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 authentication information for the two parties; this information is then used to construct a *single* "authenticated" garbled circuit which is then transmitted and evaluated.

We also show how to efficiently instantiate the preprocessing phase using our own optimized version of the TinyOT protocol. Our overall 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 total with an online time of just 1 ms. Previous work with the best online time (also 1 ms) requires 124 ms in total; previous work with the best total time requires 62 ms (with 14 ms online time).
- In the amortized setting where the time is amortized over 1024 executions, each AES computation runs in just 6.7 ms overall, with roughly the same online time as above. The best previous work in this setting requires roughly the same total time but does not support preprocessing independent of the function to be evaluated.

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

As a by-product of our framework, we also obtain the *first* constant-round maliciously-secure two-party computation with $O(|\mathcal{C}|\kappa)$ bits of communication, by instantiating the preprocessing using the IPS compiler under the Φ-hiding assumption. This protocol achieves a constant communication overhead compared to Yao's semi-honest protocol.

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 have been attracting the interest of the security community since the Fairplay implementation [MNPS04] of Yao's garbled-circuit protocol [Yao86], leading to several subsequent improvements [HEKM11, ZRE15, KS08, KMR14, ALSZ13, BHKR13, PSSW09]. The field has advanced to the point where semi-honest secure computations that were considered out of reach 10 years ago can now be done easily. For example, Fairplay was able to evaluate 30 gates per second; we can evaluate 6 million gates per second using off-the-shelf hardware.

While these results are impressive, semi-honest security—which assumes that both parties follow the protocol yet may try to learn additional information from the execution—is clearly not sufficient for all applications, and this has motivated researchers to explore the stronger notion of malicious security. There have been incredible advances in the efficiency of protocols for maliciously secure two-party computation over the last decade. One popular approach for designing such protocols is to apply the "cut-and-choose" technique [LP07, SS11, LP11, HKE13, Lin13, Bra13, FJN14, AMPR14] to Yao's garbled-circuit protocol [Yao86] for (semi-honest) secure two-party computation. For statistical security $2^{-\rho}$, the best protocols using this paradigm require ρ garbled circuits (which is optimal for that approach). Recently, Wang et al. [WMK17] showed a protocol based on this technique that can securely evaluate an AES circuit (in the single-execution setting with no preprocessing) in only 65 ms with moderate hardware.

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) [HKK$^+$14, LR14, LR15]. When amortizing over τ executions, only $O(\frac{\rho}{\log \tau})$ garbled circuits are needed per execution. Rindal and Rosulek [RR16] recently reported a time of 6.4 ms to evaluate an AES circuit over a 10 Gbps network, amortized over 1024 executions.

Other techniques for constant-round, maliciously secure two-party computation, with asymptotically better performance than cut-and-choose (without amortization), have also been explored. The LEGO protocol and subsequent optimizations [NO09, FJN$^+$13, FJNT15, HZ15, NST17] are based on a gate-level cut-and-choose approach that can be done during a preprocessing phase before the circuit to be evaluated is known. This class of protocols has good asymptotic performance (see Table 2) and very small online time; however, the total cost of the state-of-the-art LEGO implementation [NST17] is still higher than the total cost of the best protocol based on the cut-and-choose approach applied at the garbled-circuit level. In Table 1, we summarize the performance of state-of-the-art protocols based on different approaches under the same hardware and network conditions.

The Beaver-Micali-Rogaway compiler [BMR90] provides yet another approach to constructing constant-round protocols secure against malicious adversaries. This compiler uses an "outer" secure-computation protocol to generate a garbled circuit that can then be evaluated. Lindell et al. [LPSY15] applied this idea using SPDZ [DPSZ12] as the outer protocol. Compared to their work, our protocol is asymptotically more efficient in the function-independent preprocessing phase; more importantly, the concrete efficiency of our protocol is much better for several reasons: (1) our work is compatible with free-XOR and we do not suffer from any blowup in the size of the circuit being evaluated; (2) Lindell et al. require *five SPDZ-style multiplications* per AND gate of the underlying circuit, while we only need *one TinyOT-style AND* computation per AND gate. We provide a more thorough comparison in Section 8.2.

There are also protocols using a larger number of communication rounds. The TinyOT protocol [NNOB12] adds malicious security to the classical GMW protocol [GMW87] by adding information-theoretic MACs to shares held by both parties. TinyOT has smaller communication

AES Evaluation							
	Single-Execution Setting			Amortized Setting (1024 executions)			
	[NST17]	[WMK17]	**This paper**	[LR15]	[RR16]	[NST17]	**This paper**
Ind. Phase	89.6 ms	-	10.9 ms	-	-	13.84 ms	4.9 ms
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
Semi-Honest	2.1 ms						

SHA-256 Evaluation							
	Single-Execution Setting			Amortized Setting (1024 executions)			
	[NST17]	[WMK17]	**This paper**	[LR15]	[RR16]	[NST17]	**This paper**
Ind. Phase	478.5 ms	-	96 ms	-	-	183.5 ms	64.8 ms
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
Semi-Honest	9.6 ms						

Table 1: **Summary of state-of-the-art constant-round maliciously secure 2PC protocols.** All timings are based on an Amazon EC2 `c4.8xlarge` instance over a LAN. Single-execution time does not include the base-OTs, which are the same for all protocols ($\sim$20 ms). Timings for the semi-honest protocol are based on the same garbling code used in our protocol, and also do not include time for the base-OTs. See Section 8 for more details.

Protocol	Function-independent preprocessing	Function-dependent preprocessing	Online phase	Storage
Cut-and-choose [Lin13, AMPR14, WMK17]	—	$O(\|\mathcal{C}\|\rho)$	$O(\mathcal{I}\|\rho)$	$O(\|\mathcal{C}\|\rho)$
Amortized [HKK+14, LR14]	—	$O\left(\|\mathcal{C}\|\frac{\rho}{\log\tau}\right)$	$O\left(\|\mathcal{I}\|\frac{\rho}{\log\tau}\right)$	$O\left(\frac{\|\mathcal{C}\|\rho}{\log\tau}\right)$
LEGO [NO09, FJN+13]	$O\left(\frac{\|\mathcal{C}\|\rho}{\log\tau+\log\|\mathcal{C}\|}\right)$	$O(\|\mathcal{C}\|)$	$O((\|\mathcal{I}\|+\|\mathcal{O}\|))$	$O\left(\frac{\|\mathcal{C}\|\rho}{\log\tau+\log\|\mathcal{C}\|}\right)$
SPDZ-BMR [LPSY15, KOS16]*	$O(\|\mathcal{C}\|\kappa)$	$O(\|\mathcal{C}\|)$	$O((\|\mathcal{I}\|+\|\mathcal{O}\|))$	$O(\|\mathcal{C}\|)$
This paper (with Section 6)	$O\left(\frac{\|\mathcal{C}\|\rho}{\log\tau+\log\|\mathcal{C}\|}\right)$	$O(\|\mathcal{C}\|)$	$\|\mathcal{I}\|+\|\mathcal{O}\|$	$O(\|\mathcal{C}\|)$
This paper (with IPS)	$O(\|\mathcal{C}\|)$	$O(\|\mathcal{C}\|)$	$\|\mathcal{I}\|+\|\mathcal{O}\|$	$O(\|\mathcal{C}\|)$

Table 2: **Communication and computational complexity of constant-round 2PC protocols.** $|\mathcal{I}|$ represents the length of the inputs, $|\mathcal{O}|$ the length of the outputs, and $|\mathcal{C}|$ the circuit size. The first three columns show the number of symmetric-key operations, which is also the number of symmetric-key ciphertexts sent. The statistical security parameter is ρ, and the computational security parameter is $\kappa \geq \rho$. We let τ be the number of protocol executions in the amortized setting. "Storage" is the size of the state generated by the preprocessing phase(s).
* Although the complexity of function-independent preprocessing can be reduced to $O(|\mathcal{C}|\kappa)$ using somewhat homomorphic encryption [DPSZ12], doing so requires a number of *public-key* operations proportional to $|\mathcal{C}|$.

complexity than the LEGO family of protocols, but it—just like the GMW protocol—has round complexity linear in the depth of the circuit being evaluated. The IPS compiler [IPS08, LOP11] has asymptotic complexity (in the OT-hybrid model) proportional to the size of the circuit being evaluated. It, too, has the disadvantage of requiring a number of rounds linear in the depth of the circuit. A more serious drawback is that the concrete complexity of the protocol is unclear, since it has not yet been implemented (and appears quite difficult to implement). Note that these protocols suffer a lot from the network latency. Even in the LAN setting, each round-trip requires at least 0.5 ms: for the AES circuit with a depth about 50, this means that the cost will be at least 25 ms.

In Table 2, we summarize the complexity of various constant-round 2PC protocols. Following [NST17], we divide execution of protocols into three phases:

- **Function-independent preprocessing.** During this phase, the parties do not need to know their inputs nor 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 phase*.

- **Online phase.** In this phase, two parties evaluate the agreed-upon function on their respective inputs.

Our contributions. We propose a new approach for constructing constant-round 2PC protocols with extremely high efficiency. At a high level (further details are in Section 3), and following ideas of [NNOB12], our protocol relies on 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 [NNOB12], however, the parties in our protocol use this information in the online phase to generate a *single* "authenticated" garbled circuit. (Conceptually similar ideas were used by Damgård and Ishai [DI05] in the context of multi-party computation with honest majority, and by Choi et al. [CKMZ14] for three-party computation with dishonest majority.) 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 the *semi-honest* garbled-circuit protocol, the cost of the function-dependent preprocessing phase of our protocol is only about $2\times$ higher (assuming 128-bit computational security and 40-bit statistical security), and the cost of the online phase is essentially unchanged.

