Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation

Xiao Wang
University of Maryland
wangxiao@cs.umd.edu

Samuel Ranellucci
University of Maryland
George Mason University
samuel@umd.edu

Jonathan Katz
University of Maryland
jkatz@cs.umd.edu

ABSTRACT

We propose a simple and efficient framework for obtaining efficient constant-round protocols for maliciously secure two-party computation. Our framework uses a function-independent preprocessing phase to generate authenticated information for the two parties; this information is then used to construct a *single* "authenticated" garbled circuit which is transmitted and evaluated. We also show how to efficiently instantiate the preprocessing phase with a new, highly optimized version of the TinyOT protocol by Nielsen et al.

Our protocol outperforms existing work in both the single-execution and amortized settings, with or without preprocessing:

- In the single-execution setting, our protocol evaluates an AES circuit with malicious security in 37 ms with an online time of 1 ms. Previous work with the best overall time requires 62 ms (with 14 ms online time); previous work with the best online time (also 1 ms) requires 124 ms overall.
- If we amortize over 1024 executions, each AES computation requires just 6.7 ms with roughly the same online time as above. The best previous work in the amortized setting has roughly the same total time but does not support function-independent preprocessing.

Our work shows that the performance penalty for maliciously secure two-party computation (as compared to semi-honest security) is much smaller than previously believed.

CCS CONCEPTS

• **Theory of computation → Cryptographic protocols**;

KEYWORDS

Two-party Computation; Secure Computation; Garbled Circuit

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134053

1 INTRODUCTION

Protocols for secure two-party computation (2PC) allow two parties to compute an agreed-upon function of their inputs without revealing anything additional to each other. Although originally viewed as impractical, protocols for generic 2PC in the semi-honest setting based on Yao's garbled-circuit protocol [49] have seen tremendous efficiency improvements over the past several years [2, 5, 17, 25, 27, 36, 42, 50].

While these results are impressive, semi-honest security—which assumes that both parties follow the protocol honestly yet may try to learn additional information from the execution—is clearly not sufficient for all applications. This has motivated researchers to construct protocols achieving the stronger notion of *malicious* security. One popular approach for designing constant-round maliciously secure protocols is to apply the "cut-and-choose" technique [1, 6, 18, 28–31, 44, 45, 47] to Yao's garbled-circuit protocol. For statistical security $2^{-\rho}$, the best approaches using this paradigm require ρ garbled circuits (which is optimal); the most efficient instantiation of this approach, by Wang et al. [47], securely evaluates an AES circuit in 62 ms.

The cut-and-choose approach incurs significant overhead when large circuits are evaluated 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) [19, 33, 34, 43]. When amortizing over τ executions, only $O(\frac{\rho}{\log \tau})$ garbled circuits are needed per execution. Rindal and Rosulek [43] report an amortized time of 6.4 ms to evaluate an AES circuit, where amortization is over 1024 executions. More recently, Nielsen and Orlandi [41] proposed a protocol with *constant* amortized overhead, but only when τ is at least the number of gates in the circuit. Also, their protocol allows for amortization only over *parallel* executions, whereas the works cited above allow amortization even over *sequential* executions, where inputs to the different executions need not be known all at once.

Other techniques for constant-round, maliciously secure two-party computation, with asymptotically better performance than cut-and-choose in the single-execution setting, have also been explored. The LEGO protocol [40] and subsequent optimizations [13, 14, 26, 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 LEGO implementation [38] still has a higher end-to-end running time than the best protocol based on the cut-and-choose approach applied at the garbled-circuit level.

AES Evaluation (2.1 ms in the semi-honest setting)							
	Single-Execution Setting			Amortized Setting (1024 executions)			
	[38]	[47]	**This paper**	[34]	[43]	[38]	**This paper**
Function-ind. phase	89.6 ms	-	10.9 ms	-	-	13.84 ms	4.9 ms
Function-dep. phase	13.2 ms	28 ms	4.78 ms	74 ms	5.1 ms	0.74 ms	0.53 ms
Online	1.46 ms	14 ms	0.93 ms	7 ms	1.3 ms	1.13 ms	1.23 ms
Total	104.26 ms	42 ms	16.61 ms	81 ms	6.4 ms	15.71 ms	6.66 ms
SHA-256 Evaluation (9.6 ms in the semi-honest setting)							
	Single-Execution Setting			Amortized Setting (1024 executions)			
	[38]	[47]	**This paper**	[34]	[43]	[38]	**This paper**
Function-ind. phase	478.5 ms	-	96 ms	-	-	183.5 ms	64.8 ms
Function-dep. phase	164.4 ms	350 ms	51.7 ms	206 ms	48 ms	11.7 ms	8.7 ms
Online	11.2 ms	84 ms	9.3 ms	33 ms	8.4 ms	9.6 ms	11.3 ms
Total	654.1 ms	434 ms	157 ms	239 ms	56.4 ms	204.8 ms	84.8 ms

Table 1: Constant-round 2PC protocols with malicious security. All timings are based on statistical security 2^{-40} and are benchmarked using Amazon EC2 `c4.8xlarge` instances over a LAN, averaged over 10 executions. Single-execution times do not include the base-OTs, which require the same time (~20 ms) for all protocols. Timings for the semi-honest protocol are based on the same garbling code used in our protocol, and also do not include the base-OTs. See Section 8 for more details.

