ABSTRACT

Title of dissertation: A NEW PARADIGM FOR PRACTICAL MALICIOUSLY SECURE MULTI-PARTY COMPUTATION

Xiao Wang, Doctor of Philosophy, 2018

Dissertation directed by: Professor Jonathan Katz
Department of Computer Science

Secure Multi-Party Computation (MPC) protocols allow a group of mutually distrusting users to compute a function jointly on their inputs without revealing any information beyond the output. For many years, implementations of MPC protocols have targeted security against *semi-honest* adversaries, i.e., attackers are assumed to execute the protocol honestly but try to learn private information after the fact. Protocols secure against stronger and more realistic *malicious* adversaries, who could behave arbitrarily during the protocol execution, were known to exist but were much less efficient.

This thesis introduces a new paradigm to construct extremely efficient MPC protocols with malicious security. In particular, this thesis consists of three major contributions.

1. We introduce the authenticated garbling framework, and present an efficient concrete instantiation of the protocol. The resulting protocol partially closes the gap between semi-honest and malicious MPC protocols asymptotically;

the implementation of the protocol represents the state-of-the-art system for malicious two-party computation.

2. We discuss how to apply authenticated garbling to the multi-party setting, where all-but-one parties can be corrupted by the adversary. The resulting protocol improves upon the best previous constant-round protocol by orders of magnitude. We also present a system that, for the first time, enables MPC executions among hundreds of parties, distributed globally.

3. We present a series of optimizations to two-party authenticated garbling by interpreting authenticated garbling in a new way. The improved malicious protocol has essentially the same concrete efficiency as the best semi-honest protocol in the preprocessing model.

4. We develop these protocols in EMP-toolkit, a practical and efficient MPC tool that can be used to build new protocols and to develop applications using our existing protocols.

A New Paradigm for Practical Maliciously Secure Multi-Party Computation

by

Xiao Wang

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
2018

Acknowledgments

I would like to first thank my advisor Jonathan Katz for his continuous support, guidance, and encouragement. It is a great honor to work with him. Jon's philosophy on how to do research and how to treat others has deeply shaped how I am as a human being and as a researcher, which is the best thing that could ever happen to me. He always talks to me as a peer and let me think and research independently, and provide help when I'm stuck. I will never forget his patience during hours of discussion in 2016 that led to the foundation of this thesis.

I had a great time in my three internships with Stratis Ioannidis, Nina Taft, Udi Weinsberg, Mariana Raykova, and Vladimir Kolesnikov. I really appreciate their patience during the mentoring.

I also want to thank Dov Gordon, Yan Huang, Alex Malozemoff, Kartik Nayak, Aishwarya Thiruvengadam, Ni Trieu, and Samee Zahur for their numerous collaboration and discussion during these five years. I am also grateful to Samuel Ranellucci and Mike Rosulek, who has contributed to this thesis.

I am grateful to all of my committee members, Jonathan Katz, Nikhil Chopra, Dana Dachman-Soled, Michael Hicks, Vladimir Kolesnikov, for taking the time to review my dissertation and participate in my defense.

Finally, I would like to thank my family members, Huiling Hu and my parents, for their years of mental support and understanding. Meeting Huiling during my Ph.D. has made me the luckiest man in the world. I am grateful that she always reminds me how research is only a small portion of a happy life.

Table of Contents

Chapter 1: Introduction

Protocols for secure multi-party computation (MPC) allow a set of mutually distrusting parties to jointly compute an agreed upon function on their own inputs. The security of MPC protocols ensures that no party can learn any information about other parties' inputs except the output. The past ten years have witnessed a huge improvement on the concrete efficiency of MPC protocols. Numerous companies have been founded using MPC to solve real-life problems that cannot be solved otherwise. For example, Dyadic [1] uses MPC to help secure cryptographic keys; Sharemind [2] uses MPC to process financial data [3] and prevent satellites from colliding without disclosing trajectories [4]; Partisia [5] uses MPC for privacy-preserving auctions.

While these results are impressive, the semi-honest security model is used in almost all applications, which assumes that both parties follow the protocol honestly yet may try to learn additional information from the execution. This is clearly not sufficient for all applications and has motivated researchers to construct protocols achieving the stronger notion of *malicious* security, where the adversary can cheat in arbitrary ways. Malicious protocols provide strong security but come with a very high performance penalty. Indeed, the state-of-the-art protocol with malicious security in 2014 [6] is still orders of magnitude slower than the protocols in the semi-honest setting.

This thesis introduces a new paradigm for constructing maliciously secure MPC protocols with extremely high efficiency. Existing protocols for MPC can be categorized into two main classes: 1) protocols based on garbled circuits that require high bandwidth but only a constant number of roundtrips; 2) protocols based on secret-sharing that usually need less communication but number of roundtrips proportional to the circuit depth. Such a trade-off between throughput and latency seems difficult to avoid especially for concretely efficient protocols. Authenticated garbling is a new paradigm that shows how to integrate these two classes of protocols efficiently such that the communication is only marginally increased compared to the secret-sharing protocols, while achieving constant-round.

Introducing Authenticated Garbling. In Chapter 3, we introduce the authenticated garbling framework, and present an efficient concrete instantiation of the protocol. From the high-level idea, this protocol follows the preprocessing model, where a big amount of computation is performed offline, and used later for efficient secure computation. The main feature of this framework is that, in the preprocessing model, the rest part of the protocol shares similar cost compared to a semi-honest protocol. We further improve the preprocessing by more than 2×. As a result, the resulting protocol closes the gap between semi-honest and malicious MPC protocols asymptotically; the implementation of the protocol represents the state-of-the-art for two-party computation.

This work was published in ACM CCS 2017 [7], and is a recipient of best paper award.

Applying Authenticated Garbling for the Multiparty Setting. In Chapter 4, we discuss how to apply the authenticated garbling technique to the multiparty setting, where all-but-one parties can be corrupted by the adversary. Almost all existing protocol with practical efficiency requires rounds of communication proportional to the depth of the circuit to compute. However, the protocol proposed here only requires a very small number of roundtrips independent of the circuit. The resulting protocol improves upon previous best constant-round protocol by at least three orders of magnitude. We also present a system that, for the first time, enables MPC execution among hundreds of parties, distributed globally.

This work was published in ACM CCS 2017 [8].

New Understanding and Optimization for Authenticated Garbling. In Chapter 5, we present a series of optimizations to the two-party authenticated garbling by interpreting authenticated garbling in a new way. The improved malicious protocol has essentially the same concrete efficiency as the best semi-honest protocol in the preprocessing model.

This work was published in Crypto 2018 [9].

EMP-Toolkit for Efficient Multi-Party Computation. In the course of this thesis, we also developed an efficient MPC toolkit, namely EMP-toolkit [10]. EMP aims to make it easy for cryptography researchers to develop fast prototypes of their protocols and for application developers who uses MPC as a black-box. See

Chapter 6 for more details.

1.1 Related Work

We will have a brief overview of prior works on maliciously secure two-party and multi-party computation protocols, focusing especially on practical protocols.

1.1.1 Two-party Computation

Protocols for generic 2PC in the semi-honest setting based on Yao's garbled-circuit protocol [11] have seen tremendous efficiency improvements over the past several years [12, 13, 14, 15, 16, 17, 18, 19].

Cut-and-choose is one of the most popular approach to strengthen the garbled circuit protocol with malicious security [20, 21, 22, 23, 24, 25, 26, 27, 6, 28]. For statistical security $2^{-\rho}$, the best approaches using this paradigm require ρ garbled circuits (which is optimal).

The cut-and-choose approach incurs significant overhead, especially for large circuits, precisely because ρ garbled circuits need to be transmitted (typically, $\rho \geq 40$). In order to mitigate this, recent works have explored secure computation in an *amortized* setting where the same function is evaluated multiple times (on different inputs) [29, 30, 31, 32]. When amortizing over τ executions, only $O(\frac{\rho}{\log \tau})$ garbled circuits are needed per execution. More recently, Nielsen and Orlandi [33] proposed a protocol with *constant* amortized overhead, but only when τ is at least the number of gates in the circuit.

Other techniques for constant-round, maliciously secure two-party computation in the single-execution setting, with asymptotically better performance than circuit-level cut-and-choose, have also been explored. The LEGO protocol [34] and subsequent optimizations [35, 36, 37, 38] are based on a gate-level cut-and-choose subroutine that can be carried out during a preprocessing phase before the circuit to be evaluated is known. This class of protocols has good asymptotic performance and small online time; however, the best reported implementation of LEGO [37] still has a higher overall running time than the best protocol based on a circuit-level cut-and-choose approach.

The Beaver-Micali-Rogaway compiler [39] provides yet another way to construct constant-round protocols with malicious security [40, 41]. This compiler uses an "outer" secure-computation protocol to generate a garbled circuit that is then evaluated. Lindell et al. [42, 43] suggested applying this idea using SPDZ [44] (based on somewhat homomorphic encryption) as the outer protocol, but did not provide an implementation of the resulting scheme.

There are also protocols whose round complexity is linear in the depth of the circuit being evaluated. The TinyOT protocol [45] extends the classical GMW protocol [46] by adding information-theoretic MACs to shares held by the parties; The IPS protocol [47] has excellent asymptotic complexity, but its concrete complexity is unclear since it has never been implemented (and appears quite difficult to implement).

1.1.2 Multi-Party Computation

Most existing MPC protocols rely on some variant of the secret-sharing paradigm introduced by Goldreich, Micali, and Wigderson [46]. At a high level, this technique requires the parties to maintain the invariant of holding a linear secret sharing of the values on the circuit wires, along with some sort of authentication information on those shares. Linear gates in the circuit (e.g., XOR, ADD) can be processed locally, while non-linear operations (e.g., AND, MULT) are handled by having the parties interact with each other to maintain the desired invariant. The most notable example of a protocol in this framework is perhaps SPDZ [44, 48, 49], which supports arithmetic circuits; protocols for boolean circuits have also been designed [50, 51].

Implementations of MPC protocols. The first implementations of generic MPC assumed a semi-honest adversary corrupting a minority of the parties. Early work in this area includes FairplayMP [52] for boolean circuits, and VIFF [53] and SEPIA [54] for arithmetic circuits. Implementations of protocols handling an arbitrary number of corrupted parties, but still in the semi-honest setting, were shown by Choi et al. [55] and Ben-Efraim et al. [56], the latter running in a constant number of rounds.

There are fewer implementations of MPC protocols handling *malicious* attackers. Jakobsen et al. [57] developed the first such system. SPDZ and its subsequent improvements [44, 48, 58, 49] greatly improved the efficiency. As noted earlier, all existing implementations of MPC tolerating malicious attackers have round complexity linear in the depth of the circuit.

Another line of work has specifically targeted *three*-party computation. Implementations here include Sharemind [59, 3], the sugar-beet auction run by Bogetoft et al. [60], and the recent work of Araki et al. [61]; these each tolerate only semi-honest behavior. Mohassel et al. [62] and Furukawa et al. [63] tolerate a malicious attacker corrupting only one party.

Constant-round MPC. *Constant-round* MPC protocols tolerating any number of malicious corruptions have also been designed. The basic approach [64] is to have the parties run a linear-round secure-computation protocol to compute a garbled circuit [11] for the function f of interest; the parties can then evaluate that garbled circuit using a constant number of additional rounds. Since the circuit for computing the garbling of f has depth independent of f, the overall number of rounds is constant. FairplayMP [52] and Ben-Efraim et al. [56] implemented this approach in the semi-honest setting. Other researchers have proposed approaches without providing an implementation, possibly because an implementation would be too complex or because the concrete efficiency of the resulting protocol would be uncompetitive with nonconstant-round protocols. As examples, Damgård and Ishai [40] proposed a protocol making black-box use of the underlying cryptographic primitives, and Choi et al. [41] looked at the three-party setting with malicious corruption of two parties. Lindell et al. [42, 43] considered optimizations of the BMR approach in the malicious setting.

Chapter 2: Background

2.1 Notation

We use κ to denote the computational security parameter (i.e., security should hold against attackers running in time $\approx 2^{\kappa}$), and ρ for the statistical security parameter (i.e., an adversary should succeed in cheating with probability at most $2^{-\rho}$). We use "=" to denote equality and ":=" to denote assignment. We denote the parties running the protocol by $\mathsf{P_A}$ and $\mathsf{P_B}$.

A circuit is represented as a list of gates having the format $(\alpha, \beta, \gamma, T)$, where α and β denote the indices of the input wires of the gate, γ is the index of the output wire of the gate, and $T \in \{\oplus, \wedge\}$ is the type of the gate. We use $\mathcal{I}_1$ to denote the set of indices of $\mathsf{P_A}$'s input wires, $\mathcal{I}_2$ to denote the set of indices of $\mathsf{P_B}$'s input wires, $\mathcal{W}$ to denote the set of indices of the output wires of all AND gates, and $\mathcal{O}$ to denote the set of indices of the output wires of the circuit.

When there are more than two parties, we denote them by $P_1, \ldots, P_n$. We use $\mathcal{I}_i$ to denote the set of input-wire indices for P_i, $\mathcal{W}$ to denote the set of output-wire indices for all AND gates, and $\mathcal{O}$ to denote the set of output-wire indices of the circuit. (We assume all parties learn the output.) $\mathcal{M}$ is used to denote the set of all corrupted parties, with $\mathcal{H} = [n] \setminus \mathcal{M}$ denoting the set of all honest parties. Our protocol operates by having the parties distributively construct a garbled circuit that is evaluated by one of the parties; we let P_1 be the circuit evaluator.

Functionality $\mathcal{F}$

Private inputs: P_i has input $x_i \in \{0,1\}^{n_i}$.

1. Receive $(\mathsf{input}, id, x_i)$ from P_i and store the messages if no such messages have been received.
2. If $(\mathsf{input}, id, x_i)$ is present for all i, computes $z = f(x_1, ..., x_n)$ and send the z to P_1.

Figure 2.1: Functionality $\mathcal{F}$ for multi-party secure computation.

2.2 Preliminaries

2.2.1 Secure Computation

We use the standard definition of security for two-party and multi-party computation in the presence of malicious adversaries [65, Chapter 7]. We present the MPC ideal functionality in Figure 2.1.

2.2.2 Information-theoretic MACs in the Two-Party Setting

We use the information-theoretic message authentication codes (IT-MACs) of [45], which we briefly recall. $\mathsf{P_A}$ holds a uniform *global key* $\Delta_\mathsf{A} \in \{0,1\}^\kappa$. A bit b known by $\mathsf{P_B}$ is authenticated by having $\mathsf{P_A}$ hold a uniform key $\mathsf{K}[b]$ and having $\mathsf{P_B}$ hold the corresponding tag $\mathsf{M}[b] := \mathsf{K}[b] \oplus b\Delta_\mathsf{A}$. Symmetrically, $\mathsf{P_B}$ holds an independent global key Δ_B; a bit b known by $\mathsf{P_A}$ is authenticated by having $\mathsf{P_B}$ hold a uniform key $\mathsf{K}[b]$ and having $\mathsf{P_A}$ hold the tag $\mathsf{M}[b] := \mathsf{K}[b] \oplus b\Delta_\mathsf{B}$. We use $[b]_\mathsf{A}$ to denote an authenticated bit known to $\mathsf{P_A}$ (i.e., $[b]_\mathsf{A}$ means $\mathsf{P_A}$ holds $(b, \mathsf{M}[b])$ and $\mathsf{P_B}$ holds $\mathsf{K}[b]$), and define $[b]_\mathsf{B}$ symmetrically.

Observe that this MAC is XOR-homomorphic: given $[b]_\mathsf{A}$ and $[c]_\mathsf{A}$, the parties can (locally) compute $[b \oplus c]_\mathsf{A}$ by having $\mathsf{P_A}$ compute $\mathsf{M}[b \oplus c] := \mathsf{M}[b] \oplus \mathsf{M}[c]$ and $\mathsf{P_B}$ compute $\mathsf{K}[b \oplus c] := \mathsf{K}[b] \oplus \mathsf{K}[c]$.

It is possible to extend the above idea to authenticate secret values by using XOR-based secret sharing and authenticating each party's share. That is, we can authenticate a bit λ, known to neither party, by letting r, s be uniform subject to $\lambda = r \oplus s$, and then having $\mathsf{P_A}$ hold $(r, \mathsf{M}[r], \mathsf{K}[s])$ and $\mathsf{P_B}$ hold $(s, \mathsf{M}[s], \mathsf{K}[r])$. It can be observed that this scheme is also XOR-homomorphic.

2.2.3 Information-theoretic MACs in the Multi-Party Setting

Authenticated bits. The idea above can be extended to the multiparty setting as follows. Each player P_i holds a global MAC key $\Delta_i \in \{0,1\}^\kappa$. When P_i holds a bit x authenticated by P_j, this means that P_j is given a random key $\mathsf{K}_j[x] \in \{0,1\}^\kappa$ and P_i is given the MAC tag $\mathsf{M}_j[x] := \mathsf{K}_j[x] \oplus x\Delta_j$. We let $[x]^i$ denote an authenticated bit where the value of x is known to P_i, and is authenticated to all other parties. In more detail, $[x]^i$ means that $(x, \{\mathsf{M}_k[x]\}_{k \neq i})$ is given to P_i, and $\mathsf{K}_j[x]$ is given to P_j for $j \neq i$.

Note that $[x]^i$ is XOR-homomorphic: given two authenticated bits $[x]^i, [y]^i$ known to the same party P_i, it is possible to locally compute the authenticated bit $[z]^i$ with $z = x \oplus y$ as follows:

- P_i computes $z := x \oplus y$, and $\{\mathsf{M}_j[z] := \mathsf{M}_j[x] \oplus \mathsf{M}_j[y]\}_{j \neq i}$;
- P_j (for $j \neq i$) computes $\mathsf{K}_j[z] := \mathsf{K}_j[x] \oplus \mathsf{K}_j[y]$.

Parties can also locally negate $[x]^i$, resulting in $[z]^i$ with $z = \bar{x}$:

- P_i computes $z := x \oplus 1$ and $\{\mathsf{M}_j[z] := \mathsf{M}_j[x]\}_{j \neq i}$;
- P_j (for $j \neq i$) computes $\mathsf{K}_j[z] := \mathsf{K}_j[x] \oplus \Delta_j$.

We let $\mathcal{F}^n_{\mathsf{abit}}$ denote an ideal functionality that distributes authenticated bits to the parties. Note that the above representation assumes that P_i uses a single global MAC key Δ_i. In cases where other keys are used, we explicitly add a subscript to the representation, i.e., we use $\mathsf{K}_i[x]_{G_i}$ and $\mathsf{M}_i[x]_{G_i} = \mathsf{K}_i[x]_{G_i} \oplus xG_i$ to denote the key and MAC tag in this case.

Authenticated shares. In the above construction, x is known to one party. To generate an authenticated *shared* bit x, where x is not known to any party, we generate XOR-shares for x (i.e., shares $\{x^i\}$ with $\bigoplus_i x^i = x$) and then distribute the authenticated bits $\{[x^i]^i\}$. We let $\langle x \rangle$ denote the collection of these authenticated shares for x; that is, $\langle x \rangle$ means that each party P_i holds $(x^i, \{\mathsf{M}_j[x^i], \mathsf{K}_i[x^j]\}_{j \neq i})$.

2.2.4 Secure Computation in the Preprocessing Model

Depending on the practical application of secure computation protocols, the execution can be divided into different phases:

- **Setup.** Here, the parties generate information that can be used for computation of multiple, possibly different functions. For example, base-OT can be performed in this phase.

- **Function-independent preprocessing.** Here, the parties begin execution of the protocol for a particular computation. At this point, the parties only need to know an upper bound on the size of the circuit that will be computed.

- **Function-dependent preprocessing.** Here the parties know the function being computed, but need not know their inputs.

- **Online phase.** The parties evaluate the function on their inputs.

Some applications require fast response given upon inputs, and thus want to minimize the online phase; other applications, where the parties involved in the secure computation is fixed, may want to minimize the cost of Function-dependent preprocessing and Online phase.

The function-independent preprocessing, denoted as $\mathcal{F}_{\mathsf{Pre}}$, is described in Figure 2.2. This functionality is used to set up correlated values between the parties along with their corresponding IT-MACs. The functionality chooses uniform global keys for each party, with the malicious party being allowed to choose its global key. Then, when the parties request a random authenticated bit, the functionality generates an authenticated secret sharing of the uniform bit $\lambda = r \oplus s$. (The adversary may choose the "random values" it receives, but this does not reveal anything to the adversary about $r \oplus s$ or the other party's global key.) Finally, the parties may also submit authenticated shares of two bits; the functionality then computes a (fresh) authenticated share of the AND of those bits.

Functionality $\mathcal{F}_{\mathsf{Pre}}$

Honest parties:

1. Upon receiving init from all parties, sample $\{\Delta_i \in \{0,1\}^\kappa\}_{i\in[n]}$ and send Δ_i to P_i.
2. Upon receiving random from all P_i, sample a random bit r and a random authenticated share $\langle r\rangle = \{(r^i, \{\mathsf{M}_j[r^i], \mathsf{K}_i[r^j]\}_{j\neq i})\}_{i\in[n]}$. For each $i \in [n]$, the box sends $(r^i, \{\mathsf{M}_j[r^i], \mathsf{K}_i[r^j]\}_{j\neq i})$ to P_i.
3. Upon receiving $\big(\mathtt{AND}, (r^i, \{\mathsf{M}_j[r^i], \mathsf{K}_i[r^j]\}_{j\neq i}), (s^i, \{\mathsf{M}_j[s^i], \mathsf{K}_i[s^j]\}_{j\neq i}\big)$ from P_i for all $i \in [n]$, the box checks that all MACs are valid, computes $t := \Big(\bigoplus_{i\in[n]} r^i\Big) \wedge \Big(\bigoplus_{i\in[n]} s^i\Big)$ and picks a random authenticated share $\langle t\rangle = \{(t^i, \{\mathsf{M}_j[t^i], \mathsf{K}_i[t^j]\}_{j\neq i})\}_{i\in[n]}$. For each $i \in [n]$, the box sends $(t^i, \{\mathsf{M}_j[t^i], \mathsf{K}_i[t^j]\}_{j\neq i})$ to P_i.

Corrupted parties: Corrupted parties can choose randomness used to compute the value they receive from the functionality.

Global key queries: The adversary at any point can send some (p, Δ') and will be told if $\Delta' = \Delta_p$.

Figure 2.2: The multi-party preprocessing functionality.

Chapter 3: Authenticated Garbling for Efficient Two-Party Computation

In this chapter, we will introduce our new paradigm for constructing constant-round, maliciously secure 2PC protocols with extremely high efficiency. At a high level (further details are in Section 3.1), and following ideas of Nielsen et al. [45], our protocol uses a function-independent preprocessing phase (namely $\mathcal{F}_{\mathsf{Pre}}$ introduced in Section 2.2.4) that sets up correlated randomness between the two parties, which they can use during the online phase for information-theoretic authentication of different values. In contrast to prior work, however, the parties in our protocol use this information in the online phase to generate a *single* "authenticated" garbled circuit. As in the semi-honest case, this garbled circuit can then be transmitted and evaluated in just one additional round.

Regardless of how we realize $\mathcal{F}_{\mathsf{Pre}}$, our protocol is extremely efficient in the function-dependent preprocessing phase and the online phase. Specifically, compared to Yao's *semi-honest* garbled-circuit protocol, the cost of the function-dependent preprocessing phase of our protocol is only about $2\times$ higher (assuming 128-bit computational security and 40-bit statistical security), and the cost of the online phase is essentially unchanged.

We show how to instantiate $\mathcal{F}_{\mathsf{Pre}}$ efficiently by developing a highly optimized version of the TinyOT protocol (adapting [45]), described in Section 3.3. Instantiating our framework in this way, we obtain a protocol with the same asymptotic communication complexity as recent protocols based on LEGO, but with two advantages. First, our protocol has much better *concrete* efficiency (see Section 3.6). For example, it requires only 16.6 ms total to evaluate AES, a $6\times$ improvement compared to a recent implementation of a LEGO-style approach [37]. Furthermore, the storage needed by our protocol is asymptotically smaller (see Table 3.1), something that is especially important when very large circuits are evaluated.

Instantiating our framework with the realization of $\mathcal{F}_{\mathsf{Pre}}$ described in Section 3.3 yields a protocol with the best concrete efficiency, and is the main focus. However, we note that our framework can also be instantiated in other ways:

- When $\mathcal{F}_{\mathsf{Pre}}$ is instantiated using the IPS compiler [47] and the bit-OT protocol by Ishai et al. [66], we obtain a maliciously secure constant-round 2PC protocol with total communication complexity $O(\kappa|\mathcal{C}|)$. Up to constant factors, this matches the complexity of *semi-honest* 2PC based on garbled circuits.

  The only previous work that achieves similar communication complexity [67] requires a constant number of public-key operations *per gate* of the circuit, and would have concrete performance much worse than our protocol.

- We can also realize $\mathcal{F}_{\mathsf{Pre}}$ using an offline, (semi-)trusted server. In that case we obtain a constant-round protocol for server-aided 2PC with complexity $O(\kappa|\mathcal{C}|)$. Previous work in the same model [68] achieves the same complexity

Table 3.1: Asymptotic complexity of constant-round 2PC protocols with malicious security. Communication (Comm.) is measured as the number of symmetric-key ciphertexts, and computation (Comp.) is measured as the number of symmetric-key operations. Storage refers to the number of symmetric-key ciphertexts stored after the offline stage. $|\mathcal{C}|, |\mathcal{I}|$, and $|\mathcal{O}|$ are the circuit size, input length, and output length, respectively; low-order terms independent of these parameters are ignored. The statistical security parameter is ρ, the computational security parameter is κ, and the number of protocol executions in the amortized setting is τ.

*Although the complexity of function-independent preprocessing can be reduced to $O(|\mathcal{C}|)$ using somewhat-homomorphic encryption [44], doing so requires a number of *public-key* operations proportional to $|\mathcal{C}|$.

Protocol	Function-ind. (Comm./Comp.)	Function-dep. (Comm./Comp.)	Online (Comm.)	Online (Comp./ Storage)
Cut-and-choose [26, 6, 28]	—	$O(\lvert\mathcal{C}\rvert\rho)$	$O(\lvert\mathcal{I}\rvert\rho)$	$O(\lvert\mathcal{C}\rvert\rho)$
Amortized [29, 30]	—	$O\left(\frac{\lvert\mathcal{C}\rvert\rho}{\log\tau}\right)$	$O\left(\frac{\lvert\mathcal{I}\rvert\rho}{\log\tau}\right)$	$O\left(\frac{\lvert\mathcal{C}\rvert\rho}{\log\tau}\right)$
LEGO [34, 35]	$O\left(\frac{\lvert\mathcal{C}\rvert\rho}{\log\tau+\log\lvert\mathcal{C}\rvert}\right)$	$O(\lvert\mathcal{C}\rvert)$	$O(\lvert\mathcal{I}\rvert+\lvert\mathcal{O}\rvert)$	$O\left(\frac{\lvert\mathcal{C}\rvert\rho}{\log\tau+\log\lvert\mathcal{C}\rvert}\right)$
SPDZ-BMR [42, 49]*	$O(\lvert\mathcal{C}\rvert\kappa)$	$O(\lvert\mathcal{C}\rvert)$	$O(\lvert\mathcal{I}\rvert+\lvert\mathcal{O}\rvert)$	$O(\lvert\mathcal{C}\rvert)$
This paper (with Section 3.3)	$O\left(\frac{\lvert\mathcal{C}\rvert\rho}{\log\tau+\log\lvert\mathcal{C}\rvert}\right)$	$O(\lvert\mathcal{C}\rvert)$	$\lvert\mathcal{I}\rvert+\lvert\mathcal{O}\rvert$	$O(\lvert\mathcal{C}\rvert)$
This paper (with [47])	$O(\lvert\mathcal{C}\rvert)$			

but with number of rounds proportional to the circuit depth.

3.1 Protocol Intuition

We give a high-level overview of our protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. Our protocol has the parties compute a garbled circuit in a distributed fashion, where the garbled circuit is "authenticated" in the sense that the circuit generator ($\mathsf{P_A}$ in our case) cannot change the logic of the circuit. We describe the intuition behind our construction in several steps.

We begin by reviewing standard garbled circuits. Each wire α of a circuit is associated with a random "mask" $\lambda_\alpha \in \{0,1\}$ known to $\mathsf{P_A}$. If the actual value of that wire (i.e., the value when the circuit is evaluated on the parties' inputs) is x, then the masked value observed by the circuit evaluator (namely, $\mathsf{P_B}$) on that wire will be $\hat{x} = x \oplus \lambda_\alpha$. Using the free-XOR technique [15], each wire α is also associated with two labels $\mathsf{L}_{\alpha,0}$ and $\mathsf{L}_{\alpha,1} := \mathsf{L}_{\alpha,0} \oplus \Delta$ known to $\mathsf{P_A}$. If the masked bit on that wire is $\hat{x}$, then $\mathsf{P_B}$ learns $\mathsf{L}_{\alpha,\hat{x}}$.

Let H be a hash function modeled as a random oracle. The garbled table for, e.g., an AND gate $(\alpha, \beta, \gamma, \wedge)$ with wires α, β, γ having values x, y, z, respectively, is given by:

$\hat{x}$ $\hat{y}$	truth table	garbled table
0 0	$\hat{z}_{00} = (\lambda_\alpha \wedge \lambda_\beta) \oplus \lambda_\gamma$	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 00) \oplus (\hat{z}_{00}, \mathsf{L}_{\gamma,\hat{z}_{00}})$
0 1	$\hat{z}_{01} = (\lambda_\alpha \wedge \overline{\lambda_\beta}) \oplus \lambda_\gamma$	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}, \gamma, 01) \oplus (\hat{z}_{01}, \mathsf{L}_{\gamma,\hat{z}_{01}})$
1 0	$\hat{z}_{10} = (\overline{\lambda_\alpha} \wedge \lambda_\beta) \oplus \lambda_\gamma$	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}, \gamma, 10) \oplus (\hat{z}_{10}, \mathsf{L}_{\gamma,\hat{z}_{10}})$
1 1	$\hat{z}_{11} = (\overline{\lambda_\alpha} \wedge \overline{\lambda_\beta}) \oplus \lambda_\gamma$	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}, \gamma, 11) \oplus (\hat{z}_{11}, \mathsf{L}_{\gamma,\hat{z}_{11}})$

$\mathsf{P_B}$, holding $(\hat{x}, \mathsf{L}_{\alpha,\hat{x}})$ and $(\hat{y}, \mathsf{L}_{\beta,\hat{y}})$, evaluates this garbled gate by picking the $(\hat{x}, \hat{y})$-th row and decrypting using the garbled labels it holds, thus obtaining $(\hat{z}, \mathsf{L}_{\gamma,\hat{z}})$.

The standard garbled circuit just described ensures security against a malicious $\mathsf{P_B}$, since (intuitively) $\mathsf{P_B}$ learns no information about the true values on any of the wires. Unfortunately, it provides no security against a malicious $\mathsf{P_A}$ who can potentially cheat by corrupting rows in the various garbled tables. One particular attack $\mathsf{P_A}$ can carry out is a *selective-failure* attack. Say, for example, that a malicious $\mathsf{P_A}$ corrupts only the $(0,0)$-th row of the garbled table for the gate above, and assume $\mathsf{P_B}$ aborts if it detects an error during evaluation. If $\mathsf{P_B}$ aborts, then $\mathsf{P_A}$ learns that the masked values on the input wires of the gate above were $\hat{x} = \hat{y} = 0$,

Table 3.2: Our final construction of an authenticated garbled table for an AND gate.

$x \oplus \lambda_\alpha$	$y \oplus \lambda_\beta$	$\mathsf{P_A}$'s share of garbled table	$\mathsf{P_B}$'s share of garbled table
0	0	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 00) \oplus (r_{00}, \mathsf{M}[r_{00}], \mathsf{L}_{\gamma,0} \oplus r_{00}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{00}])$	$(s_{00} = \hat{z}_{00} \oplus r_{00}, \mathsf{K}[r_{00}], \mathsf{M}[s_{00}])$
0	1	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}, \gamma, 01) \oplus (r_{01}, \mathsf{M}[r_{01}], \mathsf{L}_{\gamma,0} \oplus r_{01}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{01}])$	$(s_{01} = \hat{z}_{01} \oplus r_{01}, \mathsf{K}[r_{01}], \mathsf{M}[s_{01}])$
1	0	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}, \gamma, 10) \oplus (r_{10}, \mathsf{M}[r_{10}], \mathsf{L}_{\gamma,0} \oplus r_{10}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{10}])$	$(s_{10} = \hat{z}_{10} \oplus r_{10}, \mathsf{K}[r_{10}], \mathsf{M}[s_{10}])$
1	1	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}, \gamma, 11) \oplus (r_{11}, \mathsf{M}[r_{11}], \mathsf{L}_{\gamma,0} \oplus r_{11}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{11}])$	$(s_{11} = \hat{z}_{11} \oplus r_{11}, \mathsf{K}[r_{11}], \mathsf{M}[s_{11}])$

from which it learns that the true values on those wires were λ_α and λ_β.

The selective-failure attack just mentioned can be prevented if the masks are hidden from $\mathsf{P_A}$: in that case, even if $\mathsf{P_B}$ aborts and $\mathsf{P_A}$ learns the masked wire values, $\mathsf{P_A}$ learns nothing about the true wire values. Since knowledge of the garbled table would leak information about the masks to $\mathsf{P_A}$, the garbled table must be hidden from $\mathsf{P_A}$ as well. That is, we now want to set up a situation in which $\mathsf{P_A}$ and $\mathsf{P_B}$ hold *secret shares* of the garbled table, as follows:

$\hat{x}$ $\hat{y}$	$\mathsf{P_A}$'s share of garbled table	$\mathsf{P_B}$'s share of garbled table
0 0	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 00) \oplus (r_{00}, \mathsf{L}^\mathsf{A}_{\gamma,00})$	$(s_{00} = \hat{z}_{00} \oplus r_{00}, \mathsf{L}^\mathsf{B}_{\gamma,00})$
0 1	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}, \gamma, 01) \oplus (r_{01}, \mathsf{L}^\mathsf{A}_{\gamma,01})$	$(s_{01} = \hat{z}_{01} \oplus r_{01}, \mathsf{L}^\mathsf{B}_{\gamma,01})$
1 0	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}, \gamma, 10) \oplus (r_{10}, \mathsf{L}^\mathsf{A}_{\gamma,10})$	$(s_{10} = \hat{z}_{10} \oplus r_{10}, \mathsf{L}^\mathsf{B}_{\gamma,10})$
1 1	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}, \gamma, 11) \oplus (r_{11}, \mathsf{L}^\mathsf{A}_{\gamma,11})$	$(s_{11} = \hat{z}_{11} \oplus r_{11}, \mathsf{L}^\mathsf{B}_{\gamma,11})$

(Here, e.g., $\mathsf{L}^\mathsf{A}_{\gamma,00}, \mathsf{L}^\mathsf{B}_{\gamma,00}$ represent abstract XOR-shares of $\mathsf{L}_{\gamma,\hat{z}_{00}}$, i.e., $\mathsf{L}_{\gamma,\hat{z}_{00}} = \mathsf{L}^\mathsf{A}_{\gamma,00} \oplus \mathsf{L}^\mathsf{B}_{\gamma,00}$.) Once $\mathsf{P_A}$ sends its shares of all the garbled gates, $\mathsf{P_B}$ can XOR those shares with its own and then evaluate the garbled circuit as before.

Informally, the above ensures *privacy* against a malicious $\mathsf{P_A}$ since (intuitively) the results of any changes $\mathsf{P_A}$ makes to the garbled circuit are *independent* of $\mathsf{P_B}$'s inputs. However, $\mathsf{P_A}$ can still affect *correctness* by, e.g., flipping the masked value in a row. This can be addressed by adding an information-theoretic MAC on $\mathsf{P_A}$'s share of the masked bit. The shares of the garbled table now take the following form:

$\hat{x}$ $\hat{y}$	$\mathsf{P_A}$'s share of garbled table	$\mathsf{P_B}$'s share of garbled table
0 0	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 00) \oplus (r_{00}, \mathsf{M}[r_{00}], \mathsf{L}^{\mathsf{A}}_{\gamma,00})$	$(s_{00} = \hat{z}_{00} \oplus r_{00}, \mathsf{K}[r_{00}], \mathsf{L}^{\mathsf{B}}_{\gamma,00})$
0 1	$H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}, \gamma, 01) \oplus (r_{01}, \mathsf{M}[r_{01}], \mathsf{L}^{\mathsf{A}}_{\gamma,01})$	$(s_{01} = \hat{z}_{01} \oplus r_{01}, \mathsf{K}[r_{01}], \mathsf{L}^{\mathsf{B}}_{\gamma,01})$
1 0	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}, \gamma, 10) \oplus (r_{10}, \mathsf{M}[r_{10}], \mathsf{L}^{\mathsf{A}}_{\gamma,10})$	$(s_{10} = \hat{z}_{10} \oplus r_{10}, \mathsf{K}[r_{10}], \mathsf{L}^{\mathsf{B}}_{\gamma,10})$
1 1	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}, \gamma, 11) \oplus (r_{11}, \mathsf{M}[r_{11}], \mathsf{L}^{\mathsf{A}}_{\gamma,11})$	$(s_{11} = \hat{z}_{11} \oplus r_{11}, \mathsf{K}[r_{11}], \mathsf{L}^{\mathsf{B}}_{\gamma,11})$

Once $\mathsf{P_A}$ sends its shares of the garbled circuit to $\mathsf{P_B}$, the garbled circuit can be evaluated as before. Now, however, $\mathsf{P_B}$ will verify the MAC on $\mathsf{P_A}$'s share of each masked bit that it learns. This limits $\mathsf{P_A}$ to only being able to cause $\mathsf{P_B}$ to abort; as before, though, any such abort will occur independently of $\mathsf{P_B}$'s actual input.

Note that $\mathsf{P_A}$'s shares of the wire labels need not be authenticated, since a corrupted wire label can only cause input-independent abort; if $\mathsf{P_B}$ does not abort, the MACs on the masked bits ensure that $\mathsf{P_B}$ learns the correct masked wire value, i.e., $\hat{z}$.

Efficient realization. Although the above idea is powerful, it still remains to design an efficient protocol that allows the parties to distributively compute shares of a garbled table of the above form even when one of the parties is malicious.

One important observation is that if we set Δ (the free-XOR shift) equal to $\mathsf{P_A}$'s global key Δ_A, then we can secret share, e.g., $\mathsf{L}_{\gamma,\hat{z}_{00}}$ as

$$\begin{aligned}
\mathsf{L}_{\gamma,\hat{z}_{00}} &= \mathsf{L}_{\gamma,0} \oplus \hat{z}_{00}\Delta_\mathsf{A} \\
&= \mathsf{L}_{\gamma,0} \oplus (r_{00} \oplus s_{00})\Delta_\mathsf{A} \\
&= \mathsf{L}_{\gamma,0} \oplus r_{00}\Delta_\mathsf{A} \oplus s_{00}\Delta_\mathsf{A} \\
&= \underbrace{(\mathsf{L}_{\gamma,0} \oplus r_{00}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{00}])}_{\mathsf{L}^{\mathsf{A}}_{\gamma,00}} \oplus \underbrace{(\mathsf{K}[s_{00}] \oplus s_{00}\Delta_\mathsf{A})}_{\mathsf{L}^{\mathsf{B}}_{\gamma,00}}.
\end{aligned}$$

In our construction thus far, $\mathsf{P_A}$ knows $L_{\gamma,0}$ and r_{00} (in addition to knowing Δ_A).

Our key insight is that if s_{00} is an authenticated bit known to $\mathsf{P_B}$, then $\mathsf{P_A}$ can locally compute the share $\mathsf{L}^{\mathsf{A}}_{\gamma,00} := \mathsf{L}_{\gamma,0} \oplus r_{00}\Delta_{\mathsf{A}} \oplus \mathsf{K}[s_{00}]$ from the information it has; the other share $\mathsf{L}^{\mathsf{B}}_{\gamma,00} = \mathsf{K}[s_{00}] \oplus s_{00}\Delta_{\mathsf{A}}$ is equal to the value $\mathsf{M}[s_{00}]$ that $\mathsf{P_B}$ holds! So if we rewrite the garbled table as in Table 3.2, shares of the table become easy to compute in a distributed fashion.