We also show how to instantiate $\mathcal{F}_{\mathsf{Pre}}$ efficiently using an improved version of the TinyOT protocol [NNOB12] that we develop (see Section 6). Instantiating our framework in this way, we obtain an efficient protocol with the same asymptotic communication complexity as recent protocols based on LEGO, but with two advantages. First, our protocol has better *concrete* efficiency (see Table 1 and Section 8). For example, it requires only 16.6 ms total to evaluate AES, a $6\times$ improvement compared to a recent implementation of a LEGO-style approach [NST17]. Furthermore, the storage needed by our protocol between the offline phase and the online phase is (asymptotically) smaller (see Table 2). The latter is especially important when very large circuits are evaluated.

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

- Upon receiving Δ_{A} from P_{A} and init from P_{B}, and assuming no values $\Delta_{\mathsf{A}}, \Delta_{\mathsf{B}}$ are currently stored, choose uniform $\Delta_{\mathsf{B}} \in \{0,1\}^{\rho}$ and store $\Delta_{\mathsf{A}}, \Delta_{\mathsf{B}}$. Send Δ_{B} to P_{B}.
- Upon receiving $(\mathsf{random}, r, \mathsf{M}[r], \mathsf{K}[s])$ 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 $s_3 := r_3 \oplus ((r_1 \oplus s_1) \wedge (r_2 \oplus s_2))$ and set $\mathsf{K}[r_3] := \mathsf{M}[r_3] \oplus r_3\Delta_{\mathsf{B}}$ and $\mathsf{M}[s_3] := \mathsf{K}[s_3] \oplus s_3\Delta_{\mathsf{A}}$. Send $(s_3, \mathsf{M}[s_3], \mathsf{K}[r_3])$ to P_{B}.

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

Instantiating our framework with the realization of $\mathcal{F}_{\mathsf{Pre}}$ described in Section 6 yields a protocol with the best concrete efficiency, and is the main focus of this paper. However, it is interesting to observe that our framework can also be instantiated in other ways:

- When $\mathcal{F}_{\mathsf{Pre}}$ is instantiated using the IPS compiler [IPS08] and the bit-OT protocol by Ishai et al. [IKOS09], we obtain what is (to the best of our knowledge) the *first* maliciously secure constant-round 2PC protocol with complexity $O(|\mathcal{C}|\kappa)$. Note that, up to constant factors, this matches the complexity of *semi-honest* secure two-party computation based on garbled circuits.

- 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 $O(|\mathcal{C}|\kappa)$. Previous work in the same model [MOR16] achieves the same complexity but with number of rounds proportional to the circuit depth.

1.1 Other Related Work

Nielsen and Orlandi [NO16] proposed a maliciously secure 2PC protocol that can achieve constant amortized overhead but only when the number of executions is at least linear in the size of the circuit being computed (which is potentially impractical). Further, the amortization is over *parallel* executions only, where all evaluations must be done at the same time. In contrast, we can handle amortization with *sequential* executions, where inputs to different executions do not need to be known all at once.

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 input-wire indices of the gate, γ denotes the output-wire index of the gate, and $T \in \{\oplus, \wedge\}$

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

Table 3: An authenticated garbled table for an AND gate.

denotes the type of the gate. We use $\mathcal{I}_1$ to denote the set of input-wire indices for $\mathsf{P_A}$'s input, $\mathcal{I}_2$ to denote the set of input-wire indices for $\mathsf{P_B}$'s input, $\mathcal{W}$ to denote the set of output-wire indices of all the AND gates, and $\mathcal{O}$ to denote the set of output-wire indices of the circuit itself.

2.1 Information-theoretic MACs

We use the information-theoretic message authentication codes (IT-MACs) of [NNOB12]. $\mathsf{P_A}$ holds a random *global key* $\Delta_\mathsf{A} \in \{0,1\}^\rho$. A bit b known by $\mathsf{P_B}$ is authenticated by having $\mathsf{P_A}$ hold a random key $\mathsf{K}[b]$ and having $\mathsf{P_B}$ hold the corresponding tag $\mathsf{M}[b] := \mathsf{K}[b] \oplus b\Delta_\mathsf{A}$. Symmetrically, $\mathsf{P_B}$ holds an independent global key Δ_B; a bit b known by $\mathsf{P_A}$ is authenticated by having $\mathsf{P_B}$ hold a random key $\mathsf{K}[b]$ and having $\mathsf{P_A}$ hold the tag $\mathsf{M}[b] := \mathsf{K}[b] \oplus b\Delta_\mathsf{B}$. We use $[b]_\mathsf{A}$ to denote an authenticated bit known to $\mathsf{P_A}$ (i.e., $[b]_\mathsf{A}$ means that $\mathsf{P_A}$ holds $(b, \mathsf{M}[b])$ and $\mathsf{P_B}$ holds $\mathsf{K}[b]$), with $[b]_\mathsf{B}$ defined symmetrically.

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

It is possible to extend the above idea to XOR-shared values by having each party's share be authenticated. That is, say we have a value $\lambda := r \oplus s$, where $\mathsf{P_A}$ knows r and $\mathsf{P_B}$ knows s. Then by having $\mathsf{P_A}$ hold $(r, \mathsf{M}[r], \mathsf{K}[s])$ and $\mathsf{P_B}$ hold $(s, \mathsf{K}[r], \mathsf{M}[s])$, we end up with an authenticated secret-sharing of λ. It can be observed that this scheme is also XOR-homomorphic.

As described in the Introduction, we use a preprocessing phase that realizes a stateful ideal functionality $\mathcal{F}_\mathsf{Pre}$. This functionality, described in Figure 1, is used to set up correlated values between the players along with their corresponding IT-MACs. The functionality chooses uniform global keys (once-and-for-all) 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 $r \oplus s$. (The malicious party may choose the "random values" it receives, but note that 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 their authenticated shares for two bits; the functionality then computes a (fresh) authenticated share of the AND of those bits. We defer until Section 4.2 a discussion of how $\mathcal{F}_\mathsf{Pre}$ can be instantiated.

3 Protocol Intuition

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

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

Table 4: The final construction of an authenticated garbled table for an AND gate.

We begin by reviewing standard garbled circuits. Each wire α of a circuit is associated with a random "mask" $\lambda_\alpha \in \{0,1\}$ known to $\mathsf{P_A}$. If the true value (i.e., the value when the circuit is evaluated on the parties' inputs) of that wire is x, then the masked value observed by the circuit evaluator (namely, $\mathsf{P_B}$) on that wire will be $\bar{x} = x \oplus \lambda_\alpha$. Each wire α is also associated with two labels $\mathsf{L}_{\alpha,0}$ and $\mathsf{L}_{\alpha,1} := \mathsf{L}_{\alpha,0} \oplus \Delta$ known to $\mathsf{P_A}$ (here we are using the free-XOR technique[KS08]). If the masked bit on that wire is $\bar{x}$, then $\mathsf{P_B}$ learns $\mathsf{L}_{\alpha,\bar{x}}$.

Let $H : \{0,1\}^* \to \{0,1\}^{1+2\kappa}$ be a hash function modeled as a random oracle. The garbled table for, e.g., an AND gate $(\alpha, \beta, \gamma, \wedge)$ is given by:

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

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

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

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

Once $\mathsf{P_A}$ sends its shares of all the garbled gates, $\mathsf{P_B}$ can evaluate the garbled circuit: Given $(\bar{x}, \mathsf{L}_{\alpha,\bar{x}})$ and $(\bar{y}, \mathsf{L}_{\beta,\bar{y}})$, it picks the appropriate row, decrypts $\mathsf{P_A}$'s share of that row using the garbed labels it holds, and then XORs the result with its own shares of that same row to obtain $(\bar{z}, \mathsf{L}_{\gamma,\bar{z}})$.

Informally, the above modification ensures *privacy* against a malicious P_A since (intuitively) the result of any changes P_A introduces will depend on the random masks but be *independent* of P_B's inputs. However, P_A can still affect *correctness* by, e.g., flipping the masked value in one of the rows of a garbled gate. This can be addressed by adding an information-theoretic MAC on P_A's share of the masked bit. That is, the shares of the garbled table now take the form in Table 3.

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 independent of P_B's actual input.

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 key observation is that P_A's shares of the wire labels need not be authenticated; in the worst-case, incorrect values used by P_A will cause an input-*independent* abort.

We also observe that, for example,

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

Our next key insight is that if s_{00} is an authenticated bit known to P_B, then P_A can locally compute $\mathsf{L}_{\gamma,0} \oplus r_{00}\Delta_\mathsf{A} \oplus \mathsf{K}[s_{00}]$; then the other share $\mathsf{K}[s_{00}] \oplus s_{00}\Delta_\mathsf{A}$ is just the MAC on s_{00} that P_B already knows! Thus, we can rewrite the garbled table as in Table 4. (The $\{R_{ij}\}$ values are no longer needed since the $\{s_{ij}\}$ are unknown to P_A, and that is enough to hide the masks from P_A.) Shares of the table then become easy to compute in a distributed fashion.

One final optimization is based on the simple observation that the entries in the truth table are linearly dependent. More precisely,

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

Therefore, in order to jointly compute the above garbled table, the parties just need to compute MACs on shares of the masks $\lambda_\alpha, \lambda_\beta, \lambda_\gamma$, and then compute MACs on shares of the bit $\lambda_\alpha \wedge \lambda_\beta$.

4 Our Framework and Its Instantiations

4.1 Protocol in the $\mathcal{F}_{\mathsf{Pre}}$-Hybrid Model

In Figure 2, we give the complete description of our main protocol in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid model. For clarity, we set $\rho = \kappa$ in the protocol. In section 7, we discuss how to support other values of ρ in general. Note that the calls to $\mathcal{F}_{\mathsf{Pre}}$ can be performed in parallel, so the protocol runs in constant rounds. Since $\mathcal{F}_{\mathsf{Pre}}$ can be instantiated efficiently in constant rounds (see, e.g., Section 6), we can use our approach to obtain constant-round 2PC protocols.

Although our protocol 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 [Bea92].

4.2 Instantiating $\mathcal{F}_{\mathsf{Pre}}$

We now discuss various ways $\mathcal{F}_{\mathsf{Pre}}$ can be instantiated.