The Beaver-Micali-Rogaway compiler [4] provides yet another way to construct constant-round protocols with malicious security [7, 9]. This compiler uses an "outer" secure-computation protocol to generate a garbled circuit that is then evaluated. Lindell et al. [32, 35] suggested applying this idea using SPDZ [12] (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 [39] extends the classical GMW protocol [15] by adding information-theoretic MACs to shares held by the parties; The IPS protocol [21] has excellent asymptotic complexity, but its concrete complexity is unclear since it has never been implemented (and appears quite difficult to implement). We remark that the end-to-end times of these protocols suffer significantly due to the large round complexity: even over a LAN, each communication round requires at least 0.5 ms; for evaluating an AES circuit (with a depth of about 50), this means that the time for any linear-round protocol will be at least 25 ms. The situation will be even worse over a WAN.

In Tables 1 and 2, we summarize the efficiency of various constant-round 2PC protocols with malicious security. Table 1 gives the performance of state-of-the-art implementations under fixed hardware and network conditions, while Table 2 reports the asymptotic complexity of various approaches. Following [38], we consider executions that take place in three phases:

- **Function-independent preprocessing.** In this phase, the parties need not know their inputs or the function to be computed (beyond an upper bound on the number of gates).
- **Function-dependent preprocessing.** In this phase, the parties know what function they will compute, but do not need to know their inputs.

  Often, the first two phases are combined and referred to simply as the *offline* or *preprocessing* phase.
- **Online phase.** In this phase, the parties evaluate the agreed-upon function on their respective inputs.

1.1 Our Contributions

We propose a new approach for constructing constant-round, maliciously secure 2PC protocols with extremely high efficiency. At a high level (further details are in Section 3), and following ideas of Nielsen et al. [39], our protocol uses a function-independent preprocessing phase to realize an ideal functionality that we call $\mathcal{F}_{\mathsf{Pre}}$. This preprocessing phase is used to set up correlated randomness between the two parties that 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× 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 [39]), described in Section 5. 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 Table 1

Protocol	Function-ind. (Comm./Comp.)	Function-dep. (Comm./Comp.)	Online (Comm.)	Online (Comp.) / Storage
Cut-and-choose [1, 29, 47]	—	$O(\|C\|\rho)$	$O(\|\mathcal{I}\|\rho)$	$O(\|C\|\rho)$
Amortized [19, 33]	—	$O\left(\frac{\|C\|\rho}{\log \tau}\right)$	$O\left(\frac{\|\mathcal{I}\|\rho}{\log \tau}\right)$	$O\left(\frac{\|C\|\rho}{\log \tau}\right)$
LEGO [13, 40]	$O\left(\frac{\|C\|\rho}{\log \tau + \log \|C\|}\right)$	$O(\|C\|)$	$O(\|\mathcal{I}\| + \|\mathcal{O}\|)$	$O\left(\frac{\|C\|\rho}{\log \tau + \log \|C\|}\right)$
SPDZ-BMR [24, 32]*	$O(\|C\|\kappa)$	$O(\|C\|)$	$O(\|\mathcal{I}\| + \|\mathcal{O}\|)$	$O(\|C\|)$
This paper (with Section 5)	$O\left(\frac{\|C\|\rho}{\log \tau + \log \|C\|}\right)$	$O(\|C\|)$	$\|\mathcal{I}\| + \|\mathcal{O}\|$	$O(\|C\|)$
This paper (with [21])	$O(\|C\|)$			

Table 2: Asymptotic complexity of constant-round 2PC protocols with malicious security. $|C|$, $|\mathcal{I}|$, and $|\mathcal{O}|$ are the circuit size, input size, and output size respectively; low-order terms independent of these parameters are ignored. The statistical security parameter is ρ, the computational security parameter is κ, and τ is the number of protocol executions in the amortized setting. Communication (Comm.) is measured as the number of symmetric-key ciphertexts, and computation (Comp.) is measured as the number of symmetric-key operations. "Storage" is the number of symmetric-key ciphertexts generated by the offline stage.
*Although the complexity of function-independent preprocessing can be reduced to $O(|C|)$ using somewhat homomorphic encryption [12], doing so requires a number of *public-key* operations proportional to $|C|$.

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

- Upon receiving Δ_{A} from P_{A} and init from P_{B}, and assuming no values Δ_{A}, Δ_{B} are currently stored, choose uniform $\Delta_{\mathsf{B}} \in \{0,1\}^{\rho}$ and store Δ_{A}, Δ_{B}. Send Δ_{B} to P_{B}.
- Upon receiving (random, r, $\mathsf{M}[r]$, $\mathsf{K}[s]$) from P_{A} and random from P_{B}, sample uniform $s \in \{0,1\}$ and set $\mathsf{K}[r] := \mathsf{M}[r] \oplus r\Delta_{\mathsf{B}}$ and $\mathsf{M}[s] := \mathsf{K}[s] \oplus s\Delta_{\mathsf{A}}$. Send $(s, \mathsf{M}[s], \mathsf{K}[r])$ to P_{B}.
- Upon receiving $(\mathtt{AND}, (r_1, \mathsf{M}[r_1], \mathsf{K}[s_1]), (r_2, \mathsf{M}[r_2], \mathsf{K}[s_2]), r_3, \mathsf{M}[r_3], \mathsf{K}[s_3])$ from P_{A} and $(\mathtt{AND}, (s_1, \mathsf{M}[s_1], \mathsf{K}[r_1]), (s_2, \mathsf{M}[s_2], \mathsf{K}[r_2]))$ from P_{B}, verify that $\mathsf{M}[r_i] = \mathsf{K}[r_i] \oplus r_i\Delta_{\mathsf{B}}$ and that $\mathsf{M}[s_i] = \mathsf{K}[s_i] \oplus s_i\Delta_{\mathsf{A}}$ for $i \in \{1,2\}$ and send cheat to P_{B} if not. Otherwise, set $s_3 := r_3 \oplus ((r_1 \oplus s_1) \wedge (r_2 \oplus s_2))$, set $\mathsf{K}[r_3] := \mathsf{M}[r_3] \oplus r_3\Delta_{\mathsf{B}}$, and set $\mathsf{M}[s_3] := \mathsf{K}[s_3] \oplus s_3\Delta_{\mathsf{A}}$. Send $(s_3, \mathsf{M}[s_3], \mathsf{K}[r_3])$ to P_{B}.