Another optimization is based on the observation that the masked output values take the following form:

$$\begin{aligned}
\hat{z}_{00} &= (\lambda_\alpha \wedge \lambda_\beta) \oplus \lambda_\gamma \\
\hat{z}_{01} &= (\lambda_\alpha \wedge \overline{\lambda_\beta}) \oplus \lambda_\gamma = \hat{z}_{00} \oplus \lambda_\alpha \\
\hat{z}_{10} &= (\overline{\lambda_\alpha} \wedge \lambda_\beta) \oplus \lambda_\gamma = \hat{z}_{00} \oplus \lambda_\beta \\
\hat{z}_{11} &= (\overline{\lambda_\alpha} \wedge \overline{\lambda_\beta}) \oplus \lambda_\gamma = \hat{z}_{01} \oplus \lambda_\beta \oplus 1.
\end{aligned}$$

Thus, the parties can locally compute authenticated shares $\{r_{ij}, s_{ij}\}$ of the $\{\hat{z}_{i,j}\}$ from authenticated shares of $\lambda_\alpha, \lambda_\beta, \lambda_\gamma$, and $\lambda_\alpha \wedge \lambda_\beta$.

Finally, our actual protocol pushes as much of the garbled-circuit generation as possible into the preprocessing phase.

3.2 Our Main Framework

In Figure 3.2, we give the complete description of our main protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. For clarity we set $\rho = \kappa$, but in Section 3.5 we describe how arbitrary values of ρ can be supported. Note that the calls to $\mathcal{F}_{\mathsf{Pre}}$ can be performed in parallel and (as we show later) there is an efficient constant-round protocol for $\mathcal{F}_{\mathsf{Pre}}$; thus, the protocol overall runs in constant rounds.

Although our protocol, as described, calls $\mathcal{F}_{\mathsf{Pre}}$ in the function-dependent preprocessing phase, it is easy to push this to the function-independent phase using standard techniques similar to those used with multiplication triples [69].

Protocol Π_{2pc}

Inputs: In the function-dependent phase, the parties agree on a circuit for a function $f : \{0,1\}^{|\mathcal{I}_1|} \times \{0,1\}^{|\mathcal{I}_2|} \to \{0,1\}^{|\mathcal{O}|}$.
In the online phase, $\mathsf{P_A}$ holds $x \in \{0,1\}^{|\mathcal{I}_1|}$ and $\mathsf{P_B}$ holds $y \in \{0,1\}^{|\mathcal{I}_2|}$.

Function-independent preprocessing:

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ send init to $\mathcal{F}_{\mathsf{Pre}}$, which sends Δ_{A} to $\mathsf{P_A}$ and Δ_{B} to $\mathsf{P_B}$.
2. For each wire $w \in \mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{W}$, parties $\mathsf{P_A}$ and $\mathsf{P_B}$ send random to $\mathcal{F}_{\mathsf{Pre}}$. In return, $\mathcal{F}_{\mathsf{Pre}}$ sends $(r_w, \mathsf{M}[r_w], \mathsf{K}[s_w])$ to $\mathsf{P_A}$ and $(s_w, \mathsf{M}[s_w], \mathsf{K}[r_w])$ to $\mathsf{P_B}$. Define $\lambda_w = s_w \oplus r_w$. $\mathsf{P_A}$ also picks a uniform κ-bit string $\mathsf{L}_{w,0}$ and sets $\mathsf{L}_{w,1} := \mathsf{L}_{w,0} \oplus \Delta_{\mathsf{A}}$.

Function-dependent preprocessing:

3. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \oplus)$, $\mathsf{P_A}$ computes $(r_\gamma, \mathsf{M}[r_\gamma], \mathsf{K}[s_\gamma]) := (r_\alpha \oplus r_\beta, \mathsf{M}[r_\alpha] \oplus \mathsf{M}[r_\beta], \mathsf{K}[s_\alpha] \oplus \mathsf{K}[s_\beta])$, and sets $\mathsf{L}_{\gamma,0} := \mathsf{L}_{\alpha,0} \oplus \mathsf{L}_{\beta,0}$ and $\mathsf{L}_{\gamma,1} := \mathsf{L}_{\gamma,0} \oplus \Delta_{\mathsf{A}}$. Similarly, $\mathsf{P_B}$ computes $(s_\gamma, \mathsf{M}[s_\gamma], \mathsf{K}[r_\gamma]) := (s_\alpha \oplus s_\beta, \mathsf{M}[r_\beta] \oplus \mathsf{M}[r_\beta], \mathsf{K}[r_\alpha] \oplus \mathsf{K}[r_\beta])$. Define $\lambda_\gamma = \lambda_\alpha \oplus \lambda_\beta$.
4. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \wedge)$:

   (a) $\mathsf{P_A}$ (resp., $\mathsf{P_B}$) sends $(\mathtt{and}, (r_\alpha, \mathsf{M}[r_\alpha], \mathsf{K}[s_\alpha]), (r_\beta, \mathsf{M}[r_\beta], \mathsf{K}[s_\beta]))$ (resp., $(\mathtt{and}, (s_\alpha, \mathsf{M}[s_\alpha], \mathsf{K}[r_\alpha]), (s_\beta, \mathsf{M}[s_\beta], \mathsf{K}[r_\beta])))$ to $\mathcal{F}_{\mathsf{Pre}}$. In return, $\mathcal{F}_{\mathsf{Pre}}$ sends $(r_\sigma, \mathsf{M}[r_\sigma], \mathsf{K}[s_\sigma])$ to $\mathsf{P_A}$ and $(s_\sigma, \mathsf{M}[s_\sigma], \mathsf{K}[r_\sigma])$ to $\mathsf{P_B}$, where $r_\sigma \oplus s_\sigma = \lambda_\alpha \wedge \lambda_\beta$.

   (b) $\mathsf{P_A}$ computes the following locally:

$$\begin{array}{lcllll}
(r_{\gamma,0}, \mathsf{M}[r_{\gamma,0}], \mathsf{K}[s_{\gamma,0}]) & := & (r_\sigma \oplus r_\gamma, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] &) \\
(r_{\gamma,1}, \mathsf{M}[r_{\gamma,1}], \mathsf{K}[s_{\gamma,1}]) & := & (r_\sigma \oplus r_\gamma \oplus r_\alpha, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma] \oplus \mathsf{M}[r_\alpha], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] \oplus \mathsf{K}[s_\alpha] &) \\
(r_{\gamma,2}, \mathsf{M}[r_{\gamma,2}], \mathsf{K}[s_{\gamma,2}]) & := & (r_\sigma \oplus r_\gamma \oplus r_\beta, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma] \oplus \mathsf{M}[r_\beta], & \mathsf{K}[s_\sigma] \oplus \mathsf{K}[s_\gamma] \oplus \mathsf{K}[s_\beta] &) \\
(r_{\gamma,3}, \mathsf{M}[r_{\gamma,3}], \mathsf{K}[s_{\gamma,3}]) & := & (r_\sigma \oplus r_\gamma \oplus r_\alpha \oplus r_\beta, & \mathsf{M}[r_\sigma] \oplus \mathsf{M}[r_\gamma] \oplus \mathsf{M}[r_\alpha] \oplus \mathsf{M}[r_\beta], & \mathsf{K}[s_{\gamma,1}] \oplus \mathsf{K}[s_\beta] \oplus \Delta_{\mathsf{A}} &)
\end{array}$$

   (c) $\mathsf{P_B}$ computes the following locally:

$$\begin{array}{lcllll}
(s_{\gamma,0}, \mathsf{M}[s_{\gamma,0}], \mathsf{K}[r_{\gamma,0}]) & := & (s_\sigma \oplus s_\gamma, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] &) \\
(s_{\gamma,1}, \mathsf{M}[s_{\gamma,1}], \mathsf{K}[r_{\gamma,1}]) & := & (s_\sigma \oplus s_\gamma \oplus s_\alpha, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\alpha], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] \oplus \mathsf{K}[r_\alpha] &) \\
(s_{\gamma,2}, \mathsf{M}[s_{\gamma,2}], \mathsf{K}[r_{\gamma,2}]) & := & (s_\sigma \oplus s_\gamma \oplus s_\beta, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\beta], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] \oplus \mathsf{K}[r_\beta] &) \\
(s_{\gamma,3}, \mathsf{M}[s_{\gamma,3}], \mathsf{K}[r_{\gamma,3}]) & := & (s_{\gamma,1} \oplus s_\beta \oplus 1, & \mathsf{M}[s_\sigma] \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\alpha] \oplus \mathsf{M}[s_\beta], & \mathsf{K}[r_\sigma] \oplus \mathsf{K}[r_\gamma] \oplus \mathsf{K}[r_\alpha] \oplus \mathsf{K}[r_\beta] &)
\end{array}$$

   (d) $\mathsf{P_A}$ computes $\mathsf{L}_{\alpha,1} := \mathsf{L}_{\alpha,0} \oplus \Delta_{\mathsf{A}}$ and $\mathsf{L}_{\beta,1} := \mathsf{L}_{\beta,0} \oplus \Delta_{\mathsf{A}}$, and then sends the following to $\mathsf{P_B}$:

$$\begin{array}{llll}
G_{\gamma,0} := H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 0) \oplus (r_{\gamma,0}, & \mathsf{M}[r_{\gamma,0}], & \mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,0}] \oplus r_{\gamma,0}\Delta_{\mathsf{A}}) \\
G_{\gamma,1} := H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}, \gamma, 1) \oplus (r_{\gamma,1}, & \mathsf{M}[r_{\gamma,1}], & \mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,1}] \oplus r_{\gamma,1}\Delta_{\mathsf{A}}) \\
G_{\gamma,2} := H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}, \gamma, 2) \oplus (r_{\gamma,2}, & \mathsf{M}[r_{\gamma,2}], & \mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,2}] \oplus r_{\gamma,2}\Delta_{\mathsf{A}}) \\
G_{\gamma,3} := H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}, \gamma, 3) \oplus (r_{\gamma,3}, & \mathsf{M}[r_{\gamma,3}], & \mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,3}] \oplus r_{\gamma,3}\Delta_{\mathsf{A}})
\end{array}$$

Figure 3.1: Our protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. Here $\rho = \kappa$ for clarity, but this is not necessary (cf. Section 3.5).

3.2.1 Proof of Security

We prove security of our protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model.

Theorem 3.2.1. *If H is modeled as a random oracle, the protocol in Figure 3.2*

Protocol Π_{2pc}

Online:

5. For each $w \in \mathcal{I}_2$, $\mathsf{P_A}$ sends $(r_w, \mathsf{M}[r_w])$ to $\mathsf{P_B}$, who checks that $(r_w, \mathsf{K}[r_w], \mathsf{M}[r_w])$ is valid. If so, $\mathsf{P_B}$ computes $\lambda_w := r_w \oplus s_w$ and sends $y_w \oplus \lambda_w$ to $\mathsf{P_A}$. Finally, $\mathsf{P_A}$ sends $\mathsf{L}_{w,y_w \oplus \lambda_w}$ to $\mathsf{P_B}$.
6. For each $w \in \mathcal{I}_1$, $\mathsf{P_B}$ sends $(s_w, \mathsf{M}[s_w])$ to $\mathsf{P_A}$, who checks that $(s_w, \mathsf{K}[s_w], \mathsf{M}[s_w])$ is valid. $\mathsf{P_A}$ computes $\lambda_w := r_w \oplus s_w$ and sends $x_w \oplus \lambda_w$ and $\mathsf{L}_{w,x_w \oplus \lambda_w}$ to $\mathsf{P_B}$.

Circuit evaluation:

7. $\mathsf{P_B}$ evaluates the circuit in topological order. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, T)$, $\mathsf{P_B}$ initially holds $(z_\alpha \oplus \lambda_\alpha, \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha})$ and $(z_\beta \oplus \lambda_\beta, \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta})$, where z_α, z_β are the underlying values of the wires.
   (a) If $T = \oplus$, $\mathsf{P_B}$ computes $z_\gamma \oplus \lambda_\gamma := (z_\alpha \oplus \lambda_\alpha) \oplus (z_\beta \oplus \lambda_\beta)$ and $\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma} := \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha} \oplus \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}$.
   (b) If $T = \wedge$, $\mathsf{P_B}$ computes $i := 2(z_\alpha \oplus \lambda_\alpha) + (z_\beta \oplus \lambda_\beta)$ followed by $(r_{\gamma,i}, \mathsf{M}[r_{\gamma,i}], \mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,i}] \oplus r_{\gamma,i}\Delta_\mathsf{A}) := G_{\gamma,i} \oplus H(\mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}, \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}, \gamma, i)$. Then $\mathsf{P_B}$ checks that $(r_{\gamma,i}, \mathsf{K}[r_{\gamma,i}], \mathsf{M}[r_{\gamma,i}])$ is valid and, if so, computes $z_\gamma \oplus \lambda_\gamma := (s_{\gamma,i} \oplus r_{\gamma,i})$ and $\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma} := (\mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,i}] \oplus r_{\gamma,i}\Delta_\mathsf{A}) \oplus \mathsf{M}[s_{\gamma,i}]$.

Output determination:

8. For each $w \in \mathcal{O}$, $\mathsf{P_A}$ sends $(r_w, \mathsf{M}[r_w])$ to $\mathsf{P_B}$, who checks that $(r_w, \mathsf{K}[r_w], \mathsf{M}[r_w])$ is valid. If so, $\mathsf{P_B}$ outputs $z_w := (z_w \oplus \lambda_w) \oplus r_w \oplus s_w$.

Figure 3.2: Our protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. Here $\rho = \kappa$ for clarity, but this is not necessary (cf. Section 3.5).

securely computes f against malicious adversaries with statistical security $2^{-\rho}$ in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model.

Proof. We consider separately a malicious $\mathsf{P_A}$ and $\mathsf{P_B}$.

Malicious $\mathsf{P_A}$. Let $\mathcal{A}$ be an adversary corrupting $\mathsf{P_A}$. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of $\mathsf{P_A}$ in the ideal world involving an ideal functionality $\mathcal{F}$ evaluating f. $\mathcal{S}$ is defined as follows.

1–4 $\mathcal{S}$, acting as an honest $\mathsf{P_B}$, interacts with $\mathcal{A}$. The simulator also plays the role of $\mathcal{F}_{\mathsf{Pre}}$, recording all values received from and sent to $\mathcal{A}$, as well as all values that would have been sent to $\mathsf{P_B}$.

5 $\mathcal{S}$ interacts with $\mathcal{A}$ while acting as an honest $\mathsf{P_B}$ using input y equal to the 0-string.

6 For each wire $w \in \mathcal{I}_1$, $\mathcal{S}$ receives $\hat{x}_w$ and computes $x_w := \hat{x}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{Pre}}$ in the previous steps. $\mathcal{S}$ sends $x = \{x_w\}_{w \in \mathcal{I}_1}$ to $\mathcal{F}$.

7–8 $\mathcal{S}$, acting as an honest $\mathsf{P_B}$, interacts with $\mathcal{A}$. If $\mathsf{P_B}$ would abort, $\mathcal{S}$ sends abort to $\mathcal{F}$; otherwise, it sends $\mathsf{continue}$ to $\mathcal{F}$. Finally, it outputs whatever $\mathcal{A}$ outputs.

We show that the joint distribution of the outputs of $\mathcal{A}$ and the honest $\mathsf{P_B}$ in the real world is indistinguishable from the joint distribution of the outputs of $\mathcal{S}$ and $\mathsf{P_B}$ in the ideal world. We prove this by considering a sequence of experiments, the first of which corresponds to the execution of our protocol and the last of which corresponds to execution in the ideal world, and showing that successive experiments are computationally indistinguishable.

Hybrid$_1$. This is the hybrid-world protocol, where we imagine $\mathcal{S}$ playing the role of an honest $\mathsf{P_B}$ using $\mathsf{P_B}$'s actual input y, while also playing the role of $\mathcal{F}_{\mathsf{Pre}}$.

Hybrid$_2$. Same as **Hybrid$_1$**, except that in step 6, for each wire $w \in \mathcal{I}_1$ the simulator $\mathcal{S}$ receives $\hat{x}_w$ and computes $x_w := \hat{x}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{Pre}}$; it then sends $x = \{x_w\}_{w \in \mathcal{I}_1}$ to $\mathcal{F}$. In steps 7–8, if an honest $\mathsf{P_B}$ would abort, $\mathcal{S}$ sends abort to $\mathcal{F}$; otherwise, it sends $\mathsf{continue}$ to $\mathcal{F}$ (and so $\mathsf{P_B}$ outputs $f(x, y)$).

The distributions on the view of $\mathcal{A}$ in $\mathbf{Hybrid_1}$ and $\mathbf{Hybrid_2}$ are identical. Lemma 3.2.1 shows that $\mathsf{P_B}$ generates the same output in both experiments except with probability at most $2^{-\rho}$.

$\mathbf{Hybrid_3}$. Same as $\mathbf{Hybrid_2}$, except that $\mathcal{S}$ sets y equal to the 0-string throughout the protocol.

The distributions on the view of $\mathcal{A}$ in $\mathbf{Hybrid_3}$ and $\mathbf{Hybrid_2}$ are again identical (since the $\{s_w\}_{w\in\mathcal{I}_2}$ are uniform). Moreover, if $\mathcal{S}$ does not abort (when running the protocol as $\mathsf{P_B}$), the distribution on the output of $\mathsf{P_B}$ is the same in $\mathbf{Hybrid_3}$ and $\mathbf{Hybrid_2}$. So it only remains to show that $\mathsf{P_B}$ aborts with the same probability in both experiments.

The only place where $\mathsf{P_B}$'s abort can depend on y is in steps 7(b) and 8. However, these aborts depend on which row of a garbled gate is selected to decrypt. This selection, in turn, depends on $\lambda_\alpha \oplus z_\alpha$ and $\lambda_\beta \oplus z_\beta$, which are uniformly distributed in both experiments.

Note that $\mathbf{Hybrid_3}$ corresponds to the ideal-world execution described earlier. This completes the proof for a malicious $\mathsf{P_A}$.

Malicious $\mathsf{P_B}$. Let $\mathcal{A}$ be an adversary corrupting $\mathsf{P_B}$. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of $\mathsf{P_B}$ in the ideal world involving an ideal functionality $\mathcal{F}$ evaluating f. $\mathcal{S}$ is defined as follows.

1–4 $\mathcal{S}$, acting as an honest $\mathsf{P_A}$, interacts with $\mathcal{A}$ and also plays the role of $\mathcal{F}_{\mathsf{Pre}}$.

5 For each wire $w \in \mathcal{I}_2$, $\mathcal{S}$ receives $\hat{y}_w$ and computes $y_w := \hat{y}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{Pre}}$ in the previous steps. $\mathcal{S}$ sends $y = \{y_w\}_{w \in \mathcal{I}_2}$ to $\mathcal{F}$, which responds with $z = f(x, y)$.

6–7 $\mathcal{S}$ interacts with $\mathcal{A}$ while acting as an honest $\mathsf{P_A}$ using input x equal to the 0-string.

8 For each $w \in \mathcal{O}$, if $z'_w = z_w$, then $\mathcal{S}$ sends $(r_w, \mathsf{M}[r_w])$; otherwise, $\mathcal{S}$ sends $(\overline{r_w}, \mathsf{M}[r_w] \oplus \Delta_\mathsf{B})$, where Δ_B is the value used by $\mathcal{F}_{\mathsf{Pre}}$ in the previous steps. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We now show that the distribution on the view of $\mathcal{A}$ in the real world is indistinguishable from the distribution on the view of $\mathcal{A}$ in the ideal world. (Note $\mathsf{P_A}$ has no output.)

Hybrid$_1$. This is the hybrid-world protocol, where we imagine $\mathcal{S}$ playing the role of an honest $\mathsf{P_A}$ using $\mathsf{P_A}$'s actual input x, while also playing the role of $\mathcal{F}_{\mathsf{Pre}}$.

Hybrid$_2$. Same as **Hybrid$_1$**, except that in step 5, $\mathcal{S}$ receives $\hat{y}_w$ and computes $y_w := \hat{y}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{Pre}}$. Then $\mathcal{S}$ performs the same computation that $\mathsf{P_B}$ would in step 7, to obtain a value $\hat{z}_w$ for each $w \in \mathcal{O}$. Finally, for each $w \in \mathcal{O}$, $\mathcal{S}$ computes $r'_w := \hat{z}_w \oplus s_w \oplus z_w$ and sends $(r'_w, \mathsf{K}[r'_w] \oplus r'_w \Delta_\mathsf{B})$ to $\mathcal{A}$, where $\mathsf{K}[r'_w], \Delta_\mathsf{B}$ are the values used by $\mathcal{F}_{\mathsf{Pre}}$.

Noting that $\hat{z}_w = z_w \oplus \lambda_w$, we see that the distributions on the view of $\mathcal{A}$ in **Hybrid$_2$** and **Hybrid$_1$** are identical.

Hybrid$_3$. Same as **Hybrid$_2$**, except that in step 6, $\mathcal{S}$ uses x equal to the 0-string.

It follows from the security of garbling with H modeled as a random oracle that the distributions on the views of $\mathcal{A}$ in **Hybrid$_2$** and **Hybrid$_1$** are computationally indistinguishable.

Note that **Hybrid$_3$** is identical to the ideal-world execution. □

Lemma 3.2.1. *Let $\mathsf{P_B}$ have input y. Consider an $\mathcal{A}$ corrupting $\mathsf{P_A}$ and let $x_w := \hat{x}_w \oplus s_w \oplus r_w$, where $\hat{x}_w$ is the value $\mathcal{A}$ sends to $\mathsf{P_B}$ in step 6 and s_w, r_w are the values used by $\mathcal{F}_{\mathsf{Pre}}$. Except with probability at most $2^{-\rho}$, either $\mathsf{P_B}$ aborts or $\mathsf{P_B}$ outputs $z^* = f(x, y)$.*

Proof. For a wire w, let $\hat{z}_w$ be the masked value computed by $\mathsf{P_B}$ on that wire during the protocol, and let z^*_w be the value on that wire when $f(x, y)$ is computed with x defined as in the lemma. For $w \in \mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{W}$, define $\lambda_w = r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{Pre}}$; for each XOR gate $(\alpha, \beta, \gamma, \oplus)$, inductively define $\lambda_w = \lambda_\alpha \oplus \lambda_\beta$.

We prove by induction that, except with probability at most $2^{-\rho}$, if $\mathsf{P_B}$ does not abort then $z^*_w = \hat{z}_w \oplus \lambda_w$ for all w.

Base step: It is obvious that $z^*_w = \hat{z}_w \oplus \lambda_w$ for all $w \in \mathcal{I}_1 \cup \mathcal{I}_2$, unless $\mathcal{A}$ is able to forge an IT-MAC.

Induction step: Consider a gate $(\alpha, \beta, \gamma, T)$, where the stated invariant holds for wires α, β. We show that $z^*_\gamma = \hat{z}_\gamma \oplus \lambda_w$.

- $T = \oplus$: Here we have $\hat{z}_\gamma = \hat{z}_\alpha \oplus \hat{z}_\beta$ and $z^*_\gamma = z^*_\alpha \oplus z^*_\beta$. Since $\lambda_\gamma = \lambda_\alpha \oplus \lambda_\beta$, the invariant trivially holds for γ.

Functionality $\mathcal{F}_{\mathsf{abit}}$

Honest case:

1. Upon receiving init from both parties the first time, choose uniform $\Delta_{\mathsf{A}}, \Delta_{\mathsf{B}} \in \{0,1\}^{\rho}$ and send Δ_{A} to P_{A} and Δ_{B} to P_{B}.
2. Upon receiving $(\mathsf{random}, \mathsf{A})$ from both parties, choose uniform $x \in \{0,1\}$ and $\mathsf{M}[x], \mathsf{K}[x] \in \{0,1\}^{\rho}$ with $\mathsf{M}[x] = \mathsf{K}[x] \oplus x\Delta_B$. Then send $(x, \mathsf{M}[x])$ to P_{A} and $\mathsf{K}[x]$ to P_{B}.
3. Upon receiving $(\mathsf{random}, \mathsf{B})$ from both parties, generate an authenticated bit for P_{B} in a manner symmetric to the above.

Corrupted parties: A corrupted party gets to specify the randomness used on its behalf by the functionality.

Figure 3.3: The authenticated-bit functionality.

- $T = \wedge$: Here $z^*_\gamma = z^*_\alpha \wedge z^*_\beta$. Assuming P_{B} does not abort, the only way P_{B} can compute $\hat{z}_\gamma \neq z^*_\gamma \oplus \lambda_\gamma$ is if $\mathcal{A}$ forges an IT-MAC.

In particular, except with probability at most $2^{-\rho}$, we have $\hat{z}_w = z^*_w \oplus \lambda_w$ for all $w \in \mathcal{O}$. It follows that if P_{B} does not abort, it outputs z^* unless $\mathcal{A}$ forges an IT-MAC. $\square$

3.3 Efficiently Realizing $\mathcal{F}_{\mathsf{Pre}}$

Here we show how to realize $\mathcal{F}_{\mathsf{Pre}}$ efficiently using an optimized version of the TinyOT protocol.

Our protocol relies on a stateful ideal functionality $\mathcal{F}_{\mathsf{abit}}$ (cf. Figure 3.3) for generating authenticated bits using uniform values of $\Delta_{\mathsf{A}}, \Delta_{\mathsf{B}} \in \{0,1\}^{\kappa}$ that are preserved across executions [45, 37]. Technically, the functionality also allows the adversary to make "global-key queries" that correspond to a guess about the honest party's value of Δ. Both these features are preserved in all our ideal functionalities (including $\mathcal{F}_{\mathsf{Pre}}$), but we suppress explicit mention of them in our descriptions. (Note that the global-key queries have little effect on security, since the probability that the attacker can correctly guess the honest party's value of Δ using polynomially many queries is negligible. One can also verify that they can be easily incorporated into our security proofs.)

Recall that $\mathcal{F}_{\mathsf{Pre}}$ can be used to generate authenticated values $[x_1]_{\mathsf{A}}$, $[y_1]_{\mathsf{A}}$, $[z_1]_{\mathsf{A}}$, $[x_2]_{\mathsf{B}}$, $[y_2]_{\mathsf{B}}$, and $[z_2]_{\mathsf{B}}$ such that $z_1 \oplus z_2 = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$; we refer to these collectively as an *AND triple*. In the original TinyOT protocol, the four terms that result from expanding $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$ for an AND triple (namely, x_1y_1, x_1y_2, x_2y_1, and x_2y_2) are computed individually and then combined. In our new approach, we instead compute AND triples directly.

At a high level, we use three steps to compute an AND triple.

1. The parties jointly compute $[x_1]_{\mathsf{A}}$, $[y_1]_{\mathsf{A}}$, $[z_1]_{\mathsf{A}}$, $[x_2]_{\mathsf{B}}$, $[y_2]_{\mathsf{B}}$, $[z_2]_{\mathsf{B}}$, such that if both parties are honest, these are a correct AND triple. If a party cheats, that party can modify z_2 but cannot learn the other party's bits.

2. The parties perform a checking protocol that ensures the correctness of every AND triple, while letting the malicious party guess the value of x_1 (resp., x_2). Each such guess is correct with probability 1/2, but an incorrect guess is detected and will cause the other party to abort.

   As a consequence, we can argue that (conditioned on no abort) the malicious party obtains information on at most ρ AND triples except with probability at most $2^{-\rho}$.

3. So far we have described a way for the parties to generate many "leaky" AND triples such that the attacker may have disallowed information on at most ρ of them. We then show how to distill these into a smaller number of "private" AND triples, about which the attacker is guaranteed to have no disallowed information.

Overall, when using bucket size B (see Section 3.3.2) our new TinyOT protocol requires only $(5\kappa + 3\rho)B$ bits of communication per AND triple, while the original TinyOT protocol requires $(14\kappa + 8\rho)B$ bits of communication even taking optimizations into account. For $\kappa = 128$ and $\rho = 40$, this is an improvement of $2.78\times$.

3.3.1 Half-Authenticated AND Triples

We first show a protocol that realizes a functionality in which only the x's in an AND triple are authenticated. This will serve as a building block in the following sections. This functionality, called $\mathcal{F}_{\mathsf{HaAND}}$, is described in Figure 3.4. It outputs authenticated bits $[x_1]_{\mathsf{A}}$ and $[x_2]_{\mathsf{B}}$ to the two parties. It also takes y_1 from $\mathsf{P_A}$ and y_2 from $\mathsf{P_B}$, and outputs shares of $x_1 y_2 \oplus x_2 y_1$. (Note that the parties can then locally compute $x_1 y_1$ and $x_2 y_2$, respectively, and thus generate shares of $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$.) In Figure 3.5 we show a protocol that realizes $\mathcal{F}_{\mathsf{HaAND}}$ in the $\mathcal{F}_{\mathsf{abit}}$-hybrid model.

Lemma 3.3.1. *If H is modeled as a random oracle, the protocol in Figure 3.5 securely implements $\mathcal{F}_{\mathsf{HaAND}}$ in the $\mathcal{F}_{\mathsf{abit}}$-hybrid model.*

Proof. We first show correctness. Note that $s_2 = s_1 \oplus x_2 y_1$, so $s_1 \oplus s_2 = x_2 y_1$.

Similarly, $t_1 \oplus t_2 = x_1 y_2$. Thus, v_1 and v_2 are shares of $x_1 y_2 \oplus x_2 y_1$. Moreover, when both parties are honest v_1 and v_2 are individually uniform.

We next prove security. We consider the case of a malicious $\mathsf{P_A}$; the case of a malicious $\mathsf{P_B}$ is symmetric (and is, in fact, easier since $\mathsf{P_B}$ sends (H_0, H_1) before $\mathsf{P_A}$). The simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{abit}}$, and stores all shares of $[x_1]_{\mathsf{A}}$ and $[x_2]_{\mathsf{B}}$, as well as global keys Δ_{A}, Δ_{B}.

2. $\mathcal{S}$ chooses uniform H_0, H_1 and sends them to $\mathcal{A}$. Let $t_1' := H_{x_1} \oplus H(\mathsf{M}[x_1])$.

3. $\mathcal{S}$ receives (H_0', H_1') from $\mathcal{A}$, and computes $s_0' := H_0' \oplus H(\mathsf{K}[x_2])$, $s_1' := H_1' \oplus H(\mathsf{K}[x_2] \oplus \Delta_{\mathsf{A}})$, and $y_1 := s_0' \oplus s_1'$. It sets $v_1 := s_0' \oplus t_1'$, and sends y_1, v_1 to $\mathcal{F}_{\mathsf{HaAND}}$ on behalf of $\mathsf{P_A}$. It then outputs whatever $\mathcal{A}$ does.

It is not hard to see that, if H is modeled as a random oracle, the distribution on the view of $\mathcal{A}$ in the ideal-world execution described above is computationally indistinguishable from the view of $\mathcal{A}$ in the real-world execution of the protocol. Let x_2, y_2 denote the authenticated bit $\mathsf{P_B}$ received and $\mathsf{P_B}$'s input, respectively. In a real-world execution of the protocol with transcript (H_0, H_1, H_0', H_1'), the value output by $\mathsf{P_B}$ would be

$$\begin{aligned} s_2 \oplus t_1 &= s_{x_2}' \oplus (t_1' \oplus x_1 y_2) \\ &= (1 \oplus x_2) s_0' \oplus x_2 s_1' \oplus t_1' \oplus x_1 y_2 \\ &= s_0' \oplus x_2 (s_0' \oplus s_1') \oplus t_1' \oplus x_1 y_2, \end{aligned}$$

Functionality $\mathcal{F}_{\mathsf{HaAND}}$

Honest case:

1. Generate uniform $[x_1]_{\mathsf{A}}$ and $[x_2]_{\mathsf{B}}$ and send the respective shares to the two parties.
2. Upon receiving y_1 from $\mathsf{P_A}$ and y_2 from $\mathsf{P_B}$, choose uniform v_1 and send v_1 to $\mathsf{P_A}$ and $v_2 := v_1 \oplus (x_1 y_2 \oplus x_2 y_1)$ to $\mathsf{P_B}$.

Corrupted parties: A corrupted party gets to specify the randomness used on its behalf by the functionality.

Figure 3.4: Functionality $\mathcal{F}_{\mathsf{HaAND}}$ for computing a half-authenticated AND triple.

Protocol Π_{HaAND}

$\mathsf{P_A}$ and $\mathsf{P_B}$ have input y_1 and y_2, respectively.

Protocol:

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ call $\mathcal{F}_{\mathsf{abit}}$ to obtain $[x_1]_{\mathsf{A}}$ and $[x_2]_{\mathsf{B}}$, i.e., $\mathsf{P_A}$ receives $(x_1, \mathsf{M}[x_1], \mathsf{K}[x_2])$ and $\mathsf{P_B}$ receives $(x_2, \mathsf{M}[x_2], \mathsf{K}[x_1])$.
2. $\mathsf{P_B}$ chooses uniform $t_1 \in \{0,1\}$ and computes $H_0 := H(\mathsf{K}[x_1]) \oplus t_1$, $H_1 := H(\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}}) \oplus t_1 \oplus y_2$. $\mathsf{P_B}$ sends (H_0, H_1) to $\mathsf{P_A}$, who computes $t_2 := H_{x_1} \oplus H(\mathsf{M}[x_1])$.
3. $\mathsf{P_A}$ chooses uniform $s_1 \in \{0,1\}$ and then computes $H'_0 := H(\mathsf{K}[x_2]) \oplus s_1$, $H'_1 := H(\mathsf{K}[x_2] \oplus \Delta_{\mathsf{A}}) \oplus s_1 \oplus y_1$. $\mathsf{P_A}$ sends (H'_0, H'_1) to $\mathsf{P_B}$, who computes $s_2 := H'_{x_2} \oplus H(\mathsf{M}[x_2])$.
4. $\mathsf{P_A}$ outputs $v_1 := s_1 \oplus t_2$, and $\mathsf{P_B}$ outputs $v_2 := s_2 \oplus t_1$.

Figure 3.5: Protocol Π_{HaAND} realizing $\mathcal{F}_{\mathsf{HaAND}}$.

which matches the value

$$v_1 \oplus (x_1 y_2 \oplus x_2 y_1) = (s'_0 \oplus t'_1) \oplus x_1 y_2 \oplus x_2 (s'_0 \oplus s'_1)$$

that $\mathsf{P_B}$ outputs in the ideal-world execution. □

Functionality $\mathcal{F}_{\mathsf{LaAND}}$

Honest case:

1. Generate uniform $[x_1]_{\mathsf{A}}, [y_1]_{\mathsf{A}}, [z_1]_{\mathsf{A}}$, $[x_2]_{\mathsf{B}}, [y_2]_{\mathsf{B}}, [z_2]_{\mathsf{B}}$ such that $z_1 \oplus z_2 = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$, and send the respective shares to the two parties.
2. $\mathsf{P_A}$ can choose to send a bit b. If $b = x_2$, the functionality sends correct to $\mathsf{P_A}$. If $b \neq x_2$, the functionality sends fail to both parties and abort.

Corrupted parties: A corrupted party gets to specify the randomness used on its behalf by the functionality.

Figure 3.6: Functionality $\mathcal{F}_{\mathsf{LaAND}}$ for computing a leaky AND triple.

3.3.2 Leaky AND Triples

The leaky-AND functionality $\mathcal{F}_{\mathsf{LaAND}}$ is described in Figure 3.6. This functionality generates authenticated values $[x_1]_{\mathsf{A}}$, $[y_1]_{\mathsf{A}}$, $[z_1]_{\mathsf{A}}$, $[x_2]_{\mathsf{B}}$, $[y_2]_{\mathsf{B}}$, and $[z_2]_{\mathsf{B}}$ such that $z_1 \oplus z_2 = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$, but allows a malicious $\mathsf{P_A}$ (resp., $\mathsf{P_B}$) to guess x_2 (resp., x_1). This guess is correct with probability 1/2, but an incorrect guess is revealed to the other party (who can then abort).

To realize this functionality, we begin by having the parties generate authenticated bits $[y_1]_{\mathsf{A}}$, $[z_1]_{\mathsf{A}}$, $[y_2]_{\mathsf{B}}$, and then use $\mathcal{F}_{\mathsf{HaAND}}$ to generate $[x_1]_{\mathsf{A}}$, $[x_2]_{\mathsf{B}}$ and shares of $x_1 y_2 \oplus x_2 y_1$. The parties can then locally compute shares of $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$. Note that $\mathsf{P_A}$ (resp., $\mathsf{P_B}$) can easily misbehave by, for example, sending an incorrect value of y_1 (resp., y_2) to $\mathcal{F}_{\mathsf{HaAND}}$. We address this in the next step. Looking ahead, however, we note that the way we address this issue introduces a selective-failure attack that can leak information to the attacker: if the attacker flips a y-value but the checking step described next does not abort, then it must be the case that

Protocol Π_{LaAND}

Protocol:

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ obtain random authenticated bits $[y_1]_\mathsf{A}, [z_1]_\mathsf{A}, [y_2]_\mathsf{B}, [r]_\mathsf{B}$. $\mathsf{P_A}$ and $\mathsf{P_B}$ also calls $\mathcal{F}_{\mathsf{HaAND}}$, receiving $[x_1]_\mathsf{A}$ and $[x_2]_\mathsf{B}$.
2. $\mathsf{P_A}$ picks a random bit v_1 and sends (y_1, v_1) to $\mathcal{F}_{\mathsf{HaAND}}$; $\mathsf{P_B}$ sends y_2 to $\mathcal{F}_{\mathsf{HaAND}}$, which sends v_2 to $\mathsf{P_B}$.
3. $\mathsf{P_A}$ computes $u = v_1 \oplus x_1 y_1 \oplus z_1$ and sends to $\mathsf{P_B}$. $\mathsf{P_B}$ computes $z_2 := u \oplus x_2 y_2 \oplus v_2$ and sends $d := r \oplus z_2$ to $\mathsf{P_A}$. Two parties compute $[z_2]_\mathsf{B} = [r]_\mathsf{B} \oplus d$.
4. $\mathsf{P_B}$ checks correctness as follows:
   (a) $\mathsf{P_B}$ computes:
   $T_0 := H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2 \Delta_\mathsf{B})$
   $U_0 := T_0 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_\mathsf{B})$
   $T_1 := H(\mathsf{K}[x_1], \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_\mathsf{B})$
   $U_1 := T_1 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[z_1] \oplus z_2 \Delta_\mathsf{B})$
   (b) $\mathsf{P_B}$ sends U_{x_2} to $\mathsf{P_A}$.
   (c) $\mathsf{P_A}$ chooses a uniform κ-bit string R and computes:
   $V_0 := H(\mathsf{M}[x_1], \mathsf{M}[z_1])$ $\quad V_1 := H(\mathsf{M}[x_1], \mathsf{M}[z_1] \oplus \mathsf{M}[y_1])$
   $W_{0,0} := H(\mathsf{K}[x_2]) \oplus V_0 \oplus R$ $\quad W_{0,1} := H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_1 \oplus R$
   $W_{1,0} := H(\mathsf{K}[x_2]) \oplus V_1 \oplus U_0 \oplus R$ $\quad W_{1,1} := H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_0 \oplus U_1 \oplus R$
   (d) $\mathsf{P_A}$ sends $W_{x_1,0}, W_{x_1,1}$ to $\mathsf{P_B}$ and sends R to $\mathcal{F}_{\mathsf{EQ}}$.
   (e) $\mathsf{P_B}$ computes $R' := W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_{x_2}$ and sends R' to $\mathcal{F}_{\mathsf{EQ}}$.
5. $\mathsf{P_A}$ checks correctness as follows:
   (a) $\mathsf{P_A}$ computes:
   $T_0 := H(\mathsf{K}[x_2], \mathsf{K}[z_2] \oplus z_1 \Delta_\mathsf{A})$
   $U_0 := T_0 \oplus H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}, \mathsf{K}[y_2] \oplus \mathsf{K}[z_2] \oplus (y_1 \oplus z_1)\Delta_\mathsf{A})$
   $T_1 := H(\mathsf{K}[x_2], \mathsf{K}[y_2] \oplus \mathsf{K}[z_2] \oplus (y_1 \oplus z_1)\Delta_\mathsf{A})$
   $U_1 := T_1 \oplus H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}, \mathsf{K}[z_2] \oplus z_1 \Delta_\mathsf{A})$
   (b) $\mathsf{P_A}$ sends U_{x_1} to $\mathsf{P_B}$.
   (c) $\mathsf{P_B}$ chooses a uniform κ-bit string R and computes:
   $V_0 := H(\mathsf{M}[x_2], \mathsf{M}[z_2])$ $\quad V_1 := H(\mathsf{M}[x_2], \mathsf{M}[z_2] \oplus \mathsf{M}[y_2])$
   $W_{0,0} := H(\mathsf{K}[x_1]) \oplus V_0 \oplus R$ $\quad W_{0,1} := H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}) \oplus V_1 \oplus R$
   $W_{1,0} := H(\mathsf{K}[x_1]) \oplus V_1 \oplus U_0 \oplus R$ $\quad W_{1,1} := H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}) \oplus V_0 \oplus U_1 \oplus R$
   (d) $\mathsf{P_B}$ sends $W_{x_2,0}, W_{x_2,1}$ to $\mathsf{P_A}$ and sends R to $\mathcal{F}_{\mathsf{EQ}}$,
   (e) $\mathsf{P_A}$ computes $R' := W_{x_2,x_1} \oplus H(\mathsf{M}[x_1]) \oplus T_{x_1}$ and sends R' to $\mathcal{F}_{\mathsf{EQ}}$.