TinyOT-based instantiation. We obtain the best concrete efficiency by instantiating $\mathcal{F}_{\mathsf{Pre}}$ using an improved variant of TinyOT [NNOB12]. This is the instantiation we focus on for the rest of the paper.

Our variant of TinyOT, which gives a $2.7\times$ improvement as compared to the original TinyOT protocol, is described in detail in Section 6. At a high level, the two parties execute a semi-honest secure protocol to compute shares of an AND gate and related IT-MACs as described in regard to $\mathcal{F}_{\mathsf{Pre}}$. Additional checks are performed such that an adversary who attempts a selective-failure attack is caught only with probability $1/2$; with the remaining probability it may learn one bit of additional information about the AND gate. For this reason, we refer to these as "leaky AND gates." Note that with probability at most $2^{-\rho}$ can an attacker learn information about ρ or more of these leaky AND gates without being caught.

We compute n leaky AND gates. Then, following [NNOB12], we randomly permute and partition the leaky AND gates into n/B buckets, each containing $B = \rho/\log n$ leaky AND gates. It can be proven that with all but negligible probability, each bucket contains at least one leaky AND gate for which the attacker has learned no information (although we don't know which one it is). For each bucket, we now combine all leaky AND gates into one AND gate in a way that guarantees that as long as one of them is not leaked, the resulting AND gate is secure.

One technical issue is that the functionality defined in the TinyOT paper [NNOB12] includes a global key query for technical reasons. This can be added to our $\mathcal{F}_{\mathsf{Pre}}$ functionality without affecting the proof much. We will provide further details in the full version.

IPS-based instantiation. We obtain better asymptotic performance by using the IPS protocol [IPS08] to realize $\mathcal{F}_{\mathsf{Pre}}$. In the function-dependent preprocessing phase, we need to produce a sharing of λ_i for each wire i, and a 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 with $O((\kappa + \rho) \cdot |C|)$ gates. For securely evaluating a circuit of depth d and size ℓ, the IPS protocol uses communication complexity $O(\ell) + \mathit{poly}(\kappa, d, \log \ell)$ and $O(d)$ rounds of communication. When applied to our setting, this translates to a communication complexity of $O((\kappa+\rho) \cdot |C|) + \mathit{poly}(\kappa, \log |C|)$; for sufficiently large circuits, the leading term is $O((\kappa + \rho) \cdot |C|)$.

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. [MOR16], the offline phase can also be decoupled from the identity of other party; we refer to their paper for further details.

5 Proof of Security

Theorem 5.1. *The protocol in Figure 2, where H is modeled as a random oracle securely computes f (against malicious adversaries) with statistical security $2^{-\rho}$ in the $\mathcal{F}_{\mathsf{Pre}}$-hybrid.*

Protocol Π_{2pc}

Inputs: In the function-dependent phase, the parties agree on a circuit for a function $f : \{0,1\}^{|\mathcal{I}_1|} \times \{0,1\}^{|\mathcal{I}_2|} \to \{0,1\}^{|\mathcal{O}|}$. In the input-processing phase, $\mathsf{P_A}$ holds $x \in \{0,1\}^{|\mathcal{I}_1|}$ and $\mathsf{P_A}$ 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 $\mathsf{P_B}$, where $\lambda_w = s_w \oplus r_w$. $\mathsf{P_A}$ also picks a uniform κ-bit string $\mathsf{L}_{w,0}$.

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 $\mathsf{L}_{\gamma,0} := \mathsf{L}_{\alpha,0} \oplus \mathsf{L}_{\beta,0}$. $\mathsf{P_B}$ computes $(s_\gamma, \mathsf{M}[s_\gamma], \mathsf{K}[r_\gamma]) := (s_\alpha \oplus s_\beta, \mathsf{M}[r_\beta] \oplus \mathsf{M}[r_\beta], \mathsf{K}[r_\alpha] \oplus \mathsf{K}[r_\beta])$.
4. Then, for each gate $\mathcal{G} = (\alpha, \beta, \gamma, \wedge)$:

   (a) $\mathsf{P_A}$ (resp., $\mathsf{P_B}$) sends $(\mathsf{and}, (r_\alpha, \mathsf{M}[r_\alpha], \mathsf{K}[s_\alpha]), (r_\beta, \mathsf{M}[r_\beta], \mathsf{K}[s_\beta]))$ (resp., $(\mathsf{and}, (s_\alpha, \mathsf{M}[s_\alpha], \mathsf{K}[r_\alpha]), (s_\beta, \mathsf{M}[s_\beta], \mathsf{K}[r_\beta])))$ 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:

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

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

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

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

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

Input processing:

5. For each $w \in \mathcal{I}_1$, $\mathsf{P_A}$ 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. $\mathsf{P_B}$ then sends $y_w \oplus \lambda_w := s_w \oplus y_w \oplus r_w$ to $\mathsf{P_A}$. Finally, $\mathsf{P_A}$ sends $\mathsf{L}_{w, y_w \oplus \lambda_w}$ to $\mathsf{P_B}$.
6. For each $w \in \mathcal{I}_2$, $\mathsf{P_B}$ 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. $\mathsf{P_A}$ then sends $x_w \oplus \lambda_w := s_w \oplus x_w \oplus r_w$ and $\mathsf{L}_{w, x_w \oplus \lambda_w}$ to $\mathsf{P_B}$.

Circuit evaluation:

7. $\mathsf{P_B}$ evaluates the circuit in topological order. For each gate $\mathcal{G} = (\alpha, \beta, \gamma, T)$, $\mathsf{P_B}$ initially holds $(z_\alpha \oplus \lambda_\alpha, \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha})$ and $(z_\beta \oplus \lambda_\beta, \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta})$, where z_α, z_β are the underlying values of the wires.

   (a) If $T = \oplus$, $\mathsf{P_B}$ computes $z_\gamma \oplus \lambda_\gamma := (z_\alpha \oplus \lambda_\alpha) \oplus (z_\beta \oplus \lambda_\beta)$ and $\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma} := \mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha} \oplus \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}$.

   (b) If $T = \wedge$, $\mathsf{P_B}$ computes $i := 2(z_\alpha \oplus \lambda_\alpha) + (z_\beta \oplus \lambda_\beta)$ followed by $(r_{\gamma,i}, \mathsf{M}[r_{\gamma,i}], \mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,i}] \oplus r_{\gamma,i}\Delta_\mathsf{A}) := G_{\gamma,i} \oplus H(\mathsf{L}_{\alpha, z_\alpha \oplus \lambda_\alpha}, \mathsf{L}_{\beta, z_\beta \oplus \lambda_\beta}, \gamma, i)$. Then $\mathsf{P_B}$ checks that $(r_{\gamma,i}, \mathsf{K}[r_{\gamma,i}], \mathsf{M}[r_{\gamma,i}])$ is valid and, if so, computes $z_\gamma \oplus \lambda_\gamma := (s_{\gamma,i} \oplus r_{\gamma,i})$ and $\mathsf{L}_{\gamma, z_\gamma \oplus \lambda_\gamma} := (\mathsf{L}_{\gamma,0} \oplus \mathsf{K}[s_{\gamma,i}] \oplus r_{\gamma,i}\Delta_\mathsf{A}) \oplus \mathsf{M}[s_{\gamma,i}]$.

Output determination:

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

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

(Recall that we set $\rho = \kappa$ in Figure 2 for simplicity of exposition. When modified as described in Section 7, our protocol achieves statistical security $2^{-\rho}$.)

Proof. We consider separately the case where $\mathsf{P_A}$ or $\mathsf{P_B}$ is malicious.

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 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\mathsf{P_B}$, where $\mathcal{S}$ also plays the role of $\mathcal{F}_{\mathsf{Pre}}$, recording all values that are sent to $\mathcal{A}$.

5 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\mathsf{P_B}$ using input $y = 0$.

6 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\mathsf{P_B}$. For each wire $w \in \mathcal{I}_1$, $\mathcal{S}$ receives $\bar{x}_w$ and computes $x_w = \bar{x}_w \oplus r_w \oplus s_w$, where r_w, s_w are values $\mathcal{S}$ used to play the role of $\mathcal{F}_{\mathsf{Pre}}$ in previous steps. $\mathcal{S}$ sends x to $\mathcal{F}$.

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

We now show that the joint distribution over the outputs of $\mathcal{A}$ and the honest $\mathsf{P_B}$ in the real world is indistinguishable from the joint distribution over 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 $\mathcal{S}$ plays the role of an honest $\mathsf{P_B}$ using $\mathsf{P_B}$'s actual input y. $\mathcal{S}$ also plays 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 $\bar{x}_w$ and computes $x_w = \bar{x}_w \oplus r_w \oplus s_w$, where s_w, r_w are values $\mathcal{S}$ used when playing the role of $\mathcal{F}_{\mathsf{Pre}}$. $\mathcal{S}$ sends x to $\mathcal{F}$. If an honest $\mathsf{P_B}$ would abort, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and aborts; otherwise $\mathcal{S}$ sends $\mathsf{continue}$ to $\mathcal{F}$.

The distributions on the view of the adversary in the two experiments above are exactly identical. Lemma 5.1 shows that $\mathsf{P_B}$ generates the same output in both experiments with probability $1 - 2^{-\rho}$.

Hybrid$_3$. Same as **Hybrid$_2$**, except that $\mathcal{S}$ computes $\{s_w\}_{w \in \mathcal{I}_2}$ as follows: $\mathcal{S}$ first randomly pick $\{u_w\}_{w \in \mathcal{I}_2}$, and then computes $s_w := u_w \oplus y_w$.

The above two experiments are identically distributed.

Hybrid$_4$. Same as **Hybrid$_3$**, except that $\mathcal{S}$ uses $y = 0$ as inputs throughout the protocol.

Note that although the value of y in **Hybrid$_3$** and **Hybrid$_4$** are different, the distributions of $s_w \oplus y_w$ are exactly the same. The view of the adversary in the two experiments are therefore the same. We next show that $\mathsf{P_B}$ aborts with the same probability in two experiments.