Figure 1: The preprocessing functionality, assuming P_{A} is corrupted. (If P_{B} is corrupted, the functionality is defined symmetrically. If neither party is corrupted, the functionality is adapted in the obvious way.)

and Section 8). For example, it requires only 16.6 ms total to evaluate AES, a 6× improvement compared to a recent implementation of a LEGO-style approach [38]. Furthermore, the storage needed by our protocol is asymptotically smaller (see Table 2), 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 5 yields a protocol with the best concrete efficiency, and is the main focus of this paper. However, we note that our framework can also be instantiated in other ways:

- When $\mathcal{F}_{\mathsf{Pre}}$ is instantiated using the IPS compiler [21] and the bit-OT protocol by Ishai et al. [20], we obtain a maliciously secure constant-round 2PC protocol with total communication complexity $O(|C|\kappa)$. 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 [22] 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 $O(|C|\kappa)$. Previous work in the same model [37] achieves the same complexity but with number of rounds proportional to the circuit depth.

The results described in this paper—both the idea of constructing an "authenticated" garbled circuit as well as the efficient TinyOT protocol we developed—have already found application in subsequent work [16, 48] on constant-round *multiparty* computation with malicious security.

2 NOTATION AND PRELIMINARIES

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 2PC protocol by P_{A} and 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 P_{A}'s input wires, $\mathcal{I}_2$ to denote the set of indices of 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.

2.1 Information-theoretic MACs

We use the information-theoretic message authentication codes (IT-MACs) of [39], which we briefly recall. P_{A} holds a uniform

global key $\Delta_A \in \{0,1\}^\kappa$. A bit b known by P_B is authenticated by having P_A hold a uniform key $K[b]$ and having P_B hold the corresponding tag $M[b] := K[b] \oplus b\Delta_A$. Symmetrically, P_B holds an independent global key Δ_B; a bit b known by P_A is authenticated by having P_B hold a uniform key $K[b]$ and having P_A hold the tag $M[b] := K[b] \oplus b\Delta_B$. We use $[b]_A$ to denote an authenticated bit known to P_A (i.e., $[b]_A$ means P_A holds $(b, M[b])$ and P_B holds $K[b]$), and define $[b]_B$ symmetrically.

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

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 P_A hold $(r, M[r], K[s])$ and P_B hold $(s, M[s], K[r])$. It can be observed that this scheme is also XOR-homomorphic.

As described in the previous section, we use a preprocessing phase that realizes a stateful functionality $\mathcal{F}_{\text{Pre}}$ defined in Figure 1. 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 random bit $\lambda = r \oplus s$. (The adversary may choose the "random values" it receives, but this does not reveal anything about $r \oplus s$ or the other party's global key to the adversary.) 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. In the next section we describe our protocol assuming some way of realizing $\mathcal{F}_{\text{Pre}}$; we defer until Section 5 a discussion of how $\mathcal{F}_{\text{Pre}}$ can be realized.

3 PROTOCOL INTUITION

We give a high-level overview of our protocol in the $\mathcal{F}_{\text{Pre}}$-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 (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 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, P_B) on that wire will be $\hat{x} = x \oplus \lambda_\alpha$. Using the free-XOR technique [27], each wire α is also associated with two labels $L_{\alpha,0}$ and $L_{\alpha,1} := L_{\alpha,0} \oplus \Delta$ known to P_A. If the masked bit on that wire is $\hat{x}$, then P_B learns $L_{\alpha,\hat{x}}$.

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	$\hat{z}_{00} = (\lambda_\alpha \wedge \lambda_\beta) \oplus \lambda_\gamma$	$H(L_{\alpha,0}, L_{\beta,0}, \gamma, 00) \oplus (\hat{z}_{00}, L_{\gamma,\hat{z}_{00}})$
0	1	$\hat{z}_{01} = (\lambda_\alpha \wedge \overline{\lambda_\beta}) \oplus \lambda_\gamma$	$H(L_{\alpha,0}, L_{\beta,1}, \gamma, 01) \oplus (\hat{z}_{01}, L_{\gamma,\hat{z}_{01}})$
1	0	$\hat{z}_{10} = (\overline{\lambda_\alpha} \wedge \lambda_\beta) \oplus \lambda_\gamma$	$H(L_{\alpha,1}, L_{\beta,0}, \gamma, 10) \oplus (\hat{z}_{10}, L_{\gamma,\hat{z}_{10}})$
1	1	$\hat{z}_{11} = (\overline{\lambda_\alpha} \wedge \overline{\lambda_\beta}) \oplus \lambda_\gamma$	$H(L_{\alpha,1}, L_{\beta,1}, \gamma, 11) \oplus (\hat{z}_{11}, L_{\gamma,\hat{z}_{11}})$