Figure 3.7: Protocol Π_{LaAND} realizing $\mathcal{F}_{\mathsf{LaAND}}$.

$x_1 \oplus x_2 = 0$.

Checking correctness. Now both parties check correctness of the AND triples

Functionality $\mathcal{F}_{\mathsf{aAND}}$

Honest case: Generate uniform $[x_1]_{\mathsf{A}}, [y_1]_{\mathsf{A}}, [z_1]_{\mathsf{A}}$, and $[x_2]_{\mathsf{B}}, [y_2]_{\mathsf{B}}, [z_2]_{\mathsf{B}}$, such that $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) = z_1 \oplus z_2$.

Corrupted parties: A corrupted party gets to specify the randomness used on its behalf by the functionality.

Figure 3.8: Functionality $\mathcal{F}_{\mathsf{aAND}}$ for generating AND triples

Protocol Π_{aAND}

Protocol:

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ call $\mathcal{F}_{\mathsf{LaAND}}$ a total of $\ell' = \ell B$ times to obtain $\{[x_1^i]_{\mathsf{A}}, [y_1^i]_{\mathsf{A}}, [z_1^i]_{\mathsf{A}}, [x_2^i]_{\mathsf{B}}, [y_2^i]_{\mathsf{B}}, [z_2^i]_{\mathsf{B}}\}_{i=1}^{\ell'}$.
2. $\mathsf{P_A}$ and $\mathsf{P_B}$ use coin tossing to randomly partition the results into ℓ buckets, each containing B AND triples.
3. For each bucket, the parties combine B leaky ANDs into one non-leaky AND. To combine two leaky ANDs $([x_1']_{\mathsf{A}}, [y_1']_{\mathsf{A}}, [z_1']_{\mathsf{A}}, [x_2']_{\mathsf{B}}, [y_2']_{\mathsf{B}}, [z_2']_{\mathsf{B}})$ and $([x_1'']_{\mathsf{A}}, [y_1'']_{\mathsf{A}}, [z_1'']_{\mathsf{A}}, [x_2'']_{\mathsf{B}}, [y_2'']_{\mathsf{B}}, [z_2'']_{\mathsf{B}})$ do:
   (a) The parties reveal $d' := y_1' \oplus y_1'', d'' = y_2' \oplus y_2''$ along with their MACs, and compute $d := d' \oplus d''$ if the MACs verify.
   (b) Set $[x_1]_{\mathsf{A}} := [x_1']_{\mathsf{A}} \oplus [x_1'']_{\mathsf{A}}$, $[x_2]_{\mathsf{B}} := [x_2']_{\mathsf{B}} \oplus [x_2'']_{\mathsf{B}}$, $[y_1]_{\mathsf{A}} := [y_1']_{\mathsf{A}}$, $[y_2]_{\mathsf{B}} := [y_2']_{\mathsf{B}}$, $[z_1]_{\mathsf{A}} := [z_1']_{\mathsf{A}} \oplus [z_1'']_{\mathsf{A}} \oplus d[x_1'']_{\mathsf{A}}$, $[z_2]_{\mathsf{B}} := [z_2']_{\mathsf{B}} \oplus [z_2'']_{\mathsf{B}} \oplus d[x_2'']_{\mathsf{B}}$.

   The parties iterate over all B leaky AND triples one-by-one, taking the resulting triple and combining it with the next one.

Figure 3.9: Protocol Π_{aAND} realizing $\mathcal{F}_{\mathsf{aAND}}$.

generated in the previous step. If $x_2 \oplus x_1 = 0$, then we want to check that $z_2 = z_1$; if $x_2 \oplus x_1 = 1$, then we want to to check that $y_1 \oplus z_1 = y_2 \oplus z_2$. However, an obvious problem is that neither party knows the value of $x_1 \oplus x_2$; therefore there is no way to know which relationship should be checked. We thus need to construct a checking procedure such that the computation of $\mathsf{P_A}$ is oblivious to x_2, while the computation of $\mathsf{P_B}$ is oblivious to x_1.

We describe the intuition from the point of view of an honest $\mathsf{P_B}$ holding $x_2 = 0$. Abstractly, the first step is for $\mathsf{P_B}$ to compute values T_0 and U_0 and to send U_0 to $\mathsf{P_A}$; $\mathsf{P_A}$ will then compute V_0 such that if $x_1 = 0$ then $V_0 = T_0$, but if $x_1 = 1$ then $V_0 \oplus U_0 = T_0$. We set things up such that if the AND triple is incorrect, then $\mathsf{P_A}$ cannot compute V_0 correctly. Similar constructs (namely V_1, U_1, and T_1) are computed if $x_2 = 1$. Now, depending on the value of x_1 and x_2, parties need to perform an equality comparison between different values, as summarized below.

	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	$V_0 = T_0$	$V_0 \oplus U_0 = T_0$
$x_2 = 1$	$V_1 = T_1$	$V_1 \oplus U_1 = T_1$

Unfortunately, a direct comparison is not possible since $\mathsf{P_A}$ does not know the value of x_2 and therefore does not know which comparison to perform. Our idea is to transform $\mathsf{P_A}$'s computation such that it is oblivious to x_2. In detail: if $x_1 = 0$, then $\mathsf{P_A}$ computes V_0 as if $x_2 = 0$ and computes V_1 as if $x_2 = 1$. Then $\mathsf{P_A}$ "encrypts" V_0 and V_1 such that $\mathsf{P_B}$ can only decrypt V_{x_2}. $\mathsf{P_B}$ can then locally check whether $V_{x_2} = T_{x_2}$. In the case when $x_1 = 1$, $\mathsf{P_A}$ computes and encrypts $V_0 \oplus U_0$ and $V_1 \oplus U_1$ in a similar manner.

A problem is that although a malicious $\mathsf{P_A}$ cannot cheat, a malicious $\mathsf{P_B}$ will not be caught on an incorrect AND triple because $\mathsf{P_B}$ compares the results locally and $\mathsf{P_A}$ does not learn the result of the comparison! To solve this, we let $\mathsf{P_A}$ instead send the encrypted values $V_0 \oplus R$ and $V_1 \oplus R$, for a uniform R, such that $\mathsf{P_B}$ can obtain $V_{x_2} \oplus R$, and learn R from it. Now $\mathsf{P_A}$ and $\mathsf{P_B}$ can check the equality on R using the $\mathcal{F}_{\mathsf{EQ}}$ functionality that allows both parties get the outcome. (If a party aborts, that is also detected as cheating.) Finally, the same check is performed in the opposite direction to convince both parties of the correctness of the triples.

A complete description of the protocol is shown in Figure 3.7.

3.3.3 Proof for the Leaky-AND protocol

In the following, we will discuss at a high-level how the proof works for the new TinyOT protocol. We will focus on the security of the Π_{LaAND} protocol, since the security of the Π_{aAND} protocol is fairly straightforward given the proof in the original paper [45].

Correctness

We want to show that if both parties follow the protocol then in step 4.e $W_{x_1,x_2} \oplus \mathsf{M}[x_2] \oplus T_{x_2} = R$. The checks in step 5 are symmetric to those in step 4. We proceed on a case-by-case basis.

Case 1: $x_1 = 0, x_2 = 0$**.**

Here we have $\mathsf{M}[x_1] = \mathsf{K}[x_1]$ and $\mathsf{M}[x_2] = \mathsf{K}[x_2]$. Since $x_1 \oplus x_2 = 0$, we know that $z_1 = z_2$, which further implies that

$$\mathsf{M}[z_1] = \mathsf{K}[z_1] \oplus z_1\Delta_{\mathsf{B}} = \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}}$$

. The desired equation thus holds because:

$$\begin{aligned}
&W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_{x_2} \\
&\quad = H(\mathsf{K}[x_2]) \oplus V_0 \oplus R \oplus H(\mathsf{M}[x_2]) \oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}}) \\
&\quad = V_0 \oplus T_0 \oplus R \\
&\quad = H(\mathsf{M}[x_1], \mathsf{M}[z_1]) \oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}}) \oplus R \\
&\quad = R.
\end{aligned}$$

Case 2: $x_1 = 0, x_2 = 1$.

Similar to the previous case, we know that $\mathsf{M}[x_1] = \mathsf{K}[x_1]$ and that $\mathsf{M}[x_2] = \mathsf{K}[x_2] \oplus \Delta_{\mathsf{B}}$. Then $x_1 \oplus x_2 = 1$ implies

$$\begin{aligned}
&\mathsf{M}[z_1] \oplus \mathsf{M}[y_1] \\
&\quad = \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_1 \oplus z_1)\Delta_{\mathsf{B}} \\
&\quad = \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_{\mathsf{B}}.
\end{aligned}$$

The desired equation thus holds because:

$$\begin{aligned}
&W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_{x_2} \\
&\quad = W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_1 \\
&\quad = H(\mathsf{K}[x_2] \oplus \Delta_{\mathsf{A}}) \oplus V_1 \oplus R \oplus H(\mathsf{M}[x_2]) \oplus T_1 \\
&\quad = V_1 \oplus T_1 \oplus R \\
&\quad = H(\mathsf{M}[x_1], \mathsf{M}[z_1] \oplus \mathsf{M}[y_1]) \\
&\qquad \oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}} \oplus \mathsf{K}[y_1] \oplus y_2\Delta_{\mathsf{B}}) \oplus R \\
&\quad = R.
\end{aligned}$$

Case 3: $x_1 = 1, x_2 = 0$.

Similar to the previous cases, we know that $\mathsf{M}[x_1] = \mathsf{K}[x_1] \oplus \Delta_\mathsf{B}$, $\mathsf{M}[x_2] = \mathsf{K}[x_2]$, and $\mathsf{M}[z_1] \oplus \mathsf{M}[y_1] = \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_\mathsf{B}$. Therefore:

$$\begin{aligned}
&W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_{x_2} \\
&\quad = W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_0 \\
&\quad = H(\mathsf{K}[x_2]) \oplus V_1 \oplus U_0 \oplus R \oplus H(\mathsf{M}[x_2]) \oplus T_0 \\
&\quad = V_1 \oplus U_0 \oplus R \oplus T_0 \\
&\quad = H(\mathsf{M}[x_1], \mathsf{M}[z_1] \oplus \mathsf{M}[y_1]) \oplus R \oplus T_0 \\
&\qquad \oplus T_0 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_\mathsf{B}) \\
&\quad = R.
\end{aligned}$$

Case 4: $x_1 = 1, x_2 = 1$.

Similar to the previous cases, we know that $\mathsf{M}[x_1] = \mathsf{K}[x_1] \oplus \Delta_\mathsf{B}$, $\mathsf{M}[x_2] = \mathsf{K}[x_2] \oplus \Delta_\mathsf{B}$, and $\mathsf{M}[z_1] = \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B}$. Therefore:

$$\begin{aligned}
&W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_{x_2} \\
&\quad = W_{x_1,x_2} \oplus H(\mathsf{M}[x_2]) \oplus T_1 \\
&\quad = H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_0 \oplus U_1 \oplus R \oplus H(\mathsf{M}[x_2]) \oplus T_1 \\
&\quad = V_0 \oplus U_1 \oplus R \oplus T_1 \\
&\quad = H(\mathsf{M}[x_1], \mathsf{M}[z_1]) \oplus R \oplus T_1 \\
&\qquad \oplus T_1 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B}) \\
&\quad = R.
\end{aligned}$$

We next prove security.

Lemma 3.3.2. *If $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \neq (z_1 \oplus z_2)$ then the protocol will result in* *an abort except with negligible probability.*

We will proceed on a case-by-case basis. Without loss of generality, we assume P_B is honest and P_A is corrupted.

Case 1: $x_1 = 0, x_2 = 0$**.**

To pass the test, the adversary would have to produce a pair R and $W_{0,0}$ such that:

$$W_{0,0} = H(\mathsf{M}[x_2]) \oplus T_{x_2} \oplus R$$

$$W_{0,0} = H(\mathsf{M}[x_2]) \oplus R$$

$$\oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}}).$$

Since $z_1 \oplus z_2 = 1$, this means the adversary must compute $\mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}} = \mathsf{M}[z_1] \oplus \Delta_{\mathsf{B}}$. This requires guessing a κ-bit MAC and is thus computationally infeasible. Alternatively, the adversary could try to compute T_0 from $U_0 = T_0 \oplus H(\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}}, \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_{\mathsf{B}})$. Fortunately, since $\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}} = \mathsf{M}[x_1] \oplus \Delta_{\mathsf{B}}$, this is also infeasible.

Case 2: $x_1 = 0, x_2 = 1$**.**

To pass the test, the adversary would have to produce a pair R and $W_{0,1}$ such that:

$$W_{0,1} = H(\mathsf{M}[x_2]) \oplus T_{x_2} \oplus R$$

$$W_{0,1} = H(\mathsf{M}[x_2]) \oplus R$$

$$\oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}} \oplus \mathsf{K}[y_1] \oplus y_2\Delta_{\mathsf{B}}).$$

However, since $z_1 \oplus z_2 \oplus y_1 \oplus y_2 = 1$, the last line requires the adversary to compute $\mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (z_2 \oplus y_2)\Delta_{\mathsf{B}} = \mathsf{M}[y_1] \oplus \mathsf{M}[z_1] \oplus \Delta_{\mathsf{B}}$. This requires guessing a κ-bit MAC and is thus computationally infeasible. Alternatively, the adversary could try to compute T_1 from $U_1 = T_1 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B})$. Fortunately, since $\mathsf{K}[x_1] \oplus \Delta_\mathsf{B} = \mathsf{M}[x_1] \oplus \Delta_\mathsf{B}$, this is also infeasible.

Case 3: $x_1 = 1, x_2 = 0$**.**

To pass the test, the adversary would have to produce R, $W_{1,0}$ such that:

$$W_{1,0} = H(\mathsf{M}[x_2]) \oplus T_{x_2} \oplus R$$

$$W_{1,0} = H(\mathsf{M}[x_2]) \oplus R$$
$$\oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B}).$$

Since $x_1 = 1$, the last line requires the adversary to compute $\mathsf{K}[x_1] = \mathsf{M}[x_1] \oplus \Delta_\mathsf{B}$. This requires guessing a κ-bit MAC and is thus computationally infeasible. Alternatively, the adversary could try to compute T_0 from $U_0 = T_0 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_\mathsf{B})$. Fortunately, since $y_1 \oplus y_2 \oplus z_1 \oplus z_2 = 1$ we have $\mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_\mathsf{B} = \mathsf{M}[y_1] \oplus \mathsf{M}[z_1] \oplus \Delta_\mathsf{B}$, and so this is also infeasible.

Case 4: $x_1 = 1, x_2 = 1$**.**

To pass the test, the adversary would have to produce R and $W_{1,1}$ such that:

$$W_{1,1} = H(\mathsf{M}[x_2]) \oplus T_{x_2} \oplus R$$

$$W_{1,1} = H(\mathsf{M}[x_2]) \oplus R$$
$$\oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B} \oplus \mathsf{K}[y_1] \oplus y_2\Delta_\mathsf{B}).$$

Since $x_1 = 1$, the last line requires the adversary to compute $\mathsf{K}[x_1] = \mathsf{M}[x_1] \oplus \Delta_\mathsf{B}$. This requires guessing a κ-bit MAC and is thus computationally infeasible. Alternatively, the adversary could try to compute T_1 from $U_1 = T_1 \oplus H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}, \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B})$. Fortunately, since $z_1 \oplus z_2 = 1$ we have $\mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B} = \mathsf{M}[z_1] \oplus \Delta_\mathsf{B}$, and so this is also infeasible.

Lemma 3.3.3. *The protocol in Figure 3.7 securely realizes $\mathcal{F}_{\mathsf{LaAND}}$ in the $(\mathcal{F}_{\mathsf{abit}}, \mathcal{F}_{\mathsf{HaAND}}, \mathcal{F}_{\mathsf{EQ}})$-hybrid model.*

Proof. We consider separately the case of a malicious $\mathsf{P_A}$ and a malicious $\mathsf{P_B}$.

Malicious $\mathsf{P_A}$. We construct a simulator $\mathcal{S}$ as follows:

1 $\mathcal{S}$ receives $(x_1, \mathsf{M}[x_1])$, $(y_1, \mathsf{M}[y_1])$, $(z_1, \mathsf{M}[z_1])$, $\mathsf{K}[x_2]$, $\mathsf{K}[y_2]$, $\mathsf{K}[r]$, and Δ_A that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{abit}}$. Then $\mathcal{S}$ picks a uniform bit s, sets $\mathsf{K}[z_2] := \mathsf{K}[r] \oplus s\Delta_\mathsf{A}$, and sends $(x_1, \mathsf{M}[x_1])$, $(y_1, \mathsf{M}[y_1])$, $(z_1, \mathsf{M}[z_1])$, $\mathsf{K}[x_2]$, $\mathsf{K}[y_2]$, $\mathsf{K}[z_2]$, and Δ_A to $\mathcal{F}_{\mathsf{LaAND}}$. Functionality $\mathcal{F}_{\mathsf{LaAND}}$ then sends $(x_2, \mathsf{M}[x_2])$, $(y_2, \mathsf{M}[y_2])$, $(z_2, \mathsf{M}[z_2])$, $\mathsf{K}[x_1]$, $\mathsf{K}[y_1]$, $\mathsf{K}[z_1]$, and Δ_B to $\mathsf{P_B}$.

2–3 $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{HaAND}}$ obtaining the inputs from $\mathcal{A}$, namely y_1' and the value $\mathcal{A}$ sent, namely u'. $\mathcal{S}$ uses y_1 and u to denote the value that an honest $\mathsf{P_B}$ would use. If $y_1' \neq y_1, u' \neq u$, $\mathcal{S}$ sets $g_0 = 1 \oplus x_1$, if $y_1' \neq y_1, u' = u$, $\mathcal{S}$ sets $g_0 = x_1$.

4 $\mathcal{S}$ sends a random U^* to $\mathcal{A}$, and receives some W_0, W_1 and computes some R_0, R_1, such that, if $x_1 = 0$, $W_0 := H(\mathsf{K}[x_2]) \oplus V_0 \oplus R_0$, $W_1 := H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_1 \oplus R_1$; otherwise, $W_0 := H(\mathsf{K}[x_2]) \oplus V_1 \oplus U^* \oplus R_0$ and $W_1 := H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_0 \oplus U^* \oplus R_1$.

$\mathcal{S}$ also obtains R that $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{EQ}}$. If R does not equal to either R_0 or R_1, $\mathcal{S}$ aborts; otherwise $\mathcal{S}$ computes g_1 such that $R \neq R_{g_1}$ for some $g_1 \in \{0, 1\}$.

5 $\mathcal{S}$ receives U, picks random W_0^*, W_1^* and sends them to $\mathcal{A}$. $\mathcal{S}$ obtains R' that $\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{EQ}}$.

- If both U, R' are honestly computed, $\mathcal{S}$ proceeds as normal.
- If U is not honestly computed and that $R' = W_{x_1}^* \oplus H(\mathsf{M}[x_1]) \oplus T_{x_1}$ is honestly computed, $\mathcal{S}$ set $g_2 = 0$
- If either of the following is true: 1) $x_1 = 0$ and $R' = W_{x_1}^* \oplus H(\mathsf{M}[x_1]) \oplus U \oplus H(\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}}, \mathsf{K}[y_1] \oplus (y_2 \oplus z_2)\Delta_{\mathsf{B}})$; 2) $x_1 = 1$ and $R' = W_{x_1}^* \oplus H(\mathsf{M}[x_1]) \oplus U \oplus H(\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}}, \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}})$, $\mathcal{S}$ sets $g_2 = 1$.
- Otherwise $\mathcal{S}$ aborts.

6 For each value $g \in \{g_0, g_1, g_2\}$, if $g \neq \perp$, $\mathcal{S}$ sends g to $\mathcal{F}_{\mathsf{LaAND}}$. If $\mathcal{F}_{\mathsf{LaAND}}$ abort after any guess, $\mathcal{S}$ aborts.

Note that the first 3 steps are perfect simulations. However, a malicious $\mathsf{P_A}$ can flip the value of y_1 and/or u used. According to the unforgeability proof, the protocol will abort if the relationship $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \oplus (z_1 \oplus z_2) = 0$ does not hold. Therefore, if $\mathcal{A}$ flip y_1, it is essentially guessing that $x_1 \oplus x_2 = 0$; if $\mathcal{A}$ flip both y_1 and u, it is guessing that $x_1 \oplus x_2 = 1$. Such selective failure attack is extracted by $\mathcal{S}$ and answered accordingly.

In step 4, U^* is sent in the simulation, while U_{x_2} is sent. This is a perfect simulation unless both of the input to random oracle in U_{x_2} get queried. This does not happen during the protocol, since Δ_{B} in not known to $\mathcal{A}$. In step 5, W_0^*, W_1^* are sent in the simulation, while $W_{x_2,0}, W_{x_2,0}$ are sent in the real protocol. This is also a perfect simulation unless $\mathsf{P_A}$ gets Δ_B: both R and one of $H(\mathsf{K}[x_1])$ and $H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B})$ are random.

Another difference is that $\mathsf{P_B}$ always aborts in the simulation if G_{x_2,y_2} is not honestly computed. This is also the case in the real protocol unless $\mathcal{A}$ learns Δ_B.

Malicious $\mathsf{P_B}$. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ receives $(x_2, \mathsf{M}[x_2])$, $(y_2, \mathsf{M}[y_2])$, $(r, \mathsf{M}[r])$, $\mathsf{K}[x_1]$, $\mathsf{K}[y_1]$, $\mathsf{K}[z_1]$, Δ_B that $\mathcal{A}$ sends to $\mathcal{F}_\mathsf{abit}$. Then $\mathcal{S}$ picks a random bit s, sets

$$(z_2, \mathsf{M}[z_2]) := (r \oplus s, \mathsf{M}[z_2] \oplus s\Delta_\mathsf{B}),$$

   and sends $(x_2, \mathsf{M}[x_2])$, $(y_2, \mathsf{M}[y_2])$, $(z_2, \mathsf{M}[z_2])$, $\mathsf{K}[x_1]$, $\mathsf{K}[y_1]$, $\mathsf{K}[z_1])$ to $\mathcal{F}_\mathsf{LaAND}$. Functionality $\mathcal{F}_\mathsf{LaAND}$ then sends $(x_1, \mathsf{M}[x_1])$, $(y_1, \mathsf{M}[y_1])$, $(z_1, \mathsf{M}[z_1])$, $\mathsf{K}[x_2]$, $\mathsf{K}[y_2]$, $\mathsf{K}[z_2])$ to $\mathsf{P_B}$.

2-3 $\mathcal{S}$ plays the role of $\mathcal{F}_\mathsf{HaAND}$ and obtains y_2' $\mathcal{A}$ sent. $\mathcal{S}$ also obtains d' sent by $\mathsf{P_B}$. Denoting y_2', d as values an honest $\mathsf{P_B}$ would use, if $y_2' \neq y_2, d' \neq d$, $\mathcal{S}$ sets $g_0 = 1 \oplus x_2$, if $y_2' \neq y_2, d' = d$, $\mathcal{S}$ sets $g_0 = x_2$.

4-6 Note that step 4 and step 5 of the protocol are the same with the exception that the roles of $\mathsf{P_A}$ and $\mathsf{P_B}$ are switched. We denote S' the simulator that was defined for the case where $\mathsf{P_A}$ is corrupted. $\mathcal{S}$ will employ in step 4 the same strategy that was employed by S' in step 5. $\mathcal{S}$ will employ in step 5, the same strategy that was employed by S' in step 4.

The first three steps are perfect simulation, with a malicious $\mathsf{P_B}$ having a chance to perform a selective failure attack similar to when $\mathsf{P_A}$ is malicious. If $\mathsf{P_B}$ flip y_2, it is guessing that $x_1 \oplus x_2 = 0$; if $\mathsf{P_B}$ flip y_2 and d, $\mathsf{P_B}$ is guessing $x_1 \oplus x_2 = 1$. The proof for step 4 and 5 are the same as the proof for malicious $\mathsf{P_A}$ (with order of steps switched). □

3.3.4 Combining Leaky AND Triples

The above check is vulnerable to a selective-failure attack, from which a malicious party can learn the value of x_1 or x_2 with one-half probability of not being caught. In order to get rid of the leakage, bucketing is performed analogously to (but different from) what is done by Nielsen et al. [45]. Given two potentially leaky AND triples

$$([x_1']_\mathsf{A}, [y_1']_\mathsf{A}, [z_1']_\mathsf{A}, [x_2']_\mathsf{B}, [y_2']_\mathsf{B}, [z_2']_\mathsf{B})$$

and

$$([x_1'']_\mathsf{A}, [y_1'']_\mathsf{A}, [z_1'']_\mathsf{A}, [x_2'']_\mathsf{B}, [y_2'']_\mathsf{B}, [z_2'']_\mathsf{B}),$$

we set $[x_1]_\mathsf{A} := [x_1']_\mathsf{A} \oplus [x_1'']_\mathsf{A}$, $[x_2]_\mathsf{B} := [x_2']_\mathsf{B} \oplus [x_2'']_\mathsf{B}$. Note that the result is non-leaky as long as one of the original triples is non-leaky. We can also set $[y_1]_\mathsf{A} := [y_1']_\mathsf{A}, [y_2]_\mathsf{B} := [y_2']_\mathsf{B}$ and reveal the bit $d := y_1' \oplus y_2' \oplus y_1'' \oplus y_2''$, since y's bits are all private. Observe that

$$\begin{aligned}
(x_1 \oplus x_2)(y_1 \oplus y_2) &= (x_1' \oplus x_2' \oplus x_1'' \oplus x_2'')(y_1' \oplus y_2') \\
&= (x_1' \oplus x_2')(y_1' \oplus y_2') \oplus (x_1'' \oplus x_2'')(y_1' \oplus y_2') \\
&= (x_1' \oplus x_2')(y_1' \oplus y_2') \oplus (x_1'' \oplus x_2'')(y_1'' \oplus y_2'') \\
&\quad \oplus (x_1'' \oplus x_2'')(y_1' \oplus y_2' \oplus y_1'' \oplus y_2'') \\
&= (z_1' \oplus z_2') \oplus (z_1'' \oplus z_2'') \oplus d(x_1'' \oplus x_2'')
\end{aligned}$$

$$= (z'_1 \oplus z''_1 \oplus dx''_1) \oplus (z'_2 \oplus z''_2 \oplus dx''_2).$$

Therefore, we can just set $[z_1]_{\mathsf{A}} := [z'_1]_{\mathsf{A}} \oplus [z''_1]_{\mathsf{A}} \oplus d[x''_1]_{\mathsf{A}}$ and $[z_2]_{\mathsf{B}} := [z'_2]_{\mathsf{B}} \oplus [z''_2]_{\mathsf{B}} \oplus d[x''_2]_{\mathsf{B}}$. (This corresponds to the protocol in Figure 3.9.)

3.4 Other Ways to Instantiate $\mathcal{F}_{\mathsf{Pre}}$

We briefly note other ways $\mathcal{F}_{\mathsf{Pre}}$ can be instantiated.

IPS-based instantiation. We can obtain better asymptotic performance by instantiating $\mathcal{F}_{\mathsf{Pre}}$ using the protocol of Ishai, Prabhakaran, and Sahai [47]. In the function-dependent preprocessing phase, we need to generate an authenticated sharing of λ_w for each wire w, and an authenticated sharing of $\lambda_\sigma = (\lambda_\alpha \wedge \lambda_\beta) \oplus \lambda_\gamma$ for each AND gate $(\alpha, \beta, \gamma, \wedge)$. These can be computed by a constant-depth circuit of size $O(|C|\kappa)$. For evaluating a circuit of depth d and size ℓ, the IPS protocol uses $O(d)$ rounds and a communication complexity of $O(\ell) + \mathit{poly}(\kappa, d, \log \ell)$ bits. In our setting, this translates to a communication complexity of $O(|C|\kappa) + \mathit{poly}(\kappa, \log |C|)$ bits or, for sufficiently large circuits, $O(|C|\kappa)$ bits.

Using a (semi-)trusted server. It is straightforward to instantiate $\mathcal{F}_{\mathsf{Pre}}$ using a (semi-)trusted server. By applying the techniques of Mohassel et al. [68], the offline phase can also be done without having to know the identity of the party with whom the online phase will be executed; we refer to their paper for further details.

3.5 Extensions and Optimizations

Handling $\kappa \neq \rho$. In Figure 3.2 step 4d, all MACs that $\mathsf{P_A}$ sends are κ bits long. For ρ-bit statistical security, the value $\mathsf{M}[r_{00}]$ used in step 4(d) only needs to have length ρ. Similarly, the MACs in step 5, step 6 and step 8 can be shortened to ρ bits.

Reducing the size of the garbled tables. Observe that the bits $r_{\gamma,i}$ need not be included in the garbled table, since $\mathsf{M}[r_{\gamma,i}]$ is sufficient for $\mathsf{P_B}$ to determine (and verify) that value. Furthermore, the value $\mathsf{L}_{\gamma,0}$ is uniform and so we can further reduce the size of garbled tables using ideas similar to garbled row reduction [19]. That is, instead of choosing a uniform $\mathsf{L}_{\gamma,0}$, we instead let $\mathsf{L}_{\gamma,0}$ be equal to the κ least-significant bits of $H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 0)$. This reduces the size of a garbled table to $3\kappa + 4\rho$ bits.

Pushing computation to earlier phases. For clarity, in our description of the protocol we send the values $\{r_w, \mathsf{M}[r_w]\}_{w \in \mathcal{I}_1}$ and $\{s_w, \mathsf{M}[s_w]\}_{w \in \mathcal{I}_2}$ in steps 5 and 6. However, these values can be sent in step 4 before the inputs are known, which reduces the online communication to $|\mathcal{I}|\kappa + |\mathcal{O}|\rho$.

Further optimization of our TinyOT protocol. We aimed for simplicity in Figure 3.7, but we note here several optimizations:

1. For clarity, in Figure 3.7 step4c, the value R was chosen uniformly. To reduce the communication, $W_{x_1,0}$ can be set to 0, which defines $R := H(\mathsf{K}[x_2]) \oplus V_0$. This saves two ciphertexts per leaky AND triple.

Table 3.3: Fewest AND gates needed for bucketing, for different bucket sizes and statistical security parameters.

Bucket size	3	4	5
$\rho = 40$	280K	3.1K	320
$\rho = 64$	1.2B	780K	21K
$\rho = 80$	300B	32M	330K

Table 3.4: **Circuits used in our evaluation.**

Circuit	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{O}$	$\lvert\mathcal{C}\rvert$
AES	128	128	128	6800
SHA-128	256	256	160	37300
SHA-256	256	256	256	90825
Hamming Dist.	1048K	1048K	22	2097K
Integer Mult.	2048	2048	2048	4192K
Sorting	131072	131072	131072	10223K

2. Since efficiency depends on the bucket size $B = \rho / \log |\mathcal{C}|$, we calculated the smallest circuit size needed for each bucket size based on the exact formula, so that the bucket size can be minimized. Table 3.3 shows the fewest AND gates needed in order to use different bucket sizes (B), for different values of ρ.

3.6 Evaluation

3.6.1 Implementation and Evaluation Setup

Our implementation uses the EMP-toolkit [10], and is publicly available as a part of it.

In our evaluation, we set the computational security parameter to $\kappa = 128$

Table 3.5: Comparison in the single-execution setting

	LAN				WAN			
	Ind.	Dep.	Online	Total	Ind.	Dep.	Online	Total
AES [28]	-	28 ms	14 ms	42 ms	-	425 ms	416 ms	841 ms
AES [37]	89.6 ms	13.2 ms	1.46 ms	104.3 ms	1882 ms	96.7 ms	83.2 ms	2061.9 ms
AES (here)	10.9 ms	4.78 ms	0.93 ms	16.6 ms	821 ms	461 ms	77.2 ms	1359.2 ms
SHA1 [28]	-	139 ms	41 ms	180 ms	-	1414 ms	472 ms	1886 ms
SHA1 (here)	41.4 ms	21.3 ms	3.6 ms	66.3 ms	1288 ms	603 ms	78.4 ms	1969.4 ms
SHA256 [28]	-	350 ms	84 ms	434 ms	-	2997 ms	514 ms	3511 ms
SHA256 [37]	478.5 ms	164.4 ms	11.2 ms	654.1 ms	2738 ms	350 ms	93.9 ms	3182 ms
SHA256 (here)	96 ms	51.7 ms	9.3 ms	157 ms	1516 ms	772 ms	88 ms	2376 ms

Table 3.6: Comparison in the amortized setting. All experiments evaluate AES, with τ the number of executions being amortized over.

		LAN				WAN			
	τ	Ind.	Dep.	Online	Total	Ind.	Dep.	Online	Total
[32]	32	-	45 ms	1.7ms	46.7 ms	-	282 ms	190 ms	472 ms
	128	-	16 ms	1.5 ms	17.5 ms	-	71 ms	191 ms	262 ms
	1024	-	5.1 ms	1.3 ms	6.4 ms	-	34 ms	189 ms	223 ms
[37]	32	54.5 ms	0.85 ms	1.23 ms	56.6 ms	235.8 ms	5.2 ms	83.2 ms	324.2 ms
	128	21.5 ms	0.7 ms	1.2 ms	23.4 ms	95.8 ms	3.9 ms	83.7 ms	183.4 ms
	1024	14.7 ms	0.74 ms	1.13 ms	16.6 ms	42.1 ms	2.1 ms	83.2 ms	127.4 ms
Here	32	8.9 ms	0.6 ms	0.97 ms	10.47 ms	75.2 ms	8.7 ms	76 ms	160 ms
	128	5.4 ms	0.54 ms	0.99 ms	6.93 ms	36.6 ms	8.4 ms	75 ms	120 ms
	1024	4.9 ms	0.53 ms	1.23 ms	6.66 ms	30.0 ms	7.5 ms	76 ms	113.5 ms

and the statistical security parameter to $\rho = 40$. In Figure 3.2 we describe garbling as relying on a random oracle, but in fact it can be done using any encryption scheme; in our implementation we use the JustGarble approach of Bellare et al. [18]. We use Multithreading, Streaming SIMD Extensions (SSE), and Advanced Vector Extensions (AVX) to improve performance whenever possible.

Our implementation consists mainly of three parts:

1. **Authenticated bits.** Authenticated bits can be generated using OT extension [45]. In our implementation we adopt the OT-extension protocol of Keller et al. [70] along with the optimizations of Nielsen et al. [37]. The re-

sulting protocol requires $\kappa + \rho$ bits of communication per authenticated bit.

2. $\mathcal{F}_{\mathsf{Pre}}$ **functionality.** To improve the efficiency, we spawn multiple threads that each generate a set of leaky AND triples. After these are all generated, bucketing and combining are done in a single thread.

3. **Our protocol.** The function-independent phase invokes the above two subroutines to generate random AND triples with IT-MACs. In the function-dependent phase, these random AND triples are used to construct a single garbled circuit. In the single-execution setting, we use one thread to construct the garbled circuit; in the amortized setting we use multiple threads, each constructing a different garbled circuit. (This matches what was done in prior work.) The online phase is always done using a single thread.

Evaluation setup. Our evaluation focuses on two settings:

- LAN setting: Here we use two Amazon EC2 `c4.8xlarge` machines, both in the North Virginia region, with the link between them having 10 Gbps bandwidth and less than 1ms roundtrip time.

- WAN setting: Here we use two Amazon EC2 `c4.8xlarge` machines, one in North Virginia and one in Ireland. Single-thread communication bandwidth is about 224 Mbps; the maximum total bandwidth is about 3 Gbps when using multiple threads.

In Section 3.6.2, we first compare the performance of our protocol with previous protocols in similar settings, focusing on three circuits (AES, SHA-1, and

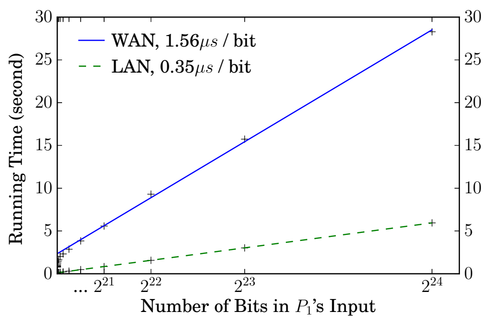
Table 3.7: **Experimental results for larger circuits.**

LAN				
	Ind. Phase	Dep. Phase	Online	Total
Hamming Dist.	1867 ms	1226 ms	74 ms	3167 ms
Integer Mult.	2860 ms	1921 ms	301 ms	5081 ms
Sorting	7096 ms	5508 ms	1021 ms	13625 ms
WAN				
	Ind. Phase	Dep. Phase	Online	Total
Hamming Dist.	11531 ms	6592 ms	133 ms	18256 ms
Integer Mult.	20218 ms	9843 ms	376 ms	30437 ms
Sorting	45155 ms	25582 ms	1918 ms	72655 ms

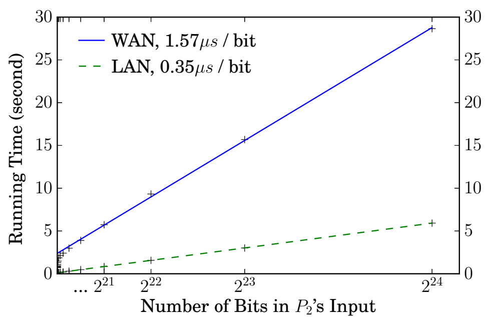
SHA-256) commonly used in prior work. Our results show that these circuits are no longer large enough to serve as benchmark circuits for malicious 2PC. Therefore, in Section 3.6.3 we also explore the performance of our protocol on some larger circuits. (These circuits are available in [10].) Parameters for all the circuits we study are given in Table 3.4. In Sections 3.6.4 and 3.6.5, we study the scalability of our protocol and compare its concrete communication complexity with prior work.

3.6.2 Comparison with Previous Work

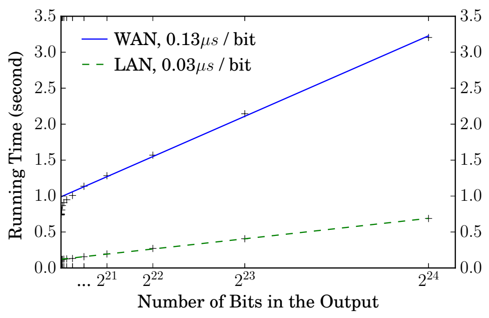
Single-execution setting. First we compare the performance of our protocol to state-of-the-art 2PC protocols in the single-execution setting. In particular, we compare with the protocol of Wang et al. [28], which is based on circuit-level cut-and-choose and is tailored for the single-execution setting, as well as the protocol of Nielsen et al. [37], which is based on gate-level cut-and-choose and is able to utilize function-independent preprocessing. For a fair comparison, all numbers are based on the same hardware configuration as we used. Our reported timings do *not* include



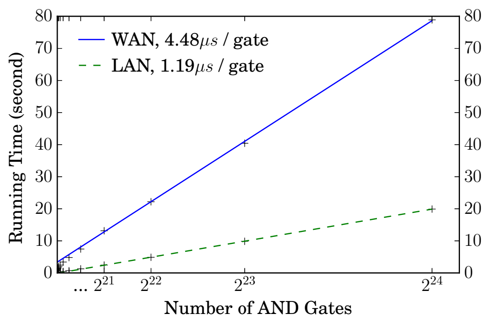
(a) Increasing $\mathsf{P_A}$'s input size ($\mathcal{I}_1$).



