ABSTRACT

Title of thesis: EFFICIENT SECURE COMPUTATION FOR REAL-WORLD SETTINGS AND SECURITY MODELS

Alexis J. Malozemoff, Doctor of Philosophy, 2016

Thesis directed by: Professor Jonathan Katz
Department of Computer Science

Secure computation involves multiple parties computing a common function while keeping their inputs private, and is a growing field of cryptography due to its potential for maintaining privacy guarantees in real-world applications. However, current secure computation protocols are not yet efficient enough to be used in practice. We argue that this is due to much of the research effort being focused on *generality* rather than *specificity*. Namely, current research tends to focus on constructing and improving protocols for the strongest notions of security or for an arbitrary number of parties. However, in real-world deployments, these security notions are often too strong, or the number of parties running a protocol would be smaller. In this thesis we make several steps towards bridging the efficiency gap of secure computation by focusing on constructing efficient protocols for specific real-world settings and security models. In particular, we make the following four contributions:

1. We show an efficient (when amortized over multiple runs) maliciously secure

two-party secure computation (2PC) protocol in the *multiple-execution setting*, where the same function is computed multiple times by the same pair of parties.

2. We improve the efficiency of 2PC protocols in the *publicly verifiable covert security model*, where a party can cheat with some probability but if it gets caught then the honest party obtains a certificate proving that the given party cheated.

3. We show how to optimize existing 2PC protocols when the function to be computed includes predicate checks on its inputs.

4. We demonstrate an efficient maliciously secure protocol in the *three*-party setting.

EFFICIENT SECURE COMPUTATION FOR REAL-WORLD SETTINGS AND SECURITY MODELS

by

Alexis J. Malozemoff

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Advisory Committee:
Professor Jonathan Katz, Chair/Advisor
Professor Michael Hicks
Professor Héctor Corrada Bravo
Doctor Vladimir Kolesnikov
Professor Lawrence C. Washington

Acknowledgments

I want to begin by thanking my advisor Jonathan Katz. He took me on from my first semester at Maryland, gave me freedom to pursue my own interests, encouraged me to apply for fellowships during my first year, always had funding available for travel to conferences and workshops, and guided me throughout my research career at Maryland. He always had faith in my ability, even when I didn't have much faith myself, and that support certainly helped me progress through my Ph.D.

I would also like to thank the three plus years of financial support I received from the National Defense Science & Engineering Graduate (NDSEG) Fellowship, which gave me a lot of freedom and stability to work on my own interests. For the time not covered by the above fellowship, I was supported through my advisor by NSF award #1111599, and am thankful for the support.

I am very grateful for the support I received during my first year at Maryland from the post-docs and senior students in the Crypto group: Seung Geol Choi, Ranjit Kumaresan, Dominique Schröder, Hong-Sheng Zhou, and Vassilis Zikas. They were all very generous with their advice, guidance, and willingness to work with me from the get-go, even before I really knew much about how to conduct research, let alone secure computation. Two of the chapters of this thesis are direct results from those initial collaborations, and I thank them all for their help and continued friendship.

I am also appreciative of the guidance and friendship of Vlad Kolesnikov, with whom I did a brief internship with at Bell Labs in the winter of 2015 and with whom

I have collaborated with ever since. I would also like to thank all my officemates and friends throughout my time at Maryland, who helped make the Ph.D. journey an enjoyable experience. And of course thanks to all my co-authors whose work has contributed to this thesis: Seung Geol Choi, Yan Huang, Jon Katz, Vlad Kolesnikov, Ranjit Kumaresan, Xiao Wang, and Vassilis Zikas.

Finally, I would like to thank all of my committee members, Jon Katz, Mike Hicks, Héctor Corrada Bravo, Vlad Kolesnikov, and Larry Washington, for taking the time to review my dissertation and participate in my defense.

Table of Contents

Chapter 1: Introduction

Secure computation describes the problem of multiple parties $P_1, \ldots, P_n$ wishing to compute some common function f on their private inputs $x_1, \ldots, x_n$; each party would like to learn $f(x_1, \ldots, x_n)$ while preventing the other parties from learning anything about its input (besides what can be learned from the output of f). First proposed by Yao [1], many of the initial results were mostly of theoretical interest [2, 3, etc]. However, since the work of Fairplay [4], which demonstrated the first implementation of secure computation, this area of cryptography has seen the research focus shift towards *practical* efficiency. Although much progress has been made over these last few years, there are still many hurdles before secure computation can be useful in practice. In this thesis, we investigate various approaches towards improving this efficiency gap by focusing on realistic use cases and security models. In particular, our improvements result from focusing on those settings where one could potentially see secure computation being used in practice, versus focusing on improving general constructions which may not be applicable to real-world settings. Namely, we focus on models and security settings which we argue are more realistic in a practical setting, and devise efficient protocols for these particular use cases.

Before discussing our contributions, we begin with a brief overview of secure computation and its security models. We mostly focus on the *two-party* case, although we briefly discuss the three-party case.

Two-party secure computation. We begin with a discussion of *two-party* secure computation (2PC), and how protocols in this setting work. One common solution to the 2PC problem is to use *garbled circuits* [5]. Suppose we have two parties, P_1 with input x_1 and P_2 with input x_2, and they would like to compute some function $f(\cdot, \cdot)$ on their inputs. We treat f as a Boolean circuit C (e.g., a circuit with AND and XOR gates). One of the parties, say P_1, acts as a *garbler* (also called the *generator*), constructing a "garbled" version of C which hides all of the internal details of the computation of C. Namely, each wire value is represented by an opaque *label* and there is a method (known only to the garbler) of mapping labels to their actual bit values. P_1 can send this garbled circuit, denoted by $\widehat{C}$, to party P_2, known as the *evaluator*. Now, the evaluator can evaluate $\widehat{C}$ given labels for each of the input wires to C. P_1 can directly send the labels corresponding to its input x_1 (recalling that these labels are opaque, this reveals no information to P_2 about what input the labels represent). The last difficulty is for P_2 to get the wire labels for its input x_2. This is accomplished through an *oblivious transfer* (OT) protocol, where a *sender* inputs two messages m_0 and m_1, and a *receiver* inputs a choice bit b; the sender receives no output and the receiver receives m_b. Thus, P_1 can input the two labels for each of P_2's input bits into the OT protocol, and P_2 can input as its choice bits the appropriate bits in x_2, allowing P_2 to learn the appropriate labels

without P_1 learning anything about P_2's input. Now, P_2 can take all of the wire labels and evaluate $\widehat{C}$ to learn the output.

Two-party secure computation from garbled circuits is very efficient. Garbling and evaluating require only symmetric key operations and can utilize hardware AES support [6]. The main bottleneck of the described protocol, then, is the OTs required for P_2's inputs, as OT protocols require costly public-key operations. However, Ishai et al. [7] showed how, given a fixed number of *base* OTs, one can construct any polynomial number of OTs using only symmetric-key operations. This process, known as *OT extension*, has had a huge impact on the efficiency of the garbled circuit approach to 2PC and is necessary for any practical instantiation of the garbled circuit protocol.

The malicious security model. While the above described protocol is secure in the *semi-honest* security setting, where parties are assumed to follow the protocol but can try to learn additional information by looking at the protocol transcript, it is not secure in the *malicious* setting, where an adversarial party can deviate arbitrarily from the protocol. As an example of what can go wrong, a malicious P_1 can input an invalid wire label for the 0-bit for one of P_2's inputs. If P_2 aborts the protocol (due to having an invalid wire label), P_1 learns that that bit of P_2's input was 0. If P_2 does *not* abort, then P_1 learns that that bit was 1. This attack, known as the *selective failure attack*, is just one example of the many things that can go wrong when the parties can deviate from the protocol description.

The garbled circuit approach can be adapted to the malicious setting using

the *cut-and-choose* paradigm [8, 9]. Instead of garbling one circuit, the generator garbles $O(\rho)$ circuits, where ρ is a statistical security parameter (i.e., a malicious party can successfully cheat with probability $2^{-\rho}$). The garbler sends these garbled circuits to the evaluator, who asks for, say, half to be opened. If any of the opened circuits are invalid garblings, the evaluator detects cheating and aborts. Otherwise, it takes the leftover (unopened) circuits and evaluates them, outputting the majority output as the output of the protocol. Although this basic approach has several issues that must be addressed (e.g., how to force the generator to use the same input in each of the evaluated circuits, avoiding the selective failure attack, etc.), this basic paradigm is a common way to construct maliciously secure 2PC protocols using garbled circuits. However, the best existing protocol based on the cut-and-choose paradigm has a garbled circuit replication factor equal to the statistical security parameter [10] (i.e., to securely compute a circuit, it must be garbled ρ times for security $2^{-\rho}$). Thus, for $\rho = 40$ we still get a $40\times$ overhead over the semi-honest setting, which is often still prohibitive for real-world use.

The multiple-execution setting. In practice, however, the *same* function can potentially be executed many times on different inputs. For example, consider the following use cases for 2PC: a bank customer performing financial transactions (e.g., payments or transfers), a cell phone customer performing private location-based queries, two businesses or government agencies querying their joint databases of customers, etc. In all of these scenarios, many of the securely evaluated functions are the same, only differing on their inputs. In fact, it seems plausible that single-

execution functions may be *less likely* to be used in commercial settings. This is because, as a rule-of-thumb of security, externally-accessible interfaces need to be clean and standardized. Allowing a small number of predetermined customer actions allows for more manageable overall security.

Additionally, many complex protocols from the research literature include multiple executions of the same function evaluated on different inputs. For example, Gordon et al. [11] propose sublinear 2PC based on oblivious RAM (ORAM). In their protocol, each ORAM step is executed by evaluating the same function using 2PC. Another frequently used subroutine is an oblivious pseudorandom function, used, e.g., in the previously mentioned sublinear 2PC work [11] as well as in private database searches [12, 13]. Likewise, work by Pappas et al. [14] traverses the database search tree by evaluating the same match function at each tree node.

> Say two parties run the same function t times. Can we construct a protocol that requires fewer than ρt garbled circuits while still retaining the malicious security guarantee with $2^{-\rho}$ security?

We consider malicious 2PC in what we call the *multiple-execution* setting, where two parties wish to securely evaluate the same circuit multiple times. As mentioned above, recent works by Lindell [10] and Huang et al. [15] have obtained optimal complexity for cut-and-choose performed over garbled circuits in the single execution setting. We show that it is possible to obtain much lower *amortized* overhead for cut-and-choose in the multiple-execution setting.

Our efficiency improvements result from a novel way to combine the "fast

cut-and-choose" technique of Lindell [10] with LEGO-based cut-and-choose techniques [16, 17]. In concrete terms, for 40-bit statistical security we obtain a 2× improvement (per execution) in communication and computation for as few as 7 executions, and require only 8 garbled circuits (i.e., a 5× improvement) per execution for as low as 3500 executions. Our results suggest the possibility that 2PC in the malicious setting can be less than an order of magnitude more expensive than in the semi-honest setting.

This work is based on work published at Crypto 2014 [18]. See Chapter 3 for details.

The covert security model. A third security model (besides semi-honest and malicious) is that of *covert* security [19]. In this setting an adversarial party can successfully cheat with some probability $1 - \epsilon$. However, with probability ϵ it gets caught and does not learn anything about the other party's input. A recent extension of this model provides *public verifiability* [20]: if a party gets caught cheating, the honest party can produce a certificate which provides proof of this cheating to any third party. This model is very compelling, as the ability to demonstrate proof of cheating is a powerful incentive not to cheat. Unfortunately, the only existing protocol in this setting [20] is not that efficient due to the need to (at a high level) use OT for each of P_2's inputs; that is, the protocol cannot take advantage of OT extension to improve its running time.

> Can we construct a more efficient protocol in the publicly verifiable covert (PVC) security model based on OT extension?

We improve the performance in the PVC model by constructing a PVC-compatible OT extension protocol as well as making several practical improvements to the existing protocol. As compared to the state-of-the-art OT extension-based two-party covert protocol, our PVC protocol adds relatively little: four signatures and a roughly 67% increase in running the OT extension protocol. This is a significant improvement over the existing protocol, which requires public-key-based OTs per input bit. We present detailed estimates showing (up to orders of magnitude) concrete performance improvements over the existing PVC protocol [20] and the best known malicious protocol [21].

This work is based on work published at Asiacrypt 2015 [22]. See Chapter 4 for details.

Predicate checks on inputs. When using cut-and-choose to construct covert or malicious protocols, the same circuit needs to be garbled multiple times. Suppose, now, that we are interested in computing functions where each party's input must satisfy some predicate. As an example, consider a setting where one party's input must contain a valid signature; that is, the party inputs its input along with a signature on that input that is checked for validity before the actual function of interest can be computed. Clearly, we can include this check within the garbled circuit; however, this means that when using cut-and-choose protocols, this check is repeated in *each* garbled circuit. For predicate checks such as the above signature example, this can be very costly, especially when the underlying function to be computed over the inputs is relatively simple.

For circuits where we want to check a predicate on either party's input, can we construct a protocol more efficient than the naive solution of including the check in each garbled circuit?

Here we show a protocol in which only the underlying function is garbled ρ times, and the predicate checks are each garbled only *once*. For certain natural examples (e.g., signature verification followed by evaluation of a million-gate circuit), this can lead to huge savings in communication (up to 80×) and computation (up to 56×). We provide detailed estimates using realistic examples to validate our claims.

This work is based on a preprint [23]. See Chapter 5 for details.

Secure three-party computation. The setting of secure computation for three or more parties, where we assume all but one of the parties may be malicious and colluding, has been much less studied, at least from a practical performance perspective. Although secure multi-party computation protocols exist, they either require a complicated (and extremely costly) setup phase [24] or are not known to be practically efficient [25]. However, in real-world settings, it seems unlikely that one would run secure computation among, say, one hundred parties. Most likely one would run secure computation among a small number of parties, say three or four. While existing multi-party protocols are designed to handle an arbitrary number of parties, it seems possible that one could design a more efficient protocol for a small *fixed* number of parties.

Can we construct more practically efficient secure computation protocols when restricting the number of parties to some fixed $n > 2$? In

particular, can we construct an efficient secure computation protocol for *three* parties?

In this work we explore the possibility of using cut-and-choose for practical secure *three-party* computation. We propose a constant-round protocol for three-party computation tolerating any number of malicious parties, whose computational cost is essentially only a small constant worse than that of state-of-the-art two-party protocols.

This work is based on work published at Crypto 2014 [26]. See Chapter 6 for details.

Summary. In this thesis we present four constructions which improve the state-of-the-art of secure computation, with a focus on *realistic settings and security models*. While there is still a lot of work to be done before secure computation can be made truly practical, this thesis presents a further step on this path towards practicality.

Chapter 2: Preliminaries

Notation

We let κ denote the computational security parameter and let ρ denote the statistical security parameter; namely, a (computationally bounded) adversary can succeed in cheating with probability $\leq 2^{-\rho} + \mathsf{negl}(\rho)$. We use PPT to denote "probabilistic polynomial time" and let $\mathsf{negl}(\cdot)$ denote a negligible function in its input.

When considering two-party protocols between parties P_1 and P_2, when we use subscript $i \in \{1, 2\}$ to denote a party we let subscript -$i = 3 - i$ denote the other party. We use $i^* \in \{1, 2\}$ to denote a malicious party and -$i^* = 3 - i^*$ to denote the associated honest party.

We use $[n]$ to denote $\{1, \ldots, n\}$ and $\|$ to denote concatenation. Let "$a := f(x_1, x_2, \ldots)$" denote setting a to be the deterministic output of f on inputs $x_1, x_2, \ldots$; the notation "$a \leftarrow f(x_1, x_2, \ldots)$" is the same except that f here is randomized. We use $a \in_R S$ to denote selecting a uniformly at random from set S. For bitstring x, we let $x[i]$ denote the ith bit of x.

Defining Security

We use the standard definition of security for two-party computation in the presence of malicious adversaries [27, Chapter 7], and we repeat the definition here for completeness and to fix notation.

We let $\mathcal{A}$ be an adversary that can corrupt one or more parties. Here we consider the *malicious* setting, which means that when $\mathcal{A}$ corrupts a party, it learns the entire internal state of said party and can deviate from the protocol arbitrarily.

Security is defined by comparing the execution of the protocol with an "idealized" world, where we have access to an *ideal functionality* which exactly captures the expected correct behavior of said protocol. If an adversary $\mathcal{A}$ is unable to tell whether it is interacting in the *ideal world* or the *real world*, then we say the protocol is a secure realization of the ideal functionality. By "unable to tell," we mean that the distributions of the two worlds are *computationally indistinguishable* from the point of view of the adversary, which we denote by $\overset{\text{c}}{\approx}$.

Below, we give a formal treatment for the case of *two-party secure computation*; however, it is easy to adapt this treatment to handle three or more parties.

Ideal model execution. In the ideal model, we have parties P_1 and P_2, and an adversary $\mathcal{A}$ with auxiliary information aux who can corrupt one of the two parties. An ideal execution for the computation of a function $f(\cdot, \cdot)$ proceeds as follows. We let $\mathcal{F}$ define the idealized execution of $f(\cdot, \cdot)$, where we assume without loss of generality [8] that only P_2 receives output.

- Party P_1 obtains input x, and party P_2 obtains input y.

- An honest party sends its given input to the ideal functionality $\mathcal{F}$, whereas a malicious party can send an arbitrary input. Denote the inputs given to $\mathcal{F}$ as x' and y'.

- The functionality $\mathcal{F}$ computes $z \leftarrow f(x', y')$ and sends z to P_2.

- An honest party outputs the given output from $\mathcal{F}$, whereas a malicious party outputs an arbitrary function of its view of the protocol execution.

We let $\text{IDEAL}_{\mathcal{F},\mathcal{A}(\mathsf{aux})}(x, y, 1^\kappa)$ denote the joint output of the adversary $\mathcal{A}$ and the honest party with inputs x and y when interacting with ideal functionality $\mathcal{F}$.

Real model execution. In the real model, we again have parties P_1 and P_2 and an adversary $\mathcal{A}$ with auxiliary information aux who can corrupt one of the two parties. In this setting, the parties execute some two-party protocol Π_f computing function $f(\cdot, \cdot)$. We let $\text{REAL}_{\Pi_f,\mathcal{A}(\mathsf{aux})}(x, y, 1^\kappa)$ denote the joint output of the adversary $\mathcal{A}$ and the honest party with inputs x and y when interacting with protocol Π_f.

Definition 2.1. *Protocol Π_f securely computes $\mathcal{F}$ if for every* PPT *adversary $\mathcal{A}$ in the real model, there exists a* PPT *simulator $\mathcal{S}$ in the ideal model such that for all x, y, and* aux*, it holds that*

$$\{\text{IDEAL}_{\mathcal{F},\mathcal{S}(\mathsf{aux})}(x, y, 1^\kappa)\} \overset{c}{\approx} \{\text{REAL}_{\Pi_f,\mathcal{A}(\mathsf{aux})}(x, y, 1^\kappa)\}.$$

Remarks. One way to prove that a protocol securely computes some ideal functionality is to construct a *simulator* with black-box access to an adversary $\mathcal{A}$ such

that the view of the simulator in the ideal world is computationally indistinguishable from the view of the adversary in the real world. This implies that the view of the adversary in the real world "looks the same" as the view of the simulator when interacting with an idealized version of the protocol, thus implying that the adversary gains no additional information in the real world than what is leaked by the ideal functionality in the ideal world.

Note that we can define ideal functionality $\mathcal{F}$ using a "functionality box" as follows.

Functionality $\mathcal{F}$

P_1 inputs input x, and P_2 inputs input y.

$\mathcal{F}$ sends $f(x, y)$ to P_2. P_1 receives no output.

This box exactly captures the ideal model behavior of $\mathcal{F}$ as explained above. Throughout this thesis we use both methods of defining a functionality interchangeably.

Garbled Circuits

For completeness, we give a description of a garbling scheme constructing garbled circuits. Let $f(\cdot, \cdot)$ be some function. A garbling scheme is a tuple of two functions $(\mathsf{Gb}, \mathsf{Ev})$. We define a garbling scheme producing garbled circuits as follows.

The garbling procedure Gb works as follows. We begin by treating f as a Boolean circuit C. We associate two random labels $X_{w,0}, X_{w,1}$ with each wire w in the circuit; label $X_{w,0}$ corresponds to the value '0' and $X_{w,1}$ corresponds to the value '1'. In addition, for each wire w we choose a random permutation (or mask) bit λ_w. Each label has an associated tag, derived from the permutation bit, which

acts as a blinding of the true value the label represents. Now, consider gate G_γ in the circuit with input wires α and β. The garbled gate of G_γ consists of an array of four encryptions: for each $(b_\alpha, b_\beta) \in \{0,1\} \times \{0,1\}$, the row (b_α, b_β) consists of an encryption of $X_{\gamma, G_\gamma(b_\alpha \oplus \lambda_\alpha, b_\beta \oplus \lambda_\beta) \oplus \lambda_\gamma}$ and its corresponding tag $G_\gamma(b_\alpha \oplus \lambda_\alpha, b_\beta \oplus \lambda_\beta) \oplus \lambda_\gamma$ under labels X_{α, b_α} and X_{β, b_β}. Let P denote a table that stores all the garbled gates; in particular, the entry $P[\gamma, b_\alpha, b_\beta]$ contains an encryption corresponding to row (b_α, b_β) of the garbled gate for G_γ.

The evaluation procedure Ev is as follows. Let α and β be input wires connected to gate G with index γ. The evaluator is given $(X_{\alpha, b_\alpha \oplus \lambda_\alpha}, b_\alpha \oplus \lambda_\alpha)$ and $(X_{\beta, b_\beta \oplus \lambda_\beta}, b_\beta \oplus \lambda_\beta)$, along with P. It takes the row $P[\gamma, b_\alpha \oplus \lambda_\alpha, b_\beta \oplus \lambda_\beta]$ and decrypts it using the labels $X_{\alpha, b_\alpha \oplus \lambda_\alpha}$ and $X_{\beta, b_\beta \oplus \lambda_\beta}$, resulting in $(X_{\gamma, G(b_\alpha, b_\beta) \oplus \lambda_\gamma}, G(b_\alpha, b_\beta) \oplus \lambda_\gamma)$. It is straightforward to verify that by continuing this evaluation, the output of each gate will be revealed masked by its corresponding mask. By picking masks of the output wires to be '0' we ensure that the evaluator receives the (unmasked) output of the circuit.

Security. For a garbling scheme to be secure, it should satisfy some notion of privacy. Namely, given some output z of f, there should exist a simulator that can produce a garbling of a circuit that outputs z that is indistinguishable from a correctly garbled circuit.

Definition 2.2. *Garbling scheme* $(\mathsf{Gb}, \mathsf{Ev})$ *satisfies* privacy *if there exists a* PPT *simulator* $\mathcal{S}$ *such that for every* PPT *adversary* $\mathcal{A}$*, for all polynomial size functions*

f and inputs x and y it holds that

$$\Pr\left[\mathcal{A}(\widehat{C}, \{X_{w,x[i]}\}, \{Y_{w,y[i]}) = 1 : (\widehat{C}, \{X_{w,b}\}, \{Y_{w,b}\}) \leftarrow \mathsf{Gb}(1^\kappa, f)\right]$$
$$\stackrel{c}{\approx}$$
$$\Pr\left[\mathcal{A}(\widehat{C}, \{X_i\}, \{Y_i) = 1 : (\widehat{C}, \{X_i\}, \{Y_i\}) \leftarrow \mathcal{S}(1^\kappa, f(x, y))\right].$$

It is well known that the above described garbling scheme satisfies privacy [28].

Using garbled circuits in secure computation. As mentioned in the Introduction, garbled circuits can be used to achieve two-party secure computation using a primitive called *oblivious transfer* (OT), where a sender inputs two messages m_0 and m_1 and a receiver receives message m_b for some choice bit b. Party P_1 with input x and acting as the garbler constructs a garbled circuit and sends it to party P_2 with input y and acting as the evaluator. In addition, P_1 and P_2 run an OT protocol for each of P_2's input bits, with P_1 inputting as the sender the two wire labels $X_{w,0}, X_{w,1}$, and P_2 receiving $X_{w,y[i]}$. P_1 then sends the input-wire labels $\{X_{w,x[i]}\}$ corresponding to its own inputs, allowing P_2 to evaluate the garbled circuit and learn the output.

The free-XOR technique. We note one optimization technique that we reference throughout the thesis. This technique, called the "free-XOR" technique [29], allows one to construct a garbled circuit such that when evaluating the garbled circuits, XOR gates can be evaluated "for free"; namely, evaluating an XOR gates only requires an XOR operation by the evaluator.[1] This is done as follows. The garbler

[1]In order to prove security of the "free-XOR" technique in the standard model, one needs to

selects some global random value Δ, and for each input wire w chooses $X_{w,0}$ at random at sets $X_{w,1} := X_{w,0} \oplus \Delta$. Now, for each XOR gate in the circuit, the garbler sets the output 0-label to be the XOR of the two input 0-labels, and sets the output 1-label to be the output 0-label XORed with Δ. AND gates are handled as before.

Because the evaluator only learns a single label, it cannot learn Δ and thus security is preserved. However, note that now, when processing an XOR gate, the evaluator only needs to XOR the two wire labels together to learn the output-wire label of that gate. Namely, given $X_\alpha := X_{\alpha,0} \oplus b\Delta$ and $X_\beta := X_{\beta,0} \oplus b'\Delta$, it holds that $X_{\gamma,b\oplus b'} = X_\alpha \oplus X_\beta$.

Achieving Malicious Security

The basic garbled circuit protocol described above is only secure against *semi-honest* adversaries, that is, adversaries that are assumed to follow the protocol but may try to deduce the other party's input from the protocol transcript. As described in the Introduction, the *cut-and-choose* paradigm is a common way to lift the garbled circuit approach to handle malicious adversaries. Cut-and-choose protocols for garbled circuits work by letting P_1 generate and send a number of garbled circuits to P_2, who then chooses a subset of circuits to open and check for correctness. If the checks pass, P_2 evaluates the remaining circuits and obtains the final output by taking majority over the individual outputs. However, using cut-and-choose introduces two possible avenues of attack: a *selective failure* attack on P_2's input and *input*

make additional assumptions about the encryption used in garbled circuits [30, 31, 29].

inconsistency on P_2's input. We discuss each in turn, as well as known approaches to solve each problem.

Selective failure: This attack proceeds as follows. Recall that P_1 sends the input-wire labels of P_2's inputs through oblivious transfer. However, a malicious P_1 could set, say, the 0-bit label of the ith input-wire among all the garbled circuits to garbage. Now, if P_2 receives these garbage labels (because its ith input bit was 0) it cannot evaluate the garbled circuits, and thus must abort, allowing P_1 to learn P_2's ith input bit.

There are two main ways to circumvent this attack. Lindell and Pinkas introduced the "XOR-tree" approach [8], where the parties modify the circuit such that instead of P_2 having input y, it has ρ inputs $\{y_1, \ldots, y_\rho\}$. P_2 chooses these values randomly such that $y = \bigoplus_i y_i$. Now, if P_1 launches a selective failure attack on a single input bit it only learns a random share of P_2's real input bit $y[i]$. Of course, P_1 can launch a selective failure attack on multiple bits, but can only learn one of P_2's input bits with probability $2^{-\rho}$. Note that the approach as described blows up P_2's input from n to ρn, where n is the length of P_2's input. However, Lindell and Pinkas [8] showed how to reduce this to $\max\{4n, 8\rho\}$.

Another approach for dealing with the selective failure attack is *cut-and-choose oblivious transfer* [9]. This protocol is similar to oblivious transfer, except now P_2 also inputs a "check set" $\mathcal{J}$, and learns *both* of P_1's inputs for those indices in $\mathcal{J}$. Thus, P_2 can execute cut-and-choose on P_1's inputs to the oblivious

transfer itself, aborting on any inconsistency.

Input inconsistency: Another issue with using cut-and-choose is that one needs to enforce that P_1 uses the same input x in all the evaluation circuits. Otherwise, P_1 could get P_2 to evaluate, for example, $f(x_1, y)$ and $f(x_2, y)$, potentially allowing P_1 to learn some information about y based on choices of x_1 and x_2.

While there are several approaches to solving this issue, we focus on the approach introduced by Lindell and Pinkas [9], which we make use of throughout this thesis. For each of its n input bits, P_1 chooses values $\{g^{a_{i,b}}\}_{b\in\{0,1\}}$ for random $a_{i,b}$. Likewise, for each of the s garbled circuits in the cut-and-choose, P_1 chooses values $\{g^{r_j}\}_{j\in[s]}$ for random r_j. Now, the input-wire label for input bit b for the ith input of the jth circuit is set to $g^{a_{i,b}\cdot r_j}$. Using this specific structure of the labels allows P_1 to efficiently prove in zero-knowledge that its choices of its ith bit are consistent across all evaluation circuits. Namely, suppose P_2 has labels X and X' and knows the values $\{g^{a_{i,b}}\}$ and $\{g^{r_j}\}$. Then P_1 can efficiently prove that for input $\{g^{a_{i,0}\cdot r_j}, g^{a_{i,1}\cdot r_j}, g^{a_{i,0}\cdot r_{j'}}, g^{a_{i,1}\cdot r_{j'}}\}$ and bit σ it holds that $X = g^{a_{i,\sigma}\cdot r_j}$ and $X' = g^{a_{i,\sigma}\cdot r_{j'}}$; namely, that the labels P_2 has for P_1's ith input bit across all evaluation circuits are consistent with P_1's input $\sigma := x[i]$.

Fast cut-and-choose using cheating punishment [10]. Prior cut-and-choose works [8, 32] required P_1 to send at least 125 circuits to guarantee security 2^{-40}. Lindell's improved technique [10] achieves $2^{-\rho}$ security while requiring P_1 to send only ρ circuits (i.e., 40 circuits for 2^{-40} security).

Lindell's protocol (which we call the "fast cut-and-choose" protocol) has two phases. In the first phase, P_1 with input x and P_2 with input y run a modified cut-and-choose which ensures that P_2 obtains a proof of cheating ϕ if it receives two inconsistent output values in any two evaluation circuits. Now, if all evaluation circuits produce the same output z, P_2 locally stores z as its output. Both parties *always* continue to the second *cheating-punishment* phase. In it, P_1 and P_2 securely evaluate (using some existing secure computation protocol) a *smaller* circuit C', which takes as inputs P_1's input x and P_2's proof ϕ. (P_2 inputs random values if it does not have ϕ.) P_1 proves in zero-knowledge the consistency of its input x between the two phases. C' outputs x to P_2 if ϕ is a valid proof of cheating; otherwise P_2 receives nothing. The efficiency improvement is due to the fact that cheating is *punished* by revealing P_1's input x to P_2 if there is any inconsistency in outputs, and thus P_2 can simply compute $f(x, y)$ itself.

Chapter 3: The Multiple-execution Setting

As mentioned in Chapter 1, the classical technique for lifting the garbled circuit approach to work in the malicious setting is *cut-and-choose*, formalized and proven secure by Lindell and Pinkas [8]. Until recently, this approach required significant overhead: to guarantee probability of cheating $\leq 2^{-\rho}$, approximately 3ρ garbled circuits needed to be generated and sent. However, in 2013 two works reduced the number of garbled circuits required in cut-and-choose to $\rho + O(\log \rho)$ per party [15] and to ρ [10].

In this chapter we present a way to further significantly reduce the replication factor for cut-and-choose-based protocols in the *multiple-execution* setting, where the same function (possibly with different inputs) is evaluated multiple times either in parallel or sequentially. To achieve this, we combine in a novel way the "fast cut-and-choose" technique of Lindell [10] (cf. Chapter 2) with the "LEGO cut-and-choose" technique [16, 17] (see below).

Notation. Besides the notation introduced in Chapter 2, we let t denote the total number of times the parties wish to evaluate a given circuit, and let $\nu = \nu(\rho, t)$ represent the number of circuits, per evaluation, that need to be generated to achieve an error probability of $\leq 2^{-\rho}$.

LEGO cut-and-choose [16, 17]. These works take a different approach than standard cut-and-choose protocols by implementing a two-stage cut-and-choose at the *gate* level. The evaluation circuit C is then constructed from the unopened garbled gates. In the first stage, P_1 sends multiple garbled gates and P_2 performs a standard cut-and-choose with replication factor $\nu(\rho) = O(\rho/\log|C|)$. P_2 aborts if any opened gate is garbled incorrectly. In the next stage, P_2 partitions the $\nu(\rho)|C|$ garbled gates into *buckets* such that each bucket contains $O(\nu(\rho))$ garbled gates. This two-stage cut-and-choose ensures that, except with probability $\leq 2^{-\rho}$, each bucket contains a *majority* of correctly constructed garbled gates.

To connect gates with one another, Nielsen and Orlandi [17] use homomorphic Pedersen commitments. The resulting computational efficiency is relatively poor as they perform several expensive public-key operations *per gate*. This is addressed in the miniLEGO work [16], where the authors (among other things) construct homomorphic commitments from oblivious transfer (OT), whose cost can be amortized by OT extension [7]. However, the overall efficiency of this construction is still lacking in concrete terms due to large constants inside the big-O notation. In particular, the communication efficiency is adversely affected by the use of asymptotically constant-rate codes that are concretely inefficient.

Naive approaches to combining fast cut-and-choose with LEGO. We now discuss two natural approaches for combining Lindell's fast cut-and-choose technique with LEGO-based cut-and-choose to achieve protocols secure in the multiple-execution setting, which yield baseline benchmarks.

The obvious and uninteresting approach is to simply run a maliciously-secure protocol multiple times. More interestingly, the following LEGO trick, implicit in the work of Nordholt et al. [33], can help. Consider a circuit $\widetilde{C}$ which consists of t copies of the original circuit C. We perform gate-level LEGO cut-and-choose directly on $\widetilde{C}$.[1] Doing this requires a replication factor of $\nu = O(\rho/\log|\widetilde{C}|) = O(\rho/(\log|C| + \log t))$. However, while this is a good asymptotic improvement, the concrete efficiency of LEGO protocols is weak due to both heavy public-key machinery per gate [17] and expensive communication [16]. Furthermore, LEGO requires a *majority* of gates in each bucket to be good.

This leads to the second natural approach: use fast cut-and-choose in LEGO and require that as long as each bucket contains at least one (as opposed to a majority) correctly constructed garbled gate, the protocol succeeds. Unfortunately, the circuit C' used in the corresponding cheating-punishment phase is no longer small. Indeed, C' has to deliver P_1's input x to P_2 if P_2 supplies a valid cheating proof ϕ. However, the number of possible proofs are now proportional to $|C|$, since such a proof could be generated from any of the $|C|$ buckets. This implies that C' is of size at least $|C|$.[2] Therefore, this approach cannot perform better than evaluating C from scratch using fast cut-and-choose.

[1] A similar approach (i.e., of directly securely evaluating $\widetilde{C}$) can be used to run Lindell's protocol [10] t times *in parallel* without having to increase the replication factor.

[2] The size of C' is also proportional to the computational security parameter κ, as the proofs are of length at least 2κ.

3.1 Our Contribution

Our main idea for the multiple-execution setting is to run two-stage LEGO-style cut-and-choose at the *circuit* level, and then use fast cut-and-choose in the second stage (thereby requiring only a single correctly constructed circuit from each bucket). In particular, now the size of C' used in each execution depends only on the input and output lengths of C, and is no longer proportional to $|C|$. In this section, we focus only on the cut-and-choose aspect of the protocol; namely, on preventing P_1's cheating by submitting incorrect garbled circuits. More detailed protocol descriptions for both the parallel and sequential settings can be found in Sections 3.3 and 3.4.

In the first-stage cut-and-choose, P_1 constructs and sends to P_2 a total of νt garbled circuits. Next, P_2 requests that P_1 open a random $\nu t/2$-sized subset of the garbled circuits. If P_2 discovers that any opened garbled circuit is incorrectly constructed, it aborts. Otherwise, P_2 proceeds to the second stage cut-and-choose, where it randomly assigns unopened circuits to t buckets such that each bucket contains $\nu/2$ circuits. Now, as in the fast cut-and-choose protocol [10], each of the t evaluations are executed in two phases. In the first phase of the kth execution, P_2 evaluates the $\nu/2$ evaluation circuits contained in the kth bucket. The circuits are designed such that if P_2 obtains different outputs from evaluating circuits in the kth bucket, then it obtains a proof of cheating ϕ_k. Next, both parties continue to the cheating-punishment phase, where P_1 and P_2 securely evaluate a smaller circuit that outputs P_1's input x_k if P_2 provides a valid proof ϕ_k.

Clearly, P_1 succeeds in cheating only if (1) it constructed $m \geq \nu/2$ bad circuits,

(2) none of these m bad circuits were caught in the first cut-and-choose stage (in particular, $m \leq \nu t/2$), and (3) in the second stage, there exists a bucket that contains all bad circuits. It is easy to see that the probability with which m bad circuits escape detection in the first stage cut-and-choose is $\binom{\nu t-m}{\nu t/2}/\binom{\nu t}{\nu t/2}$. Conditioned on this event happening, the probability that a particular bucket contains all bad circuits is $\binom{m}{\nu/2}/\binom{\nu t/2}{\nu/2}$. Applying the union bound, we conclude that the probability that P_1 succeeds in cheating is bounded by

$$t\binom{\nu t - m}{\nu t/2}\binom{m}{\nu/2} \Big/ \binom{\nu t}{\nu t/2}\binom{\nu t/2}{\nu/2}.$$

For any given t and ρ, the smallest ν, hinging on the maximal probability of P_1's successful attack, can be determined by enumerating over all possible values of m (in particular, $\{\nu/2, \nu/2+1, \ldots, \nu t/2\}$).

As an example, for $t = 20$ with $\rho = 40$, using our protocol the circuit generator needs to construct $16 \cdot t = 320$ garbled circuits, whereas using a naive application of Lindell's protocol [10] requires $40 \cdot t = 800$ garbled circuits. See Figure 3.1 for a comparison of our approach and the prior work for various settings of t.

Parallel versus sequential executions. As will be evident, it is important to distinguish between the settings where the parties carry out multiple evaluations in parallel (e.g., when all inputs are available at the start of the protocol) and where these evaluations are carried out sequentially (e.g., when not all inputs are available as they, for example, depend on the outputs of previous executions). Below, we provide an overview of the main challenges of each setting, and an outline of our

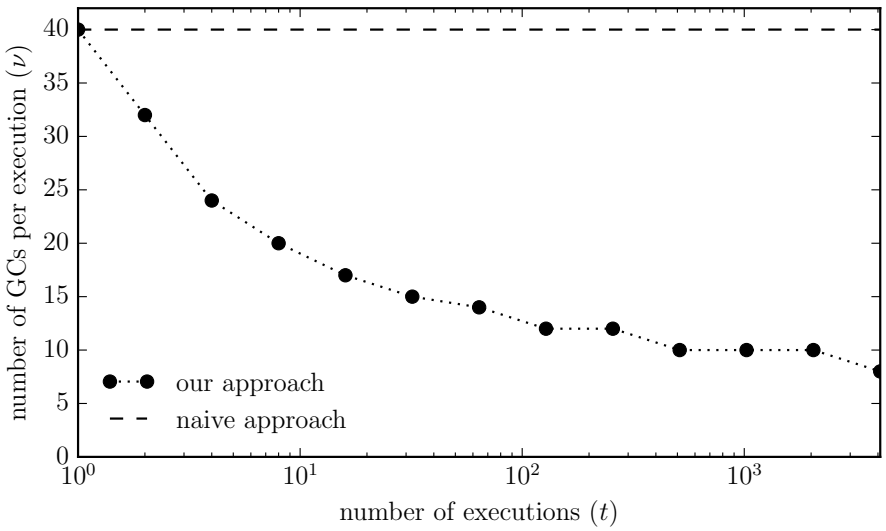


Figure 3.1: Graph depicting the number of garbled circuits required *per execution* for statistical security 2^{-40} for our approach and the naive approach which uses the fast cut-and-choose protocol [10] for each setting of t.

solutions.

Parallel executions. We apply our cut-and-choose technique in the parallel execution setting by modifying the fast cut-and-choose protocol [10] as follows. Lindell uses a primitive called *cut-and-choose oblivious transfer* (C&C OT) to prevent a malicious P_1 from learning a bit of P_2's input using the so called "selective failure attack" (discussed in more detail in Section 3.3). In this work, we construct a *generalized* C&C OT functionality that supports *multi-stage* cut-and-choose. We call this functionality $\mathcal{F}_{\mathbf{mcot}}$, and show an efficient realization that is only a factor $\nu t^2/\rho$ less efficient (per execution) than the C&C OT realization of Lindell [10]. We elaborate more on this, and other important details, in Section 3.3.

Sequential executions. To prevent a malicious evaluator from choosing its inputs

based on the garbled circuit (which is required in order to prove security), garbled circuit-based 2PC protocols perform oblivious transfer *before* the garbler sends its garbled circuits to the evaluator (i.e., before the cut-and-choose phase). This forces the parties, and in particular the evaluator, to "commit" to their inputs before performing the cut-and-choose. This, however, does not work in the sequential setting, where the parties may not know all their inputs at the beginning of the protocol. Standard solutions used in previous works [34, 35, 36] include assuming the garbled circuit is adaptively secure or using adaptively-secure garbling [37] explicitly, assuming the programmable random-oracle model. Another issue is that since now we perform OTs for each execution separately, we can no longer use C&C OT or its variants; instead we rely on the "XOR-tree" approach of Lindell and Pinkas [8] to avoid selective failure attacks. We elaborate more on this, and other details, in Section 3.4.

Our solution for the sequential setting readily carries over to the parallel setting. In particular, adapting our protocol from the sequential to the parallel setting may address situations where the cost incurred by the use of $\mathcal{F}_{\mathbf{mcot}}$ outweighs the cost of using both the XOR-tree approach and adaptively-secure garbled circuits.

Related work. Lindell and Pinkas [8] gave the first[3] rigorous 2PC protocol based on cut-and-choose. For $\rho = 40$, their protocol required at least $17\rho = 680$ garbled circuits. Subsequent work by the same authors [9] reduced the number of circuits

[3]Cut-and-choose mechanisms were previously employed in works by Pinkas [38] and Malkhi et al. [4] but these approaches were later shown to be flawed [39, 40].

to 128. This was later improved by shelat and Shen [32] to 125 using a more precise analysis of the cut-and-choose approach. In Crypto 2013, two works [15, 10] proposed (among other things) improvements to the number of garbled circuits that need to be sent. For achieving statistical security $2^{-\rho}$, Huang et al.'s protocol [15] requires $2\rho+O(\log \rho)$ circuits, where each party generates half of them, and Lindell's protocol [10] requires exactly ρ circuits, plus an additional (but inexpensive) recovery phase.

While all of the above works perform cut-and-choose over circuits, applying cut-and-choose at the gate-level has also been considered [41, 16, 33, 17]. As discussed above, this approach naturally extends to the multiple-execution setting, and furthermore is not inherently limited to considering settings where the same function is evaluated multiple times. Nielsen et al. [33] indeed show concrete efficiency improvements using gate-level cut-and-choose techniques. However, the number of rounds grows linearly with the depth of the evaluated circuit.

Finally, in independent and concurrent work, Lindell and Riva [42] also investigate the multiple-execution setting, and obtain performance improvements similar to ours. An interesting difference between our works is that while we always let the evaluator pick half the circuits to check, they show that varying the number of check circuits can lead to an additional performance improvement. In subsequent work, Lindell and Riva [43] introduced a new efficient input consistency mechanism and implemented their construction, showing that AES can be securely evaluated online in only 7 ms per execution, thus demonstrating the practicality of this approach.

3.2 Preliminaries

We consider the setting where a function is executed t times over different inputs. We assume that only one party (the evaluator) receives output. Known techniques [8] can be used to lift this setting to one in which both parties receive output.

Our constructions make use of three (standard) two-party ideal functionalities for oblivious transfer, zero-knowledge proof-of-knowledge of an exponent, and coin tossing; see below. All three functionalities have efficient and standard instantiations [9, 44, 45].

$\mathcal{F}_{\mathbf{ot}}$	On sender input (x_0, x_1) and receiver input σ, send x_σ to the receiver.
$\mathcal{F}_{\mathbf{zk}}$	On prover input $(\{g^{a_0 \cdot r_j}, g^{a_1 \cdot, r_j}, g^{a_0 \cdot r_{j'}}, g^{a_1 \cdot r_{j'}}\}, \sigma)$ and receiver input $(X_j, X_{j'}, Y_0, Y_1, Z_j, Z_{j'})$, send 1 to the receiver if it holds that $Y_0 = g^{a_0}$, $Y_1 = g^{a_1}$, $Z_j = g^{r_j}$, $Z_{j'} = g^{r_{j'}}$, $X_j = g^{a_\sigma \cdot r_j}$, and $X_{j'} = g^{a_\sigma \cdot r_{j'}}$, and 0 otherwise.
$\mathcal{F}_{\mathbf{ct}}$	Output random string r to both parties.

We also make use of *adaptively secure garbled circuits* [37], which we now define. These are similar to the garbled circuit notion (cf. Chapter 2) used throughout this work, except that the evaluator may decide on its choice of input *after* receiving the garbled circuit, and can thus base its input on a function of the received garbled circuit. We consider the *fine-grained* variant of adaptively secure garbled circuits, where the adversary can choose its input bit-by-bit (namely, it receives the input-wire label for bit i before choosing its value for bit $i + 1$).

We augment the standard privacy notion for garbling schemes (cf. Defini-

tion 2.2) as follows. In the adaptive case, the simulator $\mathcal{S}$ does not have access to the output $f(x, y)$ of the computation until the adversary $\mathcal{A}$ has specified its entire input. Thus, it must construct a fake garbled circuit "blindly." Only once $\mathcal{A}$ specifies all the input bits does the simulator learn $f(x, y)$, and at this point it must "fix" the garbled circuit to produce this as the output. Namely, $\mathcal{A}$ is given oracle access to an $\mathsf{Input}(w, b)$ function which returns the b-label of the wth input wire. This oracle can only be called once per input wire w. In the real world, $\mathcal{A}$ receives the actual input-wire label generated by Gb; that is, $X_{w,b} := \mathsf{Input}(w, b)$. In the ideal world, we have $\mathcal{S}$ "simulate" the output of Input, given only the wire index; only once labels are output for all input-wire labels does $\mathcal{S}$ learn the output $f(x, y)$. More concretely, we split $\mathcal{S}$ into two simulators, $\mathcal{S}_1$ and $\mathcal{S}_2$. $\mathcal{S}_1$ is only given as input the security parameter and must generate a garbled circuit $\widehat{C}$. $\mathcal{S}_2$ is called on each call $\mathcal{A}$ makes to Input, and receives as input the wire index w, the number of calls Q made to Input, and $f(x, y)$ if Q equals the length of the input and $\perp$ otherwise, and must output a label in such a way that $\mathcal{A}$ cannot distinguish between the two worlds.

Definition 3.1. *Garbling scheme* $(\mathsf{Gb}, \mathsf{Ev})$ *satisfies* ***adaptive privacy*** *if for every* PPT *adversary* $\mathcal{A}$*, there exists a* PPT *simulator* $\mathcal{S}$ *such that for all polynomial size func-*

tions f and inputs x and y it holds that

$$\Pr\left[\mathcal{A}^{\mathsf{Input}(\cdot,\cdot)}(\widehat{C})=1 : \begin{array}{c}(\widehat{C},\{X_{w,b}\})\leftarrow \mathsf{Gb}(1^\kappa,f)\\ X_{w,b}:=\mathsf{Input}(w,b)\end{array}\right]$$

$$\overset{c}{\approx}$$

$$\Pr\left[\mathcal{A}^{\mathsf{Input}(\cdot,\cdot)}(\widehat{C})=1 : \begin{array}{c}\widehat{C}\leftarrow \mathcal{S}_1(1^\kappa)\\ X_{w,b}\leftarrow \mathcal{S}_2(f(x,y)\ \mathit{or}\ \perp,w,Q)\end{array}\right].$$

Relatively efficient constructions of adaptively secure garbled circuits can be constructed in the random oracle model [37]. The basic idea is to mask the garbled circuit by some random string that is only revealed once the adversary receives all of its input-wire labels.

3.2.1 Security Definitions

Our security definitions allow one of the two participating parties to be corrupted by an adversary $\mathcal{A}$. We assume that there is an environment $\mathcal{Z}$ which interacts with $\mathcal{A}$ and the honest party in the way specified below. At the end of the execution, $\mathcal{Z}$ needs to distinguish between the case where $\mathcal{A}$ runs a protocol with the real honest party, and the case where $\mathcal{A}$ and the honest party invoke an ideal functionality that computes the function f, where the protocol is secure if $\mathcal{Z}$'s advantage in distinguishing the two cases is negligible.

The Parallel Execution Setting

Ideal model execution. In the ideal model, we have parties P_1 and P_2, and an adversary $\mathcal{A}$ who can corrupt one of the two parties. An ideal execution for the computation of the function f multiple times in parallel, where the parties have access to an ideal functionality $\mathcal{F}_{\mathbf{par}}$, proceeds as follows.

Auxiliary Input: P_1 and P_2 hold 1^κ, and $\mathcal{Z}$ holds auxiliary input aux. In addition, $\mathcal{Z}$ provides P_1 and P_2 a parameter t which denotes the number of times the function f is executed.

- P_1 and P_2 obtain inputs $(x_1, \ldots, x_t)$ and $(y_1, \ldots, y_t)$, respectively, from $\mathcal{Z}$, where each x_i and y_i is of length $\{0,1\}^n$.
- The honest party sends its input vector to $\mathcal{F}_{\mathbf{par}}$. The corrupted party may send any input vector of its choice.
- If an input is invalid, $\mathcal{F}_{\mathbf{par}}$ outputs $\perp$ to both parties and halts. Otherwise, $\mathcal{F}_{\mathbf{par}}$ sends $f(x_1, y_1), \ldots, f(x_t, y_t)$ to P_2.
- P_1 has no output, and P_2 has output $f(x_1, y_1), \ldots, f(x_t, y_t)$. The honest party gives whatever it was sent by $\mathcal{F}_{\mathbf{par}}$ to $\mathcal{Z}$, and the corrupted party gives an arbitrary function of its view to $\mathcal{Z}$. In the end, $\mathcal{Z}$ outputs a bit. We let $\textsc{Ideal}_{f,\mathcal{A},\mathcal{Z}(\mathsf{aux})}(1^\kappa)$ denote the output of $\mathcal{Z}$.

Real model execution. In the real model, we have parties P_1 and P_2 who execute a two-party protocol Π_f. The protocol Π_f has a parameter t initialized by $\mathcal{Z}$ which

specifies the number of times f is evaluated in parallel. P_1 and P_2 obtain their inputs $(x_1, \ldots, x_t)$ and $(y_1, \ldots, y_t)$, respectively, from $\mathcal{Z}$, and obtain output $(z_1, \ldots, z_t)$ by executing Π_f using their respective inputs. The honest party sends its output to $\mathcal{Z}$ and the adversary $\mathcal{A}$ sends its view to $\mathcal{Z}$. Throughout the protocol execution, $\mathcal{A}$ obtains the inputs of the corrupted party and sends all messages on its behalf, whereas the honest party follows the instructions of Π_f. In the end, $\mathcal{Z}$ outputs a bit. We let $\text{REAL}_{\Pi_f,\mathcal{A},\mathcal{Z}(\mathsf{aux})}(1^\kappa)$ denote the output of $\mathcal{Z}$.

Definition 3.2. *Protocol Π_f is said to securely compute $\mathcal{F}_{\mathbf{par}}$ if for every* PPT *adversary $\mathcal{A}$ in the real model, there exists a* PPT *adversary $\mathcal{S}$ in the ideal model such that for every $\mathsf{aux} \in \{0,1\}^*$, $\kappa, \rho \in \mathbb{N}$, and non-uniform* PPT *environment $\mathcal{Z}$ that specifies the number of executions as $t \in \mathsf{poly}(\kappa)$, it holds that*

$$\{\text{IDEAL}_{f,\mathcal{S},\mathcal{Z}(\mathsf{aux})}(1^\kappa)\} \overset{c}{\approx} \{\text{REAL}_{\Pi_f,\mathcal{A},\mathcal{Z}(\mathsf{aux})}(1^\kappa)\} + 2^{-\rho}.$$

Remarks. The definition above is somewhat similar to security definitions in the Universal Composability (UC) framework [46] in the way we define security as the success probability of an environment $\mathcal{Z}$ that attempts to distinguish between the ideal world and the real world. In spite of this we stress that our definition is *not* as strong as the UC definition, as the latter allows $\mathcal{Z}$ to interact arbitrarily with $\mathcal{A}$ during the protocol execution.

The Sequential Execution Setting

Ideal model execution. In the ideal model, we have parties P_1 and P_2, and an adversary $\mathcal{A}$ who can corrupt one of the two parties. An ideal execution for the computation of the function f multiple times sequentially, where the parties have access to an ideal functionality $\mathcal{F}_{\mathbf{seq}}$, proceeds as follows.

Auxiliary Input: P_1 and P_2 hold 1^κ and are *stateful*, and $\mathcal{Z}$ holds auxiliary input aux. In addition, $\mathcal{Z}$ provides P_1 and P_2 a parameter t which denotes the number of times the function f is executed.

