Silent Two-party Computation Assisted by Semi-trusted Hardware

Yibiao Lu
Zhejiang University
luyibiao@zju.edu.cn

Bingsheng Zhang
Zhejiang University
bingsheng@zju.edu.cn

Weiran Liu
Alibaba Group
weiran.lwr@alibaba-inc.com

Lei Zhang
Alibaba Group
zongchao.zl@taobao.com

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

***Abstract*—With the advancement of the *trusted execution environment* (TEE) technologies, hardware-supported secure computing becomes increasingly popular due to its efficiency. During the protocol execution, typically, the players need to contact a third-party server for remote attestation, ensuring the validity of the involved trusted hardware component, such as Intel SGX, as well as the integrity of the computation result. When the hardware manufacturer is not fully trusted, sensitive information may be leaked to the third-party server through backdoors, side-channels, steganography, and kleptography, etc. In this work, we introduce a new security notion called *semi-trusted hardware model*, where the adversary is allowed to passively and/or maliciously corrupt the hardware component. Therefore, she can learn the input of the hardware component and might also tamper the output. We show that two-party computation (2PC) can still be significantly sped up in this new model. When the semi-trusted hardware is instantiated by Intel SGX, to generate 10k random OT's, our protocol is 24X and 450X faster than the EMP-IKNP-ROT in the LAN and WAN setting, respectively. For the AES-128, SHA-256, and SHA-512 evaluation, our protocol is 4.9-5.4X and 40-46X faster than the EMP-SH2PC in the LAN and WAN setting, respectively. We also show how to achieve malicious security with little overhead.**

1. Introduction

In secure multi-party computation (MPC), two or more players want to collectively compute a function and receive its output without revealing their inputs to the other players. In the past decades, MPC has gradually transitioned from theory to practice, and it has been widely used in many security critical real-world applications, such as private set intersection and secure auction. In spite of its success, MPC is still not efficient for complicated real-time tasks due to its computational overhead and high communication cost. Meanwhile, recent development of trusted execution environment (TEE) technologies, such as Intel SGX and ARM TrustZone, enables a new approach for privacy-preserving computation. Hardware-supported secure computing can greatly accelerate an MPC process by avoiding expensive cryptographic operations. However, this kind of construction introduces additional hardware setup assumptions that require new trust roots, e.g., Intel. Recent exposure of Intel source code [1] raises a security concern on possible backdoors contained in its design. Moreover, many side-channel and micro-architecture attacks [2]–[6] have been discovered to compromise the security guarantees provided by trusted hardware components.

When the hardware manufacturer is not fully trusted, sensitive information may be leaked through backdoors, side-channels, steganography and kleptography, etc. For instance, Intel SGX uses the remote attestation mechanism to ensure the validity of the enclave execution environment and the integrity of the computation result. More specifically, Intel's (anonymous) attestation is based on an anonymous group signature scheme called Intel Enhanced Privacy ID (EPID) [7]. To verify that an outcome is computed by a pre-agreed program in a genuine SGX, Quoting Enclave (QE) will produce a quote by signing the report with the group signature. The users then need to contact the remote Intel Attestation Service (IAS) (or some other alternative servers) for verification. If Intel is malicious, sensitive information may be leaked from the SGX component to the IAS through the signatures, using for example kleptography techniques. (Currently, Intel SGX uses a 4000-bit RSA signature scheme.) That means the input of SGX might be revealed to the adversary during the protocol execution.

When the hardware provider is not allied with the MPC participants, is it possible to still use potentially malicious leaky hardware components to accelerate MPC executions with privacy assurance?[1] In this work, we answer this question affirmatively.

New model. We introduce a new semi-trusted hardware model, where the adversary $\mathcal{A}$ is allowed to passively or maliciously corrupt the hardware ideal functionality $\mathcal{F}_{\mathrm{HW}}$. $\mathcal{F}_{\mathrm{HW}}$ is parameterized with a probabilistic polynomial time (PPT) interactive Turing machine (ITM) M, which specifies its functionality. When the hardware functionality $\mathcal{F}_{\mathrm{HW}}$ is passively corrupted, the adversary $\mathcal{A}$ can learn all the incoming messages received by $\mathcal{F}_{\mathrm{HW}}$; when $\mathcal{F}_{\mathrm{HW}}$ is maliciously corrupted, in addition to leaning the incoming messages, the adversary $\mathcal{A}$ can replace the original M with an arbitrary ITM M^*; namely, $\mathcal{A}$ can fully control the execution of $\mathcal{F}_{\mathrm{HW}}$.