(b) Increasing $\mathsf{P_B}$'s input size ($\mathcal{I}_2$).



(c) Increasing output size ($\mathcal{O}$).



(d) Increasing circuit size ($|\mathcal{C}|$).

Figure 3.10: Scalability of our protocol. Initially $|\mathcal{I}_1| = |\mathcal{I}_2| = |\mathcal{O}| = 128$ and $|\mathcal{C}| = 1024$, and then one of those parameters is allowed to grow while the others remain fixed. The total running time is reported.

the time for the base-OTs for the same reason as in [37]: the time for the base-OTs is constant across all protocols and is not the focus of our work. For completeness, though, we note that our base-OT implementation (based on the protocol by Chou and Orlandi [71]) takes about 20 ms in the LAN setting and 240 ms in the WAN setting.

As shown in Table 3.5, our protocol performs better than previous protocols in terms of both overall time and online time. Compared with the protocol by Wang et al., we achieve a speedup of 2.7× overall and an improvement of about 10× for the online time. Compared with the protocol by Nielsen et al., the online time is roughly the same but our offline time is 4–7$\times$ better in the LAN setting, and 1.3-1.5$\times$ better in the WAN setting.

Compared to the recent (unimplemented) work of Lindell et al. [42], our protocol is asymptotically more efficient in the function-independent preprocessing phase. More importantly, the concrete efficiency of our protocol is much better for several reasons: (1) our work is compatible with free-XOR and we do not suffer from any blowup in the size of the circuit being evaluated; (2) Lindell et al. require five SPDZ-style multiplications per AND gate of the underlying circuit, while we need only one TinyOT-style AND computation per AND gate.

We perform a back-of-the-envelope calculation to compare the relative efficiency of our protocol and that of Lindell et al. [42]. Over a 10 Gbps network, the recent work of Keller et al. [49] can generate 55,000 SPDZ multiplication triples per second using an ideal implementation that fully saturates the network. The protocol of Lindell et al. requires 5 SPDZ multiplications per AND gate, and so the best possible end-to-end speed of their protocol is 11,000 AND gates per second. On the other hand, our actual implementation computes 833,333 AND gates per second (as shown by the scalability evaluation in Section 3.6.4). Therefore, our protocol is at least 75$\times$ better than the best possible implementation of their protocol.

Comparison with linear-round protocols. The AES circuit has depth 50 [31]. Therefore, even in the LAN setting with 0.5 ms roundtrip time, and ignoring all computation and communication, any linear-round protocol for securely computing

AES would require at least 25 ms in total, which is 1.5× slower than our protocol.

The protocol by Damgård et al. [72] has the best end-to-end running time among all linear-round protocols. Their protocol only supports amortization for *parallel* executions (where inputs to all executions are known at the outset). They report an amortized time for evaluating AES of 14.65 ms per execution, amortized over 680 executions. This is roughly on par with our *single-execution* performance without any preprocessing. When comparing their results to our amortized performance, we are more than 2× faster, and we are not limited to parallel execution.

A more recent work by Damgård et al. [73] proposes a protocol with a very efficient online phase. In the LAN setting with similar hardware, it has an online time of 1.09 ms to evaluate AES, which is similar to our reported time (0.93 ms). They also report $0.47\mu s$ online time in the parallel execution setting, which is different from our amortized setting as discussed above. We cannot compare end-to-end running times since they do not report the preprocessing time. However, we note that they use TinyOT for preprocessing, and our optimized TinyOT protocol is more efficient. (On the other hand, our new TinyOT protocol could be plugged into their work to improve the running time of the preprocessing phase in their work as well.)

Amortized setting. It is somewhat difficult to compare protocols in the amortized setting, since relative performance depends on the setting (LAN or WAN), the number of executions being amortized over, and whether one chooses to focus on the total time or the online time. Nevertheless, as shown in Table 3.6, our protocol offers a consistent improvement as compared to the best prior work of Nielsen et al. [37] and Rindal and Rosulek [32].

3.6.3 Larger Circuits

The results of the previous section show that evaluating the AES circuit using our protocol takes less time than generating the base-OTs. Thus, our work implies that AES and other existing benchmark circuits are no longer large enough for a meaningful performance evaluation of malicious 2PC protocols. We propose three new example computations and evaluate our protocol on these examples:

- **Hamming distance:** Here we consider computing the Hamming distance between two n-bit strings using an $O(n)$-size circuit. For our concrete experiments, we set $n = 1048576$; the output is a 22-bit integer.

- **Integer multiplication:** Here we consider computing the least-significant n bits of the product of two n-bit integers using a n$O(n^2)$-size circuit. For our concrete experiments, we use $n = 2048$.

- **Sorting:** Here we consider sorting n integers, each ℓ bits long, that are XOR-shared between two parties, using a circuit of size $O(n\ell \log^2 n)$. For our concrete experiments, we use $n = 4096$ and $\ell = 32$.

The parameters of the concrete circuits we use in our experiments are given in Table 3.4.

In Table 3.7 we show the performance of our protocol on the above examples. We observe that the difference in the online time between the LAN and WAN settings

Table 3.8: Communication per execution for evaluating an AES circuit. Numbers presented are for the amount of data sent from garbler to evaluator; this reflects the speed in a duplex network. For a simplex network, the communication reported here and by Rindal and Rosulek [32] should be doubled for a fair comparison.

Protocol	τ	Ind. Phase	Dep. Phase	Online
[32]	302	-	3.8 MB	25.8 KB
	128	-	2.5 MB	21.3 KB
	1024	-	1.6 MB	17.0 KB
[37]	1	14.9 MB	0.22 MB	16.1 KB
	32	8.7 MB	0.22 MB	16.1 KB
	128	7.2 MB	0.22 MB	16.1 KB
	1024	6.4 MB	0.22 MB	16.1 KB
This Paper	1	2.86 MB	0.57 MB	4.86 KB
	32	2.64 MB	0.57 MB	4.86 KB
	128	2.0 MB	0.57 MB	4.86 KB
	1024	2.0 MB	0.57 MB	4.86 KB

is about 75 ms, which is roughly the roundtrip time of the WAN network we used. This is also consistent with the fact that our protocol requires only one round of online communication (one message from each party). To compare our results with state-of-the-art *semi-honest* protocols, note that garbling can be done at the rate of about 20 million AND gates per second. So, for example, sorting could be done with an online time of about 0.5 seconds in the semi-honest setting.

3.6.4 Scalability

To explore the concrete performance of our protocol for circuits with different input, output, and circuit sizes, we study the effect on the total running time as each of these parameters is varied. The results are reported in Figure 3.10. Trend lines are also included to show the marginal effect (i.e., the slope) of each parameter. Although the optimal bucket size in our protocol becomes smaller as the circuit size increases, we fix the bucket size to 3 in Figure 3.10(d).

Our results show that the performance of our protocol scales linearly in the input, output, and circuit sizes, as expected. In the LAN setting, our protocol requires only 0.35 μs to process each input bit and 0.03 μs per output bit. Note that this is much better than circuit-level cut-and-choose protocols, mainly for two reasons: (1) Since we construct only one garbled circuit, only one set of garbled labels needs to be transferred; this is an improvement of $\rho\times$. (2) We do not need to use an XOR-Tree or a ρ-probe matrix (which can incur a huge cost when the input is large [28]) to prevent selective-failure attacks.

Our results also show that the marginal performance (for all the parameters considered) is about 3–4× slower in the WAN setting than in the LAN setting, which roughly matches the ratio of network bandwidth between the two settings.

3.6.5 Communication Complexity

In Table 3.8, we compare the communication complexity (measured in terms of the amount of data sent from the garbler to the evaluator) of our protocol to that of other work, focusing on the amortized evaluation of AES. The communication complexity of our protocol is $3 - -5\times$ less than in the protocol of Nielsen et al.. Furthermore, the communication complexity of our protocol in the *single-execution* setting is only half the communication complexity of their protocol even when amortized over 1024 executions. Note that for protocols based on cut-and-choose, the total communication required to send 40 garbled AES circuits is 8.7 MB, which is already higher than the total communication of our protocol in the single-execution setting.

We also observe that the communication complexity of our protocol in the function-dependent preprocessing phase is higher than that of the protocol of Nielsen et al.; this is due to the fact that we need to send $3\kappa + 4\rho$ bits per gate while they only need to send 2κ bits per gate. On the other hand, our online communication is extremely small: it is about $3\times$ smaller than in the protocol of Nielsen et al. and 3.5–5.3$\times$ smaller than in the protocol of Rindal and Rosulek.

Chapter 4: Applying Authenticated Garbling Technique for Multi-Party Computation

In this chapter, we will discuss how to extend the idea of authenticated garbling to the multi-party setting. By doing so, our MPC protocol is constant-round and able to tolerate any number of malicious corruptions. Previous implementations of MPC protocols in this model rely on some variant of the secret-sharing paradigm introduced by Goldreich, Micali, and Wigderson [46]. At a high level, this technique requires the parties to maintain the invariant of holding a linear secret sharing of the values on the circuit wires, along with some sort of authentication information on those shares. Linear gates in the circuit (e.g., XOR, ADD) can be processed locally, while non-linear operations (e.g., AND, MULT) are handled by having the parties interact with each other to maintain the desired invariant. The most notable example of a protocol in this framework is perhaps SPDZ [44, 48, 49], which supports arithmetic circuits; protocols for boolean circuits have also been designed [50, 51] with various efficiency.

Although this approach can lead to protocols with reasonable efficiency when run over a LAN, it suffers the inherent drawback of leading to round complexity *linear* in the depth of the circuit being evaluated. This can have a significant

Table 4.1: Asymptotic complexity (per party) for n-party MPC protocols for boolean circuits, secure against an arbitrary number of malicious corruptions. Here, κ (resp., ρ) is the computational (resp., statistical) security parameter, $|C|$ is the circuit size, d is the depth of the circuit, and $B = O(\rho / \log |C|)$.

Paper	Comm./Comp. Complexity	Rounds
[74]	$O\left(\lvert\mathcal{C}\rvert B^3 n\right)$	$O(d)$
[51]	$O\left(\lvert\mathcal{C}\rvert B^2 n\right)$	$O(d)$
[42] + [49]	$O\left(\lvert\mathcal{C}\rvert \kappa n^2\right)$	$O(1)$
[75]	$O\left(\lvert\mathcal{C}\rvert B^2 n\right)$	$O(1)$
This paper	$O\left(\lvert\mathcal{C}\rvert B n\right)$	$O(1)$

impact on the overall efficiency when the parties running the protocol are geographically separated or when the number of parties is high, and the communication latency dominates the cost of the execution. For example, the communication latency between parties located in the U.S. and Europe is around 75 ms even with the dedicated network provided by Amazon EC2. If such parties are evaluating, say, SHA-256 (which has a circuit depth of about 4,000), then a linear-round protocol requires $300,000$ ms just for the back-and-forth interaction between those parties, not even counting the time required for performing local cryptographic operations or transmitting any data.

Constant-round protocols tolerating any number of malicious corruptions have also been designed. The basic approach here, first proposed by Beaver, Micali, and Rogaway [64], is to have the parties run a linear-round secure-computation protocol to compute a garbled circuit [11] for the function f of interest; the parties can then evaluate that garbled circuit using a constant number of additional rounds. Since

Table 4.2: Selected performance results for our protocol. All results are in milliseconds, based on a statistical security of 2^{-40}. We consider the following settings (see Section 4.6 for more details.):

(a) **3PC-LAN**: three-party computation over a LAN;

(b) **128PC-LAN**: 128-party computation over a LAN;

(c) **14PC-Worldwide**: 14-party computation over a WAN, with parties located in 14 different cities across five continents;

(d) **128PC-Worldwide**: 128-party computation over a WAN, with parties located in 8 different cities across five continents (each city with 16 parties).

Setting	Setup	Indep.	Depen.	Online	Total
3PC-LAN	36	47	12	2	95
128PC-LAN	390	2727	11670	1870	16657
14PC-Worldwide	8711	9412	1947	250	20320
128PC-Worldwide	88056	30796	22659	2316	143827

the circuit for computing the garbling of f has depth independent of f, the overall number of rounds is constant. Although several recent papers have explored this approach [41, 42, 43], these investigations have remained largely theoretical since the overall cost of even the best protocol using this approach is asymptotically worse than the nonconstant-round protocols mentioned above; see Table 4.1. In fact, prior implementations of constant-round MPC consider only the *semi-honest* setting [52, 56].

Our Contributions. We take a significant step towards practical MPC tolerating an unbounded number of malicious corruptions. To this end, we propose a new, constant-round protocol for multi-party computation of boolean circuits secure in this setting. Our protocol extends and generalizes the technique introduced in the previous chapter in the two-party setting. Specifically, we first design an optimized, multi-party version of their TinyOT protocol so as to enable n parties to generate certain authenticated information as part of a preprocessing phase. Next, generalizing their main protocol, we show how to use this information to distributively construct a *single* "authenticated" garbled circuit that is evaluated by a single party. An overview of our entire protocol appears in Section 4.1, with details in the remainder of the paper.

Our protocol improves upon the state-of-the-art both asymptotically (cf. Table 4.1) and concretely (see Section 4.6.4), and in particular it allows us to give the *first* implementation of a constant-round MPC protocol with malicious security. Our experiments demonstrate that our protocol is both *efficient* and *scalable*:

- **Efficiency:** For three-party computation over a LAN, our protocol requires only 95 ms to securely evaluate AES. This is roughly a 700$\times$ improvement over the best prior work, and only 2.5$\times$ slower than the state-of-the-art solution in the two-party setting, introduced in the previous chapter. In general, for n-party computation our protocol improves upon the best prior work [42, 43] (which was not implemented) by a factor of more than $200n$.

- **Scalability:** We successfully executed our protocol with many parties located all over the world, computing (for example) AES with 128 parties across 5 continents in under 3 minutes. To the best of our knowledge, our work represents the largest-scale demonstration of secure computation to date, even considering weaker adversarial models.

4.1 Overview of Our Main Protocol

Our main protocol is designed in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model (see Figure 2.2). At a high level, $\mathcal{F}_{\mathsf{Pre}}$ generates authenticated shares on random bits x, y, z such that $z = x \wedge y$. (We refer to these as *AND triples.*) Our main protocol then uses those authenticated shares to distributively construct a single, "authenticated" garbled circuit that is evaluated by one of the parties. In the remainder of this section, we describe our main protocol; further details are in Section 4.2. In Section 4.4 we then discuss how to efficiently realize $\mathcal{F}_{\mathsf{Pre}}$.

It is highly non-trivial to extend the two-party protocol introduced in the previous chapter to the multiparty setting. The main challenge is that even when $n-1$ parties are corrupted, we still need to make sure that the adversary cannot learn any information about the honest party's inputs.

Attempted ideas. One idea, adopted by Choi et al. [41] in the three-party setting, is to let $n-1$ parties jointly compute a garbled circuit that the remaining party will evaluate. However, if the $n-1$ garblers are corrupt, there is no guarantee about the correctness of the garbled circuit they generate. For that reason, Choi et al. had to use cut-and-choose to check correctness of a random subset of ρ garbled circuits, which imposes a huge overhead.

To avoid this additional cut-and-choose, we would like all parties to be involved in the garbled-circuit generation, as in the BMR protocol [64]. However, state-of-the-art protocols based on BMR that are maliciously secure against corruption of $n-1$ parties require either $O(n)$ somewhat homomorphic encryptions [43] or $O(n)$ SPDZ multiplication subprotocols [42] per AND gate both of which are relatively inefficient. We aim instead to use "simpler" TinyOT-like functionalities as we explain next.

Multiparty TinyOT: BDOZ-style vs. SPDZ-style. We observe that in the existing literature, there are mainly two flavors on how authenticated shared are constructed.

- **BDOZ-style** [76]: For a secret bit x, each party holds a share of x. For each ordered pair of parties (P_i, P_j), P_i authenticates its own share (namely x^i) to P_j.

- **SPDZ-style** [44]: Each party holds a share of a global MAC key. For a secret bit x, each party holds a share of x and a share of the MAC on x.

Note that these protocols are constructed for arithmetic circuits, but these representations also apply to binary circuits. Existing papers prefer SPDZ-style shares to BDOZ-style shares, because SPDZ-style shares are smaller and thus more efficient to operate on. Indeed, existing papers that investigated protocols for multi-party TinyOT are all based on SPDZ-style shares [50, 74, 51].

Our key observation is that such SPDZ-style AND triple, although efficient for interactive MPC protocols, are not suitable for our use to construct constant-round MPC protocols. In particular, in the SPDZ-style shares, each parties knows Δ_i as a share of the global key $\Delta = \bigoplus_i \Delta_i$. For each bit x, they holds shares of $x\Delta$. Since

Δ is not known to any party, it is not directly related to any garbled circuit. On the contrary, in the BDOZ-style protocols, each party holds $(x^i, \{\mathsf{M}_j[x^i], \mathsf{K}_i[x^j]\}_{j\neq i})$, as we have already described in Section 2.2. In this case, they essentially hold shares of $x\Delta_i$ for all $i \in [n]$, because:

$$\begin{aligned} x\Delta_i &= \left(\bigoplus_j x^j\right)\Delta_i = x^i\Delta_i \oplus \left(\bigoplus_{j\neq i} x^j\Delta_i\right) \\ &= x^i\Delta_i \oplus \left(\bigoplus_{j\neq i} \mathsf{M}_i[x^j] \oplus \mathsf{K}_i[x^j]\right) \\ &= \left(x^i\Delta_i \oplus \bigoplus_{j\neq i} \mathsf{K}_i[x^j]\right) \oplus \bigoplus_{j\neq i} \mathsf{M}_i[x^j] \end{aligned}$$

Here, P_i knows the first value, while each P_j with $j \neq i$ knows $\mathsf{M}_i[x^j]$. In other word, a BDOZ-style share of a bit x can be used to construct shares of $x\Delta_i$ for each i. This can further be used to construct shares of garbled labels, *if we use the same Δ_i for authenticated shares and the global difference used in free-XOR*. Indeed, looking ahead to the main protocol in Figure 4.1 step 4 (d), the content of the garbled circuit can be viewed as some authentication information plus shares of the garbled output labels for each garbler.

High level picture of the protocol. Given the above discussion, we can now picture the high level idea of our protocol. Our idea, from a high level view, is to let $n - 1$ parties be garblers, each maintaining a set of garbled labels, and to let the remaining party be the evaluator. Each garbler knows its own set of garbled labels. However, for each gate and for each garbler, the *permuted garbled output labels* are secretly shared to all parties, and thus no party knows how these labels are permuted. For each garbler P_i and a garbler row, P_i has shares of permuted

Protocol Π_{mpc}

Inputs: In the function-independent phase, parties know $|\mathcal{C}|$ and $|\mathcal{I}|$; in the function-dependent phase, parties get a circuit representing function $f : \{0,1\}^{|\mathcal{I}_1|} \times ... \times \{0,1\}^{|\mathcal{I}_n|} \to \{0,1\}^{|\mathcal{O}|}$; in the input-processing phase, P_i holds $x_i \in \{0,1\}^{|\mathcal{I}_i|}$.

Function-independent phase:

1. P_i sends init to $\mathcal{F}_{\mathsf{Pre}}$, which sends Δ_i to P_i.
2. For each wire $w \in \mathcal{I} \cup \mathcal{W}, i \in [n]$, P_i sends random to $\mathcal{F}_{\mathsf{Pre}}$, which sends $\left(r^i_w, \left\{\mathsf{M}_j[r^i_w], \mathsf{K}_i[r^j_w]\right\}_{j\neq i}\right)$ to P_i, where $\bigoplus_{i\in[n]} r^i_w = \lambda_w$. For each $i \neq 1$, P_i also picks a random κ-bit string $\mathsf{L}^i_{w,0}$.

Function-dependent phase:

3. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \oplus)$, each $i \in [n]$, P_i computes $\left(r^i_\gamma, \left\{\mathsf{M}_j[r^i_\gamma], \mathsf{K}_i[r^j_\gamma]\right\}_{j\neq i}\right) := \left(r^i_\alpha \oplus r^i_\beta, \left\{\mathsf{M}_j[r^i_\alpha] \oplus \mathsf{M}_j[r^i_\beta], \mathsf{K}_i[r^j_\alpha] \oplus \mathsf{K}_i[r^j_\beta]\right\}_{j\neq i}\right)$. For each $i \neq 1$, P_i also computes $\mathsf{L}^i_{\gamma,0} := \mathsf{L}^i_{\alpha,0} \oplus \mathsf{L}^i_{\beta,0}$.
4. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \wedge)$:

   (a) For each $i \in [n]$, P_i sends $\left(\mathsf{and}, \left(r^i_\alpha, \left\{\mathsf{M}_j[r^i_\alpha], \mathsf{K}_i[r^j_\alpha]\right\}_{j\neq i}\right), \left(r^i_\beta, \left\{\mathsf{M}_j[r^i_\beta], \mathsf{K}_i[r^j_\beta]\right\}_{j\neq i}\right)\right)$ to $\mathcal{F}_{\mathsf{Pre}}$, which sends $\left(r^i_\sigma, \left\{\mathsf{M}_j[r^i_\sigma], \mathsf{K}_i[r^j_\sigma]\right\}_{j\neq i}\right)$ to P_i, where $\bigoplus_{i\in[n]} r^i_\sigma = \left(\bigoplus_{i\in[n]} r^i_\alpha\right) \wedge \left(\bigoplus_{i\in[n]} r^i_\beta\right)$.

   (b) For each $i \neq 1$, P_i computes the following locally.

$$\begin{aligned}
\left(r^i_{\gamma,0}, \left\{\mathsf{M}_j[r^i_{\gamma,0}], \mathsf{K}_i[r^j_{\gamma,0}]\right\}_{j\neq i}\right) &:= \left(r^i_\sigma \oplus r^i_\gamma, \quad \left\{\mathsf{M}_j[r^i_\sigma] \oplus \mathsf{M}_j[r^i_\gamma], \quad \mathsf{K}_i[r^j_\sigma] \oplus \mathsf{K}_i[r^j_\gamma]\right\}_{j\neq i}\right)\\
\left(r^i_{\gamma,1}, \left\{\mathsf{M}_j[r^i_{\gamma,1}], \mathsf{K}_i[r^j_{\gamma,1}]\right\}_{j\neq i}\right) &:= \left(r^i_{\gamma,0} \oplus r^i_\alpha, \left\{\mathsf{M}_j[r^i_{\gamma,0}] \oplus \mathsf{M}_j[r^i_\alpha], \quad \mathsf{K}_i[r^j_{\gamma,0}] \oplus \mathsf{K}_i[r^j_\alpha]\right\}_{j\neq i}\right)\\
\left(r^i_{\gamma,2}, \left\{\mathsf{M}_j[r^i_{\gamma,2}], \mathsf{K}_i[r^j_{\gamma,2}]\right\}_{j\neq i}\right) &:= \left(r^i_{\gamma,0} \oplus r^i_\beta, \left\{\mathsf{M}_j[r^i_{\gamma,0}] \oplus \mathsf{M}_j[r^i_\beta], \quad \mathsf{K}_i[r^j_{\gamma,0}] \oplus \mathsf{K}_i[r^j_\beta]\right\}_{j\neq i}\right)\\
\left(r^i_{\gamma,3}, \left\{\mathsf{M}_j[r^i_{\gamma,3}], \mathsf{K}_i[r^j_{\gamma,3}]\right\}_{j\neq i}\right) &:= \Big(r^i_{\gamma,1} \oplus r^i_\beta, \left\{\mathsf{M}_1[r^i_{\gamma,1}] \oplus \mathsf{M}_1[r^i_\beta], \quad \mathsf{K}_i[r^1_{\gamma,1}] \oplus \mathsf{K}_i[r^1_\beta] \oplus \Delta_i\right\}\\
&\qquad \bigcup \left\{\mathsf{M}_j[r^i_{\gamma,1}] \oplus \mathsf{M}_j[r^i_\beta], \ \mathsf{K}_i[r^j_{\gamma,1}] \oplus \mathsf{K}_i[r^j_\beta]\right\}_{j\neq i,1}\Big)
\end{aligned}$$

   (c) P_A computes the following locally.

$$\begin{aligned}
\left(r^1_{\gamma,0}, \left\{\mathsf{M}_j[r^1_{\gamma,0}], \mathsf{K}_1[r^j_{\gamma,0}]\right\}_{j\neq i}\right) &:= \left(r^1_\sigma \oplus r^1_\gamma, \qquad \left\{\mathsf{M}_j[r^1_\sigma] \oplus \mathsf{M}_j[r^1_\gamma], \quad \mathsf{K}_1[r^j_\sigma] \oplus \mathsf{K}_1[r^j_\gamma]\right\}_{j\neq i}\right)\\
\left(r^1_{\gamma,1}, \left\{\mathsf{M}_j[r^1_{\gamma,1}], \mathsf{K}_1[r^j_{\gamma,1}]\right\}_{j\neq i}\right) &:= \left(r^1_{\gamma,0} \oplus r^1_\alpha, \qquad \left\{\mathsf{M}_j[r^1_{\gamma,0}] \oplus \mathsf{M}_j[r^1_\alpha], \ \mathsf{K}_1[r^j_{\gamma,0}] \oplus \mathsf{K}_1[r^j_\alpha]\right\}_{j\neq i}\right)\\
\left(r^1_{\gamma,2}, \left\{\mathsf{M}_j[r^1_{\gamma,2}], \mathsf{K}_1[r^j_{\gamma,2}]\right\}_{j\neq i}\right) &:= \left(r^1_{\gamma,0} \oplus r^1_\beta, \qquad \left\{\mathsf{M}_j[r^1_{\gamma,0}] \oplus \mathsf{M}_j[r^1_\beta], \ \mathsf{K}_1[r^j_{\gamma,0}] \oplus \mathsf{K}_1[r^j_\beta]\right\}_{j\neq i}\right)\\
\left(r^1_{\gamma,3}, \left\{\mathsf{M}_j[r^1_{\gamma,3}], \mathsf{K}_1[r^j_{\gamma,3}]\right\}_{j\neq i}\right) &:= \left(r^1_{\gamma,1} \oplus r^1_\beta \oplus 1, \left\{\mathsf{M}_j[r^1_{\gamma,1}] \oplus \mathsf{M}_j[r^1_\beta], \ \mathsf{K}_1[r^j_{\gamma,1}] \oplus \mathsf{K}_1[r^j_\beta]\right\}_{j\neq i}\right)
\end{aligned}$$

   (d) For each $i \neq 1$, P_i computes $\mathsf{L}^i_{\alpha,1} := \mathsf{L}^i_{\alpha,0} \oplus \Delta_i$ and $\mathsf{L}^i_{\beta,1} := \mathsf{L}^i_{\beta,0} \oplus \Delta_i$, and sends the following to P_A.

$$\begin{aligned}
G^i_{\gamma,0} &:= H\left(\mathsf{L}^i_{\alpha,0}, \mathsf{L}^i_{\beta,0}, \gamma, 0\right) \oplus \left(r^i_{\gamma,0}, \left\{\mathsf{M}_j[r^i_{\gamma,0}]\right\}_{j\neq i}, \mathsf{L}^i_{\gamma,0} \oplus \left(\textstyle\bigoplus_{j\neq i} \mathsf{K}_i[r^j_{\gamma,0}]\right) \oplus r^i_{\gamma,0}\Delta_i\right)\\
G^i_{\gamma,1} &:= H\left(\mathsf{L}^i_{\alpha,0}, \mathsf{L}^i_{\beta,1}, \gamma, 1\right) \oplus \left(r^i_{\gamma,1}, \left\{\mathsf{M}_j[r^i_{\gamma,1}]\right\}_{j\neq i}, \mathsf{L}^i_{\gamma,0} \oplus \left(\textstyle\bigoplus_{j\neq i} \mathsf{K}_i[r^j_{\gamma,1}]\right) \oplus r^i_{\gamma,1}\Delta_i\right)\\
G^i_{\gamma,2} &:= H\left(\mathsf{L}^i_{\alpha,1}, \mathsf{L}^i_{\beta,0}, \gamma, 2\right) \oplus \left(r^i_{\gamma,2}, \left\{\mathsf{M}_j[r^i_{\gamma,2}]\right\}_{j\neq i}, \mathsf{L}^i_{\gamma,0} \oplus \left(\textstyle\bigoplus_{j\neq i} \mathsf{K}_i[r^j_{\gamma,2}]\right) \oplus r^i_{\gamma,2}\Delta_i\right)\\
G^i_{\gamma,3} &:= H\left(\mathsf{L}^i_{\alpha,1}, \mathsf{L}^i_{\beta,1}, \gamma, 3\right) \oplus \left(r^i_{\gamma,3}, \left\{\mathsf{M}_j[r^i_{\gamma,3}]\right\}_{j\neq i}, \mathsf{L}^i_{\gamma,0} \oplus \left(\textstyle\bigoplus_{j\neq i} \mathsf{K}_i[r^j_{\gamma,3}]\right) \oplus r^i_{\gamma,3}\Delta_i\right)
\end{aligned}$$

Figure 4.1: Our main protocol. Here ρ is set to κ for clarity, but this is not necessary.

Protocol Π_{mpc}, continued

Input Processing:

5. For each $i \neq 1, w \in \mathcal{I}_i$, for each $j \neq i$, P_j sends $(r^j_w, \mathsf{M}_i[r^j_w])$ to P_i, who checks that $(r^j_w, \mathsf{M}_i[r^j_w], \mathsf{K}_i[r^j_w])$ is valid, and computes $x^i_w \oplus \lambda_w := x^i_w \oplus \left(\bigoplus_{i\in[n]} r^i_w\right)$. P_i broadcasts the value $x^i_w \oplus \lambda_w$. For each $j \neq 1$, P_j sends $\mathsf{L}^j_{x^i \oplus \lambda_w}$ to $\mathsf{P_A}$.
6. For each $w \in \mathcal{I}_1, i \neq 1$, P_i sends $(r^i_w, \mathsf{M}_1[r^i_w])$ to $\mathsf{P_A}$, who checks that $(r^i_w, \mathsf{M}_1[r^i_w], \mathsf{K}_1[r^i_w])$ are valid, and computes $x^1_w \oplus \lambda_w := x^1_w \oplus \left(\bigoplus_{i\in[n]} r^i_w\right)$. $\mathsf{P_A}$ sends $x^1_w \oplus \lambda_w$ to P_i, who sends $\mathsf{L}^i_{w, x^1_w \oplus \lambda_w}$ to $\mathsf{P_A}$.

Circuit Evaluation:

7. $\mathsf{P_A}$ evaluates the circuit following the topological order. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, T)$, $\mathsf{P_A}$ holds $\left(z_\alpha \oplus \lambda_\alpha, \left\{\mathsf{L}^i_{\alpha, z_\alpha \oplus \lambda_\alpha}\right\}_{i\neq 1}\right)$ and $\left(z_\beta \oplus \lambda_\beta, \left\{\mathsf{L}^i_{\beta, z_\beta \oplus \lambda_\beta}\right\}_{i\neq 1}\right)$, where z_α, z_β are the underlying values of the wire.
   (a) If $T = \oplus$, $\mathsf{P_A}$ computes $z_\gamma \oplus \lambda_\gamma := (z_\alpha \oplus \lambda_\alpha) \oplus (z_\beta \oplus \lambda_\beta)$ and $\left\{\mathsf{L}^i_{\gamma, z_\gamma \oplus \lambda_\gamma} := \mathsf{L}^i_{\alpha, z_\alpha \oplus \lambda_\alpha} \oplus \mathsf{L}^i_{\beta, z_\beta \oplus \lambda_\beta}\right\}_{i\neq 1}$
   (b) If $T = \wedge$, $\mathsf{P_A}$ computes $\ell := 2(z_\alpha \oplus \lambda_\alpha) + (z_\beta \oplus \lambda_\beta)$. For $i \neq 1$, $\mathsf{P_A}$ computes
   $$\left(r^i_{\gamma,\ell}, \{\mathsf{M}_j[r^i_{\gamma,\ell}]\}_{j\neq i}, \mathsf{L}^i_\gamma\right) := G^i_{\gamma,\ell} \oplus H\left(\mathsf{L}^i_{\alpha, z_\alpha \oplus \lambda_\alpha}, \mathsf{L}^i_{\beta, z_\beta \oplus \lambda_\beta}, \gamma, \ell\right).$$
   $\mathsf{P_A}$ checks that $\left\{\left(r^i_{\gamma,\ell}, \mathsf{M}_1[r^i_{\gamma,\ell}], \mathsf{K}_1[r^i_{\gamma,\ell}]\right)\right\}_{i\neq 1}$ are valid and aborts if fails. $\mathsf{P_A}$ computes $z_\gamma \oplus \lambda_\gamma := \bigoplus_{i\in[n]} r^i_{\gamma,\ell}$, and $\left\{\mathsf{L}^i_{\gamma, z_\gamma \oplus \lambda_\gamma} := \mathsf{L}^i_\gamma \oplus \left(\bigoplus_{j\neq i} \mathsf{M}_i[r^j_{\gamma,\ell}]\right)\right\}_{i\neq 1}$

Output Processing:

8. For each $w \in \mathcal{O}, i \neq 1$, P_i sends $(r^i_w, \mathsf{M}_1[r^i_w])$ to $\mathsf{P_A}$, who checks that $(r^i_w, \mathsf{M}_1[r^i_w], \mathsf{K}_1[r^i_w])$ is valid. $\mathsf{P_A}$ computes $z_w := (\lambda_w \oplus z_w) \oplus \left(\bigoplus_{i\in[n]} r^i_w\right)$.

Figure 4.2: Our main protocol, continued. Here ρ is set to κ for clarity, but this is not necessary.

garbled output labels for all garblers. In the garbled table, P_i encrypts all these shares using its own set of garbled input labels. Further, shares of the mask value are authenticated similarly. The evaluator decrypts the same row of garbled tables from all garblers in order to recompute the garbled output labels for each garbler. Intuitively, this ensures that for any set of $n - 1$ parties, they cannot garble or evaluate any gate.

4.2 The Main Scheme

Since we have discussed the main intuition of our protocol, we will proceed to the details directly. The proof of the main protocol can be found in Section 4.3.1. In Figure 4.1 and Figure 4.2, we present the complete MPC protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. The protocol can be divided into five phases:

1. **Circuit Pre-scan.** (Step 1-3) In this phase, each party obtains their own private global MAC keys (Δ_i) from $\mathcal{F}_{\mathsf{Pre}}$, and generate authenticated shares on wire masks for all wires.

2. **Circuit Garbling.** In this phase, each party compute shares of garbled tables for each garbler (Step 4 (a) - 4 (c)). Garblers then compute the distributed garbled circuits based on these shares (Step 4 (d)).

3. **Circuit Input Processing.** For each input wire that corresponds to P_i's input, all other parties reveal their share of the wire mask to P_i. Party[i] then broadcasts the masked input values. All garblers, upon receiving this masked input value, send the corresponding key to the evaluator.

4. **Circuit Evaluation.** The evaluator evaluate the circuit following the topological order. In detail, the garbled wire labels from each garbler is used to obtain a set of shares of the wire labels for the output of the gate. The wire labels can then be constructed from the shares.

5. **Circuit Output Processing.** Now the evaluator holds masked output. All garblers reveal their shares of the output wire masks for the circuit to let evaluator unmask the value.

4.3 Security Proofs

4.3.1 Proof of the Main Protocol

Theorem 4.3.1. *If H is modeled as a random oracle, the protocol in Figures 4.1 and 4.2 securely realizes $\mathcal{F}_{\mathsf{mpc}}$ in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model with security $\mathsf{negl}(\kappa)$ against an adversary corrupting up to $n-1$ parties.*

Proof. We consider separately the case where $\mathsf{P_A} \in \mathcal{H}$ and where $\mathsf{P_A} \in \mathcal{M}$ and $\mathsf{P_B} \in \mathcal{H}$. The case where $\mathsf{P_A} \in \mathcal{M}$ and $P_i \in \mathcal{H}$ for some $i \geq 3$ is similar to the second case.

Honest $\mathsf{P_A}$. Let $\mathcal{A}$ be an adversary corrupting $\{P_i\}_{i\in\mathcal{M}}$. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of $\{P_i\}_{i\in\mathcal{M}}$ in the ideal world involving an ideal functionality $\mathcal{F}_{\mathsf{mpc}}$ evaluating f. $\mathcal{S}$ is defined as follows.

1-4 $\mathcal{S}$ acts as honest $\{P_i\}_{i\in\mathcal{H}}$ and plays the functionality of $\mathcal{F}_{\mathsf{Pre}}$, recording all outputs. If any honest party or $\mathcal{F}_{\mathsf{Pre}}$ would abort, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and then aborts.

5 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\{P_i\}_{i\in\mathcal{H}}$, using input $\{x^i := 0\}^{i\in\mathcal{H}}$. For each $i \in \mathcal{M}, w \in \mathcal{I}_i$, $\mathcal{S}$ receives $\hat{x}^i_w$ and computes $x^i_w := \hat{x}^i_w \oplus \bigoplus_{i\in[n]} r^i_w$. If any honest party would abort, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and aborts.

6 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as honest $\{P_i\}_{i\in\mathcal{H}}$, using input $x^1 := 0$.

7-8 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as honest $\{P_i\}_{i\in\mathcal{H}}$. If an honest $\mathsf{P_A}$ would abort, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and aborts; otherwise for each $i \in \mathcal{M}$, $\mathcal{S}$ sends (input, x^i) on behalf of P_i to $\mathcal{F}_{\mathsf{mpc}}$.

At any time, $\mathcal{S}$ will answer $\mathcal{A}$'s global key query honestly, since $\mathcal{S}$ knows the global keys of all parties.

Note that since the global keys are randomly selected from $\{0,1\}^\kappa$, $\mathcal{A}$ cannot guess any global key with more than negligible probability. Therefore, in the following, we will assume that it does not happen.

We now show that the joint distribution over the outputs of $\mathcal{A}$ and the honest parties in the real world is indistinguishable from the joint distribution over the outputs of $\mathcal{S}$ and the parties in the ideal world.

Hybrid$_\mathbf{1}$. Same as the hybrid-world protocol, where $\mathcal{S}$ plays the role of honest $\{P_i\}_{i\in\mathcal{H}}$, using the actual inputs $\{x^i\}^{i\in\mathcal{H}}$.

Hybrid$_\mathbf{2}$. Same as **Hybrid$_\mathbf{1}$**, except that in step 5, for each $i \in \mathcal{M}, w \in \mathcal{I}_i$, $\mathcal{S}$ receives $\hat{x}^i_w$ and computes $x^i_w := \hat{x}^i_w \oplus \bigoplus_{i\in[n]} r^i_w$. If any honest party would abort, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs; otherwise for each $i \in \mathcal{M}$, $\mathcal{S}$ sends (input, x^i) on behalf of P_i to $\mathcal{F}_{\mathsf{mpc}}$.

The views produced by the two hybrids are exactly the same. According to Lemma 4.3.1, $\mathsf{P_A}$ will learn the same output in both hybrids with all but negligible probability.

Hybrid$_\mathbf{3}$. Same as **Hybrid$_\mathbf{2}$**, except that, for each $i \in \mathcal{H}$, $\mathcal{S}$ computes $\{r^i_w\}_{w \in \mathcal{I}_i}$ as follows: $\mathcal{S}$ first randomly pick $\{u^i_w\}_{w \in \mathcal{I}_i}$, and then computes $r^i_w := u^i_w \oplus x^i_w$.

The two Hybrids produce exactly the same view.

Hybrid$_\mathbf{4}$. Same as **Hybrid$_\mathbf{3}$**, except that $\mathcal{S}$ uses $\{x^i = 0\}^{i \in \mathcal{H}}$ as input in step 5 and step 6.

Note that although the distribution of $\{x^i\}^{i \in \mathcal{H}}$ in **Hybrid$_\mathbf{3}$** and **Hybrid$_\mathbf{4}$** are different, the distribution of $\{x^i_w \oplus r^i_w\}^{i \in \mathcal{H}}$ are exactly the same. The views produced by the two Hybrids are therefore the same, we will show that P_A aborts with the same probability in both Hybrids.

Observe that the only place where P_A's abort can possibly depends on $\{x^i\}^{i \in \mathcal{H}}$ is in step 7(b). However, this abort depends on which row is selected to decrypt, that is the value of $\lambda_\alpha \oplus z_\alpha$ and $\lambda_\beta \oplus z_\beta$, which are chosen independently random in both Hybrids.