For $k \in [t]$:

- P_1 and P_2 obtain inputs $x_k \in \{0,1\}^n$ and $y_k \in \{0,1\}^n$, respectively, from $\mathcal{Z}$.
- The honest party sends its input to $\mathcal{F}_{\mathbf{seq}}$. The corrupted party may send any input of its choice.
- If an input is invalid, $\mathcal{F}_{\mathbf{seq}}$ outputs $\perp$ to both parties and halts. Otherwise, $\mathcal{F}_{\mathbf{seq}}$ sends $f(x_k, y_k)$ to P_2.
- P_1 has no output, and P_2 has output $f(x_k, y_k)$. The honest party gives whatever it was sent by $\mathcal{F}_{\mathbf{seq}}$ to $\mathcal{Z}$, and the corrupted party gives an arbitrary function of its view to $\mathcal{Z}$.

At the end of t iterations, $\mathcal{Z}$ outputs a bit. We let $\textsc{Ideal}_{f,\mathcal{A},\mathcal{Z}(\mathsf{aux})}(1^\kappa)$ denote the output of $\mathcal{Z}$.

Real model execution. In the real model, we have parties P_1 and P_2 who execute a two-party protocol Π_f. The protocol Π_f has a parameter t, initialized by $\mathcal{Z}$, which specifies the number of times f is evaluated. Protocol Π_f is *stateful* across its execution spanning t stages. In each stage, P_1 and P_2 obtain their inputs x_k respectively y_k from $\mathcal{Z}$, and obtain their output z_k by executing Π_f using their respective inputs. At the end of each stage, the honest party sends its output to $\mathcal{Z}$ and the adversary sends its view to $\mathcal{Z}$. Throughout the protocol execution, $\mathcal{A}$ obtains the inputs of the corrupted party and sends all messages on its behalf, whereas the honest party follows the instructions of Π_f. At the end of t stages of Π_f, $\mathcal{Z}$ outputs a bit. We let $\text{REAL}_{\Pi_f,\mathcal{A},\mathcal{Z}(\mathsf{aux})}(1^\kappa)$ denote the output of $\mathcal{Z}$.

Definition 3.3. *Protocol Π_f is said to securely compute $\mathcal{F}_{\mathbf{seq}}$ if for every* PPT *adversary $\mathcal{A}$ in the real model, there exists a* PPT *adversary $\mathcal{S}$ in the ideal model such that for every $\mathsf{aux} \in \{0,1\}^*$, $\kappa, \rho \in \mathbb{N}$, and non-uniform* PPT *environment $\mathcal{Z}$ that specifies the number of executions as $t \in \mathsf{poly}(\kappa)$, it holds that*

$$\{\text{IDEAL}_{f,\mathcal{S},\mathcal{Z}(\mathsf{aux})}(1^\kappa)\} \overset{c}{\approx} \{\text{REAL}_{\Pi_f,\mathcal{A},\mathcal{Z}(\mathsf{aux})}(1^\kappa)\} + 2^{-\rho}.$$

Remarks. As in the parallel execution case, this definition differs from the definition in the Universal Composability (UC) framework [46], since in our setting we restrict $\mathcal{Z}$ to interact with $\mathcal{A}$ only between stages of the protocol Π_f, but never within a stage.

3.3 The Parallel Execution Setting

Consider a setting where two parties wish to securely evaluate the same function multiple times *in parallel*. Let f denote the function of interest, let t denote the number of times the parties wish to evaluate f, and let P_1's and P_2's input in the kth execution be x_k and y_k, respectively.

We adapt Lindell's protocol [10] to support our cut-and-choose technique in the parallel execution setting. The main difficulty is the design and construction of a generalization of cut-and-choose oblivious transfer [9] which we use to avoid the "selective failure attack" where a malicious P_1 constructs invalid labels for some of P_2's input wires to try to deduce P_2's inputs based on whether P_2 aborts execution or not.

Generalizing Cut-and-Choose Oblivious Transfer

Cut-and-choose oblivious transfer (C&C OT) [9] is an extension of standard one-out-of-two oblivious transfer (OT). The sender inputs n pairs of strings, and the receiver inputs n selection bits to select one string out of each pair of sender strings. The receiver also inputs a set $\mathcal{C}$ of size $n/2$ that consists of indices where it wants *both* the sender's inputs to be revealed. We denote this set as the *check set*, and let $\mathcal{E} := [\rho]\backslash\mathcal{C}$ denote the *evaluation set*. Note that for indices in $\mathcal{E}$, only those sender inputs that correspond to the receiver's selection bits are revealed. In applications to secure computation, and in particular when transferring input-wire labels corresponding to a particular input wire across all evaluation circuits, one needs *single-choice* cut-

Functionality $\mathcal{F}_{\mathbf{ccot}}$

P_1 inputs n vectors $\{X^0_{i,1}, X^1_{i,1}, \ldots, X^0_{i,\rho}, X^1_{i,\rho}\}_{i\in[n]}$ and ρ "check values" $\chi_1, \ldots, \chi_\rho$, where each string is in $\{0,1\}^\rho$. P_2 inputs bits $\sigma_1, \ldots, \sigma_n \in \{0,1\}$ and a set $\mathcal{C} \subseteq [\rho]$.

P_1 receives no output. P_2 receives the following:

- For $i \in [n]$ and $j \in \mathcal{C}$, P_2 receives $(X^0_{i,j}, X^1_{i,j})$.
- For $i \in [n]$, P_2 receives $(X^{\sigma_i}_{i,1}, \ldots, X^{\sigma_i}_{i,\rho})$, and for $j \notin \mathcal{C}$, P_2 receives χ_j.

Figure 3.2: Modified batch single-choice cut-and-choose oblivious transfer functionality $\mathcal{F}_{\mathbf{ccot}}$ [10].

and-choose oblivious transfer, where the receiver is restricted to inputting the *same* selection bit in all the $n/2$ instances where it receives exactly one of the sender's strings. Furthermore, when transferring labels for multiple input wires, it is crucial that the check set $\mathcal{C}$ input by the receiver is the same across each instance of single-choice C&C OT to enforce that the receiver receives only one input-wire label in each evaluation circuit and both input-wire labels in each check circuit across all input wires. This variant, called *batch single-choice* C&C OT, can be realized from the decisional Diffie-Hellman assumption [9].

Lindell [10] presented a variant of batch single-choice C&C OT [9] in order to address settings where the check set $\mathcal{C}$ input by the receiver may be of arbitrary size. We denote this variant by $\mathcal{F}_{\mathbf{ccot}}$; see Figure 3.2 for the formal description. In this variant, in addition to obtaining one of the two sender inputs for pairs whose indices are not in $\mathcal{C}$, the receiver also obtains a "check value" for each index in $\mathcal{E}$. These check values are used to confirm that a circuit is indeed an evaluation circuit.

For our purposes, we introduce a new variant of C&C OT, which we call batch single-choice *multi-stage* C&C OT. We denote this primitive by $\mathcal{F}_{\mathbf{mcot}}$ and present its formal description in Figure 3.3. As we use $\mathcal{F}_{\mathbf{mcot}}$ to realize our parallel execution

Functionality $\mathcal{F}_{\mathbf{mcot}}$

P_1 inputs n vectors $\{X^0_{i,1}, X^1_{i,1}, \ldots, X^0_{i,\nu t}, X^1_{i,\nu t}\}_{i\in[n]}$, and νt^2 "check values" $\{\chi^k_1, \ldots, \chi^k_{\nu t}\}_{k\in[t]}$, where each string is in $\{0,1\}^\rho$. P_2 inputs t n-bit vectors $\{\sigma_{1,i}, \ldots, \sigma_{n,i}\}_{i\in[t]}$ and sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ that are disjoint subsets of $[\nu t]$.

P_1 receives no output. P_2 receives the following:

- If $\mathcal{E}_1, \ldots, \mathcal{E}_t$ are disjoint subsets of $[\nu t]$, then for $k \in [t]$, $j \in \mathcal{E}_k$, P_2 receives χ^k_j. For all k, k' such that $j \in \mathcal{E}_k \cap \mathcal{E}_{k'}$, P_2 receives random strings in $\{0,1\}^\kappa$ instead of χ^k_j and $\chi^{k'}_j$.
- Let $\mathcal{C} := [\nu t] \setminus \cup_{k\in[t]}\mathcal{E}_k$. For $i \in [n]$, $j \in [\nu t]$:
  - If $j \in \mathcal{C}$, then P_2 receives $(X^0_{i,j}, X^1_{i,j})$.
  - If $j \in \mathcal{E}_k$, then P_2 receives $X^{\sigma_{i,k}}_{i,j}$.

Figure 3.3: Batch single-choice multi-stage cut-and-choose OT functionality $\mathcal{F}_{\mathbf{mcot}}$.

protocol, we use the same notation in our definition of $\mathcal{F}_{\mathbf{mcot}}$; namely, we let the universal set be of size νt rather than ρ.

At a high level, $\mathcal{F}_{\mathbf{mcot}}$ differs from $\mathcal{F}_{\mathbf{ccot}}$ in that the receiver now inputs multiple *evaluation* sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ (where the check set $\mathcal{C}$ is now implicitly defined as $\mathcal{C} := [\nu t] \setminus \cup_{k\in[t]}\mathcal{E}_k$) and makes independent selections for each $\mathcal{E}_1, \ldots, \mathcal{E}_t$. As in the $\mathcal{F}_{\mathbf{ccot}}$ functionality, $\mathcal{F}_{\mathbf{mcot}}$ (1) does not require sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ to be of a particular size, and (2) delivers "check values" for indices contained in each of $\mathcal{E}_1, \ldots, \mathcal{E}_t$. These check values are used to confirm whether a circuit is an evaluation circuit in the kth bucket for some $k \in [t]$. Note that we need νt^2 check values, rather than just νt, in order to prove security of our functionality and its use in our protocol. Namely, we need to enforce that the evaluation sets are disjoint, and we do this by enforcing that a malicious P_2 who inputs intersecting sets is unable to recover an appropriate check value. To do this, we need νt check values *for each* $k \in [t]$, rather than just νt total check values.

The $\mathcal{F}_{\mathbf{mcot}}$ functionality. As in $\mathcal{F}_{\mathbf{ccot}}$, the sender P_1 inputs n vectors $\{X^0_{i,1}, X^1_{i,1},$

$\ldots, X^0_{i,\nu t}, X^1_{i,\nu t}\}_{i\in[n]}$, where each value in the vector corresponds to the wire labels for P_2's ith input in our secure computation protocol. In addition, P_1 inputs νt^2 "check values". The receiver P_2 inputs t vectors $\vec{\sigma}_1, \ldots, \vec{\sigma}_t$ each of length n, corresponding to its input in each of the t runs, and disjoint sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$, corresponding to the t evaluation buckets. Upon receiving these inputs from P_1 and P_2, the functionality computes check set $\mathcal{C} := [\nu t] \setminus \cup_{k\in[t]}\mathcal{E}_k$ and delivers the following to P_2: (1) for $j \in \mathcal{C}$, both values in the jth pair in each of the n vectors, and (2) for $k \in [t]$ and $j \in \mathcal{E}_k$, the $\sigma_{i,k}$ value in the jth pair of each of the n vectors along with the check value χ^k_j. If P_2's evaluation sets are *not* disjoint, then it receives no check values for those circuits corresponding to values in the intersection of two of the evaluation sets.

Realizing $\mathcal{F}_{\mathbf{mcot}}$ in the $\mathcal{F}_{\mathbf{ccot}}$-hybrid model. We now proceed to construct a protocol for $\mathcal{F}_{\mathbf{mcot}}$. Our goal is to provide an information-theoretic reduction from $\mathcal{F}_{\mathbf{mcot}}$ to $\mathcal{F}_{\mathbf{ccot}}$. We first consider a naive approach which serves as a warm-up to our final construction.

The naive approach. We propose the following natural approach to realizing $\mathcal{F}_{\mathbf{mcot}}$ from $\mathcal{F}_{\mathbf{ccot}}$. P_1 first performs a t-out-of-t additive secret sharing of its input vectors. Next, P_1 and P_2 interact with the $\mathcal{F}_{\mathbf{ccot}}$ functionality t times. In the kth interaction, P_1 provides the kth additive share of its input vectors plus νt check values $\chi^k_1, \ldots, \chi^k_{\nu t}$, while P_2 provides $(\sigma_{1,k}, \ldots, \sigma_{n,k})$ along with a set $[\nu t] \setminus \mathcal{E}_k$. Let $\mathcal{C} := [\nu t] \setminus \cup_{k\in[t]}\mathcal{E}_k$. At the end of the interaction, P_2 obtains (1) all t additive shares of P_1's inputs for $j \in \mathcal{C}$, and (2) all t additive shares of P_1's inputs corresponding to P_2's selection bit, along with the check values, for $j \notin \mathcal{C}$.

In the context of using $\mathcal{F}_{\mathbf{mcot}}$ for our setting, note that for the check circuits (which correspond to the set $\mathcal{C}$), P_2 does not obtain the check values, and for the evaluation circuits (which correspond to the sets $\mathcal{E}_k$), P_2 does not obtain both input labels. Thus, the above protocol seems to successfully fulfill our requirements from the $\mathcal{F}_{\mathbf{mcot}}$ functionality. However, note that there is no mechanism in place to enforce that P_2 supplies *disjoint* sets $\mathcal{E}_k$. We show that this prevents the above protocol from realizing $\mathcal{F}_{\mathbf{mcot}}$.

Let $t := 2$. A malicious P_2 may input overlapping sets $\mathcal{E}_1, \mathcal{E}_2$ to $\mathcal{F}_{\mathbf{ccot}}$. The consequence of this is that P_2 now possesses check values χ_j^1 and χ_j^2 for $j \in \mathcal{E}_1 \cap \mathcal{E}_2$. Clearly, the functionality $\mathcal{F}_{\mathbf{mcot}}$ does not allow this. However, one may wonder why this need be the case. Recall that P_1's inputs (i.e., the labels corresponding to P_2's inputs when used in our protocol) are all secret shared, and as a result P_2 does not possess valid labels corresponding to its input in garbled circuit $\widehat{C}_j$ unless its input in both executions is identical. At the surface, there does not seem to be any attack due to this malicious strategy. While P_2 can equivocate on assigning $\widehat{C}_j$ to either the first or second evaluation bucket, it either has no corresponding labels, or it has to evaluate both circuits on the same input, say y (in which case it seems immaterial whether j is revealed as part of $\mathcal{E}_1$ or $\mathcal{E}_2$).

Unfortunately, the above malicious strategy is not simulatable when used in our secure computation protocol. In particular, at the end of the interaction with $\mathcal{F}_{\mathbf{ccot}}$, the simulator successfully extracts P_2's input in the first and second execution, but is now unable to decide on how to fake the garbled circuit $\widehat{C}_j$. On the one hand, if $j \in \mathcal{E}_1$, then the fake garbled circuit has to output $z_1 := f(x_1, y)$. On the other

hand, if $j \in \mathcal{E}_2$, then the fake garbled circuit has to output $z_2 := f(x_2, y)$. Therefore, the simulator has to choose on how to fake $\widehat{C}_j$ "in the dark," which does not extend well to the case where t is large.

The discussion above motivates our definition of $\mathcal{F}_{\mathbf{mcot}}$; in particular, it reinforces why we need disjoint evaluation sets and why $\mathcal{F}_{\mathbf{mcot}}$ must deliver at most one check value per circuit.

Our approach. The high level idea behind our protocol is to let P_1 perform independent additive sharings of both the input values *and* the check values. Then P_1 and P_2 query the $\mathcal{F}_{\mathbf{ccot}}$ functionality t times to transfer the values as required by $\mathcal{F}_{\mathbf{mcot}}$. We detail this below, explaining it in the context of our secure computation protocol.

Let $(X^0_{i,j}, X^1_{i,j})$ be the input labels corresponding to P_2's ith input wire in garbled circuit $\widehat{C}_j$. First, P_1 performs a t-out-of-t additive secret sharing of these labels; that is, for $i \in [n]$ and $j \in [\nu t]$, P_1 secret shares $X^0_{i,j}$ and $X^1_{i,j}$ into $\{X^0_{i,j,k}\}_{k\in[t]}$ and $\{X^1_{i,j,k}\}_{k\in[t]}$, respectively. P_1 also chooses νt^2 check values $\{\chi^k_1, \ldots, \chi^k_{\nu t}\}_{k\in[t]}$ and performs a $(2n(t-1)+1)$-out-of-$(2n(t-1)+1)$ additive sharing of each value χ^k_j to obtain shares $\widetilde{\chi}^k_j$, $\{\chi^{0,k}_{i,j,k'}, \chi^{1,k}_{i,j,k'}\}_{k'\in[t]\setminus\{k\},i\in[n]}$. Then, instead of creating inputs to $\mathcal{F}_{\mathbf{ccot}}$ using the $X^b_{i,j,k}$ shares alone, P_1 creates a *share-block* $\mathbf{X}^b_{i,j,k} := (X^b_{i,j,k}, \chi^{b,1}_{i,j,k}, \ldots, \chi^{b,t}_{i,j,k})$. That is, a share-block $\mathbf{X}^b_{i,j,k}$ contains, in addition to a share of the input label, a share of all check values corresponding to garbled circuit $\widehat{C}_j$.

Next, P_1 and P_2 run t instances of $\mathcal{F}_{\mathbf{ccot}}$. In the kth interaction, in addition to the νt check value shares $\widetilde{\chi}^k_1, \ldots, \widetilde{\chi}^k_{\nu t}$, P_1 provides its kth share-block while P_2

provides its inputs for the kth execution along with a set $[\nu t] \setminus \mathcal{E}_k$. Let $\mathcal{C} := [\nu t] \setminus \cup_{k\in[t]}\mathcal{E}_k$. At the end of the interaction, P_2 obtains (1) for $j \in \mathcal{C}$, all t share-blocks of input-wire labels, and therefore all input-wire labels, for garbled circuit $\widehat{C}_j$, and (2) for $j \in \mathcal{E}_k$, all t share-blocks of input-wire labels that *correspond to its actual input* in the kth execution, and therefore its input-wire labels, along with a check value $\widetilde{\chi}_j^k$ for garbled circuit $\widehat{C}_j$.

Note, in particular, that for each *check* circuit $\widehat{C}_j$, P_2 does not obtain the check value χ_j^k for any k, because it always misses the check value share $\widetilde{\chi}_j^k$. For each *evaluation* circuit $\widehat{C}_j$ with $j \in \mathcal{E}_k$, P_2 does not obtain both input labels, and more importantly can obtain at most one check value (which is χ_j^k). This is because share-blocks contain shares of input labels as well as shares of check values. For an evaluation circuit, P_2 always misses a share block, and consequently shares of all values $\chi_j^{k'}$ with $k' \neq k$. Furthermore, if P_2 wants to ensure it receives χ_j^k, then it should never input $\mathcal{E}_{k''}$ such that $k'' \neq k$ and yet $j \in \mathcal{E}_{k''}$. This is because for $j \in \mathcal{E}_{k''}$, P_2 is guaranteed to miss a share block that contains an additive share of χ_j^k. Note that the above observations suffice to deal with a malicious P_2 that inputs overlapping sets since in this case P_2 fails to obtain any check values corresponding to indices in the intersection.

As an example, consider the case where $n := 1$ and $t := 2$. Then we secret-share χ_j^1 as $(\widetilde{\chi}_j^1, \chi_{1,j,2}^{0,1}, \chi_{1,j,2}^{1,1})$ and χ_j^2 as $(\widetilde{\chi}_j^2, \chi_{1,j,1}^{0,2}, \chi_{1,j,1}^{1,2})$. Likewise, share-block $\mathbf{X}_{1,j,1}^b$ equals $(X_{1,j,1}^b, \chi_{1,j,1}^{b,2})$ and share-block $\mathbf{X}_{1,j,2}^b$ equals $(X_{1,j,2}^b, \chi_{1,j,2}^{b,1})$. Now, if P_2 inputs evaluation sets such that $j \in \mathcal{E}_1 \cap \mathcal{E}_2$, then it recovers, say, $\mathbf{X}_{1,j,1}^0 := (X_{1,j,1}^0, \chi_{1,j,1}^{0,2})$ and $\mathbf{X}_{1,j,2}^0 := (X_{1,j,2}^0, \chi_{1,j,2}^{0,1})$ for input bit 0. Note that it does not have enough shares of

either χ_j^1 or χ_j^2 to recover either, and thus does not learn the check value. However, if $\mathcal{E}_1$ and $\mathcal{E}_2$ are disjoint, then it would also learn, say, $\mathbf{X}_{1,j,2}^1 := (X_{1,j,2}^1, \chi_{1,j,2}^{1,1})$, and thus be able to recover χ_j^1.

See below for the formal description.

Inputs:

- P_1 inputs n vectors $\{\vec{X}_i := (X_{i,1}^0, X_{i,1}^1, \ldots, X_{i,\nu t}^0, X_{i,\nu t}^1)\}_{i\in[n]}$ and νt^2 "check values" $\{\chi_1^k, \ldots, \chi_{\nu t}^k\}_{k\in[t]}$, where each string is in $\{0,1\}^\kappa$.
- P_2 inputs t n-bit vectors $\{\sigma_{1,i}, \ldots, \sigma_{n,i}\}_{i\in[t]}$ and sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ that are disjoint subsets of $[\nu t]$.

Protocol:

- For $i \in [n]$, P_1 performs a t-out-of-t additive secret sharing of $\vec{X}_i$ to obtain shares $\vec{X}_{i,1}, \ldots, \vec{X}_{i,t}$. For $k \in [t]$, let $\vec{X}_{i,k} = (X_{i,1,k}^0, X_{i,1,k}^1), \ldots, (X_{i,\nu t,k}^0, X_{i,\nu t,k}^1)$, let $\mathbf{X}_{i,j,k}^0 = (X_{i,j,k}^0, \chi_{i,j,k}^{0,1}, \ldots, \chi_{i,j,k}^{0,t})$, and let $\mathbf{X}_{i,j,k}^1 = (X_{i,j,k}^1, \chi_{i,j,k}^{1,1}, \ldots, \chi_{i,j,k}^{1,t})$, where $\chi_{i,j,k}^{0,1}, \ldots, \chi_{i,j,k}^{0,t}$ and $\chi_{i,j,k}^{1,1}, \ldots, \chi_{i,j,k}^{1,t}$ are random independent values in $\{0,1\}^\kappa$. Let $\vec{\mathbf{X}}_{i,k} = (\mathbf{X}_{i,1,k}^0, \mathbf{X}_{i,1,k}^1), \ldots, (\mathbf{X}_{i,\nu t,k}^0, \mathbf{X}_{i,\nu t,k}^1)$.
- For $k \in [t]$, $j \in [\nu t]$, P_1 sets $\tilde{\chi}_j^k = \chi_j^k \oplus \bigoplus_{k'\in[t]\setminus\{k\}, i\in[n]} (\chi_{i,j,k'}^{0,k} \oplus \chi_{i,j,k'}^{1,k})$.
- P_1 and P_2 run t instances of $\mathcal{F}_{\mathbf{ccot}}$ as follows. In the kth instance:
  - P_1 inputs n vectors $\{\vec{\mathbf{X}}_{i,k}\}_{i\in[n]}$ and νt "check values" $\widetilde{\chi}_1^k, \ldots, \widetilde{\chi}_{\nu t}^k$. P_2 inputs $\sigma_{i,k}, \ldots, \sigma_{n,k}$ and the set $[\nu t] \setminus \mathcal{E}_k$.
  - P_2 receives $\{\widetilde{\chi}_j^k\}_{j\in\mathcal{E}_k}$ and $\{\{\mathbf{X}_{i,j,k}^{\sigma_{i,k}}\}_{j\in\mathcal{E}_k} \cup \{(\mathbf{X}_{i,j,k}^0, \mathbf{X}_{i,j,k}^1)\}_{j\in[\nu t]\setminus\mathcal{E}_k}\}_{i\in[n]}$.
- For $k \in [t]$, $j \in \mathcal{E}_k$, P_2 reconstructs $\chi_j^k := \tilde{\chi}_j^k \oplus \bigoplus_{k'\in[t]\setminus\{k\}, i\in[n]} (\chi_{i,j,k'}^{0,k} \oplus \chi_{i,j,k'}^{1,k})$.
- Let $\mathcal{C} = [\nu t] \setminus \cup_{k\in[t]} \mathcal{E}_k$. For $i \in [n]$, $j \in [\nu t]$, P_2 does the following:
  - If $j \in \mathcal{C}$: set $X_{i,j}^0 := \bigoplus_{k\in[t]} X_{i,j,k}^0$, and $X_{i,j}^1 := \bigoplus_{k\in[t]} X_{i,j,k}^1$.
  - If there exists (unique) $k \in [t]$ such that $j \in \mathcal{E}_k$: set $X_{i,j}^{\sigma_{i,k}} := \bigoplus_{k\in[t]} X_{i,j,k}^{\sigma_{i,k}}$.
- P_2 outputs sets $\{\chi_j^1\}_{j\in\mathcal{E}_1}, \ldots, \{\chi_j^t\}_{j\in\mathcal{E}_t}$ and $\{\{X_{i,j}^0, X_{i,j}^1\}_{j\in\mathcal{C}}, \{X_{i,j}^{\sigma_{i,1}}\}_{j\in\mathcal{E}_1}, \ldots, \{X_{i,j}^{\sigma_{i,t}}\}_{j\in\mathcal{E}_t}\}_{i\in[n]}$.

Theorem 3.1. *The above protocol perfectly realizes $\mathcal{F}_{\mathbf{mcot}}$ in the $\mathcal{F}_{\mathbf{ccot}}$-hybrid model.*

Proof. We split the analysis into two cases depending on whether P_1 or P_2 is cor-

rupted.

P_1 is corrupted. The simulation is straightforward since P_1 does not receive any output. We describe it below. Let $\mathcal{S}$ be the simulator running an adversary $\mathcal{A}$ corrupting P_1.

- $\mathcal{S}$ initializes $\mathcal{A}$.
- For $k \in [t]$, $\mathcal{S}$ obtains the following from $\mathcal{A}$:
  1. vectors $\vec{\mathbf{X}}_{i,k} := (\mathbf{X}^0_{i,1,k}, \mathbf{X}^1_{i,1,k}), \ldots, (\mathbf{X}^0_{i,\nu t,k}, \mathbf{X}^1_{i,\nu t,k})$ for $i \in [n]$; and
  2. "check values" $\tilde{\chi}^k_1, \ldots, \tilde{\chi}^k_{\nu t}$.

  For $b \in \{0,1\}$, $i \in [n]$, $j \in [\nu t]$, and $k \in [t]$, $\mathcal{S}$ parses $\mathbf{X}^b_{i,j,k}$ as $(X^b_{i,j,k}, \chi^{b,1}_{i,j,k}, \ldots, \chi^{b,t}_{i,j,k})$.
- For $i \in [n]$, $\mathcal{S}$ constructs $\vec{X}_i := (X^0_{i,1}, X^1_{i,1}, \ldots, X^0_{i,\nu t}, X^1_{i,\nu t})$, where for $b \in \{0,1\}$ and $j \in [\nu t]$, $X^b_{i,j} := \bigoplus_{k \in [t]} X^b_{i,j,k}$.
- For $j \in [\nu t]$ and $k \in [t]$, $\mathcal{S}$ computes $\chi^k_j := \tilde{\chi}^k_j \oplus \bigoplus_{k' \in [t] \setminus \{k\}, i \in [n]} (\chi^{0,k}_{i,j,k'} \oplus \chi^{1,k}_{i,j,k'})$.
- $\mathcal{S}$ sends $\{\vec{X}_i\}_{i \in [n]}$ and $(\chi^1_1, \ldots, \chi^1_{\nu t}), \ldots, (\chi^t_1, \ldots, \chi^t_{\nu t})$ to $\mathcal{F}_{\mathbf{mcot}}$ and halts, outputting whatever $\mathcal{A}$ outputs.

$\mathcal{S}$ clearly perfectly simulates $\mathcal{A}$, as the view of $\mathcal{A}$ and output of an honest P_2 is exactly the same as in both the real and ideal worlds.

P_2 is corrupted. This simulation is slightly tricky, since an adversary $\mathcal{A}$ corrupting P_2 may input to $\mathcal{F}_{\mathbf{ccot}}$ sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ that are intersecting. For clarity, we denote the

sets input by $\mathcal{A}$ as $\mathcal{E}'_1, \ldots, \mathcal{E}'_t$. The key observation is that none of the input values or check values are determined until P_2 completes its *final* query to $\mathcal{F}_{\mathbf{ccot}}$. Due to symmetry and hence without loss of generality, in the following we assume P_2's last query to $\mathcal{F}_{\mathbf{ccot}}$ is its tth query. We describe the simulation below.

- $\mathcal{S}$ initializes $\mathcal{A}$.
- For $k \in [t-1]$, $\mathcal{S}$ acts as $\mathcal{F}_{\mathbf{ccot}}$ and interacts with P_2 in the following way:
  - $\mathcal{S}$ obtains the following from $\mathcal{A}$: (1) $\vec{\sigma}_k := \sigma_{1,k}, \ldots, \sigma_{n,k}$ and (2) the set $[\nu t] \setminus \mathcal{E}'_k$.
  - $\mathcal{S}$ chooses uniformly random and independent values $\mathbf{X}^0_{i,j,k} := (X^0_{i,j,k}, \chi^{0,1}_{i,j,k}, \ldots, \chi^{0,t}_{i,j,k})$ and $\mathbf{X}^1_{i,j,k} := (X^1_{i,j,k}, \chi^{1,1}_{i,j,k}, \ldots, \chi^{1,t}_{i,j,k})$ for $i \in [n]$ and $j \in [\nu t]$. In addition, $\mathcal{S}$ chooses uniformly random and independent values $\tilde{\chi}^k_1, \ldots, \tilde{\chi}^k_{\nu t}$.
  - $\mathcal{S}$ sends $\{\tilde{\chi}^k_j\}_{j \in \mathcal{E}'_k}$, $\{\{\mathbf{X}^{\sigma_{i,k}}_{i,j,k}\}_{j \in \mathcal{E}'_k} \cup \{(\mathbf{X}^0_{i,j,k}, \mathbf{X}^1_{i,j,k})\}_{j \in [\nu t] \setminus \mathcal{E}'_k}\}_{i \in [n]}$ to $\mathcal{A}$.
- Acting as $\mathcal{F}_{\mathbf{ccot}}$, $\mathcal{S}$ obtains the tth query from $\mathcal{A}$ as (1) $\vec{\sigma}_t := \sigma_{t,1}, \ldots, \sigma_{t,n}$, and (2) the set $[\nu t] \setminus \mathcal{E}'_t$.
- For $k \in [t]$, set $\mathcal{E}_k := \mathcal{E}'_k \setminus \cup_{k' \neq k} \mathcal{E}'_{k'}$. Define $\mathcal{C} = [\nu t] \setminus \cup_{k \in [t]} \mathcal{E}_k$.

  $\mathcal{S}$ sends $\vec{\sigma}_1, \ldots, \vec{\sigma}_t$ and sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ to $\mathcal{F}_{\mathbf{mcot}}$, and receives back $\{\chi^1_j\}_{j \in \mathcal{E}_1}, \ldots, \{\chi^t_j\}_{j \in \mathcal{E}_t}$ and $\{\{(X^0_{i,j}, X^1_{i,j})\}_{j \in \mathcal{C}}, \{X^{\sigma_{i,1}}_{i,j}\}_{j \in \mathcal{E}_1}, \ldots, \{X^{\sigma_{i,t}}_{i,j}\}_{j \in \mathcal{E}_t}\}_{i \in [n]}$.
- $\mathcal{S}$ chooses values $\{\tilde{\chi}^t_j\}_{j \in [\nu t]}$ as follows:
  - If $j \in \mathcal{E}_t$, then set $\tilde{\chi}^t_j := \chi^t_j \oplus \bigoplus_{k \in [t-1], i \in [n]} (\chi^{0,t}_{i,j,k} \oplus \chi^{1,t}_{i,j,k})$.
  - Else, choose $\tilde{\chi}^t_j$ uniformly at random.

- $\mathcal{S}$ chooses values $\{X^0_{i,j,t}, X^1_{i,j,t}\}_{i\in[n],j\in[\nu t]}$ as follows:
  - If $j \in \mathcal{C}$, then for $i \in [n]$ set $X^0_{i,j,t} := X^0_{i,j} \oplus \bigoplus_{k\in[t-1]} X^0_{i,j,k}$ and $X^1_{i,j,t} := X^1_{i,j} \oplus \bigoplus_{k\in[t-1]} X^1_{i,j,k}$.
  - Else (i.e., $j \in \mathcal{E}_k$ for some unique $k \in [t]$), for all $i \in [n]$ set $X^{\sigma_{i,k}}_{i,j,t} := X^{\sigma_{i,k}}_{i,j} \oplus \bigoplus_{k'\in[t-1]} X^{\sigma_{i,k}}_{i,j,k'}$, and set $X^{1-\sigma_{i,k}}_{i,j,t}$ to a random value.
- $\mathcal{S}$ chooses values $\{\chi^{0,k}_{i,j,t}, \chi^{1,k}_{i,j,t}\}_{i\in[n],j\in[\nu t],k\in[t]}$ as follows:
  - If $j \in \mathcal{E}_k$ for some (unique) $k \in [t]$, then for all $i \in [n]$ pick $\chi^{0,k}_{i,j,t}, \chi^{1,k}_{i,j,t}$ uniformly at random subject to $\bigoplus_{i\in[n]}(\chi^{0,k}_{i,j,t} \oplus \chi^{1,k}_{i,j,t}) = \tilde{\chi}^k_j \oplus \chi^k_j \oplus \bigoplus_{k'\in[t-1],i\in[n]}(\chi^{0,k}_{i,j,k'} \oplus \chi^{1,k}_{i,j,k'})$.
  - Else, for $i \in [n]$ and $k \in [t]$, pick $\chi^{0,k}_{i,j,t}, \chi^{1,k}_{i,j,t}$ uniformly at random.
- For $i \in [n]$ and $j \in [\nu t]$, let $\mathbf{X}^0_{i,j,t} := (X^0_{i,j,t}, \chi^{0,1}_{i,j,t}, \ldots, \chi^{0,t}_{i,j,t})$ and $\mathbf{X}^1_{i,j,t} := (X^1_{i,j,t}, \chi^{1,1}_{i,j,t}, \ldots, \chi^{1,t}_{i,j,t})$. Then, acting as $\mathcal{F}_{\mathbf{ccot}}$, $\mathcal{S}$ sends $\{\tilde{\chi}^t_j\}_{j\in\mathcal{E}'_t}$, $\{\{\mathbf{X}^{\sigma_{i,t}}_{i,j,t}\}_{j\in\mathcal{E}'_t} \cup \{(\mathbf{X}^0_{i,j,t}, \mathbf{X}^1_{i,j,t})\}_{j\in[\nu t]\setminus\mathcal{E}'_t}\}_{i\in[n]}$ to $\mathcal{A}$ and halts, outputting whatever $\mathcal{A}$ outputs.

First we show that if $\mathcal{A}$ inputs $\mathcal{E}'_1, \ldots, \mathcal{E}'_t$ such that these sets are pairwise non-intersecting, then its view in the above simulation is identically distributed to its view in the real execution. In this case, it is easy to see that for all $k \in [t]$ the extracted sets $\mathcal{E}_k$ in the simulation are identical to $\mathcal{E}'_k$ input by $\mathcal{A}$. Further, $\mathcal{C} = [\nu t] \setminus \cup_{k\in[t]}\mathcal{E}'_k$ also holds. Observe that for $j \neq j'$ the randomness used by an honest P_1 in the real execution to create values $\{\mathbf{X}^0_{i,j,k}, \mathbf{X}^1_{i,j,k}\}_{i,k}$ and the randomness used to create $\{\mathbf{X}^0_{i,j',k}, \mathbf{X}^1_{i,j',k}\}_{i,k}$ are independent of each other. Clearly, this is also the

case in the simulated execution. This allows us to split the analysis depending on the value of j.

- For $j \in \mathcal{E}_k$, the values $\{X_{i,j,k'}^{\sigma_{i,k}}\}_{k' \in [t]}$ are identically distributed in both executions (i.e., uniformly random and independent subject to $\bigoplus_{k' \in [t]} X_{i,j,k'}^{\sigma_{i,k}} = X_{i,j}^{\sigma_{i,k}}$). Furthermore, the view of $\mathcal{A}$ is independent of the values $X_{i,j}^{1-\sigma_{i,k}}$ since these are information-theoretically hidden from the real execution (as is the case in the ideal execution). This is because in the kth query to $\mathcal{F}_{\mathbf{ccot}}$, $\mathcal{A}$ does not receive one of the additive shares of $X_{i,j}^{1-\sigma_{i,k}}$, namely, the share $X_{i,j,k}^{1-\sigma_{i,k}}$.

  Next, it is easy to verify that the check values χ_j^k and their additive shares $\tilde{\chi}_j^k$, $\{\chi_{i,j,k'}^{0,k}, \chi_{i,j,k'}^{1,k}\}_{k' \in [t] \setminus \{k\}, i \in [n]}$ are also identically distributed in both executions.

  Finally, we claim that the view of $\mathcal{A}$ in the real execution is independent of the values $\{\chi_j^{k'}\}_{k' \neq k}$. This is because in the kth query to $\mathcal{F}_{\mathbf{ccot}}$, $\mathcal{A}$ did not receive, for every $k' \neq k$, at least one of the additive shares of $\chi_j^{k'}$, namely, the share $\chi_{1,j,k}^{0,k'}$.

- For $j \in \mathcal{C}$, the values $\{X_{i,j,k'}^0, X_{i,j,k'}^1\}_{i \in [n], k' \in [t]}$ are identically distributed in both executions (i.e., uniformly random and independent subject to $\bigoplus_{k' \in [t]} X_{i,j,k'}^0 = X_{i,j}^0$ and $\bigoplus_{k' \in [t]} X_{i,j,k'}^1 = X_{i,j}^1$). Furthermore, we claim that the view of $\mathcal{A}$ in the real execution is independent of the values $\{\chi_j^k\}_{k \in [t]}$. This is because in the kth query to $\mathcal{F}_{\mathbf{ccot}}$, $\mathcal{A}$ does not receive, for every $k \in [t]$, exactly one of the additive shares of χ_j^k, namely, share $\tilde{\chi}_j^k$.

Given the above, it follows that the view of $\mathcal{A}$ in the simulated execution is identically distributed to its view in the real execution.

Now we need to consider the case where $\mathcal{A}$ inputs sets $\mathcal{E}'_1, \ldots, \mathcal{E}'_t$ that are *not* pairwise non-intersecting. We define sets $\mathcal{E}_k = \mathcal{E}'_k \setminus \cup_{k' \neq k} \mathcal{E}'_{k'}$ for each $k \in [t]$. Also, define $\mathcal{E}_0 = [\nu t] \setminus \cup_{k \in [t]} \mathcal{E}'_k$, and $\mathcal{C} = [\nu t] \setminus \cup_{k \in [t]} \mathcal{E}_k$. As in the case where $\mathcal{E}'_1, \ldots, \mathcal{E}'_k$ were pairwise non-intersecting, we split the analysis depending on the value of j. It is easy to verify that the analysis in the case where $j \in \mathcal{E}_k$ is identical to its counterpart in the case where $\mathcal{E}'_1, \ldots, \mathcal{E}'_k$ were pairwise non-intersecting. Likewise the analysis in the cases where $j \in \mathcal{E}_0$ is identical to the analysis in the $j \in \mathcal{C}$ cases where $\mathcal{E}'_1, \ldots, \mathcal{E}'_k$ were pairwise non-intersecting.

Thus, we only need to analyze the case where $j \in \mathcal{E}_0 \setminus \mathcal{C}$. Such a j would exist only when there exist distinct $k, k' \in [t]$ such that $j \in \mathcal{E}'_k$ and $j \in \mathcal{E}'_{k'}$. In this case, note that by construction, the simulated values for $\{X^0_{i,j,k''}, X^1_{i,j,k''}\}_{i \in [n], k'' \in [t]}$ are consistent with the actual input values $\{X^0_{i,j}, X^1_{i,j}\}_{i \in [n]}$, and thus the shares obtained by $\mathcal{A}$ corresponding to the $X^0_{i,j}, X^1_{i,j}$ values are identically distributed.

It remains to show that as in the simulated execution, the view of $\mathcal{A}$ in the real execution is independent of the values $\{\chi_j^{k''}\}_{k'' \in [t]}$. Indeed, we claim that when $j \in \mathcal{E}'_k$, the value χ_j^k is independent of its view if there exists $k' \neq k$ such that $j \in \mathcal{E}'_{k'}$. This is because for $j \in \mathcal{E}'_k$, the value χ_j^k can be reconstructed only if all its additive shares $\tilde{\chi}_j^k$, $\{\chi^{0,k}_{i,j,k''}, \chi^{1,k}_{i,j,k''}\}_{k'' \in [t] \setminus \{k\}, i \in [n]}$ are obtained. However, if $j \in \mathcal{E}'_{k'}$, then in the k'th query to $\mathcal{F}_{\mathbf{ccot}}$, $\mathcal{A}$ loses its chance to receive at least one of the additive shares of χ_j^k, namely, the share $\chi^{0,k}_{1,j,k'}$. Thus, the claim holds. □

Cost of realizing $\mathcal{F}_{\mathbf{mcot}}$. As described, the cost of realizing $\mathcal{F}_{\mathbf{mcot}}$ is t times the cost of realizing $\mathcal{F}_{\mathbf{ccot}}$ for n vectors of pairs of length νt with each element of size

$(t+1)\kappa$. Thus if we use Lindell's $\mathcal{F}_{\mathbf{ccot}}$ construction [10] in order to implement $\mathcal{F}_{\mathbf{mcot}}$, then for each of the t executions we need to use $9n\nu t$ fixed-base exponentiations and $1.5n\nu t$ regular exponentiations, and need to send a total of $5n\nu t$ group elements.

Alternative approaches to realizing $\mathcal{F}_{\mathbf{mcot}}$. As discussed before, $\mathcal{F}_{\mathbf{mcot}}$ can be realized using general secure computation, but this results in extremely poor efficiency. In particular, the circuit computing $\mathcal{F}_{\mathbf{mcot}}$ is of size at least $\kappa\rho nt$, and realization by state-of-the-art secure protocols would further include a multiplicative $\kappa\rho$ overhead. We leave a more efficient realization of $\mathcal{F}_{\mathbf{mcot}}$ from either $\mathcal{F}_{\mathbf{ccot}}$ or directly from some standard assumption as an open question.

In settings where the $\nu t^2/\rho$ multiplicative overhead of realizing $\mathcal{F}_{\mathbf{mcot}}$ through our protocol is expensive relative to the size of the circuit, one may wonder whether it is possible to use an XOR-tree approach [8] to obtain better efficiency. Unfortunately, we do not know if this approach can be made to work with standard garbled circuits in the parallel setting. Specifically, it is no longer clear how P_1, without any knowledge of the evaluation sets, can batch P_2's input labels together in a way that lets P_2 learn different sets of input labels corresponding to different evaluation circuits and yet within each evaluation bucket guarantee that P_2 can learn only input labels corresponding to the same set of inputs.

However, if we assume that the garbling scheme is *adaptively secure* (cf. Section 3.2), then this lets us perform the oblivious transfer step after P_1 commits to its garbled circuits. Now P_2 can reveal its evaluation buckets one-by-one, thereby letting P_1 successfully batch P_2's input labels in the right manner. (See our protocol

for sequential executions in Section 3.4 for a full description of how to do this.)

Full Protocol

We use the $\mathcal{F}_{\mathbf{mcot}}$ construction as follows. The input vectors $\{\vec{X}_i\}_{i\in[\ell]}$ contain the labels associated with the ith input wire for P_2 in each of the νt circuits. The vector $\vec{\sigma}_k$ corresponds to the inputs used by P_2 in the kth execution. An honest P_2 chooses sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ such that they are disjoint and each set is of size exactly $\nu/2$. These sets correspond to "evaluation buckets," namely the set of circuits that will be used for a given iteration.

The main observation is that in the kth execution P_2 obtains check values χ_j^k from $\mathcal{F}_{\mathbf{mcot}}$ only for $j \in \mathcal{E}_k$. Therefore, once the parties complete the interaction with $\mathcal{F}_{\mathbf{mcot}}$ and P_1 sends all the garbled circuits, we let P_1 determine the evaluation circuits based on whether P_2 sends the corresponding check values.

Applying the cheating-punishment technique. Inspired by Lindell's protocol [10], we use the knowledge of two different garbled values for a single output wire as a "proof" that P_2 received inconsistent outputs in a given execution. P_2 can use this proof to obtain P_1's input in a cheating-punishment phase. This cheating-punishment phase is implemented via a secure computation protocol, and thus it is important that the second phase functionality has a small circuit. We employ several optimizations proposed by Lindell [10] to keep the size of this circuit small.

One important difference in our setting is that, unlike in Lindell's protocol [10], we cannot have, for a given output wire w, the same output-wire labels Z_w^0, Z_w^1 across all garbled circuits. This is because in our setting garbled circuits are assigned

to different evaluation buckets, and the circuits in each bucket can be evaluated with different input values, and thus can produce different outputs. Thus even in an honest execution P_2 could potentially learn, say, output-wire label Z_w^0 in one execution and output-wire label Z_w^1 in another.

We address this by simply removing the requirement that the set of output-wire labels across different garbled circuits are the same. Thus, the circuit for the cheating-punishment phase for the kth execution must now take as input from P_1 *all* of the output-wire labels in *all* of the evaluation circuits in the kth bucket, and from P_2 a pair of output-wire labels that serve as proof of cheating. Somewhat surprisingly, we show that the size of the circuit (measured as the number of non-XOR gates) for the cheating-punishment phase is essentially the same as the circuit in Lindell's protocol [10].

Other details. We now describe other important details of our protocol.

- *Input consistency across multiple executions.* It is important to guarantee that P_1 provides consistent inputs across all circuits in the kth execution. Fortunately, existing mechanisms [10, 9] for ensuring input consistency in the single execution setting can be readily extended to the multiple execution setting as well.

- *Encoded translation tables for garbled circuits.* As in Lindell's protocol [10], we modify the output translation tables used in the garbled circuits. Specifically, for labels Z_w^0, Z_w^1 on output wire w, we create an *encoded* output table $[H(Z_w^0), H(Z_w^1)]$, where H is some collision-resistant hash function. We re-

quire that the output labels (or more precisely, the output of H applied to the output labels) corresponding to 0 and 1 are distinct. This encoding gives us the following two properties: (1) P_2 after evaluating a garbled circuit can use the encoded translation tables to determine whether the output is 0 or 1, and (2) the encoded translation table does not reveal the other output label (since this is equivalent to inverting the hash function) to P_2.

- *Optimizing the cheating-punishment circuit.* We can apply similar techniques as shown by Lindell [10] to optimize the size of the cheating-punishment circuit to contain only n non-XOR gates; see below.

Formal description. We proceed to the formal description of our protocol.

Inputs: P_1 has input $(x_1, \ldots, x_t)$, where $x_k \in \{0,1\}^n$, and P_2 has input $(y_1, \ldots, y_t)$, where $y_k \in \{0,1\}^n$.

Auxiliary Inputs: Statistical security parameter ρ, computational security parameter κ, the description of a circuit C where $C(x,y) = f(x,y)$ for some $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{n'}$, the number of evaluations t of the function f, and $(\mathbb{G}, q, g)$ where $\mathbb{G}$ is a cyclic group with generator g and prime order q, where q is of length κ. Let $\mathsf{Ext} : \mathbb{G} \to \{0,1\}^\kappa$ be a function mapping group elements to bitstrings and let $H : \{0,1\}^\kappa \to \{0,1\}^\kappa$ be a preimage-resistant hash function. Let ν, which we call the replication factor, be defined as being the smallest $u \in \mathbb{N}$ such that for all $m \in \{u/2, \ldots, ut/2\}$ it holds that $t \cdot \binom{ut-m}{ut/2}\binom{m}{u/2}/\binom{ut}{ut/2}\binom{ut/2}{u/2} \leq 2^{-\rho}$. Finally, we also assume access to ideal functionalities $\mathcal{F}_{\mathbf{mcot}}$ and $\mathcal{F}_{\mathbf{zk}}$.

Protocol:

1. **Input/output labels and circuit preparation:**
   - P_1 chooses random values $a_1^0, a_1^1, \ldots, a_n^0, a_n^1 \in_R \mathbb{Z}_q$, $r_1, \ldots, r_{\nu t} \in_R \mathbb{Z}_q$ and $(Z_{1,1}^0, Z_{1,1}^1, \ldots, Z_{n',1}^0, Z_{n',1}^1), \ldots, (Z_{1,\nu t}^0, Z_{1,\nu t}^1, \ldots, Z_{n',\nu t}^0, Z_{n',\nu t}^1) \in_R \{0,1\}^\kappa$.
   - Let $X_{i,j}^b$ denote the label associated with bit b for P_1's ith input bit in the jth garbled circuit. P_1 sets $X_{i,j}^b$ as follows:

$$X_{i,j}^0 := \mathsf{Ext}(g^{a_i^0 \cdot r_j}) \quad \text{and} \quad X_{i,j}^1 := \mathsf{Ext}(g^{a_i^1 \cdot r_j}).$$

- Let $Z^b_{i,j}$ denote the label associated with bit b on the ith output wire in the jth garbled circuit.
- P_1 constructs νt garblings, $\widehat{C}_1, \ldots, \widehat{C}_{\nu t}$, of circuit C, using random labels except for its own input-wire labels and the output-wire labels, where the labels are set as above.

2. **Oblivious transfer:** P_1 and P_2 run $\mathcal{F}_{\mathbf{mcot}}$ as follows:
   - For $i \in [n]$, let $\vec{Y}_i$ denote a vector containing the νt pairs of labels associated with P_2's ith input bit in all the garbled circuits. P_1 inputs $\vec{Y}_1, \ldots, \vec{Y}_n$, as well as random values $\chi^1_1, \ldots, \chi^1_{\nu t}, \ldots, \chi^t_1, \ldots, \chi^t_{\nu t}$.
   - P_2 inputs random sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ which are pairwise non-intersecting subsets of $[\nu t]$ such that for all $k \in [t]$ it holds that $|\mathcal{E}_k| = \nu/2$. Let $\mathcal{C} := [\nu t] \setminus \cup_{k \in [t]} \mathcal{E}_k$. P_2 also inputs bits $(\sigma_{1,1}, \ldots, \sigma_{n,1}), \ldots, (\sigma_{1,t}, \ldots, \sigma_{n,t}) \in \{0,1\}^n$, where $\sigma_{i,k} := y_k[i]$ for $i \in [n]$ and $k \in [t]$.
   - For $j \in \mathcal{C}$, P_2 receives both input keys associated with its input wires in garbled circuit $\widehat{C}_j$, and for $k \in [t]$ and $j \in \mathcal{E}_k$, P_2 receives the keys associated with its input y_k on its input wires in garbled circuit $\widehat{C}_j$. Also, for $k \in [t]$ and $j \in \mathcal{E}_k$, P_2 receives χ^k_j.

3. **Send circuits and commitments:** P_1 sends P_2 the garbled circuits $\widehat{C}_1, \ldots, \widehat{C}_{\nu t}$, the following commitment to the labels associated with P_1's input wires:
   $$\{(i, 0, g^{a^0_i}), (i, 1, g^{a^1_i})\}_{i \in [n]} \quad \text{and} \quad \{(j, g^{r_j})\}_{j \in [\nu t]}$$
   and the encoded output translation tables:
   $$\{[(H(Z^0_{1,j}), H(Z^1_{1,j})), \ldots, (H(Z^0_{n',j}), H(Z^1_{n',j}))]\}_{j \in [\nu t]}.$$
   If $H(Z^0_{i,j}) = H(Z^1_{i,j})$ for any $i \in [n'], j \in [\nu t]$, then P_2 aborts.

4. **Cut-and-choose challenge:** P_2 sends P_1 the sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ along with values $\{\chi^1_j\}_{j \in \mathcal{E}_1}, \ldots, \{\chi^t_j\}_{j \in \mathcal{E}_t}$.
   If either (1) the check values are not valid (2) the sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ are not disjoint, or (4) there exists some $k \in [t]$ such that $|\mathcal{E}_k| \neq \nu/2$, then P_1 outputs $\perp$ and aborts. Garbled circuits $\widehat{C}_j$ for $j \in \mathcal{C}$ are called *check circuits* and garbled circuits $\widehat{C}_j$ for $j \in \mathcal{E}_k$ are called *evaluation circuits in the kth bucket.*

5. **Send labels:** For $k \in [t]$, P_1 sends the labels associated with input x_k for the evaluation circuits in the kth bucket. That is, for $j \in \mathcal{E}_k$ and $i \in [n]$, P_1 sends the value $X'_{i,j} := g^{a_i^{x_k[i]} \cdot r_j}$ and P_2 sets $X_{i,j} := \mathsf{Ext}(X'_{i,j})$.