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

The standard garbled circuit just described ensures security against a malicious P_B, since (intuitively) P_B learns no information about the true values on any of the wires. Unfortunately, it provides no security against a malicious P_A who can potentially cheat by corrupting rows in the various garbled tables. One particular attack P_A can carry out is a *selective-failure* attack. Say, for example, that a malicious P_A corrupts only the $(0,0)$-th row of the garbled table for the gate above, and assume P_B aborts if it detects an error during evaluation. If P_B aborts, then P_A learns that the masked values on the input wires of the gate above were $\hat{x} = \hat{y} = 0$, 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 P_A. (In that case, even if P_B aborts and P_A learns the masked wire values, P_A learns nothing about the true wire values.) Since knowledge of the garbled table would leak information about the masks to P_A, the garbled table must be hidden from P_A as well. That is, we now want to set up a situation in which P_A and P_B hold *secret shares* of the garbled table, as follows:

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

(Here, e.g., $L^A_{\gamma,00}, L^B_{\gamma,00}$ represent abstract XOR-shares of $L_{\gamma,\hat{z}_{00}}$, i.e., $L_{\gamma,\hat{z}_{00}} = L^A_{\gamma,00} \oplus L^B_{\gamma,00}$.) Once P_A sends its shares of all the garbled gates, 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 P_A since (intuitively) the results of any changes P_A makes to the garbled circuit are *independent* of P_B's inputs. However, 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 P_A's share of the masked bit. The shares of the garbled table now take the following form:

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

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

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

$x \oplus \lambda_\alpha$	$y \oplus \lambda_\beta$	$\mathsf{P_A}$'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	$H(\mathsf{L}_{\alpha,1}, \mathsf{L}_{\beta,1}, \gamma, 11) \oplus (r_{11}, \mathsf{M}[r_{11}], \mathsf{L}_{\gamma,0} \oplus r_{11}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{11}])$	$(s_{11} = \hat{z}_{11} \oplus r_{11}, \mathsf{K}[r_{11}], \mathsf{M}[s_{11}])$

Table 3: Our final construction of an authenticated garbled table for an AND gate.

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 $\Delta = \Delta_\mathsf{A}$ then we can secret share, e.g., $\mathsf{L}_{\gamma,\hat{z}_{00}}$ as

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

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, and then 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, shares of the table become easy to compute in a distributed fashion.

Another final 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.

4 OUR MAIN FRAMEWORK

In Figure 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 7 we describe how arbitrary values of ρ can be supported. Note that the calls to $\mathcal{F}_{\mathsf{Pre}}$ can be performed in parallel, so the protocol runs in constant rounds. Moreover, we show later that $\mathcal{F}_{\mathsf{Pre}}$ can be instantiated efficiently 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 [3].

4.1 Proof of Security

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

Theorem 4.1. *If H is modeled as a random oracle, the protocol in Figure 2 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 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 continue to $\mathcal{F}$ (and so $\mathsf{P_B}$ outputs $f(x, y)$).
The distributions on the view of $\mathcal{A}$ in **Hybrid$_1$** and **Hybrid$_2$** are identical. Lemma 4.2 shows that $\mathsf{P_B}$ generates the same output in both experiments except with probability at most $2^{-\rho}$.

Hybrid$_3$. Same as **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 **Hybrid$_3$** and **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 **Hybrid$_3$** and **Hybrid$_2$**. So it only remains to show that $\mathsf{P_B}$ aborts with the same probability in both experiments.

Protocol Π_{2pc}

Inputs: In the function-dependent phase, the parties agree on a circuit for a function $f : \{0,1\}^{|\mathcal{I}_1|} \times \{0,1\}^{|\mathcal{I}_2|} \to \{0,1\}^{|O|}$.
In the input-processing 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 (and, $(r_\alpha, \mathsf{M}[r_\alpha], \mathsf{K}[s_\alpha])$, $(r_\beta, \mathsf{M}[r_\beta], \mathsf{K}[s_\beta])$) (resp., (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 $s_\sigma \oplus r_\sigma = \lambda_\alpha \wedge \lambda_\beta$.

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

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

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

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

(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 $\mathsf{P_B}$:

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

Input processing:

(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 $w \in O$, $\mathsf{P_A}$ 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 2: Our protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. Here $\rho = \kappa$ for clarity, but this is not necessary (cf. Section 7).

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 $w \in O$, 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 $w \in O$. Finally, for each $w \in O$, $\mathcal{S}$ 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 4.2. *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.
- $T = \wedge$: Here $z^*_\gamma = z^*_\alpha \wedge z^*_\beta$. Assuming $\mathsf{P_B}$ does not abort, the only way $\mathsf{P_B}$ 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 $w \in O$. It follows that if $\mathsf{P_B}$ does not abort, it outputs z^* unless $\mathcal{A}$ forges an IT-MAC. □

5 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) for generating authenticated bits using uniform values of $\Delta_{\mathsf{A}}, \Delta_{\mathsf{B}} \in \{0,1\}^\kappa$ that are preserved across executions [38, 39].

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 $\mathsf{P_A}$ and Δ_{B} to $\mathsf{P_B}$.
(2) Upon receiving (random, 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 $\mathsf{P_A}$ and $\mathsf{K}[x]$ to $\mathsf{P_B}$.
(3) Upon receiving (random, B) from both parties, generate an authenticated bit for $\mathsf{P_B}$ 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: The authenticated-bit functionality.

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 5.2) 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×.

5.1 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 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 $x_1y_2 \oplus x_2y_1$. (Note that the parties can then locally compute x_1y_1 and x_2y_2, respectively, and thus generate shares of $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$.) In Figure 5 we show a protocol that realizes $\mathcal{F}_{\mathsf{HaAND}}$ in the $\mathcal{F}_{\mathsf{abit}}$-hybrid model.

