		128	6.65	14.85	0.68	1.15	0.09
		1024	0.84	13.84	0.74	1.13	0.08
	SHA-256	1	852.90	478.54	164.40	11.19	11.19
		32	26.82	165.26	14.87	9.14	1.42
		128	6.67	173.05	12.13	9.35	1.09
		256	3.34	183.51	11.70	9.56	1.05
AWS WAN	AES-128	1	2980.25	1881.63	96.66	83.17	83.17
		32	93.75	142.00	5.19	83.21	2.71
		128	23.44	72.31	3.96	83.65	0.73
		1024	2.96	39.18	2.12	83.15	0.62
	SHA-256	1	3043.64	2738.62	350.01	93.94	93.94
		32	92.98	670.98	42.01	92.42	4.04
		128	23.66	431.71	25.44	92.38	1.70
		256	11.75	356.48	27.97	92.74	1.87

Table 4.4: Evaluator timings measured for AES-128 and SHA-256 with $\kappa = 128$ and $s = 40$. All timings are ms per circuit.

waiting due to a latency of ∼81 ms. The latency also severely impacts the two preprocessing phases where the independent preprocessing takes around 1882 ms (20x compared to AWS LAN) and the dependent offline phase takes 96 ms (7x compared to AWS LAN). However this overhead can be somewhat mitigated when considering several circuits, down to a factor 2-4x compared to AWS LAN due to the computation and communication being more interleaved and better utilization of the bandwidth with several TCP connections.

Finally in Table 4.5 we report on the amount of data our prototype transfers from the circuit constructor to the circuit evaluator for both AES-128 and SHA-256. For clarity we have not included the communication from evaluator to constructor, but note that for 1024 AES-128 and 256 SHA-256 a total of 8.12 MB and 4.09 MB are transferred, respectively, and for both cases around 99% of the communication stems from the initial BaseOTs. The table also summarizes the bucketing parameters used in our experiments, which have been chosen so that the probability bound given by Theorem 3 in Section 4.4 is negligible. Also we set the two input bucket parameters $\lambda_g = 2\beta + 1$ and $\lambda_a = 2\alpha + 1$ which ensures a correct majority for all the input buckets and authenticators except with negligible probability. For the data numbers in Table 4.5 it can be seen in parentheses how the relative preprocessing cost of a circuit decreases as more evaluations are considered. We highlight that in this work (and previous LEGO protocols) this is due to the increasing number of *gates* produced, not by the number of circuits. As an example of this effect, going from a single AES-128 with 6928 gates[3] to 1024 AES-128 with 7 094 272 gates decreases the cost of the independent preprocessing by a factor 2.3x per AES-128, from 14.94 MB to 6.42 MB. It is worth noting that the "LEGO effect" only applies to the independent preprocessing. This is because in the subsequent dependent preprocessing step two solderings (κ bits each) are sent per gate of the circuit f and not for each garbled gate produced. In addition a small constant 2.2 kB of decommitment data is transferred in this phase for the s challenge linear combinations. For the online step the communication consists of $n_{\mathsf{A}}\kappa$ bits for the constructors input + $(n + \kappa)(n_{\mathsf{B}} + m)$ bits for the decommitments to the evaluators input and the output decoding bits where n is the code-length of the ECC $\mathcal{C}$ used in the commitment scheme. In Section 4.5 we discussed how this could further be reduced to $(n + \kappa)s + \kappa(n_{\mathsf{B}} + m)$ at the price of adding an extra round to the online phase.

Comparison with Related Work

We compare our measured timings to those reported in the recent works of [LR15] and [RR16], both of which are solely applicable in the amortized setting. In contrast our protocol can naturally handle the single execution

[3]We use the AES non-expanded circuit of [ST] which has 6800 AND gates. However we augment the circuit with identity gates on the 128 output wires in order to simplify output decoding using VerLeak.

Circuit	N	β	p_g	α	p_a	BaseOTs		Ind. Preprocessing		Dep. Preprocessing		Online	
AES-128	1	7	2^{-4}	6	2^{-3}	19.52 kB		14.94 MB		227 kB		16.13 kB	
	32	4	2^{-2}	3	2^{-2}	19.52 kB	(610 B)	279.78 MB	(8.74 MB)	7.26 MB	(227 kB)	516 kB	(16.13 kB)
	128	4	2^{-3}	3	2^{-3}	19.52 kB	(153 B)	924.68 MB	(7.22 MB)	29.04 MB	(227 kB)	2.06 MB	(16.13 kB)
	1024	4	2^{-5}	3	2^{-6}	19.52 kB	(19 B)	6.57 GB	(6.42 MB)	232.31 MB	(227 kB)	16.52 MB	(16.13 kB)
SHA-256	1	5	2^{-3}	4	2^{-4}	19.52 kB		120.34 MB		2.93 MB		22.23 kB	
	32	4	2^{-4}	3	2^{-5}	19.52 kB	(610 B)	2.73 GB	(85.19 MB)	93.63 MB	(2.93 MB)	712 kB	(22.23 kB)
	128	4	2^{-6}	3	2^{-5}	19.52 kB	(153 B)	10.31 GB	(80.54 MB)	374.52 MB	(2.93 MB)	2.85 MB	(22.23 kB)
	256	4	2^{-7}	3	2^{-5}	19.52 kB	(76 B)	20.28 GB	(79.20 MB)	749.03 MB	(2.93 MB)	5.70 MB	(22.23 kB)

Table 4.5: Bucketing parameters and data received by the evaluator in the different phases of our protocol for AES-128 and SHA-256 with $\kappa = 128$ and $s = 40$. Numbers in parentheses are data per circuit produced.

Protocol	Setting	Number	Ind. Preprocessing	Offline	Online
[LR15]	AWS LAN	32	✗	197	12
		128	✗	114	10
		1024	✗	74	7
	AWS WAN	32	✗	1126	163
		128	✗	919	164
		1024	✗	759	160
[RR16]	AWS LAN	32	✗	45	1.7
		128	✗	16	1.5
		1024	✗	5.1	1.3
	AWS WAN	32	✗	282	190
		128	✗	71	191
		1024	✗	34	189
This Work	AWS LAN	32	**54.52**	**0.85**	**1.23**
		128	**21.5**	**0.68**	**1.15**
		1024	**14.68**	**0.74**	**1.13**
	AWS WAN	32	**235.75**	**5.19**	**83.21**
		128	**95.75**	**3.96**	**83.65**
		1024	**42.14**	**2.12**	**83.15**

Table 4.6: Comparison of the reported timings for AES-128 in the AWS LAN and AWS WAN setting with $\kappa = 128$ and $s = 40$. The preprocessing column includes the cost of the BaseOTs for This Work. All timings are ms per AES-128. Best results marked in bold.

setting, along with a more general amortized setting where several *distinct* functions can be preprocessed in the same batch. However to make a meaningful comparison we focus on the "traditional" amortized setting considering 32, 128, and 1024 AES-128 computations and we summarize the comparison in Table 4.6. The independent preprocessing timings for our protocol consists of the BaseOTs + the independent preprocessing. The first thing to notice is that for applications where independent preprocessing is applicable, and can therefore be disregarded, our dependent offline performance is superior to both prior works for any number of AES-128 computations by a large margin. Compared to [LR15] our reported timings are better by 100-358x depending on the setting and number of circuits. For [RR16] the gap is smaller, but still substantial, namely by 6-54x. For applications where independent preprocessing can not be utilized we are still superior to the work of [LR15], but for most settings and number of circuits we cannot compete with the offline

phase of [RR16]. However the differences are typically within a factor 1.2-3x.

For the online timings in the AWS LAN setting for AES-128 we measure faster overall timings than [RR16] for all number of circuits by a tiny margin. As the differences are less than half a millisecond we believe the only reasonable thing to conclude is that the online times are comparable. Though when looking at raw throughput we outperform [RR16] by more than a factor 3x (0.08 ms vs. 0.26 ms). Going beyond Table 4.6 and considering the larger SHA-256 circuit we note that our online phase is not faster than [RR16] (9.35 ms vs. 8.8 ms for 128 circuits). This is again due to the design of our prototype that uses a single execution thread in the online phase per circuit, while [RR16] uses several threads. We therefore see it as an interesting problem for future research to tailor the LEGO online phase to better exploit parallelism for a single circuit.

With regards to online latency in the AWS WAN setting there is however no doubt that our two round online phase outperforms both [LR15] and [RR16]. This is directly related to the previous protocols having more rounds which in a high latency network significantly decreases performance. For comparison [LR15] has a 4 round online phase and [RR16] has 5. One thing to note however is that [RR16] delivers output to both parties in 5 rounds, whereas both our work and [LR15] would need an extra round to support this. Also the implementation of [LR15] is written in a mix of Java and C++ using JNI which definitely adds overhead to the running time. It is however unclear how much of a speedup a native implementation would achieve, but we suspect it would be substantial. The implementation of [RR16] is written solely in C++ and according to the paper also takes full advantage of parallelization.

In Table 4.7 we summarize the required communication for the previously considered protocols and our work for the same setting as Table 4.6. As was the case for the measured timings, when disregarding the cost of the independent preprocessing, our protocol requires significantly less communication in both the offline and online phase compared to the previous works. For the offline phase the communication is 5-12x less than [RR16] and 16-358x less than [LR15]. If the independent preprocessing is included as part of the offline phase our protocol however requires transferring more raw data than the previous two works for any of the considered number of circuits. However this is only so because we are considering multiple copies (32, 128, and 1024) of the same function AES-128. If we instead consider settings with few copies (the larger the better), a single copy, or several different circuits, the amount of data received in the independent + dependent preprocessing phase of our protocol can match or be lower than the dependent offline phase of [RR16], depending on the circuit sizes and number of circuits considered. Also, as [RR16] uses the dual-execution paradigm where both parties send and receive the same amount of data, the above comparison is only meaningful assuming a full-duplex channel which might not always be available. Finally, even if the amount of data received by the evaluator in our protocol is up to $\sim$4x that of

Protocol	Number	Ind. Preprocessing	Offline	Online
[LR15]	32	✗	8.13 MB	312 kB
	128	✗	5.45 MB	238 kB
	1024	✗	3.76 MB	170 kB
[RR16]*	32	✗	3.75 MB	25.76 kB
	128	✗	2.5 MB	21.31 kB
	1024	✗	1.56 MB	16.95 kB
This Work	32	8.74 MB	**226.86 kB**	**16.13 kB**
	128	7.22 MB	**226.86 kB**	**16.13 kB**
	1024	6.42 MB	**226.86 kB**	**16.13 kB**

*Dual-execution, so total offline communication is double the reported numbers.

Table 4.7: Comparison of the data received by the evaluator for different number of executions of AES-128 with $\kappa = 128$ and $s = 40$. All numbers are data per AES-128. Best results marked in bold.

[RR16] in Table 4.7, due to the highly parallelizable nature of our independent preprocessing phase, this does not translate into equivalently lower performance as can be seen in Table 4.6.

For the online phase, our protocol is more data-efficient than the previous works for any of the considered settings. In particular, we require sending 16.13 kB per AES which is around 1.05-1.6x less data than [RR16] and 10.5-19x less than [LR15], depending on the number of executions considered. Furthermore if one is willing to assume a programmable random oracle, as is already the case of both [RR16] and [LR15], our online phase can easily be modified to only sending 6.30 kB (using 3 rounds) or 9.09 kB (using 2 rounds) as explained in Section 4.5.

Finally as mentioned in Section 4.1 and summarized in Table 4.1 the best reported timings for evaluating a single AES-128 is 65 ms in [WMK17]. Based on the reported numbers in their paper we estimate that ∼20 ms of the execution time consists of the initial BaseOTs. We therefore consider the actual cost of their protocol to be around 45 ms and motivate this by observing that a single computation of BaseOTs can be reused for any number of future executions using OT extension. To give as meaningful a comparison as possible we also ran our implementation on the same AWS setup (c4.2x instance) for the single execution AES-128 case measuring 105.7 ms of ind. preprocessing time, 12.07 ms dep. preprocessing time and 1.41 ms online time on a LAN. Therefore our protocol performs around 2.5x times slower than theirs in total time (when also ignoring the cost of our initial BaseOTs). However if ind. preprocessing can be applied, then from the time the circuit is input by the parties, our

protocol takes around 13.48 ms to evaluate, which is around 3.5x faster than [WMK17]. We ran the same experiment in the WAN setting where we evaluate an AES-128 in 1837 ms of ind. preprocessing time, 82.51 ms dep. preprocessing time and 72.63 ms online time. As [WMK17] takes 1513 ms in total, when adjusting for initial BaseOT cost the difference is about a factor 1.5x in favor of the latter. However, when ignoring time for independent preprocessing our protocol can perform around an order of magnitude faster. We believe this difference in factors between LAN and WAN is due to our protocol having fewer rounds (when ignoring our preprocessing) and our implementation fully saturating the network as it sets up multiple parallel TCP connections for maximal bandwidth utilization.

Chapter 5

DUPLO: Unifying Cut-and-Choose for Garbled Circuits

The following chapter is based on the work of [KNR+17] and is therefore identical (except for minor layout modifications) to the current full version available at `https://eprint.iacr.org/2017/344`.

5.1 Introduction

Garbled Circuits (GC) are currently the most common technique for practical two-party secure computation (2PC). GC has advantages of high performance, low round complexity, low latency, and, importantly, relative engineering simplicity. Both the core garbling technique itself and its application in higher level protocols have been the subject of significant improvement. In the semi-honest model, there have been relatively few asymptotic/qualitative improvements since the original protocols of Yao [Yao86] and Goldreich et al. [GMW87]. The more challenging task of providing security in the presence of *malicious* parties has seen more striking improvements, such as reducing the number of garbled circuits needed for cut-and-choose [LP07, LP11, sS11, Lin13], exploring trade-offs between online and offline computation phases [HKK+14, LR14], and exploring slight weakenings of security [MF06, KMRR15, AO12, KM15]. These improvements have brought the malicious security setting to a polished state of affairs, and even small-factor performance improvements are rare.

Cut-and-choose. The focus of this work is to unify two leading approaches for malicious security in GC-based protocols, by viewing them as extreme points on a single continuum. We will find that optimal performance — often significantly better than the state-of-the-art — is generally found somewhere

in the middle of the continuum. We start with reviewing the idea of cut-and-choose (C&C) and the two existing approaches which we generalize.

According to the "Cut-and-Choose Protocol" entry of the Encyclopedia of Cryptography and Security [TJ11], a (non-zero-knowledge) C&C protocol was first mentioned in the protocol of Rabin [Rab77] where this concept was used to convince a party that the other party sent it a specially formed integer n. The expression "cut and choose" was introduced later by Chaum in [BCC88] in analogy to a popular cake-sharing problem: given a cake to be divided among two distrustful players, one of them cuts the cake in two shares, and lets the other one choose.

Whole-circuit C&C. Recall, Yao's basic GC protocol is not secure against a cheating GC generator, who can submit a maliciously garbled circuit. Today, C&C is the standard tool in achieving malicious security in secure computation. At the high level, it proceeds in two phases.

C&C phase. The GC generator generates a number of garbled circuits and sends them to GC evaluator, who chooses a subset of them (say, half) at random to be opened (with the help of the generator) and verifies their correctness.

Evaluation phase. If all opened circuits were constructed correctly, the players proceed to securely evaluate the unopened circuits, and take the majority (or other protocol-prescribed) output.

A statistical analysis shows that the probability of the GC generator violating security (by making the evaluator accept an incorrect output) is exponentially small in the number of circuits n.

Significant progress has been made [Lin13, HKE13, Bra13, LR14, HKK$^+$14] in reducing the concrete value of n needed to achieve a given failure probability. Specifically, if the evaluation phase of the protocol requires a *majority* of unopened circuits to be correct (as in [sS11]), then $\sim 3s$ circuits are required in total for statistical security 2^{-s}. If the evaluation phase merely requires at least *one* unopened circuit to be correct (e.g., [Lin13, Bra13]), then only s circuits are required for the same security. This multiplicative overhead in garbling material due to replication, the **replication factor**, in the above protocols is $3s$ and s, respectively. In the amortized setting where parties perform N independent evaluations of the same circuit, all evaluations can share a common C&C phase where only a small fraction of circuits needs to be opened. Here, the (amortized) replication factor per evaluation is $O(1) + O(s/\log N)$ for statistical security 2^{-s} [LR14, HKK$^+$14]. As an example, for $N = 1024$ and $s = 40$ the amortized replication factor is around 5.

LEGO. The LEGO paradigm (Large Efficient Garbled-circuit Optimization), introduced by Nielsen & Orlandi [NO09], works somewhat differently. First, the generator produces many independent *garbled gates* (e.g., NAND gates). Similarly to the whole-circuit C&C, the evaluator chooses a random subset of these gates to be opened and checked. Now, the evaluator randomly assigns the unopened gates into *buckets.* The garbled gates in each bucket are

carefully combined in a certain way, so that, as long as a majority of gates in each bucket are correct, the bucket as a whole behaves like a *correct* logical garbled NAND gate. These buckets are then assembled into the final garbled circuit, which is finally evaluated.

The extra step in the LEGO paradigm of randomly assigning unopened gates into buckets improves the protocol's asymptotic replication factor. More precisely, if the evaluated function has N gates, then the LEGO protocol has replication factor $2 + O(s/\log N)$ for security 2^{-s} (compared to s or $3s$ for conventional whole-circuit C&C). The main disadvantage of the LEGO approach is that there is a nontrivial cost to connect independently generated gates together ("soldering," in LEGO terminology). Since soldering needs to be performed for each wire of the Boolean circuit, LEGO's asymptotic advantages overtake whole-circuit C&C in performance only for circuits of large size. In Section 5.3 we give more details about the LEGO paradigm.

DUPLO: building garbled circuits from big pieces

We introduce DUPLO (**D**UPLO **U**nifying **P**rocedure for **LEGO**), a new approach for malicious-secure two-party computation.

As discussed above, the two standard approaches for malicious-secure 2PC perform C&C at the level of entire circuits (whether in the single-execution setting or in the multi-execution setting [LR15, RR16]), or at the level of individual gates (LEGO). DUPLO is a single unifying approach that spans the *entire continuum between these extremes.* The DUPLO approach performs C&C at the level of arbitrary garbled subcircuits (which we refer to as *components*). After the C&C phase has completed, the parties can use the resulting garbled components in any number of 2PC executions, of any (possibly different) circuits that can be built from these components.

What is the value in generalizing C&C in this way? In short, the DUPLO approach unlocks a new degree of freedom in optimizing practical secure computation. To understand its role, we first review in more detail the costs associated with the C&C techniques (including LEGO).

The most obvious (and often the most significant) cost is the GC replication factor, discussed above. When evaluating a function consisting of N components (either entire circuits, gates, or generalized components explored in this work), the replication factor is $O(1) + O(s/\log N)$, for desired security 2^{-s}. Clearly, using smaller components improves the replication factor, since N is increased.

The replication factor converges to a lower limit of 2 [ZH17]. As the number of components grows, the benefit of amortization quickly reaches its effective maximum. With practical parameters, there is little improvement to be gained beyond a few million components.

It is when the number of components is "maxed out" that the flexibility of DUPLO starts to have its most pronounced effect. There will be a wide range of different component sizes that all give roughly the same replication factor. Among these choices for component size, it is now best to choose the *largest*, thereby reducing the *cost of soldering*, or connecting the components. This cost is proportional to the number of input/output wires of a component (whole-circuit C&C can be also seen this way, since we have special processing for the inputs and outputs). When a circuit is decomposed into *larger* components, a smaller fraction of wires will cross a boundary between components and therefore require soldering.

In other words, we expect a "sweet spot" for ideal component size, and for computations of realistic size this sweet spot is expected to be between the extremes of gate-level and whole-circuit components. We confirm this analysis by the empirical performance of our prototype implementation. We indeed find such a "sweet spot" between the extremes of component size, as we start considering computations with millions of gates. For these realistic problem sizes, the DUPLO approach improves performance by 4-7x over gate-based and circuit-based C&C. Details are given in Section 4.6.

Is it realistic to express computations in terms of moderately sized components? We note that the C&C components need to garble identical circuits, i.e. be interchangeable in GC evaluation. Indeed, all NAND gates in LEGO and all circuits in whole-circuit C&C are interchangeable in the sense that they are garblings of the same functionality. One may rightly ask whether it is reasonable to expect realistic computations to be naturally decomposable into interchangeable and non-trivial (i.e. not a single-gate or entire-circuit) subcircuits.

We argue that this is indeed a frequent occurrence in standard (insecure) computation. Standard programming language constructs (standard-size arithmetic operations, subroutine calls, loops, etc.) naturally generate identical subcircuits. Given the recent and growing tendency to automate circuit generation and to build 2PC compilers for higher-level languages [MNPS04, HFKV12, ZE15, LWN+15, MGC+16], it is natural to presume that many practical circuits evaluated by 2PC will incorporate many identical components. Specifically, consider the following scenarios:

- Circuits compiled from higher level languages containing multiple calls to the same subroutine (e.g. algebraic calculations), loops, etc. For example, a boolean circuit for matrix multiplication can be expressed in terms of subcircuits for multiplication and addition.
- Two parties know they will perform many secure computations of a CBC-MAC-based construction (e.g., CMAC) using AES as the block cipher, where one party provides the key and the other provides the message to be authenticated. They can use the AES circuit (or a CBC-composition

of several AES circuits) as the main DUPLO component, and use as many components as needed for each evaluation of CMAC. Another example involving AES is to consider the AES round function as the DUPLO component. As this is the same function used internally in AES-128, AES-192 and AES-256 (only the key schedule and number of rounds differ) this preprocessing becomes more independent of the final functionality.

- Two parties agree on a predetermined low-level instruction set, where for each instruction (represented as a circuit), the parties can produce a large number of preprocessed garbled components without knowing *a priori* the final programs/functionalities to be computed securely. This CPU/ALU emulation setting has recently been considered in the context of secure computation of MIPS assembly programs [SHS+15, WGMK16]. The DUPLO approach elegantly and efficiently provides a way to elevate these results to the malicious setting.

In Section 5.7 we investigate several of these scenarios in detail, and compare our performance to that of previous work.

Related work

Maliciously secure 2PC using Yao's garbled circuit technique has seen dramatic improvements in recent years, both algorithmic/theoretical and implementations. Since the first implementation in [LPS08], tremendous effort has been put into improving concrete efficiency [LP07, NO09, PSSW09, LP11, sS11, HEKM11, KsS12, FJN+13, Bra13, FN13, HKE13, Lin13, MR13, sS13, HMsG13, AMPR14, HKK+14, LR14, LR15, RR16, WMK17, NST17, ZH17, KRW17b] yielding current state-of-the-art prototypes able to securely evaluate an AES-128 computation in 6 ms (multi-execution) or 65 ms (single-execution). Multi-execution refers to evaluating the same function several times (either in serial or parallel) on distinctly chosen inputs while the more general single-execution setting treats the computation atomically. In addition, some of these protocols allow for dividing the computation into different phases to utilize preprocessing. In the most general case the computation can be split into three consecutively dependent phases. Following the convention of [NST17] we have:

Function-independent preprocessing depends only on the statistical and computational security parameters s and κ. It typically prepares a given number of gates/components that can be used for later computation.

Function-dependent preprocessing uses the previously computed raw function-independent material and stitches it together to compute the desired function f.

Online/Eval phase lastly depends on the parties inputs to the actual computation and is typically much lighter than the previous two phases.

Of notable interest are the protocols of [RR16] and [WMK17] which represent the current state-of-the-art protocols/prototypes for the multi- and single-execution settings, respectively. Both protocols also support function-*dependent* preprocessing. With regards to constant-round function-*independent* preprocessing the works of [NST17, ZH17, KRW17b] are the most efficient, however at this time only the work of [NST17] provides a public prototype implementation.

In addition to the above garbled circuit approaches another very active and fruitful research area in secure computation is the secret-sharing based protocols [NNOB12, DPSZ12, DKL$^+$13, KSS13, DZ13, DLT14, LOS14, BLN$^+$15, KOS16, DZ16, DNNR17]. These protocols share a common blueprint in that initially the parties secret share their inputs and interactively compute the function in question. This has the advantage of being less bandwidth demanding than the garbled circuit approach, but at the cost of requiring $\mathcal{O}(\mathsf{depth}(f))$ rounds of interaction to securely evaluate f. This approach also has the benefit of usually supporting function-independent preprocessing and allowing for n participating parties rather natively. In contrast, it seems considerably harder adapting the garbled circuit approach to n-parties [BMR90, LPSY15, LSS16].