As **Hybrid$_\mathbf{4}$** is the ideal-world execution, this completes the proof when P_A is honest.

Malicious P_A and honest P_B. Let $\mathcal{A}$ be an adversary corrupting $\{P_i\}_{i \in \mathcal{M}}$. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of $\{P_i\}_{i \in \mathcal{M}}$ in the ideal world involving an ideal functionality $\mathcal{F}_{\mathsf{mpc}}$ evaluating f. $\mathcal{S}$ is defined as follows.

1-4 $\mathcal{S}$ acts as honest $\{P_i\}_{i \in \mathcal{H}}$ and plays the functionality of $\mathcal{F}_{\mathsf{Pre}}$, recording all outputs. If any honest party would abort, $\mathcal{S}$ output whatever $\mathcal{A}$ outputs and aborts.

5-6 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as honest $\{P_i\}_{i\in\mathcal{H}}$, using input $\{x^i := 0\}^{i\in\mathcal{H}}$. For each $i \in \mathcal{M}, w \in \mathcal{I}_i$, $\mathcal{S}$ receives $\hat{x}^i_w$ and computes $x^i_w := \hat{x}^i_w \oplus \bigoplus_{i\in[n]} r^i_w$. If any honest party would abort, $\mathcal{S}$ output whatever $\mathcal{A}$ outputs and aborts.

8 For each $i \in \mathcal{M}$, $\mathcal{S}$ sends (input, x^i) on behalf of P_i to $\mathcal{F}_{\mathsf{mpc}}$. If $\mathcal{F}_{\mathsf{mpc}}$ abort, $\mathcal{S}$ aborts, outputting whatever $\mathcal{A}$ outputs. Otherwise, if $\mathcal{S}$ receives z as the output, $\mathcal{S}$ computes $z' := f(y^1, ..., y^n)$, where $\{y^i := 0\}^{i\in\mathcal{H}}$, and $\{y^i := x^i\}^{i\in\mathcal{M}}$. For each $i \in \mathcal{H}, w \in \mathcal{O}$, if $z'_w = z_w$, $\mathcal{S}$ sends $(r^i_w, \mathsf{M}_1[r^i_w])$ on behalf of P_i to $\mathcal{A}$; otherwise, $\mathcal{S}$ sends $(r^i_w \oplus 1, \mathsf{M}_1[r^i_w] \oplus \Delta_1)$.

At any time, $\mathcal{S}$ will answer $\mathcal{A}$'s global key query honestly, since $\mathcal{S}$ knows the global keys of all parties.

Note that since the global keys are randomly selected from $\{0,1\}^\kappa$, $\mathcal{A}$ cannot guess any global key with more than negligible probability. Therefore, in the following, we will assume it does not happen.

We now show that the joint distribution over the outputs of $\mathcal{A}$ and honest parties in the real world is indistinguishable from the joint distribution over the outputs of $\mathcal{S}$ and honest parties in the ideal world.

Hybrid$_1$. Same as the hybrid-world protocol, where $\mathcal{S}$ plays the role of honest $\{P_i\}_{i\in\mathcal{H}}$ using the actual inputs $\{x^i\}^{i\in\mathcal{H}}$.

Hybrid$_2$. Same as **Hybrid$_1$**, except that in step 5 and step 6, for each $i \in \mathcal{M}, w \in \mathcal{I}_i$, $\mathcal{S}$ receives $\hat{x}^i_w$ and computes $x^i_w := \hat{x}^i_w \oplus \bigoplus_{i\in[n]} r^i_w$. If any honest party would abort, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs; otherwise for each $i \in \mathcal{M}$, $\mathcal{S}$ sends (input, x^i) on behalf of P_i to $\mathcal{F}_{\mathsf{mpc}}$.

$\mathsf{P_A}$ does not have output; furthermore the view of $\mathcal{A}$ does not change between the two Hybrids.

Hybrid$_3$. Same as **Hybrid$_2$**, except that in step 5 and step 6, $\mathcal{S}$ uses $\{x^i := 0\}^{i\in\mathcal{H}}$ as input and in step 8, $\mathcal{S}$ computes z' as defined. For each $w \in \mathcal{O}$, if $z'_w = z_w$, $\mathcal{S}$ sends $(r^i_w, \mathsf{M}_1[r^i_w])$; otherwise, $\mathcal{S}$ sends $(r^i_w \oplus 1, \mathsf{M}_1[r^i_w] \oplus \Delta_1)$.

$\mathcal{A}$ has no knowledge of r^i_w, therefore r^i_w and $r^i_w \oplus 1$ are indistinguishable.

Note that since $\mathcal{S}$ uses different values for x between the two Hybrids, we also need to show that the distribution of garbled rows opened by $\mathsf{P_A}$ are indistinguishable for the two Hybrids. According to the security of garbled circuits, $\mathsf{P_A}$ is able to open only one garble rows in each garbled table $G_{\gamma,i}$. Therefore, given that $\{\lambda_w\}_{w\in\mathcal{I}_1\cup\mathcal{W}}$ values are not known to $\mathsf{P_A}$, masked values and garbled keys are indistinguishable between two Hybrids.

As **Hybrid$_3$** is the ideal-world execution, the proof is complete. □

Lemma 4.3.1. *Consider an $\mathcal{A}$ corrupting parties $\{P_i\}_{i\in\mathcal{M}}$ such that $\mathsf{P_A} \in \mathcal{H}$, and denote $x^i_w := \hat{x}^i_w \oplus \bigoplus_{i=1}^n r^i_w$, where $\hat{x}_w$ is the value $\mathcal{A}$ sent, r^i_w are the values from $\mathcal{F}_{\mathsf{Pre}}$. With all but negligible probability , $\mathsf{P_A}$ either aborts or learns $z = f(x^1, ..., x^n)$.*

Proof. Define z^*_w as the correct wire values computed using x defined above and y, z_w as the actually wire values $\mathsf{P_A}$ holds in the evaluation.

We will first show that $\mathsf{P_A}$ learns $\{z^w \oplus \lambda_w = z^*_w \oplus \lambda_w\}_{w\in\mathcal{O}}$ by induction on topology of the circuit.

Base step: It is obvious that $\{z_w^* \oplus \lambda_w = z_w \oplus \lambda_w\}_{w \in \mathcal{I}_1 \cup \mathcal{I}_2}$, unless $\mathcal{A}$ is able to forge an IT-MAC.

Induction step: Now we show that for a gate $(\alpha, \beta, \gamma, T)$, if $\mathsf{P_A}$ has $\{z_w^* \oplus \lambda_w = z_w \oplus \lambda_w\}_{w \in \{\alpha,\beta\}}$, then $\mathsf{P_A}$ also obtains $z_\gamma^* \oplus \lambda_\gamma = z_\gamma \oplus \lambda_\gamma$.

- $T = \oplus$: It is true according to the following: $z_\gamma^* \oplus \lambda_\gamma = (z_\alpha^* \oplus \lambda_\alpha) \oplus (z_\beta^* \oplus \lambda_\beta) = (z_\alpha \oplus \lambda_\alpha) \oplus (z_\beta \oplus \lambda_\beta) z_\gamma \oplus \lambda_\gamma$

- $T = \wedge$: According to the protocol, $\mathsf{P_A}$ will open the garbled row defined by $i := 2(z_\alpha \oplus \lambda_\alpha) + (z_\beta \oplus \lambda_\beta)$. If $\mathsf{P_A}$ learns $z_\gamma \oplus \lambda_\gamma \neq z_\gamma^* \oplus \lambda_\gamma$, then it means that $\mathsf{P_A}$ learns $r_{\gamma,i}^* \neq r_{\gamma,i}$. However, this would mean that $\mathcal{A}$ forge a valid IT-MAC, happening with negligible probability.

Now we know that $\mathsf{P_A}$ learns the correct masked output. $\mathsf{P_A}$ can therefore learn the correct output $f(x, y)$ unless $\mathcal{A}$ is able to flip $\{r_w\}_{w \in \mathcal{O}}$, which, again, only happens with negligible probability. □

4.4 Efficiently Realizing $\mathcal{F}_{\mathsf{Pre}}$

In this section, we describe an efficient instantiation of $\mathcal{F}_{\mathsf{Pre}}$, which is a multi-party version of TinyOT protocol. All previous related protocols [50, 74, 51] for multi-party TinyOT rely on cut-and-choose to ensure correctness and another bucketing to ensure privacy, resulting in a communication/computation complexity at least $\Omega(B^2 n^2)$ per AND triple, where bucket size $B = \rho / \log |C|$ (See Table 4.1 for more detail). Furthermore, these protocols output SPDZ-style shares that are not compatible with our main protocol. Our new protocol introduced in this section works with BDOZ-style shares; furthermore the complexity per AND triple is $O(Bn^2)$ with a very small constant. The new protocol features a new distributed AND triple checking protocol that checks the correctness of an AND triple *without* cut-and-choose. The adversary is still able to perform selective failure attacks on a triple with probability of being caught at least one-half. Such leakage can be easily eliminated using bucketing.

In the following, we will build our protocol from the bottom up. In Section 4.4.1 and Section 4.4.2, we discuss the multi-party authenticated bits and authenticated shares that we also introduced in Section 2.2; in Section 4.4.4, we discuss an AND triple generation protocol that allows an adversary to perform selective failure attacks; the final protocol that eliminates such an attack follows the bucketing protocols used in previous works [45, 7], and is detailed in Section 4.4.5.

Note that similar to the previous chapter, our protocol also relies on two-party $\mathcal{F}_{\mathsf{abit}}$ functionality, which has a random global key for each party preserved across all executions and allows an adversary to make "global-key queries" to honest parties' global keys. Both these features are preserved in all our ideal functionalities, but we suppress explicit mention of them in our descriptions. Global-key queries have little effect on security, since the probability that the attacker can correctly guess the honest party's value of Δ using polynomially many queries is negligible.

Functionality $\mathcal{F}^n_{\mathsf{abit}}$

Honest Parties: The box receives $(\mathsf{input}, i, \ell)$ from all parties and picks random bit-string $x \in \{0,1\}^\ell$. For each $j \in [\ell], k \neq i$, the box picks random $\mathsf{K}_k[x_j]$, and computes $\{\mathsf{M}_k[x_j] := \mathsf{K}_k[x_j] \oplus x_j\Delta_k\}_{k\neq i}$, and sends them to parties. That is, for each $j \in [\ell]$, it sends $\{\mathsf{M}_k[x_j]\}_{k\neq i}$ to P_i and sends $\mathsf{K}_k[x_j]$ to P_k for each $k \neq i$.

Corrupted parties: Corrupted parties can choose their output from the protocol.

Figure 4.3: Functionality for multi-party authenticated bit.

4.4.1 Multi-Party Authenticated Bit

The first step of our protocol is to generate multi-party authenticated bit. The functionality $\mathcal{F}^n_{\mathsf{abit}}$, also discussed in Section 2.2, is shown in Figure 4.3. Notice that if we set $n = 2$, then $\mathcal{F}^2_{\mathsf{abit}}$ is the original two-party authenticated bit functionality [45]. One naive solution to realize $\mathcal{F}^n_{\mathsf{abit}}$ is to let P_i run the two-party authenticated bit protocol with every other party using the same bit x. This solution is not secure, since a malicious P_i can potentially use inconsistent values when running $\mathcal{F}^2_{\mathsf{abit}}$ with other parties. In our protocol, we use this general idea and we also perform additional checks to ensure that P_i uses consistent values. The check is similar to the recent malicious OT extension protocol by Keller et al. [70], where parties perform checks based on random linear combination: a malicious P_i who uses inconsistent values is able to pass ρ checks with probability at most $2^{-\rho}$. Note that these checks also reveal some linear relationship of x's. To eliminate this leakage, a small number of random authenticated bits are computed and checked together. They are later discarded to break the linear relationships. The protocol is described in Figure 4.4 with proof in Section 4.5.1.

Protocol Π^n_{aBit}

Protocol:

1. Set $\ell' := \ell + 2\rho$. P_i picks random bit-string $x \in \{0,1\}^{\ell'}$.
2. For each $k \neq i$, P_i and P_k runs $\mathcal{F}^2_{\mathsf{abit}}$, where P_i sends $\{x_j\}_{j\in[\ell']}$ to $\mathcal{F}^2_{\mathsf{abit}}$. From the functionality, P_i gets $\{\mathsf{M}_k[x_j]\}_{j\in[\ell']}$, P_k gets $\{\mathsf{K}_k[x_j]\}_{j\in[\ell']}$.
3. For $j \in [2\rho]$, all parties perform the following:
   (a) All parties sample a random ℓ'-bit strings r.
   (b) P_i computes $\mathcal{X}_j = \bigoplus_{m=1}^{\ell'} r_m x_m$, and broadcast $\mathcal{X}_j$, and computes $\left\{\mathsf{M}_k[\mathcal{X}_j] = \bigoplus_{m=1}^{\ell'} r_m \mathsf{M}_k[x_m]\right\}_{k\neq i}$.
   (c) P_k computes $\mathsf{K}_k[\mathcal{X}_j] = \bigoplus_{m=1}^{\ell'} r_m \mathsf{K}_k[x_m]$.
   (d) P_i sends $\mathsf{M}_k[\mathcal{X}_j]$ to P_k who check the validity.
4. All parties return the first ℓ objects.

Figure 4.4: The protocol Π^n_{aBit} instantiating $\mathcal{F}^n_{\mathsf{abit}}$.

4.4.2 Multi-Party Authenticated Shares

In this section, we aim to construct a protocol that allows multiple parties to obtain authenticated shares of a secret bit, as shown in Figure 4.5. One straightforward idea is to call $\mathcal{F}^n_{\mathsf{abit}}$ n times, where in the i-th execution, they compute $[x^i]^i$ for some random x^i known only to P_i. However, the adversary is still able to perform an attack: a malicious P_i can potentially use different global MAC keys (Δ_i) in different executions of $\mathcal{F}^n_{\mathsf{abit}}$. The result is that $[x^j]^j$ is authenticated with a global MAC key Δ_i, while some other $[x^k]^k$ is authenticated with a different global MAC key Δ_i'. This attack does not happen in the two-party setting, because each party is authenticated to only one party.

Our key idea is based on the observation that the two-party authenticated bit protocol already ensured that, when P_i and P_j compute multiple authenticated

Functionality $\mathcal{F}_{\mathsf{aShare}}$

Honest Parties: The box receives (input, ℓ) from all parties and picks random bit-strings $x \in \{0,1\}^{\ell}$ and random authenticated shares $\{\langle x_j \rangle\}_{j\in[\ell]}$, and sends them to parties.
In detail, the box picks random bit strings $\{x^i\}^{i\in[n]}$, each of length ℓ bits. For each $i \in [n], j \in [\ell]$, The box picks random multi-party authenticated bits $[x^i_j]^i$ and sends them to parties. That is, for each $j \in [\ell]$, it sends $(x^i_j, \{\mathsf{M}_k[x^i_j], \mathsf{K}_i[x^k_j]\}_{k\neq i})$ to P_i.
Corrupted parties: Corrupted parties can choose their output from the protocol.

Figure 4.5: Functionality for multi-party authenticated share.

Protocol Π_{aShare}

Protocol:

1. Set $\ell' := \ell + \rho$. For each $i \in [n]$, P_i picks random bit-string $x^i \in \{0,1\}^{\ell'}$.
2. For each $i \in [n]$, all parties compute multi-party authenticated bits by sending (i, ℓ') to $\mathcal{F}^n_{\mathsf{abit}}$, which sends $\{[x^i_j]^i\}_{j\in[\ell']}$ to parties.
3. For $r \in [\rho]$, all parties perform the following:
   (a) For each $i \in [n]$, P_i parses $\{[x^k_{\ell+r}]^k\}_{k\in[n]}$ as $(x^i_{\ell+r}, \{\mathsf{M}_k[x^i_{\ell+r}], \mathsf{K}_i[x^k_{\ell+r}]\}_{k\neq i})$. Each P_i computes commitments $(\mathsf{c}^0_i, \mathsf{d}^0_i) \leftarrow \mathsf{Com}(\bigoplus_{k\neq i} \mathsf{K}_i[x^k_{\ell+r}])$, $(\mathsf{c}^1_i, \mathsf{d}^1_i) \leftarrow \mathsf{Com}(\bigoplus_{k\neq i} \mathsf{K}_i[x^k_{\ell+r}] \oplus \Delta_i)$, and $(\mathsf{c}^M_i, \mathsf{d}^M_i) \leftarrow \mathsf{Com}(x^i_{\ell+r}, \{\mathsf{M}_k[x^i_{\ell+r}]\}_{k\neq i})$, and broadcast $(\mathsf{c}^m_i, \mathsf{c}^0_i, \mathsf{c}^1_i)$.
   (b) For each $i \in [n]$, after receiving all commitments, P_i broadcasts d^M_i.
   (c) For each $i \in [n]$, P_i computes $b^i := \bigoplus_{k\neq i} x^k_{\ell+r}$, and broadcast $\mathsf{d}^{b^i}_i$.
   (d) For each $i \in [n]$, P_i performs the following to check the consistency of Δ's: For each $j \neq i$, P_i computes $K^j \leftarrow \mathsf{Open}(\mathsf{c}_{b^j}, \mathsf{d}_{b^j})$ and check if it equals to $\bigoplus_{k\neq j} \mathsf{M}_j[x^k_{\ell+r}]$. If any check fails, the party aborts.
4. All parties return the first ℓ objects.

Figure 4.6: The protocol Π_{aShare} instantiating $\mathcal{F}_{\mathsf{aShare}}$.

bits, P_i uses the same Δ_i across different authenticated bits. Therefore, in the above insecure attempt, if one authenticated share has consistent global MAC keys, then all authenticated shares have consistent global MAC keys, and vice versa. In our secure construction, we first let all parties compute $\ell + \rho$ number of multi-party authenticated shares as described above, which may not be secure. Then we partially open the last ρ tuples to check the consistency of global MAC keys. A malicious party who uses inconsistent Δ_i's will get caught with probability one-half for each partially opened shares.

In more detail, each player P_i will take the role of a prover once to prove that he uses a consistent Δ_i and the remaining players will take the role of verifier for the given prover. The basic idea is that if the prover used a consistent Δ_i, then these authenticated bits across different parties are XOR homomorphic. Taking a three-party setting as an example. Say P_1 has $\mathsf{K}_1[x]$, $\mathsf{K}_2[y]$ with global keys Δ_1^x and Δ_1^y which are potentially different; P_2 has $(x, \mathsf{M}_1[x])$; P_3 has $(y, \mathsf{M}_1[y])$. In our checking protocol, we let P_1 commit to values $\mathsf{K}_1[x] \oplus \mathsf{K}_1[y]$ and $\mathsf{K}_1[x] \oplus \mathsf{K}_1[y] \oplus \Delta_1$. For an adversary who uses inconsistent global keys, it needs to choose two values out of the following four values to commit.

$x = 0$	$y = 0$	$\mathsf{K}_1[x] \oplus \mathsf{K}_2[y]$
$x = 0$	$y = 1$	$\mathsf{K}_1[x] \oplus \mathsf{K}_2[y] \oplus \Delta_1^y$
$x = 1$	$y = 0$	$\mathsf{K}_1[x] \oplus \mathsf{K}_2[y] \oplus \Delta_1^x$
$x = 1$	$y = 1$	$\mathsf{K}_1[x] \oplus \mathsf{K}_2[y] \oplus \Delta_1^x \oplus \Delta_1^y$

Later in the protocol, the adversary is asked to open the MAC for $x \oplus y$. If inconsistent global keys are used, this value can be any of the four values each with one-fourth probability, therefore, the adversary can win with probability at most $2/4 = 1/2$. The details of the protocol are shown in Figure 4.6 with proof in Section 4.5.2.

Functionality $\mathcal{F}_{\mathsf{HaAND}}$

Honest Parties:

1. The box picks random $\langle x \rangle$ and sends it to all parties.
2. Upon receiving $(i, \{y^i_j\}_{j \neq i})$ from all P_i, the box picks random bits $\{v^i\}_{i \in [n]}$ such that $\bigoplus_i v^i := \bigoplus_i \bigoplus_{j \neq i} x^i y^j_i$. The box sends v^i to P_i.

Corrupted parties: Corrupted parties can choose their output from the protocol.

Figure 4.7: The Half Authenticated AND Functionality

Protocol Π_{HaAND}

Protocol:

1. All parties call $\mathcal{F}_{\mathsf{aShare}}$ to obtain $\langle x \rangle$.
2. For each $i, j \in [n]$, such that $i \neq j$,
   (a) P_i picks a random bit s^j, and computes $H_0 := \mathsf{lsb}(H(\mathsf{K}_i[x^j])) \oplus s^j$, $H_1 = \mathsf{lsb}(H(\mathsf{K}_i[x^j] \oplus \Delta_i)) \oplus s^j \oplus y^i_j$.
   (b) P_i sends (H_0, H_1) to P_j, who computes $t^i := H_{x^j} \oplus \mathsf{lsb}(H(\mathsf{M}_i[x^j]))$.
3. For each $i \in [n]$, P_i obtains $v^i := \bigoplus_{k \neq i} (t^k \oplus s^k)$.

Figure 4.8: Protocol Π_{HaAND} instantiating $\mathcal{F}_{\mathsf{HaAND}}$.

4.4.3 Half-Authenticated AND Triple

Before introducing the protocol for leaky authenticated AND triples, there is yet another tool that we need. As described in Figure 4.7, the functionality $\mathcal{F}_{\mathsf{HaAND}}$ is introduced to compute cross terms in AND triples. It takes unauthenticated and potentially inconsistent y's and outputs authenticated share $\langle x \rangle$ as well as unauthenticated shares of cross product terms. Details about the protocol is in Figure 4.8 with proof included in Section 4.5.2.

4.4.4 Multi-Party Leaky Authenticated AND Triple

Now we are ready to discuss the protocol for leaky authenticated AND triples. It can be divided into following steps:

1. Call $\mathcal{F}_{\mathsf{aShare}}$ to obtain some random $\langle y \rangle$ and $\langle r \rangle$.
2. Call $\mathcal{F}_{\mathsf{HaAND}}$ with y to obtain a random $\langle x \rangle$ and compute shares $\{z^i\}$, such that $(\bigoplus_i x^i) \wedge (\bigoplus_i y^i) = \bigoplus_i z^i$.
3. Reveal $d = z \oplus r$ and computes $\langle z \rangle := \langle r \rangle \oplus d$.
4. Perform additional check to ensure the correctness of the AND relationship.

In the above steps, the adversary is able to cheat by using inconsistent values of y and z between step 1 and 2. However, this only allows the adversary to perform selective failure attack on x^i's. For example, the AND relationship checked is

$$\left(\bigoplus_i x^i\right) \wedge \left(\bigoplus_i y^i\right) = \left(\bigoplus_i z^i\right).$$

The adversary can guess that the value of $\bigoplus x^i = 0$ and flip y^j for some $j \in \mathcal{M}$. If the guess is correct, than the check will go through and the protocol will proceed as normal. However, if the guess is wrong, then the checking will abort and the adversary is caught.

One main challenge is what leakage we should aim for in this functionality. We would like to limit the leakage to be possible only on x^i's, otherwise we would need more bucketing for each possible leakage, as also noted by Nielsen et al. [45]. On the other hand, the adversary can do more attacks than the one mentioned above:

it is also possible to, for example, learn $\bigoplus_{i \in S} x^i$ for some set $S \subset [n]$. We find that the best way to abstract such attack is to allow the adversary to perform a linear check on the value of x^i's. As shown in Figure 4.9, the adversary is allowed to send a list of coefficients and check if the inner product between the coefficients and x values is zero or not.

Our checking phase differs substantially from existing works. We design an efficient checking protocol, that ensures the correctness of the triple (if no party aborts) which allows malicious parties to learn k bits of some specific information with probability at most 2^{-k}. In the two-party protocol, one party constructs "checking tables" and lets the other party to evaluate/check. In the multi-party protocol here, we instead let all parties distributively construct the "checking tables". Interestingly, distributively constructing these checks is inspired by the main protocol where parties distributively construct garbled tables. As noted before, this protocol is vulnerable to selective failure attacks. The full description of this protocol is presented in Figure 4.10.

In the following, we will show the correctness and unforgeability of the protocol, which are crucial to the security proof of the protocol.

4.4.4.1 Correctness of the protocol

We want to show that the protocol will compute a correct triple and will not abort if all parties are honest. Notice that the value we are checking can be written as:

Functionality $\mathcal{F}_{\mathsf{LaAND}}$

Honest parties: For each $i \in [n]$, the box picks random $\langle x\rangle, \langle y\rangle, \langle z\rangle$ such that $(\bigoplus x^i) \wedge (\bigoplus y^i) = \bigoplus z^i$.

Corrupted parties:

1. Corrupted parties can choose all their randomness.
2. An adversary can send $(Q, \{R_i\}_{i\in[n]})$, which are κ-bit strings, to the box and perform a linear combination test. The box checks

$$Q \oplus \bigoplus_i x^i R_i = 0$$

If the check is incorrect, the box outputs fail and terminates, otherwise the box proceeds as normal.

3. An adversary can also send $(q, \{r_i\}_{i\in[n]})$, which are all bits, to the box and perform a linear combination test. The box checks

$$q \oplus \bigoplus_i x^i r_i = 0$$

If the check is incorrect, the box outputs fail and terminates, otherwise the box proceeds as normal.

Figure 4.9: Functionality $\mathcal{F}_{\mathsf{LaAND}}$ for leaky AND triple.

$$\begin{aligned}
&\bigoplus_i H_i \\
&= \bigoplus_i \left(x^i\Phi_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[x^k]_{\Phi_i} \oplus \mathsf{M}_k[x^i]_{\Phi_k} \right) \oplus z^i\Delta_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[z^k] \oplus \mathsf{M}_k[z^i] \right) \right) \\
&= \bigoplus_i \left(x^i\Phi_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[x^k]_{\Phi_i} \oplus \mathsf{M}_k[x^i]_{\Phi_k} \right) \right) \oplus \bigoplus_i \left(z^i\Delta_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[z^k] \oplus \mathsf{M}_k[z^i] \right) \right) \\
&= \bigoplus_i \left(x^i\Phi_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[x^k]_{\Phi_i} \oplus \mathsf{M}_i[x^k]_{\Phi_k} \right) \right) \oplus \bigoplus_i \left(z^i\Delta_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[z^k] \oplus \mathsf{M}_i[z^k] \right) \right) \\
&= \bigoplus_i \left(x^i\Phi_i \oplus \left(\bigoplus_{k\neq i} x^k\Phi_i \right) \right) \oplus \bigoplus_i \left(z^i\Delta_i \oplus \left(\bigoplus_{k\neq i} z^k\Delta_i \right) \right) \\
&= \left(\bigoplus_i x^i \right) \cdot \left(\bigoplus_i \Phi_i \right) \oplus \left(\bigoplus_i z^i \right) \cdot \left(\bigoplus_i \Delta_i \right)
\end{aligned}$$

Notice further that

$$\bigoplus_i \Phi_i = \bigoplus_i \left(y^i\Delta_i \oplus \left(\bigoplus_{k\neq i} \mathsf{K}_i[y^k] \oplus \mathsf{M}_k[y^i] \right) \right) = \left(\bigoplus_i y^i \right) \cdot \left(\bigoplus_i \Delta_i \right)$$

Protocol Π_{LaAND}

Triple computation.

1. For each $i \in [n]$ each party calls $\mathcal{F}_{\mathsf{aShare}}$ and obtains random authenticated shares $\{\langle y \rangle, \langle r \rangle\}$. All parties also calls $\mathcal{F}_{\mathsf{HaAND}}$ to obtain random authenticated share $\langle x \rangle$.
2. For each $i \in [n]$, P_i sends $(i, \{y^i\}_{j \neq i})$ to $\mathcal{F}_{\mathsf{HaAND}}$ and gets back some v^i.
3. For each $i \in [n]$, P_i computes $z^i := x^i y^i \oplus v^i$ and $e^i := z^i \oplus r^i$. P_i broadcasts e^i to all other parties. All parties computes $[z^i]^i := [r^i]^i \oplus e^i$.

Triple checking.

4. For each $i \in [n]$, P_i computes: $\Phi_i := y^i \Delta_i \oplus \left(\bigoplus_{k \neq i} \mathsf{K}_i[y^k] \oplus \mathsf{M}_k[y^i] \right)$.
5. For every pair of $i, j \in [n]$, such that $i \neq j$, P_i computes $\mathsf{K}_i[x^j]_{\Phi_i} := H(\mathsf{K}_i[x^j])$ and $U_{i,j} := H(\mathsf{K}_i[x^j] \oplus \Delta_i) \oplus \mathsf{K}_i[x^j]_{\Phi_i} \oplus \Phi_i$, and sends $U_{i,j}$ to P_j. P_j computes $\mathsf{M}_i[x^j]_{\Phi_i} := x^j U_{i,j} \oplus H(\mathsf{M}_i[x^j])$.
6. For $i \in [n]$, P_i computes
$H_i := x^i \Phi_i \oplus \left(\bigoplus_{k \neq i} \mathsf{K}_i[x^k]_{\Phi_i} \oplus \mathsf{M}_k[x^i]_{\Phi_k} \right) \oplus z^i \Delta_i \oplus \left(\bigoplus_{k \neq i} \mathsf{K}_i[z^k] \oplus \mathsf{M}_k[z^i] \right)$.
All parties simultaneously broadcast H_i by first broadcasting the commitment of H_i and send the decommitment after receiving commitments from all parties.
7. Each party check if $\bigoplus_i H_i = 0$ and abort if not true.

Figure 4.10: The protocol Π_{LaAND}.

Therefore we know that

$$\begin{aligned}
\bigoplus_i H_i &= \left(\bigoplus_i x^i \right) \cdot \left(\bigoplus_i \Phi_i \right) \oplus \left(\bigoplus_i z^i \right) \cdot \left(\bigoplus_i \Delta_i \right) \\
&= \left(\bigoplus_i x^i \right) \cdot \left(\bigoplus_i y^i \right) \cdot \left(\bigoplus_i \Delta_i \right) \oplus \left(\bigoplus_i z^i \right) \cdot \left(\bigoplus_i \Delta_i \right) \\
&= \left(\left(\bigoplus_i x^i \right) \cdot \left(\bigoplus_i y^i \right) \oplus \left(\bigoplus_i z^i \right) \right) \cdot \left(\bigoplus_i \Delta_i \right)
\end{aligned}$$

Since $\bigoplus_i \Delta_i$ is non-zero, $\bigoplus_i H_i = 0$ if and only if the logic of this AND is correct.

4.4.4.2 Unforgeability

Now we want to show that any incorrect AND triple cannot pass the check.

Lemma 4.4.1. *Define x^i, y^i from $\langle x \rangle, \langle y \rangle$ which are outputs from $\mathcal{F}_{\mathsf{aShare}}$ and $\mathcal{F}_{\mathsf{HaAND}}$; define $z^i := r^i \oplus e^i$, where $\langle r \rangle$ is output from $\mathcal{F}_{\mathsf{aShare}}$, e^i is the value broadcast from*

P_i. If $(\bigoplus_i x^i) \wedge (\bigoplus_i y^i) \neq (\bigoplus z^i)$ then the protocol results in an abort except with negligible probability.

We use $U^*_{i,j}$ and H^*_i to denote the values that an honest party would have computed, and define $Q_{i,j} = U^*_{i,j} \oplus U_{i,j}$, $Q_i = H^*_i \oplus H_i$. In the following, we will assume that the logic of the AND does not hold while at the same time that the check passes, and we will derive a contradiction from it.

First note that if P_i uses some $Q_{i,j}$, then P_j will obtain $\mathsf{M}_i[x^j]_{\Phi_i}$ with an additive error of $x^j Q_{i,j}$. Note that

$$\bigoplus_i H^*_i = \left(\left(\bigoplus_i x^i\right) \cdot \left(\bigoplus_i y^i\right) \oplus \left(\bigoplus_i z^i\right)\right) \cdot \left(\bigoplus_i \Delta_i\right) = \bigoplus_i \Delta_i$$

Therefore, we know that

$$\begin{aligned}
\bigoplus_i H_i &= \bigoplus_{i\in\mathcal{M}} H_i \oplus \bigoplus_{i\in\mathcal{H}} H_i \\
&= \bigoplus_{i\in\mathcal{M}} (H^*_i \oplus Q_i) \oplus \bigoplus_{i\in\mathcal{H}} \left(H^*_i \oplus \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right)\right) \\
&= \bigoplus_i H^*_i \oplus \bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{i\in\mathcal{H}} \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right) \\
&= \bigoplus_i \Delta_i \oplus \bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{i\in\mathcal{H}} \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right)
\end{aligned}$$

In order to make $\bigoplus_i H_i$ to be 0, the adversary needs to find paddings such that

$$\bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{i\in\mathcal{H}} \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right) = \bigoplus_i \Delta_i$$

The above happens with at most negligible probability.

Theorem 4.4.1. *Assuming an adversary corrupting up to $n-1$ parties, the protocol in Figure 4.10, where H is modeled as a random oracle, securely instantiates $\mathcal{F}_{\mathsf{LaAND}}$ functionality in the $(\mathcal{F}_{\mathsf{aShare}}, \mathcal{F}_{\mathsf{HaAND}})$-hybrid model.*

Note that since no party has private input, the simulation proof is straightforward given the lemmas above. We provide full details of the proof in Section 4.5.4.

Functionality $\mathcal{F}_{\mathsf{aAND}}$

Honest parties: For each $i \in [n]$, the box picks random $\langle x \rangle, \langle y \rangle, \langle z \rangle$ such that $(\bigoplus x^i) \wedge (\bigoplus y^i) = \bigoplus z^i$.

Corrupted parties: Corrupted parties get to choose all of their randomness.

Figure 4.11: Functionality $\mathcal{F}_{\mathsf{aAND}}$ for generating AND triples

Protocol Π_{aAND}

Protocol:

1. P_i call $\mathcal{F}_{\mathsf{LaAND}}$ $\ell' = \ell B$ times and obtains $\{\langle x_j \rangle, \langle y_j \rangle, \langle z_j \rangle\}_{j \in [\ell']}$.
2. All parties randomly partition all objects into ℓ buckets, each with B objects.
3. For each bucket, parties combine B leaky ANDs into one non-leaky AND. To combine two leaky ANDs, namely $(\langle x_1 \rangle, \langle y_1 \rangle, \langle z_1 \rangle)$ and $(\langle x_2 \rangle, \langle y_2 \rangle, \langle z_2 \rangle)$:
   (a) Parties reveal $d := y_1 \oplus y_2$ with its MACs checked.
   (b) Each party P_i sets $\langle x \rangle := \langle x_1 \rangle \oplus \langle x_2 \rangle$, $\langle y \rangle := \langle y_1 \rangle$, $\langle z \rangle := \langle z_1 \rangle \oplus \langle z_2 \rangle \oplus d \langle x_2 \rangle$.

   Parties iterate all B leaky objects, by taking the resulted object and combine with the next element.

Figure 4.12: Protocol Π_{aAND} instantiating $\mathcal{F}_{\mathsf{aAND}}$.

4.4.5 Multi-Party Authenticated AND Triple

Once we have a protocol for leaky authenticated AND triple, it is straightforward to obtain a non-leaky authenticated AND triple, using the combine protocol described in the previous chapter. We show the details of the protocol in Figure 4.12.

4.5 Proofs of Security

4.5.1 Multi-Party Authenticated Bits

Theorem 4.5.1. *The protocol in Figure 4.4 securely realizes* $\mathcal{F}^{n}_{\mathsf{abit}}$ *with statistical security* $2^{-\rho}$ *in the* $\mathcal{F}^{2}_{\mathsf{abit}}$*-hybrid model.*

Proof. We consider two cases.

Case 1: $P_i \in \mathcal{H}$**.** Note that in this case, the only way malicious parties can break the protocol is by learning some information about $\{x_i\}_{i\in[\ell]}$ in the checking step. However, we will show that, because we "throw out" the last 2ρ authenticated bits, the adversary can learn nothing about x's.

We use s_j to denote the last 2ρ bits of r in the j-th check. According to Lemma 4.5.1 and the parameters we chose, the probability that any subset of $\{s_j\}_{j\in[2\rho]}$ is linearly independent is $1 - 2^{-\rho}$. Now we will show that if linear independence holds then the adversary cannot learn anything.

For the j-checking, $\mathcal{X} = \left(\bigoplus_{m=1}^{\ell} r_m x_m\right) \oplus \left(\bigoplus_{m=1}^{2\rho} s_m x_{\ell+m}\right)$. Note that $\bigoplus_{m=1}^{2\rho} s_m x_{\ell+m}$ from each checking are independent random bits, where $\{x_m\}_{m=\ell}^{\ell'}$ is random. This is true because the s_i's are linearly independent. Therefore, $\bigoplus_{m=1}^{2\rho} s_m x_{\ell+m}$ acts as one-time pad to $\bigoplus_{m=1}^{\ell} r_m x_m$. Given the above, the simulation is straightforward. Note that for all global key queries, $\mathcal{S}$ can answer them honestly, since $\mathcal{S}$ knows the global key for both parties.

Case 2: $P_i \in \mathcal{M}$**.** The simulation is straightforward if we could show that for any $\mathcal{A}$ who uses inconsistent x's can pass all 2ρ checks with at most negligible probability.

This is what we will proceed to show.

Suppose that $\mathcal{A}$ sends x^1 to $\mathcal{F}_{\mathsf{abit}}^2$ when interacting with one honest party, and uses a different x^2 with another honest party, where $x^1 \neq x^2$. We also assume that $\mathcal{A}$ passes all checks. Note that for the j-th checking, if $\mathcal{A}$ is not able to forge a MAC, then the probability that the checking passes is the probability that $\mathcal{X}_j = \bigoplus_m r_m x_m^1$ and that $\mathcal{X}_j = \bigoplus_m r_m x_m^2$.

$$
\begin{aligned}
&\Pr\left\{\bigoplus_m r_m x_m^1 = \bigoplus_m r_m x_m^2\right\} \\
&= \Pr\left\{\bigoplus_m r_m (x_m^1 \oplus x_m^2) = 0\right\} \\
&= \Pr\left\{\bigoplus_{m \in I} r_m = 0 : I \text{ is the set of indices where } x_m^1 \neq x_m^2\right\} \\
&= 1/2
\end{aligned}
$$

Each checking is independent as long as r is selected independently. Therefore, $\mathcal{A}$ can pass all checks with probability at most $2^{-2\rho}$. □

Lemma 4.5.1. *Let $r_1, \ldots, r_\ell$ be random bit vectors of length k. With probability at most $2^{\ell-k}$, there exists some subset $I \subset [\ell]$, such that*

$$\bigoplus_{i \in I} r_i = 0$$

Proof. Note that given a fixed interval $I \subset [\ell]$, the probability that $\bigoplus_{i \in I} r_i = 0$ is 2^{-k}. According to the union bound, the probability that any subset $I \subset [\ell]$ has $\bigoplus_{i \in I} r_i = 0$ is $2^{-k} \times 2^\ell = 2^{\ell-k}$. □

4.5.2 Multi-Party Authenticated Bits

Theorem 4.5.2. *The protocol in Figure 4.6 securely realizes $\mathcal{F}_{\mathsf{aShare}}$ with statistical security $2^{-\rho}$ in the $\mathcal{F}^n_{\mathsf{abit}}$-hybrid model.*

Proof. Without loss of generality, we consider the case when $\mathsf{P_A}$ is honest. The simulator plays the role of $\mathcal{F}^n_{\mathsf{abit}}$ honestly, recording all values it sends to $\mathcal{A}$ and values $\mathcal{A}$ sent to $\mathcal{F}^n_{\mathsf{abit}}$. $\mathcal{S}$ acts as honest parties and check for each $i \in \mathcal{M}$, if P_i sent consistent Δ_i in all instructions to $\mathcal{F}^n_{\mathsf{abit}}$. If not, $\mathcal{S}$ aborts outputting whatever $\mathcal{A}$ outputs.

Note that this simulator has a $2^{-\rho}$ statistical difference to the real world execution given Lemma 4.5.2 □

Lemma 4.5.2. *When a malicious P_i computes MACs with P_j, denote Δ_i^j as the value P_i sent to $\mathcal{F}^n_{\mathsf{abit}}$. If for some $\Delta_i^{j1} \neq \Delta_i^{j2}$, the honest parties abort with probability at least $1 - 2^{-\rho}$.*

Proof. In the following, we will prove that malicious party passes each single test with probability 1/2, independently. Since malicious parties are ensured to use the same Δ for each party, it can either cheat for all bits or be honest for all bits. Therefore cheating malicious parties cannot pass all checks with more than $2^{-\rho}$ probability.