Lemma 5.1. *If H is modeled as a random oracle, the protocol in Figure 5 securely implements $\mathcal{F}_{\mathsf{HaAND}}$ in the $\mathcal{F}_{\mathsf{abit}}$-hybrid model.*

Proof. We first show correctness. Note that $s_2 = s_1 \oplus x_2y_1$, so $s_1 \oplus s_2 = x_2y_1$. Similarly, $t_1 \oplus t_2 = x_1y_2$. Thus, v_1 and v_2 are shares of $x_1y_2 \oplus x_2y_1$. 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

$$\begin{aligned} s_2 \oplus t_1 &= s'_{x_2} \oplus (t'_1 \oplus x_1y_2) \\ &= (1 \oplus x_2)s'_0 \oplus x_2s'_1 \oplus t'_1 \oplus x_1y_2 \\ &= s'_0 \oplus x_2(s'_0 \oplus s'_1) \oplus t'_1 \oplus x_1y_2, \end{aligned}$$

which matches the value

$$v_1 \oplus (x_1y_2 \oplus x_2y_1) = (s'_0 \oplus t'_1) \oplus x_1y_2 \oplus x_2(s'_0 \oplus s'_1)$$

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

5.2 Leaky AND Triples

The leaky-AND functionality $\mathcal{F}_{\mathsf{LaAND}}$ is described in Figure 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 $x_1y_2 \oplus x_2y_1$. The parties can then locally compute shares of $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2)$. Note that

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 $v_2 := v_1 \oplus (x_1y_2 \oplus x_2y_1)$ to $\mathsf{P_B}$.

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

Figure 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 5: Protocol Π_{HaAND} realizing $\mathcal{F}_{\mathsf{HaAND}}$.

$\mathsf{P_A}$ (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 $x_1 \oplus x_2 = 0$.

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

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

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 6: 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 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 7: Protocol Π_{LaAND} realizing $\mathcal{F}_{\mathsf{LaAND}}$.

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 7; the proof of security is in Appendix A.

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

Honest case: Generate uniform $[x_1]_A$, $[y_1]_A$, $[z_1]_A$, and $[x_2]_B$, $[y_2]_B$, $[z_2]_B$, 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 8: Functionality $\mathcal{F}_{\mathsf{aAND}}$ for generating AND triples

Protocol Π_{aAND}

Protocol:

(1) P_A and P_B call $\mathcal{F}_{\mathsf{LaAND}}$ a total of $\ell' = \ell B$ times to obtain $\{[x_1^i]_A, [y_1^i]_A, [z_1^i]_A, [x_2^i]_B, [y_2^i]_B, [z_2^i]_B\}_{i=1}^{\ell'}$.

(2) P_A and 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 $([x_1']_A, [y_1']_A, [z_1']_A, [x_2']_B, [y_2']_B, [z_2']_B)$ and $([x_1'']_A, [y_1'']_A, [z_1'']_A, [x_2'']_B, [y_2'']_B, [z_2'']_B)$ 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 $[x_1]_A := [x_1']_A \oplus [x_1'']_A$, $[x_2]_B := [x_2']_B \oplus [x_2'']_B$, $[y_1]_A := [y_1']_A$, $[y_2]_B := [y_2']_B$, $[z_1]_A := [z_1']_A \oplus [z_1'']_A \oplus d[x_1'']_A$, $[z_2]_B := [z_2']_B \oplus [z_2'']_B \oplus d[x_2'']_B$.

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

Figure 9: Protocol Π_{aAND} realizing $\mathcal{F}_{\mathsf{aAND}}$.

5.3 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. [39]. Given two potentially leaky AND triples

$$\left([x_1']_A, [y_1']_A, [z_1']_A, [x_2']_B, [y_2']_B, [z_2']_B\right)$$

and

$$\left([x_1'']_A, [y_1'']_A, [z_1'']_A, [x_2'']_B, [y_2'']_B, [z_2'']_B\right),$$

we set $[x_1]_A := [x_1']_A \oplus [x_1'']_A$, $[x_2]_B := [x_2']_B \oplus [x_2'']_B$. Note that the result is non-leaky as long as one of the original triples is non-leaky. We can also set $[y_1]_A := [y_1']_A$, $[y_2]_B := [y_2']_B$ 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

$$\begin{aligned}
(x_1 \oplus x_2)(y_1 \oplus y_2) &= (x_1' \oplus x_2' \oplus x_1'' \oplus x_2'')(y_1' \oplus y_2') \\
&= (x_1' \oplus x_2')(y_1' \oplus y_2') \oplus (x_1'' \oplus x_2'')(y_1' \oplus y_2') \\
&= (x_1' \oplus x_2')(y_1' \oplus y_2') \oplus (x_1'' \oplus x_2'')(y_1'' \oplus y_2'') \\
&\quad \oplus (x_1'' \oplus x_2'')(y_1' \oplus y_2' \oplus y_1'' \oplus y_2'') \\
&= (z_1' \oplus z_2') \oplus (z_1'' \oplus z_2'') \oplus d(x_1'' \oplus x_2'') \\
&= (z_1' \oplus z_1'' \oplus dx_1'') \oplus (z_2' \oplus z_2'' \oplus dx_2'').
\end{aligned}$$

Therefore, we can just set $[z_1]_A := [z_1']_A \oplus [z_1'']_A \oplus d[x_1'']_A$ and $[z_2]_B := [z_2']_B \oplus [z_2'']_B \oplus d[x_2'']_B$. (This corresponds to the protocol in Figure 9.)

6 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 [21]. 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 $O(\ell) + poly(\kappa, d, \log \ell)$ bits. In our setting, this translates to a communication complexity of $O(|C|\kappa) + poly(\kappa, \log |C|)$ 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. [37], 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.