6. **Circuit evaluation:** For $k \in [t]$, P_2 does the following:
   For $j \in \mathcal{E}_k$, $i \in [n']$, P_2 learns $Z'_{i,j}$ by evaluating $\widehat{C}_j$. We call an output-wire label $Z'_{i,j}$ *valid* if it exists in the encoded output translation table sent in Step 3 (note that if $Z'_{i,j}$ is valid then P_2 can map it to its associated bit using the translation table). If P_2 receives exactly *one* bit per output wire, then let z_k denote the

output. In this case, P_2 chooses random values $Z_k^0, Z_k^1 \in_R \{0,1\}^\kappa$. If P_2 receives *two* valid outputs on any output wire then it sets $Z_k^0 := Z'_{i,j_1}$ and $Z_k^1 := Z'_{i,j_2}$, where $j_1, j_2 \in \mathcal{E}_k$ denote the conflicting circuit indices. If P_2 receives *no* valid output values on any output wire, then P_2 aborts.

7. **Cheating detection:** For $k \in [t]$, P_1 and P_2 do the following:

   P_1 defines a circuit C_{sc} with the values $\{Z^0_{1,j}, Z^1_{1,j}, \ldots, Z^0_{n',j}, Z^1_{n',j}\}_{j \in \mathcal{E}_k}$ hard-coded. The circuit computes the following function:

   - P_1 inputs $x_k \in \{0,1\}^n$ and has no output.
   - P_2 inputs a pair of values Z_k^0, Z_k^1.
   - If there exist values $i \in [n']$ and $j_1, j_2 \in \mathcal{E}_k$ such that $Z_k^0 = Z^0_{i,j_1}$ and $Z_k^1 = Z^1_{i,j_2}$, then P_2's output is x_k; otherwise it receives no output.

   P_1 and P_2 run the protocol of Lindell and Pinkas [9] on C_{sc} as follows:

   - P_1 inputs x_k; P_2 inputs Z_k^0 and Z_k^1 as computed in Step 6.
   - The garbled circuits constructed by P_1 use the same a_i^0, a_i^1 values as were chosen in Step 1, and the parties use 3ρ copies of the circuit for the cut-and-choose.

   If this computation results in an abort, then both parties halt.

8. **Check circuits for computing $\mathcal{F}_{\mathbf{par}}$:**

   - For $j \in \mathcal{C}$, P_1 sends r_j to P_2, and P_2 checks that these values are consistent with the pairs $\{(j, g^{r_j})\}_{j \in \mathcal{C}}$ received in Step 3. If not, P_2 aborts.
   - For $j \in \mathcal{C}$, P_2 uses the $g^{a_i^0}, g^{a_i^1}$ values received in Step 3 and the r_j values received above to compute the keys for P_1's input wires as $X^0_{i,j} := \mathsf{Ext}(g^{a_i^0 \cdot r_j}), X^1_{i,j} := \mathsf{Ext}(g^{a_i^1 \cdot r_j})$. In addition, P_2 uses the keys obtained from $\mathcal{F}_{\mathbf{mcot}}$ in Step 2 for its own input wires. P_2 verifies that $\widehat{C}_j$ is a correct garbling of C. If there exists a circuit for which this does not hold, then P_2 aborts.

9. **Verify consistency of P_1's input:** For $k \in [t]$, let $\widehat{\mathcal{E}}_k$ be the set of evaluation circuits used in the 2PC computation in Step 7, let $\widehat{r}_{j,k}$ be the analogous value of r_j used in that computation, and let $\widehat{X}_{i,j}$ be the analogous value of $X'_{i,j}$ used in that computation. For $k \in [t]$, P_1 and P_2 do the following:

   - For $i \in [n]$, P_1 uses $\mathcal{F}_{\mathbf{zk}}$ to proves that there exists some $\sigma_{i,k}$ such that for $j \in \mathcal{E}_k$ and $j' \in \widehat{\mathcal{E}}_k$, it holds that $X'_{i,j} = g^{a_i^{\sigma_{i,k}} \cdot r_j}$ and $\widehat{X}_{i,j} = g^{a_i^{\sigma_{i,k}} \cdot \widehat{r}_{j',k}}$.

   If any of the t proofs fail, then P_2 aborts.

10. **Output evaluation:** For $k \in [t]$, P_2 does the following:

    If P_2 received no inconsistent outputs in Step 6, then it uses the encoded translation tables to decode the outputs it received, and sets z_k to that value. If P_2 received inconsistent output, then let x_k be the output that P_2 received from the circuit in Step 7. Let $z_k := f(x_k, y_k)$ be the output in this case.

    P_2 outputs $(z_1, \ldots, z_t)$.

Theorem 3.2. *Let ρ (resp., κ) be the statistical (resp., computational) security parameter. If the decisional Diffie-Hellman assumption holds in $(\mathbb{G}, g, q)$, H is a collision-resistant function, and the underlying circuit garbling procedure is private, then for all $t \in \mathsf{poly}(\kappa)$, the protocol described above securely computes $\mathcal{F}_{\mathbf{par}}$ in the presence of a malicious adversary with error at most $2^{-\rho} + \mathsf{negl}(\kappa)$ in the $(\mathcal{F}_{\mathbf{mcot}}, \mathcal{F}_{\mathbf{zk}})$-hybrid model.*

Proof. We prove security in a hybrid model where we have access to $\mathcal{F}_{\mathbf{mcot}}$ and the zero-knowledge proof-of-knowledge functionality $\mathcal{F}_{\mathbf{zk}}$ in Step 9. We split the analysis into two cases depending on whether P_1 or P_2 is corrupted.

P_1 is corrupted. The intuition is that P_1 can cheat only if it can construct incorrect circuits. To do this, P_1 needs to construct a *small* enough number of incorrect circuits such that it will not get caught in the first cut-and-choose stage; however, it need also construct a *large* enough number such that one of the buckets contains all incorrect circuits. This is due to the fact that P_2 aborts if it finds an invalid check circuit, and learns P_1's input (and thus the correct output) if a given bucket contains at least one correctly constructed circuit. This implies that the number of corrupt circuits m constructed by a malicious P_1 must be such that $\nu/2 \leq m \leq \nu t/2$. We

stress that m is fixed once P_1 sends the circuits in Step 3; that is, P_1 cannot further "corrupt" circuits after this step. Now observe that the probability with which m bad circuits escape detection in the first stage cut-and-choose is $\binom{\nu t-m}{\nu t/2}/\binom{\nu t}{\nu t/2}$. Conditioned on this event happening, the probability that a particular bucket contains all bad circuits is $\binom{m}{\rho/2}/\binom{\rho t/2}{\rho/2}$. Applying the union bound, we conclude that the probability that P_1 succeeds in cheating is bounded by

$$t\binom{\rho t-m}{\rho t/2}\binom{m}{\rho/2}\Big/\binom{\rho t}{\rho t/2}\binom{\rho t/2}{\rho/2}.$$

Since it is given that the maximum value of this expression is less than $2^{-\rho}$ for parameter ν chosen in the protocol, we have that the probability of cheating is at most $2^{-\rho}$. We now proceed to the formal proof.

Let $\mathcal{A}$ be an adversary controlling P_1 with input $(x_1, \ldots, x_t)$. Since $\mathcal{A}$ receives no output, we need only show that the difference in probability that P_2 aborts in the real world versus the ideal world is negligible. We construct a simulator $\mathcal{S}$ with access to functionality $\mathcal{F}_{\mathbf{par}}$ as follows:

1. $\mathcal{S}$ acts as an honest P_2 would for the entire protocol execution, using input $(0^n, \ldots, 0^n)$ throughout.

2. For $k \in [t]$, let $x_k = \sigma_{1,k}, \ldots, \sigma_{n,k}$ be P_1's witness to the zero-knowledge proof-of-knowledge in Step 9. $\mathcal{S}$ extracts these values through the ideal functionality interface.

3. If P_2 would abort at any point in the protocol, then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathbf{par}}$ and

halts, outputting whatever $\mathcal{A}$ outputs. Otherwise, it sends $(x_1, \ldots, x_t)$ to $\mathcal{F}_{\mathbf{par}}$.

4. $\mathcal{S}$ halts and outputs whatever $\mathcal{A}$ outputs.

We now claim that the distributions from $\mathcal{A}$ interacting with P_2 in the hybrid world versus $\mathcal{A}$ interacting with $\mathcal{S}$ in the ideal world are indistinguishable. We show this by a series of hybrids.

$\mathbf{H_1}$. The hybrid-world execution.

$\mathbf{H_2}$. We extract $\mathcal{A}$'s input $(x_1, \ldots, x_t)$ from its query to $\mathcal{F}_{\mathbf{zk}}$ in Step 9. Instead of outputting P_2's output from the execution of the protocol, we instead pass $(x_1, \ldots, x_t)$ to $\mathcal{F}_{\mathbf{par}}$ and output whatever $\mathcal{F}_{\mathbf{par}}$ outputs.

These two hybrids differ if the output of P_2 differs from the output computed by $\mathcal{F}_{\mathbf{par}}$. Note that whether a garbled circuit is "correct" or not is fixed after Step 2. That is, P_1 cannot change the correctness of a garbled circuit after Step 3. We now argue that the only case in which these two hybrids differ is if one of the evaluation buckets contains all maliciously constructed circuits.

Suppose a bucket, say the kth, has at least one correct garbled circuit. In this case, P_2 evaluated this garbled circuit to $f(x_k, y_k)$ as intended. If there exists another *incorrect* garbled circuit within this bucket producing a different output, then P_2 receives two different output-wire labels and can use the cheating recovery to learn x_k and thus compute $f(x_k, y_k)$ itself. (Note that by the security of the second stage 2PC protocol in Step 7, P_2 either learns x_k or aborts.) Alternatively, the incorrect garbled circuit can produce garbage

output-wire labels, in which case P_2 ignores this circuit. Thus, for P_2 to *not* learn $f(x_k, y_k)$, it must be the case that all garbled circuits in a bucket are incorrect. As was shown above, this happens with probability $\leq 2^{-\rho}$, and thus we conclude that these hybrids are statistically indistinguishable.

$\mathbf{H_3}$. P_2 uses input $(0^n, \ldots, 0^n)$ throughout.

As P_2 only uses its input as input to the $\mathcal{F}_{\mathbf{mcot}}$ functionality in Step 2, we conclude that these two hybrids are perfectly indistinguishable.

As $\mathbf{H_3}$ is the same as the simulator $\mathcal{S}$ given above, we conclude that the protocol is statistically indistinguishable.

P_2 is corrupted. The intuition for security in the case that P_2 is corrupt is standard: for the evaluation circuits, P_2 learns nothing, and in each bucket, P_2 learns the correct output. We utilize a simulator for the garbled circuit generation to "fix" the output of the evaluation circuits to be the expected output for the given bucket.

Let $\mathcal{A}$ be an adversary controlling P_2 with input $(y_1, \ldots, y_t)$. We assume the existence of a simulator $\mathcal{S}_{gc}$ which constructs garbled circuits with fixed outputs which are indistinguishable from correctly garbled circuits. Such a simulator is known to exist [28, 8]. Also, we use the simulator for the maliciously secure 2PC protocol of Lindell and Pinkas [9], which we denote as $\mathcal{S}_{LP11}$.

We construct a simulator $\mathcal{S}$ with access to functionality $\mathcal{F}_{\mathbf{par}}$. $\mathcal{S}$ runs the protocol as an honest P_1 would, except as follows:

1. $\mathcal{S}$ extracts $\mathcal{A}$'s input $(y_1, \ldots, y_t)$ and evaluation sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ from its call to $\mathcal{F}_{\mathbf{mcot}}$. $\mathcal{S}$ sets $\mathcal{C} := [\nu t] \setminus \cup_k \mathcal{E}_k$.

2. $\mathcal{S}$ sends $(y_1, \ldots, y_t)$ to $\mathcal{F}_{\mathbf{par}}$, receiving back either $(z_1, \ldots, z_t)$ or $\perp$.

3. For every $j \in \mathcal{C}$, $\mathcal{S}$ constructs a valid garbled circuit. For $k \in [t]$ and $j \in \mathcal{E}_k$, $\mathcal{S}$ uses $\mathcal{S}_{gc}$ to construct a garbled circuit that outputs the fixed string z_k irrespective of the input.

4. $\mathcal{S}$ uses $\mathcal{S}_{LP11}$ to simulate the 2PC protocol in Step 7.

5. Upon protocol termination, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

We now claim that the distributions from $\mathcal{A}$ interacting with P_1 in the hybrid world versus $\mathcal{A}$ interacting with $\mathcal{S}$ in the ideal world are indistinguishable. To do so, we again construct a series of hybrids.

$\mathbf{H_1}$. The hybrid-world execution.

$\mathbf{H_2}$. We extract $\mathcal{A}$'s input $(y_1, \ldots, y_t)$ and the sets $\mathcal{E}_1, \ldots, \mathcal{E}_t$ from the call to $\mathcal{F}_{\mathbf{mcot}}$ in Step 2. Let $(z_1, \ldots, z_t)$ be the output of $\mathcal{F}_{\mathbf{par}}$, and let $\mathcal{C} = [\nu t] \setminus \cup_k \mathcal{E}_k$. For $j \in \mathcal{C}$, we construct correctly garbled circuits, and for $j \in \mathcal{E}_k$ for all k, we use $\mathcal{S}_{gc}$ to construct a circuit which always outputs z_k.

We claim that these two hybrids are computationally indistinguishable. Note that $\mathcal{A}$ can distinguish if either it can open one of the simulated garbled circuits, or it can evaluate a simulated garbled circuit in bucket k on something other than y_k. The only way for one of the above situations to occur is if (1) $\mathcal{A}$ can guess either the check values or the input-wire labels for the garbled circuits in Step 4, or (2) $\mathcal{A}$ can distinguish the use of $\mathcal{S}_{gc}$. In the former case,

as these values are random, this happens with probability $\leq 2^{-\kappa}$, and in the latter case, by the security of $\mathcal{S}_{gc}$ this happens with negligible probability.

$\mathbf{H_3}$. We replace the real 2PC execution in Step 7 with a simulated execution using $\mathcal{S}_{LP11}$.

Due to the security of $\mathcal{S}_{LP11}$ [9], we conclude that these two hybrids are indistinguishable.

$\mathbf{H_4}$. We use $(0^n, \ldots, 0^n)$ as P_1's input throughout.

Note that this affects Step 5, where $\mathcal{A}$ receives P_1's inputs $g^{a_i^{x_k[i]} \cdot r_j}$; however, by the decisional Diffie-Hellman assumption, $\mathcal{A}$ cannot extract $a_i^{x_k[i]}$ from this expression, and thus cannot deduce that P_1's input is as defined above. Thus, the two hybrids are computationally indistinguishable.

As $\mathbf{H_4}$ is the same as the simulator $\mathcal{S}$ given above, we conclude that the protocol is computationally indistinguishable. □

Optimizing the Circuit in Step 7

We use an optimization inspired by Lindell [10] to construct an alternate circuit that minimizes the number of non-XOR gates. Specifically, Lindell [10] shows how to efficiently construct a garbled circuit that checks if a given κ-bit string is contained in a set S of size $|S|$. The garbled circuit has the property that it only requires n non-XOR gates, and thus can be essentially computed for free using the free-XOR technique [29] (cf. Chapter 2). This optimization relies on the fact that to take a

κ-wise AND of two κ-bit strings, it suffices to encrypt the output 1-label with the 1-labels on the input wires. Therefore, to compare two κ-bit strings, we first XOR the two strings bit-by-bit, take the NOT of these bits, and finally output the κ-wise AND of the resulting bits using the trick described above. Next, to check that a κ-bit string equals any of the strings in S, we need to evaluate the $|S|$-wise OR of each of these comparisons. Instead of using $|S|-1$ OR gates, we can set the 1-label on all of the output wires from the κ-wise ANDs above to be the 1-label on the output wire of the OR. Since XOR and NOT gates can be evaluated for free [29], it follows that the above circuit can essentially be securely evaluated for free.

We now adapt these optimizations to our setting, while still minimizing the number of non-XOR gates. For string b and set S, we use the notation $b \in_? S$ to denote a boolean expression that evaluates to 1 if and only if $b \in S$. In our protocol we require a circuit that takes, in addition to an n-bit string x (representing P_1's actual input), a pair of κ-bit strings, say b_0, b_1, and two sets S_0, S_1 of κ-bit strings, each set of size $\nu n'/2$, and outputs x if and only if $((b_0 \in_? S_0) \wedge (b_1 \in_? S_1)) \vee ((b_0 \in_? S_1) \wedge (b_1 \in_? S_0)) = 1$, an additional cost of 3 non-XOR gates. Alternatively, we may instead evaluate the expression $b_0 \oplus b_1 \in_? S$, where $S = \{b \oplus b' \ : \ b \in S_0, b' \in S_1\}$. (Note that a cheating P_2 can guess a value in S only with negligible probability.) This has the additional advantage of reducing P_2's input length from 2κ to κ (and the resulting gains from performing a lesser number of cut-and-choose oblivious transfers). In summary, it is possible to design the circuit in Step 7 using *exactly* n non-XOR gates (i.e., n AND gates to select P_1's input depending on whether the relevant conditions are satisfied). It follows from the protocol description that the

total number of garbled gates sent in Step 7 is $3n\rho$ in each of the t executions.

3.4 The Sequential Execution Setting

We now consider the setting where the parties securely evaluate the same function f multiple times *sequentially*. Again, we let t denote the number of times the parties wish to evaluate f and let P_1's and P_2's input in the kth execution be denoted by x_k and y_k, respectively. We let $\mathcal{F}_{\mathbf{seq}}$ denote the functionality that computes f a total of t times sequentially.

The main difference between this setting and the parallel setting discussed in Section 3.3 is that in the sequential setting the parties may not know their inputs to all executions at the start of the protocol. In particular, inputs may depend on outputs from previous executions. Thus, the parallel execution protocol does not immediately carry over to the sequential setting. To see why, observe for instance that $\mathcal{F}_{\mathbf{mcot}}$ requires P_2 to submit *all* its inputs at once. This is not possible since in the sequential setting we cannot assume that P_2 has all its inputs at the beginning of the protocol.

Instead, we take a different approach. Namely, we use the "XOR-tree" [8] to protect against the so-called "selective failure attack" [39, 40, 32]. (In the parallel execution setting, this attack was implicitly avoided due to the use of $\mathcal{F}_{\mathbf{mcot}}$.) In this approach, the circuit C to be evaluated is first modified into an equivalent circuit C_{XT}, where each of P_2's input bits is now secret-shared into ρ shares, thus expanding P_2's input length from n to ρn (although this expansion factor can be

reduced using known techniques [8, 47]). Then, P_1 sends commitments to the input labels corresponding to P_2's input wires in C_{XT}. The corresponding decommitments are revealed to P_2 via a standard one-out-of-two oblivious transfer[4]. In order to prevent P_2 from using different inputs across evaluation circuits within the same bucket, P_1 batches together the decommitments corresponding to a particular input wire across all evaluation circuits in a given bucket.

Note that herein lies an opportunity for a malicious P_1 to force P_2 to abort the protocol depending on its input. (This can be done for instance by sending incorrect decommitments for say only the 0-label on a particular wire.) However, the modified circuit C_{XT} is such that the success of any such "selective failure attack" is statistically independent of P_2's actual input value. Therefore, if an honest P_2 receives an invalid decommitment and is unable to decrypt the evaluation circuit, then it simply aborts knowing that its privacy is not compromised.

We stress that the oblivious transfer step happens *after* P_1 sends all the garbled circuits to P_2. This is because P_2's inputs to all t executions are not available at the beginning of the protocol. Further, P_2's inputs may depend on previous outputs, which can be obtained only by decrypting evaluation circuits, i.e., after the evaluation bucket for the current execution is fully determined. Note that our cut-and-choose technique guarantees that there is at least one good evaluation circuit in every bucket under the assumption that P_1 has already committed to all its (good and bad) garbled circuits before the check sets and the evaluation sets are

[4]We note that since we use one-out-of-two oblivious transfer (as opposed to $\mathcal{F}_{\mathbf{mcot}}$), we can leverage oblivious transfer extension techniques [7, 25, 33] to obtain better efficiency.

determined.

Unfortunately, the above ordering of the oblivious transfer step and the garbled circuit sending step now allows a malicious P_2 to choose its input as a function of the garbled circuits it receives, which is not simulatable. To counter this, we need to use *adaptively secure garbling schemes* [37] (cf. Section 3.2) instead of standard garbled circuits; adaptively secure garbling schemes can be constructed efficiently in the programmable random oracle model [37]. Note that we do not need the use of adaptively secure garbling schemes for implementing the cheating-punishment phase. Indeed, all the inputs for that subprotocol are known before the phase begins, and therefore, the oblivious transfer step can be carried out before P_1 sends its garbled circuits for that phase.

Formal description. We now proceed to the formal description of the protocol.

Auxiliary Input: Statistical security parameter ρ, computational security parameter κ, the description of a circuit C where $C(x,y) = f(x,y)$ for some $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{n'}$, the number of evaluations t of the function f, and $(\mathbb{G}, q, g)$ where $\mathbb{G}$ is a cyclic group with generator g and prime order q, where q is of length κ. Let $\mathsf{Ext} : \mathbb{G} \to \{0,1\}^\kappa$ be a function mapping group elements to bitstrings and let $H : \{0,1\}^\kappa \to \{0,1\}^\kappa$ be a preimage-resistant hash function. Let ν, which we call the replication factor, be defined as being the smallest $u \in \mathbb{N}$ such that for all $m \in \{u/2, \ldots, ut/2\}$ it holds that $t \cdot \binom{ut-m}{ut/2}\binom{m}{u/2} / \binom{ut}{ut/2}\binom{ut/2}{u/2} \leq 2^{-\rho}$. Finally, we also assume access to ideal functionalities $\mathcal{F}_{\mathbf{ct}}$, $\mathcal{F}_{\mathbf{ot}}$, and $\mathcal{F}_{\mathbf{zk}}$.

Additional Notation: Let C_{XT} denote the circuit C enhanced with the XOR-tree, and let n, n_{XT}, and n' denote the length of P_1's input, P_2's (XOR-tree expanded) input, and the output, respectively, of $C_{\mathsf{XT}}(x,y)$.

Offline Phase:

1. **Input/output labels and circuit preparation:**
   - P_1 chooses random values $a_1^0, a_1^1, \ldots, a_n^0, a_n^1, r_1, \ldots, r_{\nu t} \in_R \mathbb{Z}_q$ and $(Z_{1,1}^0, Z_{1,1}^1, \ldots, Z_{n',1}^0, Z_{n',1}^1), \ldots, (Z_{1,\nu t}^0, Z_{1,\nu t}^1, \ldots, Z_{n',\nu t}^0, Z_{n',\nu t}^1) \in_R \{0,1\}^\kappa$.

- Let $X^b_{i,j}$ denote the label associated with bit b for P_1's ith input bit in the jth circuit. P_1 sets $X^b_{i,j}$ as follows:

$$X^0_{i,j} := \mathsf{Ext}(g^{a^0_i \cdot r_j}) \quad \text{and} \quad X^1_{i,j} := \mathsf{Ext}(g^{a^1_i \cdot r_j}).$$

- Let $Y^b_{i,j}$ denote the label associated with bit b for P_2's ith input bit in the jth circuit. P_1 picks the labels for P_2's input wires uniformly at random, and computes (standard) commitments

$$e^0_{i,j} \leftarrow \mathsf{com}(Y^0_{i,j}) \quad \text{and} \quad e^1_{i,j} \leftarrow \mathsf{com}(Y^1_{i,j}).$$

  Let $d^0_{i,j}$ and $d^1_{i,j}$ denote the corresponding decommitments.

- Let $Z^b_{i,j}$ denote the label associated with bit b on the ith output bit in the jth circuit.

- P_1 constructs νt independent *adaptively secure* garblings of circuit C_{XT}, denoted $\widehat{C}_1, \ldots, \widehat{C}_{\nu t}$, using random labels except the input wires, where the labels are set as above.

2. **Send circuits and commitments:** P_1 sends P_2 the garbled circuits, the commitments to the garbled values associated with P_1's input wires:

$$\{(i, 0, g^{a^0_i}), (i, 1, g^{a^1_i})\}_{i \in [n]} \quad \text{and} \quad \{(j, g^{r_j})\}_{j \in [\nu t]},$$

   the encoded output translation tables:

$$\{[(H(Z^0_{1,j}), H(Z^1_{1,j})), \ldots, (H(Z^0_{n',j}), H(Z^1_{n',j}))]\}_{j \in [\nu t]},$$

   and the commitments to the garbled values associated with P_2's input wires:

$$\{e^0_{i,j}, e^1_{i,j}\}_{i \in [n_{\mathrm{XT}}], j \in [\nu t]}.$$

   If $H(Z^0_{i,j}) = H(Z^1_{i,j})$ for any $i \in [n']$, $j \in [\nu t]$, then P_2 aborts.

3. **Cut-and-choose challenge:** P_1 and P_2 run $\mathcal{F}_{\mathbf{ct}}$ to compute a set $\mathcal{C} \subset [\nu t]$ such that $|\mathcal{C}| = \nu t/2$. Garbled circuits $\widehat{C}_j$ for $j \in \mathcal{C}$ are called *check circuits*.

4. **Check circuits for computing $\mathcal{F}_{\mathbf{seq}}$:**

   - **Send labels:** For every check circuit $\widehat{C}_j$, P_1 sends the value r_j to P_2, and P_2 checks that these are consistent with the pairs $\{(j, g^{r_j})\}_{j \in \mathcal{C}}$ received in Step 2. If not, P_2 aborts.

   - **Send decommitments:** For every check circuit $\widehat{C}_j$, P_1 sends the decommitments $\{d^0_{i,j}, d^1_{i,j}\}_{i \in [n_{\mathrm{XT}}]}$ for commitments $\{e^0_{i,j}, e^1_{i,j}\}_{i \in [n_{\mathrm{XT}}]}$, and P_2 checks that these are valid decommitments, and computes the corresponding labels $\{Y^0_{i,j}, Y^1_{i,j}\}_{i \in [n_{\mathrm{XT}}]}$. If not, P_2 aborts.

- **Check correctness:** For $j \in \mathcal{C}$, P_2 uses the $g^{a_i^0}, g^{a_i^1}$ values received in Step 2 and the r_j values received above to compute the labels $X_{i,j}^0 := \mathsf{Ext}(g^{a_i^0 \cdot r_j})$ and $X_{i,j}^1 := \mathsf{Ext}(g^{a_i^1 \cdot r_j})$ associated with P_1's input.

  Given labels for all input wires in $\widehat{C}_j$, P_2 verifies that the circuit is a garbling of C_{XT}, using the encoded translation tables for the output values. If there exists a circuit for which this does not hold, then P_2 aborts.

Online Phase: For $k \in [t]$, execute the following sequentially:

5. **Receive inputs:** P_1 and P_2 obtain inputs x_k and y_k, respectively. P_2 transforms its input y_k for circuit C into an equivalent "secret-shared" input $\widetilde{y}_k$ for circuit C_{XT}.

6. **Second-stage cut-and-choose:** P_2 picks $\mathcal{E}_k \subseteq [\nu t] \setminus \mathcal{C}$ of size $\nu/2$ such that $\mathcal{E}_1, \ldots, \mathcal{E}_k$ are disjoint. P_2 sends $\mathcal{E}_k$ to P_1, who aborts the protocol if $|\mathcal{E}_k| \neq \nu/2$ or $\mathcal{E}_k$ intersects with a previously sent subset. We call $\mathcal{E}_k$ the *kth evaluation bucket*.

7. **Oblivious transfer:** For $i \in [n_{\mathrm{XT}}]$, let $D_{i,k}^0 := \{d_{i,j}^0\}_{j \in \mathcal{E}_k}$ and $D_{i,k}^1 := \{d_{i,j}^1\}_{j \in \mathcal{E}_k}$. P_1 and P_2 engage in n_{XT} invocations of $\mathcal{F}_{\mathbf{ot}}$ where in the ith invocation:

   - Acting as the sender, P_1 inputs $(D_{i,k}^0, D_{i,k}^1)$.
   - Acting as the receiver, P_2 inputs $\widetilde{y}_k[i]$, and receives $D_{i,k}^{\widetilde{y}_k[i]}$.

   If there exists $j \in \mathcal{E}_k$ and $i \in [n_{\mathrm{XT}}]$ such that $d_{i,j}^{\widetilde{y}_k[i]}$ is not a valid decommitment to $e_{i,j}^{\widetilde{y}_k[i]}$, then P_2 aborts and outputs $\perp$. Otherwise, P_2 computes the labels $\{Y_{i,j}^{\widetilde{y}_k[i]}\}_{i \in [n_{\mathrm{XT}}], j \in \mathcal{E}_k}$ corresponding to the decommitments it received. Let $Y_{i,j} := Y_{i,j}^{\widetilde{y}_k[i]}$

8. **Send labels:** P_1 sends the input labels associated with its input x_k for the evaluation circuits in the kth bucket. That is, for $j \in \mathcal{E}_k$ and $i \in [n]$, P_1 sends $X'_{i,j} := g^{a_i^{x_k[i]} \cdot r_j}$ and P_2 sets $X_{i,j} := \mathsf{Ext}(X'_{i,j})$.

9. **Circuit evaluation:** For $j \in \mathcal{E}_k$, $i \in [n']$, P_2 learns $Z'_{i,j}$ by evaluating $\widehat{C}_j$. We call an output-wire label $Z'_{i,j}$ *valid* if it exists in the encoded output translation table sent in Step 2. If P_2 receives exactly *one* valid output-wire label per output wire, then let z_k denote the output. In this case, P_2 chooses random values $Z_k^0, Z_k^1 \in_R \{0,1\}^\kappa$. If P_2 receives *two* valid outputs on any output wire then it sets $Z_k^0 := Z'_{i,j_1}$ and $Z_k^1 := Z'_{i,j_2}$, where $j_1, j_2 \in \mathcal{E}_k$ denote the conflicting circuit indices. If P_2 receives *no* valid output values on any output wire, then P_2 aborts.

10. **Cheating detection:** P_1 defines a circuit C_{sc} with the values $\{Z_{1,j}^0, Z_{1,j}^1, \ldots, Z_{n',j}^0, Z_{n',j}^1\}_{j \in \mathcal{E}_k}$ hardcoded. The circuit computes the following function:

- P_1 inputs $x_k \in \{0,1\}^n$ and has no output.
- P_2 inputs a pair of values Z_k^0, Z_k^1.
- If there exist values $i \in [n']$ and $j_1, j_2 \in \mathcal{E}_k$ such that $Z_k^0 = Z_{i,j_1}^0$ and $Z_k^1 = Z_{i,j_2}^1$, then P_2's output is x_k; otherwise it receives no output.

P_1 and P_2 run the protocol of Lindell and Pinkas [9] on C_{sc} as follows:

- P_1 inputs x_k; P_2 inputs Z_k^0 and Z_k^1 as computed in Step 9.
- The garbled circuits constructed by P_1 use the same a_i^0, a_i^1 values as were chosen in Step 1, and the parties use 3ρ copies of the circuit for the cut-and-choose.

If this computation results in an abort, then both parties halt.

11. **Verify consistency of P_1's input:** Let $\widehat{\mathcal{E}}_k$ be the set of evaluation circuits used in the 2PC computation in Step 10, let $\widehat{r}_{j,k}$ be the analogous value of r_j used in that computation, and let $\widehat{X}_{i,j}$ be the analogous value of $X'_{i,j}$ used in that computation. P_1 and P_2 do the following:

    - For $i \in [n']$, P_1 uses $\mathcal{F}_{\mathbf{zk}}$ to proves that there exists some $\sigma_{i,k}$ such that for $j \in \mathcal{E}_k$ and $j' \in \widehat{\mathcal{E}}_k$, it holds that $X'_{i,j} = g^{a_i^{\sigma_{i,k}} \cdot r_j}$ and $\widehat{X}_{i,j} = g^{a_i^{\sigma_{i,k}} \cdot \widehat{r}_{j',k}}$.

    If any of the proofs fail, then P_2 aborts.

12. **Output evaluation:** If P_2 received no inconsistent outputs in Step 10, then it outputs z_k. If P_2 *did* receive inconsistent output, then let x_k be the output that P_2 received from the 2PC computation in Step 10; P_2 outputs $z_k := f(x_k, y_k)$.

Theorem 3.3. *Let ρ (resp., κ) be the statistical (resp., computational) security parameter. If the decisional Diffie-Hellman assumption holds in $(\mathbb{G}, g, q)$, H is a preimage-resistant hash function, and the circuit is garbled using an adaptively secure garbling scheme, then for all $t \in \mathsf{poly}(\kappa)$, the protocol described above securely computes $\mathcal{F}_{\mathbf{seq}}$ in the presence of a malicious adversary with error at most $2^{-\rho} + \mathsf{negl}(\kappa)$ in the $(\mathcal{F}_{\mathbf{ct}}, \mathcal{F}_{\mathbf{ot}}, \mathcal{F}_{\mathbf{zk}})$-hybrid model.*

Proof. The proof is very similar to the parallel case. The two major changes are the use of the XOR-tree to avoid the selective failure attack (in place of $\mathcal{F}_{\mathbf{mcot}}$), and the

use of adaptively secure garbling.

Malicious P_1. The intuition here is the same as for the parallel execution setting, and thus we jump straight to the simulator. Let $\mathcal{A}$ be an adversary controlling P_1. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ acts as an honest P_2 would for the entire offline phase of the protocol.

2. $\mathcal{S}$ acts as an honest P_2 would for each of the t executions of the online phase, using $y_k := 0^\ell$ as its input for each iteration, except that $\mathcal{S}$ extracts $\mathcal{A}$'s input x_k from its query to $\mathcal{F}_{\mathbf{zk}}$ in Step 11 and sends it to $\mathcal{F}_{\mathbf{seq}}$.

3. $\mathcal{S}$ halts and outputs whatever $\mathcal{A}$ outputs.

We now claim that the probability that $\mathcal{A}$ aborts when interacting with P_2 is negligibly different from the probability that $\mathcal{A}$ aborts when interacting with $\mathcal{S}$. We show this by a series of hybrids.

$\mathbf{H_1}$. The hybrid-world execution.

$\mathbf{H_2}$. $\mathcal{S}$ extracts $\mathcal{A}$'s input x_k from its query to $\mathcal{F}_{\mathbf{zk}}$ in Step 11 and sends it to $\mathcal{F}_{\mathbf{seq}}$.

These two hybrids differ if the output of $\mathcal{F}_{\mathbf{seq}}$ differs from what P_2 receives in $\mathbf{H_1}$. The argument is very similar to the one made in the parallel case. Note that after Step 2, whether a garbled circuit is "correct" or not is fixed. We now need to argue that for P_2's outputs to differ in these two hybrids it must be the case that all of the circuits in a given bucket are incorrect, which happens with probability $\leq 2^{-\rho}$. This follows directly as is done in the parallel case, and thus we do not repeat the details here.

$\mathbf{H_3}$. $\mathcal{S}$ uses input $y_k := 0^\ell$.

These two hybrids differ if the probability of $\mathcal{S}$ aborting based on its real input y_k and its fake input 0^ℓ differs. This happens if $\mathcal{A}$ is able to "guess" the secret-sharing of one of $\mathcal{S}$'s input bits in the XOR-tree construction. However, by the security of the XOR-tree, this happens with probability $\leq 2^{-\rho}$.

As $\mathbf{H_3}$ is the same as the simulator $\mathcal{S}$ given above, we conclude that the two worlds are statistically indistinguishable.

Malicious P_2. Let $\mathcal{A}$ be an adversary controlling P_2. Again, the intuition here is similar to the parallel execution setting. However, we cannot use the standard simulator for garbled circuits anymore, as we need adaptively secure garbled circuits. Instead, we make use of an adaptively secure garbling simulator [37]. In particular, we need to use a simulator for the all2 definition of security, which provides fine-grained adaptive security in terms of privacy, obliviousness, and authenticity. Bellare, Hoang, and Rogaway [37] show the existence of such a simulator, which we denote by $\mathcal{S}_{gc} = (\mathcal{S}_{gc_1}, \mathcal{S}_{gc_2})$, in the random oracle model. This simulator has two "stages": $\mathcal{S}_{gc_1}$ constructs a simulated garbled circuit, and $\mathcal{S}_{gc_2}$, given input y, constructs simulated input labels for y. We also utilize the simulator for the maliciously secure 2PC protocol of Lindell and Pinkas [9], which we denote by $\mathcal{S}_{LP11}$. We construct our simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ acts exactly as an honest P_1 would for the entire offline phase of the protocol, except for the following:

   - Prior to Step 1, $\mathcal{S}$ chooses a random string $r \in \{0,1\}^{\nu t}$ such that half of

the bits in r are set to one. For $i \in [\nu t]$, if $r[i] = 1$ then $\mathcal{S}$ constructs a correctly garbled circuit and otherwise $\mathcal{S}$ uses $\mathcal{S}_{gc_1}$ to construct a simulated *adaptively-secure* garbled circuit. Now, for those circuits with $r[i] = 1$, $\mathcal{S}$ uses the input-wire labels generated by $\mathcal{S}_{gc_1}$, and otherwise $\mathcal{S}$ constructs the input-wire labels as specified in the protocol.

- In Step 3, $\mathcal{S}$ sets the output of $\mathcal{F}_{\mathbf{ct}}$ to r.

2. In the online phase, $\mathcal{S}$ runs exactly as an honest P_1 would except as follows:

   - $\mathcal{S}$ uses $x_k := 0^\ell$ as its input for each iteration.
   - In Step 7, $\mathcal{S}$ receives P_2's input $\widetilde{y}_k$ in its call to $\mathcal{F}_{\mathbf{ot}}$, computes y_k from $\widetilde{y}_k$, and sends y_k to $\mathcal{F}_{\mathbf{seq}}$, receiving back z_k. It then runs $\mathcal{S}_{gc_2}$ on z_k, receiving back encoded values $(D^0_{i,k}, D^1_{i,k})$, and sends $D^{\widetilde{y}_k[i]}_{i,k}$ to P_2 as the response from $\mathcal{F}_{\mathbf{ot}}$.
   - In Step 10, $\mathcal{S}$ uses $\mathcal{S}_{LP11}$ to simulate the execution of C_{sc}.

We now claim that the distributions from $\mathcal{A}$ interacting with P_2 in the hybrid world versus $\mathcal{A}$ interacting with $\mathcal{S}$ in the ideal world are indistinguishable. We do so by constructing a series of hybrids.

$\mathbf{H_1}$. The hybrid-world protocol.

$\mathbf{H_2}$. $\mathcal{S}$ fixes the output of $\mathcal{F}_{\mathbf{ct}}$ to be some random string $r \in \{0,1\}^{\nu t}$ as described above, and uses $\mathcal{S}_{gc_1}$ to construct simulated adaptively-secure garbled circuits for those cases where $r[i] = 1$. It then extracts $\mathcal{A}$'s input y_k from the call to

$\mathcal{F}_{\mathbf{ot}}$ in Step 7 and sends y_k to $\mathcal{F}_{\mathbf{seq}}$, receiving back z_k. It then runs $\mathcal{S}_{gc_2}$ as specified in the simulator description.

These two hybrids are computationally indistinguishable by the security of the adaptively-secure garbled circuit simulator [37].

$\mathbf{H_3}$. $\mathcal{S}$ replaces the real 2PC execution in Step 10 with a simulated execution using $\mathcal{S}_{LP11}$.

These two hybrids are indistinguishable by the security of $\mathcal{S}_{LP11}$ [9].

$\mathbf{H_4}$. $\mathcal{S}$ uses $x_k = 0^\ell$ throughout.

As in the parallel case, computational indistinguishability holds by the decisional Diffie-Hellman assumption.

As $\mathbf{H_4}$ is the same as the simulator $\mathcal{S}$ given above, we conclude that the protocol is computationally indistinguishable. □

Chapter 4: The Publicly Verifiable Covert Setting

As mentioned in Chapter 1, Aumann and Lindell [19] introduced a very practical compromise between the semi-honest and malicious security models, that of *covert* security. In the covert security model, a party can deviate arbitrarily from the protocol description but is caught with a fixed probability ϵ, called the *deterrence factor*. In many practical scenarios, this guaranteed risk of being caught (likely resulting in loss of business and/or embarrassment) is sufficient to deter would-be cheaters. Importantly, covert protocols are much more efficient and simpler than their malicious counterparts.

At the same time, the cheating deterrent introduced by the covert model is relatively weak. Indeed, a party catching a cheater certainly knows what happened and can respond accordingly, for example by taking their business elsewhere. However, the impact is largely limited to this, since the honest player cannot credibly *accuse* the cheater publicly. If, however, credible public accusation were possible, the deterrent for the cheater would be immeasurably greater: suddenly, *all* the cheater's customers would be aware of the cheating and thus any cheating may affect the cheater's global customer base.

The addition of credible accusation greatly improves the covert model even in

scenarios with a small number of players, such as those involving the government. Consider, for example, the setting where two agencies are engaged in secure computation on their respective classified data. The covert model may often be insufficient here. Indeed, consider the case where one of the two players deviates from the protocol, perhaps due to an insider attack. The honest player detects this, but we are now faced with the problem of identifying the culprit across two domains, where the communication is greatly restricted due to trust, policy, data privacy legislation, or all of the above. On the other hand, credible accusation immediately provides the ability to exclude the honest player from the suspect list, and focus on tracking the problem *within one organization/trust domain*, which is dramatically simpler.

PVC definition and protocol. Asharov and Orlandi [20] proposed a security model, *covert with public verifiability*, and an associated protocol, addressing these concerns. At a high level, they proposed that when cheating is detected, the honest player is able to publish a "certificate of cheating" which can be checked by any third party. In this work, we abbreviate their model as *PVC: publicly verifiable covert*. Their proposed protocol (which we call the "AO protocol") has performance similar to the original covert protocol of Aumann and Lindell [19], with the exception of requiring signed-OT, a special form of oblivious transfer (OT). Their signed-OT construction is based on the OT of Peikert et al. [48], and thus requires several expensive public-key operations.

In this work, we propose several critical performance improvements to the AO protocol. Our most technically involved contribution is a novel signed-OT *extension*

protocol which eliminates per-instance public-key operations. Before discussing our contributions and technical approach in Section 4.1, we review the AO protocol.

The Asharov-Orlandi (AO) PVC protocol [20]. The AO protocol is based on the covert construction of Aumann and Lindell [19]. Let P_1 be the circuit generator, P_2 be the evaluator, and $C(\cdot,\cdot)$ be the circuit to be computed. Recall the standard garbled circuit construction in the semi-honest model: P_1 constructs a garbling of C and sends it to P_2 along with the wire labels associated with its input. The parties then run OT, with P_1 acting as the sender and inputting the wire labels associated with P_2's input, and P_2 acting as the receiver and inputting as its choice bits the associated bits of its input.

We now adapt this protocol to the PVC setting. Recall the "selective failure" attack on P_2's input wires, where P_1 can send P_2 via OT an invalid wire label for one P_2's two inputs and learn one of P_2's input bits based on whether P_2 aborts. To protect against this attack, the parties construct $C'(x_1, x_2^1, \ldots, x_2^\nu) = C(x_1, \bigoplus_{i\in[\nu]} x_2^i)$, where ν is the *XOR-tree replication factor*, and compute C' instead of C. P_1 then constructs λ (the *garbled circuit replication factor*) garblings of C' and P_2 checks that $\lambda - 1$ of the garbled circuits are correctly constructed, evaluating the remaining garbled circuit to derive the output. The main difficulty of satisfying the PVC model is ensuring that neither party can improve its odds by aborting (e.g., based on the other party's challenge). For example, if P_1 could abort whenever P_2's challenge would reveal P_1's cheating, this would enable P_1 to cheat without the risk of generating a proof of cheating. Thus, P_1 sends the garbled circuits to P_2 through

a 1-out-of-λ OT; namely, in the ith input to the OT P_1 provides openings for all the garbled circuits but the ith, as well as the input-wire labels needed to evaluate $\widehat{C}_i$. Party P_2 inputs a random γ, checks that all garbled circuits besides $\widehat{C}_\gamma$ are constructed correctly, and if so, evaluates $\widehat{C}_\gamma$.

Finally, it is necessary for P_1 to operate in a *verifiable* manner, so that an honest P_2 has proof if P_1 tries to cheat and gets caught. (Note that garbled circuits guarantee that P_2 cannot cheat in the evaluation step at all, so we only worry about catching P_1.) The AO protocol addresses this by having P_1 sign all its messages and the parties using *signed*-OT in place of all standard OTs (including wire label transfers and garbled circuit openings). Informally, the signed-OT functionality proceeds as follows: rather than the receiver P_2 getting message m_b from the sender P_1 for choice bit b, P_2 receives $((b, m_b), \sigma)$, where σ is P_1's signature of (b, m_b). This guarantees that if P_2 detects any cheating by P_1, it has P_1's signature on an inconsistent set of messages, which can be used as proof of this cheating. Asharov and Orlandi show that this construction is ϵ-PVC-secure for $\epsilon = (1-1/\lambda)(1-2^{-\nu+1})$.

4.1 Our Contribution

Our main contribution is a signed-OT extension protocol built on the recent malicious OT extension of Asharov et al. [49]. Informally, signed-OT extension ensures that (1) a cheating sender P_1 is held accountable in the form of a "certificate of cheating" that the honest receiver P_2 can generate, and (2) a malicious P_2 cannot *defame* an honest P_1 by presenting a false "certificate of cheating". Achieving the

first goal is fairly straightforward by having P_1 simply sign all its messages. The challenge is in simultaneously protecting against a malicious P_2. In particular, we need to commit P_2 to its particular choices throughout the OT extension protocol to prevent it from defaming an honest P_1, while maintaining that those commitments do not leak any information about P_2's choices.

In the standard OT extension protocol of Ishai et al. [7] (cf. Figure 4.3), P_2 constructs a random matrix M, and P_1 obtains a matrix M' derived from M, P_1's random string s and P_2's vector of OT inputs r. The key challenge of adapting this protocol to the signed variant is to efficiently prevent P_2 from submitting a malleated M as part of the proof without it ever explicitly revealing M to P_1 (as this would leak P_2's choice bits). We achieve this by observing that P_1 does in fact learn some of M, as in the OT extension construction some of the columns of M and M' are the same (i.e., those corresponding to zero bits of P_1's string s). We prevent P_2 from cheating by having P_1 include in its signature carefully selected information from the columns in M which P_1 sees. Finally, we require that P_2 generates each row of M from a seed, and that P_2's proof of cheating includes this seed such that the row rebuilt from the seed is consistent with the columns included in P_1's signature. We show that this makes it infeasible for P_2 to successfully present an invalid row in the proof of cheating. We describe this approach in greater detail in Section 4.3.[1]

As another contribution, we present a more communication efficient PVC protocol, building off the AO protocol; see Section 4.4. Our main (simple) trick there

[1]Our construction is also interesting from a theoretical perspective in that we construct signed-OT from *any* maliciously secure OT protocol, whereas Asharov and Orlandi [20] build a specific construction based on the decisional Diffie-Hellman assumption.

is a careful amendment allowing us to send garbled circuit *hashes* instead of the garbled circuits themselves; this is based on an idea from Goyal et al. [50].

All of our results are in the random oracle model, a slight strengthening of the assumptions needed for standard OT extension and free-XOR, two standard secure computation tools.

Comparison with existing approaches. The cost of our protocol is almost the same as that of the covert protocol of Goyal et al. [50]; the only extra cost is essentially a $\approx 67\%$ wider OT extension matrix and four signatures. This often negligible additional overhead (versus covert protocols) provides us with dramatically stronger (than covert) deterrent. We believe that our PVC protocol could be used in many applications where covert security is insufficient at the order-of-magnitude cost advantage over previously-needed malicious protocols or the PVC protocol of Asharov and Orlandi [20]. See Section 4.5 for more details.

4.2 Preliminaries

Let τ denote the field size. When considering concrete costs, we utilize the security parameter and field size settings for key lengths recommended by NIST [51]; see Figure 4.1.

Our constructions are in the $\mathcal{F}_{\mathsf{PKI}}$ model, where each party P_i can register a verification key, and other parties can retrieve P_i's verification key by querying $\mathcal{F}_{\mathsf{PKI}}$ on id_i. We use the notation $\mathsf{Sign}_{P_i}(\cdot)$ to denote a signature signed by P_i's secret key, and we assume that this signature can be verified by any third party. We often leave

Security	κ	FCC	ECC
Short	80	1024	160
Long	128	3072	256

Figure 4.1: Settings for (computational) security parameter κ and field size τ for various security settings as recommended by NIST [51]. FCC denotes the setting of τ when using finite field cryptography and ECC denotes the setting of τ when using elliptic curve cryptography.

off the subscript if the identity of the signing party is clear.

4.2.1 Covert Security

We review the definition of covert security by Aumann and Lindell [19], and in particular, their "strong explicit cheat" formulation. The main idea with covert security is that a malicious party is allowed to cheat with some probability $1 - \epsilon$, but gets caught with probability ϵ. In the following, we give the definition for the specific case of two-parties, although the definition can be easily generalized to the multi-party setting.

Ideal model execution. In the ideal model, we have parties P_1 and P_2, and an adversary $\mathcal{A}$ with auxiliary input aux who can corrupt one of the two parties. Let $\mathcal{F}$ define the ideal functionality implementing $f(\cdot, \cdot)$.

- P_1 obtains input x and P_2 obtains input y.
- An honest party sends its given input to the ideal functionality $\mathcal{F}$, whereas a corrupted party can send an arbitrary input. Denote the inputs send to $\mathcal{F}$ as x' and y'.
- A corrupted party may send one of the following messages to $\mathcal{F}$:

– abort: In this case, $\mathcal{F}$ sends abort to the honest party and halts.

– corrupted: In this case, $\mathcal{F}$ sends corrupted to the honest party and halts.

– cheat: In this case, there are two possibilities:

* With probability ϵ, $\mathcal{F}$ sends corrupted to both parties and halts;

* With probability $1 - \epsilon$, $\mathcal{F}$ sends undetected and the honest party's input to the corrupted party, waits for an output value z from the corrupted party, and sends z to the honest party.

– continue: In this case, $\mathcal{F}$ continues.

- $\mathcal{F}$ computes $z := f(x', y')$ and sends z to the corrupted party.
- The corrupted party sends either abort or continue to $\mathcal{F}$. If $\mathcal{F}$ receives continue it sends z to the honest party, and if $\mathcal{F}$ receives abort it sends abort to the honest party and halts.
- The honest party outputs the given output from $\mathcal{F}$, whereas the corrupted party outputs an arbitrary function of its view of the protocol execution.

Let $\text{IDEALC}^{\epsilon}_{\mathcal{F},\mathcal{A}(\mathsf{aux})}(x, y, 1^{\kappa})$ denote the joint output of the adversary $\mathcal{A}$ and the honest party with inputs x and y when interacting with ideal functionality $\mathcal{F}$.

Real model execution. This is the same as the real model execution as described in Chapter 2 for malicious security.

Definition 4.1. *Protocol Π_f secure computes $\mathcal{F}$ in the presence of covert adversaries with ϵ-deterrent if for every* PPT *adversary $\mathcal{A}$ in the real model, there exists a* PPT

simulator $\mathcal{S}$ *in the ideal model such that for all* x *and* y *it holds that*

$$\{\text{IDEAL}^{\epsilon}_{\mathcal{F},\mathcal{S}(\mathsf{aux})}(x, y, 1^{\kappa})\} \overset{c}{\approx} \{\text{REAL}_{\Pi_f,\mathcal{A}(\mathsf{aux})}(x, y, 1^{\kappa})\}.$$

4.2.2 Publicly Verifiable Covert Security

We now review the *publicly verifiable covert* (PVC) security model of Asharov and Orlandi [20]. When we say a protocol is "secure in the covert model" we assume it is secure under Definition 4.1.

Let π be a two-party protocol between parties P_1 and P_2 implementing function f. Following Aumann and Lindell [19], we call π *non-halting* if for honest P_i and fail-stop adversary[2] $P_{\text{-}i}$, the probability that P_i outputs $\mathsf{corrupted}$ is negligible. Consider the triple of algorithms $(\pi', \mathsf{Blame}, \mathsf{Judgment})$ defined as follows:

- Protocol π' is the same as π except that if an honest party $P_{\text{-}i^*}$ outputs $\mathsf{corrupted}$ when executing π, it computes $\mathsf{Cert} \leftarrow \mathsf{Blame}(\mathsf{id}_{i^*}, \mathsf{key}, \mathsf{View}_{\text{-}i^*})$, where key denotes the type of cheating detected, and sends Cert to P_{i^*}.

- Algorithm Blame is a deterministic algorithm which takes as input a cheating identity id, a cheating type key, and a view View of a protocol execution, and outputs a certificate Cert.

- Algorithm $\mathsf{Judgment}$ is a deterministic algorithm which takes as input a certificate Cert and outputs either an identity id or $\perp$.

[2] A *fail-stop adversary* is one which acts semi-honestly but may halt at any time.