We will prove by contradiction. Suppose at least one party uses inconsistent value and the check passes. We use K^i to denote the value P_i opened in step 3 (d), and use $\{M^i_k\}_{k \neq i}$ to denote the value committed in step 3 (c). First compute

$Q^i := K^i \oplus \bigoplus_{k \neq i} \mathsf{K}_i[x^k_{\ell+r}]$ and and $Q^i_k = M^i_k \oplus \mathsf{M}_k[x^i_{\ell+r}]$. Since the check for P_i passes, we know that the following is zero.

$$\begin{aligned}
K^i \oplus \bigoplus_{k \neq i} M^k_i &= \left(\bigoplus_{k \neq i} \mathsf{K}_i[x^k_{\ell+r}]\right) \oplus Q^i \oplus \left(\bigoplus_{k \neq i} \mathsf{M}_i[x^k_{\ell+r}] \oplus Q^k_i\right) \\
&= \left(\bigoplus_{k \neq i} \mathsf{K}_i[x^k_{\ell+r}]\right) \oplus Q^i \\
&\quad \oplus \left(\bigoplus_{k \neq i} \mathsf{K}_i[x^k_{\ell+r}] \oplus x^k_{\ell+r}\Delta^k_i\right) \oplus \bigoplus_{k \neq i} Q^k_i \\
&= \left(Q^i \oplus \bigoplus_{k \neq i} Q^k_i\right) \oplus \left(\bigoplus_{k \neq i} x^k_{\ell+r}\Delta^k_i\right)
\end{aligned}$$

If P_i uses $\Delta^{k1}_{\ell+r} \neq \Delta^{k2}_{\ell+r}$ for some $k1 \neq k2$, $k1, k2 \in \mathcal{H}$, then the value of $\left(Q^i \oplus \bigoplus_{k \neq i} Q^k_i\right)$ that makes the equation as 0 is different depending on the value of $x^{k1}_{\ell+r}$ and $x^{k2}_{\ell+r}$. This means that P_i needs to guess at least one of them to pass the check. □

4.5.3 Half-Authenticated AND Triple

Lemma 4.5.3. *If H is modeled as a random oracle, the protocol in Figure 4.8 securely realizes $\mathcal{F}_{\mathsf{HaAND}}$ in the $\mathcal{F}_{\mathsf{aShare}}$-hybrid model.*

Proof. Note that for each $i \in [n]$, P_i has values $\{s^j, t^j\}_{j \neq i}$. We denote the value s^j, t^j held by P_i as s^j_i, t^j_i.

Correctness. First we will show the correctness of the protocol. We further first show that for any $i \neq j$, $s^j_i \oplus t^i_j = x^j y^i_j$. We will discuss in two cases:

- $x^j = 0$. In this case, P_j obtains $t^i_j = s^j$.
- $x^j = 1$. In this case, P_j obtains $t^i_j = s^j \oplus y^i_j$.

In any case, the above equation holds. Now the correctness of the protocol can be seen given the following equation.

$$\begin{aligned}
\bigoplus_i \bigoplus_{j\neq i} x^i y_i^j &= \bigoplus_i \bigoplus_{j\neq i} (s_i^j \oplus t_j^i) \\
&= \bigoplus_i \bigoplus_{j\neq i} s_i^j \oplus \bigoplus_i \bigoplus_{j\neq i} t_j^i \\
&= \bigoplus_i \bigoplus_{j\neq i} s_j^i \oplus \bigoplus_i \bigoplus_{j\neq i} t_j^i \\
&= \bigoplus_i \left(\bigoplus_{j\neq i} s_j^i \oplus t_j^i \right) \\
&= \bigoplus_i v^i
\end{aligned}$$

Simulation proof. We will prove the security assuming $\mathsf{P_A}$ is honest, that is, $\mathsf{P_A} \in \mathcal{H}$. The simulation is as follows:

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{aShare}}$ storing all values used.

2. For each pair $i \neq j$, such that $i \in \mathcal{M}$, $\mathcal{S}$ obtains (H_0, H_1) sent by malicious P_i. $\mathcal{S}$ computes $s^j := H_0 \oplus \mathsf{lsb}(H(\mathsf{K}_i[x^j]))$ and $y_j^i := H_1 \oplus \mathsf{lsb}(H(\mathsf{K}_i[x^j] \oplus \Delta_i)) \oplus s^j$. For each $i \in \mathcal{M}$, $\mathcal{S}$ sends $(i, \{y_j^i\}_{j\neq i})$ to $\mathcal{F}_{\mathsf{HaAND}}$, which sends back $\{v^i\}_{i\in\mathcal{M}}$.

3. For each $i \in \mathcal{M}$, $\mathcal{S}$ picks random $\{t'^k\}_{k\in\mathcal{H}}$, such that $\bigoplus_{k\neq i} s^i \oplus \bigoplus_{k\neq i, k\in\mathcal{M}} t_i^k \oplus \bigoplus_{k\neq i, k\in\mathcal{H}} t'^i = v^i$. For each $j \in \mathcal{H}$, $\mathcal{S}$ computes $H_{x^i} = \mathsf{lsb}(H(\mathsf{K}[x_i] \oplus x_i \Delta_j)) \oplus t'^j$, and picks a random $H_{1\oplus x^i}$. $\mathcal{S}$ sends (H_0, H_1) to P_i on behalf of an honest P_j.

First of all, the first two steps are perfect simulation. For the last step, it is also a perfect simulation: first the one that is not opened is random since H is a random oracle. The other value is also random, depending on the value of s^j. However, in order to make the joint distribution of the value $\mathcal{A}$ learns here and the output of an honest $\mathsf{P_B}$ indistinguishable between ideal and real world protocol, t^k are tweaked such that $\mathcal{A}$ will learn the same value in both Hybrids. □

4.5.4 Multi-Party Leaky Authenticated AND Triple

We have described the protocol and the key ideas of the proof in the main body. Here we will directly proceed to the proof.

Theorem 4.5.3. *If H is modeled as a random oracle, the protocol in Figure 4.10 securely realizes $\mathcal{F}_{\mathsf{LaAND}}$ in the $(\mathcal{F}_{\mathsf{aShare}}, \mathcal{F}_{\mathsf{HaAND}})$-hybrid model.*

Proof. We constructor a simulator in the following. For all global key queries, $\mathcal{S}$ redirect them to $\mathcal{F}_{\mathsf{aShare}}$ and redirect the answer to $\mathcal{A}$.

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{aShare}}$ storing all information sent to parties.

2-3 $\mathcal{S}$ obtains $(i, \{y^i_j\}_{j\neq i})$ for each $P_i \in \mathcal{M}$. $\mathcal{S}$ also obtains $\{e^i\}_{i\in\mathcal{M}}$ $\mathcal{A}$ broadcasts. $\mathcal{S}$ first computes e^{*i}, which are what an honest P_i would have broadcast and compute $q_i := e^i \oplus e^{*i}$. $\mathcal{S}$ further computes $r_{i,j} := y^i_j \oplus y^i$, where y^i is the value $\mathcal{S}$ used when playing the role of $\mathcal{F}_{\mathsf{aShare}}$. $\mathcal{S}$ computes $r_i := \bigoplus_{j\in\mathcal{M}, j\neq i} r_{j,i}$ $q := \bigoplus_{i\in\mathcal{M}} q_i$, and sends $(q, \{r_i\}_n)$ to $\mathcal{F}_{\mathsf{LaAND}}$. If $\mathcal{F}_{\mathsf{LaAND}}$ terminates, $\mathcal{S}$ follows the protocol as honest parties and abort in step 7.

4-5. For each $i \in \mathcal{M}$, $\mathcal{S}$ receives $\{U_{i,j}\}_{j\in\mathcal{H}}$ from P_i. $\mathcal{S}$ picks random $\{U_{j,i}\}_{j\in\mathcal{H}}$ and sends them to P_i playing the role of P_j for each $j \in \mathcal{H}$.

6-7 If $\mathcal{F}_{\mathsf{LaAND}}$ terminates in step 2, then $\mathcal{S}$ follows the protocol as honest parties and abort in step 7. If the equation hold, $\mathcal{S}$ will extract another selective failure attack query.

Similar to the unforgeability proof, we use $U^*_{i,j}$ and H^*_i to denote the values that an honest party would have compute, and define $Q_{i,j} = U^*_{i,j} \oplus U_{i,j}$, $Q_i = H^*_i \oplus H_i$. This means that is a malicious P_i uses some $Q_{i,j}$, then P_j will obtain some $\mathsf{M}_i[x^j]_{\Phi_i}$ with an additive error of $x^j Q_{i,j}$. $\mathcal{S}$ defines $R_k = \bigoplus_{i \neq k, i \in \mathcal{H}} Q_{k,i}$. $\mathcal{S}$ sends $(\bigoplus_{i \in \mathcal{M}} Q_i, \{R_i\}_{i \in [n]})$ to $\mathcal{F}_{\mathsf{LaAND}}$. If $\mathcal{F}_{\mathsf{LaAND}}$ terminates, $\mathcal{S}$ aborts outputting whatever $\mathcal{A}$ outputs; otherwise, $\mathcal{S}$ obtains $\{H_i\}_{i \in \mathcal{M}}$ and picks random $\{H_i\}_{i \in \mathcal{H}}$ such that $\bigoplus_i H_i = 0$.

Note that the first five steps are perfectly indistinguishable given that H is a random oracle, except that $\mathcal{A}$ can perform a selective failure attack. We will show that the probability of abort due to this attack is the same between real-world protocol and ideal-world protocol. The probability that the value $\mathcal{A}$ sent in step 2 and 3 cause an abort is the same as $\mathcal{S}$'s query to $\mathcal{F}_{\mathsf{LaAND}}$, noticing that the following is true.

$$\begin{aligned}
&\bigoplus_i \bigoplus_{j \neq i} x_i y_i^j \oplus \bigoplus_i x^i y^i \oplus \bigoplus_i (e^i \oplus r^i) \\
&= \bigoplus_i \bigoplus_{j \in \mathcal{M}, j \neq i} x_i y_i^j \oplus \bigoplus_i \bigoplus_{j \in \mathcal{H}, j \neq i} x_i y^j \oplus \bigoplus_i x^i y^i \oplus \bigoplus_i z^i \oplus \bigoplus_{i \in \mathcal{M}} q_i \\
&= \bigoplus_i \bigoplus_{j \in \mathcal{M}, j \neq i} x_i r_{j,i} \oplus \bigoplus_i \bigoplus_{j \neq i} x_i y^j \oplus \bigoplus_i z^i \oplus \bigoplus_{i \in \mathcal{M}} q_i \\
&= \bigoplus_i \bigoplus_{j \in \mathcal{M}, j \neq i} x_i r_{j,i} \oplus \bigoplus_{i \in \mathcal{M}} q_i \\
&= \left(\bigoplus_i x_i \bigoplus_{j \in \mathcal{M}, j \neq i} r_{j,i} \right) \oplus \bigoplus_{i \in \mathcal{M}} q_i
\end{aligned}$$

We will focus on the last step. If in step 6, it is the case that $(\bigoplus_i x^i)(\bigoplus_i y^i) \neq (\bigoplus_i z^i)$, then it is easy to see that the views are indistinguishable: all parties behave the same between hybrids. According to the unforgeability lemma, the protocol will abort with all but negligible probability. In the following, we will further focus on the case when the equation holds.

Note that in the idea world protocol, all H_i from $\mathcal{H}$ are picked randomly. We need to show that in the real world protocol all H_i's is also a random share of 0. In particular, we define

$$F_i^* = \left(\bigoplus_{k \neq i} \mathsf{K}_i[x^k]_{\Phi_i} \oplus \mathsf{M}_k[x^i]_{\Phi_k} \right)$$

and will first show that for any proper subset $S \subset \mathcal{H}$, $\bigoplus_{i \in S} F_i$ is indistinguishable from random to the $\mathcal{A}$. We use e to denote an honest party such that $e \in \mathcal{H}, e \notin S$. Such e always exists, since S is a proper subset of $\mathcal{H}$.

$$\begin{aligned}
\bigoplus_{i \in S} F_i^* &= \bigoplus_{i \in S} \bigoplus_{k \neq i} \left(\mathsf{K}_i[x^k]_{\Phi_i} \oplus \mathsf{M}_k[x^i]_{\Phi_k} \right) \\
&= \bigoplus_{i \in S} \bigoplus_{k \neq i} \left(\mathsf{K}_i[x^k]_{\Phi_i} \right) \oplus \bigoplus_{i \in S} \bigoplus_{k \neq i} \left(\mathsf{M}_k[x^i]_{\Phi_k} \right) \\
&= \bigoplus_{i \in S} \bigoplus_{k \neq i} \left(\mathsf{K}_i[x^k]_{\Phi_i} \right) \oplus \bigoplus_{k \in S} \bigoplus_{i \neq k} \left(\mathsf{M}_i[x^k]_{\Phi_i} \right) \\
&= \bigoplus_{i \in S} \bigoplus_{k \neq i} \left(\mathsf{K}_i[x^k]_{\Phi_i} \right) \oplus \bigoplus_{i \in [n]} \bigoplus_{k \in S, k \neq i} \left(\mathsf{M}_i[x^k]_{\Phi_i} \right)
\end{aligned}$$

From the equation, it is clear that for $i \in S$, $\mathsf{K}_e[x^i]$ is not in the computation, while $\mathsf{M}_e[x^i]$ is. Since $\mathsf{K}_e[x^i]$ is randomly picked by P_e, we know $\bigoplus_{i \in S} F_i^*$ is random. Therefore we can see that for any proper subset $S \subset \mathcal{H}$, $\bigoplus_{i \in S} H_i$ is indistinguishable from random.

Finally we need to show that the probability of abort due to selective failure attack is also the same. This is straightforward given the equation used in the

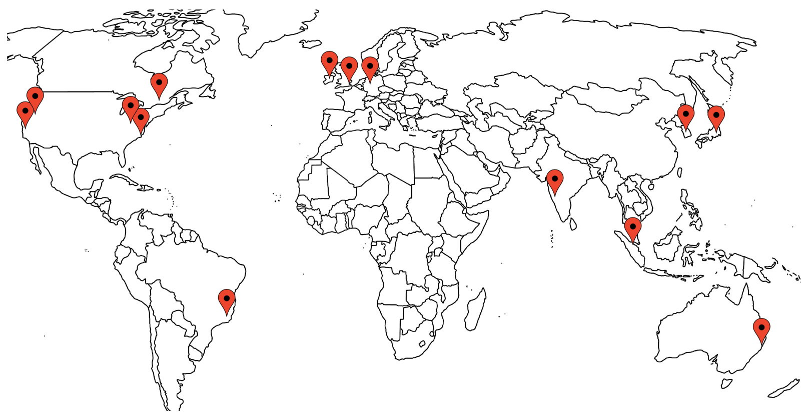


Figure 4.13: Amazon EC2 regions used in the WAN experiment. Details see Table 4.8.

unforgeability proof:

$$\begin{aligned}
\bigoplus_i H_i &= \bigoplus_{i\in\mathcal{M}} H_i \oplus \bigoplus_{i\in\mathcal{H}} H_i \\
&= \bigoplus_{i\in\mathcal{M}} (H_i^* \oplus Q_i) \oplus \bigoplus_{i\in\mathcal{H}} \left(H_i^* \oplus \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right)\right) \\
&= \bigoplus_i H_i^* \oplus \bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{i\in\mathcal{H}} \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right) \\
&= \bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{i\in\mathcal{H}} \left(\bigoplus_{k\neq i} x^k Q_{k,i}\right) \\
&= \bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{k} \left(x^k \bigoplus_{i\in\mathcal{H}, i\neq k} Q_{k,i}\right) \\
&= \bigoplus_{i\in\mathcal{M}} Q_i \oplus \bigoplus_{k} x^k R_k
\end{aligned}$$

□

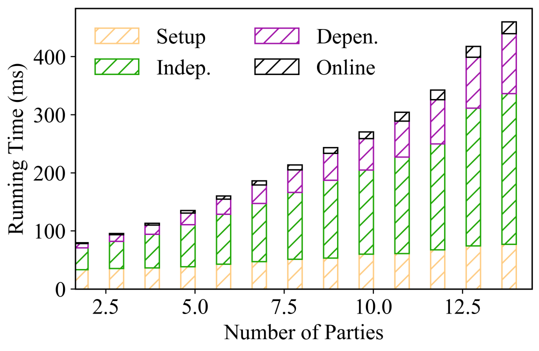
4.6 Evaluation

4.6.1 Implementation Details

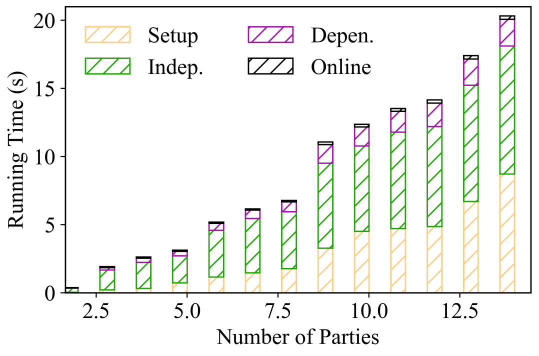
We implemented our protocol in the EMP-toolkit [10] framework and will be made publicly available as a part of it. To fully explore performance characteristics of our

Table 4.3: **Circuits used in our evaluation.**

Circuit	n_1	n_2	n_3	$\lvert\mathcal{C}\rvert$
AES	128	128	128	6800
SHA-128	256	256	160	37300
SHA-256	256	256	256	90825



(a) LAN setting.



(b) WAN setting.

Figure 4.14: Running time breakdown for evaluating AES. In the LAN setting, all parties are located in the same region; In the WAN setting, all parties are located in different regions worldwide, for example, **2PC**: within US-east; **5PC**: within North America; **8PC**: within North America and Europe; **12PC**: within North America, Europe and Asia; **13PC**: further adds Sydney; **14PC**: all parties in Figure 4.13.

protocol, we evaluate our implementation in three different settings:

- **LAN setting.** Machines are located in the same Amazon EC2 region. Experiments are performed for up to 14 parties.

- **WAN setting.** Each Machine is located in a *different* Amazon EC2 region (locations shown in Figure 4.13). For a k-party computation experiment, a prefix subset of the machines in Table 4.8 are selected. For example, parties in 3PC experiments are located in North Virginia, Ohio, and North California respectively. Experiments are run for up to 14 parties, which is the number of different Amazon EC2 regions available.

- **Crowd setting.** In the LAN case, we evaluate up to 128 parties all located in the same Amazon EC2 region. In the global-scale case, we choose 8 different cities across 5 continents and open up to 16 parties in each city (totally 128 parties).

All machines are of type `c4.8xlarge`, with 36 cores and 60 GB RAM. Network bandwidth within the same region is about 10Gbps. The bandwidth across different regions depends on the location of the machines. All experiments are based on $\rho = 40, \kappa = 128$. We extend justGarble [18] to support garbling of longer tables in a straightforward manner. In the implementation, we used the "broadcast with abort" protocol by Goldwasser and Lindell [77] and achieves the notion of secure computation with abort. We observe small variance when running in the LAN setting and slightly higher variance in the WAN setting. All numbers reported in LAN setting are on average of 10 runs, ones in WAN setting are on average of 20 runs, and ones in the Crowd setting are on average of 5 runs due to the lengthy experiments.

4.6.2 Performance on Basic Circuits

We evaluate commonly used benchmark circuits on our protocol, including AES, SHA-1, and SHA-256. Information about these circuits can be found in Table 4.3.

Table 4.4: Detailed experiment results for the crowd setting. Timings are measured in terms of milliseconds. In the LAN setting, all parties are located in the same region. In the Worldwide setting. 8PC is performed with each party located in a different region; 16PC is performed with 2 parties located in each region; others can be interpreted similarly.

	WAN setting				
n	Setup	Indep.	Depen.	Online	Total
8	16736.0	30647.4	5905.2	783.3	54071.9
16	18708.1	21699.6	12243.5	598.8	53250.0
32	35838.8	19038.4	8242.3	716.6	63836.1
64	71913.8	25280.7	29416.6	1564.0	128175.2
128	88055.9	30795.9	22659.1	2316.2	143827.0

	LAN setting				
n	Setup	Indep.	Depen.	Online	Total
8	49.4	122.2	35.7	7.8	215.1
16	78.8	227.8	121.1	29.4	457.0
32	129.9	627.0	446.2	112.3	1315.5
64	212.9	1182.2	2630.1	476.5	4501.7
128	383.0	2727.4	11669.6	1870.2	16650.2

We plot in Figure 4.14 the results for AES in different network setting and performance breakdown as described above. Detailed timings and more results for all three circuits can be found in Table 3.4 in the Appendix. First, the performance of three-party computation is extremely efficient: it takes 95 ms to evaluate a circuit for AES, with 2 ms online time. We also find that in the LAN setting, the slowdown from 2PC to 3PC is roughly 1.5×; the slowdown in the WAN setting is larger. This is caused by the network latency: the first two parties are both located in the U.S. east coast, while the third party is located in the U.S. west coast with much higher latency.

We also find that the cost of the one-time setup is almost independent of number of parties for small number of parties. This is mainly due to the parallelization in the implementation that allows all base-OT to run at the same time.

World-wide MPC experiment. We would like to emphasize that in the case of WAN setting with 14 parties, it is a "world-wide" MPC experiment over 5 continents. To the best of our knowledge, we are the first to conduct MPC over such large range even considering semi-honest MPC protocols.

We also notice a big "jump" in running time from 8 parties to 9 parties in the WAN setting. We believe this is because of the network condition: for experiments up to 8 parties, it is within the US/Europe area; the ninth party is located in asia, where the communication to US/Europe is much slower. More details see Figure 4.14b.

4.6.3 Evaluation in a Global Setting

In this section, we focus on the performance of our protocol with a large number of parties. We summarize our results in Figure 4.4. We notice that our protocol scales very well with increasing number of parties. Even in a setting with 128 parties located in the same LAN, where up to 127 of them can be corrupted, it takes less than 17 seconds end-to-end running time to compute AES. Note that the performance of a 64-party computation on AES is comparable to the performance of what used to be the state-of-the-art malicious *2-party* computation three years ago [6], and we believe further optimizations and improvements based on our work will flourish too.

When comparing the running time of 128 parties to the one of 8 parties, we find that the cost of function-dependent phase increases much faster than the cost of function-independent phase. This is because our function-independent phase is symmetric and all communication loads are evenly distributed among all parties; while in the function-dependent phase, $n-1$ garblers send the garbled circuit to the evaluator, and the bandwidth of the evaluator becomes the bottleneck. Therefore in the case where there are a lot of parties, when we double the number of parties, the running time of the function-dependent phase almost doubles.

We also run the same experiment in the worldwide range. We choose 8 most separate regions out of 14 and open up to 16 machines in each region (thus totally 128 machines). The performance is also shown in Table 4.4: it takes slightly more than a minute for 64 parties to compute AES and about 2.5 minutes for 128 parties located all around the world. We also observe that setup takes much more time; we believe it is due to the high latency.

4.6.4 Comparison to Other Work

Malicious MPC on AES. Evaluating AES with malicious security against $n-1$ corruption was studied by Damgård et al. [78]. They reported 240 ms *online time* for 3 parties and 340 ms online time for 10 parties. The offline time for 3 and 10 parties are around 4200 seconds and 15000 seconds respectively. Our protocol takes 95 ms total time to evaluate AES for 3 parties with online time as small as 2 ms;

and 268 ms total time with online time 12 ms. The improvement for online phase ranges from 28× to 120×; and the improvement for total time ranges from 44000× to 56000×. This is a huge improvement even considering hardware differences.

BMR-style protocols. Lindell et al. [42, 43] studied how to use SPDZ and SHE to construct a BMR-style protocol. Since their protocol is not implemented, we compare the communication complexity. After incorporating various optimizations, every AND gate still need $3n + 1$ SPDZ multiplication triples. Together with the most recent advance in SPDZ triple generation by Keller et al. [49], generating one SPDZ triple with n parties requires communication about $180(n - 1)$ kilobits per party. Therefore the communication cost per AND gate per party is about $540n(n - 1)$ kilobits. In our protocol, each AND gate only needs one AND triple from $\mathcal{F}_{\mathsf{Pre}}$, which, using our new protocol in Section 4.4, requires communication roughly $2.28(n - 1)$ kilobits per party. Therefore, the improvement of our protocol compared to the best-optimized BMR protocol based on Lindell et al. is about $237n\times$ with n parties. For a three-party setting, it is an improvement of 711×; for the 128-party computation that we perform, the improvement is as high as $30,000\times$!

Ben-Efraim et al. [56] presented a protocol secure in the *semi-honest* model based on BMR. Surprisingly, given the fact that our protocol is maliciously secure, while theirs only has a semi-honest security, our implementation has roughly the same performance as theirs. In Table 4.5, we compare the running time of our protocol with the running time of theirs based on the same hardware. We notice that for both online time and total time, the performance of the two protocols are

Table 4.5: Compare with Ben-Efraim et al. [56] Timings are in terms of milliseconds. Their protocol works in the semi-honest setting; ours is maliciously secure. Comparison based on SHA-256, with the same hardware configuration.

		3	8	16	32
[56]	Online	80	150	400	1500
	Total	228	2000	5900	-
This work	Online	24	95	370	1632
	Total	618	1945	6711	18828

roughly the same.

Malicious 3PC protocols with honest majority. Mohassel et al. [62] proposed an efficient protocol for malicious 3PC with honest majority. Their protocol requires only one garbled circuit to be sent and therefore has a smaller communication complexity than us. We estimate that our protocol requires about 14$\times$ more communication than theirs. However interestingly we also find that the online time of two protocols are roughly the same: their protocol requires 31 ms evaluation time, while ours needs 23.4 ms evaluation time. We belive this is due to the fact that their protocol needs to check that the garbled circuit received from two garblers are the same, while it is not needed in our protocol.

Furukawa et al. [63] also presented a malicious 3PC protocol with honest majority. Their protocol has a smaller communication overhead compared to the protocol above by Mohassel et al. but requires at least one round of communication per level of the circuit. In addition to a stronger security guarantee that we support dishonest majority, our protocol has a better latency, especially for deep circuit (e.g.

Table 4.6: Compare bandwidth consumption with Hazay et al. [75]. All numbers are the maximum amount of data one party needs to send in the function-independent phase, measured in terms of megabytes (MB). Numbers for $\rho = 80$ are calculated based on the complexity of both protocols.

	ρ	3	8	16
[75]	40	14	49	105
	80	55	193	413
This work	40	4.8	16.9	36.4
	80	8.6	30	64.5

SHA-256 has a depth of 4000), while their protocol has a better throughput.

Compare with Hazay et al. [75]. We also compare with the concurrent work by Hazay et al.. As their protocol is benchmarked on different hardware and network configurations, we only compare the bandwidth usage. We find that both protocols has similar function-dependent cost and online cost. However, due to our improved preprocessing protocol, our function-independent cost is much smaller than theirs. In Table 4.6, we compare the function-independent cost of for AES evaluation with different value of ρ. Our protocol uses $3\times$ to $6.5\times$ less communication compared to theirs. Note that the cost of function-independent phase dominates the overall cost, therefore the speed up here also translates to the speed up to the whole computation.

4.6.5 Communication Complexity

In this section, we evaluate the communication complexity of our protocol. All numbers reported here are the maximum amount of data sent from one party. All

Table 4.7: Communication complexity of our protocol. Bandwidth are measured for evaluating AES and SHA-256. All numbers are the maximum amount of data one party needs to send.

n	Setup	Indep.	Depen.	Online	Total
			AES		
3	57.1 KB	4.8 MB	1.3 MB	4.5 KB	6.2 MB
4	85.7 KB	7.2 MB	1.8 MB	4.5 KB	9.1 MB
5	114.2 KB	9.7 MB	2.2 MB	4.5 KB	12.0 MB
6	142.8 KB	12.1 MB	2.7 MB	4.5 KB	14.9 MB
7	171.4 KB	14.5 MB	3.1 MB	4.5 KB	17.8 MB
8	199.9 KB	16.9 MB	3.5 MB	4.5 KB	20.7 MB
16	428.4 KB	36.4 MB	7.1 MB	4.5 KB	44.0 MB
			SHA-256		
3	57.1 KB	63.3 MB	17.4 MB	9.0 KB	80.8 MB
4	85.7 KB	95.0 MB	23.4 MB	9.0 KB	118.5 MB
5	114.2 KB	126.6 MB	29.4 MB	9.0 KB	156.2 MB
6	142.8 KB	158.3 MB	35.4 MB	9.0 KB	193.9 MB
7	171.4 KB	190.0 MB	41.4 MB	9.0 KB	231.6 MB
8	199.9 KB	221.7 MB	47.4 MB	9.0 KB	269.3 MB
16	428.4 KB	475.1 MB	95.4 MB	9.0 KB	570.9 MB

numbers are obtained by running our implementation, which are slightly higher than calculated values due to implementation details. In Table 4.7, we summarize the bandwith use for AES and SHA-256 for up to 16 parties.

We can see from the figure that the communication required per party grows linearly with the number of parties. In addition, the communication cost of the Setup phase and the Online phase are very small. The total communication cost is dominated by the function-independent phase.

Table 4.8: List of all Amazon EC2 regions used in the WAN experiment.

Continent	Region	
North America	North Virginia	Ohio
	North California	Oregon
	Toronto	
Europe	Ireland	London
	Frankfurt	
Asia	Mumbai	Tokyo
	Seoul	Singapore
Australia	Sydney	
South America	São Paulo	

Chapter 5: Improved Authenticated Garbling in the Two-Party Setting

In this chapter, we will discuss more optimizations that further reduces the communication and computation complexity of authenticated garbling technique in the two-party. In particular, We show several improvements to the protocol described in Chapter 3 that, overall, improve its performance by a factor of 2–3×:

- We show how to make the "authenticated garbling" compatible with the half-gate optimization of Zahur et al. We also show that it is possible to avoid sending an information-theoretic MAC for each garbled row. These two optimizations result in a 2.6× improvement in communication and, somewhat surprisingly, result in a protocol for malicious secure two-party computation in which the communication complexity of the online phase is essentially equivalent to that of state-of-the-art *semi-honest* secure computation.

- The function-dependent phase of the WRK protocol involves the computation of (shared) "AND triples" between the parties. We show various optimizations of that step that result in a 1.5× improvement in the communication and a 2× improvement in the computation. Our optimizations also simplify the protocol significantly.

We can combine these improvements in various ways, and suggest in particular two instantiations: one protocol that minimizes the total cost across all phases, and one that trades off increased communication in the function-independent preprocessing step for reduced communication in the function-dependent preprocessing step. These protocols improve upon the state-of-the-art by a significant margin.

Outline In Section 5.1 we provide some background about the WRK protocol. We provide the high-level intuition behind our improvements in Section 5.2. In Section 5.3, we describe in detail our optimizations of the online phase of the WRK protocol, and in Section 5.4 we discuss our optimizations of the preprocessing phase. In Section 5.5, we compare our resulting protocols to prior work.

5.1 Background

We begin by describing some general background, followed by an in-depth review of the authenticated garbling technique. In the section that follows, we give a high-level overview of our optimizations and improvements.

Opening authenticated values. An authenticated bit $[b]_{\mathsf{A}}$ known to P_{A} can be opened by having P_{A} send b and $\mathsf{M}[b]$ to P_{B}, who then verifies that $\mathsf{M}[b] = \mathsf{K}[b] \oplus b\Delta_{\mathsf{B}}$. As observed in prior work [44], it is possible to open n authenticated bits with less than n times the communication. Specifically, P_{A} can open $[b_1]_{\mathsf{A}}, \ldots, [b_n]_{\mathsf{A}}$ by sending $b_1, \ldots, b_n$ along with $h := H(\mathsf{M}[b_1], \ldots, \mathsf{M}[b_n])$, where H is a hash function modeled as a random oracle. P_{A} then simply checks whether $h = H(\mathsf{K}[b_1] \oplus b_1\Delta_{\mathsf{B}}, \ldots, \mathsf{K}[b_n] \oplus b_n\Delta_{\mathsf{B}})$.

We let $\mathsf{Open}([b_1]_\mathsf{A}, \ldots)$ denote the process of opening one or more authenticated bits, and overload this notation so that $\mathsf{Open}(\langle b_1 \mid b_2 \rangle)$ denotes the process of having each party open its portion of an authenticated share.

Circuit-dependent preprocessing. We consider boolean circuits with gates represented as a tuple $(\alpha, \beta, \gamma, T)$, where α and β are (the indices of) the input wires of the gate, γ is the output wire of the gate, and $T \in \{\oplus, \wedge\}$ is the type of the gate. We use $\mathcal{W}$ to denote the output wires of all AND gates, $\mathcal{I}_1, \mathcal{I}_2$ to denote the input wires for each party, and $\mathcal{O}$ to denote the output wires.

Wang et al. [7] introduced an ideal functionality called $\mathcal{F}_\mathsf{pre}$ (cf. Figure 5.1) that is used by the parties in a circuit-dependent, but input-*independent*, preprocessing phase. This functionality sets up information for the parties as follows:

1. For each input wire of the circuit as well as output wire of an AND gate, namely w, generate a random authenticated share $\langle r_w \mid s_w \rangle$. We refer to the value $\lambda_w \stackrel{\text{def}}{=} r_w \oplus s_w$ as the *mask* on wire w.

2. For the output wire γ of each XOR gate $(\alpha, \beta, \gamma, \oplus)$, generate a random authenticated share $\langle r_\gamma \mid s_\gamma \rangle$ whose value $r_\gamma \oplus s_\gamma$ is the XOR of the masks on the input wires α, β.

3. For each AND gate $(\alpha, \beta, \gamma, \wedge)$, generate a random authenticated share $\langle r^*_\gamma \mid s^*_\gamma \rangle$ such that
$$r^*_\gamma \oplus s^*_\gamma = (r_\alpha \oplus s_\alpha) \wedge (r_\beta \wedge s_\beta).$$

We refer to a triple of authenticated shares $(\langle r_\alpha \mid s_\alpha \rangle, \langle r_\beta \mid s_\beta \rangle, \langle r^*_\gamma \mid s^*_\gamma \rangle)$ for which

Functionality $\mathcal{F}_{\mathsf{pre}}$

1. Choose uniform $\Delta_{\mathsf{A}}, \Delta_{\mathsf{B}} \in \{0,1\}^{\rho}$. Send Δ_{A} to P_{A} and Δ_{B} to P_{B}.
2. For each wire $w \in \mathcal{W} \cup \mathcal{I}$, generate a random authenticated share $\langle r_w \,|\, s_w \rangle$.
3. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, T)$, in topological order:
   - If $T = \oplus$, generate a random authenticated share $\langle r_\gamma \,|\, s_\gamma \rangle$ for which $r_\gamma \oplus s_\gamma = r_\alpha \oplus s_\alpha \oplus r_\beta \oplus s_\beta$.
   - If $T = \wedge$, generate a random authenticated share $\langle r^*_\gamma \,|\, s^*_\gamma \rangle$ for which $r^*_\gamma \oplus s^*_\gamma = (r_\alpha \oplus s_\alpha) \wedge (r_\beta \oplus s_\beta)$.

Figure 5.1: Preprocessing functionality for some fixed circuit.

$r^*_\gamma \oplus s^*_\gamma = (r_\alpha \oplus s_\alpha) \wedge (r_\beta \oplus s_\beta)$ as an *authenticated AND triple*. These are just (authenticated) Beaver triples [69] over the field $\mathbb{F}_2$.

From authenticated shares to shared labels. One important optimization in the WRK protocol is to compute shares of labels efficiently using authenticated shares. Assume the parties hold an authenticated share $\langle r \,|\, s \rangle$ of some mask value $\lambda = s \oplus r$. It is then easy to compute a share of $\lambda\Delta_{\mathsf{A}}$, since

$$\lambda\Delta_{\mathsf{A}} = (r \oplus s)\Delta_{\mathsf{A}} = \Big(r\Delta_{\mathsf{A}} \oplus \mathsf{K}[s]\Big) \oplus \Big(\mathsf{M}[s]\Big).$$

Since P_{A} has r, Δ_{A}, and $\mathsf{K}[s]$ while P_{B} has $\mathsf{M}[s]$, two parties can locally compute share of $\lambda\Delta_{\mathsf{A}}$, namely $[\lambda\Delta_{\mathsf{A}}]$ given only $\langle r \,|\, s \rangle$.

Now we can use this important fact to compute shares of labels for secret masked bit efficiently. Assuming that the global authentication key (Δ_{A}) is also used as the free-XOR Delta, then we know that $\mathsf{L}_{\gamma,\hat{z}_{u,v}} = \mathsf{L}_{\gamma,0} \oplus \hat{z}_{u,v}\Delta_{\mathsf{A}}$. Therefore, the task of computing share of labels reduces to the task of computing share of

$\hat{z}_{u,v}\Delta_{\mathsf{A}}$, since $\mathsf{L}_{\gamma,0}$ is known to P_{A}. Notice that

$$\begin{aligned}\hat{z}_{u,v}\Delta_{\mathsf{A}} &= ((\lambda_\alpha \oplus u) \wedge (\lambda_\beta \oplus v) \oplus \lambda_\gamma)\,\Delta_{\mathsf{A}} \\ &= \lambda_\alpha\lambda_\beta\Delta_{\mathsf{A}} \oplus u\lambda_\alpha\Delta_{\mathsf{A}} \oplus v\lambda_\beta\Delta_{\mathsf{A}} \oplus uv\Delta_{\mathsf{A}} \oplus \lambda_\gamma\Delta_{\mathsf{A}}\end{aligned}$$

Assume the parties hold an authenticated AND triple $(\langle r_\alpha \,|\, s_\alpha\rangle, \langle r_\beta \,|\, s_\beta\rangle, \langle r^*_\gamma \,|\, s^*_\gamma\rangle)$, and a random authenticated share $\langle r_\gamma \,|\, s_\gamma\rangle$, such that $\lambda_\alpha = r_\alpha \oplus s_\alpha$, $\lambda_\beta = r_\beta \oplus s_\beta$, $\lambda_\alpha \wedge \lambda_\beta = r^*_\gamma \oplus s^*_\gamma$, $\lambda_\gamma = r_\gamma \oplus s_\gamma$. Now, two parties can locally computes shares of $\lambda_\alpha\Delta_{\mathsf{A}}$, $\lambda_\beta\Delta_{\mathsf{A}}$, $\lambda_\gamma\Delta_{\mathsf{A}}$, and $(\lambda_\alpha \wedge \lambda_\beta)\Delta_{\mathsf{A}}$, and compute share of $\hat{z}_{u,v}\Delta_{\mathsf{A}}$ by linearly combine the above shares.

5.2 Overview of Our Optimizations

We separately discuss our optimizations for the authenticated garbling and the preprocessing phases. Details and proofs can be found in Sections 5.3 and 5.4.

5.2.1 Improving Authenticated Garbling

As a high level, the key ideas behind authenticated garbling are that (1) it is possible to share garbled circuits such that neither party knows how rows in the garbled tables are permuted (since no party knows the masks on the wires); moreover, (2) information-theoretic MACs can be used to ensure correctness of the garbled tables. In the original protocol by Wang et al., these two aspects are tightly integrated: each garbled row includes an encryption of the corresponding MAC tag, so the evaluator only learns one such tag for each gate.

Here, we take a slightly different perspective on how authenticated garbling works. In particular, we (conceptually) divide the protocol into two parts:

- In the first part, the parties compute a shared garbled circuit, without any authentication, and let the evaluator reconstruct and evaluate that garbled circuit. We stress here that, even though there is no authentication, corrupting one or more garbled rows does not allow a selective-failure attack for the same reason as in the WRK protocol: any failure depends only on the *masked* wire values, but neither party knows those masks.

  This part is achieved by the encrypted wire labels alone, which have the form $H(\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) \oplus [\mathsf{L}_{\gamma,\hat{z}_{u,v}}]$. These require 4κ bits of communication per gate.

- In the second part, the evaluator holds masked wire values for every wire of the circuit. It then checks correctness of all these masked values. For example, it will ensure that for every AND gate, the underlying (real) values on the wires form an AND relationship. Such verification is needed for masked values that $\mathsf{P_B}$ obtains during the evaluation of the garbled circuit.