Observe that the only place where $\mathsf{P_B}$'s abort can possibly depends on y is in step 7(b). However, this abort depends on which row is selected to decrypt, that is the value of $\lambda_\alpha \oplus z_\alpha$

and $\lambda_\beta \oplus z_\beta$, which are chosen uniformly and independently in both experiments. Therefore, the two experiments are identically distributed.

Note that $\mathbf{Hybrid_4}$ corresponds to the ideal-world execution, so 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 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\mathsf{P_A}$ and plays the functionality of $\mathcal{F}_{\mathsf{Pre}}$. If an honest $\mathsf{P_A}$ would abort, $\mathcal{S}$ output whatever $\mathcal{A}$ outputs and aborts.

5 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\mathsf{P_A}$, receives $y_w \oplus \lambda_w$ from $\mathcal{A}$, and computes $y_w := \bar{y}_w \oplus s_w \oplus r_w$, where s_w, r_w are values $\mathcal{S}$ used when playing the role of $\mathcal{F}_{\mathsf{Pre}}$. $\mathcal{S}$ sends y to $\mathcal{F}$, which sends $z = f(x, y)$ to $\mathcal{S}$.

6 $\mathcal{S}$ interacts with $\mathcal{A}$ acting as an honest $\mathsf{P_A}$ using input $x = 0$. If an honest $\mathsf{P_A}$ would abort, $\mathcal{S}$ output whatever $\mathcal{A}$ outputs and aborts.

8 $\mathcal{S}$ computes $z' = f(0, y)$. For each $w \in \mathcal{O}$, if $z'_w = z_w$, $\mathcal{S}$ sends $(r_w, \mathsf{M}[r_w])$; otherwise, $\mathcal{S}$ sends $(r_w \oplus 1, \mathsf{M}[r_w] \oplus \Delta_\mathsf{B})$, where Δ_B is the value $\mathcal{S}$ used when playing the role of $\mathcal{F}_{\mathsf{Pre}}$.

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

$\mathbf{Hybrid_1}$. Same as the hybrid-world protocol, where $\mathcal{S}$ plays the role of an honest $\mathsf{P_A}$ using the actual input x.

$\mathbf{Hybrid_2}$. Same as $\mathbf{Hybrid_1}$, except that, in step 5, $\mathcal{S}$ receives $y_w \oplus \lambda_w$ from $\mathcal{A}$, and computes $y_w := \bar{y}_w \oplus s_w \oplus r_w$, where s_w, r_w are values $\mathcal{S}$ used when playing the role of $\mathcal{F}_{\mathsf{Pre}}$. $\mathcal{S}$ then sends y to $\mathcal{F}$, and receives $z = f(x, y)$. In Step 8, for each $w \in \mathcal{O}$, $\mathcal{S}$ computes $r'_w := z_w \oplus s_w$, and sends $(r'_w, \mathsf{K}[r'_w] \oplus r'_w \Delta_\mathsf{B})$, where Δ_B is the value $\mathcal{S}$ used to play the role of $\mathcal{F}_{\mathsf{Pre}}$.

$\mathsf{P_A}$ does not have output; furthermore the view of $\mathcal{A}$ does not change between the two **Hybrid**s since the value z that $\mathcal{S}$ obtains from $\mathcal{F}$ is the same as the one $\mathcal{A}$ obtains in $\mathbf{Hybrid_1}$.

$\mathbf{Hybrid_3}$. Same as $\mathbf{Hybrid_2}$, except that in step 6, $\mathcal{S}$ uses $x = 0$ as input.

Note that since $\mathcal{S}$ uses different values for x between two **Hybrid**s, we also need to show that the garbled rows $\mathsf{P_B}$ opened are indistinguishable between two **Hybrid**s. According to Lemma 5.2, $\mathsf{P_B}$ is able to open only one garble rows in each garbled table $G_{\gamma,i}$. Therefore, given that $\{\lambda_w\}_{w \in \mathcal{I}_1 \cup \mathcal{W}}$ values are not known to $\mathsf{P_B}$, masked values and garbled keys are indistinguishable between the two **Hybrid**s.

As $\mathbf{Hybrid_3}$ is the ideal-world execution, the proof is complete. □

Lemma 5.1. *Consider an $\mathcal{A}$ corrupting $\mathsf{P_A}$ and denote $x_w := \bar{x}_w \oplus s_w \oplus r_w$, where $\bar{x}_w$ is the value $\mathcal{A}$ sent to $\mathsf{P_B}$, s_w, r_w are the values from $\mathcal{F}_{\mathsf{Pre}}$. With probability $1 - 2^{-\rho}$, $\mathsf{P_B}$ either aborts or only learns $z = f(x, y)$.*

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

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

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

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

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

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

Lemma 5.2. *Consider an $\mathcal{A}$ corrupting $\mathsf{P_B}$, with negligible, probability, $\mathsf{P_B}$ learns both garbled keys for some wire.*

Proof. The proof is very similar to the proof of security for garbled circuits in the semi-honest setting.

Base step: $\mathsf{P_B}$ can only learn one garbled keys for each input wire, since $\mathsf{P_A}$ only sends one garbled wire, and $\mathsf{P_B}$ cannot learn Δ_A in the protocol.

Induction step: It is obvious that $\mathsf{P_B}$ cannot learn the other label for an XOR gate and so we focus on AND gates. Note that $\mathsf{P_B}$ only learns one garbled key each for input wires α and β. However, each row is encrypted using different combinations of $\{L_{\alpha,b}\}_{b \in \{0,1\}}$ and $\{\mathsf{L}_{\beta,b}\}_{b \in \{0,1\}}$. In order for $\mathsf{P_B}$ to open two rows in the garbled table, $\mathsf{P_B}$ needs to learn both garbled keys for some input wire, which contradict with assumptions in the induction step. ☐

6 Improved TinyOT protocol

In this section, we describe an improvement to the TinyOT protocol. For a bucket size of $B = \frac{\rho}{\log |\mathcal{C}|} + 1$, the original protocol requires $14B + 2$ authenticated bits for each AND gate. In the following, we will introduce an improved version where only $6B$ authenticated bits are needed for each AND gate. For a circuit of size 2^{20}, with $\rho = 40$, this is an improvement of $2.4\times$.

6.1 Half Authenticated AND

Before describing the main protocol, we will first show how to compute an AND triple with only x's being authenticated ($\mathcal{F}_{\mathsf{HaAND}}$). This will serve as a building block for the following sections. The functionality $\mathcal{F}_{\mathsf{HaAND}}$ is described in Figure 3. It outputs authenticated bits $[x_1]_\mathsf{A}$ and $[x_2]_\mathsf{B}$ to the two parties, it also gets y_1 from $\mathsf{P_A}$ and y_2 from $\mathsf{P_B}$ without authentication. The functionality then outputs random shares of $x_1 y_2 \oplus x_2 y_1$. Looking ahead to the next subsection, this prevents

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

1. The box picks random $[x_1]_\mathsf{A}$ and $[x_2]_\mathsf{B}$ and sends them to the two parties.
2. Upon receiving y_1 from $\mathsf{P_A}$ and y_2 from $\mathsf{P_B}$, the box samples two random bits v_1, v_2 such that $v_1 \oplus v_2 = x_1 y_2 \oplus x_2 y_1$. The box sends v_1 to $\mathsf{P_A}$, v_2 to $\mathsf{P_B}$.

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

Figure 3: Functionality $\mathcal{F}_{\mathsf{HaAND}}$ that computes a half authenticated AND triple.

Protocol Π_{HaAND}

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ call $\mathcal{F}_{\mathsf{abit}}$ to obtain $[x_1]_\mathsf{A}$ and $[x_2]_\mathsf{B}$.
2. $\mathsf{P_A}$ picks random bit s_1 and computes $H_0 := \mathsf{Lsb}(H(\mathsf{K}[x_2])) \oplus s_1$, $H_1 := \mathsf{Lsb}(H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A})) \oplus s_1 \oplus y_1$. $\mathsf{P_A}$ sends (H_0, H_1) to $\mathsf{P_B}$, who computes $s_2 := H_{x_2} \oplus \mathsf{Lsb}(H(\mathsf{M}[x_2]))$.
3. $\mathsf{P_B}$ picks random bit t_1 and computes $H_0 := \mathsf{Lsb}(H(\mathsf{K}[x_1])) \oplus t_1$, $H_1 := \mathsf{Lsb}(H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B})) \oplus t_1 \oplus y_2$. $\mathsf{P_B}$ sends (H_0, H_1) to $\mathsf{P_A}$, who computes $t_2 := H_{x_1} \oplus \mathsf{Lsb}(H(\mathsf{M}[x_1]))$.
4. $\mathsf{P_A}$ computes $v_1 := s_1 \oplus t_2$, $\mathsf{P_B}$ computes $v_2 := s_2 \oplus t_1$.

Figure 4: Protocol Π_{HaAND} instantiating $\mathcal{F}_{\mathsf{HaAND}}$.

parties from flipping x's, which would cause a selective failure attack on y, but would still allows parties to flip y's, which would cause a selective failure attack on x. The protocol that instantiates this functionality is simple due to the fact that not all bits are authenticated. In the proof, we will essentially show that if an adversary "corrupts" any message, it is equivalent to using some other input.

Lemma 6.1. *Assuming H is a random oracle, the protocol in Figure 4 securely implements the functionality in Figure 3 in the $\mathcal{F}_{\mathsf{abit}}$-hybrid model.*

Proof. First we will show the correctness of the protocol. We will show that $s_1 \oplus s_2 = x_2 y_1$ and that $t_1 \oplus t_2 = x_1 y_2$. Without loss of generality, we will show the first equation. There are two cases:

- $x_2 = 0$. In this case, $\mathsf{P_B}$ obtains $s_2 = s_1$.
- $x_2 = 1$. In this case, $\mathsf{P_B}$ obtains $s_2 = s_1 \oplus y_1$.