We formalize our model in the Universal Composibility (UC) framework [8]. In this security framework, the adversary $\mathcal{A}$ is allowed to control the network and corrupt some machines (ideal

Bingsheng Zhang is the corresponding author.

[1] In this work, we assume trusted hardware components can be securely realized in the coming future, and we don't address the information leakage problem from SGX to the host PC during the execution via the side-channel attacks.

functionalities and/or MPC players). In this work, we focus on two-party computation. We circumvent some impossibility results, and we introduce some restrictions to the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$ to enable efficient constructions. More precisely, we assume the hardware manufacturer does not collude with the MPC players; therefore, we restrict the adversary $\mathcal{A}$ to only corrupt either the semi-trusted hardware functionality $\mathcal{F}_{\mathrm{HW}}$ or the player(s) P_1 (and/or P_2). Notice that our model is different from the *server-aided* model [9], where the server cannot be maliciously corrupted. Therefore, protocols like authenticated garbling [10] cannot be naively adopted in the malicious setting. Moreover, our main observation is that $\mathcal{F}_{\mathrm{HW}}$ can be instantiated by local trusted hardware components to save communication. We note that our constructions can be easily modified to ensure security in the scenario where P_1 and $\mathcal{F}_{\mathrm{HW}}$ are colluding[2]; while we don't have an efficient solution to handle the collusion between P_2 and $\mathcal{F}_{\mathrm{HW}}$ yet, as this setting could be reduced to standard two-party computation without assistance of $\mathcal{F}_{\mathrm{HW}}$ in practice.

Our constructions. We propose a new type of two-party computation (2PC) protocols called *silent MPC* that uses semi-trusted hardware to significantly reduce the communication between the 2PC players. The main idea is to use semi-trusted hardware for those MPC computation that does not depend on the actual protocol inputs; thus no sensitive information is leaked to the hardware components. Take random OT (ROT) generation as an example, assume the Receiver uses an SGX-enabled machine, while there is no special hardware requirement to the Sender. During the ROT protocol, the Sender only needs to send a random seed k_1^0 to the Receiver's SGX enclave via a secure channel, and the Receiver also sends a random seed k_2^0 to the enclave locally. Both parties can then generate polynomially many ROT instances without any further communication. Namely, for a ROT instance, the Sender locally computes $R^0_{\mathsf{ctr}} \leftarrow \mathsf{PRF}_{k_1^0}(\mathsf{ctr}, 0)$ and $R^1_{\mathsf{ctr}} \leftarrow \mathsf{PRF}_{k_1^0}(\mathsf{ctr}, 1)$ from the seed k_1^0 using some pseudo-random function PRF, where ctr is the counter; meanwhile, the SGX generates the ROT choice bit b_{ctr} from the seed k_2^0 using some pseudo-random number generator PRG, and then it computes $R^{b_{\mathsf{ctr}}}_{\mathsf{ctr}} \leftarrow \mathsf{PRF}_{k_1^0}(i, b_{\mathsf{ctr}})$. The SGX locally outputs $\{R^{b_{\mathsf{ctr}}}_{\mathsf{ctr}}\}_{\mathsf{ctr}}$ to the Receiver.

For garbled circuit (GC) evaluation, the communication between the 2PC players can also be dramatically reduced. Similarly, we assume the GC Evaluator uses an SGX-enabled machine, while there is no special hardware requirement to the GC Garbler. Note that the main cost of a GC-based 2PC protocol is the transmission of the garbled tables of the entire circuit. Analogously, during the GC protocol, the Garbler sends a random seed k_1^1 to the Evaluator's SGX enclave via a secure channel. The SGX can then internally generate the garbled tables and locally outputs them to the Evaluator without network communication. The only communication needed is for transmitting the input labels from the Garbler to the Evaluator. Hence, the overall communication is linear with the input size and independent of the circuit size.