The idea of connecting distinct garbled circuits has also previously been studied in [MGBF14] by mapping previous output garbled values to garbled input values in a following computation. Their model and approach is different from ours and is mainly motivated by enabling garbled state to be reusable for multiple computations. Finally we point out the recent work of [GLMY16] for the *semi-honest* case of secure 2PC using garbled circuits. [GLMY16] likewise considers splitting the function of interest into sub-circuits and processes these independently. As there is no cut-and-choose overhead in the semi-honest setting, their approach is motivated primarily by allowing function-independent preprocessing using the garbled components as building blocks. Although the high-level idea is similar to ours, we apply it in a completely different setting and use different techniques. Further, while malicious security is often significantly more expensive, the efficiency gap in the linking and online phase between [GLMY16] and our protocol is surprisingly small. In the application of computing an AES-128 (by preprocessing the required round functions) we see that [GLMY16] sends 82 kB in the online phase (link + evaluate) vs. 88 kB using our protocol. For the offline step the gap is larger due to the overhead of C&C in the malicious case. However utilizing amortization this can be reduced significantly and in some cases be as low as 3-5x that of the semi-honest protocols'. We also highlight that our extension to the Frigate compiler for transforming a high-level C-style program into Boolean circuit sub-components should be directly applicable for this related work. To the best of our knowledge [GLMY16] does not provide such a tool.

Our Contributions and Outline of the Work

The main contribution of the paper is putting forward and technically and experimentally supporting the idea of generalizing C&C protocols to arbitrary subcircuits. Due to the generality of the approach and the performance benefits we demonstrate, we believe the DUPLO approach will be the standard technique in 2PC compilers. As a lower-level technical contribution, we propose several improvements to garbling and soldering for this setting.

We implemented our solution and integrated it with the state-of-the-art compiler framework Frigate [MGC+16]. Experimentally, we report of a 4-7x improvement in total running time compared to [WMK17] for certain circuits. For the multi-execution setting we also improve the performance of [RR16] by up to 5× in total running time. We accomplish the above while at the same time retaining the desirable preprocessing and reactive capabilities of LEGO.

We start our presentation with a more technical overview of the state of the art in LEGO, including soldering techniques in Section 5.3. We then present the technical overview of our approach and improvements in Section 5.4. We present the overview of our DUPLO framework, including several implementation optimizations and the Frigate extensions in Section 5.6. We report on performance in Section 5.7.

5.2 Preliminaries

Our DUPLO protocol is a protocol for 2PC that is secure in the presence of malicious adversaries. We define security for 2PC using the framework of Universal Composition (UC), due to Canetti [Can01]. This framework is demanding, as it guarantees security when such protocols are executed concurrently, in arbitrary environments like the Internet.

A detailed treatment of UC security is beyond the scope of this work. At the high level, security is defined in the real-ideal paradigm. We imagine an ideal interaction, in which parties give their inputs to a trusted third party who computes the desired function f and announces the result. In this interaction, the only thing a malicious party can do is select its input to f. In the real interaction, honest parties interact following the prescribed protocol, while malicious parties may arbitrarily deviate from the protocol. We say that the protocol *securely realizes* f if the real world is "as secure as" the ideal world. More formally, for every adversary attacking the real protocol, there is an adversary (called "simulator") "attacking" the ideal interaction achieving the same effect.

We assume some familiarity with modern garbled circuit constructions, in particular, the *Free-XOR* optimization of Kolesnikov & Schneider [KS08]. This is reviewed in Section 5.3. Free-XOR garbled circuits are secure under a circular correlation-robust hash assumption [CKKZ12].

5.3 Overview of the LEGO Paradigm

We now give more details about the mechanics of the LEGO paradigm. Here we describe the MiniLEGO approach of [FJN+13]. We chose MiniLEGO as it is the simplest LEGO protocol to present. At the same time, it contains and conveys all relevant aspects of the paradigm.

Soldering via XOR-Homomorphic Commitments

The sender generates many individual garbled NAND gates. Each garbled gate g is associated with *wire labels* L_g^0, L_g^1 for the left input wire, labels R_g^0, R_g^1 for the right input wire, and labels O_g^0, O_g^1 for the output wire. Here the superscript of each label indicates the truth value that it represents. In MiniLEGO, all gates are garbled using the Free-XOR optimization of Kolesnikov & Schneider [KS08]. Therefore, there is a global (secret) value Δ so that $L_g^1 = L_g^0 \oplus \Delta$ and $R_g^1 = R_g^0 \oplus \Delta$ and $O_g^1 = O_g^0 \oplus \Delta$. More generally, a wire label K_g^b can be written as $K_g^b = K_g^0 \oplus b \cdot \Delta$. Importantly, the same Δ is used for *all* garbled gates.

The garbled gate consists of the garbled table itself (i.e., for a single NAND gate, the garbled table consists of two ciphertexts when using the scheme of [ZRE15]) along with **XOR-homomorphic commitments** to the "zero" wire labels L_g^0, R_g^0, and O_g^0. A global homomorphic commitment to Δ is also generated and shared among all gates.

To assemble assorted garbled gates into a circuit, the LEGO paradigm uses a technique called **soldering**. Imagine two wires (attached to two unrelated garbled gates) whose zero-keys are A^0 and B^0, respectively. The sender can "solder" these wires together by decommiting to $S = A^0 \oplus B^0$. We require that such a decommitment can be performed given separate commitments to A^0 and B^0, and that the decommitment reveals no more than S. Importantly, S is enough information to allow the receiver to transfer a garbled truth value from the first wire to the second (and vice-versa). For example, if the receiver holds wire label A^b (for unknown b), he can compute

$$A^b \oplus S = (A^0 \oplus b \cdot \Delta) \oplus S = B^0 \oplus b \cdot \Delta = B^b,$$

which is the garbled encoding of the same truth value, but on the other wire.

Gates are assigned to buckets by the receiver, where each bucket, while possibly containing malicious gates, will be assembled to correctly implement the NAND gate. For the gates inside a bucket, the sender therefore solders all their left wires together, all their right wires together, and all their output wires together with the effect that the bucket can operate on a single set of input labels and produce a single set of output labels. For β gates in a bucket, this gives β ways to evaluate the first gate (use solder values to transfer its garbled inputs to the ith bucket gate, evaluate it, then transfer the result back to the first gate). In the most basic form of LEGO, the cut-and-choose ensures

that the majority of gates within the bucket are good. Hence the evaluator can evaluate the bucket in β ways and take the majority output wire label. Each bucket therefore logically behaves like a *correct* garbled gate.

The buckets are then assembled into a complete garbled circuit by soldering output wires of one bucket to the input wires of another.

Recent LEGO Improvements

In recent years several improvements to the LEGO approach has been proposed in the literature. The TinyLEGO protocol [FJNT15] provide several concrete optimizations to the above MiniLEGO protocol, most notably a more efficient bucketing technique. The subsequent implementation [NST17] further optimized the protocol and showed that, combined with the XOR-homomorphic commitment scheme of [FJNT16, CDD$^+$16], the LEGO paradigm is competitive with previous state-of-the-art protocols for malicious 2PC, in particular in scenarios where preprocessing is applicable.

In addition to the above works, the protocol of [ZH17] also explores optimizations of LEGO using a different soldering primitive, dubbed XOR-Homomorphic Interactive Hash (XOR-HIH). This technique has a number of advantages over commitments as they allow for a better probability than MiniLEGO and TinyLEGO of catching cheating in the C&C phase. XOR-HIH also yields buckets only requiring a single "correct" gate, whereas MiniLEGO requires a majority and TinyLEGO requires a mixed majority of gates and wire authenticator gadgets. However, due to the communication complexity of the proposed XOR-HIH instantiation being larger than that of the [FJNT16, CDD$^+$16] commitment schemes, the overall communication complexity of [ZH17] is currently larger than that of TinyLEGO.

5.4 Overview of Our Construction

DUPLO protocol big picture. At the high level, our idea is to extend the LEGO paradigm to support components of arbitrary size and distinct functionalities, rather than just a single kind of component that is either a single gate or the entire circuit. The approach is similar in many ways to the LEGO protocol and is broken up into three phases.

In the *function-independent phase*, the garbler generates many independent garblings of each kind of component, along with related commitments required for soldering. For each kind of component, the parties perform a cut-and-choose over all garbled components. The receiver asks the garbler to open some fraction of these components, which are checked for correctness. The remaining components are assembled randomly into *buckets.* The soldering required to connect components into a bucket is done at this step.

In the *function-dependent phase*, the parties agree on circuits that can be assembled from the available components. The parties perform soldering that connects different buckets together, forming the desired circuits.

In the *online phase*, the parties have chosen their inputs for an evaluation of one of the assembled circuits. They perform oblivious transfers for the evaluator to receive its garbled input, and the garbler also releases its own garbled inputs. The evaluator then evaluates the DUPLO garbled circuit and receives the result.

Challenges and New Techniques. The seemingly simple high-level idea described above encounters several significant technical challenges in its realization. We address the issues in detail in Section 5.5. Here we mention that the main challenge is that the LEGO paradigm uses the same Free-XOR offset Δ for all garbled components, and its soldering technique crucially relies on this fact. This is not problematic when components are single gates, but turns out to lead to scalability issues for larger components. As a result, we must change the fundamental garbling procedure, and therefore change the soldering approach.

The TinyLEGO approach uses an *input recovery* technique inspired by [Lin13]. The idea is that if the garbler cheats in some components, then the resulting garbled circuit will either give the correct garbled output, or else it will leak the garbler's entire input! In the latter case, the evaluator can simply evaluate the function in the clear. As above, the TinyLEGO approach to this input recovery technique relies subtly on the fact that the components are small, and as a result it does not scale for large components. We introduce an elegant new technique that works for components of any size, and improves the concrete cost of the input recovery mechanism.

Implementation, Evaluation, Integration. We implemented a high-performance prototype of our protocol to explore the effect of varying component sizes in the C&C paradigm. We study a variety of scenarios and parameter choices and find that our generalizations of C&C can lead to significant performance improvement. Details are given in Section 4.6.

We have adapted the Frigate circuit compiler of Mood et al. [MGC+16], which compiles a variant of C into circuits suitable for garbled circuit 2PC applications. We modified Frigate so that subroutines are treated as DUPLO components. As an example, a CBC-MAC algorithm that makes calls to an AES subroutine will be compiled into an "outer circuit" built from atomic AES components, as well as an "inner circuit" that implements the AES component from boolean gates. In our implementation, the inner circuits are then garbled as DUPLO components, and the outer circuits are used to assemble the components into high-level functionalities.

5.5 DUPLO Protocol Details

We now give more details about the challenges in generalizing the LEGO paradigm, and our techniques to overcome them.

Different Δ's

The most efficient garbling schemes use the Free-XOR optimization of [KS08]. MiniLEGO/TinyLEGO are compatible with Free-XOR, and in fact they enforce that all garbled gates use the same global Free-XOR difference Δ. However, having a common Δ does lead to some drawbacks. In particular, consider the part of the cut-and-choose step in which the receiver chooses some garbled gates to be opened/checked. If we fully open a garbled gate, both wire labels are revealed for each wire. In MiniLEGO, this would reveal Δ and compromise the security of the *unopened* gates, which share the same Δ. To avoid this, the MiniLEGO approach is to make the sender reveal only *one out of the four* possible input combinations to each opened gate (by homomorphically decommitting to the input wire labels). Note that the receiver may now have only a 1/4 probability of detecting an incorrectly garbled gate (the technique of [ZH17] improves this probability to 1/2). The cut-and-choose analysis must account for this probability.

This approach of only partially opening garbled gates does not scale well for large components. If a component has n input wires, then the receiver will detect bad components with probability $1/2^n$ in the worst case. In the DUPLO protocol, we garble each component c with a separate Free-XOR offset Δ_c (so each gate inside the garbled component uses Δ_c, but other garbled components use different offset). Hence, DUPLO components can be *fully opened* in the cut-and-choose phase, while XOR gates are still free inside each component.

As a result:

- Bad components are detected with probability 1, so the statistical analysis for DUPLO cut-and-choose is better than Mini/TinyLEGO by a constant factor.
- We can use a variant of the optimization suggested in [GMS08] to save bandwidth for cut-and-choose. Initially the sender only sends a short hash of each garbled component. Then to open a component, the sender decommits to the input and output keys as well as the Δ_c used for garbling the component. Hence, communication for the opened components is minimal.

Adapting soldering. It remains to describe how to adapt the soldering procedure to solder wires with different Free-XOR offsets (the MiniLEGO approach relies on the offsets being the same). Here we adapt a technique of [AHMR15] for soldering wires. Using the point-and-permute technique for

garbled circuits [BMR90], the two wire labels for each wire have random and opposite least-significant bits. We refer to this bit as the *color bit* for a wire label. The evaluator sees the color bit of a wire, but not the truth value of a wire.

In MiniLEGO, the garbler commits to the "zero-key" for each wire, which is the wire label encoding *truth value false.* In DUPLO, we have the garbler generate homomorphic commitments to the following:

- For each wire, commit to the wire label *with color bit zero.* In this section we therefore use notation K^b to denote a wire label with *color bit* (not necessarily truth value) b.
- For each wire, commit to an indicator bit σ for each wire that denotes the color bit of the *false* wire label. Hence, wire label K^b has truth value $b \oplus \sigma$.
- For each component c, commit to its Free-XOR offset Δ_c.

Consider a wire i with labels $(K_i^0, K_i^1 = K_i^0 \oplus \Delta_i)$ and indicator bit σ_i, and another wire j in a different component with labels $(K_j^0, K_j^1 = K_j^0 \oplus \Delta_j)$ and indicator bit σ_j. To solder these wires together, the garbler will give homomorphic decommitments to the following solder values:

$$s^\sigma = \sigma_i \oplus \sigma_j; \quad S^K = K_i^0 \oplus K_j^0 \oplus s^\sigma \cdot \Delta_j; \quad S^\Delta = \Delta_i \oplus \Delta_j$$

Note that the decommitment to S^Δ can be reused for all wires soldered between these two components. Now when the evaluator learns wire label K_i^b (with color bit b visible), he can compute:

$$\begin{aligned} K_i^b \oplus S^K \oplus b \cdot S^\Delta &= K_i^b \oplus (K_i^0 \oplus K_j^0 \oplus s^\sigma \cdot \Delta_j) \oplus b \cdot (\Delta_i \oplus \Delta_j) \\ &= b \cdot \Delta_i \oplus (K_j^0 \oplus s^\sigma \cdot \Delta_j) \oplus b \cdot \Delta_i \oplus b \cdot \Delta_j \\ &= K_j^0 \oplus (s^\sigma \oplus b) \cdot \Delta_j = K_j^{s^\sigma \oplus b} \end{aligned}$$

Also note that a common truth value has opposite color bits on wires i & j if and only if $s^\sigma = \sigma_i \oplus \sigma_j = 1$. Hence, the receiver obtains the wire label $K_j^{s^\sigma \oplus b}$ which encodes the same truth value as K_i^b.

DUPLO bucketing. In Section 5.3 we described how [FJN+13] used a bucket size that guaranteed a majority of correct AND gates in each bucket. In this work we use the original bucketing technique of [NO09] that only requires a single correct component in each bucket, but requires a majority bucket of wire authenticator (WA) gadgets on each output wire. The purpose of a WA is to accept or reject a wire label as "valid" without revealing the semantic value on the wire, and as such a simple construction can be based on a hash function and C&C. A WA consists of a "soldering point" (homomorphic commitments to a Δ and a zero-key), along with an unordered

pair $\{\mathcal{H}(K_i^0), \mathcal{H}(K_i^0 \oplus \Delta)\}$. A wire label K can be authenticated checking for membership $\mathcal{H}(K) \in \{\mathcal{H}(K_i^0), \mathcal{H}(K_i^0 \oplus \Delta)\}$. In order to defeat cheating a C&C step is carried out on the WAs to ensure that a majority of any WA bucket only accepts committed wire labels. The choice of using WAs in this work is motivated by the fact that DUPLO components can be of arbitrary size and are often much larger than a single gate. By requiring fewer such components in total, we therefore achieve much better overall performance as WAs are significantly cheaper to produce in comparison to garbled components.

Avoiding commitments to single bits. We also point out that the separate commitments to the zero-label K_i^0 and the indicator bit σ_i can be combined into a single commitment. The main idea is that the least significant bit of K_i^0 is always zero (being the wire label with color bit zero). Similarly, when using Free-XOR, the offset Δ must always have least significant bit 1. Hence in the solder values S and S^Δ, the evaluator knows *a priori* what the least significant bit will be. We can instead use the least significant bits of the K_i^0 commitments to store the indicator bit σ_i so that homomorphic openings convey $\sigma_i \oplus \sigma_j$. This approach saves s bits of communication per wire commitment over the naive approach of instantiating the bit-commitments using [FJNT16] using a bit-repetition code with length s.

In the online evaluation phase, the garbler decommits to the indicator bits of the evaluators designated input and output. In this case, the garbler does not want to decommit the entire wire label as this would potentially let the evaluator learn the global difference Δ (if the evaluator learned the opposite label through the OTs or evaluation). To avoid this, we have the garbler generate many commitments to values of the special form $R\|0$ for random $R \in \{0,1\}^{\kappa-1}$. Using the homomorphic properties of these commitments, this can be done efficiently by having the garbler decommit s random linear combinations of these commitments to ensure that *all* of them have the desired form with probability $1 - 2^{-s}$. Then when the garbler wants to decommit to a wire label's indicator bit only, it gives a homomorphic decommitment to the wire label XOR a mask $R\|0$, which hides everything but the indicator bit.

Improved Techniques for Circuit Inputs

We also present a new, more efficient technique for input recovery. The idea of input recovery [Lin13] is that if the sender in a 2PC protocol cheats, the receiver will learn the sender's input (and can hence compute the function output).

Within each DUPLO bucket, the cut-and-choose guarantees at least one correctly garbled component and a majority of correct output-wire authenticators. As such, the evaluator is guaranteed to learn, for each output wire of a component, either 1 or 2 *valid* garbled outputs. If only one garbled output is obtained, then it is guaranteed to be the correct one. Otherwise, the receiver

learns both wire labels and hence the Free-XOR offset Δ_c for that component. The receiver can then use the solder values to iteratively learn *both* wire labels on *all* wires in the circuit (at least all the wires in the connected component in which the sender cheated).

However, knowing both wire labels does not necessarily guarantee that the receiver learns their corresponding *truth values*. We need a mechanism so that the receiver learns the truth value for the sender's garbled inputs.

Our approach is to consider special input-components. These consist of an *empty garbled circuit* but homomorphic commitments to a zero-wire-label K and a Free-XOR offset Δ that serve as soldering points. Suppose for every input to the circuit, we use such an input component that is soldered to other components. The sender gives his initial garbled input by homomorphically decommitting to either the zero wire-label K or $K \oplus \Delta$. If the sender cheats within the computation, the receiver will learn Δ. The key novelty in our approach is to use **self-authenticating wire labels.** In an input-gadget, the false wire label must be $H(\Delta)$ and the true wire label must be $H(\Delta) \oplus \Delta$ (the sender will still commit to whichever has color bit zero). Then when the sender cheats, the receiver learns Δ, and can determine whether the sender initially opened $H(\Delta)$ (false) or $H(\Delta) \oplus \Delta$ (true).

This special form of wire labels can be checked in the cut-and-choose for input components. In the final circuit, we assemble input-components into buckets to guarantee that a majority within each bucket is correct. Then the receiver can extract a cheating sender's input according to the majority of input-components in a bucket.

Formal Description, Security

Our protocol implements secure reactive two-party computation [NR16], *i.e.*, the computation has several rounds of secret inputs and secret outputs, and future inputs and as well as the specification of future computations might depend on previous outputs.

To be more precise, let $\mathcal{F}$ denote the ideal functionality $\mathcal{F}_{\mathsf{R2PC}}^{\mathbb{L},\Phi}$ in Fig. 9 on page 1040 in [NR16]. Recall that this functionality allows to specify a reactive computation by dynamically specifying the functionality of sub-circuits and how they are linked together. The command $(\textsc{Func}, t, f)$ specifies that the sub-circuit identified by t has circuit f. The command $(\textsc{Input}, t, i, x)$ gives input x to wire i on sub-circuit t. Only one party supplies x, the other party inputs $(\textsc{Input}, t, i, ?)$ to instruct $\mathcal{F}$ that the other party is allowed to give an input to the specified wire. The command defines the wire to have value x. The command $(\textsc{Link}, t_1, i_1, t_2, i_2)$ specifies that output wire i_1 of sub-circuit t_1 should be soldered on input wire i_2 of sub-circuit t_2. When an output value becomes defined to some x, this in turn defines the linked input wire to also have value x. The command $(\textsc{Garble}, t, f)$ evaluates the sub-circuit t. It assumes that all the input wires have already been defined. It runs f on these

values and defines the output wires to the outputs of f. There are also output commands that allow to output the value of a wire to a given party. They may be called only on wires that had their value defined.

The set $\mathbb{L}$ allows to restrict the set of legal sequences of calls to the functionality. We need the restriction that all (FUNC, t, f) commands are given before any other command. This allows us to compute how many times each f is used and do our preprocessing. The function Φ allows to specify how much information about the inputs and outputs of $\mathcal{F}$ is allowed to leak to the adversary. We need the standard setting that we leak the entire sequence of inputs and outputs to the adversary, except that when an honest party has input (INPUT, t, i, x), then we only leak $(\text{INPUT}, t, i, ?)$ and when an honest party has output (OUTPUT, t, i, y), then we only leak $(\text{OUTPUT}, t, i, ?)$.

With many components, many buckets, and many 2PC executions, the formal description of our protocol is rather involved. It is therefore deferred to Section 5.8 while we in Section 5.9 prove the following theorem.

Theorem 4. Our protocol implements $\mathcal{F}$ in the UC framework against a static, poly-time adversary.

5.6 System Framework

In this section we give an overview of the DUPLO framework and our extension to the Frigate compiler that allows to transform a high-level C-style program into a set of boolean circuit components that can be fed to the DUPLO system for secure computation. We base our protocol on the recent TinyLEGO protocol [FJNT15], but adapted for supporting larger and distinct components. Our protocol has the the following high-level interface:

Setup A one-time setup phase that initializes the XOR-homomorphic commitment protocol.

PreprocessComponent(n, f) produces n garbled representations F_j of f that can be securely evaluated.

PrepareComponents(i) produces i input authenticators that can be used to securely transfer input keys from garbler G to evaluator E. In addition, for all F_j previously constructed using PreprocessComponent, this call constructs and attaches all required output authenticators. These gadgets ensure that only a single valid key will flow on each wire of all garbled components (otherwise the evaluator learns the generator's private input).

Build(**C**) Takes a program C as input, represented as a DAG where nodes consist of the input/output wires of a set of (possibly distinct) components $\{f_i\}$ and edges consist of links from output wires to input wires for all of these f_i's. The Build call then looks up all previously constructed F_j for each f_i and stitches these together using the XOR-homomorphic commitments so that they together securely compute the computation

specified by C. This call also precomputes the required oblivious transfers (OTs) for transferring E's input securely.

Evaluate(x, y) Given the plaintext input x of garbler G and y of evaluator E, the parties can now compute a garbled output Z, representing the output of the $f(x, y)$. The system allows both parties to learn the full output, but also distinct output, *e.g.* G can learn the first half of $f(x, y)$ and E learn the second half.

Decode Finally the system allows the parties to decode their designated output. The reason why we have a dedicated decode procedure is to allow partial output decoding. Based on the decoded values the parties can then start a new secure computation on the remaining non-decoded output, potentially adding fresh input as well. The input provided and the new functionality to compute can thus depend on the partially decoded output. This essentially allows branching within the secure computation.