Before proceeding to the definition, we first introduce some notation. Let $\mathsf{Exec}_{\pi,\mathcal{A}(z)}(x_1, x_2; 1^\kappa)$ denote the transcript (i.e., messages and output) produced by P_1 with input x_1 and P_2 with input x_2 running protocol π, where adversary $\mathcal{A}$ with auxiliary input z can corrupt parties before execution begins. Let $\mathsf{Output}_{P_i}(\mathsf{Exec}_{\pi,\mathcal{A}(z)}(x_1, x_2; 1^\kappa))$ denote the output of P_i on the input transcript.

Definition 4.2. *We say that* $(\pi', \mathsf{Blame}, \mathsf{Judgment})$ *securely computes* f *in the presence of a publicly verifiable covert adversary with* ϵ*-deterrent (or, is* ϵ*-PVC-secure) if the following conditions hold:*

1. *The protocol* π' *is a non-halting and secure realization of* f *in the covert model with* ϵ*-deterrent.*

2. *(Accountability) For every* PPT *adversary* $\mathcal{A}$ *corrupting party* P_{i^*}*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that if* $\mathsf{Output}_{P_{\text{-}i^*}}(\mathsf{Exec}_{\pi,\mathcal{A}(z)}(x_1, x_2; 1^\kappa)) = \mathsf{corrupted}$ *then*

$$\Pr\left[\mathsf{Judgment}(\mathsf{Cert}) = \mathsf{id}_{i^*}\right] > 1 - \mathsf{negl}(\kappa),$$

   where $\mathsf{Cert} \leftarrow \mathsf{Blame}(\mathsf{id}_{i^*}, \mathsf{key}, \mathsf{View}_{\text{-}i^*})$ *and the probability is over the randomness used in the protocol execution.*

3. *(Defamation-free) For every* PPT *adversary* $\mathcal{A}$ *corrupting party* P_{i^*} *and outputting a certificate* Cert*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that* $\Pr\left[\mathsf{Judgment}(\mathsf{Cert}) = \mathsf{id}_{\text{-}i^*}\right] < \mathsf{negl}(\kappa)$*, where the probability is over the randomness used by* $\mathcal{A}$*.*

Note that, in particular, the PVC definition implicitly disallows Blame to reveal $P_{\neg i^*}$'s input. This is because π' specifies that Cert is sent to P_{i^*}.

4.2.3 Signed Oblivious Transfer

A central functionality for constructing PVC protocols is *signed oblivious transfer* (signed-OT), introduced by Asharov and Orlandi [20]. We can define the basic signed-OT functionality $\mathcal{F}$ as

$$(\perp, (m_b, \mathsf{Sign}_{\mathsf{sk}}(b, m_b))) \leftarrow\!\!\$\ \mathcal{F}((m_0, m_1, \mathsf{sk}), (b, \mathsf{vk})),$$

where the signature scheme is assumed to be existentially unforgeable under adaptive chosen message attack ($\mathsf{EU\text{-}CMA}$). Namely, the sender P_1 inputs two messages m_0 and m_1 along with a signing key sk; the receiver P_2 inputs a choice bit b and a verification key vk; P_1 receives no output whereas P_2 receives m_b alongside a signature on (b, m_b).

However, as in prior work [20], this definition is too strong for our signed-OT extension construction to satisfy. We introduce a relaxed signed-OT variant (slightly different from Asharov and Orlandi's variant [20]) which is tailored for OT extension and is sufficient for obtaining PVC-security. Essentially, we need a signature scheme that satisfies a weaker notion than $\mathsf{EU\text{-}CMA}$ in which the signing algorithm takes randomness, a portion of which can be controlled by the adversary.[3]

[3]Our notion is similar to the ρ-$\mathsf{EU\text{-}CMRA}$ notion introduced by Asharov and Orlandi [20]. It differs in that we allow different portions of the randomness to be corrupted, but not both portions at once. Looking forward, this is needed because the sender in our signed-OT functionality is only allowed to control some of the randomness.

This is because in our signed-OT extension construction, a malicious party can influence the randomness used in the signing algorithm. In addition, we introduce an *associated data* parameter to the signing algorithm which allows the signer to specify some additional information unrelated to the message being signed but used in the signature. In our construction, we use the associated data to tie the signature to a specific counter (such as a session ID or message ID), preventing a malicious receiver from "mixing" properly signed values to defame an honest sender.

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a tuple of PPT algorithms over message space $\mathcal{M}$, associated data space $\mathcal{D}$, and randomness spaces $\mathcal{R}_1$ and $\mathcal{R}_2$, defined as follows:

1. $\mathsf{Gen}(1^\kappa)$: On input security parameter 1^κ, output key pair $(\mathsf{vk}, \mathsf{sk})$.

2. $\mathsf{Sign}_{\mathsf{sk}}(m, a; (r_1, r_2))$: On input secret key sk, message $m \in \mathcal{M}$, associated data $a \in \mathcal{D}$, and randomness $r_1 \in \mathcal{R}_1$ and $r_2 \in \mathcal{R}_2$, output signature $\sigma = (a, \sigma')$.

3. $\mathsf{Verify}_{\mathsf{vk}}(m, \sigma)$: On input verification key vk, message $m \in \mathcal{M}$, and signature σ, output 1 if σ is a valid signature for m and 0 otherwise.

For security, we need the condition that unforgeability remains even if the adversary inputs some arbitrary r_1 or r_2. However, the adversary is prevented from inputting values for *both* r_1 and r_2. This reflects the fact that in our signed-OT extension construction, a malicious sender can control only r_1 and a malicious receiver can control only r_2. We place a further restriction that the choice of r_1 must be *consistent*; namely, all queries to Sign must use the same value for r_1. Looking ahead, this property exactly captures the condition we need (r_1 corresponds to the zero bits in

the sender's column selection string in the OT extension), where the choice of r_1 is made once and then fixed throughout the protocol execution.

Towards our definition, we define an oracle $\mathcal{O}_{\mathsf{sk}}(\cdot,\cdot,\cdot,\cdot)$ as follows. Let $\perp$ be a special symbol. On input (m, a, r_1, r_2), proceed as follows. If neither r_1 nor r_2 equal $\perp$, output $\perp$. Otherwise, proceed as follows. If $r_1 = \perp$ and r'_1 has not been set, set r'_1 uniformly at random; if $r_1 \neq \perp$ and r'_1 has not been set, set $r'_1 = r_1$; if $r_2 = \perp$, set r'_2 uniformly at random; otherwise, set $r'_2 = r_2$. Finally, output $\mathsf{Sign}_{\mathsf{sk}}(m, a; (r'_1, r'_2))$.

Now, consider the following game $\mathsf{Sig\text{-}forge}^{\mathsf{CMPRA}}_{\mathcal{A},\Pi}(\kappa)$ for signature scheme Π between PPT adversary $\mathcal{A}$ and PPT challenger $\mathcal{C}$.

1. $\mathcal{C}$ runs $(\mathsf{vk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{Gen}(1^\kappa)$ and sends vk to $\mathcal{A}$.

2. $\mathcal{A}$, who has oracle access to $\mathcal{O}_{\mathsf{sk}}(\cdot,\cdot,\cdot,\cdot)$, outputs a tuple $(m, (a, \sigma'))$. Let $\mathcal{Q}$ be the set of messages and associated data pairs input to $\mathcal{O}_{\mathsf{sk}}(\cdot,\cdot,\cdot,\cdot)$.

3. $\mathcal{A}$ succeeds if and only if (1) $\mathsf{Verify}_{\mathsf{vk}}(m, (a, \sigma')) = 1$ and (2) $(m, a) \notin \mathcal{Q}$.

Definition 4.3. *Signature scheme* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is existentially unforgeable under adaptive chosen message and partial randomness attack (EU-CMPRA) if for all* PPT *adversaries* $\mathcal{A}$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that* $\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{CMPRA}}_{\mathcal{A},\Pi}(\kappa)] < \mathsf{negl}(\kappa)$.

Signed-OT functionality. We are now ready to introduce our relaxed signed-OT functionality. As is our EU-CMPRA signature, it is tailored for OT extension, and is sufficient for building PVC protocols. This functionality, denoted by $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, is parameterized by an EU-CMPRA signature scheme Π and is defined in Figure 4.2. As

Functionality $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$

The functionality is parameterized by an EU-CMPRA signature scheme $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.

Input: The sender inputs messages m_0 and m_1 such that $|m_0| = |m_1|$, secret key sk, associated data a, randomness r_1^*, and signatures σ_0^* and σ_1^*. The receiver inputs choice bit b, verification key vk, and randomness r_2^*. If the sender (resp., the receiver) is honest, then $r_1^* = \sigma_0^* = \sigma_1^* = \perp$ (resp., $r_2^* = \perp$).

Output: The functionality computes $\sigma_b = \mathsf{Sign}_{\mathsf{sk}}((b, m_b), a; (r_1^*, r_2^*))$ for $b \in \{0, 1\}$. The sender receives no output. The receiver receives the following output based on if the sender is corrupt or not:

- If $\sigma_0^* \neq \perp$ or $\sigma_1^* \neq \perp$, the functionality outputs $((b, m_b), \sigma_b^*)$ if and only if $\mathsf{Verify}_{\mathsf{vk}}((0, m_0), \sigma_0^*) = \mathsf{Verify}_{\mathsf{vk}}((1, m_1), \sigma_1^*) = 1$, where $\sigma_b^* \leftarrow \sigma_b$ if $\sigma_b^* = \perp$; otherwise it outputs abort.
- If $\sigma_0^* = \sigma_1^* = \perp$, the functionality outputs $((b, m_b), \sigma_b)$.

Figure 4.2: Signed oblivious transfer functionality.

in standard OT, the sender inputs two messages (of equal length) and the receiver inputs a choice bit. However, in this formulation we allow a malicious sender to specify some random value r_1^* as well as signatures σ_0^* and σ_1^*. Likewise, a malicious receiver can specify some random value r_2^*. (Honest players input $\perp$ for these values.) If both players are honest, the functionality computes $\sigma \leftarrow \mathsf{Sign}((b, m_b); (r_1, r_2))$ with uniformly random values r_1 and r_2 and outputs $((b, m_b), \sigma)$ to the receiver. However, if either party is malicious and specifies some random value, this is fed into the Sign algorithm. Likewise, if the sender is malicious and specifies some signature $\sigma_b^* \neq \perp$, this value is used as the signature sent to the receiver.

Note that $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ is nearly identical to the signed-OT functionality presented by Asharov and Orlandi [20, Functionality 2]; it differs in the use of EU-CMPRA signature schemes instead of ρ-EU-CMRA schemes. We also note that it is straightforward to adapt $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ to realize OTs with more than two inputs from the sender. We let $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ denote a 1-out-of-λ variant of $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$.

A compatible commitment scheme. Our construction of an EU-CMPRA signa-

ture scheme uses a non-interactive commitment scheme, which we define here. Our definition follows the standard commitment definition, except we tweak the Com algorithm to take an additional associated data value.

Let $\Pi_{\mathsf{Com}} = (\mathsf{ComGen}, \mathsf{Com})$ be a tuple of PPT algorithms over message space $\mathcal{M}$ and associated data space $\mathcal{D}$, defined as follows:

1. $\mathsf{ComGen}(1^\kappa)$: On input security parameter 1^κ, compute parameters params.

2. $\mathsf{Com}(m, a; r)$: On input message $m \in \mathcal{M}$, associated data $a \in \mathcal{D}$, and randomness r, output commitment com.

A commitment can be opened by revealing the randomness r used to construct that commitment.

We now define security for our commitment scheme. We only consider the *binding* property; namely, the inability for a PPT adversary to open a commitment to some other value than that committed to. Security is the same as for standard commitment schemes, except we allow the adversary to control the randomness used in ComGen.

Consider the game $\mathsf{Com\text{-}bind}_{\mathcal{A},\Pi_{\mathsf{Com}}}(\kappa)$ for commitment scheme Π_{Com} between a PPT adversary $\mathcal{A}$ and a PPT challenger $\mathcal{C}$, defined as follows.

1. $\mathcal{A}$ sends randomness r to $\mathcal{C}$.

2. $\mathcal{C}$ computes $\mathsf{params} \leftarrow \mathsf{ComGen}(1^\kappa; r)$ and sends params to $\mathcal{A}$.

3. $\mathcal{A}$ outputs $(\mathsf{com}, m_1, a_1, r_1, m_2, a_2, r_2)$ and wins if and only if (1) $m_1 \neq m_2$, and (2) $\mathsf{com} = \mathsf{Com}(\mathsf{params}, m_1, a_1; r_1) = \mathsf{Com}(\mathsf{params}, m_2, a_2; r_2)$.

Definition 4.4. *A commitment scheme* $\Pi_{\mathsf{Com}} = (\mathsf{ComGen}, \mathsf{Com})$ *is (computationally)* ***binding*** *if for all* PPT *adversaries* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that* $\Pr[\mathsf{Com\text{-}bind}_{\mathcal{A},\Pi_{\mathsf{Com}}}(\kappa)] < \mathsf{negl}(\kappa)$.

4.3 Signed Oblivious Transfer Extension

We now present our main contribution: an efficient instantiation of signed oblivious transfer (signed-OT) extension. We begin by describing in detail the logic of the construction, iteratively building it up from the passively secure protocol of Ishai et al. [7]. We then motivate the need for EU-CMPRA signature schemes and present a compatible such scheme.

Intuition for the Construction

Consider the OT extension protocol of Ishai et al. [7] in Figure 4.3, run between sender P_1 and receiver P_2. This protocol is secure against a semi-honest P_2 and malicious P_1. We show how to convert this protocol into one which satisfies the $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ functionality defined in Figure 4.2. For clarity of presentation, we build on the protocol of Figure 4.3 and later discuss how to support a malicious P_2 as well, based on the malicious OT extension protocol of Asharov et al. [49].

As a first attempt, suppose P_1 simply signs all its messages in Step 3. Recall that we will use this construction to have P_1 send the appropriate input wire labels to P_2; namely, P_1 acts as P_1 in the OT extension and inputs the wire labels for P_2's input wires whereas P_2 acts as P_2 and inputs its input bits. Thus, our first

P_1**'s inputs:** Message pairs $\{(X_j^0, X_j^1)\}_{j\in[m]}$, where each $X_j^0, X_j^1 \in \{0,1\}^n$.
P_2**'s inputs:** Selection bit vector $r \in \{0,1\}^m$.
Common inputs: Security parameter κ; number of base OTs ℓ $(= \kappa)$; hash function $H : \mathbb{N} \times \{0,1\}^\ell \to \{0,1\}^n$; ideal functionality $\mathcal{F}_{\mathbf{ot}}$.

1. Initial OT Phase:
   - P_1 computes $s \in_R \{0,1\}^\ell$.
   - P_2 generates a random $m \times \ell$ matrix T, where the jth row is t_j and the ith column is t^i. Likewise, P_2 generates a random $m \times \ell$ matrix V, where the jth row is v_j and the ith column is v^i.
   - P_1 and P_2 run $\mathcal{F}_{\mathbf{ot}}$ ℓ times in parallel, where P_1 acts as the *receiver* with input s_i in the ith OT and P_2 acts as the *sender* with input (t^i, v^i) in the ith OT.
2. OT Extension Phase (Part I):
   - For $i \in [m]$, P_2 sets $u^i := t^i \oplus v^i \oplus r$, and sends u^i to P_1.
3. OT Extension Phase (Part II):
   - Let Q be the $m \times \ell$ matrix where each column $q^i = (s_i \cdot (u^i \oplus v^i)) \oplus ((1 - s_i) \cdot t^i)$. Note that $q^i = (s_i \cdot r) \oplus t^i$ and $q_j = (r[j] \cdot s) \oplus t_j$.
   - For $j \in [m]$, P_1 computes $\widehat{X}_j^0 := X_j^0 \oplus H(j, q_j)$ and $\widehat{X}_j^i := X_j^i \oplus H(j, q_j \oplus s)$, and sends $\widehat{X}_j^0$ and $\widehat{X}_j^1$ to P_2.
   - For $j \in [m]$, P_2 computes $X_j := \widehat{X}_j^{r[j]} \oplus H(j, t_j)$.
4. Output:
   - P_1 outputs $\perp$ and P_2 outputs $\{X_j\}_{j\in[m]}$.

Figure 4.3: Protocol implementing passively secure OT extension [52, 7].

step is to enhance the protocol in Figure 4.3 to have P_1 send $\sigma' \leftarrow \mathsf{Sign}(j, \widehat{X}_j^0)$ and $\sigma'' \leftarrow \mathsf{Sign}(j, \widehat{X}_j^1)$ in Step 3.

Now, if P_2 gets an invalid (with respect to a signed garbled circuit sent in the PVC protocol of Section 4.4) wire label X_j, it can easily construct a certificate Cert which demonstrates P_1's cheating. Namely, it outputs as its certificate the tuple $\left(b, j, \widehat{X}_j^0, \widehat{X}_j^1, \sigma', \sigma'', t_j\right)$ along with the (signed by P_1 and opened) garbled circuit containing the invalid wire label. A third party can (1) check that σ' and σ'' are valid signatures and (2) compute $X_j^b := H(j, t_j) \oplus \widehat{X}_j^b$ and check that X_j^b is indeed

an invalid wire label for the given garbled circuit.

This works for protecting against a malicious P_1; however, note that P_2 can easily *defame* an honest P_1 by outputting $t_j^* \neq t_j$ as part of its certificate (in which case $X_j^b := H(j, t_j^*) \oplus \widehat{X}_j^b$ will very likely be an invalid wire label). Thus, the main difficulty in constructing signed-OT extension is tying P_2 to its choice of the matrix T generated in Step 1 of the protocol so it cannot blame an honest P_1 by using invalid rows t_j^* in its certificate.

Towards this end, consider the following modification. In Step 1, P_2 now additionally sends commitments to each t_j to P_1, and P_1 signs these and sends them as part of its messages in Step 3. This prevents P_2 from later changing t_j to blame P_1. This does not quite work, however, as P_2 could simply commit to an incorrect t_j^* in the first place! Clearly, P_2 cannot send T to P_1, as this would leak P_2's selection bits, yet we still need P_2 to somehow be committed to its choice of the matrix T. The key insight is noting that P_1 does in fact know *some* of the bits of T; namely, it knows those columns at which $s_i = 0$ (as it learns t^i in the base OT). We can use this information to tie P_2 to its choice of T such that it cannot later construct some matrix $T^* \neq T$ to defame P_1.

We do this by enhancing Step 3 as follows. Let I^0 be the set of indices i such that $s_i = 0$ (recall that s is the random selection bits of P_1 input to the base OTs in Step 1). Let $t_{j,i}$ denote the ith bit in row t_j. Note that P_1 knows the values of $t_{j,i}$ for $i \in I^0$, and could thus compute $\{(i, t_{j,i})\}_{i \in I^0}$ as a "binding" of P_2's choice of t_j. By including this information in its signature, P_1 enforces that any t_j^* that P_2 tries to use to blame P_1 must match in the given positions. This brings us closer to our

goal; however, there are still two issues that we need to resolve:

1. Sending $\{(i, t_{j,i})\}_{i \in I}$ to P_2 leaks s, which allows P_2 to learn both of P_1's inputs. We address this by increasing the number of base OTs in Step 1 and having P_1 only send some subset $I \subseteq I^0$ such that $|I| = \kappa$. Thus, while P_2 learns that $s_i = 0$ for $i \in I$, by increasing the number of base OTs enough, P_2 does not have enough information to recover s.

2. P_2 can still flip one bit in t_j and pass the check with high probability. We fix this by having each t_j be generated by a seed k_j. Namely, P_2 computes $t_j \leftarrow G(k_j)$ in Step 1, where G is a random oracle[4]. Then, when blaming P_1, P_2 must reveal k_j instead of t_j. Thus, with high probability a malicious P_2 cannot find some $k_j^* \neq k_j$ such that the Hamming distance between $G(k_j^*)$ and $G(k_j)$ is small enough that the above check succeeds.

Finally, note that we have thus far considered the passively secure OT extension protocol, which is insecure against a malicious P_2. We thus utilize the maliciously secure OT extension protocol of Asharov et al. [49]. The only way P_2 can cheat in passively secure OT extension is by using different r values in Step 2. Asharov et al. add a "consistency check" phase between Steps 2 and 3 to enforce that r is consistent. This does not affect our construction, and thus we can include this step to complete the protocol.[5] We refer the reader to Asharov et al. [49] for

[4]Note that G *cannot* be a pseudorandom generator because the input to G is not necessarily uniform as the inputs may be adversarially chosen by P_2.

[5]The reason this does not affect our construction is because the consistency check phase only involves P_2 sending messages to P_1. A malicious P_2 cannot defame P_1 because we are only enforcing that P_2's value r is consistent.

the justification and intuition of this step; as far as this work is concerned we can treat this consistency check as a "black box".

We make two key observations regarding our construction:

1. **OT extension matrix size**: We set ℓ, the number of base OTs, so that leaking κ bits to P_2 does not allow it to recover s and thus both messages. We do this as follows. Let ℓ' be the number of base OTs required in malicious OT extension [49]. We set $\ell = \ell' + \kappa$ and require that when P_1 chooses s, it first fixes κ randomly selected bits to zero before randomly setting the rest of the bits. Now, when P_1 reveals I to P_2, the number of unknown bits in s is equal to ℓ' and thus the security of the Asharov et al. scheme carries over to our setting. Asharov et al. set $\ell' \approx 1.6\kappa$, and thus us using κ extra columns results in an $\approx 67\%$ matrix size increase.

2. **Batching signatures**: The main computational cost of our protocol is the signatures sent by P_1 in Step 4. This cost can easily be brought to negligible, as follows. Recall that when using our protocol for transferring the input wire labels of a garbled circuit using free-XOR we can optimize the communication slightly by setting $X_j^0 := H(j, q_j)$ and $\widehat{X}_j^1 := X_j^0 \oplus \Delta \oplus H(j, q_j \oplus s)$, where Δ is the free-XOR global offset. Thus, P_1 only needs to send (and sign) $\widehat{X}_j^1$.

   The most important idea, however, is to batch messages across OT executions and have P_1 sign (and send) only *one* signature which includes all the necessary information across many OTs. Namely, using the free-XOR optimization above, P_1 signs and sends the tuple $(I, \{\widehat{X}_j^1, \{t_{j,i}\}_{i \in I}\}_{j \in [m]})$ to P_2. We note that

the j values need not be sent as they are implied by the protocol execution.

Figure 4.4 gives the full protocol for signed-OT extension. For clarity of presentation, this description, and the following proof of security, does not take into account the batching signatures optimization described above.

Towards a Proof of Security

Before presenting the security proof, we first motivate the need for EU-CMPRA signature schemes. Ideally we could just have P_1 sign everything using an EU-CMA signature scheme; however, this presents opportunities for P_2 to defame P_1. Thus, we need to enforce that P_2 cannot output an x_j^b value different from the one sent by P_1. We do so by using a binding commitment scheme $\Pi_{\mathsf{Com}} = (\mathsf{ComGen}, \mathsf{Com})$, and show that the messages sent by P_1 in Step 4 are essentially binding commitments to the underlying X_j^b values.

We define Π_{Com} as follows, where $G : \{0,1\}^\kappa \to \{0,1\}^\ell$ and $H : \mathbb{N} \times \{0,1\}^\ell \to \{0,1\}^\kappa$ are random oracles, and $\ell \geq \kappa$.

1. $\mathsf{ComGen}(1^\kappa)$: choose set $I \subseteq [\ell]$ uniformly at random subject to $|I| = \kappa$; output $\mathsf{params} := I$.

2. $\mathsf{Com}(\mathsf{params}, m, j; r)$: On input parameters $I := \mathsf{params}$, message m, counter j, and randomness $r \in \{0,1\}^\kappa$, proceed as follows. Compute $t := G(r)$, set $\mathsf{com} := (j, m \oplus H(j,t), I, \{t_i\}_{i \in I})$, and output com.

We make the assumption that given I, one can derive the randomness input to ComGen. (We use this when defining our EU-CMPRA signature scheme below,

which uses a generic binding commitment scheme). We can satisfy this by simply letting the randomness input to ComGen be the set I.

In our signed-OT extension protocol, the set I chosen by P_1 is used as params and the k_j values chosen by P_2 are used as the randomness to Com. The commitment value com is exactly the message signed and sent by P_1 in Step 4. Thus, ignoring the signatures for now, we have an OT extension protocol that binds P_1 to its X_j^b values, and thus prevents a malicious P_2 from defaming an honest P_1. Adding in the signatures gives us an $\mathsf{EU\text{-}CMPRA}$ signature scheme. Namely, P_1 is tied to its messages due to the signatures and P_2 is prevented from "changing" the messages to defame P_1 due to the binding property of the commitment scheme.

We now prove that the commitment scheme described above is binding. We actually prove something stronger than what is required in our protocol. Namely, we prove that an adversary who can control *both* random values still cannot win, whereas when we use this commitment scheme in our signed-OT extension protocol, only one of the two random values can be controlled by any one party.

Theorem 4.1. *Protocol* Π_{Com} *is binding according to Definition 4.4.*

Proof. Adversary $\mathcal{A}$ needs to come up with choices of I, m, m', j, j', r, and r' such that $(j, m \oplus H(j,t), I, \{t_i\}_{i\in I}) = (j', m' \oplus H(j',t'), I, \{t'_i\}_{i\in I'})$, where $t := G(r)$ and $t' := G(r')$. Clearly, $j = j'$. Thus, $\mathcal{A}$ must find t and t' such that $t_i = t'_i$ for all $i \in I$. However, by the property that G is a random oracle, the values t and t' are distributed uniformly at random in $\{0,1\}^\ell$. Thus, the probability that $\mathcal{A}$ finds two bitstrings t and t' that match in κ bits is negligible, regardless of the choice of I. $\square$

An EU-CMPRA Signature Scheme

We now show that the messages sent by P_1 in Step 4 form an EU-CMPRA signature scheme. Let $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Verify}')$ be an EU-CMA signature scheme and $\Pi_{\mathsf{Com}} = (\mathsf{ComGen}, \mathsf{Com})$ be a commitment scheme satisfying Definition 4.4. Consider the scheme $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ defined as follows.

1. $\mathsf{Gen}(1^\kappa)$: On input 1^κ, run $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Gen}'(1^\kappa)$ and output $(\mathsf{vk}, \mathsf{sk})$.

2. $\mathsf{Sign}_{\mathsf{sk}}(m, j; (r_1^*, r_2^*))$: On input message $m \in \{0,1\}^\kappa$, counter $j \in \mathbb{N}$, and randomness r_1^* and r_2^*, proceed as follows. Compute $\mathsf{params} := \mathsf{ComGen}(1^\kappa; r_1^*)$ and $\mathsf{com} := \mathsf{Com}(\mathsf{params}, m, j; r_2^*)$. Next, choose $m' \in_R \{0,1\}^\kappa$ and compute $\mathsf{com}' := \mathsf{Com}(\mathsf{params}, m', j; r_2^*)$.[6] Output $\sigma := (j, \mathsf{params}, r_2^*, \mathsf{com}, \mathsf{com}', \mathsf{Sign}'_{\mathsf{sk}}((\mathsf{params}, \mathsf{com})), \mathsf{Sign}'_{\mathsf{sk}}((\mathsf{params}, \mathsf{com}')))$.

3. $\mathsf{Verify}_{\mathsf{pk}}(m, \sigma)$: On input message m and signature σ, parse σ as $(j, \mathsf{params}, r, \mathsf{com}', \mathsf{com}'', \sigma', \sigma'')$, and output 1 if and only if (1) $\mathsf{Com}(\mathsf{params}, m; r) = \mathsf{com}'$, (2) $\mathsf{Verify}'_{\mathsf{vk}}((\mathsf{params}, \mathsf{com}'), \sigma') = 1$, and (3) $\mathsf{Verify}'_{\mathsf{vk}}((\mathsf{params}, \mathsf{com}''), \sigma'') = 1$; otherwise output 0.

As will be clear later, this signature scheme exactly captures the behavior of P_1 in our signed-OT extension protocol. We now prove that this is indeed an EU-CMPRA signature scheme.

Theorem 4.2. *Given an EU-CMA signature scheme* $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Verify}')$ *and a commitment scheme* $\Pi_{\mathsf{Com}} = (\mathsf{ComGen}, \mathsf{Com})$ *secure according to Definition 4.4,*

[6] This extra commitment on a random message is needed for our signed-OT extension proof.

then $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *described above is an* ***EU-CMPRA*** *signature scheme.*

Proof. Let $\mathcal{A}$ be a PPT adversary attacking Π. We construct an adversary $\mathcal{B}$ attacking Π'. Adversary $\mathcal{B}$ receives vk from the challenger and initializes $\mathcal{A}$ with vk as input. Let (m, j, r_1^*, r_2^*) be the input of $\mathcal{A}$ to its signing oracle. Adversary $\mathcal{B}$ emulates the execution of $\mathcal{A}$'s signing oracle as follows: it computes $\mathsf{params} := \mathsf{ComGen}(1^\kappa; r_1^*)$ and $\mathsf{com} := \mathsf{Com}(\mathsf{params}, m, j; r_2^*)$, chooses m' uniformly at random and computes $\mathsf{com}' := \mathsf{Com}(\mathsf{params}, m', j; r_2^*)$, constructs $\sigma := (j, \mathsf{params}, r_2^*, \mathsf{com}, \mathsf{com}', \mathsf{Sign}'_{\mathsf{sk}}((\mathsf{params}, \mathsf{com})), \mathsf{Sign}'_{\mathsf{sk}}((\mathsf{params}, \mathsf{com}')))$, and sends σ to $\mathcal{A}$. After each of $\mathcal{A}$'s queries, $\mathcal{B}$ stores (m, j) in set $\mathcal{Q}_\mathcal{A}$ and stores all the messages it sent to its signing oracle in set $\mathcal{Q}_\mathcal{B}$.

Eventually, $\mathcal{A}$ outputs $(m, (j, \sigma'))$ as its forgery. Adversary $\mathcal{B}$ checks that $\mathsf{Verify}_{\mathsf{vk}}(m, (j, \sigma')) = 1$ and that $(m, j) \notin \mathcal{Q}_\mathcal{A}$. If not, $\mathcal{B}$ outputs 0. Otherwise, $\mathcal{B}$ parses σ' as $(\mathsf{params}, r, \mathsf{com}', \mathsf{com}'', \sigma', \sigma'')$ and checks that $\mathsf{com}' \notin \mathcal{Q}_\mathcal{B}$. If so, it outputs (com', σ'); otherwise it outputs 0.

Note that $\mathsf{Sig\text{-}forge}^{\mathsf{CMPRA}}_{\mathcal{A},\Pi}(\kappa) = 1$ and $\mathsf{Sig\text{-}forge}^{\mathsf{CMA}}_{\mathcal{B},\Pi'}(\kappa) = 0$ if and only if $\mathsf{Verify}_{\mathsf{vk}}(m, (j, \mathsf{params}, r, \mathsf{com}', \mathsf{com}'', \sigma', \sigma'')) = 1$ and $(m, j) \notin \mathcal{Q}_\mathcal{A}$ but $\mathsf{com}' \in \mathcal{Q}_\mathcal{B}$. Fix some $(m, (j, \mathsf{params}, r, \mathsf{com}_1, \mathsf{com}_{1'}, \sigma_1, \sigma_{1'}))$ such that this is the case. Thus it holds that $\mathsf{com}_1 \in \mathcal{Q}_\mathcal{B}$. This implies that $\mathcal{B}$ queried Sign' on com_1, which means that $\mathcal{A}$ queried *its* signing oracle on some (m', j', r_1^*, r_2^*), where $m' \neq m$, and received back $(j', \mathsf{params}, r', \mathsf{com}_1, \mathsf{com}_{2'}, \sigma_{1''}, \sigma_{2'})$. However, this implies that $\mathsf{Com}(\mathsf{params}, \mathsf{com}_1; r) = m$ and $\mathsf{Com}(\mathsf{params}, \mathsf{com}_1; r') = m'$. Thus, $\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{CMPRA}}_{\mathcal{A},\Pi}(\kappa)] = \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{CMA}}_{\mathcal{B},\Pi}(\kappa)] + \Pr[\mathsf{Com\text{-}bind}_{\mathcal{B}',\Pi_{\mathsf{Com}}}(\kappa)]$ for some PPT adversary $\mathcal{B}'$. We now bound $\Pr[\mathsf{Com\text{-}bind}_{\mathcal{B}',\Pi_{\mathsf{Com}}}(\kappa)]$.

Adversary $\mathcal{B}'$ runs almost exactly like $\mathcal{B}$. On the first query (m, j, r_1^*, r_2) by $\mathcal{A}$,

it sets $r = r_1^*$ if $r_1^* \neq \perp$ and otherwise it sets r uniformly at random; $\mathcal{B}'$ then sends r to $\mathcal{C}$, receiving back params.

Let (m_1, j_1, r_1^*, r_2^*) and $(m_2, j_2, r_1^*, r_2^{*'})$ be the two queries made by $\mathcal{A}$ resulting in a common commitment value, and let $(j_1, \mathsf{params}, r_1, \mathsf{com}_1, \mathsf{com}'_1, \sigma_1, \sigma_{1'})$ and $(j_2, \mathsf{params}, r_2, \mathsf{com}_1, \mathsf{com}'_2, \sigma_{1''}, \sigma_{2'})$ be the respective signatures resulting from $\mathcal{A}$'s queries. Then $\mathcal{B}'$ sends $(\mathsf{com}_1, m_1, j_1, r_2^*, m_2, j_2, r_2^{*'})$ to its challenger and wins with probability one, contradicting the security of the commitment scheme. Thus, we have that $\Pr[\mathsf{Com\text{-}bind}_{\mathcal{B}', \Pi_{\mathsf{Com}}}(\kappa)] < \mathsf{negl}(\kappa)$, completing the proof. □

Proof of Security

We are now ready to prove the security of our signed-OT extension protocol. Most of the proof complexity is hidden in the proofs of the associated EU-CMPRA signature scheme and commitment scheme. Thus, the signed-OT extension simulator is relatively straightforward, and mostly involves parsing the output of $\mathcal{F}_{\mathbf{signedOT}}^{\Pi}$ and passing the correct values to the adversary. The analysis follows almost exactly that of Asharov et al. [49] and thus we elide most of the details.

Theorem 4.3. *Let* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *be the EU-CMPRA signature scheme presented above. Then the protocol in Figure 4.4 is a secure realization of* $\mathcal{F}_{\mathbf{signedOT}}^{\Pi}$ *in the* $\mathcal{F}_{\mathbf{ot}}$*-hybrid model.*

Proof. We separately consider the case where P_1 is malicious and P_2 is malicious. The case where the parties are either both honest or both malicious is straightforward.

Malicious P_1. Let $\mathcal{A}$ be a PPT adversary corrupting P_1. We construct a simulator $\mathcal{S}$ as follows.

1. The simulator $\mathcal{S}$ acts as an honest P_2 would in Step 1, extracting s from $\mathcal{A}$'s input to $\mathcal{F}_{\mathbf{ot}}$.

2. The simulator $\mathcal{S}$ acts as an honest P_2 would in Steps 2 and 3, using a random choice for r.

3. Let I and $\left(j, \widehat{X}_j^0, \widehat{X}_j^1, \{t_{j,i}\}_{i\in I}, \sigma'_{j,0}, \sigma'_{j,1}\right)$, for $j \in [m]$, be the messages sent by $\mathcal{A}$ in Step 4. If any of these form invalid signatures, $\mathcal{S}$ sends abort to $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ and simulates P_2 aborting, outputting whatever $\mathcal{A}$ outputs.

4. For $j \in [m]$, proceed as follows. The simulator $\mathcal{S}$ extracts $X_j^0 := \widehat{X}_j^0 \oplus H(j, q_j)$ and $X_j^1 := \widehat{X}_j^1 \oplus H(j, q_j \oplus s)$, constructs $\sigma^*_{j,b} := (j, I, k_j, (I, (j, \widehat{X}_j^b, I, \{t_{j,i}\}_{i\in I})), (I, (j, \widehat{X}_j^{1-b}, I, \{t_{j,i}\}_{i\in I})), \sigma'_{j,b}, \sigma'_{j,1-b})$ for $b \in \{0,1\}$, and sends X_j^0, X_j^1, $\sigma^*_{j,0}$, and $\sigma^*_{j,1}$ to $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$.

5. For $j \in [m]$, $\mathcal{S}$ parses $\sigma_{j,b}$ as $(j, I, k_j, (I, (j, \widehat{X}_j^b, I, \{t_{j,i}\}_{i\in I})), (I, (j, \widehat{X}_j^{1-b}, I, \{t_{j,i}\}_{i\in I})), \sigma'_{j,b}, \sigma'_{j,1-b})$, constructs message $\sigma_j := (j, \widehat{X}_j^0, y_j^1, \{t_{j,i}\}_{i\in I}, \sigma'_{j,0}, \sigma'_{j,1})$, and acts as an honest P_2 would when receiving messages I and $\{\sigma_j\}_{j\in[m]}$.

6. The simulator $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

It is easy to see that this protocol perfectly simulates a malicious sender since $\mathcal{S}$ acts exactly as an honest P_2 would (beyond feeding the appropriate messages to $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$).

Malicious P_2. Let $\mathcal{A}$ be a PPT adversary corrupting P_2. We construct a simulator $\mathcal{S}$ as follows.

1. The simulator $\mathcal{S}$ acts as an honest P_1 would in Step 1, extracting matrices T and V through P_1's $\mathcal{F}_{\mathbf{ot}}$ inputs, and thus the values $\{k_j\}_{j\in[m]}$ through the calls to the random oracle.

2. The simulator $\mathcal{S}$ uses the values extracted above to extract selection bits r after receiving the u^i values from $\mathcal{A}$ in Step 2.

3. The simulator $\mathcal{S}$ acts as an honest P_1 would in Step 3.

4. Let I^0 be the indices at which s (generated in Step 1) is zero, and let $I \subseteq I^0$ be a set of size κ. For $j \in [m]$, $\mathcal{S}$ sends $r[j]$, vk, and I to $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back $((r[j], X_j^{r[j]}), \sigma_{j,r[j]})$; $\mathcal{S}$ parses $\sigma_{j,r[j]}$ as $(j, I, r, (I, (j, c_{r[j]}, I, \{t_{j,i}\}_{i\in I})), (I, (j, c_{1-r[j]}, I, \{t_{j,i}\}_{i\in I})), \sigma'_{j,r[j]}, \sigma'_{j,1-r[j]})$.

5. In Step 4, $\mathcal{S}$ sends I and $(j, c_0, c_1, \{t_{j,i'}\}_{i'\in I'}, \sigma'_{j,0}, \sigma'_{j,1})$, for $j \in [m]$, to $\mathcal{A}$.

6. The simulator $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

The analysis is almost exactly that of the malicious receiver proof in the construction of Asharov et al. [49]; we thus give an informal security argument here and refer the reader to the aforementioned work for the full details.

A malicious P_2 has two main attacks: using inconsistent choices of its selection bits r and trying to cheat in the signature creation in Step 4. This latter attack is prevented by the security of our EU-CMPRA signature scheme. The former is prevented by the consistency check in Step 3. Namely, Asharov et al. show that the

consistency check guarantees that: (1) most inputs are consistent with some string r, and (2) the number of inconsistent inputs is small and thus allow P_2 to only learn a small number of bits of s. Thus, for specific choices of ℓ and μ, the probability of a malicious P_2 cheating is negligible. Asharov et al. provide concrete parameters for various settings of the security parameter [49, §3.2]; let ℓ' denote the number of base OTs used in their protocol. Now, in our protocol we set $\ell = \ell' + \kappa$; P_1 leaks κ bits of s when revealing the set I in Step 4, and so is left with ℓ' unknown bits of s. Thus, the security argument presented by Asharov et al. carries over into our setting. □

4.4 Our Protocol

As noted above, the main technical challenge of the PVC model is in the signed-OT construction and model definitions. The AO protocol in the $\mathcal{F}_{\mathbf{signedOT}}^{\Pi}$-hybrid model is relatively straightforward: the natural (but careful) combination of taking a non-halting covert protocol, having the GC generator P_1 sign appropriate messages, and replacing OTs with signed-OTs works. In particular, our signed-OT extension can be naturally modified and used in place of the signed-OT primitive in the AO protocol.

In this section we present a new PVC protocol based on signed-OT extension. Our protocol is similar to the AO protocol in the $\mathcal{F}_{\mathbf{signedOT}}^{\Pi}$-hybrid model, but with applying several simple yet very effective optimizations, resulting in a much lower communication cost.

We present our protocol by starting off with the AO protocol and pointing out the differences. We presented the AO protocol intuition at the beginning of this Chapter; see Figure 4.5 for its formal description. In presenting our changes, we sketch the improvement each of them brings. Thus, we start by reviewing the communication cost of the AO protocol.

Communication cost of the AO protocol. Using state-of-the-art optimizations [29, 53, 54], the size of each GC sent in Step 5 is $2\kappa|G_C|$, where $|G_C|$ is the number of non-XOR gates in circuit C (note that $|G_C| = |G_{C'}|$ for circuit C' generated in Step 1 since the XOR-tree only adds XOR gates to the circuit, which are "free" [29]). Let τ be the field size (in bits), ν the XOR-tree replication factor, λ the GC replication factor, and n the length of the inputs, and assume that each signature is of length τ and the commitment and decommitment values are of length κ. Using the signed-OT instantiations of Asharov and Orlandi [20, Protocols 1 and 2], we get a total communication cost of

$$\begin{aligned}
&\tau(7\nu n + 11) + 2\lambda\kappa\nu n && \text{(Step 4)}\\
&+ \ell(2\kappa|G_C| + \tau) && \text{(Step 5)}\\
&+ 2n\lambda(\kappa + \tau) && \text{(Step 6)}\\
&+ \tau(3 + 2\lambda + 11(\lambda - 1)) + \lambda\kappa(2(n + \nu n)(\lambda - 1) + 2n(\lambda - 1) + n). && \text{(Step 7)}
\end{aligned}$$

As an example, consider the secure computation of AES(m, k), where P_1 inputs message $m \in \{0,1\}^{128}$ and P_2 inputs key $k \in \{0,1\}^{128}$, and suppose we set both

the GC replication factor λ and the XOR-tree replication factor ν to 3, giving a cheating probability of $\epsilon = 1/2$. Letting $\kappa = 128$ and $\tau = 256$, we have a total communication cost of 9.3 Mb (where we assume that the AES circuit has 9,100 non-XOR gates [55]).

Our modifications. We make the following modifications to the AO protocol:

- In Step 6, instead of using a commitment scheme we can use a hash function. This saves on communication in Step 7 as P_1 no longer needs to send the openings $\{o^i_{w_p,b}\}$ to the commitments in the signed-OT, and is secure when treating H as a random oracle since the labels are generated uniformly at random and thus it is infeasible for P_2 to guess the committed values. The total savings are $2n(\lambda-1)\kappa\lambda$ bits; in our example, this saves us 196 Kb.

- In Step 3, we use a random seed to generate the input-wire labels. Namely, for all $j \in [\lambda]$ we compute $s_j \in_R \{0,1\}^\kappa$, and compute the input-wire labels for circuit j as $X^j_{w_1,0}\|X^j_{w_1,1}\|\cdots\|X^j_{w_{n+\nu n},0}\|X^j_{w_{n+\nu n},1} := G(s_j)$, where G is a pseudorandom generator. Now, in the 1-out-of-λ signed-OT in Step 7 we can just send the seeds to the input-wire labels rather than the input-wire labels themselves. The total savings are $2(n+\nu n)(\lambda-1)\lambda\kappa - n(\lambda-1)\lambda\kappa$ bits; in our example, this saves us 688 Kb.

- In Step 5, P_1 generates each $\widehat{C}_j$ from a seed $s^j_{\widehat{C}}$. (This idea was first put forward by Goyal et al. [50].) That is, $s^j_{\widehat{C}}$ specifies the randomness used to construct all wire labels *except* for the input-wire labels which were set in Step 3. Instead of P_1 sending each GC to P_2 in Step 5, P_1 instead sends a

commitment $c^j_{\widehat{C}} := H(\widehat{C}_j)$. Now, in Step 7, P_1 can send the appropriate seeds $\{s^j_{\widehat{C}}\}_{j\in[\lambda]\setminus\{j\}}$ in the jth input of the 1-out-of-λ signed-OT to allow P_2 to check the correctness of the check GCs. We then add an additional step where, if the checks pass, P_1 sends $\widehat{C}_\gamma$ (along with a signature on $\widehat{C}_\gamma$) to P_2, who can check whether $H(\widehat{C}_\gamma) = c^\gamma_{\widehat{C}}$. Note that this does not violate the security conditions required by the PVC model because P_2 catches any cheating of P_1 before the evaluation circuit is sent. If P_1 tries to cheat here, P_2 already has a commitment to the circuit so can detect any cheating. The total savings are $(\lambda-1)2\kappa|G_C| - \lambda\tau - \lambda\kappa(\lambda-1)$ bits; in our example, this saves us 4.6 Mb.

Our PVC protocol and its cost. Below we present our optimized protocol. For simplicity, we sign each message in Steps 5 and 6 separately; however, we note that we can group all the messages in a given step into a single signature.

Private inputs: P_1 has input $x_1 \in \{0,1\}^n$; P_2 has input $x_2 \in \{0,1\}^n$.
Common inputs: Security parameter κ; XOR-tree replication factor ν; garbled circuit replication factor λ; circuit $C(\cdot,\cdot)$; hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$; pseudorandom generator $G : \{0,1\}^\kappa \to \{0,1\}^{2(n+\nu n)\kappa}$; ideal functionalities $\mathcal{F}^\Pi_{\mathbf{signedOT}}$ and $\binom{\lambda}{1}$-$\mathcal{F}^\Pi_{\mathbf{signedOT}}$ for EU-CMPRA signature scheme Π.

1. P_1 and P_2 define a new circuit $C'(x_1, x_2^1, \ldots, x_2^\nu) = C(x_1, \bigoplus_{i\in[\nu]} x_2^i)$. Let $w_1, \ldots, w_n$ denote the input wires of x_1 and let $w_{n+(i-1)\nu}, \ldots, w_{n+i\nu}$ denote the input wires of x_2^i.
2. For $i \in [\nu-1]$, P_2 chooses $x_2^i \in_R \{0,1\}^n$ and sets $x_2^\nu := (\bigoplus_{i\in[\nu-1]} x_2^i) \oplus x_2$.
3. For $j \in [\lambda]$, P_1 chooses $s_j \in_R \{0,1\}^\kappa$ and computes $X^j_{w_1,0}\|X^j_{w_1,1}\|\cdots\|X^j_{w_{n+\nu n},0}\|X^j_{w_{n+\nu n},1} := G(s_j)$.
4. P_1 and P_2 run $\mathcal{F}^\Pi_{\mathbf{signedOT}}$, where in the ith execution P_1 acts as the sender with input $(X^1_{w_{n+i},0}\|\cdots\|X^\lambda_{w_{n+i},0}, X^1_{w_{n+i},1}\|\cdots\|X^\lambda_{w_{n+i},1})$ and P_2 acts as the receiver with input $x_2^{\lceil i/n\rceil}[i \bmod \nu]$. If P_i's output is abort, it outputs abort.
5. For $j \in [\lambda]$, P_1 computes $s^j_{\widehat{C}} \in_R \{0,1\}^\kappa$ and uses $s^j_{\widehat{C}}$ as the randomness used to generate garbled circuit $\widehat{C}_j$, where for $i \in [n+\nu n]$ the labels for input wire w_i are $X^j_{w_i,0}$ and $X^j_{w_i,1}$. P_1 computes $c^j_{\widehat{C}} := H(GC_j)$ and sends $(c^j_{\widehat{C}}, \mathsf{Sign}(c^j_{\widehat{C}}))$ to P_2, who checks that the signature is valid; if not, P_2 outputs abort.

6. For $i \in [n]$ and $j \in [\lambda]$, P_1 computes $c^j_{w_i,0} := H(X^j_{w_i,0})$ and $c^j_{w_i,1} := H(X^j_{w_i,1})$, and sends $(c_{w_i,b}, \mathsf{Sign}(c_{w_i,b})), (c_{w_i,\bar{b}}, \mathsf{Sign}(c_{w_i,\bar{b}}))$ to P_2, where $b \in_R \{0,1\}$. P_2 checks that the signatures are valid; if not, P_2 outputs abort.
7. P_1 and P_2 run $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ with P_1 as the sender and P_2 as the receiver. P_2 uses $\gamma \in_R [\lambda]$ as its input and P_1 uses $(\{s_i, s^i_{\widehat{C}}\}_{i\in[\lambda]\setminus\{j\}}, \{X^j_{w_i,x_1[i]}\}_{i\in[n]})$ as its jth input. If P_i's output is abort, it outputs abort.
8. P_2 does the following:
   - For $j \in [\lambda]\setminus\{\gamma\}$, $i \in [n]$, and $b \in \{0,1\}$, P_2 checks that $H(X^j_{w_i,b}) = c^j_{w_i,b}$. If not, P_2 sets $\mathsf{key} := \mathsf{InvalidDecommitment}$ and moves to Step 12.
   - For $j \in [\lambda]\setminus\{\gamma\}$, P_2 uses s_j and $s^j_{\widehat{C}}$ received from $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ to check that $\widehat{C}_j$ is a correctly garbled circuit and that $H(\widehat{C}_j) = c^j_{\widehat{C}}$. If not, P_2 sets $\mathsf{key} := \mathsf{InvalidCircuit}$ and moves to Step 12.
   - For $j \in [\lambda]\setminus\{\gamma\}$, P_2 checks that the labels received in $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ match the labels generated by s_j received in Step 7. If not, P_2 sets $\mathsf{key} := \mathsf{SelectiveOTAttack}$ and moves to Step 12.
9. Let $((\gamma, m_\gamma), \sigma)$ be P_2's output of $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$. P_2 sends (γ, σ) to P_1, who checks that the signature is valid and otherwise outputs abort.
10. P_1 sends $(\widehat{C}_\gamma, \mathsf{Sign}(\widehat{C}_\gamma))$ to P_2, who checks that the signature is valid; if not, P_2 outputs abort.
11. P_2 checks that $H(\widehat{C}_\gamma) = c^\gamma_{\widehat{C}}$. If not, P_2 sets $\mathsf{key} := \mathsf{InvalidCircuitHash}$ and moves to Step 12.
12. If any of the above checks fail, P_2 computes $\mathsf{Cert} := \mathsf{Blame}(\mathsf{id}_1, \mathsf{key}, \mathsf{View}_2)$, publishes Cert, and outputs $\mathsf{corrupted}_1$. Otherwise, P_2 uses the labels to compute $C'(x_1, x_2^1, \ldots, x_2^\nu)$ and outputs the result.

The Blame and $\mathsf{Judgment}$ algorithms described below are straightforward.

Blame outputs the relevant parts of the view, including the cheater's signatures:

Input: Cheating identity id, error key key, and view View.
Output: A certificate of cheating $\mathsf{Cert} = (\mathsf{id}, \mathsf{key}, \mathsf{msg})$.

- If $\mathsf{key} = \mathsf{InvalidDecommitment}$, set $\mathsf{msg} := (c, o, X, \sigma, \sigma')$, where (c, o) is the invalid commitment-decommitment pair of label X (i.e., $X \neq \mathsf{Open}(c, o)$), σ is the signature of c obtained in Step 6, and σ' is the signature obtained in the signed-OT in Step 7. Output $(\mathsf{id}, \mathsf{key}, \mathsf{msg})$.
- If $\mathsf{key} = \mathsf{InvalidCircuit}$, set $\mathsf{msg} := (\widehat{C}, \{X\}, \sigma_1, \sigma_2)$, where $\widehat{C}$ is the invalid garbled circuit received in Step 5, $\{X\}$ are the labels received in Step 7, σ_1 is the signature of the invalid garbled circuit received in Step 5, and σ_2 is the signature of the labels received in Step 7. Output $(\mathsf{id}, \mathsf{key}, \mathsf{msg})$.
- If $\mathsf{key} := \mathsf{SelectiveOTAttack}$, set $\mathsf{msg} = (m, \{X\}, \sigma_1, \{\sigma\})$, where m is the bitstring received in the signed-OT in Step 4 and σ_1 is the associated signature, and $\{X\}$ and $\{\sigma\}$ are the labels and associated signatures sent in Step 7. Output $(\mathsf{id}, \mathsf{key}, \mathsf{msg})$.
- If $\mathsf{key} = \mathsf{InvalidCircuitHash}$, set $\mathsf{msg} := (c^\gamma_{\widehat{C}}, \widehat{C}_\gamma, \sigma, \sigma')$, where $(c^\gamma_{\widehat{C}}, \sigma)$ is the commitment to $\widehat{C}_\gamma$ and associated signature sent in Step 5, and $(\widehat{C}_\gamma, \sigma')$ is the circuit and signature

sent in Step 10. Output $(\mathsf{id}, \mathsf{key}, \mathsf{msg})$.

- Otherwise, output $\perp$.

Judgment checks that the signatures output by Blame are valid:

Input: A certificate of cheating $\mathsf{Cert} = (\mathsf{id}, \mathsf{key}, \mathsf{msg})$.
Output: The cheating identity id, or $\perp$.