Remark. We would like to emphasize that naively using the secure hardware components, such as SGX, and a simulatable private garbling scheme in a blackbox fashion to prepare the ROT instances and GC tables in an offline phase won't result in a (UC) simulatable 2PC protocol. This is because the simulator cannot extract the malicious Evaluator's input in the offline phase, yet it needs to learn the MPC output (from the ideal functionality) to invoke the GC simulator (cf. Def. 2) to produce the (fake) GC tables as well as the (fake) decoding information in the real/hybrid world. As described in Sec. 4 later, the protocol should invoke the secure hardware component to execute certain tasks at the right moment along with the 2PC protocol execution.

In Sec. 6, we show how to further reduce the online communication in the semi-honest setting using the masked GC technique. The idea of using masks to hide the truth table of each gate can trace back to the point-and-permute technique [12]. In standard GC, all the masks are known to P_1, while in our scheme, the masks for P_2's input can be generated by the SGX, which is unknown to P_1. Therefore, P_2 can directly fetch its input labels without using OT.

To handle malicious adversaries, the 2PC players need to check the correctness of the garbled circuit and ROT instances. Intuitively, we let P_1 and SGX independently generate the garbled circuit using the same seed. During the protocol, SGX outputs the garbled circuit to P_2, and P_1 only sends the corresponding hash digest. Therefore, P_2 can check the consistency of the garbled circuit generated by SGX and P_1 to ensure correctness, as the adversary cannot simultaneously corrupt both P_1 and SGX. Similarly, for ROT, we let P_1 sends $H(R^0)$ and $H(R^1)$ to P_2 to ensure the correctness of R^b. Moreover, we also use hash to prevent P_1 from sending wrong GC labels, launching a selective failure attack for instance.

Efficiency. We mainly compare the performance of our protocols with the well-known EMP-toolkit maintained by Wang *et al.* [11]. Table. 1 shows the performance comparison between the passively secure IKNP [13] and Ferret [14] OT extension protocol implemented in EMP-toolkit [11] and our silent ROT protocol (semi-honest security). We perform the experiments on an SGX-enabled Dell OptiPlex 7080 equipped with an Intel Core 8700 CPU @ 3.20 GHz with 32 GB RAM in single thread. In the LAN setting (Bandwidth: 1Gbps, Delay: 1ms), our silent ROT protocol is 22-39X faster w.r.t. the sender's running time and 9-14X faster w.r.t. the receiver's running time than the EMP-IKNP-ROT; it is 18-1117X faster w.r.t. the sender's running time and 6-524X faster w.r.t. the receiver's running time than the EMP-Ferret-ROT. In the WAN setting (Bandwidth: 100Mbps, Delay: 25ms), our silent ROT protocol is 189-333X faster w.r.t. the sender's running time and 93-451X faster w.r.t. the receiver's running time than the EMP-IKNP-ROT; it is 20-1075X faster w.r.t. the sender's running time and 10-800X faster w.r.t. the receiver's running time than the EMP-Ferret-ROT.

Table. 2 shows the performance comparison between EMP-

[2] For instance, in the semi-honest setting, we just let P_1 and P_2 produce ROT's by themselves instead of using $\mathcal{F}_{\mathrm{HW}}$; for malicious setting, we can use the cut-and-choose technique to ensure GC correctness. Note that the overall communication can still be made independent to the output size and the circuit size.

TABLE 1: Performance comparison between EMP-toolkit [11] and our ROT protocol. Result obtained from SGX-enabled Dell OptiPlex 7080 (Intel Core 8700 CPU @ 3.20 GHz, 32 GB RAM, OS: Ubuntu 18.04 LTS).