Following the terminology introduced in [NST17] we have that the **Setup**, PreprocessComponent, and PrepareComponents calls can be done independently of the final functionality C. These procedures can therefore be used for function-independent preprocessing by restricting the functionality C to be expressible from a predetermined set of instructions. The **Build** procedure clearly depends on C, but not on the inputs of the final computation, so this phase can implement function-dependent preprocessing. Finally the **Evaluate** and Decode procedures implement the online phase of the system and depend on the previous two phases to run.

For a detailed pseudocode description of the system as well as a proof of its security we refer the reader to Section 5.8 and Section 5.9, respectively.

Implementation optimizations

As part of our work we developed a prototype implementation in C++ using the latest advances in secure computation engineering.[1] As the basis for our protocol we start from the libOTe library for efficient oblivious transfer extension [Rin]. As we in this work require UC XOR-homomorphic commitments to the input and output wires of all components we instantiate our protocol with the efficient construction of [FJNT16] and use the implementation of [RT17] in our prototype.

As already mentioned, our protocol is described in detail in Section 5.8. However, for reasons related to efficiency our actual software implementation deviates from the high-level description in several aspects

- In the homomorphic commitment scheme of [FJNT16], commitments to random values (chosen by the protocol itself) are cheaper than com-

[1] Available at `https://github.com/AarhusCrypto/DUPLO`

mitments to values chosen by the sender. Hence, whenever applicable we let the committed key-values be defined in this way. This optimization saves a significant amount of communication since the majority of commitments are to random values.
- Along the same lines we heavily utilize the batch-opening mechanism described in [FJNT16]. The optimization allows a sender to decommit to n values with total bandwidth $n\kappa + \mathcal{O}(s)$ as opposed to the naive approach which requires $\mathcal{O}(n\kappa s)$.
- In the PrepareComponents step we construct all output-wire key authenticators using a single global difference Δ_{ka}. This saves a factor $2x$ in terms of the required number of commitments and solderings, at the cost of an incorrect authenticator only getting caught with probability $1/2$ (as opposed to prob. 1 using distinct differences). However as the number of required key authenticators depends on the total number of output wires of all garbled components the effect of this difference in catching probability does not affect performance significantly when considerings many components.

In addition to the above optimizations, our implementation takes full advantage of modern processors' multi-core capabilities and instruction sets. We also highlight that our code leaves a substantially lighter memory footprint than the implementation of [NST17] which stores all garbled circuits and commitments in RAM. In addition to bringing down the required number of commitments on the protocol level, our implementation also makes use of disk storage in-between batches of preprocessed component types. This has the downside of requiring disk reads of the garbled components during the online phase, but we advocate that the added flexibility and possibility of streaming preprocessing is well worth this trade-off in performance.

Frigate Extension

The introduction of Fairplay [MNPS04], the first compiler targeted for secure computation (SC), has stimulated significant interest from the research community. Since then, a series of new compilers with enhanced performance and functionality have been proposed, such as CBMC [HFKV12], Obliv-C [ZE15], and ObliVM [LWN+15]. Importantly, the state-of-the-art compiler, Frigate [MGC+16], features a modular and extensible design that simplifies the circuit generation in secure computation. Relying on its rich language features, we provide an extension to the original Frigate framework, in which we divide the **specific** input program into distinct functions. We can then generate a circuit representation for each function which is fully independent from the circuit representation of other functions. Due to this independence we can easily garble each distinct function separately using the DUPLO framework and afterwards solder these back together such that they compute the original

source program. As an additional improvement, which is tangential to the main thrust of this work, we construct an AES module that optimizes the number of uneven gates (all even gates can be garbled and evaluated without communication using *e.g.* [ZRE15]).

In the following, we describe the details of our compiler extension. Similar to the Frigate output format, our circuit output contains a set of input and output calls, gate operations, and function calls. The input and output calls consist of wires, which we enumerate and manage. We also use wires to represent declared variables in the source program. Each wire (or, rather its numeric id) is placed in a pool, and is ready for use whenever a new variable is introduced. Our function representation however differs from that of Frigate. In that work, each function reserves a specific set of wire values which requires no overlap among the functions' wires. As a result, Frigate's function representation is dependent on that of other functions. We remove this dependency by creating and managing separate wire pools for each function. In particular, every time a variable is introduced, our compiler searches for the free wires with the smallest indices in the pool of the current working function. Similarly to the original Frigate, our compiler will free the wires it can after each operation or variable assignment. Hence, our function is represented independently of other functions.

We now describe our strategy for constructing our optimized AES circuit. A key component of AES is the Rijndael S-Box [DR02] which is a fixed non-linear substitution table used in the byte substitution transformation and the key expansion routine. The circuit optimization in our AES-128 source program is described in the context of this S-Box. We note that if we generate the S-Box dynamically using the Frigate compiler, this will not optimize the number of uneven gates substantially. Hence, we create an AES-128 source program that embed a highly optimized S-Box circuit statically. To the best of our knowledge, [BP09] presents one of the most efficient S-Box circuit representation which contains only 32 uneven gates in a total of 115 gates. Therefore, we integrate this S-Box into our AES-128 source program, which allows our Frigate extension to optimize the number of uneven gates. For the *key-expanded* AES-128 circuit, which takes a 128-bit plaintext and ten 128-bit round keys as input and outputs a 128-bit ciphertext, this results in 5,120 uneven gates. This is almost a 2x reduction compared the AES-128 circuit originally reported in Frigate. Furthermore, our AES-128 circuit has 640 fewer uneven gates than the circuit reported in TinyGarble [SHS+15] which is the current best compiler written in Verilog. For completeness we note that for the *non-expanded* version of AES-128, our compiled circuit results in 6,400 uneven gates.

5.7 Performance

In order to evaluate the performance of our prototype we run a number of experiments on a single server with simulated network bandwidth and latency. The server has two 36-core Intel(R) Xeon(R) E5-2699 v3 2.30 GHz CPUs and 256 GB of RAM. That is, 36 cores and 128 GB of RAM per party. As both parties are run on the same host machine we simulate a LAN and WAN connection using the Linux *tc* command: a LAN setting with 0.02 ms round-trip latency, 1 Gbps network bandwidth; a WAN setting with 96 ms round-trip latency, 200 Mbps network bandwidth.

For both settings, the code was compiled using GCC-5.4. Throughout this section, we performed experiments with a statistical security parameter $s = 40$ and computational security parameter $\kappa = 128$. The running times recorded are an average over 10 trials.

We demonstrate the scalability of our implementation by evaluating the following circuits:

AES-128 circuit consisting of 6,400 AND gates. The circuit takes a 128-bit key from one party and a 128-bit block from another party and outputs the 128-bit ciphertext to both. (Note that this functionality is somewhat artificial for secure computation as the AES function allows decryption with the same key; thus the player holding the AES key can obtain the plaintext block. We chose to include the ciphertext output to the keyholder to measure and demonstrate the performance for the case where both parties receive output.)

CBC-MAC circuit with different number of blocks $m \in \{16, 32, 64, 128, 256, 1024\}$ using AES-128 as the block cipher. The circuit therefore consists of $6,400m$ AND gates. The circuit takes a 128-bit key from one party and m 128-bit blocks from another party and outputs a 128-bit block to both.

Mat-Mul circuit consisting of around 4.2 million AND gates. The circuit takes one 16×16 matrix of 32-bit integers from each party as input and outputs the 16×16 matrix product to both.

Random circuit consisting of 2^n AND gates for various n where topology of the circuit is chosen at random. The circuit takes 128-bit input from each party and outputs a 128-bit value to both.

Effect of Decomposition

In this section we show how DUPLO scales for the above-mentioned circuits, when considering subcomponents of varying size. As discussed in Section 5.1, we expect the performance of our protocol to be optimal for a subcomponent size somewhere inbetween the extremes of whole-circuit and gate-level C&C. We empirically validate this hypothesis by running two kinds of experiments, one for the randomly generated circuits and one for the real-world AES-128,

CBC-MAC-16 and Mat-Mul circuits. The purpose of the random circuit experiment is to explore the trade-offs in overall performance between different decomposition strategies. For the latter experiment we aim to find their optimal decomposition strategy, both to see how this aligns to the random circuit experiment, but also for use in our later performance comparison in Section 5.7.

Random Circuits. In order to build a random circuit consisting of 2^n AND gates that is easily divisible into different subcomponent sizes we initially generate a number of smaller random circuit containing 2^t AND gates with 256 input wires and 128 output wires. This is done by randomly generating non-connected XOR and AND until exactly 2^t AND gates have been generated. Then for each of these generated gates i we assign their two input wires at random from the set of gates with index smaller than i (the gate id i is also the gate's output wire). Finally we solder 2^{n-t} copies of these components together into a final circuit C, thus consisting of 2^n AND gates overall. We consider $n \in \{10, 12, 14, 16, 18, 19, 20\}$ in this experiment, and for each of these we build a circuit of size 2^n using several values of t.

As we are only considering relative performance between different strategies in these experiments we run our implementation using a single thread for each party on the previously mentioned LAN setup.[2] We summarize our findings in Figure 5.1. The x-axis of the figure represents the continuum from whole-circuit C&C ($t = n$) towards gate-level C&C ($t = 0$). The overall trend of our experiments is strikingly clear, initially as the number of subcomponents increases (t decreases) the running time goes down as well due to our protocol taking advantage of the amortization benefits offered by the LEGO paradigm. However for all circuit sizes considered it is also apparent that at some point this benefit is outweighed by the overhead of soldering and committing to the increasing number of input/output wires between the components. It is at exactly this point (the vertex of each graph), in the sweet spot between substantial LEGO amortization and low soldering overhead, that DUPLO has it's optimal performance. We thus conclude that for an ideally decomposable circuit such as the ones generated in this experiment the viability of the DUPLO approach is apparent.

Real-world circuits. The experiments for the random circuits show that the DUPLO approach for C&C does have merit for circuits that can be divided into multiple identical subcomponents. Clearly, this is a very narrow class of functions so in addition we also evaluate our prototype on the previously mentioned real-world circuits in order to investigate their optimal decompo-

[2]For best absolute performance, we would always run our implementation using several threads per party.

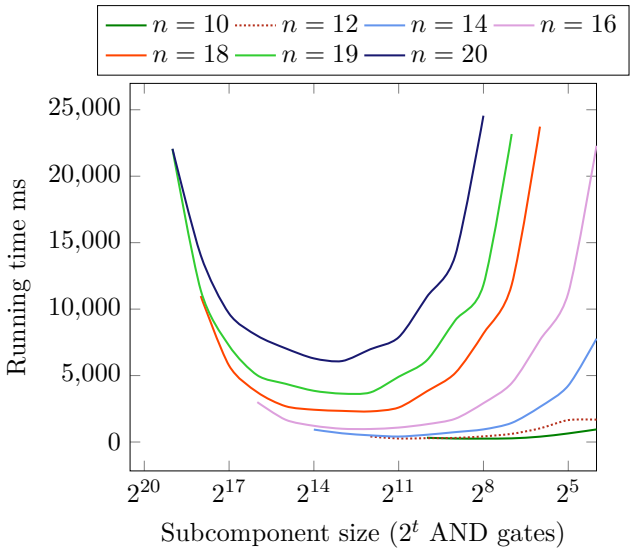

Figure 5.1: DUPLO performance for random circuits consisting of 2^n AND gates divided into 2^{n-t} subcomponents.

sition strategy. We first describe our approach of dividing these circuits into subcomponents.

AES-128 We consider the following three strategies:

- Five kinds of subcomponents: each computing one of the functions of the AES algorithm, that is 1x Key Expansions (1,280 AND gates), 11x AddRoundKey, 10x SubBytes (512 AND gates), 10x ShiftRows, and 9x MixColumns.
- Three kinds of subcomponents: 1x Key Expansions and Initial Round (1,280 AND gates); 9x AES Round Functions (each 512 AND gates); 1x AES Final Round (512 AND gates).
- A single component consisting of the entire AES-128 circuit (6,400 AND gates), *i.e.* whole-circuit C&C.

CBC-MAC-16 We consider decomposing this circuit into a single subcomponent of varying size. In each case, the component contains $i \in \{16, 8, 4, 2, 1\}$ AES-128 blocks, meaning each of these consists of $6,400i$ AND gates.

Mat-Mul In order to multiply two matrices A, B use the block-matrix algorithm: We divide A, B into $m \times m$ 32-bit submatrices $A_{i,j}, B_{i,j}$ for $i, j \in [1, 16/m]$. To compute AB, the block entries $A_{i,k}$ are first multiplied by the block entries $B_{k,j}$ for $k \in [1, m]$, while summing the results over k. It is therefore the case that the experiment contains two different kinds of components, $m \times m$ 32-bit matrix product and $m \times m$

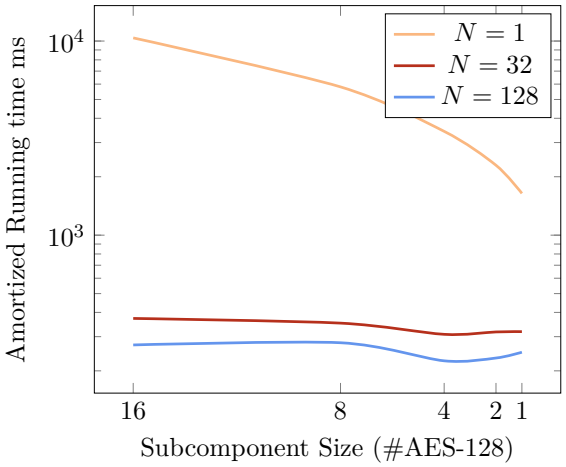

Figure 5.2: DUPLO performance for $N = 1, 32, 128$ parallel executions of the CBC-MAC-16 circuit using different decomposition strategies.

32-bit matrix addition. In our experiment we consider block matrix sizes $m \in \{16, 8, 4, 2\}$ and the concrete number of AND gates for each kind of component are reported in Table 5.1.

When performing $N = 1, 32, 128$ executions of AES-128 in parallel, we observe that our protocol performs best when considering the entire circuit as a single component. This is in contrast to what we observed in the random circuit experiment, but can be explained by the non-uniformity of the considered decomposition strategies. The fact that we split the AES-128 into three or five relatively small subcomponents, some of which are only used once, has a very negative influence on DUPLO performance as there is some overhead associated with preparing each component type while at the same time no LEGO-style amortization can be exploited when preparing only a single copy.

For the CBC-MAC-16 circuit however whole-circuit C&C is not the optimal approach and we summarize the observed performance for the different decompositions in Figure 5.2. Here we see that the best strategy is to decompose the circuit into many *identical* subcomponents. The trend observed is similar to the random circuit experiments where initially it is best to optimize for many identical subcomponents. In particular for a single execution of CBC-MAC-16 it is best to decompose into 16 copies of the AES-128 circuit yielding around 5x performance increase over the whole-circuit approach. For the parallel executions (which contain overall many more AES-128 circuits) we can see that it is best to consider subcomponents consisting of 4xAES-128 circuits each. The lower relative performance difference between the strategies for the parallel executions is due to there being a minimum of N circuits for utilizing LEGO amortization, even for the whole-circuit approach. However as the

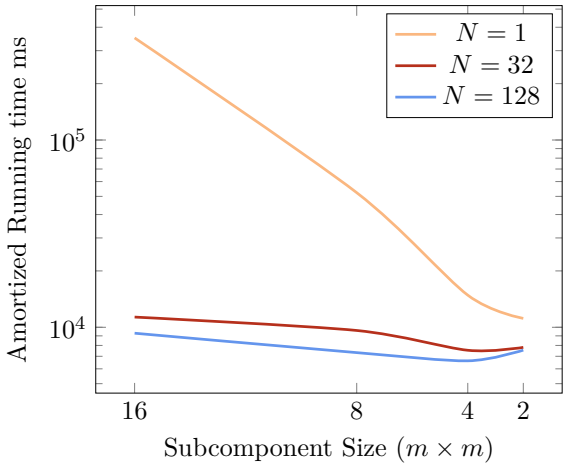

Figure 5.3: DUPLO performance for $N = 1, 32, 128$ parallel executions of the Mat-Mul circuit using different decomposition strategies.

number of total subcomponents grow it can be seen that there are savings to be had by grouping executions together.

Block Size	Component Size		Number Executions N		
	Mult	Add	1	32	128
2x2	8,192	124	**11,160**	7,815	7,554
4x4	65,536	496	14,847	**7,539**	**6,622**
8x8	524,288	1,984	52,334	9,615	7,324
16x16	4,194,304	0	351,002	11,338	9298

Table 5.1: Component sizes and amortized running time per execution for Mat-Mul (ms). Best performance marked in bold.

Finally for the Mat-Mul circuit we see a similar overall trend as in the CBC-MAC-16 experiment (Figure 5.3 and Table 5.1). Most notably is the performance increase for a single execution yielding around 31x by considering blocks of size 2×2 instead of a single whole-circuit 16×16. This experiment indeed highlights the performance potential of the DUPLO approach for large computations that can naturally be decomposed into distinct repeating subcomponents, in this case matrix product and matrix addition. This is in contrast to the previous AES-128 example where this approach was penalized. The difference however is that in the Mat-Mul experiment each subcomponent is repeated several times and therefore all benefit from LEGO amortization.

Experiment Discussion. The above real-world examples show that the DUPLO approach has merit, but the exact performance gains depend signifi-

cantly on the circuit in question. As a general rule of thumb DUPLO performs best when the circuit can be decomposed into many identical subcomponents as can be seen from the CBC-MAC-16 and Mat-Mul experiments (the more the better). As there is no immediate way of decomposing the AES-128 circuit in this way, we see that performance suffers when the circuit cannot be decomposed into distinct *repeating* parts. However the Mat-Mul experiments show that decomposing the circuit into distinct circuits can certainly have merit, however it is crucial that each subcomponent is repeated a minimal number of times or the non-repeating part of the computation is relatively small.[3]

Comparison with Related Work

We also compared our prototype to three related high-performance open-source implementations of malicious-secure 2PC. All experiments use the same hardware configuration described at the beginning of this section. For all experiments we have tried tuning the calling parameters of each implementation to obtain the best performance.

When reporting performance of our DUPLO protocol, we split the offline part of the computation into an independent preprocessing (**Setup** + PreprocessComponent + PrepareComponents) whenever our analysis shows that dividing the computation into subcomponents is optimal — *i.e.*, when evaluating AES-128 we do not have any function-independent preprocessing since the optimal configuration is to let the component consist of the entire circuit. We summarize our measured timings for all the different protocols in Table 5.2, and now go into more detail:

Better Amortization by Subdivision. The protocol of Rindal & Rosulek (RR in our tables) [RR16] is currently the fastest malicious-secure 2PC protocol in the multi-execution setting. The protocol of Nielsen et al. (NST) [NST17] is the fastest that allow for function-independent preprocessing, using the LEGO paradigm.[4]

The general trend in Table 5.2 is that as the total complexity (combined cost of all computations) grows, the efficiency of the DUPLO approach becomes more and more apparent. For example, DUPLO is 1.5x times faster (counting total offline+online time) than RR whole-circuit C&C for 1024 AES-128 LAN. For the larger CBC-MAC-16 scenario, the difference 2.5x. For the even larger case of 32 Mat-Mul executions, the difference is 5x. Our experiments clearly confirm that DUPLO scales significantly better than state-of-the-art amortizing protocols.

[3] This is not the case for the AES-128 circuit as the non-repeating part consists of around 40% of the entire computation.

[4] The recent 2PC protocol of [KRW17b] appears to surpass NST in terms of performance in this setting, but as this implementation is not publicly available at the time of writing we do not consider it for these experiments.

Protocol	Setting	N	AES			CBC-MAC-16			Mat-Mul		
			Ind. Prep	Offline	Online	Ind. Prep	Offline	Online	Ind. Prep	Offline	Online
WMK	LAN	1	✗	(✗)	125	✗	(✗)	1,177	✗	(✗)	43,930
	WAN	1	✗	(✗)	2,112	✗	(✗)	11,443	✗	(✗)	368,190
RR	LAN	1	✗	198	8.83	✗	3,495	35.52	✗	120,200	913
		32	✗	67.77	3.60	✗	1,296	20.71	✗	36,437	1,247
		128	✗	40.70	2.86	✗	863	18.38	-	-	-
		1024	✗	24.90	3.06	✗	471	17.48	-	-	-
	WAN	1	✗	941	527	✗	12039	565	✗	467,711	1,550
		32	✗	311	472	✗	4202	471	✗	157,928	1,677
		128	✗	192	557	✗	2743	573	✗	-	-
		1024	✗	115	577	✗	1762	597	✗	-	-
NST	LAN	1	**1,506**	**22.34**	**2.54**	2,594	230	**18.14**	-	-	-
		32	**119**	**2.42**	**0.22**	965	38.63	**1.34**	-	-	-
		128	**75.64**	**2.08**	**0.16**	922	37.90	**0.87**	-	-	-
		1024	**60.48**	**1.85**	**0.14**	-	-	-	-	-	-
	WAN	1	**9,325**	**223**	**195**	13,812	699	219	-	-	-
		32	**599**	**15.17**	**6.71**	4,158	151	8.54	-	-	-
		128	**341**	**12.75**	6.24	3,810	148	7.15	-	-	-
		1024	**256**	**11.81**	5.56	-	-	-	-	-	-
DUPLO	LAN	1	✗	371	8.62	**799**	**29.83**	41.94	**10,268**	**569**	**118**
		32	✗	47.03	0.65	**303**	**10.18**	3.72	**7,124**	**331**	**83.73**
		128	✗	27.77	0.41	**213**	**11.54**	2.49	**6,260**	**303**	**58.65**
		1024	✗	17.58	0.30	**175**	**13.30**	**1.61**	-	-	-
	WAN	1	✗	7,391	585	**8,970**	1,370	620	**50,856**	**1,775**	**744**
		32	✗	347	19.37	**1,477**	**49.21**	23.62	**32,098**	**517**	**135**
		128	✗	148	**5.55**	**990**	**22.23**	8.85	**27,613**	**388**	**101**
		1024	✗	74.03	**1.53**	**733**	**15.93**	**3.83**	-	-	-

Table 5.2: All timings are ms per circuit. Best results are marked in bold. Cells with "✗" denote setting not supported. Cells with "-" denote program out of memory.

When comparing to the LEGO C&C protocol of NST things are harder to compare as they use a much slower BaseOT implementation than we do (1200 ms vs. 200 ms) which especially matters for lower complexity computations. However even when accounting for this difference, in total time, our approach has 2-3x better total performance for AES-128. We note that if Ind. Prep. is applicable for an application then DUPLO cannot compete with NST for small computations, but as demonstrated from our CBC-MAC-16 experiments, once the computation reaches a certain size and we can decompose the target circuit into smaller subcomponents, DUPLO overtakes NST in performance by a factor 5x.

It is interesting to note that the online time of NST is vastly superior to RR and DUPLO, especially for small circuits (2-4x). This is due to the difference between whole-circuit C&C and gate-level C&C where the NST bucket size is relatively small (and thus online computation) even for a single circuit, whereas it needs to be 5-10x larger for the whole-circuit approach. As the number of executions increase we however see that this gap decreases significantly. We believe the reason why NST still outperforms DUPLO in all online running times is that the NST implementation is RAM only, whereas DUPLO writes components and solderings to disk in the offline phases and reads them in the online phase as needed. For RR we notice some anomalies for their online times that we cannot fully explain. We conclude that the throughput measured and reported in our experiments might not be completely fair towards the RR protocol, but might be explained by implementation decisions that work poorly for our particular scenarios. In any case, we do expect DUPLO to perform as fast or faster than RR in the online phase due to less online rounds and data transfer.

Amortized-grade Performance for Single-Execution. The current fastest protocol for single-execution 2PC is due to Wang et al. (WMK) [WMK17]. When comparing to their protocol, we ran all experiments using the "everything online" version of their code since this typically gives the best overall running time. We stress however that the protocol also supports a function-dependent preprocessing phase, but since this is not the primary goal of that work we omit it here.

Unsurprisingly, the protocols designed for the multi-execution settings (including DUPLO) are significantly faster than WMK when considering several executions. However, even in the single-execution setting, we see that DUPLO scales better and eventually catches up to the performance of WMK for large computations. WMK is 3x faster than DUPLO when the subcomponent is an entire AES-128 circuit. Then, already for CBC-MAC-16 the ability to decompose this into 16 independent AES-128 circuits yields around 1.4x factor improvement over WMK. We further explore this comparison in Table 5.3, by evaluating even larger circuits in the single-execution setting. For larger CBC-MAC circuits, DUPLO is around 4.7x faster on LAN and 7.4x on WAN.