- If $\mathsf{key} = \mathsf{InvalidDecommitment}$, parse msg as $(c, o, X, \sigma, \sigma')$, and check that $X \neq \mathsf{Open}(c, o)$, σ is a valid and appropriate signature of c signed by id, and σ' is a valid and appropriate signature containing o and signed by id. If so, output id; otherwise output $\perp$.
- If $\mathsf{key} = \mathsf{InvalidCircuit}$, parse msg as $(\widehat{C}, \{X\}, \sigma_1, \sigma_2)$, and check that $\widehat{C}$ is indeed an invalid garbled circuit using input-wire labels $\{X\}$, and σ_1, σ_2 are valid and appropriate signatures signed by id. If so, output id; otherwise output $\perp$.
- If $\mathsf{key} = \mathsf{SelectiveOTAttack}$, parse msg as $(m, \{X\}, \sigma_1, \{\sigma\})$, check that the signatures are valid, and check that there is indeed a mismatch between the labels in m and $\{X\}$. If so, output id; otherwise output $\perp$.
- If $\mathsf{key} = \mathsf{InvalidCircuitHash}$, parse msg as $(c, \widehat{C}, \sigma, \sigma')$, check that the signatures are valid, and check that $H(\widehat{C}) \neq c$. If so, output id; otherwise output $\perp$.
- Otherwise, output $\perp$.

Theorem 4.4. *Let $\lambda \in \mathsf{poly}(\kappa)$ and $\nu \in \mathsf{poly}(\kappa)$ be parameters to the protocol, and set $\epsilon := (1 - 1/\lambda)(1 - 2^{-\nu+1})$. Let f be a polynomial sized function, let H be a random oracle, let $\mathcal{F}_{\mathbf{signedOT}}^{\Pi}$ and $\binom{\lambda}{1}$-$\mathcal{F}_{\mathbf{signedOT}}^{\Pi}$ be the $\binom{2}{1}$-signed-OT and $\binom{\lambda}{1}$-signed-OT ideal functionalities, respectively, where Π is an EU-CMPRA signature scheme. Then the protocol above securely computes f in the presence of (1) an ϵ-PVC adversary corrupting P_1 and (2) a malicious adversary corrupting P_2.*

Proof. The proof closely follows that of Aumann and Lindell [19, §6.2]. Let $\mathcal{S}_{\widehat{C}}(1^\kappa, y, \Phi(C))$ be a garbled circuit simulator, which takes as input the security parameter 1^κ, an output bitstring y, and circuit leakage $\Phi(C)$, and outputs a garbled circuit $\widehat{C}$ which is indistinguishable from a correctly garbled circuit with output y [56]. We use $\mathcal{S}_{\widehat{C}}$ in the proof for a corrupted P_2 below.

Clearly, the protocol is non-halting by inspection: an honest party only outputs

corrupted if it detects deviation from the protocol by the other party; this only happens if the other party is malicious.

The rest of the proof involves four steps. We first demonstrate a simulator for a corrupted P_2 and prove that this simulator produces a transcript indistinguishable from an adversary running the real protocol. We then proceed to show a simulator for a corrupted P_1. We then prove the accountability and defamation-free properties required by the PVC security model.

P_2 is corrupted. Let $\mathcal{A}$ be a PPT malicious adversary corrupting P_2. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ acts like P_1 up through Step 3.
2. In Step 4, $\mathcal{S}$ receives $\mathcal{A}$'s inputs to $\mathcal{F}^{\Pi}_{\textbf{signedOT}}$ and proceeds as follows:
   (a) If $\mathcal{A}$'s input is abort, then $\mathcal{S}$ sends abort to the trusted party and simulates P_1 aborting, outputting whatever $\mathcal{A}$ outputs.
   (b) If the input is a bit b, then $\mathcal{S}$ sends $\mathcal{A}$ the appropriate labels generated in Step 3.
3. $\mathcal{S}$ constructs x_2 based on $\mathcal{A}$'s inputs to $\mathcal{F}^{\Pi}_{\textbf{signedOT}}$ extracted above and sends x_2 to the trusted party, receiving back output y_2.
4. $\mathcal{S}$ chooses $\rho \in_R [\lambda]$. For $j \in [\lambda] \backslash \{\rho\}$, $\mathcal{S}$ acts like P_1 in Step 5. For $j = \rho$, $\mathcal{S}$ computes $\widehat{C}_\rho \leftarrow \mathcal{S}_{\widehat{C}}(1^\kappa, y_2, \phi(C))$. It then computes $c := H(\widehat{C}_\rho)$ and sends $(c, \mathsf{Sign}_{P_1}(c))$ to $\mathcal{A}$.
5. $\mathcal{S}$ acts as P_1 in Step 6.

6. In Step 7, $\mathcal{S}$ receives $\mathcal{A}$'s input to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ and proceeds as follows:

   - If $\mathcal{A}$'s input is abort, then $\mathcal{S}$ sends abort to the trusted party and simulates P_1 aborting, outputting whatever $\mathcal{A}$ outputs.
   - If the input is a choice bit γ, $\mathcal{S}$ does the following. If $\gamma \neq \rho$, $\mathcal{S}$ rewinds to Step 4 above, unless $\mathcal{S}$ has rewound $\kappa\lambda$ times, it which case it outputs fail and halts. Otherwise, $\mathcal{S}$ inputs $(\{s_i, s^j_{\widehat{C}}\}_{i\in[\lambda]\setminus\{\rho\}}, \{X^j_{w_i,r[i]}\}_{i\in[n]})$ as the jth input to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, and then proceeds as an honest P_1 would.

7. $\mathcal{S}$ acts like P_1 for the rest of the protocol, outputting whatever $\mathcal{A}$ outputs.

The proof that $\mathcal{S}$ correctly simulates a malicious P_2 follows closely to the proof by Aumann and Lindell [19] and thus we do not repeat it here.

P_1 is corrupted. Let $\mathcal{A}$ be a PPT covert adversary corrupting P_1. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ acts as P_2 up through Step 3.
2. In Step 4, $\mathcal{S}$ receives $\mathcal{A}$'s inputs to $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ and proceeds as follows:
   (a) If $\mathcal{A}$ inputs abort in any iteration, $\mathcal{S}$ sends abort to the trusted party and simulates P_2 aborting, outputting whatever $\mathcal{A}$ outputs.
   (b) Otherwise, $\mathcal{S}$ parses the inputs as mn tuples where the i tuple is

$$(X^1_{w_{n+i},0}\| \cdots \|X^\lambda_{w_{n+i},0}, X^1_{w_{n+i},1}\| \cdots \|X^\lambda_{w_{n+i},1}).$$

3. $\mathcal{S}$ acts as P_2 through Step 6.

4. In Step 7, $\mathcal{S}$ receives $\mathcal{A}$'s input to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ and proceeds as follows:

   (a) If $\mathcal{A}$ inputs abort, $\mathcal{S}$ sends abort to the trusted party and simulates P_2 aborting, outputting whatever $\mathcal{A}$ outputs.

   (b) Otherwise, $\mathcal{S}$ parses the input as λ tuples, where the jth tuple is constructed as

$$\left(\{s_i, s^j_{\widehat{C}}\}_{i\in[\lambda]\setminus\{j\}}, \{X^j_{w_i,x_1[i]}\}_{i\in[n]}\right).$$

5. For $\gamma \in [\lambda]$, $\mathcal{S}$ sends γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back

$$\left((\gamma, \{s_i, s^\gamma_{\widehat{C}}\}_{i\in[\lambda]\setminus\{\gamma\}}, \{X^\gamma_{w_i,x_1[i]}\}_{i\in[n]}), \sigma\right).$$

   If σ is not a valid signature, $\mathcal{S}$ aborts as an honest P_2 would, outputting whatever $\mathcal{A}$ outputs. Otherwise, $\mathcal{S}$ rewinds to before it sent γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$. At this stage, $\mathcal{S}$ has (possibly invalid) openings of all circuits as well as (possibly invalid) labels associated with $\mathcal{A}$'s input. There exist four cases to consider. We follow similar terminology to that of Aumann and Lindell [19, §6.2]. We call a *legitimate circuit* one that can be correctly opened; an *illegitimate circuit* is one that cannot be correctly opened. An *inconsistent label* is one that differs from the label committed to by P_1. An *inconsistent wire* is a wire such that for some garbled circuit either the 0-label or the 1-label is inconsistent. Finally, a *totally inconsistent input* is one where all of the wires associated with the share of that input are inconsistent.

   (a) There exists an illegitimate circuit. Let $\widehat{C}_{j_0}$ be the first such circuit. $\mathcal{S}$

sends cheat_1 to the trusted party. There are two cases to consider.

i. $\mathcal{S}$ receives $\mathsf{corrupted}_1$ from the trusted party. Then it chooses $\gamma \neq j_0$ uniformly at random, and inputs γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back the appropriate output. $\mathcal{S}$ then simulates P_2 aborting due to the detected cheating, outputting whatever $\mathcal{A}$ outputs.

ii. $\mathcal{S}$ receives $\mathsf{undetected}$ and P_2's input x_2 from the trusted party. With probability $p = \frac{1}{\lambda(1-\epsilon)}$, $\mathcal{S}$ chooses $\gamma = j_0$ and with probability $1 - p$ it chooses $\gamma \neq j_0$ uniformly at random, inputting γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ and receiving back the appropriate output. $\mathcal{S}$ then emulates an honest P_2 with input x_2 for the rest of the protocol execution. Let z be the resulting output. $\mathcal{S}$ sends z to the trusted party and outputs whatever $\mathcal{A}$ outputs.

(b) There exists a totally inconsistent input. Assume without loss of generality that the ith input bit $x_2[i]$ is totally inconsistent and that all the inconsistent labels are 0-labels. $\mathcal{S}$ sends cheat_1 to the trusted party. There are two cases to consider.

i. $\mathcal{S}$ receives $\mathsf{corrupted}_1$ from the trusted party. $\mathcal{S}$ chooses bits for the wires $w_{n+(i-1)\nu+1}, \ldots, w_{n+i\nu-1}$ uniformly at random subject to all wires not being one. Let wire w_k be the first zero wire and let $\widehat{C}_{j_0}$ be the first garbled circuit with inconsistent labels for w_k. $\mathcal{S}$ chooses $\gamma \neq j_0$ uniformly at random and inputs γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back the appropriate output. $\mathcal{S}$ then emulates an honest P_2 aborting

and outputs whatever $\mathcal{A}$ outputs.

ii. $\mathcal{S}$ receives $\mathsf{undetected}$ and P_2's input x_2 from the trusted party. $\mathcal{S}$ sets the shares of the ith input and the OT choice γ as follows:

- With probability $p = 2^{-m+1}/(1-\epsilon)$, $\mathcal{S}$ sets the wires $w_{n+(i-1)\nu+1}$, $\ldots, w_{n+i\nu-1}$ to one and sets $w_{n+i\nu} := x_2[i] \oplus \bigoplus_{t\in[m-1]} w_{n+(i-1)\nu+t}$. $\mathcal{S}$ sets $\gamma \in_R \{0,1\}^\lambda$.
- With probability $1-p$, $\mathcal{S}$ sets the wires $w_{n+(i-1)\nu+1}, \ldots, w_{n+i\nu-1}$ to a uniformly random value subject to all wires not being one, and sets $w_{n+i\nu} := x_2[i] \oplus \bigoplus_{t\in[m-1]} w_{n+(i-1)\nu+t}$. Let w_k be the first wire that is set to zero, and let j_0 be the first circuit such that the label of w_k is inconsistent. $\mathcal{S}$ sets $\gamma := j_0$.

$\mathcal{S}$ inputs γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back the appropriate output. $\mathcal{S}$ then continues by emulating an honest P_2 using the shares chosen above, and outputs whatever $\mathcal{A}$ outputs.

(c) $\mathcal{S}$ reaches this case if all circuits are legitimate and there exist no totally inconsistent inputs. However, there may still be inconsistent wires. $\mathcal{S}$ proceeds as follows. It chooses a random value for each inconsistent wire and checks if the given value corresponds to an inconsistent label. There are two cases to consider.

i. $\mathcal{S}$ chooses bits with inconsistent labels. Let w_k be the first wire with an inconsistent label, and let $\widehat{C}_{j_0}$ be the first circuit with said inconsistent label. $\mathcal{S}$ sends cheat_1 to the trusted party. Again, we

have two cases.

A. $\mathcal{S}$ receives $\mathsf{corrupted}_1$ from the trusted party. It chooses $\gamma \neq j_0$ uniformly at random and inputs γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back the appropriate output. $\mathcal{S}$ then simulates P_2 aborting, outputting whatever $\mathcal{A}$ outputs.

B. $\mathcal{S}$ receives $\mathsf{undetected}$ and x_2 from the trusted party. $\mathcal{S}$ chooses bits for the consistent wires at random subject to the shares equaling $x_2[i]$. With probability $p = \frac{1/\lambda}{1-\epsilon}$ the simulator $\mathcal{S}$ sets $\gamma := j_0$ and with probability $1-p$ the simulator $\mathcal{S}$ chooses $\gamma \neq j_0$ uniformly at random. $\mathcal{S}$ inputs γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back the appropriate output, and continues by emulating an honest P_2 using the shares chosen above, and outputs whatever $\mathcal{A}$ outputs.

ii. $\mathcal{S}$ chooses bits with consistent labels. Thus, the circuits and labels $\mathcal{S}$ receives from $\mathcal{A}$ are equivalent to those sent by an honest P_1, and thus $\mathcal{S}$ proceeds as follows. $\mathcal{S}$ chooses $\gamma \in_R [\lambda]$ and sends γ to $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, receiving back the appropriate output. If the signatures output by $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ are invalid, then $\mathcal{S}$ sends abort to the trusted party and simulates P_2 aborting, outputting whatever $\mathcal{A}$ outputs. Otherwise, if there is any other inconsistency, $\mathcal{S}$ sends $\mathsf{corrupted}_1$ to the trusted party and simulates P_2 aborting, outputting whatever $\mathcal{A}$ outputs.

6. $\mathcal{S}$ acts as P_2 in Steps 9 through 11.

7. $\mathcal{S}$ uses the circuit openings retrieved during the rewinding to open the circuit $\widehat{C}_\gamma$ and extracts $\mathcal{A}$'s input x_1'. $\mathcal{S}$ then sends x_1' to the trusted party, along with the $\mathsf{continue}$ message, and outputs whatever $\mathcal{A}$ outputs.

The proof that $\mathcal{S}$ correctly simulates a covert P_1 follows closely to the proof by Aumann and Lindell [19], and thus we do not repeat it here.

Accountability. Let $\mathcal{A}$ be a PPT covert adversary corrupting party P_1 and fix inputs x_1, x_2 such that $\mathsf{Output}(\mathsf{Exec}_{\pi,\mathcal{A}(z)}(x_1, x_2; 1^\kappa)) = \mathsf{corrupted}_1$. The fact that $\Pr[\mathsf{Judgment}(\mathsf{Cert}) = \mathsf{id}_1] > 1 - \mathsf{negl}(\kappa)$ follows directly from the construction and the Blame and $\mathsf{Judgment}$ algorithms. Namely, at any point that $\mathcal{A}$ is detected cheating, P_2 has proof of such cheating by way of $\mathcal{A}$'s signatures on the messages it sent.

Defamation-free. Let $\mathcal{A}$ be a PPT adversary corrupting P_2 (the case where $\mathcal{A}$ corrupts P_1 is similar). We show that $\Pr[\mathsf{Judgment}(\mathsf{Cert}) = \mathsf{id}_1 : \mathsf{Cert} \leftarrow \mathcal{A}] < \mathsf{negl}(\kappa)$. This follows from the security of the underlying $\mathsf{EU\text{-}CMA}$ signature scheme. Namely, if there exists an adversary that succeeds with non-negligible probability, we can convert this directly into an adversary $\mathcal{B}$ which breaks the signature scheme. We construct $\mathcal{B}$ as follows.

On input verification key vk, $\mathcal{B}$ proceeds by emulating $\mathcal{A}$, playing the role of an honest P_1 with verification key vk and using the signing oracle to compute the required signatures. If $\mathcal{A}$ outputs Cert such that $\mathsf{Judgment}(\mathsf{Cert}) = \mathsf{id}_1$, it must have constructed a signature on some message not queried by P_1. Thus, $\mathcal{B}$ outputs this message and the associated signature, succeeding with the same probability as

$\mathcal{A}$. □

The total communication cost of our optimized protocol is

$$
\begin{aligned}
&\mathsf{Cost}(\text{signed-OT/signed-OT extension}) && \text{(Step 4)}\\
&+ \lambda\kappa + \tau && \text{(Step 5)}\\
&+ 2n\lambda\kappa + \tau && \text{(Step 6)}\\
&+ \tau(3 + 2\lambda + 11(\lambda - 1)) + \lambda(2\kappa(\lambda - 1) + n\kappa) && \text{(Step 7)}\\
&+ \log(\lambda) + \tau && \text{(Step 9)}\\
&+ 2\kappa|G_C| + \tau. && \text{(Step 10)}
\end{aligned}
$$

Using our AES circuit example, we find that the total communication cost is now 2.5 Mb, plus the cost of signed-OT/signed-OT extension. In this particular example, signed-OT requires around 1 Mb and signed-OT extension requires around 1.4 Mb. However, as we show below, as the number of OTs required grows, signed-OT extension quickly outperforms signed-OT, both in communication and computation.

4.5 Evaluation

We now compare our signed-OT extension construction (including optimizations, and in particular, the signature batching optimization) with the signed-OT protocol of Asharov and Orlandi [20], along with a comparison of existing covert and malicious protocols and our PVC protocol using both signed-OT and signed-OT extension. All comparisons are done through calculating the number of bits trans-

ferred and estimated running times based on the relative cost of public key versus symmetric operations. We use a very conservative (low-end) estimate on the public/symmetric speed ratio. We note that this ratio does vary greatly across platforms, being much higher on low power mobile devices, which often employ a weak CPU but have hardware AES support. For such platforms our numbers would be even better.[7]

Recall that τ is the field size (in bits), ν is the XOR-tree replication factor, λ is the GC replication factor, n is the input length, and we assume that each signature is of length τ.

Communication cost. We first look at the *communication cost* of the two protocols. The signed-OT protocol of Asharov and Orlandi [20] is based on the maliciously secure OT protocol of Peikert et al. [48], and inherits similar costs. Namely, the communication cost of executing ℓ OTs each of length n is $(6\ell + 11)\tau$ if $n \leq \tau$, and $(6\ell + 11)\tau + 2n\ell$ if $n > \tau$. Signed-OT requires the additional communication of a signature per OT, adding an additional $\tau\ell$ bits. In the underlying secure computation protocol we have that $n = \lambda\kappa$, where λ is the garbled circuit replication factor. For simplicity, we set $\lambda = 3$ (which along with an XOR-tree replication factor of three equates to a deterrence factor of $\epsilon = 1/2$) and thus $n = 3\kappa$. Thus, the total communication cost of executing t signed-OTs is $\boldsymbol{\tau(7t + 11)}$ bits if $3\kappa \leq \tau$ and $\boldsymbol{\tau(7t + 11) + 6\kappa t}$ bits otherwise.

[7] The code for computing the numbers in the subsequent figures can be found at https://gist.github.com/amaloz/82367afc83ff4c41d6df.

On the other hand, the cost of signed-OT extension for t OTs is

$$
\begin{aligned}
&(6\ell + 11)\tau + 2\ell t && \text{(Step 1)}\\
&+ \ell t && \text{(Step 2)}\\
&+ \mu\ell \log \ell + 4\mu\ell\kappa && \text{(Step 3)}\\
&+ \kappa \log \ell + (n + \kappa)t + \tau. && \text{(Step 4)}
\end{aligned}
$$

Asharov et al. [49, §3.2] present concrete choices of μ and ℓ for various security parameters. However, in our setting we need to increase ℓ by κ bits. Thus, let ℓ' be the particular choice of ℓ specified by Asharov et al.; we set $\ell = \ell' + \kappa$. Thus, for the short security parameter we set $\ell = 133 + 80 = 213$ and $\mu = 3$, and for the long security parameter we set $\ell = 190 + 128 = 318$ and $\mu = 2$. Thus, the total communication cost of executing t signed-OTs when using signed-OT extension is $\boldsymbol{(6\ell + 12)\tau + (3\ell + n + \kappa)t + \mu\ell\log\ell + 4\mu\ell\kappa + \kappa\log\ell}$ bits.

Figure 4.6 presents a comparison of the communication cost of both approaches when executing 1,000 and 10,000 OTs, for various keylength settings and underlying public key cryptosystems. We see improvements from 1.1–10.3×, depending on the number of OTs, the underlying public key cryptosystem, and the size of the security parameter. Note that for a smaller number of OTs (such as 100), signed-OT is more efficient, which makes sense due to the overhead of OT extension and the need to compute the base OTs. However, as the number of OTs grows, we see that signed-OT extension is superior across the board.

Computational cost. We now look at the *computational cost* of the two protocols. Let ξ denote the cost of a public key operation (we assume exponentiations and signing take the same amount of time), and let ζ denote the cost of a symmetric key operation (where we let ζ denote the cost of operating over κ bits; e.g., hashing a 2κ-bit value costs 2ζ). We assume all other operations are "free". This is obviously a very coarse analysis; however, it gives a general idea of the performance characteristics of the two approaches.

The cost of executing ℓ OTs on n-bit messages is $(14\ell + 12)\xi$ if $n \leq \tau$ and $(14\ell + 12)\xi + 2\ell\frac{n}{\kappa}\zeta$ if $n > \tau$. Signed-OT requires an additional $2\ell\xi$ operations (for signing and verifying). We again set $n = 3\kappa$, and thus the cost of executing t signed-OTs is $\mathbf{(16t + 12)\xi}$ if $3\kappa \leq \tau$ and $\mathbf{(16t + 12)\xi + 6t\zeta}$ otherwise.

The cost of our signed-OT extension protocol for t OTs (where we assume $t > \kappa$ and we hash the input prior to signing in Step 4) is

$$\begin{aligned}
&\frac{\ell}{\kappa}t\zeta + (14\ell + 12)\xi + 2\ell\frac{t}{\kappa}\zeta && \text{(Step 1)}\\
&+ 6\ell\mu\frac{t}{\kappa}\zeta && \text{(Step 3)}\\
&+ 2\log\ell\zeta + 2t\frac{\ell + n + \kappa}{\kappa}\zeta + 2\xi. && \text{(Step 4)}
\end{aligned}$$

As above, we set $\ell = 213$ and $\mu = 3$ for the short security parameter, $\ell = 318$ and $\mu = 2$ for the long security parameter, and $n = 3\kappa$. Thus, the cost of executing t signed-OTs is $\mathbf{(14\ell + 14)\xi + ((5 + 6\mu)\frac{\ell}{\kappa} + 8)t\zeta + 2\log\ell\zeta}$.

Figure 4.7 presents a comparison of the computational cost of both approaches when executing 1,000 and 10,000 OTs, for various keylength settings and underlying

public key cryptosystems. Here we see that regardless of the number of OTs and public key cryptosystem used, signed-OT extension is (often much) more efficient, and as the number of OTs increases so does this improvement. For as few as 1,000 OTs we already see a 3.5–5.1× improvement, and for 10,000 OTs we see a 30.9–42.4× improvement.

Comparing covert, PVC, and malicious protocols. We now compare the *computation* cost of our optimized PVC protocol, using both signed-OT and signed-OT extension, with the covert protocol of Goyal et al. [50] and the malicious protocol of Afshar et al. [21], which are the most efficient protocols for their respective security models that we are aware of.

The cost of Goyal et al.'s protocol is $\lambda 10|G_C|\zeta + \lambda 4(\nu n + n)\zeta + \lambda(2\nu n + 2n)\zeta + (\lambda - 1)10|G_C|\zeta + (\lambda - 1)4(\nu n + n)\zeta + (4|G_C| + n + \nu n)\zeta + \mathsf{Cost}(\text{OT extension})$, where we use the malicious OT extension of Asharov et al. [49].[8]

The cost of Afshar et al.'s protocol [21] is $\mathsf{Cost}(\rho \text{ OTs}) + \mathsf{Cost}(\text{OT extension}) + \xi + 4n\xi + \rho(6n\xi + 9n\zeta + 8|G_C|\zeta) + \rho/2(8|G_C|\zeta) + \rho/2(5n\zeta + 2n\xi + 2|G_C|\zeta) + n\xi$.

[8] While one can use the covert OT extension of Asharov et al. [49], this decreases the deterrence factor and thus the GC and/or XOR-tree replication factor must be increased to maintain a deterrence factor of $\epsilon = 1/2$.

The cost of our optimized protocol is

$$
\begin{aligned}
& 2\lambda(\nu n + n)\zeta && \text{(Step 3)} \\
& + \mathsf{Cost}(\lambda n \text{ signed-OTs on } \lambda\kappa\text{-bit inputs}) && \text{(Step 4)} \\
& + 10\lambda|G_C|\zeta + 2(\lambda\zeta + \xi) && \text{(Step 5)} \\
& + 2\lambda n\zeta + 2(2\lambda n\zeta + \xi) && \text{(Step 6)} \\
& + \mathsf{Cost}(\text{1-out-of-}\lambda \text{ signed-OT on } (2(\lambda-1)+n)\kappa\text{-bit inputs}) && \text{(Step 7)} \\
& + (\lambda-1)(2n + 10|G_C| + 2(\nu n + n))\zeta && \text{(Step 8)} \\
& + 2|G_C|\zeta + 2\xi && \text{(Step 10)} \\
& + 2|G_C|\zeta, && \text{(Step 12)}
\end{aligned}
$$

where we assume that all signed values are first hashed. Using the 1-out-of-λ signed-OT protocol of Asharov and Orlandi [20, Protocol 2], we have a cost of $12(\lambda-1)\xi + 2\xi + 4\lambda\xi + 2(\frac{(4\lambda+2)\tau}{\kappa}\zeta + \xi) + 2(\lambda+1)(2(\lambda-1)+n)\zeta + 2\xi$ in Step 7. For the signed-OTs in Step 4 we use the costs computed previously.

Figure 4.8 presents a comparison of the computation cost of our protocol using both signed-OT (Ours^{sOT}) and signed-OT extension ($\text{Ours}^{\text{sOT-ext}}$), as well as comparisons to the Goyal et al. protocol (GMS) and Afshar et al. protocol (AMPR). We fix $\kappa = 128$, $\lambda = \nu = 3$ (giving a deterrence factor of $\epsilon = 1/2$), and assume the use of elliptic curve cryptography (and thus $\tau = 256$). We expect public key operations to take between 125–1250× more than symmetric key operations, depending on implementation details, whether one uses AES-NI, etc. This range is a

very conservative estimate using the Crypto++ benchmark [59], experiments using OpenSSL, and estimated ratios of running times between finite field and elliptic curve cryptography [57].

When comparing against GMS, we find that Ours$^{\text{sOT-ext}}$ is slightly more expensive, due almost entirely to the larger number of base OTs in the signed-OT extension. We note that in practice, however, a deterrence factor of 1/2 may not be sufficient for a covert protocol but may indeed be sufficient for a PVC protocol, due to the latter's ability to "name-and-shame" the perpetrator. When increasing the deterrence factor for the covert protocol to $\epsilon \approx .9$, the cost ratios favor Ours$^{\text{sOT-ext}}$. For example, for 16×16 matrix multiplication, the ratio becomes 3.60–3.53× (versus 1.00–0.98×), depending on the cost of public key operations.

Comparing Ours$^{\text{sOT-ext}}$ with Ours$^{\text{sOT}}$, we find that the former is 1.0–86.7× more efficient, depending largely on the characteristics of the underlying circuit. For circuits with a large number of inputs but a relatively small number of gates (e.g., 16384-bit Comp., Hamming 16000, and 1024-bit Sum) this difference is greatest, which makes sense, as the cost of the OT operations dominates. The circuits for which the ratio is around 1.0 (e.g., 1024-bit RSA) are those that have a huge number of gates compared to the number of inputs, and thus the cost of processing the GC far outweighs the cost of signed-OT/signed-OT extension.

Finally, comparing Ours$^{\text{sOT-ext}}$ with AMPR, the former is 9.6–567.2× more efficient, again depending in a large part on the characteristics of the circuit. For example, for the Hamming 16000 circuit, we get an improvement of 67.4–399.7×. These results demonstrate that for settings where public shaming is enough of a

deterrent from cheating, Ours$^{\text{sOT-ext}}$ may present a better security/efficiency trade-off than existing malicious protocols.

P_1**'s inputs:** Messages $\{(X_j^0, X_j^1)\}_{j\in[m]}$ where $X_j^0, X_j^1 \in \{0,1\}^n$; signing key sk.
P_2**'s inputs:** Selection bit vector $r \in \{0,1\}^m$; verification key vk.
Common inputs: Security parameter κ; statistical security parameter ρ; number of base OTs ℓ; number of check functions μ; random oracle $G : \{0,1\}^\kappa \to \{0,1\}^\ell$; random oracle $H : \mathbb{N} \times \{0,1\}^\ell \to \{0,1\}^n$; random oracle $H' : \{0,1\}^m \to \{0,1\}^\kappa$; $\mathsf{EU\text{-}CMA}$ signature scheme $\Pi = (\mathsf{KeyGen}', \mathsf{Sign}', \mathsf{Verify}')$; ideal functionality $\mathcal{F}_{\mathbf{ot}}$.

1. Initial OT Phase:
   - P_1 computes $s \in \{0,1\}^\ell$ as follows. Let I be a set of indices, where $|I| = \kappa$. For $i \in I$, P_1 sets $s_i = 0$. Then, P_1 fills the remaining bits at random.
   - For $j \in [m]$, P_2 chooses $k_j \in_R \{0,1\}^\kappa$ and sets $t_j := G(k_j)$.
   - Let T be an $m \times \ell$ matrix, where the jth row is t_j and the ith column is t^i. Let V be an $m \times \ell$ matrix, where the jth row is v_j and the ith column is v^i. P_1 and P_2 run $\mathcal{F}_{\mathbf{ot}}$ ℓ times in parallel, where P_1 acts as the *receiver* with input s_i and P_2 acts as the *sender* with input (t^i, v^i).
2. OT Extension Phase (Part I):
   - For $i \in [\ell]$, P_2 sets $u^i := t^i \oplus v^i \oplus r$, and sends u^i to P_1.
3. Consistency check of r:
   - For $i \in [\mu]$, P_1 chooses function $\phi_i : [\ell] \to [\ell]$ uniformly at random, and sends ϕ_i to P_2.
   - For $\alpha \in [\ell]$, $i \in [\mu]$, let $\beta := \phi_i(\alpha)$. P_2 computes $h_{\alpha,\beta}^{b,b'} := H'(w_b^\alpha \oplus w_{b'}^\beta)$ for $b \in \{0,1\}, b' \in \{0,1\}$, where $w_0^\alpha = t^\alpha$ and $w_1^\alpha = v^\alpha$. P_2 sends $\{h_{\alpha,\beta}^{b,b'}\}_{b\in\{0,1\},b'\in\{0,1\}}$ to P_1.
   - For $\alpha \in [\ell]$, $i \in [\mu]$, P_1 defines $\beta := \phi_i(\alpha)$ and checks that $h_{\alpha,\beta}^{s_\alpha,s_\beta} = H'(w_{s_\alpha}^\alpha \oplus w_{s_\beta}^\beta)$, $h_{\alpha,\beta}^{\bar{s_\alpha},\bar{s_\beta}} = H'(w_{s_\alpha}^\alpha \oplus w_{s_\beta}^\beta \oplus u^\alpha \oplus u^\beta)$, and $u^\alpha \neq u^\beta$. If any check fails, P_1 outputs abort.
4. OT Extension Phase (Part II):
   - Let Q be the $m \times \ell$ matrix where each column $q^i = (s_i \cdot (u^i \oplus v^i)) \oplus ((1 - s_i) \cdot t^i)$. Note that $q^i = (s_i \cdot r) \oplus t^i$ and $q_j = (r[j] \cdot s) \oplus t_j$.
   - Let I be the set defined in Step 1, and let $t_{j,i}$ denote the ith bit in row t_j. P_1 sends I to P_2, who checks that $|I| = \kappa$ and otherwise aborts.
   - For $j \in [m]$, P_1 computes $\widehat{X}_j^0 := X_j^0 \oplus H(j, q_j)$ and $\widehat{X}_j^1 := X_j^1 \oplus H(j, q_j \oplus s)$ and signatures $\sigma_j' \leftarrow \mathsf{Sign}'_{\mathsf{sk}}\left((I, j, \widehat{X}_j^0, \{t_{j,i}\}_{i\in I})\right)$, and $\sigma_j'' \leftarrow \mathsf{Sign}'_{\mathsf{sk}}\left((I, j, \widehat{X}_j^1, \{t_{j,i}\}_{i\in I})\right)$, and sends $\left(j, \widehat{X}_j^0, \widehat{X}_j^1, \{t_{j,i}\}_{i\in I}, \sigma_j', \sigma_j''\right)$ to P_2.
   - For $j \in [m]$, P_2 computes $X_j := \widehat{X}_j^{r[j]} \oplus H(j, t_j)$.
5. Output:
   - P_1 outputs $\perp$; P_2 outputs $\left\{X_j, \left(j, r[j], k_j, I, \widehat{X}_j^0, \widehat{X}_j^1, \{t_{j,i}\}_{i\in I}, \sigma_j', \sigma_j''\right)\right\}_{j\in[m]}$.

Figure 4.4: Signed-OT extension, based on the OT extension protocol of Asharov et al. [49].

Private inputs: P_1 has input $x_1 \in \{0,1\}^n$ and P_2 has input $x_2 \in \{0,1\}^n$.
Common inputs: Security parameter κ; XOR-tree replication factor ν; garbled circuit replication factor λ; circuit $C(\cdot,\cdot)$; commitment scheme $\Pi_{\mathsf{Com}} = (\mathsf{Com}, \mathsf{Open})$; ideal functionalities $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ and $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ for EU-CMPRA signature scheme Π.

1. P_1 and P_2 define a new circuit $C'(x_1, x_2^1, \ldots, x_2^\nu) = C(x_1, \bigoplus_{i\in[\nu]} x_2^i)$. Let $w_1, \ldots, w_n$ denote the input wires of x_1 and let $w_{n+(i-1)\nu}, \ldots, w_{n+i\nu}$ denote the input wires of x_2^i.
2. For $i \in [\nu - 1]$, P_2 chooses $x_2^i \in_R \{0,1\}^n$. P_2 sets $x_2^\nu := (\bigoplus_{i\in[\nu-1]} x_2^i) \oplus x_2$.
3. For $j \in [\lambda]$, $i \in [n + \nu n]$, and $b \in \{0,1\}$, P_1 chooses $X^j_{w_{n+i},b} \in_R \{0,1\}^\kappa$.
4. P_1 and P_2 run $\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$, where in the ith execution P_1 acts as the sender with input $(X^1_{w_{n+i},0} \| \ldots \| X^\lambda_{w_{n+i},0}, X^1_{w_{n+i},1} \| \ldots \| X^\lambda_{w_{n+i},1})$ and P_2 acts as the receiver with input $x_2^{\lceil i/n \rceil}[i \bmod \nu]$. If P_2's output is abort, it outputs abort.
5. For $j \in [\lambda]$, P_1 constructs garbled circuit $\widehat{C}_j$ of circuit C', where for $i \in [n + \nu n]$ the labels for input wire w_i are $X^j_{w_i,0}$ and $X^j_{w_i,1}$. P_1 sends $(\widehat{C}_j, \mathsf{Sign}(\widehat{C}_j))$ to P_2, who checks that the signature is valid; if not, P_2 outputs abort.
6. For $i \in [n]$ and $j \in [\lambda]$, P_1 chooses $b \in_R \{0,1\}$, computes commitments $(c^j_{w_i,0}, o^j_{w_i,0}) \leftarrow \mathsf{Com}(X^j_{w_i,0})$ and $(c^j_{w_i,1}, o^j_{w_i,0}) \leftarrow \mathsf{Com}(X^j_{w_i,1})$, and sends $(c_{w_i,b}, \mathsf{Sign}(c_{w_i,b}))$ and $(c_{w_i,\bar{b}}, \mathsf{Sign}(c_{w_i,\bar{b}}))$ to P_2, who checks that the signatures are valid; if not, P_2 outputs abort.
7. P_1 and P_2 run $\binom{\lambda}{1}$-$\mathcal{F}^{\Pi}_{\mathbf{signedOT}}$ with P_1 as the sender inputting $(\{X^i_{w_p,b}\}_{i\in[\lambda]\setminus\{j\},p\in[n+\nu n],b\in\{0,1\}}, \{o^i_{w_p,b}\}_{i\in[\lambda]\setminus\{j\},p\in[n],b\in\{0,1\}}, \{X^j_{w_i,x_1[i]}\}_{i\in[n]})$ as its jth input and P_2 as the receiver inputting $\gamma \in_R [\lambda]$ as its input; if P_2's output is abort, it outputs abort.
8. P_2 does the following:
   - For $j \in [\lambda]\setminus\{\gamma\}$, $i \in [n]$, and $b \in \{0,1\}$, P_2 checks that $\mathsf{Open}(c^j_{w_i,b}, o^j_{w_i,b}) = X^j_{w_i,b}$. If not, P_2 sets $\mathsf{key} := \mathsf{InvalidDecommitment}$ and moves to Step 9.
   - For $j \in [\lambda]\setminus\{\gamma\}$, P_2 uses the input wire keys received from the signed-OT in Step 7 to check that $\widehat{C}_j$ is a correctly garbled circuit. If not, P_2 sets $\mathsf{key} := \mathsf{InvalidCircuit}$ and moves to Step 9.
   - For $j \in [\lambda]\setminus\{\gamma\}$, P_2 checks that the keys received in the signed-OT in Step 4 match the keys sent by P_1 in Step 7. If not, P_2 sets $\mathsf{key} := \mathsf{SelectiveOTAttack}$ and moves to Step 9.
9. If any of the above checks fail, P_2 computes $\mathsf{Cert} := \mathsf{Blame}(\mathsf{id}_1, \mathsf{key}, \mathsf{View}_2)$, publishes Cert, and outputs $\mathsf{corrupted}_1$. Otherwise, P_2 uses the keys to compute $C'(x_1, x_2^1, \ldots, x_2^\nu)$ and outputs the result.

Figure 4.5: The AO PVC protocol [20, Protocol 3].

	1,000 OTs			10,000 OTs		
Security	sOT	sOT-ext	Improvement	sOT	sOT-ext	Improvement
Short (FFC)	7,179	2,539	2.8×	71,691	11,305	6.3×
Short (ECC)	1,602	1,398	1.1×	16,002	10,164	1.6×
Long (FFC)	21,538	7,694	2.8×	215,074	20,888	10.3×
Long (ECC)	2,563	2,288	1.1×	25,603	15,482	1.7×

Figure 4.6: Communication cost (in kbits) of transferring the input wire labels for P_2 when using signed-OT (sOT) versus signed-OT extension (sOT-ext) for 1,000 and 10,000 OTs.

	1,000 OTs			10,000 OTs		
Security	sOT	sOT-ext	Improvement	sOT	sOT-ext	Improvement
Short (FFC)	16.0	3.1	5.1×	160.0	3.8	42.4×
Short (ECC)	5.3	1.1	4.9×	53.3	1.7	30.9×
Long (FFC)	144.1	40.2	3.6×	1440.1	40.7	35.4×
Long (ECC)	14.4	4.1	3.5×	144.1	4.5	31.9×

Figure 4.7: Computation cost (in millions of "time units") of transferring the input wire labels for P_2 when using signed-OT (sOT) versus signed-OT extension (sOT-ext) for 1,000 and 10,000 OTs. We assume symmetric-key operations take 1 "time unit", FFC (resp., ECC) operations take 1000 (resp., 333) "time units" for the short security parameter, and FFC (resp., ECC) operations take 9000 (resp., 900) "time units" for the long security parameter [57].

f	# inputs	# gates	$\frac{\text{GMS}}{\text{Ours}^{\text{sOT-ext}}}$	$\frac{\text{Ours}^{\text{sOT}}}{\text{Ours}^{\text{sOT-ext}}}$	$\frac{\text{AMPR}}{\text{Ours}^{\text{sOT-ext}}}$
16384-bit Comp.	16,384	32,229	0.85–0.73	17.1–86.7	103.0–533.4
Hamming 16000	16,000	97,175	0.90–0.79	11.0–67.0	67.4–399.7
16×16 Matrix Mult.	8192	4,186,368	1.00–0.98	1.2–3.1	10.8–21.9
1024-bit Sum	1,024	2,977	0.71–0.61	6.7–10.2	41.0–61.5
1024-bit Mult.	1,024	6,371,746	1.00–0.99	1.0–1.2	9.7–10.5
1024-bit RSA	1,024	15,149,856,895	1.00–1.00	1.0–1.0	9.6–9.6

Figure 4.8: Ratio of computation cost of various secure computation protocols with our signed-OT extension construction, using a deterrence factor of 1/2 for the covert and PVC protocols. *GMS* denotes the covert protocol of Goyal et al. [50], $Ours^{sOT}$ denotes the optimized Asharov-Orlandi protocol run using signed-OT, $Ours^{sOT\text{-}ext}$ denotes the same protocol using signed-OT extension, and *AMPR* denotes the protocol of Afshar et al. [21]. We let f denote the function being computed, *# inputs* denote the number of input bits required as input by P_2, and *# gates* denote the number of non-XOR gates in the resulting circuit. All circuit information is taken from the PCF compiler [58, Table 5]. We report each ratio as a range; the first number uses $\xi = 125$ as the cost of public-key operations and the second number uses $\xi = 1250$, where we assume a symmetric-key operation costs $\zeta = 1$.

Chapter 5: The Input Validity Setting

Even with the significant progress in improving the performance of protocols in the malicious setting over the last several years, most practical 2PC research still focuses on the semi-honest setting. We argue that this is due to several reasons. For one, a slowdown of $40\times$ to achieve security 2^{-40} is still significant. Moreover, even a protocol that is secure in the malicious model offers no assurance on its own that the adversarial party uses a "valid" input (for some definition of valid). Finally, in the semi-honest setting parties can rely on (some) *local computation* which can greatly reduce the size of the circuit that needs to be garbled. In contrast, in the malicious setting such local computation cannot (in general) be relied upon because there is no guarantee that an adversary correctly computes said computation. Below, we describe these latter two issues in more detail and describe how they can be addressed (inefficiently) using existing protocols before describing our solution.

Input validity. One inherent limitation of the malicious security model is that a malicious party can choose an arbitrary value as its input. This potentially allows a malicious party to learn a significant amount of information, or violate correctness (at least in an intuitive sense). As an example of the former, consider a shortest-path computation where one party holds a weighted graph, the other holds

a source-destination pair, and both parties learn the length of the shortest path. By manipulating edge weights, the first party can ensure that it learns the source-destination pair of the other party. As an example of the latter, consider computing the average of several temperature readings, where one party uses a temperature of 1000°C.

One possible solution to this input-validity problem is to let the two parties verify that the other party's input is signed by some trusted party, or satisfies some other predicate. However, verifying a signature can require more than *one hundred billion* non-free gates [58]. Recalling that malicious security requires an additional $O(\rho)$ multiplicative overhead due to cut-and-choose, this approach appears impractical, especially if the underlying function to be computed is small.

Local computation. One popular technique to improve efficiency in the semi-honest model is to utilize local computation. Namely, instead of each party submitting their input directly, each party first performs some local computation on their input and submits the result of that local computation as input to some secure computation. (An interactive approach, in which a secure computation is run to generate intermediate values which are further processed by the parties locally before further secure computation is done, can also be used.) Some works have shown that for specific examples this approach improves the running time of (semi-honest) secure computation by orders of magnitude, including private set intersection [60] and edit-distance estimation [61]. One common characteristic shared by these works is that most of the computation is done locally such that the part of the function

requiring secure computation is significantly, and in many cases asymptotically, smaller. However, in the malicious setting, local computation is not beneficial at all, since there is no guarantee that the malicious party provides the correct result of a local computation starting from some input. Thus, all computation must be integrated into the secure-computation protocol itself.

Abstracting the problem. We observe that the two problems mentioned above relate to a common problem where the two parties, holding inputs x and y, respectively, wish to compute a function of the form

$$f(x, y) := \text{“if } f_1(x) \text{ and } f_2(y) \text{ then } g(x, y) \text{ else } \perp\text{”},$$

where $f_1(\cdot)$ and $f_2(\cdot)$ are (public) predicates on each party's input and $g(\cdot, \cdot)$ is the underlying function the parties would like to compute. Note that this directly captures the input-validity problem, in that the predicate functions could check validity however the parties choose to define it. Likewise, for the local-computation problem we can have the predicates verify that the local computation was done correctly—something which can often be more efficient than re-doing the computation.

As $f(\cdot, \cdot)$ is a two-party function, we can compute it securely using any existing malicious 2PC protocol. We refer to this as the “generic solution.” In this work we show how it is possible to do *much* better by using cut-and-choose only on $g(\cdot, \cdot)$. For the predicate checks, we use the zero-knowledge-using-garbled-circuits (ZKGC) approach of Jawurek et al. [62] to evaluate $f_1(\cdot)$ and $f_2(\cdot)$. This allows us to garble $f_1(\cdot)$ and $f_2(\cdot)$ only *once*, while garbling only $g(\cdot, \cdot)$ a total of ρ times. Combining

these protocols in a naive way, however, does not guarantee that a malicious party uses consistent inputs between the predicate circuits (namely $f_1(\cdot)$ and $f_2(\cdot)$) and the computation circuit (namely $g(\cdot,\cdot)$). In order to solve this consistency problem efficiently, we extend the protocol of Afshar et al. [21], the best known cut-and-choose-based 2PC protocol we are aware of, to support secure composition with the ZKGC approach. See details below.

To understand the performance gains of our protocol versus the generic solution, we present a detailed cost analysis, comparing the computation and communication costs of our protocol with that of Afshar et al. We obtain savings of up to $\approx 80\times$ in communication and $\approx 56\times$ in computation for many realistic examples. We refer to Section 5.4 for more details.

Building Blocks

Because our protocol relies heavily on the existing works of Jawurek et al. [62] and Afshar et al. [21], we briefly recap how these constructions work.

Efficient zero-knowledge using garbled circuits [62]. In a zero-knowledge proof-of-knowledge (ZKPoK), two parties, a *prover* and a *verifier*, have some common predicate $f(\cdot)$, and the prover would like to demonstrate to the verifier that it knows some *witness* w such that $f(w) = 1$, without revealing w to the verifier. Such a protocol is a particular case of 2PC, so any generic secure-computation protocol, with malicious security, could be used. Jawurek et al. [62] showed, however, that one can do much better, and devised a ZKPoK protocol with essentially the same cost as a semi-honest garbled-circuit protocol for the predicate f.

The basic idea is as follows. The verifier sends a garbling of $f(\cdot)$ to the prover, who evaluates it using the input-wire labels it receives through OT, learning an output-wire label Z. The prover commits to this value, and then asks the verifier to open the garbled circuit so the prover can verify that the garbled circuit sent by the verifier indeed corresponds to the correct predicate $f(\cdot)$. If this is the case, the prover decommits to reveal Z to the verifier; if Z is the output-wire label corresponding to '1' then the verifier learns that the prover supplied a valid witness. Security of the OT implies that the prover's input w is hidden from the verifier; security of the garbled circuit implies that the prover cannot learn the correct output-wire label Z if its witness does not satisfy the predicate.

Efficient malicious two-party computation [21]. Afshar et al. [21] propose an optimized variant of Lindell's "fast cut-and-choose with cheating punishment" protocol [10], which garbles ρ circuits for $2^{-\rho}$ statistical security (cf. Chapter 2).[1] Recall that the basic idea with Lindell's protocol is that if any of the evaluation circuits lead to inconsistent outputs, these inconsistencies can be used to recover the circuit generator's input x, allowing the evaluator to locally compute $f(x, y)$. Lindell's protocol requires running an additional secure computation protocol for the "cheating punishment" phase; Afshar et al. show how to remove this (computationally expensive) step. Their idea is as follows.

The circuit generator P_1 begins by committing to its input bits using a specific ElGamal commitment scheme. Namely, for all $i \in [n_1]$, where n_1 is P_1's input length,

[1] While Afshar et al. also show how their protocol can be used to provide *non-interactive secure computation*, we do not utilize this property in our setting.

P_1 computes $\mathsf{EGCommit}_h(x[i]; r) = (g^r, h^r g^{x[i]})$, where $h = g^w$ for some secret value w known to P_1, and sends these commitments to P_2. Note that if the evaluator P_2 learns w it can break the commitments and thus learn x. Party P_1 then constructs garbled circuits such that if P_2 learns both output-wire labels in an evaluation circuit, then it learns w. Thus, if P_1 tries to cheat, P_2 can recover w and thus learn P_1's input, allowing P_2 to compute $f(x, y)$ locally. Party P_1's input consistency is enforced by having P_1 prove that the input-wire labels it provides for the evaluation circuits are commitments to the bits P_1 initially committed to.

Our Contribution

In this work, we combine the works of Jawurek et al. [62] and Afshar et al. [21] to handle functions with predicate checks on each party's input. The parties first prove (in zero-knowledge) that their inputs satisfy the requisite predicate, and if so, the parties compute the underlying function. The main technical difficulty is devising a mechanism for tying together the inputs of the predicate checks with the inputs to the underlying computation function. Namely, we need to enforce that, for example, the input P_1 supplies to $f_1(\cdot)$ is the *same* input used when computing $g(\cdot, \cdot)$. We describe how we do this for each party in turn.

Enforcing consistency on P_1's input. Recall that in the protocol of Afshar et al., P_1 commits (using a specific ElGamal commitment scheme) to each individual input bit of its input x at the beginning of the protocol, and then proves in zero-knowledge that the input-wire labels it provides to the evaluation circuits are commitments to those same input bits. Thus, in order to support input consistency across $f_1(\cdot)$ and

$g(\cdot,\cdot)$ we need to somehow enforce that P_1's inputs to $f_1(\cdot)$ are the same as those it committed to initially. However, f_1 is garbled by P_2, and thus it is not immediate how to enforce this without allowing P_1 to equivocate on its input. We solve this by using a specific ElGamal-based OT protocol which works with the ElGamal commitment scheme used by P_1. Namely, the ElGamal commitments to $x[i]$ sent by P_1 are used to construct P_2's OT messages encoding the input-wire labels to the garbling of $f_1(\cdot)$; P_1 can only recover those wire labels associated with the bit values it committed to.

In more detail, recall that P_1 commits to its input bits using the commitment scheme $(A, B) = (g^r, h^r g^b) := \mathsf{EGCommit}_h(b; r)$. Letting s and t be random elements in $\mathbb{Z}_p$, note that if $b = 0$ then the tuple $(g, g^r, g^s h^t, A^s B^t)$ is a Diffie-Hellman tuple. Likewise, if $b = 1$ then the tuple $(g, g^r, g^s h^t, A^s (B/g)^t)$ is a Diffie-Hellman tuple. Thus, letting (A_i, B_i) be the ElGamal commitment of input bit $x[i]$, P_2 can encode the ith input-wire labels $X_{i,0}, X_{i,1}$ to the garbling of $f_1(\cdot)$ as

$$(\widehat{A}_{i,0}, \widehat{B}_{i,0}) \leftarrow (g^{s_{i,0}} h^{t_{i,0}}, (A_i)^{s_{i,0}} (B_i)^{t_{i,0}} \cdot X_{i,0})$$
$$(\widehat{A}_{i,1}, \widehat{B}_{i,1}) \leftarrow (g^{s_{i,1}} h^{t_{i,1}}, (A_i)^{s_{i,1}} (B_i/g)^{t_{i,1}} \cdot X_{i,1}),$$

for random $s_{i,0}, t_{i,0}, s_{i,1}, t_{i,1}$, and send $\widehat{A}_{i,0}, \widehat{B}_{i,0}, \widehat{A}_{i,1}, \widehat{B}_{i,1}$ to P_1, who can only recover one of the two wire labels based on which value $x[i]$ it committed to.

Note that this OT protocol is not maliciously secure in the sense that a simulator cannot extract P_2's inputs. This is okay in our setting, as the garbling of $f_1(\cdot)$ is fully opened later in the protocol, and thus we can recover the wire labels in that

step.

Another issue is that when proving security for a malicious P_1, the simulator needs to be able to extract P_1's input x. In the protocol of Afshar et al., this extraction happens when P_1 sends the garbled circuits to P_2: the simulator can learn w and thus break the commitments sent by P_1. However, in our protocol we need to extract x *earlier*, and in particular, in the phase where we check whether $f_1(x) = 1$. We do this by having P_1 prove in zero-knowledge that it knows the exponent of h used in the commitments. When simulating, we can extract this exponent and break the commitments, learning P_1's input.

Enforcing consistency on P_2's input. In this step we need to enforce that P_2's input y is consistent between $f_2(\cdot)$ and $g(\cdot,\cdot)$. Note that P_1 garbles both of these functions: $f_2(\cdot)$ is garbled once and $g(\cdot,\cdot)$ is garbled ρ times, with around half being used as evaluation circuits and the other half being checked. Thus, we can use OT to enforce consistency by having P_1 input as the sender P_2's input-wire labels for f_2 and the input-wire labels for the ρ garblings of g. However, we have the added challenge that P_1 needs to open various pairs of messages to (1) check that f_2 is correctly garbled and (2) check that the check circuits of g are correctly garbled.