7 EXTENSIONS AND OPTIMIZATIONS

Handling $\kappa \neq \rho$. In Figure 2 step 4d, all MACs that 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 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 [42]. 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 7, but we note here several optimizations:

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

Table 4: Fewest AND gates needed for bucketing, for different bucket sizes and statistical security parameters.

Circuit	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{O}$	$\lvert C \rvert$
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

Table 5: Circuits used in our evaluation.

(1) For clarity, in Figure 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.
(2) Since efficiency depends on the bucket size $B = \rho / \log |C|$, 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 4 shows the fewest AND gates needed in order to use different bucket sizes (B), for different values of ρ.

8 EVALUATION

8.1 Implementation and Evaluation Setup

Our implementation uses the EMP-toolkit [46], and is publicly available as a part of it.

In our evaluation, we set the computational security parameter to $\kappa = 128$ and the statistical security parameter to $\rho = 40$. In Figure 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. [5]. We use Multi-threading, 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 [39]. In our implementation we adopt the OT-extension protocol of Keller et al. [23] along with the optimizations of Nielsen et al. [38]. The resulting protocol requires $\kappa + \rho$ bits of communication per authenticated bit.
(2) $\mathcal{F}_{\text{Pre}}$ **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 sub-routines 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 8.2, we first compare the performance of our protocol with previous protocols in similar settings, focusing on three circuits (AES, SHA-1, and 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 8.3 we also explore the performance of our protocol on some larger circuits. (These circuits are available in [46].) Parameters for all the circuits we study are given in Table 5. In Sections 8.4 and 8.5, we study the scalability of our protocol and compare its concrete communication complexity with prior work.

8.2 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. [47], 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. [38], 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 the time for the base-OTs for the same reason as in [38]: 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 [8]) takes about 20 ms in the LAN setting and 240 ms in the WAN setting.

As shown in Table 6, 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× better in the LAN setting, and 1.3-1.5× better in the WAN setting.

Compared to the recent (unimplemented) work of Lindell et al. [32], 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

	LAN				WAN			
	Ind. Phase	Dep. Phase	Online	Total	Ind. Phase	Dep. Phase	Online	Total
AES [47]	-	28 ms	14 ms	42 ms	-	425 ms	416 ms	841 ms
AES [38]	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 [47]	-	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 [47]	-	350 ms	84 ms	434 ms	-	2997 ms	514 ms	3511 ms
SHA256 [38]	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 6: Comparison in the single-execution setting

		LAN				WAN			
	τ	Ind. Phase	Dep. Phase	Online	Total	Ind. Phase	Dep. Phase	Online	Total
[43]	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
[38]	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

Table 7: Comparison in the amortized setting. All experiments evaluate AES, with τ the number of executions being amortized over.

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

Table 8: Experimental results for larger circuits.

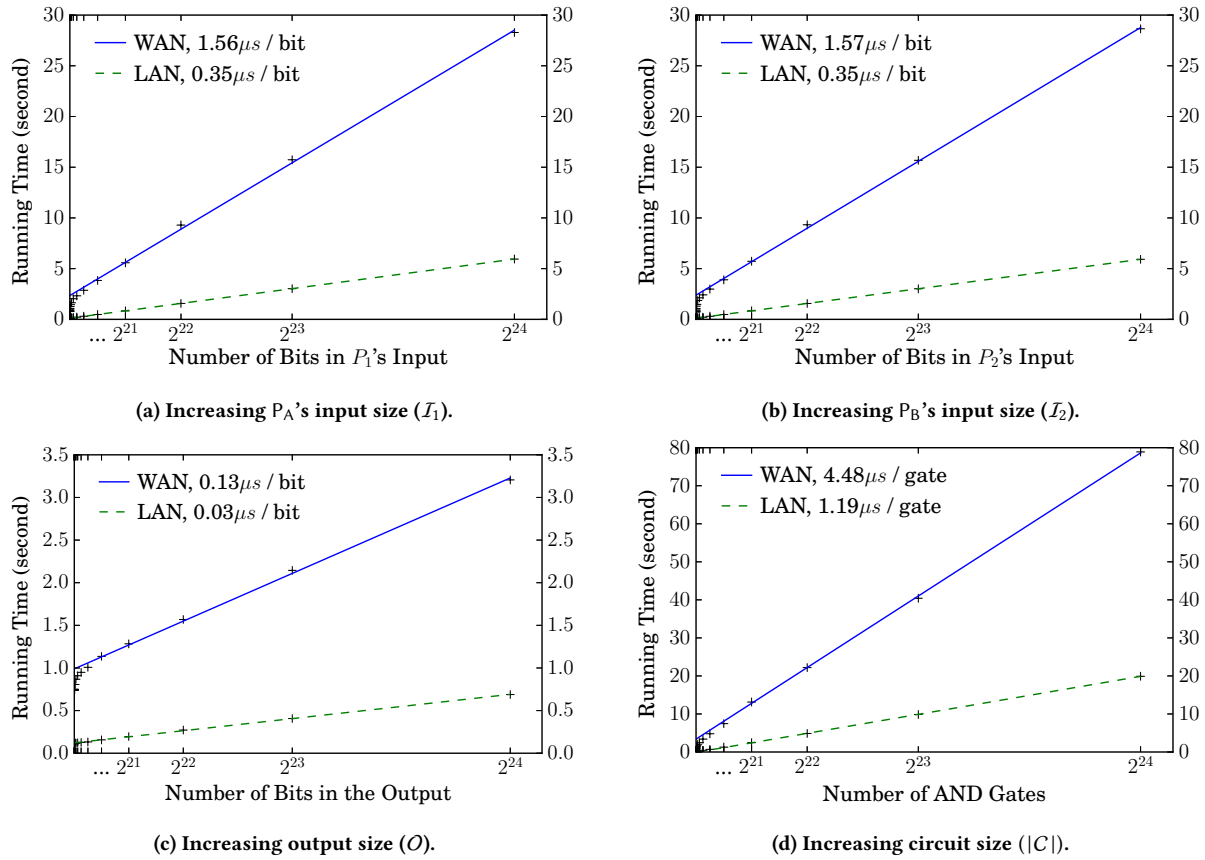
underlying circuit, while we need only one TinyOT-style AND computation per AND gate.