# ROT	Network setting	Sender's running time (in ms)			Receiver's running time (in ms)		
		Ferret-ROT [11]	IKNP-ROT [11]	Our ROT	Ferret-ROT [11]	IKNP-ROT [11]	Our ROT
1×10^4	LAN (1Gbps, 1ms)	82.726	2.889	0.074	84.948	3.908	0.162
	WAN (100Mbps, 25ms)	84.945	26.331	0.079	135.221	76.358	0.169
1×10^5	LAN (1Gbps, 1ms)	84.721	17.790	0.780	86.321	19.355	1.575
	WAN (100Mbps, 25ms)	97.244	150.502	0.795	145.475	200.030	1.477
1×10^6	LAN (1Gbps, 1ms)	112.315	154.373	6.182	109.002	150.621	15.910
	WAN (100Mbps, 25ms)	128.484	1451.043	6.402	167.520	1495.294	16.032
1×10^7	LAN (1Gbps, 1ms)	1116.614	1507.961	51.616	1063.486	1451.562	103.937
	WAN (100Mbps, 25ms)	1244.835	13859.934	51.280	1240.504	13963.502	103.435
1×10^8	LAN (1Gbps, 1ms)	10062.935	15030.832	505.289	9531.965	14470.057	995.987
	WAN (100Mbps, 25ms)	10535.485	138028.607	501.757	10053.518	137034.187	980.795

TABLE 2: Performance comparison of the generation, transmission and evaluation process of the garbled circuit in the OT-based semi-honest setting 2PC protocol. Result obtained from the same experiment environment as in Table 1. It shows the running time (in ms) for evaluating AES-128, SHA-256, and SHA-512 circuits 1000 times, respectively.

Circuit	Network setting	EMP-SH2PC [11] running time (in ms)			Our 2PC protocol running time (in ms)		
		Garbler	Transmission	Evaluator	Garbler	Transmission	Evaluator (SGX+PC)
AES-128	LAN (1Gbps, 1ms)	246.557	1742.094	229.339	6.902	≈ 0	255.154 + 167.353
	WAN (100Mbps, 25ms)	265.919	18335.009	234.264	6.760	≈ 0	253.243 + 166.239
SHA-256	LAN (1Gbps, 1ms)	829.398	6135.087	776.880	19.656	≈ 0	842.603 + 580.064
	WAN (100Mbps, 25ms)	839.626	64433.208	777.284	20.362	≈ 0	844.843 + 580.685
SHA-512	LAN (1Gbps, 1ms)	2434.915	15745.170	2388.890	40.062	≈ 0	2544.834 + 1639.701
	WAN (100Mbps, 25ms)	2303.479	163362.579	2418.025	40.394	≈ 0	2533.623 + 1640.262

TABLE 3: Performance comparison of the computation process of the malicious setting 2PC protocol. Result obtained from the same experiment environment as in Table 1. It shows the running time (in ms) for evaluating AES-128, SHA-256, and SHA-512 circuits once, respectively.

Circuit	Network setting	EMP-AG2PC [11] running time (in ms)				Our running time (in ms)	
		Garb. offline	Garb. online	Eval. offline	Eval. online	Garbler	Evaluator
AES-128	LAN (1Gbps, 1ms)	94.744	5.185	92.055	5.193	7.315	10.518
	WAN (100Mbps, 25ms)	1345.708	53.440	1240.956	53.385	109.039	161.901
SHA-256	LAN (1Gbps, 1ms)	210.676	6.303	201.701	6.272	13.751	19.076
	WAN (100Mbps, 25ms)	2299.404	52.474	2196.297	52.440	119.712	175.740
SHA-512	LAN (1Gbps, 1ms)	435.581	9.634	423.302	9.593	27.327	38.110
	WAN (100Mbps, 25ms)	4095.115	56.471	4044.428	56.426	132.298	195.837

SH2PC [11] and our semi-honest setting silent 2PC protocol. (EMP-SH2PC provides an efficient semi-honest 2PC implementation based on Yao's GC protocol with half-gates [15] optimization.) We perform the experiments on this same machine as above. We test the garbling time, the garbled tables transmission time, and the evaluation time separately. Since in our protocol, the garbling process is performed in the SGX enclave at the evaluator side, we split the evaluator running time of our protocol into two parts: (i) the SGX running time and (ii) normal mode CPU running time. The garbler running time is the time to generate the input wire labels. We take the AES-128, SHA-256, and SHA-512 circuit evaluation as benchmarks. In the LAN setting, our silent 2PC protocol is 4.9-5.4X faster than the EMP-SH2PC. In the WAN setting, our silent 2PC protocol is 40-46X faster than the EMP-SH2PC.