Protocol	Setting	CBC-MAC-32		CBC-MAC-64		CBC-MAC-128		CBC-MAC-256		CBC-MAC-1024	
		Offline	Online	Offline	Online	Offline	Online	Offline	Online	Offline	Online
WMK	LAN	(✗)	2,298	(✗)	4,539	(✗)	9,029	(✗)	18,003	(✗)	71,787
DUPLO		**1,211**	**68.29**	**1,877**	**104**	**2,991**	**196**	**5,072**	**274**	**14,167**	**1,003**
WMK	WAN	(✗)	21,460	(✗)	41,114	(✗)	79,157	(✗)	155,995	(✗)	606,329
DUPLO		**12,269**	**656**	**15,039**	**698**	**19,089**	**793**	**27,093**	**910**	**81,883**	**1,733**

Table 5.3: Comparison for CBC-MAC-XX. All timings are ms per circuit. Best results are marked in bold. "(✗)" denotes the setting is supported, but we only ran the "everything online" version of WMK.

Protocol	#Execs	Ind. Prep	Dep. Prep	Online
WMK	1	✗	✗	9.66 MB
RR	32	✗	3.75 MB	25.76 kB
	128	✗	2.5 MB	21.31 kB
	1024	✗	1.56 MB	16.95 kB
NST	1	14.94 MB	**226.86 kB**	16.13 kB
	32	8.74 MB	**226.86 kB**	16.13 kB
	128	7.22 MB	**226.86 kB**	16.13 kB
	1024	6.42 MB	**226.86 kB**	16.13 kB
WRK	1	**2.86 MB**	570 kB	**4.86 kB**
	32	**2.64 MB**	570 kB	**4.86 kB**
	128	**2.0 MB**	570 kB	**4.86 kB**
	1024	**2.0 MB**	570 kB	**4.86 kB**
DUPLO	1	✗	12.94 MB	19.36 kB
	32	✗	2.60 MB	18.97 kB
	128	✗	1.96 MB	18.96 kB
	1024	✗	1.59 MB	18.96 kB

Table 5.4: Comparison of the data sent from constructor to evaluator AES-128 with $\kappa = 128$ and $s = 40$. All numbers are per AES-128. Best results marked in bold.

Bandwidth Comparison

As a final comparison we also consider the bandwidth requirements of the different protocols. In addition to the previous three protocols we here also include the recent work of Wang et al. (WRK) [KRW17b]. To directly compare we report on the data required to transfer from constructor to receiver in Table 5.4 for different number of AES-128 executions. We stress that these numbers are all from the same AES-128 circuit [ST] and not from our optimized Frigate version. As already established for AES-128, DUPLO performs best by treating the entire circuit as a single component, hence we do not distinguish between Ind. Prep and Dep. Prep in the table. However we do stress that DUPLO only requires solderings from the input-wires to the output-wires of potentially large components, so for applicable settings we expect the Dep. Prep of DUPLO to be much lower than that of NST and WRK as they require solderings for each gate. It can be seen that for a single AES-128 component DUPLO cannot compare with the protocol of WRK in terms of overall bandwidth. This is natural as the replication factor is much lower for gate-level C&C in this case. However as the number of circuits grows we see that DUPLO's bandwidth requirement decreases significantly per AES-128 to a

point where it is actually better than WRK by a factor 1.6x at 1024 executions. For the online phase it is clear that WRK's bandwidth is better than our protocol as we require decommitting the garbled input keys for the evaluator which induces some overhead. However we note that our implementation is not optimal in terms of online bandwidth in that we have chosen flexibility over minimizing rounds and bandwidth. For a dedicated application DUPLO's online bandwidth can be reduced by around 2x by combining the evaluate and decode phases and running batch-decommit of the evaluator input wires along with the output indicator bits.

5.8 Protocol Details

We describe and analyse the protocol in the UC framework. We will here give an abstract description that lends itself to a security analysis. In Section 5.6 we describe some of the optimisations that were done in the implementation and why they do not affect the security analysis. We describe the protocol for two parties, the garbler G and the evaluator E. We will describe the protocol in the hybrid model with ideal functionalities $\mathcal{F}_{\mathsf{HCOM}}$ and $\mathcal{F}_{\mathsf{OT}}$ for xor-homomorphic commitment and one-out-of-two oblivious transfer. The precise description of the ideal functionalities are standard by now and can be found in [FJNT16] and [KOS15]. Here we will denote the use of the functionalities by some pseudo-code conventions. When using $\mathcal{F}_{\mathsf{HCOM}}$ it is G that is the committer and E that is the receiver. When G executes $\mathbf{Commit}(\mathtt{cid}, x)$ for $\mathtt{cid} \in \{0,1\}^*$ and $x \in \{0,1\}^\kappa$, then $\mathcal{F}_{\mathsf{HCOM}}$ stores $(\mathtt{cid}, x_{\mathtt{cid}})$ (where $x_{\mathtt{cid}} = x$) and outputs $\mathtt{cid}$ to E. When G executes $\mathbf{Open}(\mathtt{cid}_1, \ldots, \mathtt{cid}_c)$, where each $\mathtt{cid}_i$ was output to E at some point, then $\mathcal{F}_{\mathsf{HCOM}}$ outputs $(\mathtt{cid}_1, \ldots, \mathtt{cid}_c, \oplus_{i=1}^{c} x_{\mathtt{cid}_i})$ to E. When G executes an open command, then the commitment identifies $(\mathtt{cid}_1, \ldots, \mathtt{cid}_c)$ are always already known by E. If $\mathcal{F}_{\mathsf{HCOM}}$ outputs$(\mathtt{cid}'_1, \ldots, \mathtt{cid}'_c, \oplus_{i=1}^{c} x_{\mathtt{cid}_i})$ where some $\mathtt{cid}'_i \neq \mathtt{cid}_i$ then E always tacitly aborts the protocol. Similarly the $\mathtt{cid}$ used in the commit command is always known and E aborts if G uses a wrong one. When using $\mathcal{F}_{\mathsf{OT}}$ it is G that is the sender and E that is the receiver. We assume that we have access to a special OT which has a special internal state $\Delta \in \{0,1\}^\kappa$, which is chosen by G once and for all at the initialisation of the ideal functionality by executing OTInit(Δ). After that, when G executes OTsend$(\mathtt{id}, x_0)$ for $\mathtt{id} \in \{0,1\}^*$ and $x_0 \in \{0,1\}^\kappa$ and E executes OTreceive$(\mathtt{id}, b)$ for $b \in \{0,1\}$, then $\mathcal{F}_{\mathsf{OT}}$ outputs $(\mathtt{id}, x_b)$ to E, where $x_1 = x_0 \oplus \Delta$. If the protocol specifies that G is to execute OTreceive$(\mathtt{id}, b)$ and it does not or uses a wrong $\mathtt{id}$, then E will always detect this and will tacitly abort.

When we instruct a party to send a value, we tacitly assume the receiver stores it under the same name when it is received.

When we instruct a party to check a condition, we mean that the party will abort if the condition is false.

When a variable like $K_{\mathtt{id}}$ is created in our pseudo-code, it can be accessed by another routine at the same party using the same identifier. Sometimes we use the **store** and **retrieve** key-words to explicitly do this. To save on notation, it will sometimes be done more implicitly, when it cannot lead to ambiguity. In general, if an uninitialised variable like $K_{\mathtt{id}}$ is used in a protocol, then there is an implicit "**retrieve** $K_{\mathtt{id}}$" in the line before.

We assume that we have a free-xor garbling scheme $(\mathsf{Gb}, \mathsf{Ev})$ which has correctness, obliviousness and authenticity. We recall these notions now. The key length is some κ. The input to Gb is a poly-sized circuit C computing a function $C : \{0,1\}^n \to \{0,1\}^m$ along with $(K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$, where $\mathsf{lsb}(\Delta) = 1$. The output is $(L_1^0, \ldots, L_m^0) \in (\{0,1\}^\kappa)^m$ and a garbled circuit F. Here F is the garbled version of C. Define $K_i^1 = K_i^0 \oplus \Delta$. For $x \in \{0,1\}^n$ define $K^x = (K_1^{x_1}, \ldots, K_n^{x_n})$. This is the garbled input, *i.e.*, the garbled version of x. Define $L_i^1 = L_i^0 \oplus \Delta$. For $y \in \{0,1\}^m$ define $L^y = (L_1^{y_1}, \ldots, L_m^{y_m})$. This is the garbled output. The input to Ev is a garbled circuit F and a garbled input $(K_1, \ldots, K_n) \in (\{0,1\}^\kappa)^n$. The output is $\perp$ or a garbled output $(L_1, \ldots, L_m) \in (\{0,1\}^\kappa)^m$. The scheme being free-xor means the inputs and outputs are of the above form. Correctness says that if you do garbled evaluation, you get the correct output. Obliviousness says that if you are given F but not given $(K_1^0, \ldots, K_n^0, \Delta)$, then the garbled input leaks no information on the plaintext input (or output). Authenticity says that if you are given only a garbled circuit for C and a garbled input for x, then you cannot compute the garbled output for any other value than the correct value $C(x)$. These notions have been formalized in [BHR12b]. Here we recall them in the detail we need here and specialized to free-xor garbling schemes.

correctness $\forall x \in \{0,1\}^n$ and $\forall (K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$ with $\mathsf{lsb}(\Delta) = 1$ it holds for $(L_1^0, \ldots, L_m^0, F) = \mathsf{Gb}(K_1^0, \ldots, K_n^0, \Delta)$ that $\mathsf{Ev}(F, K^x) = L^{C(x)}$.

obliviousness For uniformly random $(K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$ with $\mathsf{lsb}(\Delta) = 1$ and $(\cdot, F) \leftarrow \mathsf{Gb}(K_1^0, \ldots, K_n^0, \Delta)$ and any $x_0, x_1 \in \{0,1\}^n$ it holds that (F, K^{x_0}) and (F, K^{x_1}) are computationally indistinguishable.

authenticity Let $\mathcal{A}$ be a probabilistic poly-time interactive Turing machine. Run $\mathcal{A}$ to get a circuit $C : \{0,1\}^n \to \{0,1\}^m$ and an input $x \in \{0,1\}^n$. Sample uniformly random $(K_1^0, \ldots, K_n^0, \Delta) \in (\{0,1\}^\kappa)^{n+1}$ with $\mathsf{lsb}(\Delta) = 1$ and $((L_1^0, \ldots, L_m^0), F) \leftarrow \mathsf{Gb}(K_1^0, \ldots, K_n^0, \Delta)$ and input (F, K^x) to $\mathcal{A}$. Let $y = C(x)$. Run $\mathcal{A}$ to get $(L_1, \ldots, L_n) \in (\{0,1\}^\kappa)^m$ and $y' \in \{0,1\}^m$. If $y' \neq y$ and $L = L^{y'}$, then $\mathcal{A}$ wins. Otherwise $\mathcal{A}$ loses. We require that the probability that any PPT $\mathcal{A}$ wins is negligible.

We will in fact require extended versions of these notions as we use a reactive garbling scheme in the sense of [NR16]. In a reactive garbling scheme one can make several independent garblings and then later solder an output wire $\mathtt{id}$ with keys $(K_{\mathtt{id}}^0, K_{\mathtt{id}}^1)$ onto an input wire $\mathtt{id}'$ with keys $(K_{\mathtt{id}'}^0, K_{\mathtt{id}'}^1)$ in another

circuit. This involves releasing some information to the evaluator which allows the evaluator later to compute $K^b_{\mathtt{id}'}$ from $K^b_{\mathtt{id}}$ for either $b = 0$ or $b = 1$. The notion of reactive garbling scheme is given in [NR16]. We will use the reactive garbling scheme from [AHMR15]. We will later describe how to solder in [AHMR15] and we then recall the notion of reactive garbling scheme from [NR16] to the detail that we need in our proofs.

We finally assume that we have access to a programmable random oracle $H : \{0,1\}^\kappa \to \{0,1\}^\kappa$. Note that this in particular implies that H is collision resistant.

We assume that we are to securely compute one circuit C which consist of sub-circuits C and solderings between input wires and output wires of these sub-circuits. We call the position in C in which a sub-circuit C is sitting a *slot* and each slot is identified by some identifier $\mathtt{id}$. There is a public mapping from identifiers $\mathtt{id}$ to the corresponding sub-circuit C. If $C : \{0,1\}^n \to \{0,1\}^m$, then the inputs wires and output wires of the slot are identified by $\mathtt{id}.\mathtt{in}.1, \ldots, \mathtt{id}.\mathtt{in}.n$ and $\mathtt{id}.\mathtt{out}.1, \ldots, \mathtt{id}.\mathtt{out}.m$. Sub-circuits sitting at a slot are called functional sub-circuits. There are also some special sub-circuits:

- E in-gates, with $n = 0$ and $m = 1$. These are for letting E input a bit. The output wire is identified by $\mathtt{id}.\mathtt{out}.1$.
- G in-gates, also with $n = 0$ and $m = 1$. These are for letting G input a bit. The output wire is identified by $\mathtt{id}.\mathtt{out}.1$.
- E out-gates, with $n = 1$ and $m = 0$. These are the output gates of E. The input wire is identified by $\mathtt{id}.\mathtt{in}.1$.
- G out-gates, with $n = 1$ and $m = 0$. These are the output gates of G. The input wire is identified by $\mathtt{id}.\mathtt{in}.1$.

Besides a set of named sub-circuits, the circuit C also contains a set S of solderings $(\mathtt{id}_1, \mathtt{id}_2)$, where $\mathtt{id}_1$ is the name of an output wire of a sub-circuit and $\mathtt{id}_2$ is the name of an input wire of a sub-circuit. We require that all input wires of all sub-circuits are soldered to exactly one output wire of a sub-circuit and that there are no loops. This ensures we can plaintext evaluate the circuit as follows. For each in-gate $\mathtt{id}$ assign a bit $x_{\mathtt{id}}$ and say that $\mathtt{id}.\mathtt{out}.1$ was evaluated. Now iteratively: 1) for each soldering $(\mathtt{id}_1, \mathtt{id}_2)$ where $\mathtt{id}_1$ was evaluated, let $x_{\mathtt{id}_2} = x_{\mathtt{id}_1}$ and say $\mathtt{id}_2$ was evaluated, and 2) for each sub-circuit where all input wires were evaluated, run C on the corresponding bits, assign the result to the output wires and say they are evaluated. This way all out-gates will be assigned a unique bit. The goal of our protocol is to let both parties learn their own output bits without learning any other information. We assume some given *evaluation order* of the sub-circuits that allows to plaintext evaluate in that order.

We assume that we have two functions $L^1, \alpha^1 : \mathbb{N} \to \mathbb{N}$ for setting the parameters of the cut-and-choose. Consider the following game parametrised

by $n \in \mathbb{N}$. First the adversary picks $L = L^1(n)$ balls. Let $\alpha = \alpha^1(n)$. Some of them are green and some are red. The adversary picks the colours. Then we sample uniformly at random $L - \alpha n$ of the balls. If any of the sampled balls are red, the adversary immediately loses the game. Then we uniformly at random throw the remaining αn balls into n buckets of size α. The adversary wins if there is a bucket with only red balls. We assume that L^1 and α^1 have been fixed such that the probability that any adversary wins the game is 2^{-s}, where s is the security parameter. Note that the functions depend on s, but we ignore this dependence in the notation. We assume that we have two other functions $L^2, \alpha^2 : \mathbb{N} \to \mathbb{N}$. We consider a game similar to the above, but where the adversary wins if all sampled balls are green and there is a bucket with a majority of red balls. We assume that L^2 and α^2 have been fixed such that the probability that any adversary wins the game is 2^{-s}.

Overview of Notation

- $\mathtt{id}$: generic identifier, just a bit-string naming an object.
- $\Delta_{\mathtt{id}}$: the *difference* with identifier $\mathtt{id}$. Defined to be the value in the commitment with identifier $\mathtt{id}.\mathtt{dif}$. It should hold that $\mathsf{lsb}(\Delta_{\mathtt{id}}) = 1$.
- $\sigma_{\mathtt{id}}$: the *indicator* bit with identifier $\mathtt{id}$. Defined to be the value in the commitment with identifier $\mathtt{id}.\mathtt{ind}$.
- $K_{\mathtt{id}}^{\sigma_{\mathtt{id}}}$: *base-key* with identifier $\mathtt{id}$. Defined to be the value in the commitment with identifier $\mathtt{id}.\mathtt{base}$. It should hold that $\mathsf{lsb}(K_{\mathtt{id}}^{\sigma_{\mathtt{id}}}) = 0$.
- $K_{\mathtt{id}}^{1-\sigma_{\mathtt{id}}}$: *esab-key* with identifier $\mathtt{id}$. Defined to be $K_{\mathtt{id}}^{\sigma_{\mathtt{id}}} \oplus \Delta_{\mathtt{id}}$.
- $K_{\mathtt{id}}^0$: 0-*key* with identifier $\mathtt{id}$.
- $K_{\mathtt{id}}^1$: 1-*key* with identifier $\mathtt{id}$.
- $K_{\mathtt{id}}$: key held by E. It should hold that $K_{\mathtt{id}} \in \{K_{\mathtt{id}}^0, K_{\mathtt{id}}^1\}$.
- $L^1(n)$: total number of objects to create when one component should be good per bucket and n buckets are needed.
- $\alpha^1(n)$: bucket size when one component should be good per bucket and n buckets are needed.
- $L^2(n)$: as above, but majority in each bucket is good.
- $\alpha^2(n)$: as above, but majority in each bucket is good.
- Par($\mathtt{id}$): mapping from input wire $\mathtt{id}$ to the unique parent output wire. This is well-defined given the soldering set S.
- $\mathtt{rco}$: A special wire index used in recovering inputs of a corrupted G.

Main Structure

The main structure of the protocol will be as follows.

function Main(C)
  PreProcessKA()
  PreProcessInKA()
  PreProcessOTInit()

PreProcessSub()
AssembleSubs()
AttachInKAs()
for all sub-circuit id in evaluation order **do**
if id is a G in-gate **then** InputG(id)
else if id is an E in-gate **then** InputE(id)
else if id is a G in-gate **then** OutputG(id)
else if id is a E out-gate **then** OutputE(id)
else EvSubs(id)
end if
end for
end function

We assume that E knows an input bit $x_{\mathtt{id}}$ for each E in-gate id before it is evaluated and that G knows an input bit $x_{\mathtt{id}}$ for each G in-gate id before it is evaluated. The inputs are allowed to depend adaptively on previous outputs. At the end of the protocol E knows an output bit $y_{\mathtt{id}}$ for each E out-gate id and G knows an output bit $y_{\mathtt{id}}$ for each G out-gate id.

During the pre-processing G will commit to key material for all wires. The keys $K^0_{\mathtt{id}}$ and $K^1_{\mathtt{id}}$ will be well defined from these committed values, even if G is corrupt. We then implement the input protocols and the evaluation protocols such that it is guaranteed that for each wire, E will learn $K_{\mathtt{id}} \in \{K^0_{\mathtt{id}}, K^1_{\mathtt{id}}\}$.

We then implement the input protocols such that it is guaranteed that for each G-input gate id the evaluator will learn some $K_{\mathtt{id}} \in \{K^0_{\mathtt{id}}, K^1_{\mathtt{id}}\}$. This holds even if G or E is corrupted. If G is honest, it is guaranteed that $K_{\mathtt{id}} = K^{x_{\mathtt{id}}}_{\mathtt{id}}$. If G is corrupted, then $x_{\mathtt{id}}$ is defined by $K_{\mathtt{id}} = K^{x_{\mathtt{id}}}_{\mathtt{id}}$. Furthermore, for each E-input gate E will learn $K_{\mathtt{id}} \in \{K^0_{\mathtt{id}}, K^1_{\mathtt{id}}\}$. This holds even if G or E is corrupted. If E is honest, it is guaranteed that $K_{\mathtt{id}} = K^{x_{\mathtt{id}}}_{\mathtt{id}}$. If E is corrupted, then $x_{\mathtt{id}}$ is defined by $K_{\mathtt{id}} = K^{x_{\mathtt{id}}}_{\mathtt{id}}$. This ensures that after the input protocols, an input bit $x_{\mathtt{id}}$ is defined for each input wire, called the plaintext value of the wire. This allows us to mentally do a plaintext evaluation of the circuits, which gives us a plaintext bit for each output wire and input wire of all components. We denote the bit defined for wire id by $x_{\mathtt{id}}$. We call this the correct plaintext value of the wire. Note that this value might be known to neither E nor G. However, by security of the input protocols E will learn the correct key $K^{x_{\mathtt{id}}}_{\mathtt{id}}$ for in-gates. We then implement the evaluation protocol such that E iteratively will also learn the correct keys $K^{x_{\mathtt{id}}}_{\mathtt{id}}$ for all internal wires id. For the G-output wires, the evaluator E will just send $K^{x_{\mathtt{id}}}_{\mathtt{id}}$ to G who knows $(K^0_{\mathtt{id}}, K^1_{\mathtt{id}})$ and can decode to $x_{\mathtt{id}}$. By authenticity E cannot force an incorrect output. For the E-output wires, the evaluator E will be given the indicator bit for the keys which will allow to compute exactly $x_{\mathtt{id}}$ from $K^{x_{\mathtt{id}}}_{\mathtt{id}}$. That the evaluator learns nothing else will follow from obliviousness of the garbling scheme.

We first describe some small sub-protocols and then later stitch them together to the sub-protocol used above. During the presentation of the sub-

protocols we will argue correctness of the protocols, *i.e.*, that they compute the correct keys $K^{x_{\mathtt{id}}}_{\mathtt{id}}$. In the following section we will then give a more detailed security analysis of the protocol.

Key Authenticators and Input

The following protocol is used to assign key material to an identifier $\mathtt{id}$.

function GenWire($\mathtt{id}, K, \Delta$) ▷ Require: $\mathsf{lsb}(\Delta) = 1$
  G: $K^0_{\mathtt{id}} \leftarrow K$ ▷ the 0-key
  G: $\Delta_{\mathtt{id}} \leftarrow \Delta$ ▷ the difference
  G: $K^1_{\mathtt{id}} \leftarrow K^0_{\mathtt{id}} \oplus \Delta_{\mathtt{id}}$ ▷ the 1-key
  G: $\sigma_{\mathtt{id}} \leftarrow \mathsf{lsb}(K^0_{\mathtt{id}})$ ▷ the indicator bit
  G: **Commit**($\mathtt{id.dif}, \Delta_{\mathtt{id}}$)
  G: **Commit**($\mathtt{id.ind}, \sigma_{\mathtt{id}}$)
  G: **Commit**($\mathtt{id.base}, K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}$)
end function

The key $K^b_{\mathtt{id}}$ will be used to represent the plaintext value b. From $\mathsf{lsb}(\Delta) = 1$ it follows that $\mathsf{lsb}(K^0_{\mathtt{id}}) = \mathsf{lsb}(K^1_{\mathtt{id}}) \oplus 1$. We set $\sigma_{\mathtt{id}} = \mathsf{lsb}(K^0_{\mathtt{id}})$. So, if $\mathsf{lsb}(K^0_{\mathtt{id}}) = 0$, then $\sigma_{\mathtt{id}} = 0$ and hence $\mathsf{lsb}(K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}) = 0$. And if $\mathsf{lsb}(K^0_{\mathtt{id}}) = 1$, then $\sigma_{\mathtt{id}} = 1$ and hence $\mathsf{lsb}(K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}) = \mathsf{lsb}(K^1_{\mathtt{id}}) = \mathsf{lsb}(K^0_{\mathtt{id}}) \oplus 1 = 0$. So in both cases

$$\mathsf{lsb}(K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}) = 0 \ ,$$

as required. From setting $\sigma_{\mathtt{id}} = \mathsf{lsb}(K^0_{\mathtt{id}})$ it also follows that $\sigma_{\mathtt{id}} \oplus 1 = \mathsf{lsb}(K^1_{\mathtt{id}})$, which implies that

$$\mathsf{lsb}(K^b_{\mathtt{id}}) = b \oplus \sigma_{\mathtt{id}} \ ,$$

i.e., the last bit in the key is a one-time pad encryption of the plaintext value of the key with the indicator bit. In particular, given a key and the indicator bit, one can compute the plaintext value of the key as

$$b = \mathsf{lsb}(K^b_{\mathtt{id}}) \oplus \sigma_{\mathtt{id}} \ .$$

The next protocol allows to verify key material associated with an identifier.

function VerWire($\mathtt{id}$)
  G: **Open**($\mathtt{id.dif}$); E: **receive** $\Delta_{\mathtt{id}}$
  E: **check** $\mathsf{lsb}(\Delta_{\mathtt{id}}) = 1$
  G: **Open**($\mathtt{id.ind}$); E: **receive** $\sigma_{\mathtt{id}}$
  G: **Open**($\mathtt{id.base}$); E: **receive** $K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}$
  E: **check** $\mathsf{lsb}(K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}) = 0$
  E: $K^{1-\sigma_{\mathtt{id}}}_{\mathtt{id}} \leftarrow K^{\sigma_{\mathtt{id}}}_{\mathtt{id}} \oplus \Delta_{\mathtt{id}}$
  E: **store** $K^0_{\mathtt{id}}, K^1_{\mathtt{id}}, \Delta_{\mathtt{id}}, \sigma_{\mathtt{id}}$
end function

We say that the key material associated with an identifier is *correct* if it would pass the verification protocol given that G opens all commitments. This just

means that when $K^0_{\mathtt{id}}$, $\Delta_{\mathtt{id}}$ and $\sigma_{\mathtt{id}}$ are defined to be the values in the respective commitments and $K^1_{\mathtt{id}}$ is defined to be $K^0_{\mathtt{id}} \oplus \Delta_{\mathtt{id}}$, then

$$\mathsf{lsb}(\Delta_{\mathtt{id}}) = 1$$

and

$$\mathsf{lsb}(K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}) = 0 \ .$$

Note the key material generated with GenWire is correct. For brevity we will just say that $\mathtt{id}$ is *correct* when the key material associated with $\mathtt{id}$ is correct.