We perform a back-of-the-envelope calculation to compare the relative efficiency of our protocol and that of Lindell et al. [32]. Over a 10 Gbps network, the recent work of Keller et al. [24] 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 8.4). Therefore, our protocol is at least 75× better than the best possible implementation of their protocol.

Comparison with linear-round protocols. The AES circuit has depth 50 [34]. 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. [10] 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. [11] 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 0.47μs 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



(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 ($|C|$).

Figure 10: Scalability of our protocol. Initially $|\mathcal{I}_1| = |\mathcal{I}_2| = |\mathcal{O}| = 128$ and $|C| = 1024$, and then one of those parameters is allowed to grow while the others remain fixed. The total running time is reported.

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 7, our protocol offers a consistent improvement as compared to the best prior work of Nielsen et al. [38] and Rindal and Rosulek [43].

8.3 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 5.

In Table 8 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 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.

Protocol	τ	Ind. Phase	Dep. Phase	Online
[43]	302	-	3.8 MB	25.8 KB
	128	-	2.5 MB	21.3 KB
	1024	-	1.6 MB	17.0 KB
[38]	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

Table 9: 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 [43] should be doubled for a fair comparison.

8.4 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 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 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 [47]) 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.

8.5 Communication Complexity

In Table 9, 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× 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× smaller than in the protocol of Nielsen et al. and 3.5–5.3× smaller than in the protocol of Rindal and Rosulek.

ACKNOWLEDGMENTS

This material is based on work supported by NSF awards #1111599, #1563722, and #1564088. The authors would like to thank Roberto Trifiletti, Yan Huang, and Ruiyu Zhu for their helpful comments.

REFERENCES

[1] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. 2014. Non-Interactive Secure Computation Based on Cut-and-Choose. In *Eurocrypt 2014 (LNCS)*, Vol. 8441. 387–404.
[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS 2013*. 535–548.
[3] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Crypto'91 (LNCS)*, Vol. 576. 420–432.
[4] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The Round Complexity of Secure Protocols. In *ACM STOC*. 503–513.
[5] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE Symposium on Security & Privacy*. 478–492.
[6] Luís T. A. N. Brandão. 2013. Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique. In *ASIACRYPT 2013, Part II (LNCS)*, Vol. 8270. 441–463.
[7] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. 2014. Efficient Three-Party Computation from Cut-and-Choose. In *Crypto 2014, Part II (LNCS)*, Vol. 8617. 513–530.
[8] Tung Chou and Claudio Orlandi. 2015. The Simplest Protocol for Oblivious Transfer. In *LATINCRYPT 2015 (LNCS)*, Vol. 9230. 40–58.
[9] Ivan Damgård and Yuval Ishai. 2005. Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In *Crypto 2005 (LNCS)*, Vol. 3621. 378–394.
[10] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. 2014. An Empirical Study and Some Improvements of the MiniMac Protocol for Secure Computation. In *Intl. Conf. on Security and Cryptography for Networks (LNCS)*, Vol. 8642. 398–415.
[11] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2017. The TinyTable protocol for 2-Party Secure Computation, or: Gate-scrambling Revisited. In *Crypto 2017, Part I (LNCS)*, Vol. 10401. 167–187.
[12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Crypto 2012 (LNCS)*, Vol. 7417. 643–662.
[13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. 2013. MiniLEGO: Efficient Secure Two-Party Computation from General Assumptions. In *Eurocrypt 2013 (LNCS)*, Vol. 7881. 537–556.
[14] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. 2015. TinyLEGO: An Interactive Garbling Scheme for Maliciously Secure Two-Party Computation. Cryptology ePrint Archive, Report 2015/309. (2015). http://eprint.iacr.org/2015/309.
[15] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game, or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*. 218–229.
[16] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. 2017. Low Cost Constant Round MPC Combining BMR and Oblivious Transfer. Cryptology ePrint Archive, Report 2017/214. (2017). To appear in Asiacrypt 2017.
[17] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security 2011*.
[18] Yan Huang, Jonathan Katz, and David Evans. 2013. Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose. In *Crypto 2013, Part II (LNCS)*, Vol. 8043. 18–35.
[19] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. 2014. Amortizing Garbled Circuits. In *Crypto 2014, Part II (LNCS)*,