Table. 3 shows the performance comparison between EMP-AG2PC [11] and our malicious setting silent 2PC protocol. (EMP-AG2PC implements an efficient maliciously secure two-party computation protocol, authenticated garbling [10].) We perform the experiments on this same machine as above. We take the AES-128, SHA-256, and SHA-512 circuit evaluation as benchmarks, and the results are the average of 100 tests. All the one-time expenses are omitted, e.g., creating enclave in our protocol and initialize $\mathcal{F}_{pre}$ in EMP-AG2PC. EMP-AG2PC consists of three computing phases: (i) function independent offline phase, (ii) function dependent offline phase and (iii) online phase. (i) and (ii) are collectively called offline phase. In the LAN setting, our silent 2PC protocol is 13-16X faster w.r.t. the garbler's running time and 9-11X faster w.r.t. the evaluator's running time than the EMP-AG2PC. In the WAN setting, our silent PC protocol is 12-31X faster w.r.t. the garbler's running time and 8-20X faster w.r.t. the evaluator's running time than the EMP-AG2PC.

2. Preliminaries

Notation. Throughout this paper, we use the following notations and terminologies. Let $\lambda \in \mathbb{N}$ be the security parameter. Denote the set $\{a, a+1, \ldots, b\}$ by $[a, b]$, let $[b]$ denote $[1, b]$, and let $\emptyset$ denote empty set. When A is an array, $|A|$ stands for the size of A in terms of the number of entries. We abbreviate *probabilistic polynomial time* as PPT, and *interactive Turing machine* as ITM. When S is a set, $s \leftarrow S$ stands for sampling s uniformly at random from S. When A is a randomised algorithm,

$y \leftarrow A(x)$ stands for running A on input x with a fresh random coin r. When needed, we denote $y := A(x;r)$ as running A on input x with the explicit random coin r. Let $\mathrm{poly}(\cdot)$ and $\mathsf{negl}(\cdot)$ be a polynomially-bounded function and negligible function, respectively. We assume each party has a unique PID. For readability, we refer P_i as the PID for the party P_i.

Suppose $f(x_1, x_2) = y$ is a function (circuit). Denote $f.n_1$ and $f.n_2$ as the input size of x_1 and x_2, respectively. Let $f.n = f.n_1 + f.n_2$. Denote $f.m$ as the size of the output y and $f.N$ as the overall wire number in f. For notation simplicity, we also use n_1, n_2, n, m, N to represent $f.n_1, f.n_2, f.n, f.m, f.N$ when there will be no ambiguity.

Garbling Scheme. As defined in [16], a garbling scheme GC consists of the following PPT algorithms $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$.

- $\mathsf{Gb}(1^\lambda, f)$ is the garbling algorithm that takes input as the security parameter $\lambda \in \mathbb{N}$ and a circuit f, and it returns a garbled circuit F, encoding information e, and decoding information d.
- $\mathsf{En}(e, x)$ is the encoding algorithm that takes input as the encoding information e and an input x, and it returns a garbled input X.
- $\mathsf{Ev}(F, X)$ is the evaluation algorithm that takes input as the garbled circuit F and the garbled input X, and it returns a garbled output Y.
- $\mathsf{De}(d, Y)$ is the decoding algorithm that takes input as the decoding information d and the garbled output Y, and it returns the plaintext output y.

A garbling scheme $\mathsf{GC} := (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ is called projective if e consists of $2f.n$ wire labels. For the i-th input bit, we denote the corresponding wire labels as (X_i^0, X_i^1). Let $e := \{(X_i^0, X_i^1)\}_{i\in[n]}$; the encoding algorithm $\mathsf{En}(e, x)$ simply outputs $X_i^{x[i]}, i \in [n]$, where $x[i]$ is the i-th bit of x.

Analogously, a garbling scheme is called output-projective if d consists of 2 labels for each output bits, which can be denoted as (Z_i^0, Z_i^1). Let $d := \{(Z_i^0, Z_i^1)\}_{i\in[m]}$; the decoding algorithm $\mathsf{De}(d, Y)$ outputs $y[i], i \in [m]$, where $y[i]$ is the i-th bit of y s.t. $Z_i^{y[i]} = Y_i$.