The following protocol reveals information that is used to solder two independent key materials. It will allow to translate a key for $\mathtt{id}_1$ to a key for $\mathtt{id}_2$.

function GenSold($\mathtt{id}_1, \mathtt{id}_2$)
  G: **Open**($\mathtt{id}_1.\mathtt{ind}, \mathtt{id}_2.\mathtt{ind}$); E: **receive** $\sigma_{\mathtt{id}_1,\mathtt{id}_2}$
  if $\sigma_{\mathtt{id}_1,\mathtt{id}_2} = 0$ **then**
    G: **Open**($\mathtt{id}_1.\mathtt{base}, \mathtt{id}_2.\mathtt{base}$)
  else
    G: **Open**($\mathtt{id}_1.\mathtt{base}, \mathtt{id}_2.\mathtt{base}, \mathtt{id}_2.\mathtt{dif}$)
  end if
  E: **receive** $K_{\mathtt{id}_1,\mathtt{id}_2}$
  G: **Open**($\mathtt{id}_1.\mathtt{dif}, \mathtt{id}_2.\mathtt{dif}$)
  E: **receive** $\Delta_{\mathtt{id}_1,\mathtt{id}_2}$
  check $\mathsf{lsb}(\Delta_{\mathtt{id}_1,\mathtt{id}_2}) = 0$
  check $\mathsf{lsb}(K_{\mathtt{id}_1,\mathtt{id}_2}) = \sigma_{\mathtt{id}_1,\mathtt{id}_2}$
end function

The last two checks ensure that if one of the key materials are correct, then the other one is correct too. To see this, assume that $\mathtt{id}_1$ is correct. This just means that

$$\mathsf{lsb}(\Delta_{\mathtt{id}_1}) = 1$$
$$\mathsf{lsb}(K^{\sigma_{\mathtt{id}_1}}_{\mathtt{id}_1}) = 0 \ .$$

From $\mathsf{lsb}(\Delta_{\mathtt{id}_1}) = 1$, $\mathsf{lsb}(\Delta_{\mathtt{id}_1,\mathtt{id}_2}) = 0$ and $\Delta_{\mathtt{id}_1,\mathtt{id}_2} = \Delta_{\mathtt{id}_1} \oplus \Delta_{\mathtt{id}_2}$ it follows that

$$\mathsf{lsb}(\Delta_{\mathtt{id}_2}) = 1 \ .$$

We have by construction that

$$K_{\mathtt{id}_1,\mathtt{id}_2} = K^{\sigma_{\mathtt{id}_1}}_{\mathtt{id}_1} \oplus K^{\sigma_{\mathtt{id}_2}}_{\mathtt{id}_2} \oplus \sigma_{\mathtt{id}_1,\mathtt{id}_2}\Delta_{\mathtt{id}_2} \ .$$

Since we already established that $\mathsf{lsb}(\Delta_{\mathtt{id}_2}) = 1$ and we assumed that $\mathsf{lsb}(K^{\sigma_{\mathtt{id}_1}}_{\mathtt{id}_1}) = 0$, we have that

$$\mathsf{lsb}(K_{\mathtt{id}_1,\mathtt{id}_2}) = 0 \oplus \mathsf{lsb}(K^{\sigma_{\mathtt{id}_2}}_{\mathtt{id}_2}) \oplus \sigma_{\mathtt{id}_1,\mathtt{id}_2} \ .$$

Since E checks that $\mathsf{lsb}(K_{\mathtt{id}_1,\mathtt{id}_2}) = \sigma_{\mathtt{id}_1,\mathtt{id}_2}$, it follows that

$$\mathsf{lsb}(K^{\sigma_{\mathtt{id}_2}}_{\mathtt{id}_2}) = 0 \ .$$

Hence $\mathtt{id}_2$ is correct. Showing that $\mathtt{id}_1$ is correct if $\mathtt{id}_2$ is correct follows a symmetric argument.

Note then that if both key materials are correct, then

$$\begin{aligned} K_{\mathtt{id}_1,\mathtt{id}_2} &= K_{\mathtt{id}_1}^{\sigma_{\mathtt{id}_1}} \oplus K_{\mathtt{id}_2}^{\sigma_{\mathtt{id}_2}} \oplus \sigma_{\mathtt{id}_1,\mathtt{id}_2}\Delta_{\mathtt{id}_2} \\ &= K_{\mathtt{id}_1}^{0} \oplus \sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_1} \oplus K_{\mathtt{id}_2}^{0} \oplus \sigma_{\mathtt{id}_2}\Delta_{\mathtt{id}_2} \oplus \sigma_{\mathtt{id}_1,\mathtt{id}_2}\Delta_{\mathtt{id}_2} \\ &= K_{\mathtt{id}_1}^{0} \oplus \sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_1} \oplus K_{\mathtt{id}_2}^{0} \oplus \sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_2} \\ &= K_{\mathtt{id}_1}^{0} \oplus K_{\mathtt{id}_2}^{0} \oplus \sigma_{\mathtt{id}_1}(\Delta_{\mathtt{id}_1} \oplus \Delta_{\mathtt{id}_2}) \\ &= K_{\mathtt{id}_1}^{0} \oplus K_{\mathtt{id}_2}^{0} \oplus \sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_1,\mathtt{id}_2} \ . \end{aligned}$$

We use this later.

The following protocol shows how to use the soldering information. It assumes that E already knows a key $K_{\mathtt{id}}$ for wire $\mathtt{id}$. This key is either $K_{\mathtt{id}}^0$ or $K_{\mathtt{id}}^1$, but E might not know the plaintext value, so we use the generic name $K_{\mathtt{id}}$ for the key.

function EvSold($\mathtt{id}_1, \mathtt{id}_2, K$)
  E: **return** $K \oplus K_{\mathtt{id}_1,\mathtt{id}_2} \oplus \mathsf{lsb}(K)\Delta_{\mathtt{id}_1,\mathtt{id}_2}$
end function
function EvSold($\mathtt{id}_1, \mathtt{id}_2$)
  E: **retrieve** $K_{\mathtt{id}_1}$
  E: $K_{\mathtt{id}_2} \leftarrow$ EvSold($\mathtt{id}_1, \mathtt{id}_2, K_{\mathtt{id}_1}$)
end function

Note that if both key material is correct and $K_{\mathtt{id}_1} = K_{\mathtt{id}_1}^b = K_{\mathtt{id}_1}^0 \oplus b\Delta_{\mathtt{id}_1}$, then

$$\begin{aligned} &K_{\mathtt{id}_1} \oplus K_{\mathtt{id}_1,\mathtt{id}_2} = \\ &(K_{\mathtt{id}_1}^0 \oplus b\Delta_{\mathtt{id}_1}) \oplus (K_{\mathtt{id}_1}^0 \oplus K_{\mathtt{id}_2}^0 \oplus \sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_1,\mathtt{id}_2}) = \\ &K_{\mathtt{id}_2}^0 \oplus b\Delta_{\mathtt{id}_1} \oplus \sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_1,\mathtt{id}_2} \ . \end{aligned}$$

It is easy to see that when the key materials are correct then $\mathsf{lsb}(K_{\mathtt{id}_1}) = b \oplus \sigma_{\mathtt{id}_1}$, from which it follows that

$$\sigma_{\mathtt{id}_1}\Delta_{\mathtt{id}_1,\mathtt{id}_2} \oplus \mathsf{lsb}(K_{\mathtt{id}_1})\Delta_{\mathtt{id}_1,\mathtt{id}_2} = b\Delta_{\mathtt{id}_1,\mathtt{id}_2} \ .$$

So,

$$\begin{aligned} K_{\mathtt{id}_2} &= K_{\mathtt{id}_2}^0 \oplus b\Delta_{\mathtt{id}_1} \oplus b\Delta_{\mathtt{id}_1,\mathtt{id}_2} \\ &= K_{\mathtt{id}_2}^0 \oplus b\Delta_{\mathtt{id}_2} \\ &= K_{\mathtt{id}_2}^b \ , \end{aligned}$$

so $K_{\mathtt{id}_1}^b$ is mapped to $K_{\mathtt{id}_2}^b$, as intended.

Lemma 2 (correctness of soldering)**.** If either $\mathtt{id}_1$ is correct or $\mathtt{id}_2$ is correct and $\mathtt{id}_1$ has been soldered onto $\mathtt{id}_2$ without E aborting, then *both* of them are correct and

$$\textsc{EvSold}(\mathtt{id}_1, \mathtt{id}_2, K^b_{\mathtt{id}_1}) = K^b_{\mathtt{id}_2}$$

for $b = 0, 1$.

The next protocol is used to generate key authenticators.

function GenKeyAuth($\mathtt{id}$)
  G: $K^0_{\mathtt{id}} \leftarrow \{0,1\}^\kappa$
  G: $\Delta_{\mathtt{id}} \leftarrow \{0,1\}^{\kappa-1} \times \{1\}$
  G: GenWire($\mathtt{id}, K^0_{\mathtt{id}}, \Delta^0_{\mathtt{id}}$)
  G: $A_{\mathtt{id}} \leftarrow \{H(K^0_{\mathtt{id}}), H(K^1_{\mathtt{id}})\}$
  G: P_s $A_{\mathtt{id}}$
end function

Given a value $A = \{h_1, h_2\}$ and a key K we write $A(K) = \top$ if $H(K) \in A$. Otherwise we write $A(K) = \bot$. This protocol allows to verify a key authenticator.

function VerKeyAuth($\mathtt{id}$)
  VerWire($\mathtt{id}$)
  E: **check** $A_{\mathtt{id}} = \{H(K^0_{\mathtt{id}}), H(K^1_{\mathtt{id}})\}$
end function

We call a key authenticator with identifier $\mathtt{id}$ *correct* if it would pass the verification algorithm. Note that if it is correct, then $A_{\mathtt{id}} = \{H(K^0_{\mathtt{id}}), H(K^1_{\mathtt{id}})\}$ and E knows $K^0_{\mathtt{id}}$ and $K^1_{\mathtt{id}}$ as $K^{\sigma_{\mathtt{id}}}_{\mathtt{id}}$ and $\Delta_{\mathtt{id}}$ were input to the commitment functionality. It therefore follows from the collision resistance of H that if $A(K) = \top$ for a key K computed in polynomial time, then $K \in \{K^0_{\mathtt{id}}, K^1_{\mathtt{id}}\}$ except with negligible probability. Furthermore, if in addition $A(K') = \top$ and $K' \neq K$, then $K \oplus K' = \Delta_{\mathtt{id}}$.

When we generate key authenticators for input gates, we will use the special form that $K^0_{\mathtt{id}} = H(\Delta_{\mathtt{id}})$.

function GenInKeyAuth($\mathtt{id}$)
  $\Delta_{\mathtt{id}} \leftarrow \{0,1\}^{\kappa-1} \times \{1\}$
  $K^0_{\mathtt{id}} \leftarrow H(\Delta_{\mathtt{id}})$
  GenWire($\mathtt{id}, K^0_{\mathtt{id}}, \Delta^0_{\mathtt{id}}$)
  $A_{\mathtt{id}} \leftarrow \{H(K^0_{\mathtt{id}}), H(K^1_{\mathtt{id}})\}$
  G **sends** $A_{\mathtt{id}}$
end function

To verify this special form we use this protocol.

function VerInKeyAuth($\mathtt{id}$)
  VerKeyAuth($\mathtt{id}$)
  E: **check** $K_{\mathtt{id}} = H(\Delta_{\mathtt{id}})$
end function

Note that if we are given a key $K^b_{\mathtt{id}}$ and $\Delta_{\mathtt{id}}$ for an input gate, then we can compute b as follows. First compute $K^0_{\mathtt{id}} = H(\Delta_{\mathtt{id}})$. If $K^b_{\mathtt{id}} = K^0_{\mathtt{id}}$, then $b = 0$. Otherwise $b = 1$.

The following sub-protocol prepares a lot of key authenticators and uses cut-and-choose to verify that most are correct. The unopened key authenticators are put into buckets with a majority of correct ones in each bucket.

function PreProcessKA
  $\ell \leftarrow$ #output wires of all functional sub-circuits
  Let $L = L^2(\ell)$ ▷ # KAs generated
  Let $\alpha_{\mathtt{ka}} = \alpha^2(\ell)$ ▷ bucket size
  $\forall_{i=1}^{L}$: GenKeyAuth(preka.i)
  E: Sample $V \subset [L]$ uniform of size $L - \alpha_{\mathtt{ka}}\ell$.
  E: P_s V
  $\forall_{i \in V}$: VerKeyAuth(preka.i)
  for all functional sub-circuits id **do**
    for all $j = 1, \ldots, m_{\mathtt{id}}$ **do**
      pick $\alpha_{\mathtt{ka}}$ uniform, fresh KAs $i \notin V$
      rename them to ids id.ka.$1, \ldots,$ id.ka.$\alpha_{\mathtt{ka}}$.
      $\forall_{i=2}^{\alpha_{\mathtt{ka}}}$: GenSold(id.ka.1, id.ka.i)
    end for
  end for
end function

We call id.ka.$1, \ldots,$ id.ka.$\alpha_{\mathtt{ka}}$ a KA bucket, and we identify it by id.ka. We call id.ka KM correct if the key material associated with each id.ka.j is correct. We call it KA correct if it is KM correct and furthermore a majority of the KAs are correct, as defined above. By definition of L^2 and α^2 it follows that for each id.ka there will be a majority of id.ka.j for which the key material is correct, so there is in particular at least one for which this is true (except with negligible probability). By Lemma 2 this implies that the bucket is KM correct. By definition of L^2 and α^2 it then follows that it is additionally KA correct. We get that:

Lemma 3 (robustness of KA buckets)**.** If E is honest and G is honest or corrupted, then except with negligible probability each KA bucket id.ka is KA correct.

We also use protocols PreProcessInKA, which work exactly as PreProcessKA, except that in PreProcessInKA we let ℓ be the number of input wires in C, we pre-process the key authenticators using GenInKeyAuth instead, we verify using VerInKeyAuth, and we associated the unopened key authenticators to the identifiers of the input wires instead. For an input wire id the identifiers of the key authenticator will be id.ka.$1, \ldots,$ id.ka.$\alpha_{\mathtt{inka}}$.

The following protocol evaluates a key authenticator. It selects from a set of keys, namely the keys that are accepted by a majority of the key authenticators.

If G is corrupted, then E might end up in the situation where it learns both a 0-key and a 1-key for a wire, if both of these are in the input key set. In that case we use a special recovery procedure RECOVER. This procedure will recover the plaintext input of E and use it to do a plaintext evaluation of the circuit instead of garbled evaluation. How this is done is described later.

function EVKAS($\mathtt{id}, \mathcal{K}_{\mathtt{id}}$)
  $\alpha \leftarrow \alpha_{\mathtt{ka}}$ ▷ if generated using PREPROCESSKA
  $\alpha \leftarrow \alpha_{\mathtt{inka}}$ ▷ if gen. using PREPROCESSINKA
  $\forall_{i=1}^{\alpha} : A_i \leftarrow A_{\mathtt{id.ka}.i}$ ▷ get key authenticators
  $\mathcal{L} \leftarrow \emptyset$
  for $K \in \mathcal{K}_{\mathtt{id}}$ **do**
    $K_1 = K$
    $\forall_{i=2}^{\alpha} : K_i = \text{EVSOLD}(\mathtt{id.ka}.1, \mathtt{id.ka}.i, K)$
    if $\#\{i \in \{1, \ldots, \alpha\} \mid A_i(K_i) = \top\} > \alpha/2$ **then**
      $\mathcal{L} \leftarrow \mathcal{L} \cup \{K\}$
    end if
  end for
  if $\mathcal{L} = \{K\}$ **then return** K
  else if $\mathcal{L} = \{K_0, K_1\}$ **then**
    $\Delta \leftarrow K_0 \oplus K_1$
    RECOVER($\mathtt{id}, \Delta$)
  else abort
  end if
end function

Note that if $\mathtt{id.ka}$ is KA correct, then a majority of the key authenticators are correct and all the key materials are correct. Therefore, if a key is put in $\mathcal{L}$, then it is was accepted by at least one correct key authenticator. So, if a key K is in $\mathcal{L}$, then by the properties of correct key authenticators and Lemma 2, it follows that $K \in \{K^0_{\mathtt{id.ka}.1}, K^1_{\mathtt{id.ka}.1}\}$ except with negligible probability. So, assuming that $K^b_{\mathtt{id.ka}.1} \in \mathcal{K}_{\mathtt{id}}$ the outcome of the protocol is either to return $K^b_{\mathtt{id.ka}.1}$ or to call the recover protocol with the correct $\Delta = \Delta_{\mathtt{id.ka}.1}$, except with negligible probability.

Lemma 4 (robustness of EVKAS)**.** If E is honest and G is honest or corrupted, the following holds except with negligible probability. If for some $b \in \{0, 1\}$ it holds that $K^b_{\mathtt{id.ka}.1} \in \mathcal{K}_{\mathtt{id}}$, then EVKAS returns $K^b_{\mathtt{id.ka}.1}$ or calls RECOVER($\mathtt{id}, \Delta$) with $\Delta = \Delta_{\mathtt{id.ka}.1}$. If for no $b \in \{0, 1\}$ it holds that $K^b_{\mathtt{id.ka}.1} \in \mathcal{K}_{\mathtt{id}}$, then EVKAS aborts.

The following protocol allows E to learn $K^{x_{\mathtt{id}}}_{\mathtt{id}}$ for a G in-gate, where G has input x_i, without E learning x_i.

function INPUTG($\mathtt{id}$) ▷ $\mathtt{id}$ is an ID of a G in-gate
  G: **retrieve** the input bit $x_{\mathtt{id}}$ for $\mathtt{id}$
  G: $K \leftarrow K^{x_{\mathtt{id}}}_{\mathtt{id.ka}.1}$

G: P_s K
E: $K_{\mathtt{id}} \leftarrow \text{EvKAs}(\mathtt{id}, \{K\})$
E: **store** $K_{\mathtt{id}}$
end function

Definition 2. If G is corrupted and E is honest, then after an execution of InputG($\mathtt{id}$), define $x_{\mathtt{id}}$ as follows. If $\mathtt{id.ka}$ is KA correct, then define $x_{\mathtt{id}}$ by $K^{x_{\mathtt{id}}}_{\mathtt{id.ka}.1} = K_{\mathtt{id}}$. Otherwise let $x_{\mathtt{id}} = \bot$.

By Lemma 4, if G sends a key not from $\{K^0_{\mathtt{id.ka}.1}, K^1_{\mathtt{id.ka}.1}\}$, then the procedure aborts. Otherwise it outputs that key.

Lemma 5 (robustness of InputG)**.** If G is corrupted and E is honest, then after an execution of InputG($\mathtt{id}$) that did not abort, it holds except with negligible probability that $x_{\mathtt{id}} \in \{0,1\}$ and it holds for the key $K_{\mathtt{id}}$ then stored by E that $K_{\mathtt{id}} = K^{x_{\mathtt{id}}}_{\mathtt{id}}$.

It is more complicated to give input to E as G is not to learn which key was received. To prepare for this oblivious input delivery we set up the oblivious transfer functionality such that G is also committed to the Δ chosen for the OT.

function PreProcessOTInit
G: $\Delta_{\mathtt{ot}} \leftarrow \{0,1\}^\kappa$
G: **Commit**($\mathtt{ot}, \Delta_{\mathtt{ot}}$)
G: OTInit($\Delta_{\mathtt{ot}}$)
for $i \in [s]$ **do**
G: $R_i \leftarrow \{0,1\}^\kappa$, OTSend($R_i, \mathtt{ot}_i$)
G: **Commit**($\mathtt{ot}_i, R_i$)
E: $b_i \leftarrow \{0,1\}, R_{b_i} \leftarrow$ OTReceive($b_i, \mathtt{ot}_i$)
E: P_s (R_{b_i}, b_i)
G: **receive** $(\bar{R}_i, \bar{b}_i)$; **check** $\bar{R}_i = R_{\bar{b}_i}$
if $\bar{b}_i = 0$ **then**
G: **Open**($\mathtt{ot}_i$)
else
G: **Open**($\mathtt{ot}_i, \mathtt{ot}$)
end if
E: **receive** $\tilde{R}_i$
E: **check** $\tilde{R}_i = R_{b_i}$
end for
end function

The PreProcessOTInit procedure ensures that the commitment to the identifier $\mathtt{ot}$ is a commitment to the difference $\Delta_{\mathtt{ot}}$ which G chose for the OT functionality. It was proven in [NST17] that the protocol is sound except with probability 2^{-s} and that it is straight-line zero-knowledge when the simulator controls the commitment functionality.