In both cases, the equation we want to show holds. The other equation can be proven in exactly the same way. The correctness of the protocol follows immediately from these two equations.

In a part below, we will continue to the simulation proof. The proof is straightforward, mainly due to the fact that each party's input is not authenticated and therefore $\mathcal{S}$ can extract the values easily.

Malicious $\mathsf{P_A}$. The simulator works as follows:

1. $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{abit}}$, and stores $[x_1]_\mathsf{A}$, $[x_2]_\mathsf{B}$.
2. $\mathcal{S}$ receives (H_0, H_1) from $\mathcal{A}$, and computes $s_1 := H_0 \oplus \mathsf{Lsb}(H(\mathsf{K}[x_2]))$, $y_1 := H_1 \oplus s_1 \oplus \mathsf{Lsb}(H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}))$. $\mathcal{S}$ sends y_1 to $\mathcal{F}_{\mathsf{HaAND}}$ on behalf of $\mathsf{P_A}$ and receives v_1.

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

Honest parties: The box picks random $[x_1]_\mathsf{A}, [y_1]_\mathsf{A}, [z_1]_\mathsf{A}$, and $[x_2]_\mathsf{B}, [y_2]_\mathsf{B}, [z_2]_\mathsf{B}$, such that $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) = z_1 \oplus z_2$.

Corrupted parties:

1. A corrupted P_A gets to choose all its randomness. Furthermore, it can send g to the box trying to guess x_2. If $g \neq x_2$ the box output fail and terminates, otherwise the box proceeds as normal.
2. A corrupted P_B gets to choose all its randomness. Furthermore, it can send g to the box trying to guess x_1. If $g \neq x_1$ the box output fail and terminates, otherwise the box proceeds as normal.

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

Figure 5: Functionality $\mathcal{F}_{\mathsf{LaAND}}$ for leaky AND triple generation.

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

Honest parties: The box picks random $[x_1]_\mathsf{A}, [y_1]_\mathsf{A}, [z_1]_\mathsf{A}$, and $[x_2]_\mathsf{B}, [y_2]_\mathsf{B}, [z_2]_\mathsf{B}$, such that $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) = z_1 \oplus z_2$.

Corrupted parties: A corrupted P_A gets to choose all its randomness.

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

Figure 6: Functionality $\mathcal{F}_{\mathsf{aAND}}$ for generating AND triples

3. $\mathcal{S}$ computes $H_{x_1} := \mathsf{Lsb}(H(\mathsf{K}[x_1] \oplus x_1\Delta_\mathsf{B})) \oplus v_1 \oplus s_1$ and picks $H_{1\oplus x_1}$ randomly, and sends (H_0, H_1) to P_A.

Honest P_B has the same output according to the correctness proof. It is easy to see that the first two steps are perfect simulation. The last step is also a perfect simulation: the joint distribution of (H_0, H_1) and P_B's output is perfectly indistinguishable. 1) P_A only knows either $\mathsf{K}[x_1]$ or $\mathsf{K}[x_1] \oplus \Delta_B$, which means $H_{x_1 \oplus 1}$ remains random as long as H is a random oracle. 2) P_A obtains from H_{x_1} $v_1 \oplus s_1$, which is the same for both hybrids.

Malicious P_B. The simulation is essentially the same as the case when P_A is malicious (observing that step 2 and step 3 can be done in any order). $\square$

6.2 New TinyOT Protocol

Assuming that two parties hold $[x_1]_\mathsf{A}, [y_1]_\mathsf{A}$, $[x_2]_\mathsf{B}, [y_2]_\mathsf{B}$. In the original TinyOT protocol, to compute $(x_1 \oplus x_2)(y_1 \oplus y_2)$, P_A and P_B compute $[x_1 y_1]_\mathsf{A}$, $[x_2 y_2]_\mathsf{B}$, $[x_1 y_2 + r]_\mathsf{A}$ and $[x_2 y_1 + r]_\mathsf{B}$ separately, with some random $r \in \{0, 1\}$, using various authenticated constructions proposed in their paper. Computing each entry separately incurs a lot of unnecessary cost. We observe that it is possible to compute a whole AND gate directly. Similar to the original TinyOT protocol, we propose a "leaky AND" protocol (Π_{LaAND}), where the adversary is allowed to perform selective-failure attack on one input, and later use bucketing to eliminate such leakage (Π_{aAND}). In the following, we will first discuss the intuition of the protocol. The full protocol description is in Figure 7 and Figure 8.

Protocol Π_{LaAND}

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. $\mathsf{P_A}$ sends y_1 to $\mathcal{F}_{\mathsf{HaAND}}$, $\mathsf{P_B}$ sends y_2 to $\mathcal{F}_{\mathsf{HaAND}}$, which sends v_1 to $\mathsf{P_A}$ and v_2 to $\mathsf{P_B}$.
3. $\mathsf{P_A}$ computes $u = v_1 \oplus x_1 y_1$ 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 the 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) $\mathsf{P_A}$ randomly picks a κ-bit string R and computes
   $V_0 := H(\mathsf{M}[x_1], \mathsf{M}[z_1])$ $\quad$ $V_1 := H(\mathsf{M}[x_1], \mathsf{M}[z_1] \oplus \mathsf{M}[y_1])$
   $W_{0,0} := H(\mathsf{K}[x_2]) \oplus V_0 \oplus R$ $\quad$ $W_{0,1} := H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_1 \oplus R$
   $W_{1,0} := H(\mathsf{K}[x_2]) \oplus V_1 \oplus U \oplus R$ $\quad$ $W_{1,1} := H(\mathsf{K}[x_2] \oplus \Delta_\mathsf{A}) \oplus V_0 \oplus U \oplus R$
   (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 the 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) $\mathsf{P_B}$ randomly picks a κ-bit string R and computes
   $V_0 := H(\mathsf{M}[x_2], \mathsf{M}[z_2])$ $\quad$ $V_1 := H(\mathsf{M}[x_2], \mathsf{M}[z_2] \oplus \mathsf{M}[y_2])$
   $W_{0,0} := H(\mathsf{K}[x_1]) \oplus V_0 \oplus R$ $\quad$ $W_{0,1} := H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}) \oplus V_1 \oplus R$
   $W_{1,0} := H(\mathsf{K}[x_1]) \oplus V_1 \oplus U \oplus R$ $\quad$ $W_{1,1} := H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B}) \oplus V_0 \oplus U \oplus R$
   (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

6.3 Intuition

Compute the triple in the honest case. The first step of the protocol is to generate the triple securely assuming that both parties are honest. Since x_1, y_1, z_1, x_2, y_2 are all random, we just need $\mathsf{P_B}$ to learn $z_2 = (x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \oplus z_1$. Our idea is to use the $\mathcal{F}_{\mathsf{HaAND}}$ to compute the cross terms. Note that, because y_1, y_2 are not authenticated in $\mathcal{F}_{\mathsf{HaAND}}$, a malicious party can perform a selective failure attack by switching the value of y's. If there is no abort, it means that $x_1 \oplus x_2 = 0$. Similarly, $\mathsf{P_A}$ can also flip u (or similarly, $\mathsf{P_B}$ can flip d) to guess if $x_1 \oplus x_2 = 1$. Such attacks on x's are allowed in the leaky functionality and will be eliminated by bucketing.

Protocol Π_{aAND}

1. $\mathsf{P_A}$ and $\mathsf{P_B}$ call $\mathcal{F}_{\mathsf{LaAND}}$ $\ell' = \ell B$ times and obtains $\{[x_1^i]_{\mathsf{A}}, [y_1^i]_{\mathsf{A}}, [z_1^i]_{\mathsf{A}}, [x_2^i]_{\mathsf{B}}, [y_2^i]_{\mathsf{B}}, [z_2^i]_{\mathsf{B}}\}_{i=1}^{\ell'}$.
2. $\mathsf{P_A}$ and $\mathsf{P_B}$ randomly partition all objects into ℓ buckets, each with B objects.
3. For each bucket, two parties combine B Leaky ANDs into one non-leaky AND. To combine two leaky ANDs, namely $([x_1']_{\mathsf{A}}, [y_1']_{\mathsf{A}}, [z_1']_{\mathsf{A}}, [x_2']_{\mathsf{B}}, [y_2']_{\mathsf{B}}, [z_2']_{\mathsf{B}})$ and $[x_1'']_{\mathsf{A}}, [y_1'']_{\mathsf{A}}, [z_1'']_{\mathsf{A}}, [x_2'']_{\mathsf{B}}, [y_2'']_{\mathsf{B}}, [z_2'']_{\mathsf{B}}$
   (a) Two parties reveal $d' := y_1' \oplus y_1'', d'' = y_2' \oplus y_2''$ with their MAC checked, and compute $d := d' \oplus d''$.
   (b) Set $[x_1]_{\mathsf{A}} := [x_1']_{\mathsf{A}} \oplus [x_1'']_{\mathsf{A}}$, $[x_2]_{\mathsf{B}} := [x_2']_{\mathsf{B}} \oplus [x_2'']_{\mathsf{B}}$, $[y_1]_{\mathsf{A}} := [y_1']_{\mathsf{A}}$, $[y_2]_{\mathsf{A}} := [y_2']_{\mathsf{A}}$, $[z_1]_{\mathsf{A}} := [z_1']_{\mathsf{A}} \oplus [z_1'']_{\mathsf{A}} \oplus d[x_1'']_{\mathsf{A}}$, $[z_2]_{\mathsf{B}} := [z_2']_{\mathsf{B}} \oplus [z_2'']_{\mathsf{B}} \oplus d[x_2'']_{\mathsf{B}}$.

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

Figure 8: Protocol Π_{aAND} instantiating $\mathcal{F}_{\mathsf{aAND}}$.

Verifying the correctness. After the above steps, the correctness is not guaranteed with malicious security: a malicious party can corrupt the correctness of an AND triple. Therefore, both parties need to check the correctness of the output. In the protocol, we design a verification protocol that checks the correctness while allowing a malicious party to perform a selective-failure attack on x values.