Vol. 8617. 458–475.
[20] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2009. Extracting Correlations. In *IEEE FOCS*. 261–270.
[21] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. 2008. Founding Cryptography on Oblivious Transfer - Efficiently. In *Crypto 2008 (LNCS)*, Vol. 5157. 572–591.
[22] Stanislaw Jarecki and Vitaly Shmatikov. 2007. Efficient Two-Party Secure Computation on Committed Inputs. In *Eurocrypt 2007 (LNCS)*, Vol. 4515. 97–114.
[23] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. Actively Secure OT Extension with Optimal Overhead. In *Crypto 2015, Part I (LNCS)*, Vol. 9215. 724–741.
[24] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS 2016*. 830–842.
[25] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. 2014. FleXOR: Flexible Garbling for XOR Gates That Beats Free-XOR. In *Crypto 2014, Part II (LNCS)*, Vol. 8617. 440–457.
[26] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. 2017. DUPLO: Unifying Cut-and-Choose for Garbled Circuits. In *ACM CCS 2017*.
[27] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP 2008, Part II (LNCS)*, Vol. 5126. 486–498.
[28] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. 2012. Billion-Gate Secure Computation with Malicious Adversaries. In *USENIX Security 2012*.
[29] Yehuda Lindell. 2013. Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries. In *Crypto 2013, Part II (LNCS)*, Vol. 8043. 1–17.
[30] Yehuda Lindell and Benny Pinkas. 2007. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Eurocrypt 2007 (LNCS)*, Vol. 4515. 52–78.
[31] Yehuda Lindell and Benny Pinkas. 2011. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In *TCC 2011 (LNCS)*, Vol. 6597. 329–346.
[32] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. 2015. Efficient Constant Round Multi-party Computation Combining BMR and SPDZ. In *Crypto 2015, Part II (LNCS)*, Vol. 9216. 319–338.
[33] Yehuda Lindell and Ben Riva. 2014. Cut-and-Choose Yao-Based Secure Computation in the Online/Offline and Batch Settings. In *Crypto 2014, Part II (LNCS)*, Vol. 8617. 476–494.
[34] Yehuda Lindell and Ben Riva. 2015. Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In *ACM CCS 2015*. 579–590.
[35] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. 2016. More Efficient Constant-Round Multi-party Computation from BMR and SHE. In *TCC 2016-B, Part I (LNCS)*, Vol. 9985. 554–581.
[36] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay—A Secure Two-Party Computation System. In *USENIX Security 2004*.
[37] Payman Mohassel, Ostap Orobets, and Ben Riva. 2016. Efficient Server-Aided 2PC for Mobile Phones. *Proc. Privacy Enhancing Technologies* 2 (2016), 82–99.
[38] Jesper Nielsen, Thomas Schneider, and Roberto Trifiletti. 2017. Constant-Round Maliciously Secure 2PC with Function-Independent Preprocessing Using LEGO. In *Network and Distributed System Security Symposium (NDSS)*.
[39] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *Crypto 2012 (LNCS)*, Vol. 7417. 681–700.
[40] Jesper Buus Nielsen and Claudio Orlandi. 2009. LEGO for Two-Party Secure Computation. In *TCC 2009 (LNCS)*, Vol. 5444. 368–386.
[41] Jesper Buus Nielsen and Claudio Orlandi. 2016. Cross and Clean: Amortized Garbled Circuits with Constant Overhead. In *TCC 2016-B, Part I (LNCS)*, Vol. 9985. 582–603.
[42] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure Two-Party Computation Is Practical. In *ASIACRYPT 2009 (LNCS)*, Vol. 5912. 250–267.
[43] Peter Rindal and Mike Rosulek. 2016. Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution. In *USENIX Security 2016*.
[44] Abhi Shelat and Chih-Hao Shen. 2011. Two-Output Secure Computation with Malicious Adversaries. In *Eurocrypt 2011 (LNCS)*, Vol. 6632. 386–405.
[45] Abhi Shelat and Chih-Hao Shen. 2013. Fast Two-Party Secure Computation with Minimal Assumptions. In *ACM CCS 2013*. 523–534.
[46] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-Toolkit: Efficient Multiparty Computation Toolkit. https://github.com/emp-toolkit. (2016).
[47] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2017. Faster Secure Two-Party Computation in the Single-Execution Setting. In *Eurocrypt 2017, Part II (LNCS)*, Vol. 10211. 399–424.
[48] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. In *ACM CCS 2017*.
[49] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *IEEE FOCS*. 162–167.
[50] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole—Reducing Data Transfer in Garbled Circuits Using Half Gates. In *Eurocrypt 2015, Part II (LNCS)*, Vol. 9057. 220–250.

A 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 [39].

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:

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

We next prove security.

Lemma A.1. *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:

$$\begin{aligned} W_{0,0} &= H(\mathsf{M}[x_2]) \oplus T_{x_2} \oplus R \\ W_{0,0} &= H(\mathsf{M}[x_2]) \oplus R \\ &\quad \oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B}). \end{aligned}$$

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:

$$\begin{aligned} W_{0,1} &= H(\mathsf{M}[x_2]) \oplus T_{x_2} \oplus R \\ W_{0,1} &= H(\mathsf{M}[x_2]) \oplus R \\ &\quad \oplus H(\mathsf{K}[x_1], \mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B} \oplus \mathsf{K}[y_1] \oplus y_2\Delta_\mathsf{B}). \end{aligned}$$

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:

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

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:

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

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 A.2. *The protocol in Figure 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 P_A gets Δ_B: both R and one of $H(K[x_1])$ and $H(K[x_1] \oplus \Delta_B)$ are random.

Another difference is that 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 P_B. We construct a simulator $\mathcal{S}$ as follows:

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

$$(z_2, M[z_2]) := (r \oplus s, M[z_2] \oplus s\Delta_B),$$

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

2-3 $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{HaAND}}$ and obtains y_2' $\mathcal{A}$ sent. $\mathcal{S}$ also obtains d' sent by P_B. Denoting y_2', d as values an honest 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 P_A and P_B are switched. We denote S' the simulator that was defined for the case where 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 P_B having a chance to perform a selective failure attack similar to when P_A is malicious. If P_B flip y_2, it is guessing that $x_1 \oplus x_2 = 0$; if P_B flip y_2 and d, 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 P_A (with order of steps switched). □