The following protocol allows E to learn $K_{\mathtt{id}}^{x_{\mathtt{id}}}$ for an E in-gate, where E has input x_i, without G learning x_i and without E learning anything else.

function InputE($\mathtt{id}$) ▷ $\mathtt{id}$ is an ID of an E in-gate
  G: $R_{\mathtt{ot}_{\mathtt{id}}} \leftarrow \{0,1\}^\kappa$, OTsend$(R_{\mathtt{ot}_{\mathtt{id}}}, \mathtt{ot}_{\mathtt{id}})$
  G: **Commit**$(\mathtt{ot}_{\mathtt{id}}, R_{\mathtt{ot}_{\mathtt{id}}})$
  E: $b_{\mathtt{ot}_{\mathtt{id}}} \leftarrow \{0,1\}, R_{b_{\mathtt{ot}_{\mathtt{id}}}} \leftarrow$ OTreceive$(b_{\mathtt{ot}_{\mathtt{id}}}, \mathtt{ot}_{\mathtt{id}})$
  E: **retrieve** the input bit $x_{\mathtt{id}}$ for $\mathtt{id}$
  E: P_s $f_{\mathtt{id}} = x_{\mathtt{id}} \oplus b_{\mathtt{ot}_{\mathtt{id}}}$
  G: $e_{\mathtt{id}} = f_{\mathtt{id}} \oplus \sigma_{\mathtt{id}}$
  E: $\mathtt{id}' \leftarrow \mathtt{id}.\mathtt{ka}.1$
  if $e_{\mathtt{id}} = 0$ **then**
    G: **Open**$(\mathtt{id}', \mathtt{ot}_{\mathtt{id}})$
  else
    G: **Open**$(\mathtt{id}', \mathtt{ot}_{\mathtt{id}}, \mathtt{ot})$
  end if
  E : **receive** $D = K_{\mathtt{id}'} \oplus R_{\mathtt{ot}_{\mathtt{id}}} \oplus e_{\mathtt{id}}\Delta_{\mathtt{ot}}$
  G: **Open**$(\mathtt{id}'.\mathtt{dif}, \mathtt{ot})$; E : **receive** $S_{\mathtt{id}} = \Delta_{\mathtt{id}'} \oplus \Delta_{\mathtt{ot}}$
  G: **Open**$(\mathtt{id}.\mathtt{ind})$; E : **receive** $\sigma_{\mathtt{id}}$
  E: $K = D \oplus R_{b_{\mathtt{ot}_{\mathtt{id}}}} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})S_{\mathtt{id}}$
  E: $K_{\mathtt{id}'} \leftarrow$ EvKAs$(\mathtt{id}, \{K\})$
  E: **check** $\mathsf{lsb}(K_{\mathtt{id}'}) = x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}}$
  E: **store** $K_{\mathtt{id}'}$
end function

Then identify the output wire $\mathtt{id}.1$ with $\mathtt{id}.\mathtt{ka}.1$, such that $K_{\mathtt{id}} = K_{\mathtt{id}'}$. If G in the above commits honestly to the value $R_{\mathtt{ot}_{\mathtt{id}}}$ he also inputs to the OT functionality then we indeed see that

$$\begin{aligned}
K_{\mathtt{id}} &= D \oplus R_{b_{\mathtt{ot}_{\mathtt{id}}}} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})S_{\mathtt{id}} \\
&= (K_{\mathtt{id}'} \oplus R_{\mathtt{ot}_{\mathtt{id}}} \oplus e_{\mathtt{id}}\Delta_{\mathtt{ot}}) \oplus R_{b_{\mathtt{ot}_{\mathtt{id}}}} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})(\Delta_{\mathtt{id}'} \oplus \Delta_{\mathtt{ot}}) \\
&= (K_{\mathtt{id}'} \oplus b_{\mathtt{ot}_{\mathtt{id}}}\Delta_{\mathtt{ot}} \oplus e_{\mathtt{id}}\Delta_{\mathtt{ot}}) \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})(\Delta_{\mathtt{id}'} \oplus \Delta_{\mathtt{ot}}) \\
&= K_{\mathtt{id}'} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})\Delta_{\mathtt{ot}} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})(\Delta_{\mathtt{id}'} \oplus \Delta_{\mathtt{ot}}) \\
&= K_{\mathtt{id}'} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})\Delta_{\mathtt{id}'} \\
&= K_{\mathtt{id}'}^{x_{\mathtt{id}}} = K_{\mathtt{id}}^{x_{\mathtt{id}}} \ .
\end{aligned}$$

If instead G cheats and commits to a value $\bar{R} \neq R_{\mathtt{ot}_{\mathtt{id}}}$ then

$$K = K_{\mathtt{id}'}^{x_{\mathtt{id}}} \oplus F \ ,$$

where

$$F = \bar{R} \oplus R_{\mathtt{ot}_{\mathtt{id}}} \ .$$

Note that when G is honest, then $F = 0$ and therefore $K = K_{\mathtt{id}'}^{x_{\mathtt{id}}}$. If G uses $F \notin \{0, \Delta_{\mathtt{id}'}\}$, then

$$K \notin \{K_{\mathtt{id}'}^0, K_{\mathtt{id}'}^1\} \ ,$$

so by Lemma 4 it holds that except with negligible probability the procedure aborts. Importantly, this happens independent of the value of $x_{\mathtt{id}}$. If G uses $F = \Delta_{\mathtt{id}'}$, then

$$K = K_{\mathtt{id}'}^{x_{\mathtt{id}}} \oplus \Delta_{\mathtt{id}'} = K_{\mathtt{id}'}^{1 \oplus x_{\mathtt{id}}} \ .$$

so by Lemma 4 it holds that

$$K_{\mathtt{id}'} = K_{\mathtt{id}'}^{1 \oplus x_{\mathtt{id}}} \ .$$

As we have ensured that $\mathsf{lsb}(\Delta_{\mathtt{id}'}) = 1$ for all KA buckets except with negligible probability (by Lemma 3) it follows from the check $\mathsf{lsb}(K_{\mathtt{id}}) = x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}}$ that when $K = K_{\mathtt{id}'}^{1 \oplus x_{\mathtt{id}}}$, then the procedure aborts. So, all in all, if a cheating E uses $F \neq 0$, then the procedure aborts except with negligible probability, independently of the value of $x_{\mathtt{id}}$.

Lemma 6 (robustness of InputE)**.** The following holds except with negligible probability. If G is corrupted and E is honest, then an execution of InputE($\mathtt{id}$) will abort or not independently of the value of $x_{\mathtt{id}}$. Furthermore, when it does not abort, then it holds for the key $K_{\mathtt{id}}$ stored by E that $K_{\mathtt{id}} = K_{\mathtt{id}}^{x_{\mathtt{id}}}$.

Functional Sub-Circuits

The following protocol generates a garbled circuit of circuit C and generates the key material for all input wires and output wires. All these wires share the same difference.

function GenSub($\mathtt{id}, C$) ▷ $C : \{0,1\}^n \to \{0,1\}^m$
  G: $(K_1, \ldots, K_n) \leftarrow (\{0,1\}^\kappa)^n$
  G: $\Delta_{\mathtt{id}} \leftarrow \{0,1\}^{\kappa-1} \times \{1\}$
  G: $(L_1, \ldots, L_m, F_{\mathtt{id}}) \leftarrow \mathsf{Gb}(K_1, \ldots, K_n, \Delta_{\mathtt{id}})$
  G: P_s $F_{\mathtt{id}}$
  $\forall_{i=1}^n$: GenWire($\mathtt{id.in}.i, K_i, \Delta_{\mathtt{id}}$)
  $\forall_{i=1}^m$: GenWire($\mathtt{id.out}.i, L_i, \Delta_{\mathtt{id}}$)
end function

The following protocol allows to verify that the garbling was done correctly.

function VerSub($\mathtt{id}, C$) ▷ $C : \{0,1\}^n \to \{0,1\}^m$
  $\forall_{i=1}^n$: VerWire($\mathtt{id.in}.i$)
  $\forall_{i=1}^m$: VerWire($\mathtt{id.out}.i$)
  E: $\Delta_{\mathtt{id}} \leftarrow \Delta_{\mathtt{id.in}.1}$
  E: $\forall_{i=2}^n$: **check** $\Delta_{\mathtt{id.in}.i} = \Delta_{\mathtt{id}}$
  E: $\forall_{i=1}^m$: **check** $\Delta_{\mathtt{id.out}.i} = \Delta_{\mathtt{id}}$
  E: $\forall_{i=1}^n$: $K_i \leftarrow K_{\mathtt{id.in}.i}^0$
  E: $\forall_{i=1}^m$: $L_i \leftarrow K_{\mathtt{id.out}.i}^0$
  E: **check** $(L_1, \ldots, L_m, F_{\mathtt{id}}) = \mathsf{Gb}(K_1, \ldots, K_n, \Delta_{\mathtt{id}})$
end function

We call a generated garbled sub-circuit with identifier $\mathtt{id}$ *correct* if it would pass the above verification protocol. Notice that if it is correct, then for all $x \in \{0,1\}^n$ and $y = C(x)$ it holds that $\mathsf{Ev}(F_{\mathtt{id}}, K^x_{\mathtt{id.in}}) = K^y_{\mathtt{id.out}}$, which shows that the following protocol works as intended.

function EvSub($\mathtt{id}$)
  E: $\forall_{i=1}^{n} : K_i \leftarrow$ **retrieve** $K_{\mathtt{id.in}.i}$
  E: $(L_1, \ldots, L_m) \leftarrow \mathsf{Ev}(F_{\mathtt{id}}, K_1, \ldots, K_n)$
  E: $\forall_{i=1}^{m} :$ **store** $K_{\mathtt{id.out}.i} \leftarrow L_i$
end function

Lemma 7 (correctness of EvSub)**.** If $K_{\mathtt{id.in}.i} = K^{x_i}_{\mathtt{id.in}.i}$ for $i = 1, \ldots, n$ and $\mathtt{id}$ is correct, then after an execution of EvSub($\mathtt{id}$) it holds that $K_{\mathtt{id.out}.i} = K^{y_i}_{\mathtt{id.out}.i}$ for $i = 1, \ldots, m$, where $y = C(x)$.

The following protocol will generate garblings of all the needed types of sub-circuits. It will generate more than needed. Some of these will be opened and checked. The rest will be assigned at random into buckets. Each slots $\mathtt{id}$ will be assigned some $\alpha_{\mathtt{id}}$ unopened circuits.

function PreProcessSub
  for all sub-circuit types C **do**
    Let ℓ be the number of times C is used.
    Let $L = L^1(\ell)$ ▷ #circuits generated
    Let $\alpha_{\mathtt{id}} = \alpha^1(\ell)$ ▷ bucket size
    $\forall_{i=1}^{L}$: GenSub($C.\mathtt{pre}.i, C$)
    E: Sample $V \subset [L]$ uniform of size $L - \alpha_{\mathtt{id}}\ell$.
    E: $P_s\ V$
    $\forall_{i \in V}$: VerSub($C.\mathtt{pre}.i, C$)
    for all slots $\mathtt{id}$ where C occurs **do**
      pick $\alpha_{\mathtt{id}}$ uniform, fresh circuits $i \notin V$
      rename them to have ids $\mathtt{id}.1, \ldots, \mathtt{id}.\alpha_{\mathtt{id}}$.
      $\forall_{i=2}^{\alpha_{\mathtt{id}}}$: GenSoldSub($\mathtt{id}.1, \mathtt{id}.i$)
    end for
  end for
end function

By *fresh* we mean that no circuit is assigned to more than one slot. We used this sub-protocol.

function GenSoldSub($\mathtt{id}_1, \mathtt{id}_2$)
  $\forall_{i=1}^{n}$: GenSold($\mathtt{id}_1.\mathtt{in}.i, \mathtt{id}_2.\mathtt{in}.i$)
  $\forall_{i=1}^{m}$: GenSold($\mathtt{id}_2.\mathtt{out}.i, \mathtt{id}_1.\mathtt{out}.i$)
  $\forall_{i=1}^{m}$: GenSold($\mathtt{id}.1.\mathtt{out}.i, \mathtt{id}.1.\mathtt{out}.i.\mathtt{ka}.1$)
end function

After all sub-circuits have been preprocessed the below procedure can be applied to stitch them all together to compute the final functionality C.

function AssembleSubs

for all functional sub-circuits $\mathtt{id}$ **do**
$\forall_{i=1}^{n}$: $\mathtt{id}_{\mathtt{par}.i} \leftarrow \mathrm{Par}(\mathtt{id}.1.\mathtt{in}.i)$
$\forall_{i=1}^{n}$: GenSold$(\mathtt{id}_{\mathtt{par}.i}, \mathtt{id}.1.\mathtt{in}.i)$
end for
end function

If the linking of components are done adaptively, the above soldering is done only when needed.

The following procedure is used to evaluate a bucket of garbled sub-circuits that have been soldered together. Since each sub-circuit might give different output keys (if G is corrupted) the output for each output wire might not be a single key, but a set of keys. We therefore use the associated KAs to reduce this set to one key. If the reduced set contains two keys we will again call the recovery mechanism, which we are still to describe.

function EvSubs($\mathtt{id}$)
▷ Evaluate first circuit in bucket
$\forall_{j=1}^{n}$: $\mathtt{id}_{\mathtt{par}.j} \leftarrow \mathrm{Par}(\mathtt{id}.1.\mathtt{in}.j)$
$\forall_{j=1}^{n}$: EvSold$(\mathtt{id}_{\mathtt{par}.j}, \mathtt{id}.1.\mathtt{in}.j)$
$\forall_{j=1}^{n}$: **retrieve** $K_{\mathtt{id}.1.\mathtt{in}.j}$
EvSub($\mathtt{id}.1$)
$\forall_{j=1}^{m}$: **retrieve** $K_{\mathtt{id}.1.\mathtt{out}.j}$
$\forall_{j=1}^{m}$: EvSold$(\mathtt{id}.1.\mathtt{out}.j, \mathtt{id}.1.\mathtt{out}.j.\mathtt{ka}.1)$
$\forall_{j=1}^{m}$: $\mathcal{K}_j \leftarrow \{K_{\mathtt{id}.1.\mathtt{out}.j.\mathtt{ka}.1}\}$ ▷ key sets
▷ Evaluate remaining circuits in bucket
for $i = 2, \ldots, \alpha_{\mathtt{id}}$ **do** E:
$\forall_{j=1}^{n}$: EvSold$(\mathtt{id}.1.\mathtt{in}.j, \mathtt{id}.i.\mathtt{in}.j)$
EvSub($\mathtt{id}$.i)
$\forall_{j=1}^{m}$: EvSold$(\mathtt{id}.i.\mathtt{out}.j, \mathtt{id}.1.\mathtt{out}.j)$
$\forall_{j=1}^{m}$: EvSold$(\mathtt{id}.1.\mathtt{out}.j, \mathtt{id}.1.\mathtt{out}.j.\mathtt{ka}.1)$
$\forall_{j=1}^{m}$: $\mathcal{K}_j \leftarrow \mathcal{K}_j \cup \{K_{\mathtt{id}.1.\mathtt{out}.j.\mathtt{ka}.1}\}$
end for
$\forall_{j=1}^{m}$: $K_{\mathtt{id}.1.\mathtt{out}.j} \leftarrow \mathrm{EvKAs}(\mathtt{ka}_j, \mathcal{K}_j)$
end function

Lemma 8 (robustness of EvSubs)**.** If G is corrupt and E is honest and the sub-circuit for $\mathtt{id}$ is C, then the following holds except we negligible probability. If $K_{\mathtt{id}_{\mathtt{par}.j}} = K^{x_j}_{\mathtt{id}_{\mathtt{par}.j}}$ for $j = 1, \ldots, n$, then after an execution of EvSubs($\mathtt{id}$) that did not call Recover it holds that $K_{\mathtt{id}.1.\mathtt{out}.j} = K^{y_j}_{\mathtt{id}.1.\mathtt{out}.j}$ for $j = 1, \ldots, m$, where $y = C(x)$.

Proof. All the following statements hold except with negligible probability, assuming the premise of the above lemma. As we argued when we established Lemma 4, we can argue that all the key material used in EvSubs is correct. This follows from Lemma 2 as all the key material sits in buckets that are soldered and each bucket contains at least one correct key material by construction. It

then follows from Lemma 2 that $K_{\mathtt{id}.1.\mathtt{in}.j} = K^{x_i}_{\mathtt{id}.1.\mathtt{in}.j}$. By definition of L^1 and α^1 we have that at least one circuit $\mathtt{id}.i$ is correct. By Lemma 2 and Lemma 4 it therefore follows that $K_{\mathtt{id}.1.\mathtt{out}.j} \in \mathcal{K}_j$. By Lemma 4 it then follows that either RECOVER is called, or $K_{\mathtt{id}.1.\mathtt{out}.j} = K^{y_j}_{\mathtt{id}.1.\mathtt{out}.j}$. □

Output

The following protocol allows E to get an output.

function OUTPUTE($\mathtt{id}$) ▷ $\mathtt{id}$: ID of an E output gate
  E: **retrieve** soldering $(\mathtt{id}_1, \mathtt{id}.\mathtt{out}.1)$ from C.
  E: **retrieve** $K_{\mathtt{id}_1}$.
  G: **Open**($\mathtt{id}_1.\mathtt{ind}$); E: **receive** $\sigma_{\mathtt{id}_1}$
  E: $y_{\mathtt{id}} \leftarrow \mathsf{lsb}(K_{\mathtt{id}_1}) \oplus \sigma_{\mathtt{id}_1}$.
end function

Lemma 9 (robustness of OUTPUTE)**.** If G is corrupt and E is honest and $K_{\mathtt{id}_1} = K^b_{\mathtt{id}_1}$, then after an execution of OUTPUTE that does not abort, it holds that $y_{\mathtt{id}} = b$. Furthermore, whether or not the protocol aborts is independent of $y_{\mathtt{id}}$.

The first part of the lemma follows from $\mathsf{lsb}(K^b_{\mathtt{id}_1}) = b \oplus \sigma_{\mathtt{id}_1}$. The second part is obvious as the protocol aborts only if the commitment is not opened, which is the choice of G, and G does not know $y_{\mathtt{id}}$.

The following protocol allows G to get an output.

function OUTPUTG($\mathtt{id}$) ▷ $\mathtt{id}$: ID of an G output gate
  E: **retrieve** soldering $(\mathtt{id}_1, \mathtt{id}.\mathtt{out}.1)$ from C
  E: **retrieve** $K_{\mathtt{id}_1}$
  E: P_s $K_{\mathtt{id}_1}$
  G: **receive** $K_{\mathtt{id}_1}$
  G: **check** $K_{\mathtt{id}_1} \in \{K^0_{\mathtt{id}_1}, K^1_{\mathtt{id}_1}\}$
  G: $y_{\mathtt{id}} \leftarrow b$ where $K^b_{\mathtt{id}_1} = K_{\mathtt{id}_1}$.
end function

Correctness follows from authenticity of the garbling scheme. Even a corrupt E will know some key $K_{\mathtt{id}_1} \in \{K^0_{\mathtt{id}_1}, K^1_{\mathtt{id}_1}\}$ and this by correctness of the circuits and solderings is the right one. Sending another key $K' \in \{K^0_{\mathtt{id}_1}, K^1_{\mathtt{id}_1}\}$ would break authenticity. Security against E follows from obliviousness.

The following results is immediate by inductively applying the above lemmas.

Lemma 10 (robustness without recover)**.** If G is corrupt and E is honest and the protocol does not abort and does not call RECOVER, then the following holds except with negligible probability. For each input gate $\mathtt{id}$ the evaluator holds $K_{\mathtt{id}} = K^{x_i}_{\mathtt{id}}$. For E-in-gates x_i is the correct input of E. Furthermore, for each output gate $\mathtt{id}'$, the evaluator E holds $K_{\mathtt{id}'} = K^{y_i}_{\mathtt{id}'}$ where y_i is the

plaintext value obtained by evaluating $\mathcal{C}$ in plaintext on the input values $x_{\mathtt{id}}$. Furthermore, the probability that the protocol aborts is independent of the inputs of E.

Notice that it might not be the case that whether or not Recover is called is independent of the inputs of E. It is in fact easy for a corrupt G to construct garblings for which we go into recovery mode if and only if some internal wire has plaintext value 1. This is called a selective attack.

Recovery

The description of the above procedures for evaluation and output assume that Recover has not been called. We now describe what happens when Recover is called. Recall that Recover is called in EvKAs when that procedure learns both the 0-key $K^0_{\mathtt{id}}$ and the 1-key $K^1_{\mathtt{id}}$ for some wire $\mathtt{id}$. The issue is that the procedure then cannot know which key is the right one, as it does not know the plaintext value $x_{\mathtt{id}}$ nor does it know which key is the 0-key.

When the procedure is called, it is called as Recover$(\mathtt{id}, \Delta)$, where $\mathtt{id.ka}.1, \ldots, \mathtt{id.ka}.\alpha$ are the identifier of key authenticators that have been produced using PreprocessKA or PreprocessInKA. Furthermore, Δ is the difference for the key authenticator $\mathtt{id.ka}.1$

Let $\mathcal{E}$ be the set of identifiers $\mathtt{id}$ for which $\mathtt{id}$ is an in-gate. This implies that if $\mathtt{id} \in \mathcal{E}$, then $\mathtt{id.ka}.1, \ldots, \mathtt{id.ka}.\alpha_{\mathtt{inka}}$ are identifiers of input key authenticators. By construction a majority of them are correct except with negligible probability.

Let $\mathcal{I}$ be the set of identifiers $\mathtt{id}$ with which Recover could be called but which are not in $\mathcal{E}$, these are internal wires (non-input wires) with associated KA buckets. This implies that if $\mathtt{id} \in \mathcal{I}$, then $\mathtt{id.ka}.1, \ldots, \mathtt{id.ka}.\alpha_{\mathtt{inka}}$ are identifiers of key authenticators. By construction a majority of them are correct except with negligible probability.

As a simple motivating example assume that recover is called with an identifier from $\mathcal{E}$. The following procedure shows how this allows to recover the plaintext value $x_{\mathtt{id}}$.

function RecoverInputBit$(\mathtt{id}, \Delta)$ ▷ $\Delta = \Delta_{\mathtt{id.ka}.1}$
  $\mathtt{id}' \leftarrow \mathtt{id.ka}$
  $\Delta_1 \leftarrow \Delta$
  $\alpha \leftarrow \alpha_{\mathtt{inka}}$
  $\forall^{\alpha}_{j=1}$ **retrieve** $K_j \leftarrow K_{\mathtt{id}'.j}$;
  $\forall^{\alpha}_{j=2}$ **retrieve** $\Delta_{\mathtt{id}'.1,\mathtt{id}'.j}$; $\Delta_j \leftarrow \Delta_{\mathtt{id}'.1,\mathtt{id}'.j} \oplus \Delta_1$
  if $\#\{j \in \{1, \ldots, \alpha\} \mid H(\Delta_j) = K_j\} > \alpha/2$ **then**
    $x_{\mathtt{id}} \leftarrow 0$
  else $x_{\mathtt{id}} \leftarrow 1$
  end if
end function

Lemma 11. If G is corrupt and E is honest, the following holds except with negligible probability. If $K_{\mathtt{id}} = K^b_{\mathtt{id}}$ and $\Delta = \Delta_{\mathtt{id.ka}.1}$, then after an execution of RecoverInputBit$(\mathtt{id}, \Delta)$ it holds that $x_{\mathtt{id}} = b$.

Proof. The following statements hold except with negligible probability. By premise there exists b such that $K_j = K^b_j$ for all j. By Lemma 3 it holds that $\Delta_j = \Delta_{\mathtt{id}'.j}$ for all j. This implies that for the majority of correct input key authenticators it holds that $H(\Delta_j) = K_j$ if and only if $K_j = K^0_j$. Hence, if $b = 0$, it will hold for a majority of j that $H(\Delta_j) = K_j$, and if $b = 1$ it will hold for a majority of j that $H(\Delta_j) \neq K_j$. □

Consider then the case where Recover is called with $\mathtt{id} \notin \mathcal{E}$. We want to ensure that if G is caught cheating, then E can recover all inputs of G and then evaluate $\mathcal{C}$ in plaintext. For this we only have to ensure that if Recover is called with $\mathtt{id} \in \mathcal{I}$, then it can call RecoverInputBit with all $\mathtt{id} \in \mathcal{E}$. To facilitate this, we are going to create on special wire rco and solder all $\mathtt{id} \in \mathcal{I}$ onto rco and solder rco onto all $\mathtt{id} \in \mathcal{E}$. For security reasons we do not do full solderings, we only release the difference between the Δ values. That way, if we learn the difference for any $\mathtt{id} \in \mathcal{I}$, we can learn the difference for all $\mathtt{id} \in \mathcal{E}$, and we are done.

The following procedure will be run together with all the other preprocessing protocols.

function PreProcessRecovery
  G: $\Delta_{\mathtt{rco}} \leftarrow \{0,1\}^\kappa$
  G: **Commit**$(\mathtt{rco}, \Delta_{\mathtt{rco}})$
  G: $\forall \mathtt{id} \in \mathcal{I} \cup \mathcal{E}$: Open$(\mathtt{id.ka}.1.\mathtt{dif}, \mathtt{rco})$
  E: $\forall \mathtt{id} \in \mathcal{I} \cup \mathcal{E}$: **receive** $\Delta_{\mathtt{id,rco}}$
  G: $\forall \mathtt{id} \in \mathcal{E}$: Open$(\mathtt{rco}, \mathtt{id.ka}.1.\mathtt{dif})$
  E: $\forall \mathtt{id} \in \mathcal{E}$: **receive** $\Delta_{\mathtt{rco,id}}$
end function

The following procedure uses the Δ-solderings to recover all the inputs.

function RecoverInputBits$(\mathtt{id}, \Delta)$ ▷ $\Delta = \Delta_{\mathtt{id.ka}.1}$
  retrieve $\Delta_{\mathtt{id,eecov}}$
  $\Delta_{\mathtt{rco}} \leftarrow \Delta_{\mathtt{id,rco}} \oplus \Delta$
  for all $\mathtt{id}' \in \mathcal{E}$ **do**
    retrieve $\Delta_{\mathtt{rco,id}'}$
    $\Delta_{\mathtt{id}'} \leftarrow \Delta_{\mathtt{rco,id}'} \oplus \Delta_{\mathtt{rco}}$
    RecoverInputBit$(\mathtt{id}', \Delta_{\mathtt{id}'})$
  end for
end function

The following result is immediate.