The initial idea is to adopt the check from TinyOT to our case. 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 to check $y_1 \oplus z_1 = y_2 \oplus z_2$. However, an obvious problem is that no party knows the value of $x_1 \oplus x_2$. To solve this problem, when $\mathsf{P_B}$ checks the correctness, we let $\mathsf{P_B}$ construct the checking depending on the value of x_2. $\mathsf{P_A}$ will perform the checking twice, as if x_2 is 0 and 1.

For example, using the notation in the protocol, when $x_1 = 0$, $\mathsf{P_A}$ computes V_0, V_1. $\mathsf{P_A}$ and $\mathsf{P_B}$ should have performed an equality check between V_{x_2} and T_{x_2}. All different cases (depending on the value of x_1 and x_2) are summarized in the following table.

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

However, $\mathsf{P_A}$ should not learn x_2, while $\mathsf{P_B}$ should not learn $V_{1\oplus x_2}$. One idea is to let $\mathsf{P_A}$ "encrypt" the response (V_0, V_1) such that $\mathsf{P_B}$ can only learn the response for the value of x_2 (V_{x_2}), then $\mathsf{P_B}$ can compare locally. (This is possible because $\mathsf{P_B}$'s bit x_2 is authenticated by $\mathsf{P_A}$). However, the problem is that $\mathsf{P_A}$ is not able to learn the outcome of the comparison. To solve this, we let $\mathsf{P_A}$ send encrypted $V_0 \oplus R$ and $V_1 \oplus R$ for some random R such that $\mathsf{P_B}$ learns $V_{x_2} \oplus R$, and learns R from it. Now $\mathsf{P_A}$ and $\mathsf{P_B}$ can check the equality on R using the $\mathcal{F}_{\mathsf{EQ}}$ functionality in the TinyOT paper that allows both parties get the outcome. Note that this allows $\mathsf{P_A}$ to perform an additional selective-failure attack on x_2, by sending some corrupted encrypted values. This does not introduce additional leakage, since x_2 is allowed to be learnt by $\mathcal{A}$ anyway. Now $\mathcal{A}$ is allowed to guess x_2 twice, once in step 4 and once in step 5. If the guesses are inconsistent, it is guaranteed to abort.

Combining leaky ANDs. The above check is vulnerable to a selective-failure attack, from which a malicious party can learn the value of x_1/x_2 with one-half probability of being caught. In order to get rid of the leakage, bucketing is performed similar to TinyOT. Here, the key is to devise a way to combine leaky objects. Assuming that two triple are $([x_1']_{\mathsf{A}}, [y_1']_{\mathsf{A}}, [z_1']_{\mathsf{A}}, [x_2']_{\mathsf{B}}, [y_2']_{\mathsf{B}}, [z_2']_{\mathsf{B}})$ and

$[x_1'']_\mathsf{A}, [y_1'']_\mathsf{A}, [z_1'']_\mathsf{A}, [x_2'']_\mathsf{B}, [y_2'']_\mathsf{B}, [z_2'']_\mathsf{B}$. Note that for each triple, only x_1, x_2 can be leaked. Therefore, one natural way is to set $[x_1]_\mathsf{A} := [x_1']_\mathsf{A} \oplus [x_1'']_\mathsf{A}, [x_2]_\mathsf{B} := [x_2']_\mathsf{B} \oplus [x_2'']_\mathsf{B}$. By doing this, $[x_1]_\mathsf{A}, [x_2]_\mathsf{B}$ are non-leaky as long as one triple is non-leaky. We can also set $[y_1]_\mathsf{A} := [y_1']_\mathsf{A}, [y_2]_\mathsf{B} := [y_2']_\mathsf{B}$ and reveal the bit $d := y_1' \oplus y_2' \oplus y_1'' \oplus y_2''$, since y's bits are all private. Now 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 could just set $[z_1]_\mathsf{A} := [z_1']_\mathsf{A} \oplus [z_1'']_\mathsf{A} \oplus d[x_1'']_\mathsf{A}$, $[z_2]_\mathsf{A} := [z_2']_\mathsf{A} \oplus [z_2'']_\mathsf{A} \oplus d[x_2'']_\mathsf{A}$. The security of this bucketing and merging can be proved as in [NNOB12, Appendix I].

6.4 Proof Sketch

In the following, we will discuss from a high-level view how the proof works for the new TinyOT protocol. We will focus on the security of Π_{LaAND} protocol, since the security of Π_{aAND} is fairly straightforward given the proof in the original paper [NNOB12].

Correctness

Without loss of generality, we want to show that if both players followed the protocol then in step 4.e that $W_{x_1,x_2} \oplus \mathsf{M}[x_2] \oplus T_{x_2} = R$. Checks in step 5 are perfectly symmetric to ones in step 4. We will proceed on a case per case basis.

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

The value of x_1, x_2 means that $\mathsf{M}[x_1] = \mathsf{K}[x_1]$ and that $\mathsf{M}[x_2] = \mathsf{K}[x_2]$. 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 equation holds based on the following:

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

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 $\mathsf{M}[x_2] = \mathsf{K}[x_2] \oplus \Delta_\mathsf{B}$. $x_1 \oplus x_2 = 1$ also implies that

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

The equation holds based on the following:

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

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 that $\mathsf{M}[z_1] \oplus \mathsf{M}[y_1] = \mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_{\mathsf{B}}$, which will be used to prove the following:

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

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 that $\mathsf{M}[z_1] = \mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}}$, which will be used to prove the following:

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

Unforgeability

Lemma 6.2. *If* $(x_1 \oplus x_2) \wedge (y_1 \oplus y_2) \neq (z_1 \oplus z_2)$ *then the protocol will result in an abort except with negligible probability.*

We will proceed on a case per case basis. We assume that P_B is honest and that the adversary corrupts P_A. By symmetry, this would also show that the protocol would abort when P_B is corrupt and P_A is honest.

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

The adversary to pass the test 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 $z_1 \oplus z_2 = 1$, the last line requires the adversary to compute $\mathsf{K}[z_1] \oplus z_2\Delta_{\mathsf{B}} = \mathsf{M}[z_1] \oplus \Delta_{\mathsf{B}}$. This is equivalent to forging a mac and is thus 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. This implies that an adversary cannot pass the test.

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

The adversary to pass the test 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 is equivalent to forging a mac tag which is 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. This implies that an adversary cannot pass the test.

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

The adversary to pass the test 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 is equivalent to forging a mac tag which is 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 $y_1 \oplus y_2 \oplus z_1 \oplus z_2 = 1$ then $\mathsf{K}[y_1] \oplus \mathsf{K}[z_1] \oplus (y_2 \oplus z_2)\Delta_{\mathsf{B}} = \mathsf{M}[y_1] \oplus \mathsf{M}[z_1] \oplus \Delta_{\mathsf{B}}$ This is also infeasible. This implies that an adversary cannot pass the test.

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

The adversary to pass the test 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 is equivalent to forging a mac tag which is 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 $z_1 \oplus z_2 = 1$ then $\mathsf{K}[z_1] \oplus z_2\Delta_\mathsf{B} = \mathsf{M}[z_1] \oplus \Delta_\mathsf{B}$. Thus, this is also infeasible.

Completed proof

Now we will proceed with the complete proof.

Lemma 6.3. *The protocol in Figure 7 securely implements the functionality in Figure 5 against corrupted $\mathsf{P_A}$ in the $(\mathcal{F}_\mathsf{abit}, \mathcal{F}_\mathsf{HaAND}, \mathcal{F}_\mathsf{EQ})$-Hybrid model.*

Proof. We will construct a simulator as follows:

1 $\mathcal{S}$ interacts with $\mathcal{A}$ and 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 $\mathcal{A}$ sent to $\mathcal{F}_\mathsf{abit}$. $\mathcal{S}$ picks a random bit s, sets $\mathsf{K}[z_2] := \mathsf{K}[r] \oplus s\Delta_\mathsf{A}$, and sends $(x_1, \mathsf{M}[x_1])$, $(y_1, \mathsf{M}[y_1]), (z_1, \mathsf{M}[z_1])$, $\mathsf{K}[x_2], \mathsf{K}[y_2], \mathsf{K}[z_2], \Delta_\mathsf{A})$ to $\mathcal{F}_\mathsf{LaAND}$, which sends $(x_2, \mathsf{M}[x_2]), (y_2, \mathsf{M}[y_2])$, $(z_2, \mathsf{M}[z_2]), \mathsf{K}[x_1]$, $\mathsf{K}[y_1], \mathsf{K}[z_1], \Delta_\mathsf{B})$ 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, an 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

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

Table 5: Least number of AND gates needed in the bucketing, for different bucket sizes and statistical security parameters.

sent in the real protocol. This is also a perfect simulation unless $\mathsf{P_A}$ gets Δ_B: both R and one of $H(\mathsf{K}[x_1])$ and $H(\mathsf{K}[x_1] \oplus \Delta_\mathsf{B})$ are random.

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

Lemma 6.4. *The protocol in Figure 7 securely implements the functionality in Figure 5 against corrupted $\mathsf{P_B}$ in the $(\mathcal{F}_\mathsf{abit}, \mathcal{F}_\mathsf{HaAND}, \mathcal{F}_\mathsf{EQ})$-Hybrid model.*

Proof. We will construct a simulator as follows:

1. $\mathcal{S}$ interacts with $\mathcal{A}$ and receive $(x_2, \mathsf{M}[x_2]), (y_2, \mathsf{M}[y_2]), (r, \mathsf{M}[r]), \mathsf{K}[x_1], \mathsf{K}[y_1], \mathsf{K}[z_1], \Delta_\mathsf{B}$ that $\mathcal{A}$ sent to $\mathcal{F}_\mathsf{abit}$. $\mathcal{S}$ picks a random bit s, sets $(z_2, \mathsf{M}[z_2]) := (r \oplus s, \mathsf{M}[z_2] \oplus s\Delta_\mathsf{B})$, and sends $(x_2, \mathsf{M}[x_2]), (y_2, \mathsf{M}[y_2]), (z_2, \mathsf{M}[z_2]), \mathsf{K}[x_1], \mathsf{K}[y_1], \mathsf{K}[z_1])$ to $\mathcal{F}_\mathsf{LaAND}$, which sends $(x_1, \mathsf{M}[x_1]), (y_1, \mathsf{M}[y_1]), (z_1, \mathsf{M}[z_1]), \mathsf{K}[x_2], \mathsf{K}[y_2], \mathsf{K}[z_2])$ to $\mathsf{P_B}$.