  The WRK protocol achieves this by encrypting *authenticated shares* of the form $H(\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) \oplus (r_{u,v}, \mathsf{M}[r_{u,v}])$ in each row of a garbled table. The evaluator decrypts one of the rows and checks the appropriate tag. These encrypted tags contribute 4ρ bits of communication per gate.

With this new way of viewing authenticated garbling, we can optimize each part independently. By doing so, we are able to reduce the communication of the first part to $2\kappa + 1$ bits per gate, and reduce the communication of the second part to 1 bit per gate. In the process, we also reduce the computation (in terms of hash evaluations) by about half. In the following, we discuss intuitively how these optimizations work.

Applying row-reduction techniques. In garbled circuits, *row reduction* refers to techniques that use fewer than four garbled rows per garbled gate [79, 19, 14, 80]. We review the simplest row-reduction technique here, describe the challenge of applying the technique to authenticated garbling, and then show how we overcome the challenge. This will serve as a warm-up to our final protocol that is compatible with the half-gate technique.

In classical garbling, a garbled AND gate can be written as (in our notation):

$$\begin{aligned}
G_{0,0} &= H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}) \oplus \mathsf{L}_{\gamma,\hat{z}_{0,0}} = H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}) \oplus \mathsf{L}_{\gamma,0} \oplus \hat{z}_{0,0}\Delta_{\mathsf{A}} \\
G_{0,1} &= H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}) \oplus \mathsf{L}_{\gamma,\hat{z}_{0,1}} = H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,1}) \oplus \mathsf{L}_{\gamma,0} \oplus \hat{z}_{0,1}\Delta_{\mathsf{A}} \\
G_{1,0} &= H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}) \oplus \mathsf{L}_{\gamma,\hat{z}_{1,0}} = H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,0}) \oplus \mathsf{L}_{\gamma,0} \oplus \hat{z}_{1,0}\Delta_{\mathsf{A}} \\
G_{1,1} &= H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}) \oplus \mathsf{L}_{\gamma,\hat{z}_{1,1}} = H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}) \oplus \mathsf{L}_{\gamma,0} \oplus \hat{z}_{1,1}\Delta_{\mathsf{A}}.
\end{aligned}$$

The idea behind GRR3 row reduction [79] is to choose wire labels so $G_{0,0} = 0^\kappa$. That is, the garbler chooses

$$\mathsf{L}_{\gamma,0} := H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}) \oplus \hat{z}_{0,0}\Delta_{\mathsf{A}}.$$

The garbler now needs to send only $(G_{0,1}, G_{1,0}, G_{1,1})$, reducing the communication from 4κ to 3κ bits. If the evaluator has input wires with masked values $(0, 0)$, it can simply set $G_{0,0} = 0^\kappa$ and then proceed as before.

In authenticated garbling, the preprocessing results in shares of $\{\hat{z}_{u,v}\Delta_\mathsf{A}\}$. Hence, if $\mathsf{P_A}$ could compute $\mathsf{L}_{\gamma,0}$ then the parties could locally compute shares of the $\{G_{u,v}\}$ (since $\mathsf{P_A}$ knows all the $\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}$ values and their hashes). $\mathsf{P_A}$ could then send its shares to $\mathsf{P_B}$ to allow $\mathsf{P_B}$ to recover the entire garbled gate. Unfortunately, $\mathsf{P_A}$ cannot compute $\mathsf{L}_{\gamma,0}$ because $\mathsf{P_A}$ does not know $\hat{z}_{0,0}$! Indeed, that value depends on the secret wire masks, unknown to either party.

Summarizing, row-reduction techniques in general compute one (or both) of the output-wire labels as a function of the input-wire labels **and** the secret masks, making them a challenge for authenticated garbling.

Our observation is that although $\mathsf{P_A}$ does not know $\hat{z}_{0,0}$, the garbling requires only $\hat{z}_{0,0}\Delta_\mathsf{A}$ for which the parties do have shares. Let S_A and S_B denote the parties' shares of this value, so that $S_A \oplus S_B = \hat{z}_{0,0}\Delta_\mathsf{A}$. Our main idea is for the parties to "shift" the entire garbling process by the value S_B, as follows:

1. $\mathsf{P_A}$ computes $\mathsf{L}_{\gamma,0} := H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}) \oplus S_A$. Note this value differs from the standard garbling value by a shift of S_B. Intuitively, instead of choosing $\mathsf{L}_{\gamma,0}$ so that $G_{0,0} = 0^\kappa$, we set implicitly set $G_{0,0} = S_B$. Although $\mathsf{P_A}$ does not know S_B, it only matters that the evaluator $\mathsf{P_B}$ knows it.

2. Based on this value of $\mathsf{L}_{\gamma,0}$, the parties locally compute shares of the garbled gate $G_{0,1}, G_{1,0}, G_{1,1}$ defined above, and open them to $\mathsf{P_B}$.

3. When $\mathsf{P_B}$ evaluates the gate on input $\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}$, if $(u, v) \neq (0, 0)$ then evaluation is the same as usual. If $(u, v) = (0, 0)$ then $\mathsf{P_B}$ sets $G_{0,0} = S_B$. This is equivalent to $\mathsf{P_B}$ doing the usual evaluation but shifting the result by S_B.

Using the half-gate technique. The state-of-the-art in semi-honest garbling is the half-gate construction of Zahur et al. [14]. It requires 2κ bits of communication per AND gate, while being compatible with free-XOR. We describe this idea, translated from the original work [14] to be written in terms of masks and masked wire values so as to match our notation.

The circuit garbler computes a garbled gate as:

$$G_0 := H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \oplus \lambda_\beta \Delta_{\mathsf{A}}$$

$$G_1 := H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\beta,1}) \oplus \mathsf{L}_{\alpha,0} \oplus \lambda_\alpha \Delta_{\mathsf{A}},$$

and computes the 0-label for that gate's output wire as:

$$\mathsf{L}_{\gamma,0} := H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus (\lambda_\alpha\lambda_\beta \oplus \lambda_\gamma)\Delta_{\mathsf{A}}.$$

If the evaluator P_{B} holds masked values u, v and corresponding labels $\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}$, it computes:

$$\mathsf{Eval}(u, v, \mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) := H(\mathsf{L}_{\alpha,u}) \oplus H(\mathsf{L}_{\beta,v}) \oplus uG_0 \oplus v(G_1 \oplus \mathsf{L}_{\alpha,u}).$$

This results in the value

$$\begin{aligned}\mathsf{Eval}(u, v, \mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) &= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus (uv \oplus v\lambda_\alpha \oplus u\lambda_\beta)\Delta_{\mathsf{A}} \\ &= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus \Big((u \oplus \lambda_\alpha)(v \oplus \lambda_\beta) \oplus \lambda_\alpha\lambda_\beta\Big)\Delta_{\mathsf{A}} \\ &= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus (\hat{z}_{u,v} \oplus \lambda_\alpha\lambda_\beta \oplus \lambda_\gamma)\Delta_{\mathsf{A}},\end{aligned}$$

which is the correct output $\mathsf{L}_{\gamma,\hat{z}_{u,v}} = \mathsf{L}_{\gamma,0} \oplus \hat{z}_{u,v}\Delta_{\mathsf{A}}$.

As before, this garbling technique is problematic for authenticated garbling, because the garbler P_{A} cannot compute $\mathsf{L}_{\gamma,0}$ as specified. (P_{A} does not know the wire masks, so cannot compute the term $(\lambda_\alpha\lambda_\beta \oplus \lambda_\gamma)\Delta_{\mathsf{A}}$.)

However, the parties hold[1] shares of this value; say, $S_A \oplus S_B = (\lambda_\alpha \lambda_\beta \oplus \lambda_\gamma)\mathsf{\Delta_A}$. We can thus conceptually "shift" the entire garbling procedure by S_B to obtain the following interactive variant of half-gates:

1. $\mathsf{P_A}$ computes the output wire label as

$$\mathsf{L}_{\gamma,0} := H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus S_A,$$

   which is "shifted" by S_B from what the half-gates technique specifies.

2. The parties locally compute shares of G_0, G_1 as per the half-gates technique described above. These shares are opened to $\mathsf{P_B}$, so $\mathsf{P_B}$ learns (G_0, G_1).

3. To evaluate the gate on inputs $\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}$, the evaluator $\mathsf{P_B}$ performs standard half-gates evaluation and then adds S_B as a correction value. This results in the correct output-wire label, since:

$$\begin{aligned}\mathsf{Eval}(\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) \oplus S_B &= \mathsf{Eval}(\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) \oplus (\lambda_\alpha \lambda_\beta \oplus \lambda_\gamma)\mathsf{\Delta_A} \oplus S_A \\ &= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus \hat{z}_{u,v}\mathsf{\Delta_A} \oplus S_A \\ &= \mathsf{L}_{\gamma,0} \oplus \hat{z}_{u,v}\mathsf{\Delta_A} \\ &= \mathsf{L}_{\gamma,\hat{z}_{u,v}}.\end{aligned}$$

Authentication almost for free. In the WRK scheme, suppose the actual values on the wires of an AND gate are $z_\alpha, z_\beta, z_\gamma$ with $z_\alpha \wedge z_\beta = z_\gamma$. During evaluation, $\mathsf{P_B}$ learn masked values $\hat{z}_\alpha = z_\alpha \oplus \lambda_\alpha$, $\hat{z}_\beta = z_\beta \oplus \lambda_\beta$, and $\hat{z}_\gamma = z_\gamma \oplus \lambda_\gamma$. For correctness

[1]Note that $(\lambda_\alpha \lambda_\beta \oplus \lambda_\gamma) = \hat{z}_{0,0}$, the same secret value as in the previous example.

it suffices to show that

$$\begin{aligned} z_\alpha \wedge z_\beta = z_\gamma &\iff (\hat{z}_\alpha \oplus \lambda_\alpha) \wedge (\hat{z}_\beta \oplus \lambda_\beta) = (\hat{z}_\gamma \oplus \lambda_\gamma) \\ &\iff \underbrace{(\hat{z}_\alpha \oplus \lambda_\alpha) \wedge (\hat{z}_\beta \oplus \lambda_\beta) \oplus \lambda_\gamma}_{\hat{z}_{\alpha,\beta}} = \hat{z}_\gamma. \end{aligned}$$

Note the parties already have authenticated shares of λ_α, λ_β, λ_γ, and $(\lambda_\alpha \wedge \lambda_\beta)$, so they can also derive authenticated shares of related values.

In the WRK scheme the garbler $\mathsf{P_A}$ prepares an authenticated share (MAC) of $\hat{z}_{\alpha,\beta}$ corresponding to each of the 4 possible values of $\hat{z}_\alpha$, $\hat{z}_\beta$. It encrypts this share so that it can only be opened using the corresponding wire labels. $\mathsf{P_B}$ can then decrypt and verify the relevant $\hat{z}_{\alpha,\beta}$ value (and take it to be the masked output value $\hat{z}_\gamma$).

Our approach is to apply a technique suggested for the SPDZ protocol [44]: evaluate the circuit without authentication and then perform batch authentication at the end. Thus, in our new protocol authentication works as follows:

1. $\mathsf{P_B}$ evaluates the circuit, obtaining (unauthenticated) masked values $\hat{z}_\alpha$ for every wire α.

2. $\mathsf{P_B}$ reveals the masked values of every wire (1 bit per wire). Revealing these to $\mathsf{P_A}$ does not affect privacy because the masks are hidden from both parties (except for certain input/output wires where one or both of the parties already know the underlying values).

3. $\mathsf{P_A}$ generates authenticated shares of only the relevant $\hat{z}_{\alpha,\beta}$ values and sends them. $\mathsf{P_B}$ verifies the authenticity of each share. This is equivalent to sending a MAC of $\mathsf{P_A}$'s shares. As described in Section 5.1, this can be done by sending

only a hash of the MACs.

This technique for authentication adds an extra round, but it makes the authentication almost free in terms of communication. $\mathsf{P_B}$ sends 1 bit per wire and $\mathsf{P_A}$ sends only a single hash value to authenticate.

Details of the optimizations described above can be found in Section 5.3.

5.2.2 Improving the Preprocessing Phase

We also improve the efficiency of preprocessing in the WRK protocol significantly; specifically: (1) we design a new protocol for generating so-called leaky-AND triples. Compared to the best previous protocol by Wang et al., it reduces the number of hash calls by 2.5× and reduces communication by κ bits. (2) we propose a new function-dependent preprocessing protocol that can be computed much more efficiently. We remark that the second optimization is particularly suitable for RAM-model secure computation, where CPU circuits are fixed ahead of time.

To enable the above optimizations, we need $\mathsf{lsb}(\Delta_\mathsf{A}) = 1$ and $\mathsf{lsb}(\Delta_\mathsf{B}) = 0$, where $\mathsf{lsb}(x)$ denotes the least significant bit of x.

A new leaky-AND protocol. The output of a leaky-AND protocol is a random authenticated AND triple $(\langle r_\alpha \,|\, s_\alpha \rangle, \langle r_\beta \,|\, s_\beta \rangle, \langle r^*_\gamma \,|\, s^*_\gamma \rangle)$ with one caveat: the adversary can choose to guess the value of $r_\alpha \oplus s_\alpha$. A correct guess remains undetected while an incorrect guess will be caught. (See Figure 5.3 for a formal definition.) The leaky-AND protocol by Wang et al. works in two steps. Two parties first run a protocol whose outputs are triples that are leaky without any correctness guarantee;

then a checking procedure is run to ensure correctness. The leakage is later eliminated by bucketing. In our new protocol, we observe that these two steps can be computed at the same time, reducing the number of rounds as well as the amount of computation (i.e., H-evaluations). Moreover, computing and checking can be further improved by adopting ideas from the half-gate technique. Details are below.

Recall that in the half-gate approach, if a wire is associated with wire labels $(\mathsf{L}_0, \mathsf{L}_1 = \mathsf{L}_0 \oplus \Delta_\mathsf{A})$, the first row of the gate computed by the garbler has the form

$$G = H(\mathsf{L}_0) \oplus H(\mathsf{L}_1) \oplus C,$$

for some C. An evaluator holding (b, L_b) can evaluate it as

$$\begin{aligned} E &= bG \oplus H(\mathsf{L}_b) \\ &= b(H(\mathsf{L}_0) \oplus H(\mathsf{L}_1) \oplus C) \oplus H(\mathsf{L}_b) \\ &= b(H(\mathsf{L}_0) \oplus H(\mathsf{L}_1)) \oplus H(\mathsf{L}_b) \oplus bC \\ &= H(\mathsf{L}_0) \oplus bC. \end{aligned} \tag{5.1}$$

Correctness ensures that $E \oplus H(\mathsf{L}_0) = bC$, which means that after the evaluation the two parties hold shares of bC. Note that when free-XOR is used with shift Δ_A, then a pair of garbled labels $(\mathsf{L}_0, \mathsf{L}_1)$ and the IT-MAC for a bit (i.e., $(\mathsf{K}[b], \mathsf{M}[b])$) have the same structure. Therefore the above can be reformulated and extended as follows:

$$G = H(\mathsf{K}[b]) \oplus H(\mathsf{M}[b]) \oplus C_1$$

$$E = bG \oplus H(\mathsf{M}[b]) \oplus bC_2$$

. Assuming the two parties have an authenticated bit $[b]_\mathsf{B}$, then $E \oplus H(\mathsf{K}[b]) = b(C_1 \oplus C_2)$. If we view C_1 and C_2 as shares of some value $C = C_1 \oplus C_2$, then this can be interpreted as a way to select on a shared value such that the selection bit b is known only to one party and at the same time the output (namely, $bC = H(\mathsf{K}[b]) \oplus E$) is still shared.

Now we are ready to present our protocol. We will start with a set of random authenticated bits $(\langle x_1 \,|\, x_2 \rangle, \langle y_1 \,|\, y_2 \rangle, \langle z_1 \,|\, r \rangle)$. We want the two parties to directly compute shares of

$$S = ((x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \oplus z_1 \oplus r)\,(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}).$$

Assuming $\mathsf{lsb}(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) = 1$, revealing $d = \mathsf{lsb}(S)$ allows the parties to "fix" these random authenticated shares to a valid triple (by computing $[z_2]_{\mathsf{B}} = [r]_{\mathsf{B}} \oplus d$). Once the parties hold shares of S (for example, $\mathsf{P_A}$ holds S_1 and $\mathsf{P_B}$ holds $S_2 = S \oplus S_1$), checking the correctness of d also becomes easy: d is valid if and only if $S_1 \oplus d\Delta_{\mathsf{A}}$ from $\mathsf{P_A}$ equals to $S_2 \oplus d\Delta_{\mathsf{B}}$ from $\mathsf{P_B}$. A wrong d can pass the equality check only if the adversary guesses the other party's Δ value. Now the task is to compute shares of S, where S can be rewritten as

$$S = x_1(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) \oplus x_2(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) \oplus (z_1 \oplus r)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}).$$

Here, we will focus on how to compute shares of

$$x_2(y_1\Delta_{\mathsf{A}} \oplus y_1\Delta_{\mathsf{B}} \oplus y_2\Delta_{\mathsf{A}} \oplus y_2\Delta_{\mathsf{B}}).$$

Now we apply the half-gate observation: $\mathsf{P_A}$ has $C_1 = y_1\Delta_{\mathsf{A}} \oplus \mathsf{K}[y_2] \oplus \mathsf{M}[y_1]$ and $\mathsf{P_B}$ has $C_2 = y_2\Delta_{\mathsf{B}} \oplus \mathsf{K}[y_1] \oplus \mathsf{M}[y_2]$, and we have

$$x_2(C_1 \oplus C_2) = x_2(y_1\Delta_{\mathsf{A}} \oplus y_1\Delta_{\mathsf{B}} \oplus y_2\Delta_{\mathsf{A}} \oplus y_1\Delta_{\mathsf{B}}).$$

Therefore, this value can be computed by $\mathsf{P_A}$ sending one ciphertext to $\mathsf{P_B}$. Given the above observations, the final protocol can be derived in a straightforward way. Overall this new approach improves communication by $1.2\times$ and improves computation by $2\times$.

For details and a security proof corresponding to the above, see Section 5.4.1.

New function-dependent preprocessing. Here we show how to further improve the efficiency of function-dependent preprocessing. Recall that in the WRK protocol, each AND triple is derived from B leaky-AND triples, for $B \approx \frac{\rho}{\log C}$; these triples are then used to multiply authenticated masked values for each AND gate of the circuit. Our observation is that we can reduce the number of authenticated shares needed per gate from $3B + 2$ to $3B - 1$. This idea was initially used by Araki et al. [81] in the setting of honest-majority three-party computation. See Section 5.4.2 for details.

5.3 Technical Details: Improved Authenticated Garbling

Since we already discussed the main intuition of the protocol in the previous section, we will present our main protocol in the $\mathcal{F}_{\mathsf{pre}}$-hybrid model. Detailed protocol description is shown in Figure 5.2. Each step in the protocol can be summarized as follows:

1. Parties generate circuit preprocessing information using $\mathcal{F}_{\mathsf{pre}}$.

2. $\mathsf{P_A}$ computes its own share of the garbled circuit and sends to $\mathsf{P_B}$.

Protocol Π_{2pc}

Inputs: $\mathsf{P_A}$ holds $x \in \{0,1\}^{\mathcal{I}_1}$ and $\mathsf{P_A}$ holds $y \in \{0,1\}^{\mathcal{I}_2}$. Parties agree on a circuit for a function $f : \{0,1\}^{\mathcal{I}_1} \times \{0,1\}^{\mathcal{I}_2} \to \{0,1\}^{\mathcal{O}}$.

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ call $\mathcal{F}_{\mathsf{pre}}$, which sends Δ_{A} to $\mathsf{P_A}$, Δ_{B} to $\mathsf{P_B}$, and sends $\{\langle r_w \,|\, s_w\rangle\}_{w\in\mathcal{I}\cup\mathcal{W}}$, $\{\langle r^*_w \,|\, s^*_w\rangle\}_{w\in\mathcal{W}}$ to $\mathsf{P_A}$ and $\mathsf{P_B}$. For each $w \in \mathcal{I}_1 \cup \mathcal{I}_2$, $\mathsf{P_A}$ also picks a uniform κ-bit string $\mathsf{L}_{w,0}$.
2. Following the topological order of the circuit, for each gate $\mathcal{G} = (\alpha, \beta, \gamma, T)$,
   - If $T = \oplus$, $\mathsf{P_A}$ computes $\mathsf{L}_{\gamma,0} := \mathsf{L}_{\alpha,0} \oplus \mathsf{L}_{\beta,0}$
   - If $T = \wedge$, $\mathsf{P_A}$ computes $\mathsf{L}_{\alpha,1} := \mathsf{L}_{\alpha,0} \oplus \Delta_{\mathsf{A}}$, $\mathsf{L}_{\beta,1} := \mathsf{L}_{\beta,0} \oplus \Delta_{\mathsf{A}}$, and
     $G_{\gamma,0} := H(\mathsf{L}_{\alpha,0}, \gamma) \oplus H(\mathsf{L}_{\alpha,1}, \gamma) \oplus \mathsf{K}[s_\beta] \oplus r_\beta \Delta_{\mathsf{A}}$
     $G_{\gamma,1} := H(\mathsf{L}_{\beta,0}, \gamma) \oplus H(\mathsf{L}_{\beta,1}, \gamma) \oplus \mathsf{K}[s_\alpha] \oplus r_\alpha \Delta_{\mathsf{A}} \oplus \mathsf{L}_{\alpha,0}$
     $\mathsf{L}_{\gamma,0} := H(\mathsf{L}_{\alpha,0}, \gamma) \oplus H(\mathsf{L}_{\beta,0}, \gamma) \oplus \mathsf{K}[s_\gamma] \oplus r_\gamma \Delta_{\mathsf{A}} \oplus \mathsf{K}[s^*_\gamma] \oplus r^*_\gamma \Delta_{\mathsf{A}}$
     $b_\gamma := \mathsf{lsb}(\mathsf{L}_{\gamma,0})$

     $\mathsf{P_A}$ sends $G_{\gamma,0}, G_{\gamma,1}, b_\gamma$ to $\mathsf{P_B}$.
3. For each $w \in \mathcal{I}_2$, two parties compute $r_w := \mathsf{Open}([r_w]_{\mathsf{A}})$. $\mathsf{P_B}$ then sends $y_w \oplus \lambda_w := y_w \oplus s_w \oplus r_w$ to $\mathsf{P_A}$. Finally, $\mathsf{P_A}$ sends $\mathsf{L}_{w,y_w\oplus\lambda_w}$ to $\mathsf{P_B}$.
4. For each $w \in \mathcal{I}_1$, two parties compute $s_w := \mathsf{Open}([s_w]_{\mathsf{B}})$. $\mathsf{P_A}$ then sends $x_w \oplus \lambda_w := x_w \oplus s_w \oplus r_w$ and $\mathsf{L}_{w,x_w\oplus\lambda_w}$ to $\mathsf{P_B}$.
5. $\mathsf{P_B}$ evaluates the circuit in topological order. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, T)$, $\mathsf{P_B}$ initially holds $(z_\alpha \oplus \lambda_\alpha, \mathsf{L}_{\alpha,z_\alpha\oplus\lambda_\alpha})$ and $(z_\beta \oplus \lambda_\beta, \mathsf{L}_{\beta,z_\beta\oplus\lambda_\beta})$, where z_α, z_β are the underlying values of the wires.
   (a) If $T = \oplus$, $\mathsf{P_B}$ computes $z_\gamma \oplus \lambda_\gamma := (z_\alpha \oplus \lambda_\alpha) \oplus (z_\beta \oplus \lambda_\beta)$ and $\mathsf{L}_{\gamma,z_\gamma\oplus\lambda_\gamma} := \mathsf{L}_{\alpha,z_\alpha\oplus\lambda_\alpha} \oplus \mathsf{L}_{\beta,z_\beta\oplus\lambda_\beta}$.
   (b) If $T = \wedge$, $\mathsf{P_B}$ computes $G_0 := G_{\gamma,0} \oplus \mathsf{M}[s_\beta]$, and $G_1 := G_{\gamma,1} \oplus \mathsf{M}[s_\alpha]$. $\mathsf{P_B}$ evaluates the garbled table (G_0, G_1) to obtain the output label
   $$\begin{aligned}\mathsf{L}_{\gamma,z_\gamma\oplus\lambda_\gamma} := {} & H(\mathsf{L}_{\alpha,z_\alpha\oplus\lambda_\alpha}, \gamma) \oplus H(\mathsf{L}_{\beta,z_\beta\oplus\lambda_\beta}, \gamma) \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s^*_\gamma] \\ & \oplus (z_\alpha \oplus \lambda_\alpha) G_0 \oplus (z_\beta \oplus \lambda_\beta)(G_1 \oplus \mathsf{L}_{\alpha,z_\alpha\oplus\lambda_\alpha})\end{aligned}$$
   and $z_\gamma \oplus \lambda_\gamma := b_\gamma \oplus \mathsf{lsb}(\mathsf{L}_{\gamma,z_\gamma\oplus\lambda_\gamma})$
6. For each $w \in \mathcal{W}$, $\mathsf{P_B}$ sends $\hat{z}_w := z_w \oplus \lambda_w$ to $\mathsf{P_A}$.
7. For each AND gates $(\alpha, \beta, \gamma, \wedge)$, both parties know $\hat{z}_\alpha = z_\alpha \oplus \lambda_\alpha$, $\hat{z}_\beta = z_\beta \oplus \lambda_\beta$, and $\hat{z}_\gamma = z_\gamma \oplus \lambda_\gamma$. Two parties compute authenticated share of bit c_γ defined as
   $$c_\gamma = (\hat{z}_\alpha \oplus \lambda_\alpha) \wedge (\hat{z}_\beta \oplus \lambda_\beta) \oplus (\hat{z}_\gamma \oplus \lambda_\gamma).$$
   Note that c_γ is a linear combination of λ_α, λ_β, λ_γ and $\lambda^*_\gamma = \lambda_\alpha \wedge \lambda_\beta$, therefore authenticated share of c_γ can be computed locally.
8. Two parties use Open to check that c_γ is 0 for all gates γ, and abort if any check fails.
9. For each $w \in \mathcal{O}$, two parties compute $r_w := \mathsf{Open}([r_w]_{\mathsf{A}})$. $\mathsf{P_B}$ computes $z_w := (\lambda_w \oplus z_w) \oplus r_w \oplus s_w$.

Figure 5.2: The main protocol in the $\mathcal{F}_{\mathsf{pre}}$ hybrid model

3-4. Parties process $\mathsf{P_A}$ and $\mathsf{P_B}$'s input and let $\mathsf{P_B}$ learn the corresponding masked input wire values and garbled labels.

5. $\mathsf{P_B}$ locally reconstructs the garbled circuit and evaluates it.

6-8. $\mathsf{P_B}$ sends all masked wire values (including all input, output, and internal wires) to $\mathsf{P_A}$; two parties check the correctness of all masked wire values.

9. $\mathsf{P_A}$ reveals the masks of output wires to $\mathsf{P_B}$, who can recover the output.

Note that steps 2 through 9 are performed in the online phase, with $2\kappa + 2$ bits of communication per AND gate, $\kappa + 1$ bits of communication per input bit, and 1 bit of communication per output bit.

5.3.1 Proof of Security

We start by stating our main theorem.

Theorem 5.3.1. *If H is modeled as a random oracle, the protocol in Figure 5.2 securely computes f against malicious adversaries in the $\mathcal{F}_{\mathsf{pre}}$-hybrid model.*

Before proceeding to the formal proof, we first introduce two important lemmas. The first lemma addresses correctness of our distributed garbling scheme in the semi-honest case; the second lemma addresses correctness of the whole protocol when $\mathsf{P_A}$ is corrupted.

Lemma 5.3.1. *When both parties follow the protocol honestly then, after step 5, for each wire w in the circuit $\mathsf{P_B}$ holds $(z_w \oplus \lambda_w, \mathsf{L}_{w,z_w\oplus\lambda_w})$.*

Proof. We prove this by induction on the gates in the circuit.

Base case. It is easy to verify from step 3 and step 4 that the lemma holds for input wires.

Induction step. XOR-gates are trivial and so focus on an AND gate $(\alpha, \beta, \gamma, \wedge)$. First, the garbled tables are computed distributively, therefore we first write down the table after $\mathsf{P_B}$ merged its own share as follows. Note that we ignore the gate id (γ) for simplicity.

$$\begin{aligned}
G_0 &= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \oplus \mathsf{K}[s_\beta] \oplus r_\beta \Delta_\mathsf{A} \oplus \mathsf{M}[s_\beta] \\
&= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \oplus \lambda_\beta \Delta_\mathsf{A} \\
G_1 &= H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\beta,1}) \oplus \mathsf{K}[s_\alpha] \oplus r_a \Delta_\mathsf{A} \oplus \mathsf{M}[s_\alpha] \oplus \mathsf{L}_{\alpha,0} \\
&= H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\beta,1}) \oplus \lambda_\alpha \Delta_\mathsf{A} \oplus \mathsf{L}_{\alpha,0}.
\end{aligned}$$

$\mathsf{P_A}$ locally computes the output garbled label for 0 values, namely $\mathsf{L}_{\gamma,0}$ as:

$$\mathsf{L}_{\gamma,0} := H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus \mathsf{K}[s_\gamma] \oplus r_\gamma \Delta_\mathsf{A} \oplus \mathsf{K}[s_\gamma^*] \oplus r_\gamma^* \Delta_\mathsf{A}.$$

$\mathsf{P_B}$, who holds $(z_\alpha \oplus \lambda_\alpha, \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha})$ and $(z_\beta \oplus \lambda_\beta, \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta})$ by the induction hypothesis, evaluates the circuit as follows:

$$\begin{aligned}
\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma} := {} & H(\mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \oplus H(\mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}) \oplus (z_\alpha \oplus \lambda_\alpha) G_0 \\
& \oplus (z_\beta \oplus \lambda_\beta)(G_1 \oplus \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \oplus \mathsf{M}[s_\gamma] \oplus \mathsf{M}[s_\gamma^*].
\end{aligned}$$

Observe that

$$\begin{aligned}
& (z_\alpha \oplus \lambda_\alpha) G_0 \oplus H(\mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \\
& = (z_\alpha \oplus \lambda_\alpha) \left(H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \oplus \lambda_\beta \Delta_\mathsf{A} \right) \oplus H(\mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \\
& = (z_\alpha \oplus \lambda_\alpha) \left(H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \oplus \lambda_\beta \Delta_\mathsf{A} \right) \oplus (z_\alpha \oplus \lambda_\alpha) \left(H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \right) \oplus H(\mathsf{L}_{\alpha,0}) \\
& = H(\mathsf{L}_{\alpha,0}) \oplus \lambda_\beta (z_\alpha \oplus \lambda_\alpha) \Delta_\mathsf{A},
\end{aligned}$$

and

$$
\begin{aligned}
&(z_\beta \oplus \lambda_\beta)(G_1 \oplus \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \oplus H(\mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}) \\
&= (z_\beta \oplus \lambda_\beta)\left(H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\beta,1}) \oplus \lambda_\alpha \Delta_\mathsf{A} \oplus (z_\alpha \oplus \lambda_\alpha)\Delta_\mathsf{A}\right) \oplus H(\mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}) \\
&= (z_\beta \oplus \lambda_\beta)\left(H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\beta,1}) \oplus z_\alpha \Delta_\mathsf{A}\right) \oplus (z_\beta \oplus \lambda_\beta)\left(H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\beta,1})\right) \oplus H(\mathsf{L}_{\beta,0}) \\
&= H(\mathsf{L}_{\beta,0}) \oplus (\lambda_\beta \oplus z_\beta) z_\alpha \Delta_\mathsf{A}.
\end{aligned}
$$

Therefore, we conclude that

$$
\begin{aligned}
&\mathsf{L}_{\gamma,0} \oplus \mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma} \\
&= H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\beta,0}) \oplus H(\mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \oplus H(\mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}) \oplus (z_\alpha \oplus \lambda_\alpha) G_0 \\
&\quad \oplus (z_\beta \oplus \lambda_\beta)(G_1 \oplus \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}) \oplus \lambda_\gamma \Delta_\mathsf{A} \oplus (\lambda_\alpha \wedge \lambda_\beta)\Delta_\mathsf{A} \\
&= (\lambda_\alpha \oplus z_\alpha)\lambda_\beta \Delta_\mathsf{A} \oplus (\lambda_\beta \oplus z_\beta) z_\alpha \Delta_\mathsf{A} \oplus \lambda_\gamma \Delta_\mathsf{A} \oplus (\lambda_\alpha \wedge \lambda_\beta)\Delta_\mathsf{A} \\
&= ((z_\alpha \wedge z_\beta) \oplus \lambda_\gamma)\Delta_\mathsf{A} = (z_\gamma \oplus \lambda_\gamma)\Delta_\mathsf{A}.
\end{aligned}
$$

This means that, with respect to $\mathsf{P_A}$'s definition of $\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma}$, $\mathsf{P_B}$'s label is always correct. The masked value is correct because the least-significant bit of Δ_A is 1; thus,

$$
\begin{aligned}
b_\gamma \oplus \mathsf{lsb}(\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma}) &= \mathsf{lsb}(\mathsf{L}_{\gamma,0}) \oplus \mathsf{lsb}(\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma}) \\
&= \mathsf{lsb}(\mathsf{L}_{\gamma,0} \oplus \mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma}) \\
&= \mathsf{lsb}((z_\gamma \oplus \lambda_\gamma)\Delta_\mathsf{A}) = z_\gamma \oplus \lambda_\gamma.
\end{aligned}
$$

□

Lemma 5.3.2. *Let* $x \overset{\text{def}}{=} \hat{x}_w \oplus \lambda_w$ *and* $y \overset{\text{def}}{=} \hat{y}_w \oplus \lambda_w$, *where* $\hat{x}_w$ *is what* $\mathsf{P_B}$ *sends in* *step 3, $\hat{y}_w$ is what $\mathsf{P_A}$ sends in step 4, and λ_w is defined by $\mathcal{F}_{\mathsf{pre}}$. If $\mathsf{P_A}$ is malicious, then $\mathsf{P_B}$ either aborts or outputs $f(x, y)$.*

Proof. After step 5, $\mathsf{P_B}$ obtains a set of masked values $z_w \oplus \lambda_w$ for all wires w in the circuit. In the following, we will show that if these masked values are not correct, then $\mathsf{P_B}$ will abort with all but negligible probability.

Again we will prove by induction. Note that the lemma holds for all wires $w \in \mathcal{I}_1 \cup \mathcal{I}_2$, according to how x, y are defined, as well as for XOR-gates. In the following, we will focus on an AND gate $(\alpha, \beta, \gamma, \wedge)$. Now, according to induction hypothesis, we already know that $\mathsf{P_B}$ hold correct values of $(z_\alpha \oplus \lambda_\alpha, z_\beta \oplus \lambda_\beta)$.

Recall that the checking is done by computing

$$c = (\hat{z}_\alpha \oplus \lambda_\alpha) \wedge (\hat{z}_\beta \oplus \lambda_\beta) \oplus (\hat{z}_\gamma \oplus \lambda_\gamma).$$

The correctness of input masked values means that

$$c = z_\alpha \wedge z_\beta \oplus \hat{z}_\gamma \oplus \lambda_\gamma.$$

Since Open does not abort, $c = 0$, which means that $\hat{z}_\gamma = z_\alpha \wedge z_\beta \oplus \lambda_\gamma = z_\gamma \oplus \lambda_\gamma$. This means that the output masked wire value is also correct. □

Given the above two lemmas, the proof of security of our main protocol is relatively easy. We provide all details below.

Proof. We consider separately a malicious $\mathsf{P_A}$ and $\mathsf{P_B}$.

Malicious $\mathsf{P_A}$. Let $\mathcal{A}$ be an adversary corrupting $\mathsf{P_A}$. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of $\mathsf{P_A}$ in the ideal world involving an ideal functionality $\mathcal{F}$ evaluating f. $\mathcal{S}$ is defined as follows.

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{pre}}$ and records all values that $\mathcal{F}_{\mathsf{pre}}$ sends to two parties.

2. $\mathcal{S}$ receives all values that $\mathcal{A}$ sends.

3. $\mathcal{S}$ acts as an honest $\mathsf{P_B}$ using input $y := 0$.

4. For each wire $w \in \mathcal{I}_1$, $\mathcal{S}$ receives $\hat{x}_w$ and computes $x_w := \hat{x}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{pre}}$ in the previous steps.

6. $\mathcal{S}$ picks random bits for all $\hat{z}_w$ and send them to $\mathcal{A}$.

7–9. $\mathcal{S}$ acts as an honest $\mathsf{P_B}$ If an honest $\mathsf{P_B}$ would abort, $\mathcal{S}$ aborts; otherwise $\mathcal{S}$ computes the input x of $\mathcal{A}$. from the output of $\mathcal{F}_{\mathsf{pre}}$ and the values $\mathcal{A}$ sent. $\mathcal{S}$ then sends x to $\mathcal{F}$.

We show that the joint distribution of the outputs of $\mathcal{A}$ and the honest $\mathsf{P_B}$ in the real world is indistinguishable from the joint distribution of the outputs of $\mathcal{S}$ and $\mathsf{P_B}$ in the ideal world. We prove this by considering a sequence of experiments, the first of which corresponds to the execution of our protocol and the last of which corresponds to execution in the ideal world, and showing that successive experiments are computationally indistinguishable.

Hybrid₁. This is the hybrid-world protocol, where we imagine $\mathcal{S}$ playing the role of an honest $\mathsf{P_B}$ using $\mathsf{P_B}$'s actual input y, while also playing the role of $\mathcal{F}_{\mathsf{pre}}$.

Hybrid₂. Same as **Hybrid₁**, except that in step 6, for each wire $w \in \mathcal{I}_1$ the simulator $\mathcal{S}$ receives $\hat{x}_w$ and computes $x_w := \hat{x}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{pre}}$. If an honest $\mathsf{P_B}$ would abort in any later step, $\mathcal{S}$ sends abort to $\mathcal{F}$; otherwise it sends $x = \{x_w\}_{w \in \mathcal{I}_1}$ to $\mathcal{F}$.

The distributions on the view of $\mathcal{A}$ in **Hybrid$_1$** and **Hybrid$_2$** are identical. The output $\mathsf{P_B}$ gets are the same due to Lemma 5.3.1 and Lemma 5.3.2.

Hybrid$_3$. Same as **Hybrid$_2$**, except that $\mathcal{S}$ uses $y' = 0$ in step 3 and ignore what $\mathcal{A}$ sends back. Then in step 6, $\mathcal{S}$ sends random bits instead of the value for $z_w \oplus \lambda_w$.

The distributions on the view of $\mathcal{A}$ in **Hybrid$_3$** and **Hybrid$_2$** are again identical (since the $\{s_w\}_{w \in \mathcal{I}_2}$ are uniform).

Note that **Hybrid$_3$** corresponds to the ideal-world execution described earlier. This completes the proof for a malicious $\mathsf{P_A}$.

Malicious $\mathsf{P_B}$. Let $\mathcal{A}$ be an adversary corrupting $\mathsf{P_B}$. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ as a subroutine and plays the role of $\mathsf{P_B}$ in the ideal world involving an ideal functionality $\mathcal{F}$ evaluating f. $\mathcal{S}$ is defined as follows.

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{pre}}$ and records all values sent to both parties.

2. $\mathcal{S}$ acts as an honest $\mathsf{P_A}$ and send the shared garbled tables to $\mathsf{P_B}$.

3. For each wire $w \in \mathcal{I}_2$, $\mathcal{S}$ receives $\hat{y}_w$ and computes $y_w := \hat{y}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{pre}}$ in the previous steps.

4. $\mathcal{S}$ acts as an honest $\mathsf{P_A}$ using input $x = 0$.

6–8. $\mathcal{S}$ acts as an honest $\mathsf{P_A}$. If an honest $\mathsf{P_A}$ would abort, $\mathcal{S}$ abort.

9. $\mathcal{S}$ sends y computed in step 3 to $\mathcal{F}$, which returns $z = f(x, y)$. $\mathcal{S}$ then computes $z' := f(0, y)$ and defines $r'_w = z_w \oplus z'_w \oplus r_w$ for each $w \in \mathcal{O}$. $\mathcal{S}$ then acts as

an honest $\mathsf{P_A}$ and opens values r'_w to $\mathcal{A}$. If an honest $\mathsf{P_A}$ would abort, $\mathcal{S}$ $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We now show that the distribution on the view of $\mathcal{A}$ in the real world is indistinguishable from the distribution on the view of $\mathcal{A}$ in the ideal world. (Note $\mathsf{P_A}$ has no output.)