We handle these two issues as follows. All the input-wire labels for each circuit are generated from some seed: for the f_2 garbling we use seed_0 and for the jth garbling of g we use seed_j. Now, when opening f_2 party P_1 can send seed_0 to P_2, who can check correctness, and likewise for the jth garbling of g. However, this approach as described has a *selective-failure attack* in that P_1 can use, for example,

an invalid 0-bit input-wire label as input to the OT for the ith input. If P_2's ith input is zero it aborts (since the input-wire label it receives is invalid) and otherwise it succeeds, allowing P_1 to learn the ith bit of P_2's input. This can be fixed by applying the XOR-tree approach of Lindell and Pinkas [8] (cf. Chapter 2).

5.1 Preliminaries

Besides the notation introduced in Chapter 2, we let n_1 denote the length of P_1's input, n_2 the length of P_2's input, and n_3 the output length.

Two-party functionality for enforcing predicate checks. We consider a *reactive* two-party functionality $\mathcal{F}_{\mathsf{2pc}}$ of a certain form, where each party's input must satisfy some predicate function before some underlying function (computed on both parties' inputs) is run. In case a party's input does not satisfy the necessary predicate, the functionality outputs $\perp$ to the other party.

The functionality begins by taking either an input x or $\perp$ from P_1; if the functionality receives x such that $f_1(x) = 1$ then it sends an ok message to P_2 and waits for either an input y or $\perp$ from P_2, and otherwise it halts. Likewise, if the functionality receives y such that $f_2(y) = 1$ from P_2 then it sends an ok message to P_1 and otherwise it halts. If both parties send valid inputs to the functionality, then it waits for a $\mathsf{continue}$ message from P_1, at which point it outputs $g(x, y)$ to P_2 and halts. See Figure 5.1 for the formal description.

$\mathcal{F}_{\mathsf{2pc}}$ is slightly weaker than the non-reactive functionality $\mathcal{F}'_{\mathsf{2pc}}$ that accepts inputs x and y from the two parties, and then returns $\perp$ to both parties if either

Functionality $\mathcal{F}_{\mathsf{2pc}}$

Private inputs: P_1 has input $x \in \{0,1\}^{n_1}$ and P_2 has input $y \in \{0,1\}^{n_2}$.
Common input: Circuit $C_0 : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to \{0,1\}^{n_3}$, where

$$C_0(x,y) := \text{if } f_1(x) \text{ and } f_2(y) \text{ then } g(x,y) \text{ else } \perp.$$

1. Upon receiving either (input, x) or $(\mathsf{input}, \perp)$ from P_1, proceed as follows:
   - If x was received and $f_1(x) = 1$, then send $(\mathsf{received}, \mathsf{ok})$ to P_2 and continue.
   - If either $\perp$ was received or $f_1(x) = 0$, send $(\mathsf{received}, \perp)$ to P_2 and halt.
2. Upon receiving either (input, y) or $(\mathsf{input}, \perp)$ from P_2, proceed as follows:
   - If y was received and $f_2(y) = 1$, then send $(\mathsf{received}, \mathsf{ok})$ to P_1 and continue.
   - If $\perp$ was received or $f_2(y) = 0$, send $(\mathsf{received}, \perp)$ to P_1 and halt.
3. Upon receiving either (abort) or $(\mathsf{continue})$ from P_1, proceed as follows:
   - If abort was received, send $(\mathsf{output}, \perp)$ to P_2 and halt.
   - If $\mathsf{continue}$ was received, send $(\mathsf{output}, g(x,y))$ to P_2 and halt.

Figure 5.1: Functionality $\mathcal{F}_{\mathsf{2pc}}$ for two-party secure computation with predicate checks.

$\mathcal{F}_{\mathbf{ot}}$ On receiver input (choose, i, b) and sender input $(\mathsf{transfer}, i, \{m_j^0\}, \{m_j^1\})$, send $(\mathsf{transferred}, i, \{m_j^b\})$ to the receiver.

$\mathcal{F}_{\mathbf{zkpok}}$ On prover input (prove, h, w), if $h := g^w$ then send $(\mathsf{verified}, h)$ to the verifier, and otherwise send $(\mathsf{verified}, \perp)$.

Figure 5.2: Ideal functionalities for oblivious transfer ($\mathcal{F}_{\mathbf{ot}}$) and zero-knowledge proof-of-knowledge ($\mathcal{F}_{\mathbf{zkpok}}$).

$f_1(x) = 0$ or $f_2(y) = 0$, and $g(x,y)$ otherwise. In particular, $\mathcal{F}_{\mathsf{2pc}}$ allows P_2 to learn whether $f_1(x) = 1$ even if $f_2(y) = 0$—something that is not possible when interacting with the non-reactive functionality $\mathcal{F}'_{\mathsf{2pc}}$ just described. In most practical scenarios, however, we expect that an honest P_1 would only ever use an input for which $f_1(x) = 1$, and so "leaking" that information to an attacker is insignificant.

Additional ideal functionalities. We make use of two additional (standard) ideal functionalities for oblivious transfer and zero-knowledge proof-of-knowledge of (h, w) under the relation that $h = g^w$; see Figure 5.2.

5.2 Our Protocol

Our construction carefully combines Jawurek et al.'s ZK-using-garbled-circuits protocol [62] with the maliciously secure 2PC protocol of Afshar et al. [21], where the functions we are interested in are of the form

$$f(x, y) = \text{“if } f_1(x) = 1 \text{ and } f_2(y) = 1 \text{ then } g(x, y) \text{ else } \perp\text{”}.$$

We begin by giving a brief intuition before presenting the full protocol description.

P_1 begins by choosing some $w \in_R \mathbb{Z}_p$ which will act as the trapdoor to P_1's input. P_1 commits to its input bits by computing $\mathsf{EGCommit}_h(x[i], r_i)$, where $h = g^w$, for some randomness r_i, and P_2 uses these commitments to form an OT protocol in order to transfer the wire labels associated with P_1's input for the garbled circuit of f_1. P_1 can evaluate this garbled circuit, learning the output-wire label Z_{f_1}, which it commits to. Now, P_2 can open the garbled circuit, allowing P_1 to check correctness before it decommits to the label it received, allowing P_2 to learn whether $f_1(x) = 1$.

The next step is to check that $f_2(y) = 1$. P_1 and P_2 run OT, where P_2 receives both the appropriate input-wire label for the garbled circuit of f_2 *and* the appropriate input-wire labels for the ρ garbled circuits of g. The input-wire labels for the jth garbling of g are generated using some seed seed_j, and likewise the input-wire labels for f_2 are generated using some seed seed_0. Now, P_2 evaluates the garbled circuit of f_2, learning output-wire label Z_{f_2}, which it commits to. P_1 can now open the garbled circuit by sending seed_0, allowing P_2 to check correctness

before it decommits to the label it received, allowing P_1 to learn whether $f_2(y) = 1$. Note that as written, this protocol is subject to a selective-failure attack in that P_1 can input invalid labels into the OT; however, this is easily prevented using the XOR-tree approach.

Finally, the parties need to compute $g(x, y)$. Note that the input-wire labels of both parties at this point are fixed: P_1 needs to use those input-wire labels corresponding to its commitments at the beginning of the protocol, and P_2 needs to use those input-wire labels corresponding to the labels received in the OT protocol. We enforce that P_1 uses the correct input-wire labels as follows. The input-wire labels for P_1's inputs are derived by computing hashes of the output of $\mathsf{EGCommit}$. Namely, the ith input-wire label $X^b_{j,i}$ for circuit j and bit b is set to $H(\mathsf{EGCommit}_h(b; r^b_{j,i}))$ for some randomness $r^b_{j,i}$, where H is a hash function. In addition, P_1 sends a commitment to the output of $\mathsf{EGCommit}$, and provides decommitments for those values corresponding to its input x, and proves equality between $\mathsf{EGCommit}_h(b; r^b_{j,i})$ and P_1's originally committed inputs $\mathsf{EGCommit}_h(x[i], r_i)$. It does this as follows. Let $(u^b_{j,i}, v^b_{j,i}) := \mathsf{EGCommit}_h(b; r^b_{j,i})$, and let $(A_i, B_i) := \mathsf{EGCommit}_h(x[i], r_i)$. P_2 can check that $(u^b_{j,i}, v^b_{j,i})$ commits to the bit committed by P_1 originally by having P_1 send $r_i - r^b_{j,i}$ and checking whether $g^{r_i - r^b_{j,i}} \cdot u^b_{j,i} = A_i$ and $h^{r_i - r^b_{j,i}} \cdot v^b_{j,i} = B_i$.

The next challenge is to enforce that if P_1 cheats by constructing an invalid garbling of g in one or more of the ρ garbled circuit it produces, then P_2 can recover w, learn P_1's input x, and compute $g(x, y)$ on its own. Here we follow the approach of Afshar et al. [21]. P_1 garbles g a total of ρ times, where the jth garbled circuit uses randomness based on some seed seed_j. In addition, P_1 encrypts information

for checking consistency of values using some key k_j. Now, the parties run OT a total of ρ times, where P_2 learns *either* seed_j or k_j. If it learns seed_j, then P_2 can check correctness of the circuit, aborting on any inconsistency. If it learns k_j, then P_2 can decrypt the information sent by P_1 and use this information to evaluate the garbled circuit and check that evaluation "succeeded". The point is that if two or more evaluation circuits "succeed", then P_2 either learns the appropriate output or there is some inconsistency in the output-wire labels. It can then use this inconsistency and the information it decrypted to learn w, and thus decommit the initial commitments made by P_1 to its input x. Thus, on any inconsistency in output-wire labels, P_2 learns x and can thus compute $g(x, y)$ itself.

In more detail, each garbling of g has the same set of output wires $\{Z_i^b\}$. P_1 sends *output commitments* for each output wire of g, where each output commitment is a secret sharing of its trapdoor w. More concretely, for $i \in [n_3]$, P_1 sends $g^{w_i^0}, g^{w_i^1}$ to P_2, where $w_i^0 + w_i^1 = w$. In addition, it also sends *output recovery commitments* $g^{w_i^0 + K_{j,i}^0}, g^{w_i^1 + K_{j,i}^1}$ for each circuit j, their decommitments $w_i^0 + K_{j,i}^0, w_i^1 + K_{j,i}^1$, along with encryptions of $K_{j,i}^b$ under Z_i^b. Now, for output-wire label Z_i^b recovered by P_1, it learns $K_{j,i}^b$ by decrypting the encryption and can thus recover w_i^b. Note that if it also learns Z_i^{1-b}, it recovers w_i^{1-b} and can thus recover w by computing $w_i^b + w_i^{1-b}$, allowing it to learn P_1's input x.

Formal description. We now proceed to the formal description of the protocol.

Private inputs: P_1 has input $x \in \{0,1\}^{n_1}$ and P_2 has input $y \in \{0,1\}^{n_2}$.
Auxiliary inputs: Computational security parameter κ; statistical security parameter ρ; group $\mathbb{G}$ with (prime) order p and generator g; hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$; (ex-

tractable and equivocal) commitment scheme $(\mathsf{Com}, \mathsf{Open})$; circuit $C_0 : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to \{0,1\}^{n_3}$, where $C_0(x,y) :=$ "if $f_1(x)$ and $f_2(y)$ then $g(x,y)$ else $\perp$"; ideal functionalities $\mathcal{F}_{\mathbf{zkpok}}$ and $\mathcal{F}_{\mathbf{ot}}$.

Protocol:

Check that $f_1(x) = 1$:

1. If $f_1(x) = 0$ then P_1 sends $\perp$ to P_2, who aborts.
2. P_1 chooses $w \in_R \mathbb{Z}_p$, computes $h := g^w$, and sends h to P_2. P_1 then sends (prove, h, w) to $\mathcal{F}_{\mathbf{zkpok}}$, which sends $(\mathsf{verified}, h)$ to P_2.
3. P_2 constructs garbled circuit $\widehat{C}_{f_1}$ of function f_1. Let $\{X_i^b\}_{i\in[n_1]}^{b\in\{0,1\}}$ denote the input-wire labels.
4. For $i \in [n_1]$: P_1 computes $(A_i, B_i) := \mathsf{EGCommit}_h(x[i]; r_i)$ for random r_i, and sends (A_i, B_i) to P_2. Denote these as P_1's *input commitments*.
5. For $i \in [n_1]$: P_2 computes

$$(\widehat{A}_i^0, \widehat{B}_i^0) := (g^{s_i^0} h^{t_i^0}, A_i^{s_i^0} B_i^{t_i^0} \cdot X_i^0)$$
$$(\widehat{A}_i^1, \widehat{B}_i^1) := (g^{s_i^1} h^{t_i^1}, A_i^{s_i^1} (B_i/g)^{t_i^1} \cdot X_i^1),$$

   for random $s_i^0, t_i^0, s_i^1, t_i^1$, and sends $\widehat{A}_i^0$, $\widehat{B}_i^0$, $\widehat{A}_i^1$, and $\widehat{B}_i^1$ to P_1.
6. For $i \in [n_1]$: P_1 computes $X_i^{x[i]} := \widehat{B}_i^{x[i]} / (\widehat{A}_i^{x[i]})^{r_i}$.
7. P_2 sends $\widehat{C}_{f_1}$ to P_1, who evaluates it using input-wire labels $X_i^{x[i]}$, learning output-wire label Z_{f_1}. P_1 computes $(c_{f_1}, d_{f_1}) \leftarrow \mathsf{Com}(Z_{f_1})$, where Com is an equivocal and extractable commitment scheme, and sends c_{f_1} to P_2.
8. P_2 sends $\{s_i^0, t_i^0, s_i^1, t_i^1\}_{i\in[n_1]}$ to P_1, who recovers all the input-wire labels $\{X_i^b\}$, using the labels to check that $\widehat{C}_{f_1}$ was constructed correctly and aborting if not. Otherwise, P_1 sends d_{f_1} to P_2, who computes $Z_{f_1} := \mathsf{Open}(c_{f_1}, d_{f_1})$. If Z_{f_1} is the 1-bit output-wire label of $\widehat{C}_{f_1}$ then P_2 continues. Otherwise, P_2 outputs $\perp$.

Check that $f_2(y) = 1$:

9. If $f_2(y) = 0$ then P_2 sends $\perp$ to P_1, who aborts.
10. P_1 constructs garbled circuit $\widehat{C}_{f_2}$ of function $f_2'(y_1, \ldots, y_{\rho n_2}) = f_2(\bigoplus y_i)$ using seed seed_0 as the initial randomness. Let $\{Y_i^b\}_{i\in[\rho n_2]}^{b\in\{0,1\}}$ denote the input-wire labels.
11. For $j \in [\rho]$: P_1 chooses seed $\mathsf{seed}_j \in_R \{0,1\}^\kappa$ and key $k_j \in_R \{0,1\}^\kappa$.
12. For $j \in [\rho]$, $i \in [\rho n_2]$: P_1 chooses $Y_{j,i}^0 \in_R \{0,1\}^\kappa$ and $Y_{j,i}^1 \in_R \{0,1\}^\kappa$ using randomness derived from seed_j.
13. For $i \in [\rho n_2]$ the parties run $\mathcal{F}_{\mathbf{ot}}$: P_2 inputs $(\mathsf{choose}, i, y[i])$ and P_1 inputs $(\mathsf{transfer}, i, (Y_i^0, \{Y_{j,i}^0\}_{j\in[\rho]}), (Y_i^1, \{Y_{j,i}^1\}_{j\in[\rho]}))$, with P_2 receiving $(\mathsf{transferred}, i, (Y_i^{y[i]}, \{Y_{j,i}^{y[i]}\}_{j\in[\rho]}))$.
14. P_1 sends $\widehat{C}_{f_2}$ to P_2, who evaluates it using wire labels $Y_i^{y[i]}$, learning output wire label Z_{f_2}. P_2 computes $(c_{f_2}, d_{f_2}) \leftarrow \mathsf{Com}(Z_{f_2})$, where Com is an extractable commitment, and sends c_{f_2} to P_1.

15. P_1 sends seed_0 to P_2, who uses seed_0 to regenerate $\widehat{C}_{f_2}$ and check it was constructed correctly; if not, or if the input-wire labels generated from seed_0 do not match those it received from $\mathcal{F}_{\mathbf{ot}}$, then P_2 aborts. Otherwise, P_2 sends d_{f_2} to P_1, who computes $Z_{f_2} := \mathsf{Open}(c_{f_2}, d_{f_2})$. If Z_{f_2} is the 1-bit output-wire label of $\widehat{C}_{f_2}$ then P_1 continues. Otherwise, P_1 outputs $\perp$.

Evaluate $g(x, y)$:

16. For $i \in [n_3]$: P_1 chooses $w_i^0 \in_R \mathbb{Z}_p$, sets $w_i^1 := w - w_i^0$, computes *output commitments* $h_i^0 := g^{w_i^0}$ and $h_i^1 := g^{w_i^1}$, and sends h_i^0 and h_i^1 to P_2. P_2 checks that $h_i^0 \cdot h_i^1 = h$, aborting if not.
17. For $j \in [\rho]$:
    (a) The parties run $\mathcal{F}_{\mathbf{ot}}$: P_2 inputs (choose, j, b) for $b \in_R \{0,1\}$ and P_1 inputs $(\mathsf{transfer}, j, k_j, \mathsf{seed}_j)$, with P_2 receiving $(\mathsf{transferred}, j, k_j)$ or $(\mathsf{transferred}, j, \mathsf{seed}_j)$.
    (b) For $i \in [n_1]$, $b \in \{0,1\}$: P_1 computes $(u_{j,i}^b, v_{j,i}^b) := \mathsf{EGCommit}_h(b; r_{j,i}^b)$, where $r_{j,i}^b$ is derived from seed_j.
    (c) P_1 constructs garbling $\widehat{C}_j$ of function $g'(x, y_1, \ldots, y_{\rho n_2}) = g(x, \bigoplus y_j)$, where P_1's ith input-wire labels are defined as $(H(u_{j,i}^0, v_{j,i}^0), H(u_{j,i}^1, v_{j,i}^1))$, P_2's ith input-wire labels are defined as $(Y_{j,i}^0, Y_{j,i}^1)$, and the randomness used to construct $\widehat{C}_j$ is derived from seed_j. Let $\{Z_i^b\}$ be the output-wire labels, which are the same across each circuit. P_1 sends $\widehat{C}_j$ to P_2.
    (d) For $i \in [n_1]$: P_1 computes $(c_{j,i}^0, d_{j,i}^0) \leftarrow \mathsf{Com}(u_{j,i}^0, v_{j,i}^0)$, $(c_{j,i}^1, d_{j,i}^1) \leftarrow \mathsf{Com}(u_{j,i}^1, v_{j,i}^1)$, and sends $\{c_{j,i}^\pi, c_{j,i}^{1-\pi} : \pi \in_R \{0,1\}\}$ to P_2.
    (e) For $i \in [n_3]$: P_1 chooses $K_{j,i}^0, K_{j,i}^1 \in_R \mathbb{Z}_p$ and sends *output recovery commitments* $h_i^0 \cdot g^{K_{j,i}^0}$ and $h_i^1 \cdot g^{K_{j,i}^1}$ and encryptions $\mathsf{Enc}_{Z_i^0}(K_{j,i}^0)$, $\mathsf{Enc}_{Z_i^1}(K_{j,i}^1)$ to P_2.
    (f) Let

$$\begin{aligned}\mathsf{Inputs}_j &:= \{c_{j,i}^{x[i]}, d_{j,i}^{x[i]}\}_{i\in[n_1]}\\ \mathsf{InputEquality}_j &:= \{r_i - r_{j,i}^{x[i]}\}_{i\in[n_1]}\\ \mathsf{OutputDecom}_j &:= \{(w_i^0 + K_{j,i}^0, w_i^1 + K_{j,i}^1)\}_{i\in[n_3]}.\end{aligned}$$

    P_1 sends $\mathsf{Enc}_{k_j}(\mathsf{Inputs}_j, \mathsf{InputEquality}_j, \mathsf{OutputDecom}_j)$ to P_2.
18. For all check circuits j (i.e., where P_2 received seed_j in Step 17a), proceed as follows:
    (a) P_2 checks that seed_j generates $\widehat{C}_j$ and the other values constructed using randomness derived from seed_j, and aborts if not.
19. Set $\mathsf{cheat} := 0$. For all evaluation circuits j (i.e., where P_2 received key k_j in Step 17a), proceed as follows:
    (a) P_2 decrypts $\mathsf{Enc}_{k_j}(\mathsf{Inputs}_j, \mathsf{InputEquality}_j, \mathsf{OutputDecom}_j)$.
    (b) For $i \in [n_1]$: P_2 computes $(\widetilde{u}_{j,i}^{x[i]}, \widetilde{v}_{j,i}^{x[i]}) := \mathsf{Open}(c_{j,i}^{x[i]}, d_{j,i}^{x[i]})$ and checks that $(g^{r_i - r_{j,i}^{x[i]}} \cdot \widetilde{u}_{j,i}^{x[i]}, h^{r_i - r_{j,i}^{x[i]}} \cdot \widetilde{v}_{j,i}^{x[i]}) = (A_i, B_i)$; if not set $\mathsf{cheat} := 1$.
    (c) For $i \in [n_3]$, $b \in \{0,1\}$: P_1 checks that $g^{w_i^b + K_{j,i}^b}$ equals the output recovery commitments sent by P_1; if not set $\mathsf{cheat} := 1$.

(d) P_2 evaluates $\widehat{C}_j$ using $Y_{j,i}^{y[i]}$ as its input-wire labels and learning output-wire labels $\{Z_i\}$. P_2 uses these labels to learn the appropriate $K_{j,i}^b$ values, and uses these to check that $h_j^b \cdot g^{K_{j,i}^b}$ equals the appropriate output recovery commitment sent by P_1; if not set $\mathsf{cheat} := 1$. If this succeeds, P_2 marks the circuit as "semi-trusted".

20. If $\mathsf{cheat} = 1$ then abort. Otherwise, if all the semi-trusted circuits have the same output-wire labels, P_2 outputs that value. Otherwise, let $Z_{j,i}$ and $Z_{j',i}$ be two differing output-wire labels for garbled circuits j and j' and output wire i. P_2 can extract w_i^0 and w_i^1 by using the sets $\mathsf{OutputDecom}_j$ and $\mathsf{OutputDecom}_{j'}$, and thus learn w, allowing P_2 to decrypt P_1's initial commitments to learn x. P_2 then outputs $g(x, y)$.

Theorem 5.1. *The protocol above securely realizes $\mathcal{F}_{\mathsf{2pc}}$ in the $(\mathcal{F}_{\mathbf{ot}}, \mathcal{F}_{\mathbf{zkpok}})$-hybrid model.*

Proof. We prove security by constructing simulators for the case that either P_1 or P_2 is corrupted.

Malicious P_1. Suppose adversary $\mathcal{A}$ corrupts P_1. We construct a simulator $\mathcal{S}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{A}$ on its input.

2. If $\mathcal{A}$ sends $\perp$ in Step 1, $\mathcal{S}$ sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

3. In Step 2, $\mathcal{S}$ receives (prove, h, w) from $\mathcal{A}$. If $h \neq g^w$ then $\mathcal{S}$ sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

4. In Step 4, $\mathcal{S}$ uses w extracted above to extract $x \in \{0,1\}^{n_1} \cup \{\perp\}$ from the commitments sent by $\mathcal{A}$, where $x = \perp$ if any of the commitments are invalid.

5. $\mathcal{S}$ continues to act as an honest P_2 would, where if P_2 would abort then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{2pc}}$. In Step 8, $\mathcal{S}$ checks if either $x = \perp$ or $f_1(x) = 0$; if so, $\mathcal{S}$

sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, $\mathcal{S}$ sends (input, x) to $\mathcal{F}_{\mathsf{2pc}}$.

6. $\mathcal{S}$ extracts $\mathcal{A}$'s input to $\mathcal{F}_{\mathbf{ot}}$, and uses these values to open the garbled circuit sent by $\mathcal{A}$, thus learning the one-bit output-wire label Z^1. $\mathcal{S}$ sends $\mathsf{Com}(Z^1)$ to $\mathcal{A}$.

7. $\mathcal{S}$ receives seed_0 from $\mathcal{A}$ and checks consistency with the values received in $\mathcal{F}_{\mathbf{ot}}$ and the garbled circuit. If anything fails, $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

8. $\mathcal{S}$ continues to act as an honest P_2 would. If $\mathsf{cheat} = 0$ in Step 20 then $\mathcal{S}$ sends $(\mathsf{continue})$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, (i.e., $\mathsf{cheat} = 1$), $\mathcal{S}$ sends (abort) to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

We now prove that the view of $\mathcal{A}$ is computationally indistinguishable in the hybrid and ideal worlds. We do so by a series of hybrid experiments.

$\mathbf{H_1}$. Same as the hybrid-world execution.

$\mathbf{H_2}$. Same as **$\mathbf{H_1}$**, except that P_2 extracts w from P_1's message to $\mathcal{F}_{\mathbf{zkpok}}$ and uses w to extract P_1's input x.

These two hybrids are indistinguishable by the use of $\mathcal{F}_{\mathbf{zkpok}}$ and the security of the commitment scheme.

$\mathbf{H_3}$. Same as **$\mathbf{H_2}$**, except that P_2 aborts in Step 8 if $f_1(x) = 0$.

These hybrids are computationally indistinguishable by the hiding property of the ElGamal-based oblivious transfer and the security of the garbling scheme. Namely, in $\mathbf{H_2}$, $\mathcal{A}$ cannot recover the appropriate input-wire label in Step 5 for those input bits which are incorrectly committed and likewise can only recover one of the two input-wire labels for those input bits which are correctly committed. Thus, by the authenticity property of the garbling scheme, $\mathcal{A}$ is unable to recover the one-bit output-wire label Z^1 with high probability. Thus, if $\mathcal{A}$ can distinguish between $\mathbf{H_2}$, where P_2 aborts due to $\mathcal{A}$ committing to an invalid output-wire label, and $\mathbf{H_3}$, where P_2 aborts regardless of what $\mathcal{A}$ commits to, then this leads to an attack on the authenticity property of the garbling scheme.

$\mathbf{H_4}$. Same as $\mathbf{H_3}$, except that P_2 aborts if the input-wire labels derived from seed_0 do not match those it received from $\mathcal{A}$ when simulating $\mathcal{F}_{\mathbf{ot}}$.

These two hybrids are statistically indistinguishable by the use of the XOR-tree. Namely, for $\mathcal{A}$ to distinguish between these two hybrids it must correctly guess P_2's input to $\mathcal{F}_{\mathbf{ot}}$. However, as this input is secret shared, $\mathcal{A}$ only succeeds with probability $\leq 2^{-\rho}$.

$\mathbf{H_5}$. Same as $\mathbf{H_4}$, except that P_2 aborts if all the evaluated circuits are not correctly constructed.

These two hybrids are perfectly indistinguishable except that P_2 may abort in $\mathbf{H_5}$ and not $\mathbf{H_4}$. However, this only happens if $\mathcal{A}$ correctly guesses which circuits will end up as check versus evaluation circuits, which happens with

probability $2^{-\rho}$.

$\mathbf{H_6}$. Same as $\mathbf{H_5}$, except that P_2 uses P_1's extracted input x to compute and output $g(x, y)$ instead of evaluating the garbled circuits.

These two hybrids are perfectly indistinguishable because if $\mathcal{A}$ tries to cheat in $\mathbf{H_6}$ then P_2 can extract $\mathcal{A}$'s input and just compute $g(x, y)$ locally and otherwise P_2 retrieves $g(x, y)$ by evaluating the garbled circuits.

As $\mathbf{H_6}$ is the same as the ideal world protocol, this completes the proof for a malicious P_1.

Malicious P_2. Suppose adversary $\mathcal{A}$ corrupts P_2. We construct a simulator $\mathcal{S}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{A}$ on its input.
2. If $\mathcal{S}$ receives $(\mathsf{input}, \perp)$ from $\mathcal{F}_{\mathsf{2pc}}$, then $\mathcal{S}$ sends $\perp$ to $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs.
3. $\mathcal{S}$ acts as an honest P_1 would, using 0^{n_1} as P_1's input, until Step 7, at which point $\mathcal{S}$ commits to a random value.
4. $\mathcal{S}$ continues to act as an honest P_1 would, where in Step 8 it opens the garbled circuit sent by $\mathcal{A}$ and learns the one-bit output-wire label Z^1. If $\mathcal{S}$ fails to open the garbled circuit, it sends $\perp$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs. Otherwise, it equivocates on its previously sent commitment to make the committed value equal to Z^1.

5. In Step 9, if $\mathcal{A}$ sends $\perp$ then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{2pc}}$ and outputs whatever $\mathcal{A}$ outputs.

6. $\mathcal{S}$ extracts y from $\mathcal{F}_{\mathbf{ot}}$ and proceeds to act as an honest P_1 would until Step 14. Here, if $f_2(y) = 0$ then $\mathcal{S}$ sends $(\mathsf{input}, \perp)$ to $\mathcal{F}_{\mathsf{2pc}}$, outputting whatever $\mathcal{A}$ outputs.

7. $\mathcal{S}$ continues to act as an honest P_1 would until Step 17a. Here, $\mathcal{S}$ extracts $\mathcal{A}$'s choices as to which circuits are check circuits and which are evaluation circuits. For check circuit j, $\mathcal{S}$ replaces the key k_j input to $\mathcal{F}_{\mathbf{ot}}$ with a random string.

8. In Step 17, $\mathcal{S}$ sends $(\mathsf{input}, \mathsf{ok})$ to $\mathcal{F}_{\mathsf{2pc}}$, receiving (output, z), and proceeds as follows:

   - For the check circuits, $\mathcal{S}$ constructs them as an honest P_1 would.
   - For the evaluation circuits, $\mathcal{S}$ uses fresh randomness to generate everything related to the garbling and garbles a circuit with fixed output z. It also replaces $\mathsf{Com}(u_{j,i}^{1-y[i]}, v_{j,i}^{1-y[i]})$ with commitments to zeros, and $\mathsf{Enc}_{Z_i^{1-z[i]}}(K_{j,i}^{1-z[i]})$, with encryptions to zeros.

9. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We now prove that the view of $\mathcal{A}$ is computationally indistinguishable in the hybrid and ideal worlds. We do so by a series of hybrid experiments.

$\mathbf{H_1}$. Same as the hybrid-world execution.

$\mathbf{H_2}$. Same as $\mathbf{H_1}$, except P_1 equivocates on the commitment it sends to P_2 in Step 7 to be the output of $\widehat{C}_{f_1}$.

These two hybrids are computationally indistinguishable based on the security of the equivocal commitment scheme.

$\mathbf{H_3}$. Same as $\mathbf{H_2}$, except that in Step 15 P_1 aborts if $f_2(y) = 0$.

These two hybrids are computationally indistinguishable based on the authenticity property of the garbled circuit.

$\mathbf{H_4}$. Same as $\mathbf{H_3}$, except that P_1 replaces the k_j values for the check circuits with random values and generates the evaluation circuits using fresh randomness.

These two hybrids are perfectly indistinguishable in the $\mathcal{F}_{\mathbf{ot}}$-hybrid model.

$\mathbf{H_5}$. Same as $\mathbf{H_4}$, except that P_1 uses 0^{n_1} as its input to the check circuits.

These two hybrids are computationally indistinguishable by the security of the encryption scheme.

$\mathbf{H_6}$. Same as $\mathbf{H_5}$, except that P_1 replaces the commitments of $(u_{j,i,}^{1-y[i]}, v_{j,i}^{1-y[i]})$ with commitments to zeros in the evaluation circuits.

These two hybrids are computationally indistinguishable by the security of the commitment scheme.

$\mathbf{H_7}$. Same as $\mathbf{H_6}$, except that P_1 uses the output z of $\mathcal{F}_{\mathsf{2pc}}$ to construct fake garbled circuits with fixed output z for all evaluation circuits.

These two hybrids are computationally indistinguishable by the security of the garbling scheme.

$\mathbf{H_8}$. Same as **$\mathbf{H_7}$**, except that P_1 replaces the output encryptions for all output bits that do not correspond to z with encryptions of zero.

These two hybrids are computationally indistinguishable by the security of the encryption scheme.

$\mathbf{H_9}$. Same as **$\mathbf{H_8}$**, except that P_1 replaces its input with 0^{n_1} in the evaluation circuits and input commitments.

These two hybrids are computationally indistinguishable by the security of the ElGamal commitment scheme.

$\mathbf{H_{10}}$. Same as **$\mathbf{H_9}$**, except that P_1 replaces the input-wire labels for P_2's input that do not correspond to y with random strings.

These two hybrids are computationally indistinguishable by the security of the garbling scheme.

As **$\mathbf{H_{10}}$** is the same as the ideal world protocol, this completes the proof for a malicious P_2, and thus the proof of the theorem. □

5.3 Protocol Optimizations

We begin by noting a couple of immediate optimizations to our protocol. First off, assuming the random oracle model, we can instantiate all the commitment

operations with a hash function. We also note that we can use *privacy-free* garbled circuits [63] with the "half gate" optimization [54] for the garbling of f_1 and f_2, taking only one ciphertext per non-free gate. Finally, we can instantiate $\mathcal{F}_{\mathbf{zkpok}}$ efficiently using Schnorr's protocol [64] and $\mathcal{F}_{\mathbf{ot}}$ using the OT protocol of Chou and Orlandi [44] and malicious OT extension [49].

As our protocol requires public key operations for both P_1's and P_2's inputs, we consider optimizations to reduce the number of exponentiations required. First off, when P_1 computes values of the form $g^s h^t$ in $\mathsf{EGCommit}$, only one exponentiation is needed since P_1 knows w such that $h = g^w$ and thus can directly compute g^{s+wt} $(= g^s h^t)$. For P_2, $g^s h^t$ can be computed more efficiently using the "Euclidean method" described by de Rooij [65]. The high level idea is to apply the following observation recursively:

$$g^s h^t = (gh^q)^s h^p, q = \lfloor \frac{t}{s} \rfloor, p = t \mod s.$$

We also note that for both P_1 and P_2, most of the exponentiations are *fixed-base* exponentiations, which can be computed much more efficiently using pre-computed tables [66].

We also note that our protocol as written only addresses the situation where all the input bits are used both in the predicate check stage (i.e., the proofs that $f_1(x) = 1$ and $f_2(y) = 1$) and the computation stage (i.e., the computation of $g(x, y)$), which may not always be the case. When only parts of the input are used in the predicate check or computation stage, we do not need the heavy machinery

we use to ensure input consistency between each party's input in the two stages.

To be more specific, we consider the input of each party as three parts:

1. Input used only in the predicate check stage (denote these inputs as x_1, y_1);
2. Inputs used in both the predicate check and computation stages (denote these inputs as x_2, y_2);
3. Inputs used only in the computation stage (denote these inputs as x_3, y_3).

For the first case (i.e., inputs x_1 and y_1) we can use committed OT which allows us to use OT extension for input x_1 and avoid the XOR-tree for input y_1. For the third case (i.e., inputs x_3 and y_3), we can handle these as in the work of Afshar et al. [21]; see below for details.

Denote P_1's input by $x = (x_1 \| x_2 \| x_3)$, P_2's input by $y = (y_1 \| y_2 \| y_3)$, and the function to be computed by:

$$f(x, y) = \text{“if } f_1(x_1, x_2) \text{ and } f_2(y_1, y_2) \text{then } g(x_2, x_3, y_2, y_3) \text{ else } \perp\text{”}.$$

We can construct a protocol for dealing with this extended case as follows. It is the same as the protocol described in Section 5.2 except with the following changes:

1. For input x_1, we can skip the input commitment steps (Steps 4–6) and checking step (Step 8). This allows us to use a committed OT which works with OT extension.
2. For input y_1, we can skip the XOR-tree (Step 10). Instead, we can use committed OT as above.

3. For inputs x_2 and y_2, these are handled as in our original protocol.

4. When computing $g(\cdot, \cdot)$, we use $\mathsf{EGCommit}$ to ensure the consistency of x_3 among computation circuits.

5. For input y_3 we do not need the XOR-tree, and can instead use committed OT during the computation stage.

For several real world examples, these extensions lead to important practical improvements; see Section 5.4.

5.4 Evaluation

In this section, we compare our protocol with generic malicious two-party computation protocols for several example functions to showcase the gains in communication and computation that our approach gives. In particular, we compare our protocol with the protocol of Afshar et al. [21], the most efficient and practical malicious 2PC construction that we are aware of. We refer to this protocol as the "generic solution" in contrast to our solution which is specifically designed for the type of functions we consider. We evaluate the improvement based on the speedup of both computation and communication. We do so by calculating the number of symmetric key operations, public key operations, and bytes sent by both our protocol and the generic solution. While obviously a rough approximation of the actual running time of an implementation, we believe this gives a good benchmark independent of implementation details, computer/network configuration, etc.

While we are aware of more efficient *customized* protocols for some of the examples discussed below, these protocols are not as flexible as our approach. For example, it is usually very difficult, and sometimes even impossible, to change or even just extend a customized protocol to support secure pre- or post-computation, which in many real-world settings seems necessary. As an example, consider the following use-case for private set intersection: a dating application would like to securely compute the intersection of two peoples' interests, and then give weights to the matched items in order to compute some expected match percentage. This requires some post-processing on the matched items, which existing customized protocols are unable to do as they reveal the items upon completion of the private set intersection protocol.

We assume a computational security parameter of $\kappa = 128$ and a statistical security parameter of $\rho = 40$. We utilize all known garbled circuit optimizations, including privacy-free garbled circuits [63] for computing the predicate checks, the "half-gates" optimization [54] for reducing the size of the garbled circuit, elliptic curve cryptography for smaller public key sizes, etc. If not specified otherwise, we use $\gamma = 1250$ as the ratio between the cost of a public key operation and a symmetric key operation. (As our protocol makes heavy use of public key operations, a smaller ratio leads directly to better results for our protocol.) This number is derived from estimates using the Crypto++ benchmark [59] and OpenSSL, and while this is of course a rough estimate, we believe it is reasonably accurate for current systems. Note that we do not separate the cost of, e.g., fixed-base exponentiations and the exponentiate-and-multiply optimizations as discussed in Section 5.3, which in a real

implementation would further reduce this ratio.

In what follows we show different examples where input checking improves the performance of realistic functions. To briefly summarize our findings, we find that in many applications our improvement yields up to about 56× improvement in terms of computation and 80× improvement in terms of communication. (The exact improvement in concrete running time will of course be a combination of these two improvements depending on the computational power of the parties and the network throughput.) Although we discuss signature checks and local computation separately, they can be used together, which makes the predicate circuit larger and our results better.

5.4.1 Signature Checks on Inputs

One of the main applications of our improved protocol is to efficiently check that the input of each party is correctly signed by some third party. The motivation here is that the malicious security model allows an attacker to carefully choose some fake but consistent input that helps it learn extra information from the other party, such as by supplying the full universe in a private set intersection computation to learn the other party's input. A solution to this problem using existing protocols is to compute a functionality that first checks a pre-signed signature on the input and then computes the original function if and only if the signature is valid. However, checking a signature within a garbled circuit is extremely expensive, and often more expensive than the underlying computation itself. Our protocol is particularly beneficial here,

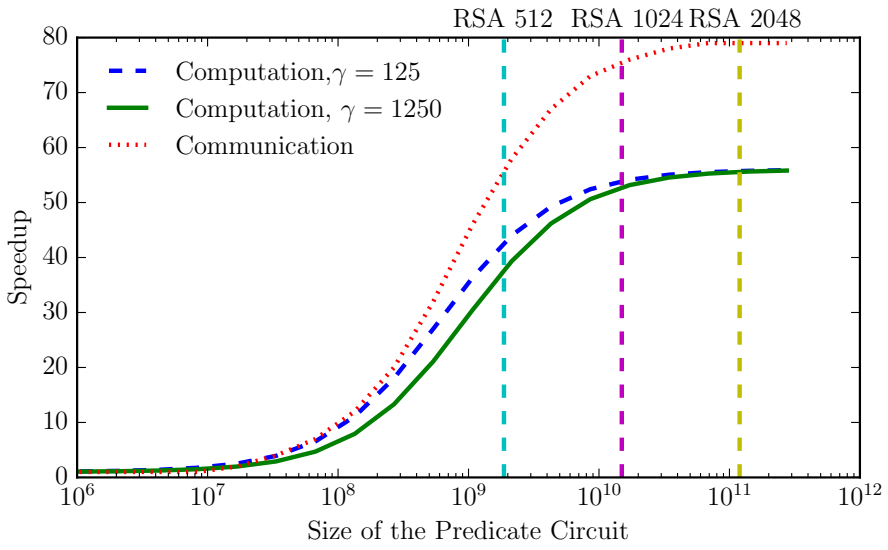


Figure 5.3: **Varying the predicate circuit size.** We fix the input size of each party to 5000 bits and the size of computation circuit $g(\cdot, \cdot)$ to ten million gates, and vary the size of the predicate circuit for party P_1. We use two ratios, $\gamma = 125$ and $\gamma = 1250$, for the public-key to symmetric-key cost. The curves represent the communication and computation improvement of our protocol compared to the generic protocol by Afshar et al., with the vertical lines denoting the sizes of the circuits for RSA 512, RSA 1024 and RSA 2048.

as it reduces the cost of the signature check by $O(\rho)$ times with only a slight increase in public key operations required.

In the following, we evaluate our protocol using both "small" and "large" inputs. For computing the signature verification, we follow the hash-and-sign paradigm and first hash the input to a 512-bit digest which we verify, and use SHA-256 as the underlying hash function.

Signature checks for "small" inputs. Suppose both parties have 5000 bits of input and P_1 also has a signature on its input. The parties would like to compute a circuit with ten million (non-free) gates if P_1's input is correctly signed.[2]

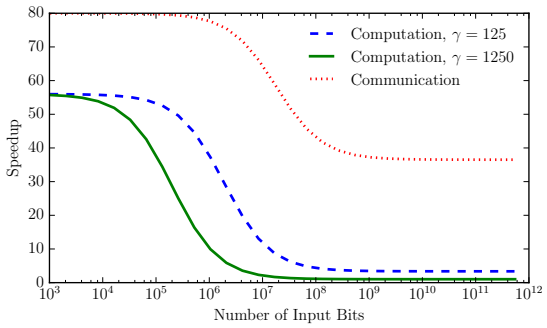
[2]We use a computation circuit with ten million gates to be able to cover many practical circuits. Using a computation circuit with smaller size only benefits our comparison.

In Figure 5.3, we show the improvement of this setting for various sizes of the predicate circuit, from 10^6 to 10^{12}. Particularly, we highlight three special cases, where the size of the predicate circuit corresponds to signature verification using either RSA 512, RSA 1024, or RSA 2048.[3] We obtained the sizes for these circuits using an existing circuit compiler work [58]. As we can see in Figure 5.3, for RSA 512 we are able to achieve an improvement of about 40× for computation and 50× for communication. For a large enough predicate circuit, such as when using RSA 2048, we are able to achieve up to about 56× speedup in computation and up to about 80× speedup in communication.

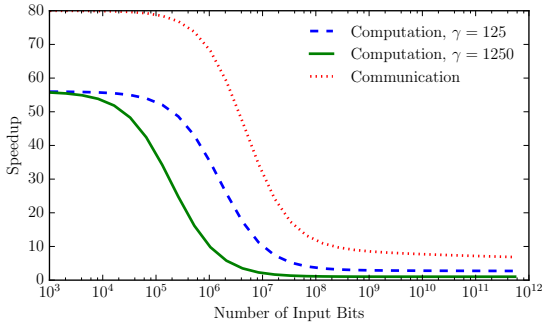
Note that these numbers agree with what we would expect asymptotically. Let $|C|$ be the size of the predicate circuit. The protocol by Afshar et al. [21] needs to perform $40 \cdot 4 \cdot |C| + 20 \cdot 4 \cdot |C| + 20 \cdot 2 \cdot |C| = 280|C|$ symmetric key operations (to garble and evaluate the circuits), and send $40 \cdot 2 \cdot |C| = 80|C|\kappa$ bits. On the other hand, our protocol only need to perform $2|C| + 2|C| + |C| = 5|C|$ symmetric key operations and send $|C|\kappa$ bits when using privacy-free garbled circuits and the "half-gates" optimization. Thus, the asymptotic improvement is $280/5 = 56$ for computation and $80/1 = 80$ for communication when calculating the predicate circuit on its own. Thus, when the predicate circuit is much larger than the computation circuit, these costs dominate the overall cost and the asymptotic bound is reached.

Signature checks for "large" inputs. In Figure 5.4, we consider a similar situation as above, but here we vary the *input size* of P_1's input, using RSA 2048 as

[3] We use an RSA-based signature scheme because this is the only signature scheme with known circuit sizes.



(a) N sized computation circuit



(b) $N \log N$ sized computation circuit

Figure 5.4: **Varying the input size.** We fix the predicate circuit to be RSA 2048 and vary P_1's input length N from 10^3–10^{12} bits, with the size of the computation circuit based on the input size. The left graph presents the speedup versus the generic approach for a computation circuit of size N, and the right graph presents the speedup versus the generic approach for a computation circuit of size $N \log N$. We present results for both $\gamma = 125$ and $\gamma = 1250$ for the ratio of public-key to symmetric-key costs.

the signature scheme. In Figure 5.4(a) the computation circuit is of size N for N bit input, while in Figure 5.4(b) the computation circuit size is $N \log N$.

We can see that the improvement is about 80 for communication and about 56 for computation up to around 10^5 input bits. When the input size becomes more than 10^7 bits, the improvement for computation is less than $10\times$, and the improvement for communication reduces to about $40\times$ for the linear computation circuit and about $10\times$ for the $N \log N$ computation circuit. Note that the main reason for such a reduction is that as the number of input bits increase the cost

of checking the signature becomes amortized away, in which case our improvement becomes less significant.

Note however, that (1) in both cases, our protocol never performs worse than that of Afshar et al. [21] in terms of computation and improves 10–40× in terms of communication, and (2) the reduction in the improvement only happens when the number of input bits is huge (about ten million).

5.4.2 Enforcing Correct Local Computation

Using local computation to reduce the cost of 2PC in the semi-honest model has been used in several existing works [60, 61]. Our protocol is able to provide some of these same benefits in the malicious model. Suppose two parties want to compute $f(x, y)$, which can be represented as $h_3(h_1(x), h_2(y))$, for some functions $h_1(\cdot)$, $h_2(\cdot)$, and $h_3(\cdot, \cdot)$. In the semi-honest setting, we let the parties compute $h_1(x)$ and $h_2(y)$ locally and then jointly perform a semi-honest secure computation on $h_3(\cdot, \cdot)$. Here, the bottleneck is now computing $h_3(\cdot, \cdot)$, as the other computations are all local. However, in the malicious setting, the advantage of local computation is completely lost: the result of the local computation cannot be trusted in the malicious setting. Therefore, a generic malicious protocol needs to compute a circuit that contains both local computation ($h_1(\cdot)$ and $h_2(\cdot)$) and joint computation ($h_3(\cdot, \cdot)$).

However, using our protocol, we can view predicate checking as a way to ensure that local computation is done honestly. That is, the two parties first locally compute $H_1 = h_1(x)$ and $H_2 = h_2(y)$. Then they use $H_1 \| x$ and $H_2 \| y$ as their

input to the protocol, using predicate functions $f_1(H_1\|x) := (H_1 \stackrel{?}{=} h_1(x))$ and $f_2(H_2\|y) := (H_2 \stackrel{?}{=} h_2(y))$ and computation function $g(x, y) = h_3(H_1, H_2)$. This is particularly beneficial when there are more efficient ways of checking that, say, $H_1 \stackrel{?}{=} h_1(x)$, than redoing the local computation itself. For example, checking that a list of N elements is sorted takes $O(N)$ time whereas sorting a list of N elements takes $O(N \log N)$ time.

Thus, using our protocol improves over generic malicious 2PC for the following two reasons:

1. We save a factor of $O(\rho)$ on the predicate circuits used to check the local computation.

2. Since x and y are not used in the underlying computation directly, they do not require the machinery needed to enforce input consistency. That is, we only need to ensure the consistency of $h_1(x)$ and $h_2(y)$, which can be much smaller than the original input (see the examples below for more details).

We look at three examples of protocols that can be improved using local computation: (1) private edit distance approximation, (2) solving a linear system, and (3) private set intersection.

Private edit distance approximation. Wang et al. [61] designed an algorithm to approximate the edit distance of two genome sequences in the semi-honest setting. They proposed several optimizations that minimize the circuit for joint computation. Let N be the number of edits in the genome compared to the reference genome, and let ϵ be the relative error we want to achieve with $2^{-\delta}$ failure probability. Dur-

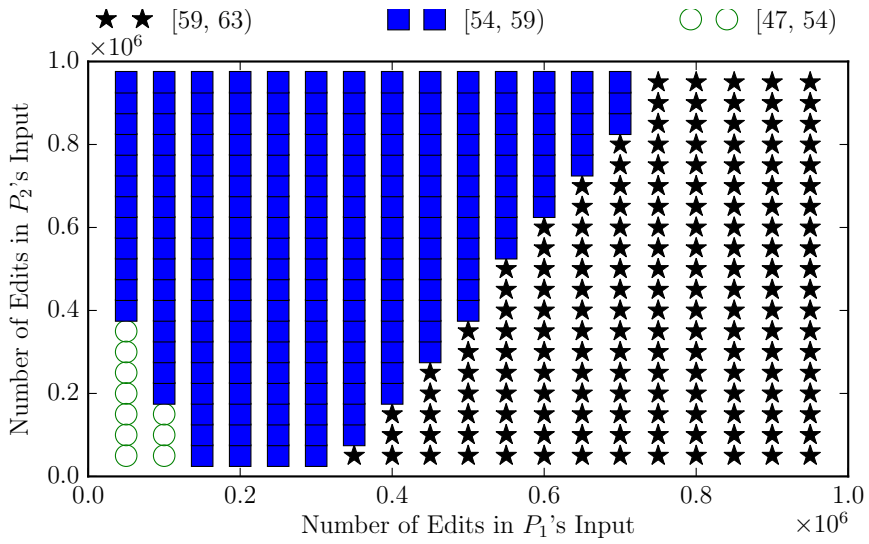


Figure 5.5: **Computation improvement for private edit distance approximation.** We vary the input size of each party and fix the ratio of public-key to symmetric-key costs to $\gamma = 1250$. ★ represents a speedup in the range $[59, 63)$, ■ represents a speedup in the range $[54, 59)$, and ○ represents a speedup in the range $[47, 54)$.

ing the local computation, each party hashes each edit to either 1 or -1 and sums them up, while the joint computation computes the square of the difference between the two sums. In order to achieve the error mentioned above, we need to compute this $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ times, each time using a new random hash function. Therefore, local computation is on the order of $O(N/\epsilon^2\log\frac{1}{\delta})$, while the joint computation has a circuit of size $O(\frac{\log N}{\epsilon^2}\log\frac{1}{\delta})$. Thus, whereas the generic solution in the malicious setting has a complexity of $O\left(\rho\kappa\left(\frac{N}{\epsilon^2}\log\frac{1}{\delta}+\frac{\log N}{\epsilon^2}\log\frac{1}{\delta}\right)\right)$, our protocol has only $O\left(\kappa\frac{N}{\epsilon^2}+\rho\kappa\frac{\log N}{\epsilon^2}\log\frac{1}{\delta}\right)$ complexity.

We compare the two protocols for a varying number of genome edits, based on an error rate of 1% with 95% confidence; see Figure 5.5. Our protocol achieves about 79× communication improvements for all combinations we tested, therefore we only show the computation improvement. We achieve a computation improvement up to

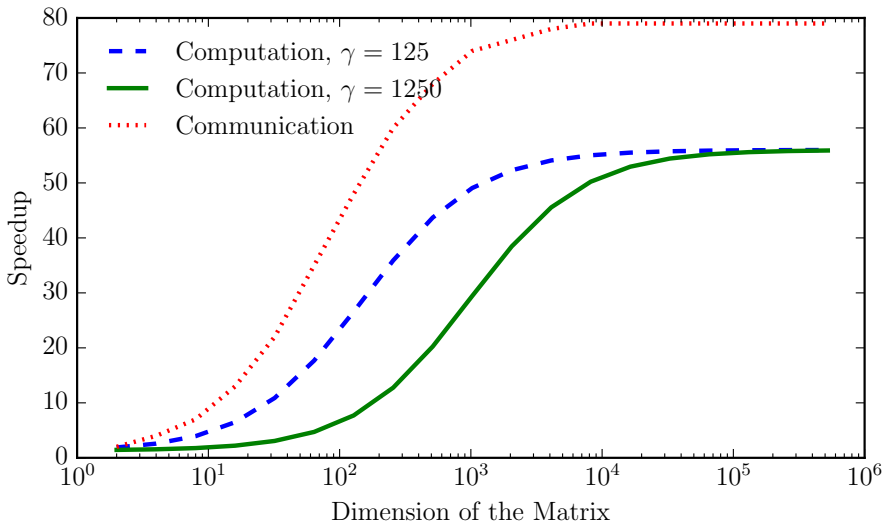


Figure 5.6: **Improvement when solving linear systems.** This graph shows the speedup in terms of computation and communication versus the naive approach when solving linear systems, where we vary P_1's input size and use $\gamma = 125$ and $\gamma = 1250$ as the ratios of public-key to symmetric-key costs.

about 63×, with the exact improvement increasing as we increase the input size of P_1 or P_2. Note that the improvement here is greater than the asymptotic bound of 56× described in Section 5.4.1 because here both parties do an input check while in the previous setting only P_1 did an input check. Having P_2 also do an input check leads to additional improvements.