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

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

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

6.5 More optimizations.

Note that the protocol description in Figure 7 does not include all possible optimizations for ease of understanding. In the following we will briefly discuss additional optimizations.

1. For clarity, R was chosen randomly in Π_LaAND. It is possible to perform garbled row reduction so that $W_{0,0}, W_{1,0}$ are zero. This saves two ciphertexts per leaky AND.

2. Only ρ bits of the R and U values need to be sent.

Circuit	n_1	n_2	n_3	$\|\mathcal{C}\|$
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 6: **Circuits used in our evaluation.**

3. Since the efficiency depends on the bucket size $B = \rho/\log|\mathcal{C}|$, we calculated the smallest circuit size needed for each bucket size based on the exact formula, so that the bucket size can be minimized. Table 5 shows the least number of AND gates needed in order to use different bucket size(B), under different statistical security parameter (ρ).

7 Extensions and Optimizations

Reducing the size of the authenticated garbled table. In the original protocol, all MACs and keys are κ-bit values, which may not always be necessary. For ρ-bit statistical security, $\mathsf{M}[r_{00}]$ encrypted in step 4(d) only needs to be of length ρ. Further, the bits $r_{\gamma,i}$ need not be put in the garbled table, since the MAC $\mathsf{M}[r_{\gamma,i}]$ is already enough for $\mathsf{P_B}$ to learn and validate the bit. This reduces the size of a garbled table from $8\kappa + 4$ bits to $4(\kappa + \rho)$ bits.

Partial garbled row reduction. Even with the above optimization, the value $\mathsf{L}_{\gamma,0}$ is still uniform, which means we can further reduce the size of garbled tables using ideas similar to garbled row reduction [PSSW09]. In detail, instead of picking $\mathsf{L}_{\gamma,0}$ randomly, it will be set such that $\mathsf{L}_{\gamma,0} = H(\mathsf{L}_{\alpha,0}, \mathsf{L}_{\beta,0}, \gamma, 0)[0 : \kappa]$, where $X[0 : \kappa]$ refers to the κ least-significant bits of a string X.

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

8 Evaluation

8.1 Implementation and Evaluation Setup

We implement our protocol to verify its efficiency. In the evaluation below, the computational security parameter is set to $\kappa = 128$, and the statistical security parameter is set to $\rho = 40$. Garbling and related operations are implemented using fixed-key AES-NI operations as in Bellare et al. [BHKR13]. Multithreading, Streaming SIMD Extensions (SSE), and Advanced Vector Extensions (AVX) are also used to improve performance whenever possible.

Our implementation consists mainly of three parts:

1. **Authenticated bits.** The protocol to compute authenticated bits is very similar to random OT extension [NNOB12]. Therefore, we adopt the most recent OT extension protocol by

	LAN				WAN			
	Ind. Phase	Dep. Phase	Online	Total	Ind. Phase	Dep. Phase	Online	Total
AES [WMK17]	-	28 ms	14 ms	42 ms	-	425 ms	416 ms	841 ms
AES [NST17]	89.6 ms	13.2 ms	1.46 ms	104.3 ms	1882 ms	96.7 ms	83.2 ms	2061.9 ms
Here	10.9 ms	4.78 ms	0.93 ms	16.6 ms	821 ms	461 ms	77.2 ms	1359.2 ms
SHA1 [WMK17]	-	139 ms	41 ms	180 ms	-	1414 ms	472 ms	1886 ms
Here	41.4 ms	21.3 ms	3.6 ms	66.3 ms	1288 ms	603 ms	78.4 ms	1969.4 ms
SHA256 [WMK17]	-	350 ms	84 ms	434 ms	-	2997 ms	514 ms	3511 ms
SHA256 [NST17]	478.5 ms	164.4 ms	11.2 ms	654.1 ms	2738 ms	350 ms	93.9 ms	3182 ms
Here	96 ms	51.7 ms	9.3 ms	157 ms	1516 ms	772 ms	88 ms	2376 ms

Table 7: Comparison in the single-execution setting

		LAN				WAN			
	τ	Ind. Phase	Dep. Phase	Online	Total	Ind. Phase	Dep. Phase	Online	Total
[RR16]	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
[NST17]	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 8: **Evaluation of AES in the amortized setting.** τ is the number of executions.

Keller et al. [KOS15] along with the optimization of Nielsen et al. [NST17]. The resulting protocol requires $\kappa + \rho$ bits of communication per authenticated bit.

2. $\mathcal{F}_{\mathsf{Pre}}$ **functionality.** In order to improve the running time, we spawn multiple threads that each generate a set of leaky AND gates. After all leaky AND gates are generated, bucketing and combining are done in a single thread.

3. **Our protocol.** The function-independent phase invokes the above two parts to generate random AND triples with IT-MACs. In the function-dependent phase, these random AND triples are used to construct a single garbled table. Note that in the single-execution setting, we use only one thread to construct the garbled circuit; in the amortized setting, we use multiple threads, each constructing a different garbled circuit for the same function but different executions. The online phase is always done using a single thread.

Evaluation setup. Our evaluation focuses on two settings:

- LAN: Amazon EC2 with instance `c4.8xlarge` machines both in the North Virginia region connected with 10 Gbps bandwidth and less than 1ms roundtrip time.
- WAN: One machine in North Virginia and one in Ireland, both of which are of the type `c4.8xlarge`. Single thread communication bandwidth is about 224 Mbps; the maximum total bandwidth is about 3 Gbps with multiple threads.

	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 9: **More examples with a much larger range of input/circuit size.**

In Section 8.2, we first compare the performance of our protocol with previous protocols in similar settings; here we focus on three circuits commonly used by other works, including AES, SHA-1, and SHA-256 (details in Table 6). Our results show that these circuits may no longer be large enough to serve as the benchmark circuits for malicious 2PC. Therefore, in Section 8.3, we also show the performance of our protocol on some larger circuits (see Table 6). We will make these circuit files publicly available upon publication of our work. In Section 8.4 and Section 8.5, we study the scalability of the protocol and compare the concrete communication complexity of our protocol with prior work.

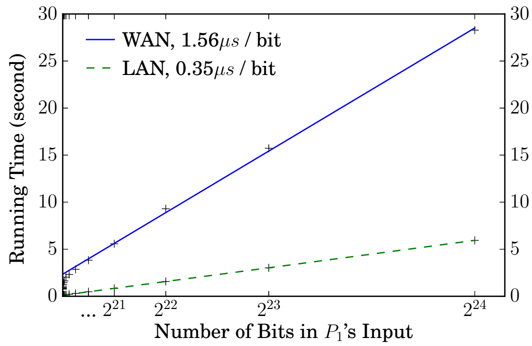
8.2 Comparison with Previous Work

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. [WMK17], 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. [NST17], which is based on gate-level cut-and-choose and is able to perform function-independent preprocessing. To make a fair comparison, we ran the implementation by Wang et al. using the same hardware; the results by Nielsen et al. are obtained from their paper, since the hardware configuration is the same. Our reported timings do not include the time for the base-OTs for the same reason as in [NST17]: the performance of base-OTs depends on the details of how the base-OTs are instantiated 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 [CO15]) takes about 20 ms in the LAN setting and 240 ms in the WAN setting.

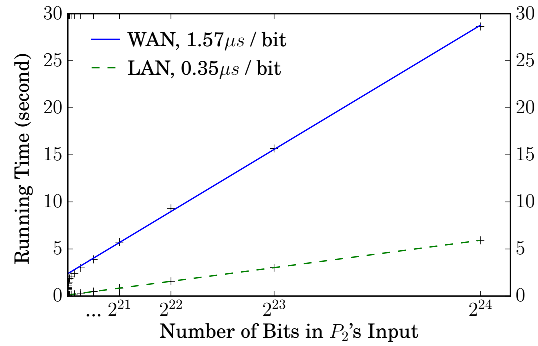
As shown in Table 7, our protocol performs better than previous protocols in terms of both overall cost and online time. Compared with the protocol by Wang et al., we achieve a speed up of 2.7× overall and an improvement of about 10× for online time. Compared with the protocol by Nielsen et al., the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

Amortized Setting. We observed that in the amortized setting, our protocol is also better than previous protocols. In particular, we achieve an improvement about 4.5× to 5.5× if only amortized over 32 executions. When the number of executions grows to 1024, [NST17] is no longer better than [RR16] in terms of total time but our protocol still outperform both protocol: in the LAN setting, the total cost is about the same as [RR16], but most of the computation are done in function-independent phase; in the WAN setting, we are 2× better than [RR16] in terms of total cost and 3× better in terms of online cost.