Lemma 12. If G is corrupt and E is honest, the following holds except with negligible probability. If $K_{\mathtt{id}'} = K^{b_{\mathtt{id}'}}_{\mathtt{id}'}$ for all input gates $\mathtt{id}'$ and $\Delta = \Delta_{\mathtt{id.ka}.1}$,

then after an execution of RecoverInputBits($\mathtt{id}, \Delta$) it holds that $x_{\mathtt{id}'} = b_{\mathtt{id}'}$ for all input gates $\mathtt{id}'$.

Notice that RecoverInputBits not only computes $x_{\mathtt{id}}$ for all input gates. It can also compute the key $K^0_{\mathtt{id}}$ and the difference $\Delta_{\mathtt{id}}$. From the inputs $x_{\mathtt{id}}$ it can compute the correct plaintext value $x_{\mathtt{id}}$ for *all* wires $\mathtt{id}$. From the keys $K^0_{\mathtt{id}}$ and the differences $\Delta_{\mathtt{id}}$ it can use Gb iteratively to compute also the correct key $K^0_{\mathtt{id}}$ and the correct difference $\Delta_{\mathtt{id}}$ for all sub-sequence wires $\mathtt{id}$, as it has all the information it needs to compute the garblings the way G ought to have done it if it started from the computed inputs keys and differences. This will in particular allow E to compute for each G-output gate $\mathtt{id}$ the plaintext output $x_{\mathtt{id}}$ and the key $K_{\mathtt{id}} = K^0_{\mathtt{id}} \oplus x_{\mathtt{id}}\Delta_{\mathtt{id}}$. This is how the outputs will be computed in recovery mode.

function RECOVER($\mathtt{id}, \Delta$) ▷ $\Delta = \Delta_{\mathtt{id.ka}.1}$
  RecoverInputBits($\mathtt{id}, \Delta$)
  go to `recovery mode`
end function
function OutputG($\mathtt{id}$) ▷ In `recovery mode`
  E: **retrieve** soldering $(\mathtt{id}_1, \mathtt{id.out}.1)$ from C
  E: **retrieve** $x_{\mathtt{id}_1}$
  E: **retrieve** $K^0_{\mathtt{id}_1}$
  E: **retrieve** $\Delta^0_{\mathtt{id}_1}$
  E: P_s $K^{x_{\mathtt{id}_1}}_{\mathtt{id}_1}$
end function
function OutputE($\mathtt{id}$) ▷ In `recovery mode`
  E: **retrieve** soldering $(\mathtt{id}_1, \mathtt{id.out}.1)$ from C.
  E: **retrieve** $x_{\mathtt{id}_1}$.
  E: $y_{\mathtt{id}} \leftarrow x_{\mathtt{id}_1}$.
end function

The following result follows from the above discussion.

Lemma 13 (robustness with recover)**.** If G is corrupt and E is honest and the protocol calls Recover, then the following holds except with negligible probability. For each input gate $\mathtt{id}$ the evaluator holds $K_{\mathtt{id}} = K^{x_i}_{\mathtt{id}}$. For E-in-gates x_i is the correct input of E. Furthermore, for each output gate $\mathtt{id}'$, the evaluator E holds $K_{\mathtt{id}'} = K^{y_i}_{\mathtt{id}'}$ where y_i is the plaintext value obtained by evaluating $\mathcal{C}$ in plaintext on the input values $x_{\mathtt{id}}$. Furthermore, the probability that the protocol aborts is independent of the inputs of E.

By combining Lemma 10 and Lemma 13 we get:

Theorem 5 (robustness)**.** If G is corrupt and E is honest and the protocol does not abort, then the following holds except with negligible probability. For each input gate $\mathtt{id}$ the evaluator holds $K_{\mathtt{id}} = K^{x_i}_{\mathtt{id}}$. For E-in-gates x_i is the correct input of E. Furthermore, for each output gate $\mathtt{id}'$, the evaluator

E holds $K_{\mathtt{id}'} = K_{\mathtt{id}'}^{y_i}$ where y_i is the plaintext value obtained by evaluating $\mathcal{C}$ in plaintext on the input values $x_{\mathtt{id}}$. Furthermore, the probability that the protocol aborts is independent of the inputs of E.

Recall the issue with the selective attack that a corrupt G can ensure that RECOVER is called based on for instance the value of an internal wire. We now see that this is handled by making sure that in recovery mode, we return the exact same values to G as we would when we are not in recovery mode,

5.9 Analysis

Our proof follows the proofs in [NST17] and [AHMR15] closely. Redoing the proofs in the full, glorious detail in the UC model would have us reiterate much of the proofs in these papers. We will instead sketch the overall structure of the formal proof and point to [NST17] and [AHMR15] for the details. We realize that this means that only the reader which is familiar with the UC model and and the mentioned papers may completely verify the proofs. However, since the UC model and our the proof techniques are standard by now we find this a reasonable level of proof detail.

Corrupt G

We first prove security for the case where G is corrupt and E is honest.

Without going into the details of the UC framework, let us just recall that the proof tasks as usual are as follows. When G is corrupted and E is honest, we should present a poly-time simulator $\mathcal{S}$. It plays the role of E in the protocol. But as opposed to E it is *not* being given the inputs x_{E} of E. Instead it has access to an oracle $\mathcal{O}_{x_{\mathsf{E}}}(\cdot)$ containing x_{E}. The simulator might once supply a possible set of inputs x_{G} of G to the oracle and learn the outputs $y_{\mathsf{G}} = \mathcal{O}_{x_{\mathsf{E}}}(x_{\mathsf{G}})$ that G would have if $\mathcal{C}$ was evaluated on the x_{E} in the oracle and the provided x_{G}.

The simulator $\mathcal{S}$ proceeds as follows. It runs as the honest E would do, but with two modifications. 1) It uses dummy inputs $x_{\mathtt{id}} = 0$ for all E-in-gates and 2) there is a modification in OUTPUTG, which we describe below.

For all G-in-gates $\mathtt{id}$ it inspects the commitment functionality and learns $K_{\mathtt{id}}^0$ and $\Delta_{\mathtt{id}}$ and computes $K_{\mathtt{id}}^1$ from these. From all G-in-gates it can then retrieve $K_{\mathtt{id}} = K_{\mathtt{id}}^{x_{\mathtt{id}}}$ which is well defined by Theorem 5. This defined the inputs $x_{\mathtt{id}}$ of G. Then it calls its oracle with those input bits of G and for all G-out-gates $\mathtt{id}$ it receives from the oracle $\mathcal{O}$ the output $y_{\mathtt{id}}$ obtained by running $\mathcal{C}$ in plaintext with those inputs $x_{\mathtt{id}}$ and the unknown input bits of E. If the protocol aborts, then the simulator aborts too. If the protocol reaches an execution of OUTPUTG, then the simulator will send the key $K_{\mathtt{id}_1}^{x_{\mathtt{id}_1}}$ computed as in recovery mode. Note that it can do this as it knows all the keys and therefore can compute $K_{\mathtt{id}_1}^0$ and $K_{\mathtt{id}_1}^1$ exactly as in recovery mode, and it was

given $y_{\mathtt{id}}$ from the oracle. It follows directly from Theorem 5 that the real protocol and the simulation aborts with the same probability and that when they do not abort, then the key returned to G from the simulator is the same that the honest E would have sent, except with negligible probability.

Corrupt E

We then prove security for the case where G is honest and E is corrupt.

Without going into the details of the UC framework, let us just recall that the proof tasks as usual are as follows. When E is corrupted and G is honest, we should present a poly-time simulator $\mathcal{S}$. It plays the role of G in the protocol. But as opposed to G it is *not* being given the input x_{G} of G. Instead it has access to an oracle $\mathcal{O}_{x_{\mathsf{G}}}(\cdot)$ containing x_{G}. The simulator might once supply a possible input x_{E} of E to the oracle and learn the outputs $y_{\mathsf{E}} = \mathcal{O}_{x_{\mathsf{G}}}(x_{\mathsf{E}})$ that E would have if $\mathcal{C}$ was evaluated on the x_{G} in the oracle and the provided x_{E}.

The simulator $\mathcal{S}$ proceeds as follows. It runs as the honest G would do, but with two modifications. 1) It uses dummy inputs $x'_{\mathtt{id}} = 0$ for all G-in-gates and 2) there is a modification in OutputE, which we describe below.

For all E-in-gates $\mathtt{id}$ it inspects the commitment functionality and learns $K^0_{\mathtt{id}}$ and $\Delta_{\mathtt{id}}$ and computes $K^1_{\mathtt{id}}$ from these. From all E-in-gates it can then retrieve $K_{\mathtt{id}} = K^{x_{\mathtt{id}}}_{\mathtt{id}}$ which is well defined as G is honest and therefore followed the protocol. This defined the inputs $x_{\mathtt{id}}$ of E. Then it calls its oracle with those input bits of E and for all E-out-gates $\mathtt{id}$ it receives from the oracle the output $y_{\mathtt{id}}$ obtained by running $\mathcal{C}$ in plaintext with those inputs $x_{\mathtt{id}}$ and the unknown input bits of G.

If the protocol aborts, then the simulator aborts too.

If the protocol reaches an execution of OutputE, then the simulator will retrieve the output $y_{\mathtt{id}}$ learned from $\mathcal{O}_{x_{\mathsf{G}}}(x_{\mathsf{E}})$. Then it retrieves the key $K_{\mathtt{id}_1}$ and computes

$$\sigma'_{\mathtt{id}_1} = \mathsf{lsb}(K_{\mathtt{id}}) \oplus y_{\mathtt{id}} \ .$$

Then it runs OutputE as in the protocol. To understand why we make this change, recall that $\mathcal{S}$ ran G with dummy inputs, so it might be the case that $K_{\mathtt{id}.1}$ encodes a different output than $y_{\mathtt{id}}$. Therefore, sending $\sigma_{\mathtt{id}.1}$ might result in E getting a wrong output, which would allow it to learn that it is in the simulation.

We should now argue that the value seen by E in the simulation and in the real execution are computationally indistinguishable. Notice that if we ran the simulation but using the real inputs x_{G} for G instead of dummy inputs, then in all executions of OutputE it would be the case that $\sigma_{\mathtt{id}_1} = \mathsf{lsb}(K_{\mathtt{id}}) \oplus y_{\mathtt{id}} = \sigma_{\mathtt{id}_1}$ for all E-out-gates. Therefore there is not really a modification of the commitment functionality when OutputE is simulated. Therefore, the simulation run with real inputs is just the real protocol. This means that it is

sufficient to argue that the values seen by E in the simulation with dummy input x'_{G} and with real input x_{G} are computationally indistinguishable.

We do that via a reduction to obliviousness of the reactive garbling scheme and the fact that H is a random oracle. The reduction will have access to the real input of x_{G} and an oracle producing garbled circuits along with encodings of either dummy inputs or the real inputs. It will then augment these to make them look like a run of the simulation with dummy inputs or real inputs.

In a bit more detail, in the obliviousness game in [NR16] we will have access to an oracle $\mathcal{O}_b$ for a uniformly random bit b. We can give $\mathcal{O}_b$ a command of the form $(\mathsf{garble}, \mathtt{id}, C)$. Then it garbles C and gives us $F_{\mathtt{id}}$, keeping the keys secret. If we later give the command $(\mathsf{reveal}, \mathtt{id})$, then we are given all keys used in garbling $F_{\mathtt{id}}$. We can also give the command $(\mathsf{link}, \mathtt{id}_1, i_1, \mathtt{id}_2, i_2)$. The oracle will release the information used to solder output wire i_1 in $F_{\mathtt{id}_1}$ onto input wire i_2 in $F_{\mathtt{id}_2}$ as specified in GenSold. We can also give the command $(\mathsf{input}, \mathtt{id}, i, x_0, x_1)$. The oracle will compute the keys K^0 and K^1 for input wire i in $F_{\mathtt{id}}$. It then gives us K^{x_b}. There are some natural restrictions. There is not allowed to be any loops in linking wires between circuits. No identifier is allowed to occur in both a reveal and a link command. No wire $(\mathtt{id}_2, i_2)$ is allow to occur in two different call to $(\mathsf{input}, \mathtt{id}_2, i_2, \cdot, \cdot)$ or two calls to $(\mathsf{link}, \cdot, \cdot, \mathtt{id}_2, i_2)$. No input wire is allowed to occur in both a call $(\mathsf{input}, \mathtt{id}_2, i_2, \cdot, \cdot)$ and a call $(\mathsf{link}, \cdot, \cdot, \mathtt{id}_2, i_2)$. At the end we have to make a guess at b. The security definition says that no poly-time adversary can guess b with better than negligible probability. The definition of authenticity say that the adversary cannot for some wire $\mathtt{id}$ compute both $K^0_{\mathtt{id}}$ and $K^1_{\mathtt{id}}$ given the information it receives in the game.

In fact, the definition in [NR16] does not have the reveal command. We add this command here, getting a notion of adaptive, reactive garbling. It is straight forward to verify that the scheme in [AHMR15] is an adaptive, reactive garbling scheme in the above sense. This is achieved by using the strong programmable random oracle model.

We need to add an additional command $(\mathsf{difdif}, \mathtt{id}_1, \mathtt{id}_2)$. In response to this the oracle will release $\Delta_{\mathtt{id}_1} \oplus \Delta_{\mathtt{id}_2}$, where $\Delta_{\mathtt{id}_i}$ is the difference used to garble $F_{\mathtt{id}_i}$. No identifier is allowed to occur in both a difdif and a reveal command, *i.e.*, one is only allowed to learn $\Delta_{\mathtt{id}_1} \oplus \Delta_{\mathtt{id}_2}$ when both $\Delta_{\mathtt{id}_1}$ and $\Delta_{\mathtt{id}_2}$ are unknown. It is straight forward to verify that the scheme in [AHMR15] is an adaptive, reactive garbling scheme even when this command is added. The scheme would trivially still be secure if the same Δ was used on two unrevealed garblings, *i.e.*, if $\Delta_{\mathtt{id}_1} = \Delta$ and $\Delta_{\mathtt{id}_2} = \Delta$ for random Δ. To see this, consider garbling two circuits C_1 and C_2 by garbling the circuit $C_1 \| C_2$ which runs the two circuits in parallel. This would exactly provide garblings of C_1 and C_2 with a common Δ. It is straight forward to go through the proof of [AHMR15] and verify that first garbling using independent $\Delta_{\mathtt{id}_1}$ and $\Delta_{\mathtt{id}_2}$ and then releasing $\Delta_{\mathtt{id}_1} \oplus \Delta_{\mathtt{id}_2}$ does not break the security. To see this note that in the proof, when $\mathtt{id}_1$ and $\mathtt{id}_2$ are unrevealed, then the distribution of

$\Delta_{\mathtt{id}_i}$ is statistically close to uniform, except that $\mathsf{lsb}(\Delta_{\mathtt{id}_i}) = 1$. Furthermore, the security depends only on $\Delta_{\mathtt{id}_i}$ being statistically close to uniform, not that the $\Delta_{\mathtt{id}_i}$ are independent, which is what allows that $\Delta_{\mathtt{id}_1} = \Delta_{\mathtt{id}_2}$. This means we can simulate $\Delta_{\mathtt{id}_1} \oplus \Delta_{\mathtt{id}_2}$ by a uniformly random value Δ with $\mathsf{lsb}(\Delta) = 0$, as it is still ensured that each $\Delta_{\mathtt{id}_i}$ has full entropy given Δ.

We will go over each of the sub-protocols and argue how to simulate the values sent to E.

When the simulation is run with dummy inputs for G, we call it the *dummy mode*. When the simulation is run with real inputs for G, we call it the *real mode*.

When we say that a protocol is trivial to simulate it means that the protocol sends no values to E, so there are no values to simulate.

$\triangleright\, \mathsf{GenWire}(\mathtt{id}, K, \Delta)$: When this sub-simulator is called the key material $(K^0_{\mathtt{id}}, \Delta_{\mathtt{id}})$ is already defined and sitting inside $\mathcal{O}$ as part of some circuit. This also defines $K^1_{\mathtt{id}}$ and $\sigma_{\mathtt{id}}$. The simulator is not given these values. We then simulate the commitments by sending E notifications that the commitments were done. These value are clearly the same in the dummy and the real mode. Notice that by this way of simulating, it holds that for each wire $\mathtt{id}$ which has associated key material in the protocol, the oracle $\mathcal{O}_b$ will hold this key material, so we can work on it using the interface of $\mathcal{O}_b$.

$\triangleright\, \mathsf{VerWire}(\mathtt{id})$: Here we send all the key material to E. The simulator will ask its oracle $\mathcal{O}_b$ to reveal the circuit that $(K^0_{\mathtt{id}}, \Delta_{\mathtt{id}})$ is part of. This will give it the needed key material. Then it patches the commitment functionality to open to those values before decommitting.

$\triangleright\, \mathsf{GenSold}(\mathtt{id}_1, \mathtt{id}_2)$: The values sent to E here are exactly the soldering values of the reactive garbling scheme [AHMR15]. The simulator can therefore request to get the values from its oracle.

$\triangleright\, \mathsf{EvSold}(\mathtt{id}_1, \mathtt{id}_2, K)$: Trivial.

$\triangleright\, \mathsf{GenKeyAuth}(\mathtt{id})$: The simulator will first ask its oracle $\mathcal{O}_b$ to make a garbling of a circuit with one input wire and one output wire and one gate which is the identity gate. Such a garbling consist simply of $K^0_{\mathtt{id}}$ and $\Delta_{\mathtt{id}}$. So now the key material $(K^0_{\mathtt{id}}, \Delta_{\mathtt{id}})$ is defined and sitting inside $\mathcal{O}$ as part of some circuit. We call the simulator for GenWire on these. We additionally have to simulate the value $A_{\mathtt{id}} = \{H(K^0_{\mathtt{id}}), H(K^1_{\mathtt{id}})\}$. This is complicated by the fact that $\mathcal{S}$ does not know $K^0_{\mathtt{id}}$ or $K^1_{\mathtt{id}}$, as the key material is sitting inside $\mathcal{O}_b$. Recall, however, that we assume a programmable random oracle. We can therefore simply sample two uniformly random value $h, h' \leftarrow \{0,1\}^\kappa$ and let $A_{\mathtt{id}} = \{h, h'\}$. If E later queries H in an unrevealed key authenticator on its key $K_{\mathtt{id}} \in \{K^0_{\mathtt{id}}, K^1_{\mathtt{id}}\}$, then we return h. Except with negligible probability it will never query on the other key, as this would break authenticity. Notice that we did not pick whether $H(K^0_{\mathtt{id}}) = h$ or $H(K^0_{\mathtt{id}}) = h'$. We do not need to do this as $A_{\mathtt{id}}$ is sent as a set. This is important as we do not know the value of $K_{\mathtt{id}}$.

▷ VerKeyAuth($\mathtt{id}$): Here we first simulate VerWire and learn $K^0_{\mathtt{id}}$ and $K^1_{\mathtt{id}}$. Then define $H(K^0_{\mathtt{id}}) := h$ or $H(K^1_{\mathtt{id}}) := h'$. If the oracle had been called on $K^0_{\mathtt{id}}$ or $K^1_{\mathtt{id}}$ before, this might give an inconsistent simulation. However, if the oracle had been called on $K^0_{\mathtt{id}}$ or $K^1_{\mathtt{id}}$ before, then E guessed one of the keys without being given any information about the key. Since the keys are uniformly random this happens with negligible probability.

▷ GenInKeyAuth($\mathtt{id}$): This protocol is simulated as GenKeyAuth, but we additionally have to ensure that $H(\Delta_{\mathtt{id}}) = K^0_{\mathtt{id}}$. The values $\Delta_{\mathtt{id}}$ and $K^1_{\mathtt{id}}$ are not known to $\mathcal{S}$, as they are sitting inside $\mathcal{O}_b$. But we can still define that $H(\Delta_{\mathtt{id}}) = K^0_{\mathtt{id}}$. We will only have to return $K^0_{\mathtt{id}}$ if H is ever evaluated on $H(\Delta_{\mathtt{id}})$. If $\mathtt{id}$ is never verified, this would involve E guessing $\Delta_{\mathtt{id}}$ after being given only one key, which would break authenticity of the garbling scheme. For the case where $\mathtt{id}$ is verified, see below.

▷ VerInKeyAuth($\mathtt{id}$): Simulated as VerInKeyAuth. This lets $\mathcal{S}$ learn $K^0_{\mathtt{id}}$ and $\Delta_{\mathtt{id}}$. Then it programs H to $H(\Delta_{\mathtt{id}}) = K^0_{\mathtt{id}}$. If the oracle had been called on $\Delta_{\mathtt{id}}$ before, this might give an inconsistent simulation. However, if the oracle had been called on $\Delta_{\mathtt{id}}$ before the wire it verified, then E guessed this value without being given any information about $\Delta_{\mathtt{id}}$. Since $\Delta_{\mathtt{id}}$ has $\kappa - 1$ bits of min-entropy this happens with negligible probability.

▷ PreProcessKA(): The protocol is simulated by running honestly and simulating all the calls to GenKeyAuth, VerKeyAuth and GenSold. No additional values are sent.

▷ EvKAs($\mathtt{id}, \mathcal{K}_{\mathtt{id}}$): Trivial.

▷ InputG($\mathtt{id}$): Let $x'_{\mathtt{id}} = 0$ be the input bit of G in dummy mode and let $x_{\mathtt{id}}$ be the input bit of G in real mode. Ask the oracle $\mathcal{O}_b$ to get the encoded input for $\mathtt{id.ka}.1$. Submit the bit-pair $(x'_{\mathtt{id}}, x_{\mathtt{id}})$. The simulator learns $K = K^x_{\mathtt{id.ka}.1}$, where $x = x'_{\mathtt{id}}$ if $b = 0$ and $x = x_{\mathtt{id}}$ if $b = 1$. Send K to E. The distribution of K is exactly as in dummy mode when $b = 0$ and exactly as in real mode when $b = 1$.

▷ PreProcessOTInit(): It was shown in [NST17] how to simulate this protocol. This can be done without knowing $\Delta_{\mathtt{ot}}$. This is important as we need to implicitly define $\Delta_{\mathtt{ot}}$ below.