Note that our protocol also works for other algorithms with a similar pattern as private edit distance approximation, such as heavy hitters, quantiles, etc.

Solving a linear system. Suppose P_1 holds an invertible matrix A and P_2 holds a vector b. The two parties want to securely solve the linear system $Ax = b$. A naive solution is to perform Gaussian elimination obliviously within the secure computation, which requires a circuit with $O(N^3)$ multiplications. A better solution in the semi-honest setting is to let P_1 compute A^{-1} locally so that the parties only

need to perform $O(N^2)$ multiplications in the secure computation portion of the protocol.

When it comes to the malicious setting, we can check that P_1 inputs a correct inverse by checking that $A^{-1}A = I$. Applying the generic solution gives us a protocol with complexity $O(\rho\kappa N^3)$ whereas our protocol achieves a complexity of $O(\kappa N^3 + \rho\kappa N^2)$.

As shown in Figure 5.6, we achieve an improvement of 10× in terms of communication when the dimension of the matrix is as small as 10. The improvement reaches the theoretical improvement calculated in Section 5.4.1 when the dimension increases to about one thousand. The computation improvement also behaves similarly to the previous example of checking signatures.

Private set intersection. We now evaluate private set intersection following the approach of Huang et al. [60]. Private set intersection has a predicate circuit of size N and a computation circuit of size $O(N \log N)$. We evaluated our protocol on this with input size up to one million and found a 1.3× improvement in computation and communication. While these gains are not as great as the order-of-magnitude gains for other functions, we note that a 30% improvement in running time is still significant.

The main reason for a smaller improvement than the order-of-magnitude improvements we see in the previous examples is because the predicate circuit is of size N for N input bits while the computation circuit size is $O(N \log N)$. This means that the cost is dominated by the computation circuit and hence we get smaller

gains.

Chapter 6: The Three Party Setting

Research on secure computation has traditionally been divided into two classes: work focusing on two-party computation (2PC), and work focusing on multi-party computation (MPC) for an arbitrary number of parties.[1] Yet, in practice, it seems that the most likely scenarios for secure MPC would involve a small number of parties. In general, as the number of parties increases, the cost of communication amongst the parties increases as well. In a wide-area network setting, this may have a huge impact on the running time of the protocol.

6.1 Our Contribution

Motivated by these observations, we initiate the study of efficient *three*-party computation (3PC) in the malicious model, tolerating an arbitrary number of corruptions. We construct the first practical, constant-round protocol for secure three-party computation of Boolean circuits. Our protocol uses player-simulation techniques in order to compile existing (cut-and-choose-based) 2PC protocols into three-party protocols. We instantiate our compiler with state-of-the-art 2PC constructions and show that the addition of a third party comes at the cost of roughly a factor eight overhead over

[1] Here we are interested in protocols tolerating an arbitrary number of corruptions. One could further distinguish work on MPC that assumes an honest majority.

the underlying 2PC protocol in terms of computation, and a factor sixteen overhead in terms of communication. This running time appears to be superior to existing state-of-the-art MPC protocols in terms of *start-to-finish* running time. Of course, computing the exact overhead requires implementations of both our protocol and the underlying 2PC protocol, which we leave as future work. As a further optimization point, our protocol makes *only three calls overall* to a broadcast channel (one with each party as sender), as opposed to existing practical MPC solutions (for more than two parties) which use broadcast for communicating all protocol messages. This may be important in certain wide-area network settings where communication (and broadcast specifically) is very expensive. The most efficient instantiation of our protocol requires the random oracle model. As a downside, our protocol does not currently support free-XOR [29] or garbled row reduction [53]; we leave such developments as future work.

Overview of our protocol. Denote the three parties by P_1, P_2, and P_3. The high-level idea of our construction is to execute a two-party protocol $\widehat{\pi}$, where one of the two parties (say $\widehat{P}_1$) is emulated by P_1 and P_2 via a two-party protocol π, and the other party is played by P_3.

Clearly, naively applying the above idea yields an inefficient construction even when state-of-the-art 2PC protocols are used for π and $\widehat{\pi}$. Assume, for example, that the most efficient 2PC protocol is used for both π and $\widehat{\pi}$, where π simply computes the circuit of $\widehat{P}_1$ among P_1 and P_2. The security of the resulting construction follows trivially from the composition theorem. However, unless the size of the circuit is very

small, this approach results in a huge blowup on the overall runtime; in particular, if t is the time π needs to compute the circuit of $\widehat{P}_1$ and $\widehat{t}$ is the time that $\widehat{\pi}$ needs to compute the three-party circuit, then the runtime of the above naive construction is $t \cdot \hat{t}$, yielding at least a quadratic blowup.

Emulating the garbler versus emulating the evaluator. One might be tempted to think that, because the role of the circuit evaluator in the protocol is more "passive" (in the sense that the computation is less complicated) than the circuit garbler, the most natural approach would be to emulate the evaluator among P_1 and P_2 (and have P_3 locally do the heavier work doing circuit generation and opening over broadcast). This seemingly direct approach fails as one needs a mechanism for P_1 and P_2 to include their inputs into the garbled circuits. Clearly, doing so by having P_1 first receive its input-wire labels via oblivious transfer (as in the standard garbled circuit constructions) and then handing them to P_2 yields an insecure protocol; indeed, an adversary corrupting P_2 and P_3 can then trivially learn P_1's inputs.

Instead, in this work we have P_1 and P_2 emulate the sender, and we have P_3 play the role of the receiver. More precisely, we adapt the distributed circuit-garbling technique [67, 68] to the two-party setting, allowing P_1 and P_2 to compute a sharing of a garbled circuit which they then reconstruct towards P_3. By appropriate optimizations, we ensure that distributed garbling requires P_1 and P_2 to compute and communicate roughly as much as the garbler in an execution of a standard two-party garbled circuit protocol (plus some oblivious transfer calls per gate); P_3 needs to do nothing during the circuit garbling. Most interestingly, our construction

features a mechanism which allows P_3 to receive the labels corresponding to its input bits for evaluating the garbled circuit by only one invocation of oblivious transfer per input-bit with each of P_1 and P_2.

Our distributed garbling scheme is secure against malicious adversaries, which ensures that an adversary corrupting only one of the parties P_1 or P_2 cannot produce a maliciously constructed garbled circuit. In order to protect against an adversary who corrupts both P_1 *and* P_2, we rely on the cut-and-choose technique. We give concrete instantiations (in the random oracle model) of our protocol using a combination of two 2PC protocols by Lindell and Pinkas [8, 9], as well as a construction based on the more recent protocol by Lindell [10] which drastically reduces the number of circuit garblings required for cut-and-choose.

Interestingly, the cut-and-choose technique does not only protect against corrupting both P_1 and P_2, but allows a considerable efficiency improvement. More precisely, it allows us to avoid using costly authenticated shares (towards P_3) for the computed (shared) garbled circuit. Instead, our distributed garbling scheme outputs, even in the malicious setting, a plain two-out-of-two sum sharing of the garbled circuit.

Outline. In Section 6.2 we cover preliminary topics. In Section 6.3 we describe our two-party distributed garbling scheme, and in Section 6.4 we discuss our three party protocol. In Section 6.5 we show how to instantiate the various two-party functionalities we utilize in our protocols and in Section 6.6 we compare our protocol with prior work.

6.2 Preliminaries

Circuit notation. In this Chapter we follow the circuit notation of Bellare et al. [56]. Let $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$ be a circuit, where n is the number of input wires, m is the number of output wires, and q is the number of gates, where each gate is indexed by its output wire. Thus, the total number of wires in the circuit is $\mathsf{n} + \mathsf{q}$. The numbering of wires starts with the inputs and ends with the outputs; i.e., we have inputs $\{1, \ldots, \mathsf{n}\}$ and outputs $\{\mathsf{n} + \mathsf{q} - \mathsf{m} + 1, \ldots, \mathsf{n} + \mathsf{q}\}$. The function L (resp., R) takes as input a gate index and returns the left (resp., right) input wire to the gate. We require $L(\gamma) < R(\gamma) < \gamma$ for any gate index γ. The function G encodes the functionality of a given gate, e.g., $G_\gamma(0, 1) = 0$ if the gate with index γ is an AND gate. Because we consider circuits with inputs from multiple parties, let $\{\mathsf{n}_{i-1} + 1, \ldots, \mathsf{n}_i\}$ denote the input wires "controlled" by party P_i, with $\mathsf{n}_0 = 0$.

We denote *input gates* as those gates with one or more input wires, *inner gates* as those gates with no input or output wires, and *output gates* as those gates with an output wire.

Secret sharing. Our constructions use two-out-of-two secret sharing. In the semi-honest setting, we use a standard (linear) sharing of strings: the secret $x \in \{0,1\}^*$ is split into two random *summands* x_1 and x_2 such that $x_1 \oplus x_2 = x$, with P_i holding the summand x_i. We denote the *sharing* of x by $[x] = ([x]^{(1)}, [x]^{(2)})$, where we refer to each $[x]^{(i)} = x_i$ as P_i's *share* of x. This sharing is linear: If $[x]$ and $[y]$ are sharings of x and y respectively, then $[x] \oplus [y]$ is a sharing of $x \oplus y$; that is, $[x \oplus y] = [x] \oplus [y]$ and thus P_i can locally compute its share as $[x \oplus y]^{(i)} = [x]^{(i)} \oplus [y]^{(i)}$.

It is straight-forward to verify that the above secret-sharing is *private* provided that the summands x_1 and x_2 are uniformly chosen (restricted only on $x_1 \oplus x_2 = x$); i.e., any single share $[x]^{(i)}$ contains no information about the secret x. Reconstructing a sharing $[x]$ is easily done by having each party announce its share $[x]^{(i)}$ and taking x to be the exclusive-or of the announced shares.

Our protocols use shares of two types of secrets: κ-bit strings $x \in \{0,1\}^\kappa$ and bits $b \in \{0,1\}$. For clarity in the presentation, we use the bracket notation introduced above for sharings of $x \in \{0,1\}^\kappa$, and use the notation $\langle \cdot \rangle$ for sharings of bits; i.e., if $b \in \{0,1\}$ then a sharing of b is denoted as $\langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)})$.

In the malicious setting we need the sharings of bits to be *authenticated*; i.e., in addition to its summand b_i, each party P_i holds an authentication tag t_i for a Message Authentication Code (MAC), with another party P_j holding the corresponding verification key k_j. More precisely, in a sharing $\langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)})$ of b, each party's share is now a tuple $\langle b \rangle^{(i)} := (b_i, t_i, k_j)$, where $b_1 \oplus b_2 = b$, and t_i is a valid MAC on b_i with key k_j. This ensures that the adversary cannot make the reconstruction output any value other than the secret b. In particular, to reconstruct some sharing $\langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)})$, each party P_i first announces its summand b_i and the corresponding authentication tag t_i; subsequently, each party P_i checks that the other party P_j announced a validly authenticated summand matching its own verification key and if this is not the case it rejects. The inability of an adversarial P_i to announce a summand other than b_i follows from the unforgeability of the MAC, as P_i does not know the key k_j matching its authentication tag.

We also assume this authentication is linear in the following sense: Given

$\langle b \rangle$ and $\langle b' \rangle$, the parties can compute $\langle b \rangle \oplus \langle b' \rangle$ *locally*. Namely, $\langle b \rangle \oplus \langle b' \rangle = (\langle b \oplus b' \rangle^{(1)}, \langle b \oplus b' \rangle^{(2)})$, where $\langle b \oplus b' \rangle^{(i)} = (b_i \oplus b'_i, t_i \oplus t'_i, k_j \oplus k'_j)$ is a valid authentication. Such an authenticated sharing can be constructed using known techniques; see Section 6.5.

6.3 Two-Party Distributed Garbling Scheme

In this Section we describe our construction of a two-party distributed garbling scheme. Our protocol combines garbled circuits (cf. Chapter 2)with distributed garbling ideas from Damgård and Ishai [68]. The main idea is the following: The players jointly compute a garbled circuit, where the gates are garbled by use of a distributed encryption scheme which takes, for each encryption, one label from each party.

Distributing the Garbling Scheme Between Two Parties

Consider the garbling scheme in Chapter 2 with the following change: Each label $X_{w,b}$ consists of two *sub-labels* $s^1_{w,b}$ and $s^2_{w,b}$; that is, $X_{w,b} = (s^1_{w,b}, s^2_{w,b})$. We now show how to emulate this garbling scheme between two parties in the *semi-honest* setting. We assume the parties have access to the following two-party ideal functionalities:

- *Gate computation* $\mathcal{F}^G_{\mathbf{gate}}(\langle a \rangle, \langle b \rangle)$: The functionality takes as input sharings $\langle a \rangle$ and $\langle b \rangle$ of bits a and b, respectively, and is parameterized by a binary gate G; it outputs a sharing $\langle G(a, b) \rangle$ of the output of G on input (a, b).

- *One-out-of-two oblivious secret sharing* $\mathcal{F}^i_{\mathbf{oshare}}(\langle b \rangle, m_0, m_1)$: The functionality

takes as input a sharing $\langle b \rangle$ of a bit b (i.e., each party inputs its share), along with two messages m_0, m_1 from P_i, and outputs a random two-out-of-two sharing $[m_b]$ of m_b.

- *Constant bit sharing* $\mathcal{F}^b_{\mathbf{const}}()$: The functionality is parameterized by a bit $b \in \{0,1\}$, and outputs a random sharing $\langle b \rangle$ of b.

- *Random bit sharing* $\mathcal{F}_{\mathbf{rand}}()$: The functionality chooses a random bit $r \in_R \{0,1\}$ and computes and outputs a random sharing $\langle r \rangle$ of r.

- *Bit secret sharing* $\mathcal{F}^i_{\mathbf{ss}}(b)$: The functionality takes input bit $b \in \{0,1\}$ from P_i and outputs a random two-out-of-two sharing $\langle b \rangle$ of b.

Each of these can be instantiated efficiently in the semi-honest setting; see Section 6.5 for details.

Distributed encryption scheme. We utilize the distributed encryption scheme of Damgård and Ishai [68]. Suppose the message and the label for the encryption scheme are distributed as follows:

- The message m is secret-shared; i.e., P_1 and P_2 hold $[m]^{(1)}$ and $[m]^{(2)}$, respectively.

- The label $X = (s^1, s^2)$ is distributed such that P_1 and P_2 hold s^1 and s^2, respectively.

The encryption of the secret-shared message m with tweak T under label $X =$

(s^1, s^2) is:

$$\mathsf{Enc}_X^T([m]) = \left(\mathsf{Enc}^1_{s^1,T}\left([m]^{(1)}\right), \mathsf{Enc}^2_{s^2,T}\left([m]^{(2)}\right)\right) = \left([m]^{(1)} \oplus F^1_{s^1}(T), [m]^{(2)} \oplus F^1_{s^2}(T)\right),$$

where F^1_k is a PRF keyed by key k. To decrypt a ciphertext $c := \mathsf{Enc}_X^T(m)$, each party P_i sends its sub-label s^i to the decrypter, who uses them to recover the shares of m and reconstruct m.

Double encryption is defined analogously. For labels $X_\alpha = (s^1_\alpha, s^2_\alpha)$ and $X_\beta = (s^1_\beta, s^2_\beta)$, where P_i holds (s^i_α, s^i_β), encryption with tweak T works as follows:

$$\mathsf{Enc}^T_{X_\alpha, X_\beta}([m]) = \left([m]^{(1)} \oplus F^1_{s^1_\alpha}(T) \oplus F^2_{s^1_\beta}(T),\ [m]^{(2)} \oplus F^1_{s^2_\alpha}(T) \oplus F^2_{s^2_\beta}(T)\right).$$

Distributed garbling scheme. We now give a high-level description of our two-party distributed garbling scheme $\Pi_{\mathbf{GC}}(P_1, P_2)$. For each wire w in the circuit we associate labels $X_{w,0} = (s^1_{w,0}, s^2_{w,0})$ and $X_{w,1} = (s^1_{w,1}, s^2_{w,1})$ corresponding to bits '0' and '1', respectively. Each sub-label is only known to one of the two parties; i.e., P_i only knows $(s^i_{w,0}, s^i_{w,1})$. Each wire is also associated with a mask bit λ_w which is secret shared between the two parties such that no party knows λ_w.

Consider gate G_γ in the circuit with input wires indexed by α and β. We construct an array containing four rows corresponding to a random permutation of the four possible outcomes of gate G_γ applied to bits b_α and b_β. However, in the distributed case neither party should know what is being encrypted. Recall that for standard garbled circuits, the garbler can easily compute $G_\gamma(\lambda_\alpha \oplus b_\alpha, \lambda_\beta \oplus b_\beta)$ to

construct the array. However, in the distributed setting, neither party knows (and should *not* know) λ_α or λ_β. Thus, the parties utilize the $\mathcal{F}_{\mathbf{gate}}$ functionality, which takes as input the shares $\langle\lambda_\alpha\rangle \oplus \langle b_\alpha\rangle$ and $\langle\lambda_\beta\rangle \oplus \langle b_\beta\rangle$, and computes a sharing of $G_\gamma(\lambda_\alpha \oplus b_\alpha, \lambda_\beta \oplus b_\beta)$. Let $\langle\sigma_{\gamma,b_\alpha,b_\beta}\rangle = \mathcal{F}^G_{\mathbf{gate}}(\langle b_\alpha\rangle \oplus \langle\lambda_\alpha\rangle, \langle b_\beta\rangle \oplus \langle\lambda_\beta\rangle) \oplus \langle\lambda_\gamma\rangle$. The value $\sigma_{\gamma,b_\alpha,b_\beta}$ denotes which label to encrypt; that is, in row (b_α, b_β) we encrypt label $X_{\gamma,\sigma_{\gamma,b_\alpha,b_\beta}}$. However, we must still enforce that neither party knows what label $X_{\gamma,\sigma_{\gamma,b_\alpha,b_\beta}}$ represents. We handle this by utilizing another functionality, $\mathcal{F}_{\mathbf{oshare}}$. For each of the four $\sigma_{\gamma,b_\alpha,b_\beta}$ values, and for each party P_i, the parties compute $\mathcal{F}^i_{\mathbf{oshare}}(\langle\sigma_{\gamma,b_\alpha,b_\beta}\rangle, s^i_{\gamma,0}, s^i_{\gamma,1})$. This produces a share of the appropriate sub-label for party P_i, with the crucial fact that P_i does not know which of his sub-labels was shared. The results of $\mathcal{F}_{\mathbf{oshare}}$ are used as the shares to be encrypted.

See below for the full description.

Auxiliary Inputs: Security parameter k, circuit $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) := C$.

Parties P_1 and P_2 generate $\langle 1\rangle \leftarrow \mathcal{F}^1_{\mathbf{const}}$, which they use throughout the protocol.

1. **Generate mask bits:**
   - *Generate masks for P_1's inputs*: For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates $\lambda_w \in_R \{0,1\}$ and computes $\langle\lambda_w\rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(\lambda_w)$.
   - *Generate masks for P_2's inputs*: For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$: P_2 generates $\lambda_w \in_R \{0,1\}$ and computes $\langle\lambda_w\rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(\lambda_w)$.
   - *Generate masks for inner wires*: For $w \in \{\mathsf{n}+1, \ldots, \mathsf{n}+\mathsf{q}-\mathsf{m}\}$: generate $\langle\lambda_w\rangle \leftarrow \mathcal{F}_{\mathbf{rand}}$.
   - *Generate masks for output wires*: For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: generate $\langle\lambda_w\rangle \leftarrow \mathcal{F}^0_{\mathbf{const}}$.[a]
2. **Generate sub-labels:**
   - For $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}\}$ and $b \in \{0,1\}$: P_i generates sub-labels $s^i_{w,b} \in_R \{0,1\}^\kappa$.
3. **Construct garbled circuit:**
   - For $\gamma \in \{\mathsf{n}+1, \ldots, \mathsf{n}+\mathsf{q}\}$:

Let $\alpha := L(\gamma)$ and $\beta := R(\gamma)$ be the indices of the left and right input wires, respectively, of the gate indexed by γ. Compute the following selector bits:

$$\begin{aligned}
&\langle\sigma_{\gamma,0,0}\rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle\lambda_\alpha\rangle, \langle\lambda_\beta\rangle) \oplus \langle\lambda_\gamma\rangle && \langle\sigma_{\gamma,0,1}\rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle\lambda_\alpha\rangle, \langle\lambda_\beta\rangle \oplus \langle 1\rangle) \oplus \langle\lambda_\gamma\rangle \\
&\langle\sigma_{\gamma,1,0}\rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle\lambda_\alpha\rangle \oplus \langle 1\rangle, \langle\lambda_\beta\rangle) \oplus \langle\lambda_\gamma\rangle && \langle\sigma_{\gamma,1,1}\rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle\lambda_\alpha\rangle \oplus \langle 1\rangle, \langle\lambda_\beta\rangle \oplus \langle 1\rangle) \oplus \langle\lambda_\gamma\rangle.
\end{aligned}$$

Next, compute sharings of the appropriate sub-labels to use for each row:

$$\begin{aligned}
&\left[\hat{s}^1_{\gamma,0,0}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle\sigma_{\gamma,0,0}\rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), && \left[\hat{s}^2_{\gamma,0,0}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle\sigma_{\gamma,0,0}\rangle, s^2_{\gamma,0}, s^2_{\gamma,1}) \\
&\left[\hat{s}^1_{\gamma,0,1}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle\sigma_{\gamma,0,1}\rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), && \left[\hat{s}^2_{\gamma,0,1}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle\sigma_{\gamma,0,1}\rangle, s^2_{\gamma,0}, s^2_{\gamma,1}) \\
&\left[\hat{s}^1_{\gamma,1,0}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle\sigma_{\gamma,1,0}\rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), && \left[\hat{s}^2_{\gamma,1,0}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle\sigma_{\gamma,1,0}\rangle, s^2_{\gamma,0}, s^2_{\gamma,1}) \\
&\left[\hat{s}^1_{\gamma,1,1}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle\sigma_{\gamma,1,1}\rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), && \left[\hat{s}^2_{\gamma,1,1}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle\sigma_{\gamma,1,1}\rangle, s^2_{\gamma,0}, s^2_{\gamma,1}).
\end{aligned}$$

Finally, compute the distributed encryptions of the (permuted) sub-labels and selector bits. That is, letting $K_{w,b} = (s^1_{w,b}, s^2_{w,b})$, compute:

$$\begin{aligned}
P[\gamma,0,0] = (P^1[\gamma,0,0], P^2[\gamma,0,0]) &:= \mathsf{Enc}^{\gamma\|0\|0}_{K_{\alpha,0},K_{\beta,0}}(\left[\hat{s}^1_{\gamma,0,0}\right] \| \left[\hat{s}^2_{\gamma,0,0}\right] \|\langle\sigma_{\gamma,0,0}\rangle), \\
P[\gamma,0,1] = (P^1[\gamma,0,1], P^2[\gamma,0,1]) &:= \mathsf{Enc}^{\gamma\|0\|1}_{K_{\alpha,0},K_{\beta,1}}(\left[\hat{s}^1_{\gamma,0,1}\right] \| \left[\hat{s}^2_{\gamma,0,1}\right] \|\langle\sigma_{\gamma,0,1}\rangle), \\
P[\gamma,1,0] = (P^1[\gamma,1,0], P^2[\gamma,1,0]) &:= \mathsf{Enc}^{\gamma\|1\|0}_{K_{\alpha,1},K_{\beta,0}}(\left[\hat{s}^1_{\gamma,1,0}\right] \| \left[\hat{s}^2_{\gamma,1,0}\right] \|\langle\sigma_{\gamma,1,0}\rangle), \\
P[\gamma,1,1] = (P^1[\gamma,1,1], P^2[\gamma,1,1]) &:= \mathsf{Enc}^{\gamma\|1\|1}_{K_{\alpha,1},K_{\beta,1}}(\left[\hat{s}^1_{\gamma,1,1}\right] \| \left[\hat{s}^2_{\gamma,1,1}\right] \|\langle\sigma_{\gamma,1,1}\rangle).
\end{aligned}$$

4. **Output circuit:**
   - Let $\widehat{C}^i := (\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, P^i)$ and let $SK^i := \left\{(s^i_{w,0}, s^i_{w,1}) : w \in \{1, \ldots, \mathsf{n}\}\right\}$.
   - P_1 outputs the tuple $\left(\widehat{C}^1, SK^1, \left\{(\langle b_w\rangle^{(1)}, \langle\lambda_w\rangle^{(1)}, b_w, \lambda_w) : w \in \{1, \ldots, \mathsf{n}_1\}\right\}\right)$.
   - P_2 outputs the tuple $\left(\widehat{C}^2, SK^2, \left\{(\langle b_w\rangle^{(2)}, \langle\lambda_w\rangle^{(2)}, b_w, \lambda_w) : w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}\right\}\right)$.

[a] Note that we do not in fact need to create 'zero' masks for the output wires; we include this step mainly for ease of presentation.

Achieving Malicious Security

The semi-honest distributed garbling scheme described above can be directly adapted to work against a malicious adversary by modifying the hybrid functionalities to work in an authenticated manner; namely, we use authenticated sharings in place of standard secret sharings:

- $\mathcal{F}^1_{\mathbf{const}}()$ and $\mathcal{F}_{\mathbf{rand}}()$: The output share is authenticated.
- $\mathcal{F}^G_{\mathbf{gate}}(\langle a\rangle, \langle b\rangle)$: The inputs and outputs are all authenticated sharings.

- $\mathcal{F}^i_{\mathbf{oshare}}(\langle b\rangle, m_0, m_1)$: The selection bit b is an authenticated sharing.

- $\mathcal{F}^i_{\mathbf{ss}}(b)$: The output is an authenticated sharing of b.

See Section 6.5 for the detailed functionalities and Section 6.5.2 for their instantiations.

We also need to define a notion of *encrypting* authenticated shares. Recall that for an authenticated share $\langle b\rangle = (\langle b\rangle^{(1)}, \langle b\rangle^{(2)})$, we have $\langle b\rangle^{(i)} = (b_i, t_i, k_j)$, where party P_i holds b_i and t_i and party P_j holds k_j. Thus, letting $X = (s^1, s^2)$, we define

$$\mathsf{Enc}^T_X(\langle b\rangle) = (\mathsf{Enc}^1_{s^1,T}(b_1\|t_1\|k_1), \mathsf{Enc}^2_{s^2,T}(b_2\|t_2\|k_2)).$$

On decryption, each party's ciphertext is decrypted and the authenticity of b_1 and b_2 are verified using the (encrypted) tags and labels. Thus, when evaluating a garbled circuit, the party checks the authenticity of the share from the decrypted row of each garbled gate; if the check fails, the party aborts.

Note that we can convert this protocol into a maliciously-secure 2PC protocol, which we denote as $\Pi_{\mathbf{2PC}}(P_1, P_2)$, as follows.

Auxiliary Inputs: Security parameter k, circuit $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) := C$.
Inputs: For $w \in \{1, \ldots, \mathsf{n}_1\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$, P_2 has inputs b_w.

1. The parties execute $\Pi_{\mathbf{GC}}(P_1, P_2)$.
2. For $w \in \{1, \ldots, \mathsf{n}_1\}$: The parties execute $\langle b_w\rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(b_w)$.
3. For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: The parties execute $\langle b_w\rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(b_w)$.
4. P_1 sends GC^1 to P_2.
5. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sends $(s^1_{w,b_w\oplus\lambda_w}, \langle b_w\rangle^{(1)} \oplus \langle\lambda_w\rangle^{(1)})$ to P_2 who reconstructs $b_w \oplus \lambda_w$ locally.
6. For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$: P_2 sends $\langle b_w\rangle^{(2)} \oplus \langle\lambda_w\rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_w$ locally. P_1 then sends $s^1_{w,b_w\oplus\lambda_w}$ to P_2.

7. P_2 evaluates the garbled circuit using the labels $(s^1_{w,b_w\oplus\lambda_w}, s^2_{w,b_w\oplus\lambda_w})$ and selector bits $b_w \oplus \lambda_w$, for $w \in \{1, \ldots, \mathsf{n}\}$.

Theorem 6.1. *Let C be an arbitrary polynomial-size circuit. Then the protocol $\Pi_{\mathbf{2PC}}(P_1, P_2)$, using authenticated hybrids, securely evaluates the circuit C in the presence of a (static) malicious adversary in the $(\mathcal{F}_{\mathbf{const}}, \mathcal{F}_{\mathbf{gate}}, \mathcal{F}_{\mathbf{oshare}}, \mathcal{F}_{\mathbf{rand}}, \mathcal{F}_{\mathbf{ss}})$-hybrid world.*

Proof. We construct simulators $\mathcal{S}_1$ and $\mathcal{S}_2$ which simulate an adversary corrupting P_1 and P_2, respectively. We note that we can ignore the negligible probability difference due to a direct attack on the authentication mechanism by a simple hybrid argument.

Malicious P_1. We simulate adversary $\mathcal{A}$ corrupting party P_1 as follows. The simulator $\mathcal{S}_1$ chooses $0^{\mathsf{n}-\mathsf{n}_1}$ as P_2's input and then runs exactly as P_2 would. In addition, $\mathcal{S}_1$ extracts $\mathcal{A}$'s input x through the calls to $\mathcal{F}^1_{\mathbf{ss}}$ and passes x to the trusted party.

As $\mathcal{S}_1$ acts exactly as P_2 does (albeit on a different input), and $\mathcal{A}$ receives no output, we need only show that the protocol aborts with equal probability across the two views. Note that $\mathcal{A}$ has three possible places in the protocol in which it can try to force the protocol to abort:

1. Sending an invalid sub-label in Step 5 or Step 6 of $\Pi_{\mathbf{2PC}}$,
2. Inputting invalid or flipped sub-labels into the calls to $\mathcal{F}_{\mathbf{oshare}}$, or
3. Encrypting the incorrect sub-labels shares or using some arbitrary string as encryptions.

We claim that the probability of aborting due to any of the above attacks is *independent* of P_2's input. Clearly, if $\mathcal{A}$ sends invalid sub-labels for its own input wires, the probability of aborting is independent of P_2's input. In the case that $\mathcal{A}$ sends an invalid sub-label in Step 6 of $\Pi_{\mathbf{2PC}}$, the probability of aborting is independent of P_2's input due to the masking by the (uniformly chosen) mask bit.

Now consider the case where $\mathcal{A}$ corrupts t rows in a given garbled gate. Note that even though $\mathcal{A}$ can control *which* rows in the garbled gate table to corrupt, the probability that any given row is hit during evaluation is exactly $1/4$ (by the security of the point-and-permute method). Thus, the probability that a given bad row is hit is $t/4$, independent of the bits on the incoming wires into the gate. Thus, as the probability of aborting is independent of P_2's input, the two views are perfectly indistinguishable.

Malicious P_2. We simulate adversary $\mathcal{A}$ corrupting party P_2 as follows. The simulator $\mathcal{S}_2$ selects 0^{n_1} as the input of P_1, and then proceeds to act as P_1 in $\Pi_{\mathbf{GC}}$. Then, $\mathcal{S}_2$ extracts $\mathcal{A}$'s input y through the calls to $\mathcal{F}_{\mathbf{ss}}^2$ and passes y to the trusted party, receiving back $f(x, y)$. Now, $\mathcal{S}_2$ continues executing as P_1, except it modifies its share for the output of $\mathcal{F}_{\mathbf{gate}}$ on the output gates so that the selector bits for all rows in the output gates contain the appropriate bit from $f(x, y)$. In more detail, right before sending its share GC^1 at Step 4 of the protocol, $\mathcal{S}_2$ does the following:

- For each output gate G_γ, let z_γ denote the correct output (i.e., the appropriate bit from $f(x, y)$) of the gate. Now, for each of the four rows of this gate, $\mathcal{S}_2$ modifies the original authenticated sharing $(\langle z_{\gamma,1}\rangle^{(1)}, \langle z_{\gamma,2}\rangle^{(2)})$ into a new

sharing $(\langle z^*_{\gamma,1}\rangle^{(1)}, \langle z_{\gamma,2}\rangle^{(2)})$ that would be reconstructed to z_γ. Note that this is possible, since $\mathcal{S}_2$ emulates the $\mathcal{F}_{\mathbf{gate}}$ functionality and has all the information necessary to construct new authenticated shares. In addition, $\mathcal{S}_2$ modifies the corresponding encryption in the garbled gate accordingly.

$\mathcal{S}_2$ continues executing as P_1, and outputs whatever $\mathcal{A}$ outputs.

We now prove that the view of the adversary when communicating with $\mathcal{S}_2$ versus the view when communicating with a real P_1 is computationally indistinguishable. We show this by constructing a set of hybrids and proving indistinguishability between them.

$\mathbf{H_0}$. The same as the execution with P_1.

$\mathbf{H_1}$. The same as $\mathbf{H_0}$, except the output of $\mathcal{F}_{\mathbf{gate}}$ in each output gate is modified to be equal to an authenticated sharing of the correct output from $f(x, y)$.

Indistinguishability follows from the security of the underlying garbling scheme; the only difference here is that $\mathcal{A}$ can try to force the protocol to abort. However, by the security of the authenticated bit sharing scheme, the output of $\mathcal{F}_{\mathbf{gate}}$ towards $\mathcal{A}$ provides no information about the underlying selector bit's value, and thus $\mathcal{A}$ acts independently of the value $\sigma_{\gamma,i,j}$.

$\mathbf{H_2}$. The same as $\mathbf{H_1}$, except input 0^{n_1} is used instead of P_1's real input.

Indistinguishability holds by the use of $\mathcal{F}_{\mathbf{ss}}$.

As $\mathbf{H_2}$ is the same as the simulator, the proof is complete. □

6.4 Three-Party Computation from Cut-and-Choose

We can directly adapt the distributed garbling scheme to work over multiple parties, and thus construct a 3PC scheme; however, in this case the underlying functionalities need to support multiple parties rather than just two parties and are thus unlikely to be efficient in practice. Thus, in this Section we show how to utilize the maliciously secure two-party distributed garbling scheme from Section 6.3 to construct a maliciously secure *three*-party secure computation protocol, using almost entirely two-party constructs (the only three-party functionality needed is that of coin-tossing).

We first cover preliminary notions, such as the ideal functionalities we need. Then, we show how to adapt a combination of two existing cut-and-choose protocols [8, 9] to the three-party setting. Finally, we use this "generic" protocol to show how to adapt Lindell's "fast cut-and-choose" protocol [10] to the three-party setting. The cost of each of these three-party protocols is roughly *eight times* the computational cost of the underlying two-party protocol they are based on, and roughly *sixteen times* the communication cost (plus the cost of a small number of OTs per gate, which can be efficiently amortized using OT extension [7, 33]), and thus we show that we can achieve efficient secure three-party computation at only a small factor of the cost of the most efficient garbled circuit-based protocol.

Preliminaries

Ideal functionalities. In addition to the ideal functionalities used in the two-party distributed garbling scheme, we need the following additional (maliciously secure) functionalities:

- *Three-party coin-flipping* $\mathcal{F}_{\mathbf{cf}}()$: The functionality outputs a random bitstring to each party.
- *One-out-of-two oblivious transfer* $\mathcal{F}_{\mathbf{ot}}^{i,j}(b, m_0, m_1)$: The functionality takes as input a choice bit b from party P_i and messages m_0, m_1 from P_j, and outputs m_b to party P_i.
- *ZKPoK of extended Diffie-Hellman tuple* $\mathcal{F}_{\mathbf{zkpok}}^{i,j}(a, (g, h_0, h_1, \{u_i, v_i\}_i))$: The functionality takes as input a from party P_i, and tuple $(g, h_0, h_1, \{u_i, v_i\}_i)$ from party P_j, and outputs 1 to party P_j if either all tuples in $\{(g, h_0, u_i, v_i)\}_i$ are Diffie-Hellman tuples with $h_0 = g^a$ or all tuples in $\{(g, h_1, u_i, v_i)\}_i$ are Diffie-Hellman tuples with $h_1 = g^a$, and 0 otherwise.

These can all be efficiently instantiated in a standard fashion. We can implement $\mathcal{F}_{\mathbf{cf}}$ in the random oracle model using three commitments and openings. The $\mathcal{F}_{\mathbf{ot}}$ functionality can be instantiated using any maliciously secure OT implementation, such as the construction by Peikert et al. [48]. Likewise, $\mathcal{F}_{\mathbf{zkpok}}$ can be efficiently instantiated using existing protocols [9, Section B].

Distributed garbled circuits for three parties. Note that the garbling protocol $\Pi_{\mathbf{GC}}$ described before only garbles a circuit containing inputs from two parties. We

can easily adapt this to support input from a third (external) party as follows. Let $\Pi'_{\mathbf{GC}}(P_1, P_2)$ be the same as $\Pi_{\mathbf{GC}}(P_1, P_2)$ except for the following modifications:

- All of the operations over P_2's input now operate over wires $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$.
- In Step 1, we add the following sub-step for generating shares for P_3's input wires:
  - For $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$: generate $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{\mathbf{rand}}$.
- In Step 4, party P_i outputs $\left\{\langle \lambda_w \rangle^{(i)} : w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}\right\}$ in addition to his normal outputs.

Achieving Malicious Security for Three Parties

Note that our two-party distributed garbling scheme has the property that if at most one of the two parties is corrupt, the garbling of circuit C either correctly evaluates C on P_1's and P_2's inputs, or causes the evaluator to abort. That is, a malicious party cannot "alter" the garbling to evaluate some circuit other than C. Now, if both P_1 and P_2 are corrupt, they can of course garble an arbitrary circuit. This suggests the following approach to three-party computation: If either P_1 or P_2 are honest, we need only construct a single garbled circuit, which is sent to P_3 to be evaluated. To cover the case where both P_1 and P_2 are corrupt, we use cut-and-choose to prevent P_3 from evaluating a maliciously constructed circuit. In what follows, we utilize existing cut-and-choose protocols from the literature [8, 9], and "plug in" our distributed garbling scheme as necessary. Thus, security mostly

follows from the security proofs of the underlying cut-and-choose protocols. We also show how we can use this protocol in an adaptation of Lindell's protocol [10] to the three-party setting.

The basic intuition for security is as follows. Cut-and-choose is used to prevent P_3 from evaluating maliciously constructed circuits when both P_1 and P_2 are malicious. For the case where either P_1 or P_2 is honest, $\Pi'_{\mathbf{GC}}(P_1, P_2)$ assures us that the garbled circuit constructed between P_1 and P_2 is either correctly constructed or causes P_3 to abort (independent of any party's input).

Protocol description. We now give a high-level description of our protocol.

1. The parties first replace the input circuit C^0 with a circuit C, where the only difference is each of P_3's input wires is replaced by an XOR of ρ new input wires, preventing either party P_1 or P_2 from launching a selective failure attack on P_3's input choices.

2. P_1 and P_2 generate the required commitments needed for input consistency, as is done in the protocol of Lindell and Pinkas [9].

3. P_1 and P_2 construct ρ garbled circuits using $\Pi'_{\mathbf{GC}}$ and the input sub-labels generated as is done in the protocol of Lindell and Pinkas [9].

4. P_1 and P_2 compute authenticated sharings (between each other; P_3 is not involved here) of their input bits.

5. P_1 and P_2 both run (separately) an OT protocol with P_3 for each of P_3's input wires, where P_1/P_2 input their sub-labels and P_3 chooses based on his input.

(Note that any cheating by P_1/P_2 here will be caught with high probability by the cut-and-choose step below.) Thus, P_3 now has labels for each of his input bits.

6. P_1 and P_2 send the (distributed) garbled circuits, along with the input consistency commitments, to P_3.

7. All three parties run a coin-tossing protocol to determine which circuits for P_3 to open and which to evaluate.

8. For the evaluation circuits, P_1 and P_2 send the sub-labels and selector bits for their inputs to P_3. Note that we need to be careful in this step, as we need to enforce that, for example, P_1 uses the same input as was shared in Step 2 above. This is accomplished as follows. Recall that P_1 and P_2 have sharings of each other's inputs and mask bits, all of which are authenticated. Thus, P_1 can send the (authenticated) share of its masked input to P_2, who can verify its authenticity, and thus reconstruct the masked input bit using its own share. This allows an honest P_2 to send the correct sub-label (correct in the sense that it corresponds to P_1's input shared in Step 2) to P_3, even with a malicious P_1.

9. For the check circuits, P_1 and P_2 send the required information for P_3 to decrypt the check circuits and verify correctness. If any of these check circuits are incorrectly constructed, P_3 aborts; otherwise, it has high confidence that the majority of the evaluation circuits are correctly constructed.

10. For the evaluation circuits, P_3 checks for input consistency against the sub-labels sent by P_1 and P_2 in Step 8 using a zero-knowledge proof-of-knowledge protocol [9], aborting on any inconsistency.

11. Finally, P_3 evaluates the evaluation circuits, outputting the majority over the circuits' output.

See below for the full protocol description.

Auxiliary Inputs: Security parameter κ, statistical security parameter ρ, circuit C^0, cyclic group $\mathbb{G}$ with (prime) order q and generator g, and randomness extractor Ext.

Inputs: For $w \in \{1, \ldots, \mathsf{n}_1\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$, P_3 has inputs b_w.

1. Each party replaces C^0 with a circuit C where each of P_3's input wires is replaced by an exclusive-or of ρ new input wires. We let $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) := C$, and denote P_3's new inputs by $\hat{b}_w$.
2. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates $a^1_{w,0}, a^1_{w,1} \in_R \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^1_{w,0}}), (w, 1, g^{a^1_{w,1}})\right\}$.
   For $w \in \{\mathsf{n}_1{+}1, \ldots, \mathsf{n}_2\}$: P_2 generates $a^2_{w,0}, a^2_{w,1} \in_R \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^2_{w,0}}), (w, 1, g^{a^2_{w,1}})\right\}$.
   For $j \in \{1, \ldots, \rho\}$: P_i, for $i \in \{1, 2\}$, generates $r^i_j \in_R \mathbb{Z}_q$ and constructs set $\left\{(j, g^{r^i_j})\right\}$.
   For $j \in \{1, \ldots, \rho\}$: P_1 and P_2 run up to Step 2 ("Generate sub-labels") of $\Pi^3_{\mathbf{GC}}(P_1, P_2)$, where the parties do the following in the jth iteration:
   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates sub-labels $s^1_{w,b\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^1_{w,b} \cdot r^1_j})$ for $b \in \{0, 1\}$.
   - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 generates sub-labels $s^2_{w,b\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^2_{w,b} \cdot r^2_j})$ for $b \in \{0, 1\}$.
   - All other sub-labels are generated in the normal fashion.
3. For $j \in \{1, \ldots, \rho\}$: P_1 and P_2 continue their executions of $\Pi^3_{\mathbf{GC}}(P_1, P_2)$, producing garbled circuit $\widehat{C}_j$.
4. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(b_w)$.
   For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(b_w)$.
5. For $j \in \{1, \ldots, \rho\}$ and $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$: P_1 and P_2 exchange $\langle \lambda_{w,j} \rangle$ with each other, reconstructing $\lambda_{w,j}$ locally. Both P_1 and P_2 send $\lambda_{w,j}$ to P_3.
   For $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$: P_i, for $i \in \{1, 2\}$, and P_3 run $\mathcal{F}_{\mathbf{ot}}$, with P_i as the sender inputting $\left(\left\{s^i_{w,\lambda_{w,j},j}\right\}_{j\in\{1,\ldots,\rho\}}, \left\{s^i_{w,\lambda_{w,j}\oplus 1,j}\right\}_{j\in\{1,\ldots,\rho\}}\right)$ and P_3 as the receiver inputting $\hat{b}_w$.

6. P_i, for $i \in \{1,2\}$, sends the sets constructed in Step 2, along with the garbled circuit $\left\{\widehat{C}_j^i\right\}_{i=1}^{\rho}$, to P_3.
7. The parties compute $r \leftarrow \mathcal{F}_{\mathbf{cf}}$. Let $\mathcal{CC} = \{i : r[i] = 1\}$, and let $\mathcal{EC} = \{1,\ldots,\rho\} \setminus \mathcal{CC}$.
8. For $j \in \mathcal{EC}$:
   - For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 sends $\langle b_w\rangle^{(1)} \oplus \langle \lambda_{w,j}\rangle^{(1)}$ to P_2, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
   - For $w \in \{\mathsf{n}_1{+}1,\ldots,\mathsf{n}\}$: P_2 sends $\langle b_w\rangle^{(2)} \oplus \langle \lambda_{w,j}\rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
9. For $j \in \mathcal{CC}$:
   - P_i, for $i \in \{1,2\}$, does the following:
     - Sends r_j^i to P_3, and P_3 checks that these values are consistent with the pairs $\left\{(j, g^{r_j^i})\right\}$ sent before.
     - For $w \in \{1,\ldots,\mathsf{n}\}$: Sends sub-labels $s^i_{w,0,j}$ and $s^i_{w,1,j}$, mask bit share $\lambda^{(i)}_{w,j}$, and the labels to the authenticated bits to P_3.
   - Given the above information, P_3 reconstructs all input labels and verifies they match with those labels sent previously. Also, using said labels, P_3 verifies that the garbled circuit is correctly constructed.
10. For $j \in \mathcal{EC}$:
    - For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 sends $g^{a^1_{w,b_w}\cdot r^1_j}$ to P_3, who computes $s^1_{w,b_w\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^1_{w,b_w}\cdot r^1_j})$.
    - For $w \in \{\mathsf{n}_1+1,\ldots,\mathsf{n}_2\}$: P_2 sends $g^{a^2_{w,b_w}\cdot r^2_j}$ to P_3, who computes $s^2_{w,b_w\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^2_{w,b_w}\cdot r^2_j})$.

    For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 and P_3 run $\mathcal{F}_{\mathbf{zkpok}}$, with P_1 as the prover inputting a^1_{w,b_w} and P_3 as the verifier inputting $\left(g, g^{a^1_{w,0}}, g^{a^1_{w,1}}, \left\{(g^{r^1_j}, g^{a^1_{w,b_w}\cdot r^1_j})\right\}_{j\in\mathcal{EC}}\right)$.

    For $w \in \{\mathsf{n}_1+1,\ldots,\mathsf{n}_2\}$: P_2 and P_3 run $\mathcal{F}_{\mathbf{zkpok}}$, with P_2 as the prover inputting a^2_{w,b_w} and P_3 as the verifier inputting $\left(g, g^{a^2_{w,0}}, g^{a^2_{w,1}}, \left\{(g^{r^2_j}, g^{a^2_{w,b_w}\cdot r^2_j})\right\}_{j\in\mathcal{EC}}\right)$.
11. For $j \in \mathcal{EC}$: P_3 evaluates circuit $\widehat{C}_j$ using $\left\{(s^1_{w,b_w\oplus\lambda_{w,j},j}, s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})\right\}_{w\in\{1,\ldots,\mathsf{n}\}}$ as inputs.

    P_3 outputs the majority output over the evaluated circuits.

Theorem 6.2. *Let C be an arbitrary polynomial-size circuit and let $\mathbb{G}$ be a cyclic group with prime order. Given access to ideal functionalities $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{gate}}$, $\mathcal{F}_{\mathbf{oshare}}$, $\mathcal{F}_{\mathbf{ot}}$, $\mathcal{F}_{\mathbf{rand}}$, and $\mathcal{F}_{\mathbf{ss}}$, and assuming that the decisional Diffie-Hellman problem is hard in $\mathbb{G}$, then $\Pi^m_{\mathbf{3PC}}(P_1,P_2,P_3)$ securely computes the circuit C in the presence of an adversary corrupting an arbitrary number of parties.*

Proof. The proof is similar to prior work in two-party garbling schemes based on cut-and-choose [8, 9, 32]. We make use of the following lemma:

Lemma. *Consider garbled gate G_γ with input wires α and β, and let $X_{w,b} = (s^1_{w,b}, s^2_{w,b}, b \oplus \lambda_w)$, for $w \in \{\alpha, \beta\}$ denote the valid labels. Fix a (valid) label $X_{w,b}$ for some fixed w and b. Let $\bar{X}_{w,b}$ be equal to $X_{w,b}$ except that two of the three components (i.e., the sub-labels and selector bit) are altered arbitrarily. Then using label $\bar{X}_{w,b}$ to decrypt G_γ causes a decryption failure with all but negligible probability.*

Proof. This follows directly from our encryption scheme and garbling scheme. □

Informally, what this lemma says is that for a given garbled gate, a sub-label / selector bit combo can only be used correctly on a single row of the garbled table, and modifying some (but not all) of the components results in a decryption failure; thus an adversary must change *both* the sub-label and permutation bit accordingly for the garbled gate to successfully decrypt. Note that in the two-party secure computation protocol described in Section 6.3 we enforce the above by authenticating all of the selector bits (thus preventing any malicious party from altering these). However, the authenticated sharing protocol only works between two parties, namely the parties doing the distributed garbling. Thus, we need a way for the evaluator to gain confidence in the sub-label / selector bit combos sent to him by P_1 and P_2. We do this by utilizing the Diffie-Hellman pseudorandom synthesizer trick of Lindell and Pinkas [9]. This enforces that P_1 and P_2 are consistent in the sub-label they send to P_3, and because at least one sub-label is correct, the adversary can at most cause P_3 to abort.

There are six possible (interesting) corruption cases. However, due to symmetries, we only need to consider four of them.

The adversary corrupts parties P_1 and P_2. We need to construct a simulator $\mathcal{S}$ with access to the adversary $\mathcal{A}$ (who controls P_1 and P_2) and a trusted third party which computes $f(x, y, z)$ given inputs x, y, and z. The simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and then runs as P_3 would until Step 10. Here, $\mathcal{S}$ uses the witnesses a^i_{w,b_w} send by P_1 and P_2 to $\mathcal{F}_{\mathbf{zkpok}}$ to extract their inputs. $\mathcal{S}$ then feeds these inputs to the trusted third party, receiving back $f(x, y, z)$. $\mathcal{S}$ continues to run as P_3 would, and halts, outputting whatever $\mathcal{A}$ outputs.

We now argue that the adversary's view in the real and ideal worlds are computationally indistinguishable. The proof closely follows existing work [9, pp. 17–21], and thus we only give some intuition here.

Note that if $\mathcal{A}$ tries to cheat in Step 5, it gets caught in the cut-and-choose step with high probability. Similarly, if $\mathcal{A}$ tries to send different labels in Step 8 (i.e., the "input inconsistency" attack), it gets caught in Step 10 when proving the consistency of the sub-labels sent.

The adversary corrupts parties P_1 and P_3. We again demonstrate a simulator, this time with $\mathcal{A}$ controlling parties P_1 and P_3. This is similar to the two-party case where P_2 is corrupted. The main challenge is that the simulator needs to construct "fake" garbled circuits in order for $\mathcal{A}$ to output the correct output; however, as shown in the proof of our two-party protocol, such a simulator exists. Thus, the simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2

would up until Step 6. $\mathcal{S}$ can extract both P_1's input x and its mask bits $\lambda_{w,j}$ through P_1's calls to $\mathcal{F}_{\mathbf{ss}}^1$ in Step 4 and Step 2. Likewise, in Step 5, $\mathcal{S}$ extracts P_3's input z through the calls to $\mathcal{F}_{\mathbf{ot}}$. $\mathcal{S}$ then passes x and z to the trusted third party, learning $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $r \in_R \{0,1\}^{3\rho}$. Then for $j \in \{1, \ldots, 3\rho\}$, $\mathcal{S}$ proceeds as follows: If $r[j] = 0$, $\mathcal{S}$ uses the simulator that is known to exist for the two-party circuit garbling protocol to construct garbled circuits which output $f(x, y, z)$. Otherwise, $\mathcal{S}$ acts as P_2 would. $\mathcal{S}$ continues to act as P_2 would, except that in Step 7 it sets the output of $\mathcal{F}_{\mathbf{cf}}$ to be equal to the ρ chosen above. Finally, $\mathcal{S}$ halts, outputting whatever $\mathcal{A}$ outputs.

The main intuition here is that, since P_1 learns nothing about the portion of the circuit garbled by P_2, this reduces to the two-party setting where P_2 is corrupt. Recall that by the security of our garbling protocol, P_1 can only construct circuits that cause the evaluator to abort. If P_1 tries to cheat in Step 5 by exchanging invalid mask shares, P_2 detects this with high probability, and likewise for Step 8.

The adversary corrupts parties P_2 and P_3. The analysis here is very similar to the case where parties P_1 and P_3 are corrupt.

The adversary corrupts party P_1. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_1 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 and P_3 would, extracting P_1's input x through the calls to $\mathcal{F}_{\mathbf{ss}}^1$ in Step 4. $\mathcal{S}$ passes x to the trusted third party, learning $f(x, y, z)$, and halts, outputting whatever $\mathcal{A}$ outputs.