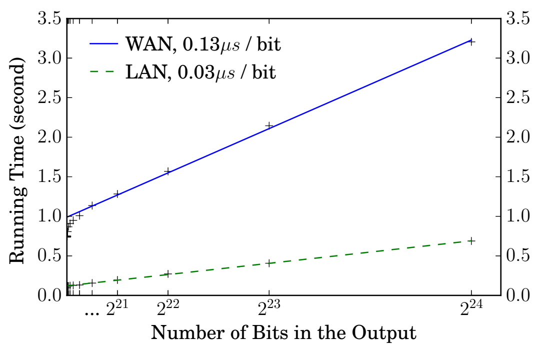
Comparison with Lindell et al. [LPSY15]. Since the protocol by Lindell et al. is not implemented, we perform a back-of-the-envelope calculation to argue that our protocol is faster. For a



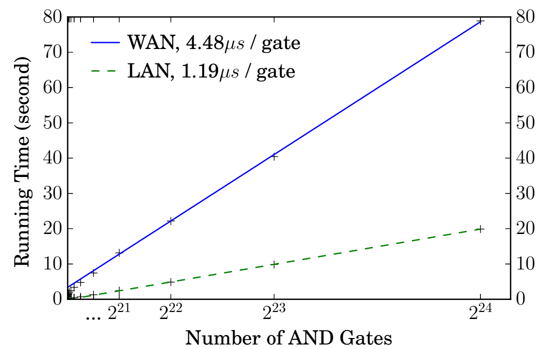
(a) Increasing $\mathsf{P_A}$'s input size (n_1).



(b) Increasing $\mathsf{P_B}$'s input size (n_2).



(c) Increasing output size (n_3).



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

Figure 9: **Scalability of our protocol.** Initially input sizes and output size are all set to 128 bit with a circuit of size 1024 gate. For each figure, one of the following values increases monotonically: $\mathsf{P_A}$'s input size, $\mathsf{P_B}$'s input size, output size, circuit size.

circuit of size $|\mathcal{C}|$, their protocol requires $5|\mathcal{C}|$ SPDZ multiplications. Over a 10 Gbps network, the recent work of Keller et al. [KOS16] can generate in principle 55,000 triples per second using an ideal implementation that fully saturates the network. Therefore, the best end-to-end speed their protocol can achieve in the two-party setting 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 [LR15]. 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, which is already 1.5× slower than our protocol.

The best linear-round protocol that allows amortization is by Damgård et al. [DLT14], which only supports *parallel execution* (where inputs to all executions need to be known at the same time). They report an amortized time for evaluating AES of 14.65 ms per execution, amortized over 680 execution. This is roughly in 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.

Circuit	n_1	n_2	n_3	$\|\mathcal{C}\|$
LAN	0.35	0.35	0.03	1.19
WAN	1.56	1.57	0.13	4.48

Table 10: Scalability of the protocol. All numbers in microseconds.

8.3 Larger Circuits

As we can see from the previous section, evaluating an AES circuit takes less time than generating the base-OT. This means that due to recent advances in 2PC, existing benchmark circuits are no longer large enough for a meaningful evaluation. We propose three new examples and evaluate their performance. The configuration of the circuits are shown in Table 6; we will briefly discuss the functionality of them:

- **Hamming Dist.** Each party inputs a bit string of length 1048576 bits; the output of the circuit is a 22-bit number containing the hamming distance of the two bit string from each party. The circuit complexity is $O(n)$ for n-bit strings.
- **Integer Mult.** Each party inputs a 2048-bit number; the circuit compute the multiplication of them, ignoring the high 2048 bits of the result. The circuit complexity is $O(n^2)$ for n bit numbers.
- **Sorting.** Each party inputs XOR-share of 4096 32-bit numbers; the circuit first XOR them to recover the underlying numbers and then sort the these numbers. The circuit complexity is $O(nl\log^2 n)$ to sort n numbers each with l bits.

Table 9 shows the performance of new examples described above. We can see that the difference of online time between LAN and WAN 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). According to the Table, our protocol is able to sort 4096 32-bit numbers in less than 14 seconds with an online time only 1 second. Other timings can be interpreted similarly.

8.4 Scalability

To explore the concrete performance of our protocol for circuits with different input, output and circuit sizes, we conduct a scalability evaluation: we start with a circuit with input and output sizes of 128 bits and 1024 AND gates and, at each time, increase one size monotonically up to 2^{24} bits/gates. The result of the evaluation is shown in Figure 9. Trend lines are also included to show the asymptotical performance. Since the bucket size of our protocol reduces as the circuit size increases, these lines are regression of the points when the bucket size is 3.

According to the figures, our implementation scales linearly in the input, output and circuit sizes as expected. We observe that, in the LAN setting, our protocol requires only 0.35 μs to process each input bit and 0.03 μs to process each output bit. Note that this is much better than circuit-level cut-and-choose protocols, mainly for two reasons: 1) Since only one garbled circuit is constructed, only one set of garbled labels need to be transferred; this is an improvement of ρ

Protocol	τ	Ind. Phase	Dep. Phase	Online
[RR16]	32	-	3.8 MB	25.8 KB
	128	-	2.5 MB	21.3 KB
	1024	-	1.6 MB	17.0 KB
[NST17]	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 11: Comparison of 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. In the setting with a simplex network, the total communication of this work and [RR16] should be doubled for a fair comparison.

times. 2) We do not need XOR-Tree or ρ-probe matrix to prevent selective failure, which can incur a huge cost when the input is large [WMK17].

The figures also show that, in the WAN setting, the ratios are about 3–4× lower than the ratios in the LAN setting. This roughly matches the ratio of network bandwidth between LAN and WAN settings.

8.5 Communication Complexity

We also record the amount of communication used in the protocol based on our implementation. In Table 11 we compare the amount of data sent from garbler to the evaluator with other related works. In detail, we focused on the AES circuit with different number of executions. Our total communication is 3× to 5× less than Nielsen et al.'s protocol. Furthermore, our cost in the *single-execution* setting is even half the cost of Nielsen et al.'s protocol when amortized with 1024 executions. Note that for protocols based on cut-and-choose, the total communication to send 40 AES garbled circuit is 8.7 MB, which is already higher than the total communication of our protocol in the single execution setting.

We also observe that our function dependent preprocessing is higher than 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. 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 the protocol of Rindal and Rosulek.

Acknowledgments

This material is based upon work supported by NSF awards #1111599 and #1563722; Samuel Ranellucci is also supported by NSF award #1564088. The author would like to thank Roberto Trifiletti, Yan Huang and Ruiyu Zhu for their helpful comments.

References

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *20th ACM Conf. on Computer and Communications Security (CCS)*, pages 535–548. ACM Press, 2013.

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In *Advances in Cryptology—Eurocrypt 2014*, volume 8441 of *LNCS*, pages 387–404. Springer, 2014.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—Crypto '91*, volume 576 of *LNCS*, pages 420–432. Springer, 1992.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security & Privacy*, pages 478–492. IEEE, 2013.

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *22nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 503–513. ACM Press, 1990.

[Bra13] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique. In *Advances in Cryptology—Asiacrypt 2013, Part II*, volume 8270 of *LNCS*, pages 441–463. Springer, 2013.

[CKMZ14] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 513–530. Springer, 2014.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. LNCS, pages 40–58, 2015.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *Advances in Cryptology—Crypto 2005*, volume 3621 of *LNCS*, pages 378–394. Springer, 2005.

[DLT14] Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the MiniMac protocol for secure computation. In *9th Intl. Conf. on Security and Cryptography for Networks (SCN)*, volume 8642 of *LNCS*, pages 398–415. Springer, 2014.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, 2012.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In *Advances in Cryptology—Eurocrypt 2013*, volume 7881 of *LNCS*, pages 537–556. Springer, 2013.

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In *9th Intl. Conf. on Security and Cryptography for Networks (SCN)*, volume 8642 of *LNCS*, pages 358–379. Springer, 2014.

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

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM Press, 1987.

[HEKM11] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium*. USENIX Association, 2011.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology—Crypto 2013, Part II*, volume 8043 of *LNCS*, pages 18–35. Springer, 2013.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 458–475. Springer, 2014.

[HZ15] Yan Huang and Ruiyu Zhu. Revisiting LEGOs: Optimizations, analysis, and their limit. Cryptology ePrint Archive, Report 2015/1038, 2015. http://eprint.iacr.org/2015/1038.

[IKOS09] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Extracting correlations. In *50th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 261–270. IEEE, 2009.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer—efficiently. In *Advances in Cryptology—Crypto 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, 2008.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, 2014.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *Advances in Cryptology—Crypto 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *23rd ACM Conf. on Computer and Communications Security (CCS)*, pages 830–842. ACM Press, 2016.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *35th Intl. Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology—Crypto 2013, Part II*, volume 8043 of *LNCS*, pages 1–17. Springer, 2013.

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In *Advances in Cryptology—Crypto 2011*, volume 6841 of *LNCS*, pages 259–276. Springer, 2011.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology—Eurocrypt 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, 2007.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *8th Theory of Cryptography Conference—TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, 2011. Available at `http://eprint.iacr.org/2010/284`.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In *Advances in Cryptology—Crypto 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, 2015.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, 2014.

[LR15] Yehuda Lindell and Ben Riva. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *22nd ACM Conf. on Computer and Communications Security (CCS)*, pages 579–590. ACM Press, 2015.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *Proc. 13th USENIX Security Symposium*, pages 287–302. USENIX Association, 2004.

[MOR16] Payman Mohassel, Ostap Orobets, and Ben Riva. Efficient server-aided 2PC for mobile phones. *Proc. Privacy Enhancing Technologies*, 2016(2):82–99, 2016.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *6th Theory of Cryptography Conference—TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, 2009.

[NO16] Jesper Buus Nielsen and Claudio Orlandi. Cross and clean: Amortized garbled circuits with constant overhead. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part I*, pages 582–603, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[NST17] Jesper Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant-round maliciously secure 2PC with function-independent preprocessing using LEGO. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[PSSW09] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. In *Advances in Cryptology—Asiacrypt 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *Proc. 25th USENIX Security Symposium*, pages 297–314. USENIX Association, 2016.

[SS11] Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In *Advances in Cryptology—Eurocrypt 2011*, volume 6632 of *LNCS*, pages 386–405. Springer, 2011.

[WMK17] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Faster two-party computation secure against malicious adverstries in the single-execution setting. In *Advances in Cryptology—Eurocrypt 2017*, LNCS. Springer, 2017.

[Yao86] Andrew C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology—Eurocrypt 2015*, LNCS, pages 220–250. Springer, 2015.