▷ InputE($\mathtt{id}$): When running this protocol the simulator will pick $R_{b_{\mathtt{ot}_{\mathtt{id}}}}$ uniformly a random and output this to E from the OT. This is needed as the simulator does not know $\Delta_{\mathtt{ot}}$. Now E receives the values D, $\sigma_{\mathtt{id}}$ an $S_{\mathtt{id}}$ along with some notification values there are trivial to simulate. It is therefore enough to show how to simulate these three values. Let $b_{\mathtt{ot}_{\mathtt{id}}}$ be the choice bit of E in the call to the OT. The simulator can inspect the ideal functionality and learn this value. Let $f_{\mathtt{id}}$ be the bit sent by E. Let $x_{\mathtt{id}} \leftarrow f_{\mathtt{id}} \oplus b_{\mathtt{ot}_{\mathtt{id}}}$. Define this to be the input of E. Ask the oracle $\mathcal{O}_b$ to get the encoded input for $\mathtt{id}'$. Submit the bit-pair $(x_{\mathtt{id}}, x_{\mathtt{id}})$. The simulator learns $K = K^{x_{\mathtt{id}}}_{\mathtt{id}'}$. By definition

we have that $\mathsf{lsb}(K_{\mathtt{id}'}) = x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}}$, which allows to simulate $\sigma_{\mathtt{id}}$ as

$$\sigma_{\mathtt{id}} = \mathsf{lsb}(K_{\mathtt{id}'}) \oplus x_{\mathtt{id}} \ ,$$

as the simulator knows $K_{\mathtt{id}'}$ and $x_{\mathtt{id}}$. To simulate $S_{\mathtt{id}}$ we send a uniformly random value. This will implicitly define $\Delta_{\mathtt{ot}}$ by $\Delta_{\mathtt{ot}} = S_{\mathtt{id}} \oplus \Delta_{\mathtt{id}'}$, where the simulator does not know $\Delta_{\mathtt{id}'}$ as it is sitting inside $\mathcal{O}$. A little more care must be done. The above simulation of $S_{\mathtt{id}}$ works for the first call to InputG. For the next call (with identifier $\widehat{\mathtt{id}}$ say) we should ensure that $\widehat{\Delta}_{\mathtt{ot}} = S_{\widehat{\mathtt{id}}} \oplus \Delta_{\widehat{\mathtt{id}}'}$ becomes defined to the same $\Delta_{\mathtt{ot}}$ as when we simulated for $\mathtt{id}$. This means that we should ensure that $S_{\mathtt{id}} \oplus \Delta_{\mathtt{id}'} = S_{\widehat{\mathtt{id}}} \oplus \Delta_{\widehat{\mathtt{id}}'}$, which is equivalent to $S_{\widehat{\mathtt{id}}} = S_{\mathtt{id}} \oplus \Delta_{\mathtt{id}'} \oplus \Delta_{\widehat{\mathtt{id}}'}$. To ensure this the simulator ask $\mathcal{O}_b$ for $\Delta_{\mathtt{id}'} \oplus \Delta_{\widehat{\mathtt{id}}'}$ using the difdif-command and computes $S_{\widehat{\mathtt{id}}}$ as required. By inspection of the protocol we see that $K = D \oplus R_{b_{\mathtt{ot}_{\mathtt{id}}}} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})S_{\mathtt{id}}$. This allows to simulate D as

$$D = K \oplus R_{b_{\mathtt{ot}_{\mathtt{id}}}} \oplus (x_{\mathtt{id}} \oplus \sigma_{\mathtt{id}})S_{\mathtt{id}} \ ,$$

as the simulator can compute K, $R_{b_{\mathtt{ot}_{\mathtt{id}}}}$, $x_{\mathtt{id}}$, $\sigma_{\mathtt{id}}$ and $S_{\mathtt{id}}$ at the time where it needs to send D. It compute K as $K = K_{\mathtt{id}'}$. It picked $R_{b_{\mathtt{ot}_{\mathtt{id}}}}$ itself. It already computed $x_{\mathtt{id}}$ and $\sigma_{\mathtt{id}}$ and $S_{\mathtt{id}}$.

▷ $\mathsf{GenSub}(\mathtt{id}, C)$: We simulate by asking $\mathcal{O}_b$ to generate and give us $F_{\mathtt{id}}$. This also defines key materials $(K_i, \Delta_{\mathtt{id}})_{i=1}^n$ and $(L_i, \Delta_{\mathtt{id}})_{i=1}^m$ sitting inside $\mathcal{O}$. We call the simulator for GenWire to simulate these.

▷ $\mathsf{VerSub}(\mathtt{id}, C)$: We simulate by asking $\mathcal{O}_b$ to reveal $F_{\mathtt{id}}$. This gives us the key materials $(K_i, \Delta_{\mathtt{id}})_{i=1}^n$ and $(L_i, \Delta_{\mathtt{id}})_{i=1}^m$ sitting inside $\mathcal{O}$. We call the simulator for VerWire with these.

▷ $\mathsf{EvSub}(\mathtt{id})$: Trivial.

▷ $\mathsf{PreProcessSub}()$: This protocol including the call to GenSoldSub is simulated by simulating the calls to GenSub, VerSub and GenSold as described above. No further messages are sent.

▷ $\mathsf{AssembleSubs}()$: This protocol is simulated by simulating the calls to GenSold as described above. No further messages are sent.

▷ $\mathsf{EvSubs}(\mathtt{id})$: Trivial.

▷ $\mathsf{OutputE}(\mathtt{id})$: Recall that here the simulator will retrieve the output $y_{\mathtt{id}}$ learned from $\mathcal{O}_{x_{\mathsf{G}}}(x_{\mathsf{E}})$ and the key $K_{\mathtt{id}}$ held by E and send $\sigma'_{\mathtt{id}_1} = \mathsf{lsb}(K_{\mathtt{id}}) \oplus y_{\mathtt{id}}$ to E. We do the same in the reduction.

▷ $\mathsf{OutputG}(\mathtt{id})$: Trivial.

▷ $\mathsf{RecoverInputBit}$: Trivial.

▷ $\mathsf{PreProcessRecovery}$: Here we have to give away all the differences $\Delta_{\mathtt{rco},\mathtt{id}}$ and $\Delta_{\mathtt{id},\mathtt{rco}}$.

Consider the first $\mathtt{id} \in \mathcal{E} \cup \mathcal{I}$. We need to compute $\Delta_{\mathtt{id},\mathtt{rco}} = \Delta_{\mathtt{id.ka.1}} \oplus \Delta_{\mathtt{rco}}$ for an unknown $\Delta_{\mathtt{id.ka.1}}$. We do this by picking $\Delta_{\mathtt{id},\mathtt{rco}}$ uniformly at random and defining

$$\Delta_{\mathtt{rco}} = \Delta_{\mathtt{id.ka.1}} \oplus \Delta_{\mathtt{id},\mathtt{rco}} \ .$$

This is possible as we never have to release $\Delta_{\mathtt{rco}}$ so we do not need to know it. Consider the remaining $\widehat{\mathtt{id}} \in \mathcal{E} \cup \mathcal{I}$. We need to compute

$$\begin{aligned}\Delta_{\widehat{\mathtt{id}},\mathtt{rco}} &= \Delta_{\widehat{\mathtt{id}}.\mathtt{ka}.1} \oplus \Delta_{\mathtt{rco}} \\ &= \Delta_{\widehat{\mathtt{id}}.\mathtt{ka}.1} \oplus \Delta_{\mathtt{id}.\mathtt{ka}.1} \oplus \Delta_{\mathtt{id},\mathtt{rco}} \ .\end{aligned}$$

We can do this as we can get $\Delta_{\widehat{\mathtt{id}}.\mathtt{ka}.1} \oplus \Delta_{\mathtt{id}.\mathtt{ka}.1}$ using the DIFDIF-command and we know $\Delta_{\mathtt{id},\mathtt{rco}}$. We compute the values $\Delta_{\mathtt{rco},\mathtt{id}}$ similarly.

▷ RecoverInputBits: Trivial.

It now follows that if $b = 0$, then the distribution of the reduction is exactly that of the dummy model and if $b = 1$, then the distribution of the reduction is exactly that of the real model. Since we only allowed queries to $\mathcal{O}_b$, it follows that the dummy mode and the real are computationally indistinguishable.

Bibliography

[AHMR15] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 702–729. Springer, April 2015. 23, 87, 107, 125, 127, 128

[AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 137–156. Springer, February 2007. 13, 46

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013. 66

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 673–701. Springer, April 2015. 16, 41, 55, 66, 69

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 387–404. Springer, May 2014. 7, 46, 81

[AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 681–698. Springer, December 2012. 77

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, 1988. 78

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, February 2009. 4, 45

[BCPV13] Olivier Blazy, Céline Chevalier, David Pointcheval, and Damien Vergnaud. Analysis and improvement of Lindell's UC-secure commitment schemes. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 2013*, volume 7954 of *LNCS*, pages 534–551. Springer, June 2013. 22, 25, 39, 40, 41

[Bea92] Donald Beaver. Foundations of secure interactive computing. In Joan Feigenbaum, editor, *CRYPTO 1991*, volume 576 of *LNCS*, pages 377–391. Springer, August 1992. 12

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *CRYPTO 1995*, volume 963 of *LNCS*, pages 97–109. Springer, August 1995. 51, 54

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC 1996*, pages 479–488. ACM Press, May 1996. 16, 41

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *STOC 1988*, pages 1–10. ACM Press, May 1988. 4, 5, 46

[BHK13] Mihir Bellare, Viet Tung Hoang, and Sriram Keelveedhi. Instantiating random oracles via UCEs. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 398–415. Springer, August 2013. 63, 64

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P 2013*, pages 478–492. IEEE Computer Society Press, May 2013. 68

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, December 2012. 63

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012. 45, 63, 106

[BLJ+17] Azer Bestavros, Andrei Lapets, Frederick Jansen, Mayank Varia, Nikolaj Volgushev, and Malte Schwarzkopf. *Design and Deployment of Usable, Scalable MPC*, 2017. `https://www.cs.bu.edu/fac/best/res/papers/tpmpc17.pdf` [Accessed: May 17th 2017]. 4

[BLN+15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multiparty computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. `https://eprint.iacr.org/2015/472`. 5, 9, 47, 52, 55, 56, 82

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, October 2008. 45

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC 1990*, pages 503–513. ACM Press, May 1990. 6, 82, 88

[BP09] Joan Boyar and Rene Peralta. New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191, 2009. `https://eprint.iacr.org/2009/191`. 94

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 1993*, pages 62–73. ACM Press, November 1993. 22, 63

[Bra13] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 441–463. Springer, December 2013. 7, 46, 78, 81

[BRC60] Raj Chandra Bose and Dwijendra K Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and control*, 3(1):68–79, 1960. 65

[Can96] Ran Canetti. *Studies in secure multiparty computation and applications*. PhD thesis, The Weizmann Institute of Science, 1996. 12

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000. 12

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*, pages 136–145. IEEE Computer Society Press, October 2001. 12, 17, 21, 62, 83

[CC06] Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 521–536. Springer, August 2006. 22

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC 1988*, pages 11–19. ACM Press, May 1988. 4, 5, 46

[CDD+15] Ignacio Cascudo, Ivan Damgård, Bernardo Machado David, Irene Giacomelli, Jesper Buus Nielsen, and Roberto Trifiletti. Additively homomorphic UC commitments with optimal amortized overhead. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 495–515. Springer, March / April 2015. 7, 8, 10, 23, 24, 25, 28, 39, 40, 41

[CDD+16] Ignacio Cascudo, Ivan Damgård, Bernardo David, Nico Döttling, and Jesper Buus Nielsen. Rate-1, linear time and additively homomorphic UC commitments. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 179–207. Springer, 2016. 23, 33, 39, 50, 52, 66, 85

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, August 2001. 15, 21

[CHK+12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 416–432. Springer, February / March 2012. 5

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 597–608. ACM Press, November 2014. 22, 25, 39, 40, 41

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, March 2012. 83

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *STOC 1986*, pages 364–369. ACM Press, May 1986. 16

[Cle17] Privacy Rights Clearinghouse. *Data Breaches*, 2017. `https://www.privacyrights.org/data-breaches` [Accessed: May 17th 2017]. 3

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC 2002*, pages 494–503. ACM Press, May 2002. 21

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 40–58. Springer, August 2015. 66, 69

[CvT95] Claude Crépeau, Jeroen van de Graaf, and Alain Tapp. Committed oblivious transfer and private multi-party computation. In Don Coppersmith, editor, *CRYPTO 1995*, volume 963 of *LNCS*, pages 110–123. Springer, August 1995. 23

[DDGN14] Ivan Damgård, Bernardo Machado David, Irene Giacomelli, and Jesper Buus Nielsen. Compact VSS and efficient homomorphic UC commitments. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 213–232. Springer, December 2014. 22

[DG03] Ivan Damgård and Jens Groth. Non-interactive and reusable non-malleable commitment schemes. In *STOC 2003*, pages 426–437. ACM Press, June 2003. 21

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, September 2013. 47, 82

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure

computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 2014*, volume 8642 of *LNCS*, pages 398–415. Springer, September 2014. 47, 82

[DN02] Ivan Damgård and Jesper Buus Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 581–596. Springer, August 2002. 22

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. Gate-scrambling revisited - or: The TinyTable protocol for 2-party secure computation. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, To appear*. Springer, 2017. 47, 48, 49, 82

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, August 2012. 46, 47, 82

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. 94

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015. 6

[Dya] Dyadic Security. `https://www.dyadicsec.com`. 4, 45

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of Boolean circuits using preprocessing. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 621–641. Springer, March 2013. 47, 82

[DZ16] Ivan Damgård and Rasmus Winther Zakarias. Fast oblivious AES A dedicated application of the MiniMac protocol. In *AFRICACRYPT 2016*, volume 9646 of *LNCS*, pages 245–264. Springer, 2016. 47, 48, 49, 82

[Ekl72] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Trans. Computers*, 21(7):801–803, 1972. 66

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 537–556. Springer, May 2013. 7, 8, 22, 23, 50, 53, 81, 84, 88

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In Michel Abdalla and Roberto De Prisco, editors, *SCN 2014*, volume 8642 of *LNCS*, pages 358–379. Springer, September 2014. 7, 46, 47

[FJNT15] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. TinyLEGO: An interactive garbling scheme for maliciously secure two-party computation. Cryptology ePrint Archive, Report 2015/309, 2015. `https://eprint.iacr.org/2015/309`. 8, 10, 23, 50, 52, 53, 57, 60, 61, 62, 63, 64, 85, 91

[FJNT16] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 542–565. Springer, January 2016. 7, 8, 10, 11, 16, 21, 50, 51, 52, 65, 85, 89, 92, 93, 105

[FM16] Pooya Farshim and Arno Mittelbach. Modeling random oracles under unpredictable queries. In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 453–473. Springer, March, 2016. 64

[FN13] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the GPU. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 2013*, volume 7954 of *LNCS*, pages 339–356. Springer, June 2013. 7, 46, 47, 81

[Fuj14] Eiichiro Fujisaki. All-but-many encryption - A new framework for fully-equipped UC commitments. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 426–447. Springer, December 2014. 22

[GIKW14] Juan A. Garay, Yuval Ishai, Ranjit Kumaresan, and Hoeteck Wee. On the complexity of UC commitments. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 677–694. Springer, May 2014. 22, 32, 42, 44

[GL91] Shafi Goldwasser and Leonid A. Levin. Fair computation of general functions in presence of immoral majority. In Alfred J. Menezes and Scott A. Vanstone, editors, *CRYPTO 1990*, volume 537 of *LNCS*, pages 77–93. Springer, August 1991. 12

[GLMY16] Adam Groce, Alex Ledger, Alex J. Malozemoff, and Arkady Yerukhimovich. CompGC: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive, Report 2016/458, 2016. `https://eprint.iacr.org/2016/458`. 82

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984. 11

[GMS08] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 289–306. Springer, April 2008. 87

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC 1987*, pages 218–229. ACM Press, May 1987. 4, 11, 46, 77

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004. 4

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security 2011*. USENIX Association, 2011. 7, 46, 57, 81

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 772–783. ACM Press, October 2012. 80, 93

[HIKN08] Danny Harnik, Yuval Ishai, Eyal Kushilevitz, and Jesper Buus Nielsen. OT-combiners via secure computation. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 393–411. Springer, March 2008. 22

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 18–35. Springer, August 2013. 7, 46, 78, 81

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 458–475. Springer, August 2014. 77, 78, 81

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010. 14

[HMQ04] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 58–76. Springer, February 2004. 22, 41

[HMsG13] Nathaniel Husted, Steven Myers, abhi shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In Charles N. Payne Jr., editor, *ACSAC 2013*, pages 169–178. ACM Press, 2013. 7, 46, 81

[Hoc59] Alexis Hocquenghem. Codes correcteurs d'erreurs. *Chiffres*, 2(147-156):8–5, 1959. 65

[HSSV17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round mpc combining bmr and oblivious transfer. Cryptology ePrint Archive, Report 2017/214, 2017. `https://eprint.iacr.org/2017/214`. 9

[HZ15] Yan Huang and Ruiyu Zhu. Revisiting LEGOs: Optimizations, analysis, and their limit. Cryptology ePrint Archive, Report 2015/1038, 2015. `https://eprint.iacr.org/2015/1038`. 50

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, August 2003. 16, 41, 42

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *STOC 2007*, pages 21–30. ACM Press, June 2007. 22

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, August 2008. 22

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *STOC 1988*, pages 20–31. ACM Press, May 1988. 23

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014. 4

[KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 210–235. Springer, November / December 2015. 77

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, August 2014. 6

[KMRR15] Vladimir Kolesnikov, Payman Mohassel, Ben Riva, and Mike Rosulek. Richer efficiency/security trade-offs in 2PC. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 229–259. Springer, March 2015. 77

[KNR+17] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. Cryptology ePrint Archive, Report 2017/344, 2017. `https://eprint.iacr.org/2017/344`. 9, 10, 77

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, August 2015. 16, 41, 44, 55, 105

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*, pages 830–842. ACM Press, 2016. 47, 82

[KRW17a] Jonathan Katz, Samuel Ranellucci, and Xiao Wang. Authenticated garbling and efficient maliciously secure multi-party computation. Cryptology ePrint Archive, Report 2017/189, 2017. `https://eprint.iacr.org/2017/189`. 9

[KRW17b] Jonathan Katz, Samuel Ranellucci, and Xiao Wang. Authenticated garbling and efficient maliciously secure two-party computation. Cryptology ePrint Archive, Report 2017/030, 2017. `https://eprint.iacr.org/2017/030`. 7, 9, 81, 82, 100, 104

[KS06] Mehmet S. Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of Yao's garbled circuit construction. In *27th Symposium on Information Theory in the Benelux*, pages 283–290, 2006. 46

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, July 2008. 6, 51, 53, 83, 84, 87

[KsS12] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security 2012*. USENIX Association, 2012. 7, 46, 81

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013. 47, 48, 49, 82

[Lar15] Enrique Larraia. Extending oblivious transfer efficiently - or - how to get active security with constant cryptographic overhead. In Diego F. Aranha and Alfred Menezes, editors, *LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 368–386. Springer, September 2015. 16, 41

[Lin11] Yehuda Lindell. Highly-efficient universally-composable commitments based on the DDH assumption. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 446–466. Springer, May 2011. 22, 25, 39, 40, 41

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 1–17. Springer, August 2013. 7, 46, 77, 78, 81, 86, 89

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 259–276. Springer, August 2011. 41

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014,*

Part II, volume 8617 of *LNCS*, pages 495–512. Springer, August 2014. 5, 47, 82

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, May 2007. 7, 46, 54, 77, 81

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009. 46

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, March 2011. 7, 41, 46, 77, 81

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 2008*, volume 5229 of *LNCS*, pages 2–20. Springer, September 2008. 46, 81

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, August 2015. 6, 82

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, August 2014. 63, 77, 78, 81

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 2015*, pages 579–590. ACM Press, October 2015. 47, 48, 49, 54, 71, 73, 74, 75, 79, 81

[LSS16] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, volume 9985 of *LNCS*, pages 554–581, 2016. 6, 82

[LWN+15] Chang Liu, Xiao Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society, 2015. 80, 93

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 458–473. Springer, April 2006. 46, 77

[MGBF14] Benjamin Mood, Debayan Gupta, Kevin R. B. Butler, and Joan Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 582–596. ACM Press, November 2014. 60, 82

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P*, pages 112–127. IEEE, 2016. 9, 80, 83, 86, 93

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - A secure two-party computation system. In *USENIX Security 2004*, pages 287–302. USENIX Association, 2004. 80, 93

[MR92] Silvio Micali and Phillip Rogaway. Secure computation (abstract). In Joan Feigenbaum, editor, *CRYPTO 1991*, volume 576 of *LNCS*, pages 392–404. Springer, August 1992. 12

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 36–53. Springer, August 2013. 7, 46, 81

[Nao90] Moni Naor. Bit commitment using pseudo-randomness. In Gilles Brassard, editor, *CRYPTO 1989*, volume 435 of *LNCS*, pages 128–136. Springer, August 1990. 21

[NFT09] Ryo Nishimaki, Eiichiro Fujisaki, and Keisuke Tanaka. Efficient non-interactive universally composable string-commitment schemes. In Josef Pieprzyk and Fangguo Zhang, editors, *ProvSec 2009*, volume 5848 of *LNCS*, pages 3–18. Springer, November 2009. 22

[Nie07] Jesper Buus Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. Cryptology ePrint Archive,

Report 2007/215, 2007. http://eprint.iacr.org/2007/215. 16, 41

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, August 2012. 5, 9, 16, 41, 47, 55, 82

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, March 2009. 7, 23, 50, 53, 60, 78, 81, 88

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139, 1999. 6

[NR16] Jesper Buus Nielsen and Samuel Ranellucci. Reactive garbling: Foundation, instantiation, application. In *ASIACRYPT 2016, Part II*, LNCS, pages 1022–1052. Springer, December 2016. 7, 90, 106, 107, 127

[NST17] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *NDSS 2017*. The Internet Society, February 2017. 8, 9, 10, 11, 45, 81, 82, 85, 92, 93, 100, 116, 125, 129

[Par] Partisia. https://www.partisia.dk. 45

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO 1991*, volume 576 of *LNCS*, pages 129–140. Springer, August 1992. 39, 50

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, December 2009. 7, 46, 81

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, August 2008. 16, 41, 44, 66, 69

[Rab77] Michael O. Rabin. Digitalized signatures. Foundations of secure computation. In *Richard AD et al. (eds): Papers presented at a 3 day workshop held at Georgia Institute of Technology, Atlanta*, pages 155–166. Academic, New York, 1977. 78

[Rin] Peter Rindal. libOTe: an efficient, portable, and easy to use oblivious transfer library. `https://github.com/osu-crypto/libOTe`. 92

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security 2016*. USENIX Association, 2016. 47, 48, 49, 52, 71, 73, 74, 75, 79, 81, 82, 83, 100

[RT17] Peter Rindal and Roberto Trifiletti. SplitCommit: Implementing and analyzing homomorphic UC commitments. Cryptology ePrint Archive, Report 2017/407, 2017. `https://eprint.iacr.org/2017/407`. 10, 92

[Sep] Sepior. `https://www.sepior.com`. 4, 45

[Sha] Sharemind. `https://sharemind.cyber.ee`. 4

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015. 81, 94

[SRG+14] Nigel P. Smart, Vincent Rijmen, Benedikt Gierlichs, Kenneth G. Paterson, Martijn Stam, Bogdan Warinschi, and Watson Gaven. Algorithms, key size and parameters report – 2014, 2014. 40

[SS06] Rudolf Schürer and Wolfgang Ch Schmid. Mint: A database for optimal net parameters. In Harald Niederreiter and Denis Talay, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2004*, pages 457–469. Springer Berlin Heidelberg, 2006. 40

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 386–405. Springer, May 2011. 7, 46, 77, 78, 81

[sS13] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 523–534. ACM Press, November 2013. 7, 46, 47, 54, 81

[ST] Nigel Smart and Stefan Tillich. Circuits of basic functions suitable for mpc and fhe. 71, 104

[SZ13] Thomas Schneider and Michael Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 275–292. Springer, April 2013. 5, 6

[TJ11] Henk C. A. Tilborg and Sushil Jajodia. *Encyclopedia of Cryptography and Security*. Springer Publishing Company, Incorporated, 2nd edition, 2011. 78

[WGMK16] Xiao Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In *ESORICS (2)*, volume 9879 of *LNCS*, pages 99–117. Springer, 2016. 81

[WMK17] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017*, volume 10212 of *LNCS*, pages 399–424, 2017. 7, 46, 47, 48, 49, 75, 76, 81, 82, 83, 102

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS 1982*, pages 160–164. IEEE Computer Society Press, November 1982. 3, 45

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS 1986*, pages 162–167. IEEE Computer Society Press, October 1986. 3, 4, 45, 77

[ZB11] Hongchao Zhou and Jehoshua Bruck. Linear extractors for extracting randomness from noisy sources. In Alexander Kuleshov, Vladimir Blinovsky, and Anthony Ephremides, editors, *ISIT 2011*, pages 1738–1742. IEEE, 2011. 56

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. `https://eprint.iacr.org/2015/1153`. 80, 93

[ZH17] Ruiyu Zhu and Yan Huang. Faster lego-based secure computation without homomorphic commitments. Cryptology ePrint Archive, Report 2017/226, 2017. `https://eprint.iacr.org/2017/226`. 79, 81, 82, 85, 87

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half

gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, April 2015. 6, 63, 84, 94