As $\mathcal{S}$ acts exactly as P_2 and P_3 would, and $\mathcal{A}$ gets no output, we need only

show that the probability that $\mathcal{A}$ aborts in both the real and ideal world is identical. In fact, this follows from our maliciously secure two-party protocol and the security of the input-consistency checks.

The adversary corrupts party P_2. The analysis here is very similar to the case where party P_1 is corrupt.

The adversary corrupts party P_3. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_3 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_1 and P_2 would, extracting P_3's input z through the $\mathcal{F}_{\mathbf{ot}}$ calls in Step 5. $\mathcal{S}$ then hands z to the trusted third party, who returns $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $r \leftarrow\!\!\$\ \{0,1\}^{3\rho}$, and then for $j \in \{1, \ldots, 3\rho\}$ $\mathcal{S}$ proceeds as follows: If $r[j] = 0$, $\mathcal{S}$ constructs a distributed garbled circuit which outputs $f(x, y, z)$, otherwise $\mathcal{S}$ proceeds as normal. Then, in Step 7, $\mathcal{S}$ fixes the output of $\mathcal{F}_{\mathbf{cf}}$ to be ρ. For the rest of the protocol, $\mathcal{S}$ acts as P_1 and P_2 would, and eventually halts, outputting what $\mathcal{A}$ outputs.

The analysis is very similar to prior work [9, pp. 22–23]. □

Adapting Lindell's Protocol to the Three-Party Setting

The 3PC protocol described above has a replication factor of roughly 3×; namely, for statistical security parameter ρ, the actual probability of cheating is roughly $2^{-0.32\rho}$ [9]. Thus, for a desired error probability of 2^{-40} a total of 128 circuits need to be garbled. Recently, Lindell [10] showed a construction which removes this replication factor in the two-party setting; that is, for a cheating probability of $2^{-\rho}$ the sender needs to garble *only* ρ circuits. In this Section we show how to adapt

this protocol to the three-party setting.

Lindell's construction works in two phases. In the first phase, the parties do a standard cut-and-choose, with P_1 constructing ρ circuits (for error probability $2^{-\rho}$) and P_2 opening half of them. If, during evaluation, P_2 finds that two or more circuits have conflicting outputs, it stores these conflicting outputs as a "proof-of-cheating" ϕ. In the second phase, the parties run a circuit which takes as input from P_1 its original input x, and from P_2 the "proof-of-cheating" ϕ. If ϕ is a valid proof, then the circuit reveals x to P_2, who can then compute the output itself; otherwise P_2 gets no output. Thus, this second phase enforces that if P_1 cheated in the cut-and-choose then P_2 learns P_1's input.

To adapt this to the three-party setting, we proceed as follows. For the circuit in the first phase, we essentially just run $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$, with the same tweaks as are used by Lindell [10] (namely, the use of encoded output translation tables and doing circuit evaluation before circuit checking).

For the second phase circuit, we run into some issues, due to Lindell's scheme being inherently a "two-party" approach. Recall that this circuit is constructed in such a way that if P_2 receives any conflicting outputs when evaluating, he inputs these outputs as a "proof-of-cheating" in order to reveal P_1's input. At first glance, it appears this technique would *not* work in the three-party setting because in that case P_3 needs to learn *both* P_1's and P_2's inputs to reconstruct the output; however, it could be the case that only *one* of these two parties is cheating. Recall, however, that our distributed garbling scheme enforces that as long as *one* of the two parties is honest, the garbled circuits are "correct" in the sense that they either correctly

compute the desired circuit are cause a failure independent of any party's input. Thus, P_3 only finds mismatched outputs in the case where *both* P_1 and P_2 cheat, making it okay at this point to reveal both those parties' inputs in the second phase circuit.

Another issue arises in how this circuit is constructed. In Lindell's two-party scheme, P_1 hardwires the output labels into the circuit. In a naive adaptation to the three-party setting, both P_1 and P_2 would need to hardwire their output *sub*-labels into the circuit. However, this would allow each party to learn the others' sub-labels for the output, which leads to the following attack by a colluding P_1 and P_3: During the construction of the second phase circuit, P_1 learns P_2's output sub-labels, and he sends these, as well as his own output sub-labels, to P_3. Now, when P_3 evaluates the circuit, he can input conflicting outputs as his "proof-of-cheating" because he knows all of the outputs labels, thus allowing P_1 and P_3 to learn P_2's input. We can fix this by having the output sub-labels of P_1 and P_2 be inputs to the circuit, rather than hardcoded. However, this raises another issue, as P_3 cannot verify that the sub-labels input by P_1 and P_2 are the correct ones. Thus, we modify the circuit to output these sub-labels in the clear, allowing P_3 to do this check.

See below for the full protocol description.

Auxiliary Inputs: Security parameter κ, statistical security parameter ρ, circuit C^0, cyclic group $\mathbb{G}$ with (prime) order q and generator g, randomness extractor Ext, one-way function H.

Inputs: For $w \in \{1, \ldots, \mathsf{n}_1\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$, P_2 has inputs b_w; for $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$, P_3 has inputs b_w.

1. Each party replaces C^0 with a circuit C where each of P_3's input wires is replaced by an exclusive-or of ρ new input wires. We let $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) := C$, and denote P_3's

new inputs by $\hat{b}_w$.

2. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates $a^1_{w,0}, a^1_{w,1} \in_R \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^1_{w,0}}), (w, 1, g^{a^1_{w,1}})\right\}$.

   For $w \in \{\mathsf{n}_1{+}1, \ldots, \mathsf{n}_2\}$: P_2 generates $a^2_{w,0}, a^2_{w,1} \in_R \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^2_{w,0}}), (w, 1, g^{a^2_{w,1}})\right\}$.

   For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: P_i, for $i \in \{1,2\}$, generates $o^i_{w,0}, o^i_{w,1} \in_R \{0,1\}^\kappa$.

   For $j \in \{1, \ldots, \rho\}$: P_i, for $i \in \{1,2\}$, generates $r^i_j \in_R \mathbb{Z}_q$ and constructs set $\left\{(j, g^{r^i_j})\right\}$.

   For $j \in \{1, \ldots, \rho\}$: P_1 and P_2 run up to Step 2 ("Generate sub-labels") of $\Pi'_{\mathbf{GC}}(P_1, P_2)$, where the parties do the following:

   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates sub-labels $s^1_{w,b\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^1_{w,b}\cdot r^1_j})$ for $b \in \{0,1\}$.
   - For $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}_2\}$: P_2 generates sub-labels $s^2_{w,b\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^2_{w,b}\cdot r^2_j})$ for $b \in \{0,1\}$.
   - For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: P_i sets $s^i_{w,b\oplus\lambda_{w,j}} := o^i_{w,b}$.
   - All other sub-labels are generated in the normal fashion.

3. For $j \in \{1, \ldots, \rho\}$: P_1 and P_2 continue their executions of $\Pi'_{\mathbf{GC}}(P_1, P_2)$, producing (distributed) garbled circuit $GC_j := (GC^1_j, GC^2_j)$.

4. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(b_w)$.
   For $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}_2\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(b_w)$.

5. For $j \in \{1, \ldots, \rho\}$ and $w \in \{\mathsf{n}_2+1, \ldots, \mathsf{n}\}$: P_1 and P_2 exchange $\langle \lambda_{w,j} \rangle$ with each other, reconstructing $\lambda_{w,j}$ locally. Both P_1 and P_2 send $\lambda_{w,j}$ to P_3.
   For $w \in \{\mathsf{n}_2+1, \ldots, \mathsf{n}\}$: P_i, for $i \in \{1,2\}$, and P_3 run $\mathcal{F}_{\mathbf{ot}}$, with P_i as the sender inputting
   $\left(\left\{s^i_{w,\lambda_{w,j},j}\right\}_{j\in\{1,\ldots,\rho\}}, \left\{s^i_{w,\lambda_{w,j}\oplus 1,j}\right\}_{j\in\{1,\ldots,\rho\}}\right)$ and P_3 as the receiver inputting $\hat{b}_w$.

6. For $i \in \{1,2\}$: P_i sends the sets constructed in Step 2, along with the garbled circuit $\left\{\widehat{C}^i_j\right\}_{j=1}^{\rho}$, to P_3. In addition, P_i sends the *encoded output translation table* $\left\{(H(o^i_{w,0}), H(o^i_{w,1}))\right\}_{w=\mathsf{n}+\mathsf{q}-\mathsf{m}+1}^{\mathsf{n}+\mathsf{q}}$ to P_3.

7. The parties compute $r \leftarrow \mathcal{F}_{\mathbf{cf}}$. Let $\mathcal{CC} = \{i : r[i] = 1\}$, and let $\mathcal{EC} = \{1, \ldots, \rho\} \setminus \mathcal{CC}$.

8. For $j \in \mathcal{EC}$ (the *evaluation* circuits):

   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sends $\langle b_w \rangle^{(1)} \oplus \langle \lambda_{w,j} \rangle^{(1)}$ to P_2, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
   - For $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}\}$: P_2 sends $\langle b_w \rangle^{(2)} \oplus \langle \lambda_{w,j} \rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sends $k^1_{w,j} := g^{a^1_{w,b_w}\cdot r^1_j}$ to P_3, who computes $s^1_{w,b_w\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^1_{w,b_w}\cdot r^1_j})$.
   - For $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}_2\}$: P_2 sends $k^2_{w,j} := g^{a^2_{w,b_w}\cdot r^2_j}$ to P_3, who computes $s^2_{w,b_w\oplus\lambda_{w,j},j} := \mathsf{Ext}(g^{a^2_{w,b_w}\cdot r^2_j})$.

- P_3 evaluates circuit GC_j using $\left\{(s^1_{w,b_w\oplus\lambda_{w,j},j}, s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})\right\}_{w\in\{1,\ldots,\mathsf{n}\}}$ as inputs.

P_3 uses the encoded output translation tables sent in Step 6 to check if he received exactly one valid output value for each output wire. If not, he stores these outputs as o^j_0 and o^j_1 and continues.

9. P_1 and P_2 construct a circuit C' as follows:

   - P_1 inputs string $x \in \{0,1\}^{\mathsf{n}_1}$ and strings $o^i_{w,0}, o^i_{w,1} \in \{0,1\}^\kappa$, for $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1,\ldots,\mathsf{n}+\mathsf{q}\}$.
   - P_1 inputs string $y \in \{0,1\}^{\mathsf{n}_2-\mathsf{n}_1}$ and strings $o^2_{w,0}, o^2_{w,1} \in \{0,1\}^\kappa$, for $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1,\ldots,\mathsf{n}+\mathsf{q}\}$.
   - P_3 inputs $o_0, o_1 \in \{0,1\}^\kappa$.
   - If there exists some j such that $o^1_{j,0}\|o^2_{j,0} = o^j_0$ and $o^1_{j,1}\|o^2_{j,1} = o^j_1$, then P_3's output is $x\|y$; otherwise P_3 receives no output.
   - The circuit also outputs the values $\{o^1_{w,0}, o^1_{w,1}, o^2_{w,0}, o^2_{w,1}\}^{\mathsf{n}+\mathsf{q}}_{w=\mathsf{n}+\mathsf{q}-\mathsf{m}+1}$ input by parties P_1 and P_2 above.

   The parties run $\Pi^m_{\mathbf{3PC}}(P_1,P_2,P_3)$ on circuit C' as follows:

   - P_1 inputs her input $x = b_1\ldots b_{\mathsf{n}_1}$; P_2 inputs his input $y = b_{\mathsf{n}_1+1}\ldots b_{\mathsf{n}_2}$.
   - If P_3 received two conflicting outputs $o^1_0\|o^2_0$ and $o^1_1\|o^2_1$ for some circuit $j \in \{1,\ldots,\rho\}$ in Step 8, then he inputs these values; otherwise he inputs garbage.
   - The garbled circuit uses the same $a^1_{w,0}, a^1_{w,1}, a^2_{w,0}, a^2_{w,1}$ values as in Step 2.

   P_3 verifies that the values $\{o^1_{w,0}, o^1_{w,1}, o^2_{w,0}, o^2_{w,1}\}^{\mathsf{n}+\mathsf{q}}_{w=\mathsf{n}+\mathsf{q}-\mathsf{m}+1}$ output by C' match those in the encoded output translation tables sent in Step 6

10. For $j \in \mathcal{CC}$ (the *check* circuits):

    - P_i, for $i \in \{1,2\}$, does the following:
      - Sends r^i_j to P_3, and P_3 checks that these values are consistent with the pairs $\left\{(j, g^{r^i_j})\right\}$ sent before.
      - For $w \in \{1,\ldots,\mathsf{n}\}$: Sends sub-labels $s^i_{w,0,j}$ and $s^i_{w,1,j}$, mask bit share $\lambda^{(i)}_{w,j}$, and the labels to the authenticated bits to P_3.
    - Given the above information, P_3 reconstructs all input labels and verifies they match with those labels sent previously. Also, using said labels, P_3 verifies that the garbled circuit is correctly constructed.

11. For the cut-and-choose computation from Step 9, let $\widehat{\mathcal{EE}}$ be the check circuits, let $\widehat{r}^i_j$ be analogous to the r^i_j values from Step 2, and let $\widehat{k}^i_{w,j}$ be analogous to the $k^i_{w,j}$ from Step 6.

    For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 and P_3 run a zero-knowledge proof-of-knowledge, with P_1 proving that there exists some $b_w \in \{0,1\}$ such that for every $j \in \mathcal{EE}$ and for every $j' \in \widehat{\mathcal{EE}}$, $k^1_{w,j} = g^{a^1_{w,b_w}\cdot r^1_j}$ and $\widehat{k}^1_{w,j'} = g^{a^1_{w,b_w}\cdot \widehat{r}^1_{j'}}$.
    For $w \in \{\mathsf{n}_1+1,\ldots,\mathsf{n}_2\}$: P_2 and P_3 run a zero-knowledge proof-of-knowledge, with P_2 proving that there exists some $b_w \in \{0,1\}$ such that for every $j \in \mathcal{EE}$ and for every $j' \in \widehat{\mathcal{EE}}$, $k^2_{w,j} = g^{a^2_{w,b_w}\cdot r^2_j}$ and $\widehat{k}^2_{w,j'} = g^{a^2_{w,b_w}\cdot \widehat{r}^2_{j'}}$.

12. P_3 either outputs the output received in the evaluation circuits, or, if P_3 received any inconsistent inputs in Step 8, then it locally computes $f(x, y, z)$, where x and y are the inputs P_3 received in Step 9, and z is P_3's own input.

Theorem 6.3. *Let C be an arbitrary polynomial-size circuit and let $\mathbb{G}$ be a cyclic group with prime order. Given access to ideal functionalities $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{gate}}$, $\mathcal{F}_{\mathbf{oshare}}$, $\mathcal{F}_{\mathbf{ot}}$, $\mathcal{F}_{\mathbf{rand}}$, and $\mathcal{F}_{\mathbf{ss}}$, and assuming that the decisional Diffie-Hellman problem is hard in $\mathbb{G}$, then $\Pi_{\mathbf{3PC}}^{m-lindell}(P_1, P_2, P_3)$ securely computes the circuit C in the presence of an adversary corrupting an arbitrary number of parties.*

Proof. The analysis here is nearly identical to the previous proof as well as the proof for the two-party case [10], and thus we just present the simulators for each corruption case.

The adversary corrupts parties P_1 and P_2. The simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_3 would, using input 0^{n-n_2}, until Step 11. Here, $\mathcal{S}$ uses the witnesses a^i_{w,b_w} sent by P_1 and P_2 for the zero-knowledge proof-of-knowledge to extract their inputs. $\mathcal{S}$ then feeds these inputs to the trusted third party, receiving back $f(x, y, z)$. $\mathcal{S}$ continues to run as P_3 would, and halts, outputting whatever $\mathcal{A}$ outputs.

The adversary corrupts parties P_1 and P_3. The simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 would up until Step 6. $\mathcal{S}$ can extract both P_1's input x and its mask bits $\lambda_{w,j}$ through P_1's calls to $\mathcal{F}_{\mathbf{ss}}^1$ in Step 2 and Step 4. Likewise, in Step 5, $\mathcal{S}$ extracts P_3's input z through the calls to $\mathcal{F}_{\mathbf{ot}}$. $\mathcal{S}$ then passes x and z to the trusted third party, learning $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $r \leftarrow\!\!\$\, \{0,1\}^\rho$. Then for $j \in \{1, \ldots, \rho\}$, $\mathcal{S}$ proceeds as follows: If $r[j] = 0$, $\mathcal{S}$ uses the simulator that is known to exist for the two-party circuit garbling protocol

to construct garbled circuits which output $f(x, y, z)$. Otherwise, $\mathcal{S}$ acts as P_2 would. $\mathcal{S}$ continues to act as P_2 would, except that in Step 7 it sets the output of $\mathcal{F}_{\mathbf{cf}}$ to be equal to the r chosen above. Finally, $\mathcal{S}$ halts, outputting whatever $\mathcal{A}$ outputs.

The adversary corrupts parties P_2 and P_3. The analysis here is the same as the case where parties P_1 and P_3 are corrupt.

The adversary corrupts party P_1. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_1 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 and P_3 would, extracting P_1's input x through the calls to $\mathcal{F}_{\mathbf{ss}}^1$ in Step 4. $\mathcal{S}$ passes x to the trusted third party, learning $f(x, y, z)$. For the rest of the protocol, $\mathcal{S}$ acts as P_2 and P_3 would, and eventually halts, outputting whatever $\mathcal{A}$ outputs.

The adversary corrupts party P_2. The analysis here is the same as the case where party P_1 is corrupt.

The adversary corrupts party P_3. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_3 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_1 and P_2 would, extracting P_3's input z through the $\mathcal{F}_{\mathbf{ot}}$ calls in Step 5. $\mathcal{S}$ then hands z to the trusted third party, who returns $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $r \leftarrow\!\!\$\ \{0,1\}^\rho$, and then for $j \in \{1, \ldots, \rho\}$ $\mathcal{S}$ proceeds as follows: If $r[j] = 0$, $\mathcal{S}$ constructs a distributed garbled circuit which outputs $f(x, y, z)$, otherwise $\mathcal{S}$ proceeds as normal. Then, in Step 7, $\mathcal{S}$ fixes the output of $\mathcal{F}_{\mathbf{cf}}$ to be r. For the rest of the protocol, $\mathcal{S}$ acts as P_1 and P_2 would, and eventually halts, outputting whatever $\mathcal{A}$ outputs. □

Efficiency

We now briefly argue why our 3PC protocols are roughly eight times as expensive in terms of computation as the underlying 2PC protocols we utilize, and roughly sixteen times as expensive in terms of communication; see Section 6.6 for a more detailed analysis and comparison with prior work.

Both protocols are very similar to the underlying 2PC protocol they are based on; the major changes in terms of computation cost are that (1) the cost of encrypting a single row increases due to the use of the distributed encryption scheme, and (2) P_3 needs to do twice the work (due to needing to communicate with *both* P_1 and P_2) as compared to the evaluator in the underlying 2PC protocol. Indeed, it takes about eight PRF calls (where one PRF call equals outputting κ bits) to encrypt a single row of the garbled circuit, and thus the cost and size of a garbled circuit increases by a factor of eight. The cost for P_1 and P_2 to distributively garble a circuit is a small number of OTs per gate, and this can be amortized using OT extension techniques [7].

In terms of communication cost, both P_1 and P_2 need to send their half of the distributed garbled circuit to P_3, and the communication cost of actually constructing a distributed garbled circuit is roughly the cost of a standard garbled circuit. Since each garbled circuit is eight times larger than in the underlying 2PC protocol, we find that the overall communication size increases by approximately sixteen.

6.5 Hybrid Functionalities

We now describe in more detail the ideal functionalities described in Section 6.3, as well as efficient implementations of them in both the semi-honest and malicious settings.

Secret sharing of a constant bit. The functionality $\mathcal{F}^b_{\mathbf{const}}$ is parameterized by a bit b, and outputs a sharing (authenticated, in the malicious setting) of that bit.

Functionality $\mathcal{F}^b_{\mathbf{const}} \to \langle b \rangle$

Output: The functionality does the following:

1. Choose bit $r \in_R \{0,1\}$.
2. (Semi-honest setting) Output r to P_i and $r \oplus b$ to P_j.
3. (Malicious setting) Construct authenticated bits $r_i = \langle r \rangle^{(i)}$ and $r_j = \langle r \oplus b \rangle^{(j)}$, and output r_i to party P_i and r_j to party P_j.

In the semi-honest setting, this functionality can be instantiated by having P_i generate a random bit r and sending r to P_j, who computes $r \oplus b$. In the malicious setting we can instantiate this using the protocol described by Nielsen et al. [33, Figure 3].

Bit secret sharing. The functionality $\mathcal{F}_{\mathbf{ss}}$ in the semi-honest setting is the standard secret sharing functionality. In the malicious setting, the functionality creates an authenticated sharing of the input bit.

Functionality $\mathcal{F}^i_{\mathbf{ss}}(b) \to \langle b \rangle$

Input: Party P_i inputs a bit b.

Output: The functionality does the following:

1. Select $r \in_R \{0,1\}$ uniformly at random.
2. (Semi-honest setting) Output r to party P_j, and $b \oplus r$ to party P_i.

3. (Malicious setting) Construct authenticated bits $b_j = \langle r \rangle^{(j)}$ and $b_i = \langle b \oplus r \rangle^{(i)}$, and output b_j to party P_j and b_i to party P_i.

Implementing the functionality in the semi-honest setting is trivial. In the malicious setting we can use the **Input** protocol described by Nielsen et al. [33, Figure 6].

One-out-of-two oblivious secret sharing. The functionality $\mathcal{F}_{\textbf{oshare}}$ is used to share the sub-labels of the garbled table in an oblivious fashion while preserving consistency with respect to the circuit such that the circuit evaluation succeeds given the correct input sub-labels. More precisely, $\mathcal{F}_{\textbf{oshare}}$ interacts with two parties, called the *sender* P_j and the *receiver* P_i; it expects two inputs, m_0 and m_1, from the sender along with a two-out-of-two sharing $\langle b \rangle$ of a selection bit b between the sender and receiver, and outputs a random two-out-of-two sharing $[m_b]$ of m_b. In the malicious setting, $\langle b \rangle$ is an authenticated bit share. The functionality does not leak any information about b to the parties. Furthermore, when the sender is honest, it leaks no information on its inputs to the receiver. However, to ensure simulatability we allow a corrupted sender to choose its output share, $y^{(j)}$, at will.

Functionality $\mathcal{F}^{i,j}_{\textbf{oshare}}(\langle b \rangle^{(i)}, (\langle b \rangle^{(j)}, m_0, m_1, y^{(j)})) \to [m_b]$

Input: Party P_i inputs share $b_i = \langle b \rangle^{(i)}$. Party P_j inputs share $b_j = \langle b \rangle^{(j)}$ and vector $(m_0, m_1) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$. In the malicious setting, b_i and b_j are authenticated. Additionally, party P_j inputs a value $y^{(j)} \in \{0,1\}^\kappa \cup \{\perp\}$; if P_j is honest it sets $y^{(j)} = \perp$, otherwise $y^{(j)}$ can be arbitrary.

Output: The functionality does the following:

1. (Malicious setting) If either $\langle b \rangle^{(i)}$ or $\langle b \rangle^{(j)}$ is not a correctly authenticated bit then abort.
2. Compute $b := b_i \oplus b_j$
3. If $y^{(j)} = \perp$, then choose $y^{(j)} \in_R \{0,1\}^\kappa$.
4. Output $m_b \oplus y^{(j)}$ to P_i and $y^{(j)}$ to P_j.

See Section 6.5.1 for an instantiation of $\mathcal{F}_{\textbf{oshare}}$ in the semi-honest setting, and

Section 6.5.2 for an instantiation in the malicious setting.

Oblivious secret sharing. The oblivious secret sharing functionality $\mathcal{F}_{\mathbf{rand}}$ takes no inputs, and outputs a sharing of a random bit. In the malicious setting, this output sharing is authenticated.

Functionality $\mathcal{F}_{\mathbf{rand}} \to \langle r \rangle$

Output: The functionality does the following:

1. Choose bits $r, r' \in_R \{0,1\}$.
2. (Semi-honest setting) Output r' to P_i and $r \oplus r'$ to P_j.
3. (Malicious setting) Construct authenticated bits $r_i = \langle r' \rangle^{(i)}$ and $r_j = \langle r \oplus r' \rangle^{(j)}$, and output r_i to party P_i and r_j to party P_j.

In the semi-honest setting, this can be easily instantiated by each party choosing a random bit r_i and letting $r = r_i \oplus r_j$. In the malicious setting we need to construct authenticated shares and thus need additional machinery: we can utilize the **Rand** protocol by Nielsen et al. [33, Figure 6].

Oblivious gate evaluation. The functionality $\mathcal{F}^G_{\mathbf{gate}}$ takes as inputs shares of bits a and b, and outputs a share of $G(a,b)$ for some binary gate G. In the malicious setting, both input and output shares are authenticated.

Functionality $\mathcal{F}^G_{\mathbf{gate}}(\langle a \rangle, \langle b \rangle) \to \langle G(a,b) \rangle$

Input: The parties input bit shares $\langle a \rangle$ and $\langle b \rangle$. In the malicious setting, these shares are authenticated.

Auxiliary Input: The description of a binary gate G.

Output: The functionality does the following:

1. (Malicious setting) If any of the shares are not correctly authenticated then abort.
2. Compute a and b from the shares.
3. Compute $c = G(a,b)$ and output a sharing (authenticated in the malicious setting) of c.

As our circuits require only AND and XOR gates, we only consider those choices

for G here. In the semi-honest setting, we can compute XOR gates locally and AND gates using one-out-of-four OT, as is done in the GMW protocol [2]. In the malicious setting we can efficiently instantiate the functionality for XOR and AND gates using the **XOR** and **AND** protocols by Nielsen et al. [33, Figure 6].

Oblivious transfer. The functionality $\mathcal{F}_{\mathbf{ot}}$ implements standard oblivious transfer.

Functionality $\mathcal{F}_{\mathbf{ot}}^{i,j}(b, (m_0, m_1)) \to m_b$

Input: Party P_i inputs a choice bit b. Party P_j inputs two messages m_0 and m_1.

Output: The functionality outputs m_b to P_i, and $\perp$ to P_j.

This can be implemented efficiently in both the semi-honest and malicious setting using known existing protocols [48].

6.5.1 Semi-honest Implementation of $\mathcal{F}_{\mathbf{oshare}}$

The following protocol implements the $\mathcal{F}_{\mathbf{oshare}}$ functionality, where $m \odot b = m$ if $b = 1$ and the zero-string otherwise.

Protocol $\Pi_{\mathbf{oshare}}^{i,j}(\langle b \rangle^{(i)}, (\langle b \rangle^{(j)}, m_0, m_1, \perp))$

Input: Party P_i inputs its share $b_i = \langle b \rangle^{(i)}$ of $\langle b \rangle$. Party P_j inputs its share $b_j = \langle b \rangle^{(j)}$ of $\langle b \rangle$ along with two strings $m_0, m_1 \in \{0,1\}^{\kappa}$.

1. P_j chooses $r \in_R \{0,1\}^{\kappa}$ uniformly at random.
2. Execute $\mathcal{F}_{\mathbf{ot}}$ with P_j as the sender having inputs $(s_0, s_1) = (m_0 \oplus r, m_1 \oplus r)$, and P_i as the receiver having input $b' = 1 \oplus b_i$; denote P_i's output as y_i.

Outputs: P_j outputs $y_j = ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r$ and P_i outputs y_i.

Lemma 6.1. *The protocol* $\Pi_{\mathbf{oshare}}^{i,j}$ *securely implements the functionality* $\mathcal{F}_{\mathbf{oshare}}^{i,j}$ *in the presence of a semi-honest adversary in the* $\mathcal{F}_{\mathbf{ot}}$*-hybrid world.*

Proof. First, we show correctness; namely, we argue that the output of the protocol

is a two-out-of-two sharing of m_b, that is, $y_i \oplus y_j = m_b$. Indeed,

$$\begin{aligned} y_i \oplus y_j &= m_{1\oplus b_i} \oplus r \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r \\ &= m_{1\oplus b_i} \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)). \end{aligned}$$

Note that if $b_j = 0$, we have

$$y_i \oplus y_j = m_{1\oplus b_i} \oplus m_0 \oplus m_1 = m_b.$$

Likewise, if $b_j = 1$, we have

$$y_i \oplus y_j = m_{1\oplus b_i} = m_b.$$

To prove that the protocol is simulatable, observe that (1) P_j receives no information in the protocol (which follows from the privacy of OT) and (2) P_i only sees y_i, where the value $m_{1\oplus b_i}$ is perfectly blinded by r. Hence, similarly to the ideal evaluation of $\mathcal{F}_{\mathbf{oshare}}^{i,j}$, the values seen by the parties give them no information. More formally, we consider the following cases:

P_i is corrupted: The simulator, emulating the execution of $\mathcal{F}_{\mathbf{ot}}$ towards P_i, waits for $\mathcal{A}$ to input his bit $1 \oplus \langle b \rangle^{(i)}$. The simulator extracts $\langle b \rangle^{(i)}$, submits it to $\mathcal{F}_{\mathbf{oshare}}$, forwards the reply to $\mathcal{A}$, and halts with $\mathcal{A}$'s output.

P_j is corrupted: The simulator receives the messages $(m_0 \oplus r, m_1 \oplus r)$ from $\mathcal{A}$ and extracts r. The simulator then submits $(\langle b \rangle^{(j)}, m_0, m_1, y^{(j)})$ to $\mathcal{F}_{\mathbf{oshare}}$, where

$y^{(j)} = (m_0 \oplus m_1) \odot (1 \oplus \langle b \rangle^{(j)}) \oplus r$, and halts with $\mathcal{A}$'s output.

Noting that each of these simulators perfectly simulates the adversary in the $\mathcal{F}_{\mathbf{ot}}$-hybrid world, the protocol is secure. □

6.5.2 The Malicious Setting

In the malicious setting we utilize ideas from the protocol by Nielsen et al. [33]. In particular, each party P_i holds a global key Δ_i, which they use to construct authenticated bit shares. For a bit b authenticated towards P_i, P_i holds both the bit b and a MAC M_b, with P_j holding the authentication key K_b, with the condition that $M_b = K_b \oplus b\Delta_j$. To ease notation, in this Section we let $\langle b \rangle^{(i)} = (b, M_b, K_b)$.

Authenticated bit. We repeat here the $\mathcal{F}_{\mathbf{aBit}}$ functionality [33, Figure 5].

Functionality $\mathcal{F}_{\mathbf{aBit}} \rightarrow \langle r \rangle^{(i)}$

Auxiliary Input: Party P_j inputs its global key $\Delta_j \in \{0,1\}^\kappa$.

Output:

1. (If P_i is malicious) Set $\langle b \rangle^{(i)} = (b, M, M \oplus b\Delta)$.
2. (If P_j is malicious) Let $b \in_R \{0,1\}$ and set $\langle b \rangle^{(i)} = (b, K \oplus b\Delta_j, K)$.
3. (If both are honest) Let $b \in_R \{0,1\}$ and $K \in_R \{0,1\}^\kappa$, and set $\langle b \rangle^{(i)} = (b, K \oplus b\Delta_j, K)$.
4. Output $(b, K \oplus b\Delta_j)$ to P_i and (K, Δ_j) to P_j.

The implementation of $\mathcal{F}_{\mathbf{aBit}}$ is detailed in the work of Nielsen et al. [33, Section 4] and not repeated here. Note that the parties can utilize $\mathcal{F}_{\mathbf{aBit}}$ to construct a constant bit by P_i setting $M = 0^\kappa$ and P_j setting $K = b\Delta_j$.

Receiver-authenticated one-out-of-two oblivious transfer. We first define the functionality for receiver-authenticated oblivious transfer, which we utilize in our construction of a maliciously secure implementation of $\mathcal{F}_{\mathbf{oshare}}$.

Functionality $\mathcal{F}_{\mathbf{raot}}^{i,j}(\langle b\rangle^{(i)}, (m_0, m_1)) \to m_b$

Input: Party P_i inputs an authenticated choice bit b. Party P_j inputs two messages m_0 and m_1.

Output: The functionality does the following:
1. If P_i's choice bit b is not correctly authenticated then abort.
2. Output m_b to P_i.

In order to efficiently implement $\mathcal{F}_{\mathbf{raot}}$, we need the following functionality:

Functionality $\mathcal{F}_{\mathbf{eq}}(a, b) \to \{0, 1\}$

Input: Party P_i inputs $a \in \{0,1\}^\kappa$, and party P_j inputs $b \in \{0,1\}^\kappa$.

Output: The functionality outputs 1 if $a = b$, and 0 otherwise.

This can be instantiated efficiently using two calls to a random oracle H [33, pg. 7].

We can thus instantiate $\mathcal{F}_{\mathbf{raot}}$ as follows:

Protocol $\Pi_{\mathbf{raot}}(\langle b\rangle^{(i)}, (m_0, m_1))$

Let $\langle b\rangle^{(i)} = (b, M_b, K_b)$.

1. The parties compute $\langle r\rangle^{(i)} = (r, M_r, K_r) \leftarrow \mathcal{F}_{\mathbf{aBit}}^{i}$.
2. P_i computes $d = b \oplus r$ and sends d to P_j.
3. P_i sends $M_b \oplus M_r$ to $\mathcal{F}_{\mathbf{eq}}$, and P_j sends $(K_b \oplus K_r) \oplus d\Delta_j$ to $\mathcal{F}_{\mathbf{eq}}$, to check the equality of the two values. If they are not equal, P_j aborts the protocol.
4. The parties then compute $\langle d\rangle^{(i)} \leftarrow \mathcal{F}_{\mathbf{const}}^{d}$.
5. Let $\langle w\rangle^{(i)} = \langle r\rangle^{(i)} \oplus \langle d\rangle^{(i)}$. P_j sends $X_0 = H(K_w) \oplus m_0$ and $X_1 = H(K_w \oplus \Delta_j) \oplus m_1$ to P_i.

Output: P_i outputs $X_w \oplus H(M_w)$ and P_j outputs $\perp$.

Lemma. *The protocol $\Pi_{\mathbf{raot}}$ securely implements the functionality $\mathcal{F}_{\mathbf{raot}}$ in the presence of a malicious adversary in the Random Oracle model.*

Proof. Correctness is immediate. To prove simulatability, we consider the following corruption cases:

P_i is corrupted: The simulator $\mathcal{S}$ simulating an adversary $\mathcal{A}$ corrupting P_i proceeds as follows. $\mathcal{S}$ forwards its input (b, M_b) to $\mathcal{A}$ as input to the protocol. If $\mathcal{A}$ sends

a message v to $\mathcal{F}_{\mathbf{eq}}$ such that $v \neq M_b \oplus M_r$, $\mathcal{S}$ aborts the protocol. Otherwise, $\mathcal{S}$ sends $\langle b \rangle^{(i)}$ to the trusted party, receiving back m_b. $\mathcal{S}$ then generates two random bit-strings X_0 and X_1, and programs H so that $H(M_w) = X_b \oplus m_b$. Finally, $\mathcal{S}$ sends X_0 and X_1 to $\mathcal{A}$.

P_j is corrupted: The simulator $\mathcal{S}$ simulating an adversary $\mathcal{A}$ corrupting P_j proceeds as follows. $\mathcal{S}$ forwards its input (m_0, m_1, K_b) to $\mathcal{A}$ as input to the protocol. If $\mathcal{A}$ sends a message v to $\mathcal{F}_{\mathbf{eq}}$ such that $v \neq (K_b \oplus K_r) \oplus d\Delta_j$, $\mathcal{S}$ aborts the protocol. Next, $\mathcal{S}$ waits until P_j sends X_0 and X_1 to P_i. It then uses its knowledge of $\langle r \rangle^{(i)}$ and $\langle d \rangle^{(i)}$ to extract m_0 and m_1, which it feeds to the trusted party.

Noting that each of these simulators perfectly simulate the adversary in the ($\mathcal{F}_{\mathbf{aBit}}$, $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{eq}}$)-hybrid world, the protocol is secure. □

One-out-of-two oblivious secret sharing. We can instantiate $\mathcal{F}_{\mathbf{oshare}}$ in the malicious setting using a protocol similar to $\Pi_{\mathbf{oshare}}$ with the following modifications:

1. We use receiver-authenticated OT in place of standard OT;

2. To ensure that the simulator can extract consistent inputs from a corrupted sender, we do two invocations of $\mathcal{F}_{\mathbf{raot}}$;

3. In order to extract P_j's input in the case P_j is corrupt, we need an addition authentication check requiring one invocation of $\mathcal{F}_{\mathbf{aBit}}$.

Protocol $\Pi^{i,j}_{\mathbf{oshare-m}}(\langle b\rangle^{(i)}, (\langle b\rangle^{(j)}, m_0, m_1, \perp))$

Let $\langle b\rangle^{(i)} = (b_i, M_{b_i}, K_{b_i})$ and $\langle b\rangle^{(j)} = (b_j, M_{b_j}, K_{b_j})$.

1. P_j chooses $r_0, r_1 \in_R \{0,1\}^\kappa$ uniformly at random.
2. Execute $\mathcal{F}_{\mathbf{raot}}$ with P_j as sender having inputs $(s_0, s_1) = (m_0 \oplus r_0, m_1 \oplus r_1)$, and P_i as the receiver having input $\langle 1\rangle^{(i)} \oplus \langle b\rangle^{(i)}$; P_i denotes his output by y_i.
3. Execute $\mathcal{F}_{\mathbf{raot}}$ with P_j as sender having inputs $(s_0, s_1) = (r_0, r_1)$, and P_i as the receiver having input $\langle b\rangle^{(i)}$; P_i denotes his output by r_{b_i}.
4. Execute $\langle r\rangle^{(j)} = (r, M_r, K_r) \leftarrow \mathcal{F}^j_{\mathbf{aBit}}$. P_j sends (d, M_d) to P_i, where $d = r \oplus b_j$ and $M_d = M_r \oplus M_{b_j}$, and P_i checks if $(d, M_d, K_d \oplus K_r)$ is a valid authenticated bit, aborting if not.

Outputs: P_j outputs $y^{(j)} = ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r_0 \oplus r_1$ and P_i outputs $y_i \oplus r_{b_i}$.

Lemma. *The protocol $\Pi_{\mathbf{oshare-m}}$ securely implements the functionality $\mathcal{F}_{\mathbf{oshare}}$ in the presence of a malicious adversary in the ($\mathcal{F}_{\mathbf{aBit}}$, $\mathcal{F}_{\mathbf{raot}}$)-hybrid world.*

Proof. We first demonstrate correctness. It suffices to show that $y_i \oplus y_j = m_b$. Indeed,

$$\begin{aligned} y_i \oplus y_j &= (m_{1\oplus b_i} \oplus r_{1\oplus b_i} \oplus r_{b_i}) \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r_0 \oplus r_1 \\ &= m_{1\oplus b_i} \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)), \end{aligned}$$

and the derivation follows exactly as in the semi-honest case.

To prove that the protocol is simulatable, we consider the following corruption cases:

P_i is corrupted: The simulator $\mathcal{S}$ waits for $\mathcal{A}$ to input his choice bit $\langle b'\rangle^{(i)}$ to the first $\mathcal{F}_{\mathbf{raot}}$ invocation. If $\langle b'\rangle$ is not a valid authenticated bit, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ returns to $\mathcal{A}$ a random string y'_i. Likewise, in the second $\mathcal{F}_{\mathbf{raot}}$ invocation, $\mathcal{S}$ retrieves $\langle b\rangle^{(i)}$ from $\mathcal{A}$, aborting if the authentication check fails. $\mathcal{S}$ then invokes $\mathcal{F}_{\mathbf{oshare}}$ with $\langle b\rangle^{(i)}$, receiving P_i's output y_i. $\mathcal{S}$ computes $r'_i = y_i \oplus y'_i$

and sends r'_i to $\mathcal{A}$ as the output of the second $\mathcal{F}_{\mathbf{raot}}$. Finally, $\mathcal{S}$ acts as P_j would in Step 4, and halts with $\mathcal{A}$'s output.

P_j is corrupted: The simulator $\mathcal{S}$ emulates the two executions of $\mathcal{F}_{\mathbf{raot}}$ towards the adversary $\mathcal{A}$ controlling P_j, from which $\mathcal{S}$ receives (x_0, x_1) and (r'_0, r'_1), respectively. In Step 4, $\mathcal{S}$ extracts $\langle b_j \rangle^{(j)}$. $\mathcal{S}$ then computes $(m'_0, m'_1) = (x_0 \oplus r'_0, x_1 \oplus r'_1)$ and submits the message $(\langle b_j \rangle^{(j)}, m'_0, m'_1, y^{(j)})$ to $\mathcal{F}_{\mathbf{oshare}}$, where $y^{(j)} = ((m'_0 \oplus m'_1) \odot (1 \oplus b_j)) \oplus r'_0 \oplus r'_1$, and halts with $\mathcal{A}$'s output.

Noting that each of these simulators perfectly simulates the adversary in the ($\mathcal{F}_{\mathbf{aBit}}$, $\mathcal{F}_{\mathbf{raot}}$)-hybrid world, the protocol is secure. □

6.6 Evaluation

For simplicity we assume each party's input has length ℓ. Since we apply the XOR-tree technique to P_3's input, we let $\ell' = \max\{4\ell, 8\rho\}$ be the new input length.

Protocol based on cut-and-choose. Table 6.1 details the specific computational cost of each step for $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$. Note that these numbers are across *all* parties; the actual per-party cost is less. Each of the hybrid calls can be instantiated efficiently using known techniques: $\mathcal{F}_{\mathbf{cf}}$ can be instantiated very efficiently in the random oracle model requiring only three oracle calls, $\mathcal{F}_{\mathbf{zkpok}}$ can be instantiated using $3\rho/2 + 18$ exponentiations [9, pg. 36], and $\mathcal{F}_{\mathbf{ot}}$ can be computed using three exponentiations [48].

Protocol based on Lindell's protocol. The concrete computational cost for this protocol are similar to the ones above, except ρ in this case is smaller to achieve the

Step	Exponentiations	Hybrids Calls	Symmetric Ops
1			
2	$4\ell + 2(3\rho)$	$(2\ell \cdot \mathcal{F}_{\mathbf{ss}} + (\ell' + \mathsf{q}) \cdot \mathcal{F}_{\mathbf{rand}} + 4\ell \cdot H) \cdot (3\rho)$	
3		$(4\mathsf{q} \cdot \mathcal{F}_{\mathbf{gate}} + 8\mathsf{q} \cdot \mathcal{F}_{\mathbf{oshare}}) \cdot (3\rho)$	$8(3\rho)\mathsf{q}$
4		$2\ell \cdot \mathcal{F}_{\mathbf{ss}}$	
5		$(2\ell' \cdot \mathcal{F}_{\mathbf{ot}}) \cdot (3\rho)$	
6			
7		$\mathcal{F}_{\mathbf{cf}}$	
8			
9	3ρ		$4(3\rho)\mathsf{q}$
10		$(2\ell \cdot H + 2\ell \cdot \mathcal{F}_{\mathbf{zkpok}}) \cdot (3\rho)$	
11			$(3\rho)\mathsf{q}$

Table 6.1: Computational cost for each step of $\Pi^{m}_{\mathbf{3PC}}(P_1, P_2, P_3)$.

Step	Exponentiations	Hybrids Calls	Symmetric Ops
1			
2	$4\ell + 2\rho$	$(2\ell \cdot \mathcal{F}_{\mathbf{ss}} + (\ell' + \mathsf{q})\mathcal{F}_{\mathbf{rand}} + 4\ell \cdot H) \cdot \rho$	
3		$(4\mathsf{q} \cdot \mathcal{F}_{\mathbf{gate}} + 8\mathsf{q} \cdot \mathcal{F}_{\mathbf{oshare}}) \cdot \rho$	$8\rho\mathsf{q}$
4		$2\ell \cdot \mathcal{F}_{\mathbf{ss}}$	
5		$(2\ell' \cdot \mathcal{F}_{\mathbf{ot}}) \cdot \rho$	
6		$(2\mathsf{m} \cdot H) \cdot \rho$	
7		$\mathcal{F}_{\mathbf{cf}}$	
8		$((2\ell + \mathsf{m}) \cdot H) \cdot \rho$	$\rho\mathsf{q}$
9	— Cost of running $\Pi^{m}_{\mathbf{3PC}}(P_1, P_2, P_3)$ on circuit of size roughly $O(\ell + \mathsf{m})$ —		
10	ρ		$4\rho\mathsf{q}$
11		$(2\ell \cdot \mathcal{F}_{\mathbf{zkpok}}) \cdot \rho$	
12			

Table 6.2: Computational cost for each step of $\Pi^{m-lindell}_{\mathbf{3PC}}(P_1, P_2, P_3)$.

same level of security. However, we must run the above protocol as a sub-protocol. See Table 6.2 for the concrete efficiency counts.

Comparison with SPDZ. We compare our three-party protocol with the SPDZ protocol [69, 24, 70, 71, 72], an efficient MPC protocol which works for n parties and arbitrary corruptions over arithmetic circuits, and follows the preprocessing paradigm. SPDZ represents the state-of-the-art at the time of writing in terms of efficiency in the multi-party setting. Here we focus on the differences between

both SPDZ and our protocol, and discuss their strengths and weaknesses. Due to the different characteristics of each protocol (e.g., arithmetic versus boolean, linear versus constant round, etc.), these protocols are somewhat "incomparable". However, we hope to give a general idea of the efficiency trade-offs of both protocols.

There are several key differences between the SPDZ protocol and our own. For one, SPDZ works over arithmetic circuits, whereas our protocol works over boolean circuits.[2] In terms of communication, the SPDZ protocol requires rounds linear in the depth of the circuit, whereas our protocol is constant-round. While it is difficult to compare the impact of this without an implementation and experiments, it seems intuitive that as the latency between machines increases, the cost of each additional communication round increases as well; this intuition has been backed up by experiments in the semi-honest setting [74].

Finally, we consider the start-to-finish execution time (i.e., including the cost of preprocessing) for running an AES circuit. The preprocessing in our protocol is basically that found in the TinyOT protocol [33], and, using the numbers presented there, is fairly efficient (around 1 minute [33, Figure 21]). Efficiency comes from the fact that the preprocessing is only between two parties, namely, the circuit generators. The on-line running time is conjectured to be around that of maliciously secure two-party protocols using cut-and-choose.

The SPDZ protocol, on the other hand, has a very efficient (information-theoretic) online phase but a much costlier offline phase (around 17 minutes for

[2]Damgård and Zakarias [73] develop a SPDZ-like protocol for Boolean circuits; however, its practical efficiency is unclear.

three parties [70, Table 2]). In addition, it has a one-time setup phase which is very costly: the parties need to execute an MPC protocol for a circuit which generates a key pair with the secret key secret-shared among the parties. Executing this on its own would likely eclipse the running time of our protocol.[3] Thus, given preprocessing, it seems likely that SPDZ would out-perform our protocol; however, in the setting of executing the protocol from start to finish, we conjecture that our protocol would be more efficient.

Finally, our protocol is most efficient in the random oracle model, whereas SPDZ works in the standard model.

[3]We note that the work of Damgård et al. [71] presents an efficient protocol for this one-time setup phase in the weaker *covert* security model [75].

Chapter 7: Conclusion

In this dissertation we presented four improvements to the state-of-the-art in secure computation for various security models and settings. In Chapter 3 we show how to achieve an upwards of 5× improvement over the state-of-the-art maliciously secure two-party computation (2PC) protocols when considering the *multiple-execution* setting, where the two parties would like to compute the same function multiple times. In Chapter 4, we show a protocol in the *publicly-verifiable covert* (PVC) setting, where a cheating party produces a certificate of cheating if caught, which is nearly as efficient as the best known protocol in the covert setting. In Chapter 5, we present a maliciously secure 2PC protocol for functions with predicate checks on their inputs. And finally, in Chapter 6, we show an efficient *three*-party secure computation protocol which utilizes ideas from the two-party setting to construct a protocol more efficient than known multi-party protocols.

While these results bring the community closer to making secure computation a truly practical tool, there is still a lot of work that needs to be done before we can expect secure computation to be a viable approach in real-world settings. First and foremost, robust implementations need to be developed to benchmark the various proposed protocols, including several presented in this dissertation. We have

done some initial work on this with the release of `libgarble`[1], a garbled circuit library based on an implementation by Bellare et al. [6]. However, more needs to be done. Well developed libraries for oblivious transfer and other secure computation primitives need to be developed, and efficient and easy-to-use frameworks for implementing secure computation protocols need to be released to further the development of more efficient protocols. With regards to the (theoretical) research side of things, a better understanding of what security models are applicable to what settings needs to be studied, to determine whether semi-honest, PVC, or malicious protocols need to be used for certain real-world applications. Finally, developing ever-more practical protocols is always an important step towards making secure computation deployable in practice.

[1] https://github.com/amaloz/libgarble

Bibliography

[1] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[2] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[3] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[4] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In Matt Blaze, editor, *13th USENIX Security Symposium*, San Diego, California, USA, August 9–13, 2004. USENIX Association.

[5] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[6] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

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

[8] 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, Heidelberg, May 2007.

[9] 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, Heidelberg, March 2011.

[10] 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, Heidelberg, August 2013.

[11] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 513–524. ACM Press, October 2012.

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

[13] Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 875–888. ACM Press, November 2013.

[14] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE Computer Society Press, May 2014.

[15] 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, Heidelberg, August 2013.

[16] 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, Heidelberg, May 2013.

[17] 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, Heidelberg, March 2009.

[18] 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, Heidelberg, August 2014.

[19] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, April 2010.

[20] 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, Heidelberg, December 2012.

[21] 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, Heidelberg, May 2014.

[22] 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, Heidelberg, November / December 2015.

[23] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. https://eprint.iacr.org/2016/184.

[24] 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, Heidelberg, August 2012.

[25] 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, Heidelberg, August 2008.

[26] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 513–530. Springer, Heidelberg, August 2014.

[27] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[28] 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.

[29] 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, Heidelberg, July 2008.

[30] Benny Applebaum. Garbling XOR gates "for free" in the standard model. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 162–181. Springer, Heidelberg, March 2013.

[31] 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, Heidelberg, March 2012.

[32] 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, Heidelberg, May 2011.

[33] 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, Heidelberg, August 2012.

[34] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 166–184. Springer, Heidelberg, August 2013.

[35] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, Heidelberg, August 2010.

[36] 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, Heidelberg, August 2013.

[37] 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, Heidelberg, December 2012.

[38] Benny Pinkas. Fair secure two-party computation. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 87–105. Springer, Heidelberg, May 2003.

[39] Mehmet 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, June 2006.

[40] 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, Heidelberg, April 2006.

[41] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 558–576. Springer, Heidelberg, August 2010.

[42] 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, Heidelberg, August 2014.

[43] 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 15*, pages 579–590. ACM Press, October 2015.

[44] 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, Heidelberg, August 2015.

[45] 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, Heidelberg, August 2015.

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

[47] David P. Woodruff. Revisiting the efficiency of malicious two-party computation. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 79–96. Springer, Heidelberg, May 2007.

[48] 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, Heidelberg, August 2008.

[49] 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, Heidelberg, April 2015.

[50] 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, Heidelberg, April 2008.

[51] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management — Part 1: General (Revision 3). NIST Special Publication 800-57, July 2012.

[52] 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 13*, pages 535–548. ACM Press, November 2013.

[53] 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, Heidelberg, December 2009.

[54] 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, Heidelberg, April 2015.

[55] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Towards billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *21st USENIX Security Symposium*, Bellevue, Washington, USA, August 8–10, 2012. USENIX Association.

[56] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, October 2012.

[57] The case for elliptic curve cryptography. https://www.nsa.gov/business/programs/elliptic_curve.shtml. Accessed 2015-05-07.

[58] Ben Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In Sam King, editor, *22nd USENIX Security Symposium*, Washington, D.C., USA, August 14–16, 2013. USENIX Association.

[59] Crypto++ 5.6.0 benchmarks. http://www.cryptopp.com/benchmarks.html. Accessed 2015-05-08.

[60] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[61] Xiao Shaun Wang, Yan Huang, Yongan Zhao, Haixu Tang, XiaoFeng Wang, and Diyue Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 492–503. ACM Press, October 2015.

[62] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 955–966. ACM Press, November 2013.

[63] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[64] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[65] Peter de Rooij. Efficient exponentiation using procomputation and vector addition chains. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 389–399. Springer, Heidelberg, May 1995.

[66] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In Rainer A. Rueppel, editor, *EUROCRYPT'92*, volume 658 of *LNCS*, pages 200–207. Springer, Heidelberg, May 1993.

[67] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[68] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 378–394. Springer, Heidelberg, August 2005.

[69] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[70] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 241–263. Springer, Heidelberg, September 2012.

[71] 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, Heidelberg, September 2013.

[72] 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 13*, pages 549–560. ACM Press, November 2013.

[73] 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, Heidelberg, March 2013.

[74] 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, Heidelberg, April 2013.

[75] 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, Heidelberg, February 2007.