Hybrid$_1$. This is the hybrid-world protocol, where $\mathcal{S}$ acts as an honest $\mathsf{P_A}$ using $\mathsf{P_A}$'s actual input x, while playing the role of $\mathcal{F}_{\mathsf{pre}}$.

Hybrid$_2$. Same as **Hybrid$_1$**, except that in step 3, $\mathcal{S}$ receives $\hat{y}_w$ and computes $y_w := \hat{y}_w \oplus r_w \oplus s_w$, where r_w, s_w are the values used by $\mathcal{F}_{\mathsf{pre}}$. If an honest $\mathsf{P_A}$ abort in any step, send abort to $\mathcal{F}$.

Hybrid$_3$. Same as **Hybrid$_2$**, except that in step 4, $\mathcal{S}$ acts as an honest $\mathsf{P_A}$ with input $x = 0$. $\mathcal{S}$ sends x computed in step 3 to $\mathcal{F}$, which returns $z = f(x, y)$. $\mathcal{S}$ then computes $z' := f(0, y)$ and defines $r'_w = z_w \oplus z'_w \oplus r_w$ for each $w \in \mathcal{O}$. $\mathcal{S}$ then acts as an honest $\mathsf{P_A}$ and opens values r'_w to $\mathcal{A}$. If an honest $\mathsf{P_A}$ would abort, $\mathcal{S}$ $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

The distributions on the view of $\mathcal{A}$ in **Hybrid$_3$** and **Hybrid$_2$** are identical.

Note that **Hybrid$_3$** is identical to the ideal-world execution. □

5.4 Technical Details: Improved Preprocessing

In this section, we provide details for our two optimizations of the preprocessing phase. The first optimization improves the efficiency to compute a leaky AND gate.

Leaky AND gate is a key component towards a preprocessing with full security. This functionality ($\mathcal{F}_{\mathsf{LaAND}}$) outputs triples with guaranteed correctness but the adversary can choose to guess the x value from the honest party: an incorrect guess will be caught immediately; while a correct guess remain undetected.

The second optimization focuses on how to combine leaky triples in a more efficient way. In particular, we observe that a recent optimization in the honest-majority secret sharing protocol by Araki et al. [81], can be applied to our setting too. As a result, we can roughly reduce the bucket size by one.

5.4.1 Improved Leaky AND

Before giving the details, we point out a minor difference in the leaky-AND functionality ($\mathcal{F}_{\mathsf{LaAND}}$) as compared to [7]. As shown in Figure 5.3, instead of letting $\mathcal{A}$ directly learn the value of x, the functionality allows $\mathcal{A}$ to send a query in a form of (P_1, p_2, P_3) and return if $P_3 \oplus x_2 P_1 = (p_2 \oplus x_2 \mathsf{lsb}(P_1))\Delta_{\mathsf{B}}$. It can be seen that this special way is no more than a query on x and two queries on Δ, and the $\mathcal{A}$ cannot learn any information on y or z.

The main intuition of the protocol is already discussed in Section 5.2.2. We will proceed to present the protocol, in Figure 5.4.

Theorem 5.4.1. *The protocol in Figure 5.4 securely realizes $\mathcal{F}_{\mathsf{LaAND}}$ in the ($\mathcal{F}_{\mathsf{abit}}$,$\mathcal{F}_{\mathsf{EQ}}$)-hybrid model.*

Proof. As the first step, we will show that the protocol is correct if both parties are honest. We recall that

Functionality $\mathcal{F}_{\mathsf{LaAND}}$

Honest case:

1. Generate uniform $\langle x_1 \,|\, x_2 \rangle$, $\langle y_1 \,|\, y_2 \rangle$, $\langle z_1 \,|\, z_2 \rangle$ such that $z_1 \oplus z_2 = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$, and send the respective shares to the two parties.
2. $\mathsf{P_A}$ can choose to send $(P_1, p_2, P_3) \in \{0,1\}^\kappa \times \{0,1\} \times \{0,1\}^\kappa$. The functionality checks
$$P_3 \oplus x_2 P_1 = (p_2 \oplus x_2 \mathsf{lsb}(P_1))\, \Delta_{\mathsf{B}}.$$
If the check fails, the functionality sends fail to both parties and abort. ($\mathsf{P_B}$ can do the same symmetrically.)

Corrupted parties: A corrupted party gets to specify the randomness used on its behalf by the functionality.

Figure 5.3: Functionality $\mathcal{F}_{\mathsf{LaAND}}$ for computing a leaky AND triple.

Protocol Π_{LaAND}

Protocol:

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ obtain random authenticated shares $(\langle x_1 \,|\, x_2 \rangle, \langle y_1 \,|\, y_2 \rangle, \langle z_1 \,|\, r \rangle)$.
$\mathsf{P_A}$ locally computes $C_{\mathsf{A}} := y_1 \Delta_{\mathsf{A}} \oplus \mathsf{K}[y_2] \oplus \mathsf{M}[y_1]$, and
$\mathsf{P_B}$ locally computes $C_{\mathsf{B}} := y_2 \Delta_{\mathsf{B}} \oplus \mathsf{M}[y_2] \oplus \mathsf{K}[y_1]$.
2. $\mathsf{P_A}$ sends $G_1 := H(\mathsf{K}[x_2] \oplus \Delta_{\mathsf{A}}) \oplus H(\mathsf{K}[x_2]) \oplus C_{\mathsf{A}}$ to $\mathsf{P_B}$.
$\mathsf{P_B}$ computes $E_1 := x_2 G_1 \oplus H(\mathsf{M}[x_2]) \oplus x_2 C_{\mathsf{B}}$.
3. $\mathsf{P_B}$ sends $G_2 := H(\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}}) \oplus H(\mathsf{K}[x_1]) \oplus C_{\mathsf{B}}$ to $\mathsf{P_A}$.
$\mathsf{P_A}$ computes $E_2 := x_1 G_2 \oplus H(\mathsf{M}[x_1]) \oplus x_1 C_{\mathsf{A}}$.
4. $\mathsf{P_A}$ computes $S_1 := H(\mathsf{K}[x_2]) \oplus E_2 \oplus (z_1 \Delta_{\mathsf{A}} \oplus \mathsf{K}[r] \oplus \mathsf{M}[z_1])$, $\mathsf{P_B}$ computes $S_2 := H(\mathsf{K}[x_1]) \oplus E_1 \oplus (r \Delta_{\mathsf{B}} \oplus \mathsf{M}[r] \oplus \mathsf{K}[z_1])$. $\mathsf{P_A}$ sends $\mathsf{lsb}(S_1)$ to $\mathsf{P_B}$; $\mathsf{P_B}$ sends $\mathsf{lsb}(S_2)$ to $\mathsf{P_A}$. Both parties computes $d := \mathsf{lsb}(S_1) \oplus \mathsf{lsb}(S_2)$.
5. $\mathsf{P_A}$ sends $L_1 := S_1 \oplus d\Delta_{\mathsf{A}}$ to $\mathcal{F}_{\mathsf{EQ}}$, $\mathsf{P_B}$ sends $L_2 := S_2 \oplus d\Delta_{\mathsf{B}}$ to $\mathcal{F}_{\mathsf{EQ}}$. If $\mathcal{F}_{\mathsf{EQ}}$ returns 0, parties abort, otherwise, they compute $[z_2]_{\mathsf{B}} := [r]_{\mathsf{B}} \oplus d$.

Figure 5.4: Our improved leaky-AND protocol.

1. $G_1 := H(\mathsf{K}[x_2] \oplus \Delta_{\mathsf{A}}) \oplus H(\mathsf{K}[x_2]) \oplus C_{\mathsf{A}}$

2. $G_2 := H(\mathsf{K}[x_1] \oplus \Delta_{\mathsf{B}}) \oplus H(\mathsf{K}[x_1]) \oplus C_{\mathsf{B}}$

3. $C_{\mathsf{A}} := y_1 \Delta_{\mathsf{A}} \oplus \mathsf{K}[y_2] \oplus \mathsf{M}[y_1]$

4. $C_{\mathsf{B}} := y_2\Delta_{\mathsf{B}} \oplus \mathsf{M}[y_2] \oplus \mathsf{K}[y_1]$

Note that

$$E_1 \oplus H(\mathsf{K}[x_2]) = x_2G_1 \oplus H(\mathsf{M}[x_2]) \oplus x_2C_{\mathsf{B}} \oplus H(\mathsf{K}[x_2]).$$

When $x_2 = 0$, we have

$$\begin{aligned} E_1 \oplus H(\mathsf{K}[x_2]) &= x_2G_1 \oplus H(\mathsf{M}[x_2]) \oplus x_2C_{\mathsf{B}} \oplus H(\mathsf{K}[x_2]) \\ &= H(\mathsf{M}[x_2]) \oplus H(\mathsf{K}[x_2]) \\ &= 0 = x_2(C_{\mathsf{A}} \oplus C_{\mathsf{B}}). \end{aligned}$$

When $x_2 = 1$, we have

$$\begin{aligned} E_1 \oplus H(\mathsf{K}[x_2]) &= x_2G_1 \oplus H(\mathsf{M}[x_2]) \oplus x_2C_{\mathsf{B}} \oplus H(\mathsf{K}[x_2]) \\ &= x_2(G_1 \oplus C_{\mathsf{B}}) \oplus H(\mathsf{M}[x_2]) \oplus H(\mathsf{K}[x_2]) \\ &= x_2(G_1 \oplus C_{\mathsf{B}}) \oplus H(\mathsf{K}[x_2] \oplus \Delta_{\mathsf{A}})) \oplus H(\mathsf{K}[x_2]) \\ &= x_2(C_{\mathsf{A}} \oplus C_{\mathsf{B}}). \end{aligned}$$

Therefore,

$$\begin{aligned} E_1 \oplus H(\mathsf{K}[x_2]) &= x_2(C_{\mathsf{A}} \oplus C_{\mathsf{B}}) \\ &= x_2(y_1\Delta_{\mathsf{A}} \oplus \mathsf{K}[y_2] \oplus \mathsf{M}[y_1] \oplus y_2\Delta_{\mathsf{B}} \oplus \mathsf{M}[y_2] \oplus \mathsf{K}[y_1])) \\ &= x_2(y_1\Delta_{\mathsf{A}} \oplus y_2\Delta_{\mathsf{A}} \oplus y_1\Delta_{\mathsf{B}} \oplus y_2\Delta_{\mathsf{B}}) \\ &= x_2(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}). \end{aligned}$$

Similarly,

$$E_2 \oplus H(\mathsf{K}[x_1]) = x_1(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}).$$

Taking these two equations, we know that

$$\begin{aligned}
S_1 \oplus S_2 &= (E_1 \oplus H(\mathsf{K}[x_2])) \oplus (E_2 \oplus H(\mathsf{K}[x_1])) \\
&\quad \oplus (z_1\Delta_{\mathsf{A}} \oplus \mathsf{K}[r] \oplus \mathsf{M}[z_1] \oplus r\Delta_{\mathsf{B}} \oplus \mathsf{M}[r] \oplus \mathsf{K}[z_1]) \\
&= (x_1 \oplus x_2)(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) \\
&\quad \oplus (z_1\Delta_{\mathsf{A}} \oplus \mathsf{K}[z_1] \oplus \mathsf{M}[z_1] \oplus r\Delta_{\mathsf{B}} \oplus \mathsf{K}[r] \oplus \mathsf{M}[r]) \\
&= (x_1 \oplus x_2)(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) \\
&\quad \oplus (z_1\Delta_{\mathsf{A}} \oplus z_1\Delta_{\mathsf{B}} \oplus r\Delta_{\mathsf{B}} \oplus r\Delta_{\mathsf{A}}) \\
&= (x_1 \oplus x_2)(y_1 \oplus y_2)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) \oplus (z_1 \oplus r)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) \\
&= ((x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \oplus z_1 \oplus r)(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}).
\end{aligned}$$

Since $\mathsf{lsb}(\Delta_{\mathsf{A}} \oplus \Delta_{\mathsf{B}}) = 1$, it holds that

$$d = \mathsf{lsb}(S_1 \oplus S_2) = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \oplus z_1 \oplus r.$$

Therefore, $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) = d \oplus z_1 \oplus r = z_1 \oplus z_2$.

Now we will focus on the security of the protocol in the malicious setting. First note that the protocol is symmetric, therefore we only need to focus on the case of a malicious P_{A}. The local computation of both parties is deterministic, with all inputs sent from $\mathcal{F}_{\mathsf{abit}}$. Therefore, all messages sent during the protocol can be anticipated (emulated) by $\mathcal{S}$ after $\mathcal{S}$ sending out the shares. This is not always possible if $\mathcal{A}$ uses local random coins or if $\mathcal{A}$ has private inputs. This fact significantly reduces the difficulty of the proof. Intuitively, $\mathcal{S}$ will be able to immediately catch $\mathcal{A}$ cheating by comparing what it sends with what it would have sent (which $\mathcal{S}$ knows by locally emulating). The majority of the work then is to extract $\mathcal{A}$'s attempt to perform a selective failure attack.

Define a simulator $\mathcal{S}$ as follows.

0a. $\mathcal{S}$ interacts with $\mathcal{F}_{\mathsf{LaAND}}$ and obtains $\mathsf{P_A}$'s share of $(\langle x_1 \,|\, x_2\rangle, \langle y_1 \,|\, y_2\rangle, \langle z_1 \,|\, z_2\rangle)$. $\mathcal{S}$ also gets Δ_{A} from $\mathcal{F}_{\mathsf{abit}}$. $\mathcal{S}$ randomly picks Δ_{B} and $\mathsf{P_B}$'s share of $(\langle x_1 \,|\, x_2\rangle, \langle y_1 \,|\, y_2\rangle, \langle z_1 \,|\, z_2\rangle)$ in a way that makes it consistent with $\mathsf{P_A}$'s share. $\mathcal{S}$ now randomly picks d and computes $[r]_{\mathsf{B}} := [z_2]_{\mathsf{B}} \oplus d$.

0b. Using values $(\langle x_1 \,|\, x_2\rangle, \langle y_1 \,|\, y_2\rangle, \langle z_1 \,|\, r\rangle)$ from both parties, $\mathcal{S}$ locally emulates all messages sent by each party, namely (G_1, d_1, L_1) sent by an honest $\mathsf{P_A}$ and (G_2, d_2, L_2) sent by an honest $\mathsf{P_B}$.

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{abit}}$ and sends out $(\langle x_1 \,|\, x_2\rangle, \langle y_1 \,|\, y_2\rangle, \langle z_1 \,|\, r\rangle)$ as defined above.

2. $\mathcal{S}$ acts as an honest $\mathsf{P_B}$ and receive G_1' sent by $\mathcal{A}$. $\mathcal{S}$ computes $P_1 = G_1' \oplus G_1$.

3. $\mathcal{S}$ randomly picks a G_2 and send it to $\mathcal{A}$.

4. $\mathcal{S}$ acts as an honest $\mathsf{P_B}$ and receives d_1'. $\mathcal{S}$ computes $p_2 := d_1' \oplus d_1$.

5. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{EQ}}$ and obtain L_1. $\mathcal{S}$ computes $P_3 = L_1' \oplus L_1$. $\mathcal{S}$ sends (P_1, p_2, P_3) to $\mathcal{F}_{\mathsf{LaAND}}$ as the selective failure attack query. If $\mathcal{F}_{\mathsf{LaAND}}$ abort, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{EQ}}$ and aborts. If the value d in the protocol equals to r defined in step 0a, $\mathcal{F}_{\mathsf{EQ}}$ returns 0; otherwise $\mathcal{F}_{\mathsf{EQ}}$ returns 1.

6. $\mathcal{S}$ sends (P_1, p_2, P_3) to $\mathcal{F}_{\mathsf{LaAND}}$ as the selective failure query. If $\mathcal{F}_{\mathsf{LaAND}}$ returns fail, $\mathcal{S}$ sends 0 to $\mathcal{A}$ as the output of $\mathcal{F}_{\mathsf{EQ}}$.

Protocol Π_{pre}

Inputs: Two parties agree on a circuit for a function $f : \{0,1\}^{\mathcal{I}_1} \times \{0,1\}^{\mathcal{I}_2} \rightarrow \{0,1\}^{\mathcal{O}}$.
Protocol:

1. Two parties initialize $\mathcal{F}_{\mathsf{abit}}$, which sends Δ_{A} to $\mathsf{P_A}$ and Δ_{B} to $\mathsf{P_B}$.
2. For each wire $w \in \mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{W}$, two parties obtain an authenticated share $\langle r_w \,|\, s_w \rangle$ from $\mathcal{F}_{\mathsf{abit}}$.
3. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \oplus)$, two parties compute $\langle r_\gamma \,|\, s_\gamma \rangle := \langle s_\alpha \,|\, r_\alpha \rangle \oplus \langle r_\beta \,|\, s_\beta \rangle$.
4. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, \wedge)$, two parties have $(\langle r_\alpha \,|\, s_\alpha \rangle, \langle r_\beta \,|\, s_\beta \rangle)$, and run step 2 to step 5 in Π_{LaAND} to obtain $\langle r^*_\gamma \,|\, s^*_\gamma \rangle$, such that $r^*_\gamma \oplus s^*_\gamma = (r_\alpha \oplus s_\alpha) \wedge (r_\beta \oplus s_\beta)$
5. $\mathsf{P_A}$ and $\mathsf{P_B}$ call $\mathcal{F}_{\mathsf{LaAND}}$ to obtain $(B-1)|\mathcal{C}|$ number of leaky AND triples $(\langle x_1 \,|\, x_2 \rangle, \langle y_1 \,|\, y_2 \rangle, \langle z_1 \,|\, z_2 \rangle)$.
6. Two parties perform secure coin-flipping to determine a random permutation and permute the triples obtained in step 4. For each AND gate $\mathcal{G} = (\alpha, \beta, \gamma, \wedge)$ in the circuit, perform secure merging for $B-1$ times.
   (a) Obtain the next triple in the permuted list, namely $(\langle x_1 \,|\, x_2 \rangle, \langle y_1 \,|\, y_2 \rangle, \langle z_1 \,|\, z_2 \rangle)$
   (b) Compute $\langle d_1 \,|\, d_2 \rangle := \langle y_1 \,|\, y_2 \rangle \oplus \langle r_\beta \,|\, s_\beta \rangle$, and $d := \mathsf{Open}(\langle d_1 \,|\, d_2 \rangle)$.
   (c) Update triple: $\langle r_\alpha \,|\, s_\alpha \rangle := \langle r_\alpha \,|\, s_\alpha \rangle \oplus \langle x_1 \,|\, x_2 \rangle$, $\langle r^*_\gamma \,|\, s^*_\gamma \rangle := \langle r^*_\gamma \,|\, s^*_\gamma \rangle \oplus \langle z_1 \,|\, z_2 \rangle \oplus d \langle x_1 \,|\, x_2 \rangle$.

Figure 5.5: Protocol Π_{pre} instantiating $\mathcal{F}_{\mathsf{pre}}$ in the ($\mathcal{F}_{\mathsf{abit}}$, $\mathcal{F}_{\mathsf{LaAND}}$)-hybrid model.

Note that messages that $\mathcal{S}$ sends to $\mathcal{A}$ in the protocol are changed from (G_2, d_2, L_2) to $(G_2, d_2 \oplus x_2 \mathsf{lsb}(P_1), L_2 \oplus x_2 P_1 \oplus d' \Delta_{\mathsf{B}})$, where $d' = p_2 \oplus x_2 \cdot \mathsf{lsb}(P_1)$ and the equality checking in step 5 changed from comparing $L_1 = L_2$ to

$$L_1 \oplus P_3 = L_2 \oplus x_2 P_1 \oplus (p_2 \oplus x_2 \mathsf{lsb}(P_1))\, \Delta_{\mathsf{B}},$$

that is

$$P_3 \oplus x_2 P_1 = (p_2 \oplus x_2 \mathsf{lsb}(P_1))\, \Delta_{\mathsf{B}}.$$

This is the same form as the selective failure query in $\mathcal{F}_{\mathsf{LaAND}}$. $\square$

5.4.2 Improved Function-Dependent Preprocessing

In this section, we will focus on improving the preprocessing in the Leaky AND triple generation ($\mathcal{F}_{\mathsf{LaAND}}$) hybrid model. The main observation is that in the protocol of WRK, each wire is associated with a mask (in the authenticated share format). Then the AND of input masks are computed using one AND triple. This is a waste of randomness, since we also directly construct all triples in place for all wires. Note that the idea is similar to Araki et al. [81]. The detailed protocol is presented in Figure 5.5

Note that although the above optimization aims to reduce the overall cost of the protocol, but it turns out that even in this case, most of the computation and communication (including computation of all authenticated bits as well as all leaky-AND triples in step 5) can be still done in the function-independent phase. The function-dependent cost is increased by only κ bits per AND gate only. Therefore, here we have an option to trade-off between total communication and communication in the offline stage. By increasing the function-dependent cost by κ bits per gate, we reduce bucket size by 1. We believe both versions can be useful depending on the application, and the concrete cost of both versions of the protocol are presented in the performance section.

5.5 Performance

In this section, we discuss the concrete efficiency of our protocol. We consider two variants of our protocol that optimize the cost of different phases: The first version

Table 5.1: **Communication complexity of different protocols for evaluating AES, rounded to two significant figures.** One-way communication refers to the maximum communication one party sends to the other; two-way communication refers to the sum of both parties' communication. The best prior number in each column is bolded for reference.

	One-way Communication (Max)				Two-way Communication			
	Ind. (MB)	Dep. (MB)	Online (KB)	Total (MB)	Ind. (MB)	Dep. (MB)	Online (KB)	Total (MB)
Single execution								
[37]	15	**0.22**	16	15	15	0.22	16	15
[7]	**2.9**	0.57	**4.9**	**3.4**	**5.7**	**0.57**	**6.0**	6.3
[82]	-	3.4	≥ 4.9	**3.4**	-	3.4	≥ 4.9	**3.4**
This work, v. 1	1.9	0.33	5.0	2.2	3.8	0.33	5.0	4.2
This work, v. 2	2.5	0.22	5.0	2.7	4.9	0.22	5.0	5.1
Amortized cost over 1024 executions								
[32]	-	1.6	17	**1.6**	-	3.2	17	3.2
[37]	6.4	**0.22**	16	6.6	6.4	**0.22**	16	6.6
[38]	-	1.6	19	**1.6**	-	1.6	19	**1.6**
[7]	**2.0**	0.57	**4.9**	2.6	**4.0**	0.57	**6.0**	4.6
This work, v. 1	1.4	0.33	5.0	1.7	2.7	0.33	5.0	3.1
This work, v. 2	1.9	0.22	5.0	2.1	3.8	0.22	5.0	4.0

of our protocol is optimized to minimize the total communication; the second version is optimized to minimize the communication in the function-dependent phase. (The cost of the online phase is identical in both versions.)

5.5.1 Communication Complexity

Table 5.1 shows the communication complexity of recent two-party computation protocols in the malicious setting. Numbers for these protocols are obtained from the respective papers, while numbers for our protocol are calculated. We tabulate both one-way communication and total communication. If parties' data can be sent at the same time over a full-duplex network, then one-way communication is a better reflection of the running time. In general, for a circuit that requires a bucket size of B, we can obtain an estimation of the concrete communication cost: our first version has function dependent cost of 3κ per gate, and function independent cost of $(4B-2)\kappa+(3B-1)\rho$ per gate; our second version has a function dependent cost of 2κ per gate, and a function independent cost of $(4B+2)\kappa+(3B+2)\rho$ per gate.

We see that our protocol and the protocol by Nielsen et al. [37] are the only ones that, considering the function-dependent phase and the online phase, have cost similar to that of the state-of-the-art semi-honest garbled-circuit protocol. In other words, *the overhead induced by malicious security can be completely pushed to the preprocessing stage*. Compared to the protocol by Nielsen et al., we are able to reduce the communication in the preprocessing stage by $6\times$ in the single-execution setting, and by $3.4\times$ in the amortized setting. Our protocol also has the best total communication complexity in both settings, excepting the work of [32, 38] which are 6% better but do not support function-independent preprocessing.

5.5.2 Computational Complexity

Since the WRK protocol represents the state-of-the-art as far as implementations are concerned, we compare the computational complexity of our protocol to theirs. We also include a comparison to the more recent protocol by Hazay et al. [82] (the *HIV protocol*), which has not yet been implemented.

Table 5.2: **Number of H-evaluations.** We align the security parameters in both protocols and set $B = \rho/\log C + 1$ for a fair comparison.

	Ind.	Dep.	Online	Total
WRK	$10B$	8	2	$10B + 10$
This work, v. 1	$4B - 4$	8	2	$4B + 6$
This work, v. 2	$4B$	4	2	$4B + 6$

Comparing to the WRK protocol. Our protocol follows the same high-level approach as the WRK protocol. Almost all H-evaluations in our protocol can be accelerated using fixed-key AES, as done in [18]. We tabulate the number of H-evaluations for both protocols in Table 5.2. Due to our improved $\mathcal{F}_{\mathsf{LaAND}}$, we are able to achieve a 2–2.5$\times$ improvement.

Comparing to the HIV protocol. As noted by the authors, the HIV protocol has polylogarithmic computational overhead compared to semi-honest garbled circuits. This is due to their use of the MPC-based zero-knowledge proof by Ames et al. [83]. On the other hand, in our protocol, the computation is linear in the circuit size. Furthermore, almost all cryptographic operations in our protocol can be accelerated using hardware AES instructions.

Taking an AES circuit as example, the ZK protocol by Ames et al. for a circuit of that size has a prover running time of around 70 ms and a verifier running time of around 30 ms. Therefore, even if we ignore the cost of computing and sending the garbled circuit, the oblivious transfers, and other operations, the end-to-end running time of the HIV protocol will still be at least 100 ms. On the other hand, the entire WRK protocol runs within 17 ms for the same circuit. As our protocol results in at least a 2× improvement, our protocol will be at least an order of magnitude faster than the HIV protocol.

5.6 Challenges in Extending to the Multi-Party Case

Wang et al. [8] have also shown how to extend their authenticated-garbling protocol to the multi-party case. In this section, we discuss the challenges involved in applying our new techniques to that setting. Note that Ben-Efraim et al. [84] recently proposed new techniques for multi-party garbling, making it compatible with some of the half-gate optimizations. Despite being based on half-gates, they still require 4 garbled rows per AND gate, and thus their work still leaves open the question of reducing the communication complexity of the online phase in the multi-party case.

In the multi-party WRK protocol, there are $n-1$ garbling parties and one evaluating party. For each wire, each garbler chooses their own set of wire labels (called "subkeys"). As in the 2-party case, the preprocessing defines some authenticated bits, and as a result all parties can locally compute additive shares of *any garbler's* subkey corresponding to *any authenticated value*.

In each gate, each garbler P_i generates standard Yao garbled gate consisting of 4 rows. Each row of P_i's gate is encrypted by only P_i's subkeys, and the payload of the row is P_i's shares of *all garblers'* subkeys. That way, the evaluator can decrypt the correct row of everyone's garbled gates, obtain everyone's shares of everyone's subkeys, and combine them to get everyone's appropriate subkey for the output wire.

Now suppose we modify things so each garbler generates a half-gates-style garbled gate instead of a standard Yao garbled gate. The half-gate uses garbler P_i's subkeys as its "keys" and encodes P_i's shares of all subkeys as its "payloads". Now the protocol may not be secure against an adversary corrupting the evaluator and a garbler. In particular, half-gates garbling defines $G_0 = H(\mathsf{L}_{\alpha,0}) \oplus H(\mathsf{L}_{\alpha,1}) \oplus \lambda_\beta \Delta$. When P_i is acting as garbler, these $\mathsf{L}_{\alpha,u}$ values correspond to P_i's subkeys. Now suppose P_i colludes with the evaluator. If the evaluator comes to learn G_0 (which is necessary to evaluate the gate in half of the cases), then the adversary can learn the secret mask λ_β since it is the only unknown term in G_0. Clearly revealing the secret wire mask breaks the privacy of the protocol. This is not a problem with Yao garbled gates, where each row can be written as $G_{u,v} = H(\mathsf{L}_{\alpha,u}, \mathsf{L}_{\beta,v}) \oplus$ [payload already known to garbler]. The secret masks do not appear in the garbled table, except indirectly through the payloads (subkey shares).

It is even unclear if row-reduction can be made possible. In the multi-party setting, the garbler has no control over the "payload" (i.e., output wire label) of the garbled gate when using row-reduction. Indeed, this is what makes it possible to reduce the size of a garbled gate. This is not a problem in the two-party case, where there is only one garbler who has control over all garbled gates and all wire labels. He generates a garbled table, and then computes his output wire label (subkey) as a function of the payload in the table. However, in the multi-party case, P_i generates a half-gate whose payloads include P_i's shares of P_j's subkeys! We would need P_j's choice of subkeys to depend on the payloads of P_i's garbling (for all i and j!). It is not clear how this can be done, and even if it were possible it would apparently require additional rounds proportional to the depth of the circuit.

Chapter 6: EMP-toolkit: Efficient Multi-Party Computation Toolkit

As part of this thesis, we also developed the EMP toolkit, which contains the implementation of protocols mentioned in this thesis as well as other commonly used protocols and building blocks. From a high-level view, EMP is developed with two goals in mind:

1. It should allow cryptography researchers to develop fast prototypes of their protocols to confirm and demonstrate the efficiency and scalability.
2. It should also allow application developers from academia and industry to develop reliable and scalable applications on top of MPC, without the need to understand the underlying cryptographic techniques.

In the following, we will discuss how EMP achieves the above two goals.

6.1 Prototyping MPC Protocols

EMP aims to help cryptography researchers to develop protocol prototypes with ease. To this end, we developed EMP in a layered structure. The bottom layer of EMP is a rich set of building blocks that are commonly needed in cryptographic protocols, including the pseudorandom generator (PRG), hash function, network communication, wrappers for elliptic-curve cryptography (ECC) operations and Advanced Vector Extensions (AVX) operations. All building blocks are carefully optimized to take advantage of state-of-the-art optimizations and instructions to achieve the best performance. In addition, they are encapsulated as EMP object for ease-of-use. The following code snippet is an example of how PRG can be used in EMP.

```
#include<emp-tool/emp-tool.h>
using namespace emp;
PRG prg;//using a secure random seed

int rand_int, rand_ints[100];
block rand_block[3];
mpz_t integ;mpz_init(integ);

prg.random_data(&rand_int, sizeof(rand_int)); //fill rand_int with 32 random bits
prg.random_block(rand_block, 3);              //fill rand_block with 128*3 random bits

prg.reseed(&rand_block[1]);                   //reset the seed and counter in prg
prg.random_data_unaligned(rand_ints+2, sizeof(int)*98);  //when the array is not 128-bit-aligned
prg.random_mpz(integ, 1024);                  //random number with 1024 bits.
```

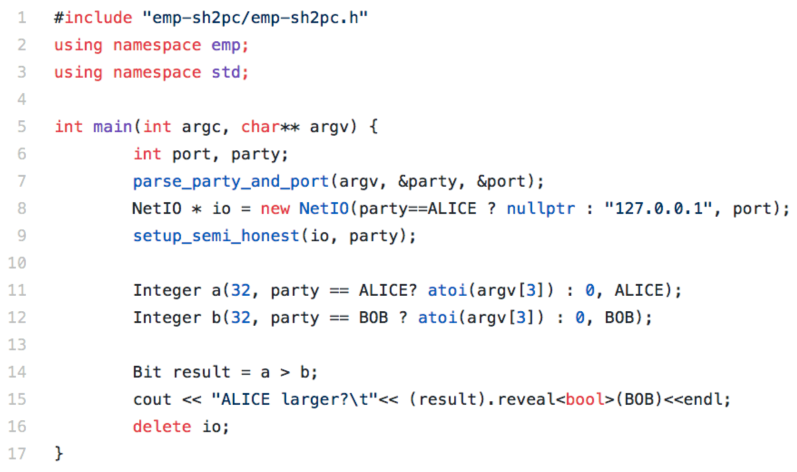
On top of the basic layer is a set of commonly used functionalities specific for MPC protocols, including garbling, Oblivious Transfer (OT) and OT extensions. Similar to basic building blocks, they are implemented as EMP objects with similar interfaces. The following code snippet is an example of how to set up network communication and run semi-honest OT between two parties.

```
#include<emp-tool/emp-tool.h> // for NetIO, etc
#include<emp-ot/emp-ot.h>   // for OTs

block b0[length], b1[length];
bool c[length];
NetIO io(party==ALICE ? nullptr:"127.0.0.1", port); // Create a network with Bob connecting to 127.0.0.1
NPOT<NetIO> np(&io); // create a Naor Pinkas OT using the network above
if (party == ALICE)
// ALICE is sender, with b0[i] and b1[i] as messages to send
    np.send(b0, b1, length);
else
// Bob is receiver, with c[i] as the choice bit
// and obtains b0[i] if c[i]==0 and b1[i] if c[i]==1
    np.recv(b0, c, length);
```

6.2 Developing MPC Applications.

EMP provides a rich set of ways to develop applications easily. The following is an implementation of the millionaire problem in EMP based on the semi-honest garbled circuit protocol.



```
 1  #include "emp-sh2pc/emp-sh2pc.h"
 2  using namespace emp;
 3  using namespace std;
 4
 5  int main(int argc, char** argv) {
 6          int port, party;
 7          parse_party_and_port(argv, &party, &port);
 8          NetIO * io = new NetIO(party==ALICE ? nullptr : "127.0.0.1", port);
 9          setup_semi_honest(io, party);
10
11          Integer a(32, party == ALICE? atoi(argv[3]) : 0, ALICE);
12          Integer b(32, party == BOB ? atoi(argv[3]) : 0, BOB);
13
14          Bit result = a > b;
15          cout << "ALICE larger?\t"<< (result).reveal<bool>(BOB)<<endl;
16          delete io;
17  }
```

From line 1 to line 3, we include necessary libraries and namespaces. From line 6 to line 9, we set up the network and the garbled circuit protocol. Line 11 and line 12 specifies inputs from each party. In particular, line 11 says that Alice inputs a 32-bit integer as inputs (from argv[3]); line 12 says that Bob also has a 32-bit integer as the input. Line 14 computes a comparison circuit in garbled circuits, and the result is revealed only to Bob in the end.

EMP has implemented most commonly used operations for integer, and floating-point numbers. It also implements numerous useful circuits that are essential to building efficient circuits in MPC.

Chapter 7: Conclusion

This dissertation shows a new paradigm to design maliciously secure multi-party computation protocols that are secure against all-but-one corruption with high efficiency. Specifically,

1. We present a maliciously secure two-party computation protocol that significantly deviates from existing approach and achieves extremely high concrete efficiency. The protocol can also be instantiated such that one gate only needs $O(\kappa)$ bits communication.

2. We present a maliciously secure multi-party computation protocol that scales to hundreds to parties distributed globally. In a high latency network, this protocol achieves the best performance both asymptotically and concretely.

3. We present further optimizations to the two-party protocol to benefit from state-of-the-art garbling optimizations. The resulting protocol has a similar cost to the semi-honest protocol in the preprocessing model.

4. Finally, we present EMP-toolkit, an efficient and easy-to-use framework to building MPC protocols and applications.

Bibliography

[1] Dyadic Security. https://www.dyadicsec.com/.

[2] Sharemind. https://sharemind.cyber.ee/.

[3] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: A privacy-preserving social study using secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2016.

[4] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 2014.

[5] Partisia Market Design. http://www.partisia.dk/.

[6] 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, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

[7] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 21–37, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[8] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 39–56, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[9] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In *CRYPTO 2018*, LNCS, Santa Barbara, CA, USA, August 20–24, 2018. Springer, Heidelberg, Germany.

[10] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit, 2016.

[11] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

[12] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *USENIX Security 2004*, 2004.

[13] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security 2011*, 2011.

[14] 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, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[15] 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, Reykjavik, Iceland, July 7–11, 2008. Springer, Heidelberg, Germany.

[16] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[17] 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, Berlin, Germany, November 4–8, 2013. ACM Press.

[18] 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, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[19] 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, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[20] 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, Barcelona, Spain, May 20–24, 2007. Springer, Heidelberg, Germany.

[21] 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, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[22] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 523–534, Berlin, Germany, November 4–8, 2013. ACM Press.

[23] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security 2012*, 2012.

[24] 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, Providence, RI, USA, March 28–30, 2011. Springer, Heidelberg, Germany.

[25] 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, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[26] 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, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[27] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 441–463, Bengalore, India, December 1–5, 2013. Springer, Heidelberg, Germany.

[28] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of

LNCS, pages 399–424, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[29] 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, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[30] 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, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[31] 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, Denver, CO, USA, October 12–16, 2015. ACM Press.

[32] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security 2016*, 2016.

[33] Jesper Buus Nielsen and Claudio Orlandi. Cross and clean: Amortized garbled circuits with constant overhead. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 582–603, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany.

[34] 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, Germany, March 15–17, 2009.

[35] 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, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

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

[37] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society.

[38] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 3–20, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[39] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

[40] 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, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[41] 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, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[42] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[43] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages

554–581, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany.

[44] 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, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[45] 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, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[46] 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, New York City, NY, USA, May 25–27, 1987. ACM Press.

[47] 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, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[48] 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, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

[49] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.

[50] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 495–512, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[51] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

[52] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh

Jha, editors, *ACM CCS 08*, pages 257–266, Alexandria, Virginia, USA, October 27–31, 2008. ACM Press.

[53] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

[54] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In Ian Goldberg, editor, *19th USENIX Security Symposium*, Washington, D.C., USA, August 11–13, 2010. USENIX Association.

[55] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 416–432, San Francisco, CA, USA, February 27 – March 2, 2012. Springer, Heidelberg, Germany.

[56] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 578–590, Vienna, Austria, October 24–28, 2016. ACM Press.

[57] Thomas P. Jakobsen, Marc X. Makkes, and Janus Dam Nielsen. Efficient implementation of the Orlandi protocol. In Jianying Zhou and Moti Yung, editors, *ACNS 10*, volume 6123 of *LNCS*, pages 255–272, Beijing, China, June 22–25, 2010. Springer, Heidelberg, Germany.

[58] 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, Berlin, Germany, November 4–8, 2013. ACM Press.

[59] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.

[60] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343, Accra Beach, Barbados, February 23–26, 2009. Springer, Heidelberg, Germany.

[61] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel,

Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 805–817, Vienna, Austria, October 24–28, 2016. ACM Press.

[62] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 591–602, Denver, CO, USA, October 12–16, 2015. ACM Press.

[63] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[64] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513. ACM, 1990.

[65] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.

[66] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Extracting correlations. In *50th FOCS*, pages 261–270, Atlanta, GA, USA, October 25–27, 2009. IEEE Computer Society Press.

[67] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In Moni Naor, editor, *EUROCRYPT 2007*, volume

4515 of *LNCS*, pages 97–114, Barcelona, Spain, May 20–24, 2007. Springer, Heidelberg, Germany.

[68] Payman Mohassel, Ostap Orobets, and Ben Riva. Efficient server-aided 2pc for mobile phones. *Proc. Privacy Enhancing Technologies*, (2):82–99, 2016.

[69] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

[70] 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, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[71] 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, Guadalajara, Mexico, August 23–26, 2015. Springer, Heidelberg, Germany.

[72] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 398–415, Amalfi, Italy, September 3–5, 2014. Springer, Heidelberg, Germany.

[73] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling

revisited. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[74] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. http://eprint.iacr.org/2015/472.

[75] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[76] 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, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[77] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18(3):247–287, July 2005.

[78] 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, Amalfi, Italy, September 5–7, 2012. Springer, Heidelberg, Germany.

[79] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *1st ACM Conference on Electronic Commerce*, 1999.

[80] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 567–578, Denver, CO, USA, October 12–16, 2015. ACM Press.

[81] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[82] Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Actively secure garbled circuits with constant communication overhead in the plain model. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 3–39, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[83] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup.

In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 2087–2104, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[84] Aner Ben-Efraim. On multiparty garbling of arithmetic circuits. Cryptology ePrint Archive, Report 2017/1186, 2017. https://eprint.iacr.org/2017/1186.