In this work, we assume the garbling scheme GC is both projective and output-projective.

Definition 1 (Correctness [16]). *We say a garbling scheme* $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is correct if for all functions* f *and input* x*:*

$$\Pr[(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f) : \mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = f(x)] = 1 \ .$$

Definition 2 (Simulatable Privacy [16]). *We say a garbling scheme* $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is simulatable private if for all functions* f *and input* x*, there exists a PPT simulator* Sim *such that for all PPT adversary* $\mathcal{A}$ *the following holds:*

$$\Pr\left[\begin{array}{l}(F_0, e_0, d_0) \leftarrow \mathsf{Gb}(1^\lambda, f); X_0 \leftarrow \mathsf{En}(e, x);\\ (F_1, X_1, d_1) \leftarrow \mathsf{Sim}(1^\lambda, f(x), \Phi(f));\\ b \leftarrow \{0,1\}; b^* \leftarrow \mathcal{A}(F_b, X_b, d_b) : b = b^*\end{array}\right] = \mathsf{negl}(\lambda) \ .$$

where Φ *is the side-information function.*

Yao's GC Optimizations and Our Choice. Throughout the past decades, several optimization techniques have been proposed to improve the efficiency of Yao's garbled circuit (GC). In this section, we examine a few Yao's GC optimizations and analyze their suitability for our work to achieve the best performance, the concrete performance analysis is taken from the work of Zahur *et al.* [15].

In the classical garbling scheme, the GC generator needs to invoke the hash function H 4 times for each gate to create a garbled table consists of 4 ciphertexts. The GC evaluator also needs to invoke H up to 4 times for each gate to decrypt all these ciphertexts and obtains an output wire label.

Beaver *et al.* [12] introduced a technique called *point-and-permute*. By appending a select bit to each wire label, one can easily determine the places of the corresponding ciphertexts. Therefore, for a garbled table, the GC evaluator can decide which ciphertext to decrypt according to the select bit and only invoke H once. Nevertheless, each garbled table still contains 4 ciphertexts, and it takes 4 H invocations to generate. We adopt this technique in our design, as it greatly reduces the GC evaluator's computational cost, and it is compatible with other optimizations.

Naor *et al.* [17] introduced a *garbled row-reduction* technique known as GRR3 to reduce the garbled table size. The main idea is to fix 1 of the 4 ciphertexts, e.g., the top one, in each garbled table to be 0, and thus can be eliminated. In our construction, the memory of the enclave is limited, and this technique can reduce memory usage of GC generation.

Kolesnikov *et al.* [18] introduced the *free-XOR* technique. This technique allows us to garble and evaluate XOR gates for free. To do this, the offset between each wire's 0-label and 1-label in the entire circuit is fixed to Δ. Therefore, one can generate or evaluate an XOR gate via a simple XOR operation. This technique can greatly improve the performance of our scheme.

We note that, in a conventional 2PC setting, the other optimization techniques, such as *GRR2* [19] and *half-gates* [15], may be helpful to further improve scheme performance. However, GRR2 is not compatible with free-XOR. Although half-gates is compatible with the aforementioned three optimizations, it is not ideal for our construction. The reason is that the main benefit of half-gates is to reduce the non-XOR gate garbled table size to 2, but it needs 2 H invocations to evaluate. Whereas, in our design, the GC size is not the bottleneck of our overall performance, because the GC table is transmitted between the SGX enclave and the host locally. While, without half-gates, each non-XOR gate garbled table only needs 1 H invocation to evaluate.

Intel SGX. Intel Software Guard Extensions (SGX) is a widely used technology that enhances security of data and code. It allows developers to create guarded private region called enclave in processor reserved memory (PRM) and execute programs in the enclave. The enclave is an isolated execution environment, high-level softwares, including operating system and BIOS, can't break down the integrity and confidentiality guarantees of its computation. In execution, a party can remotely attest the genuinity of an enclave, provide private information to the enclave and verify the outcome is computed by a pre-agreed program with an advanced feature of Intel SGX called remote attestation. More specifically, Intel's (anonymous) attestation is based on

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

It interacts with parties $\mathcal{P} := \{P_1, P_2\}$ and the adversary $\mathcal{S}$. Let $\mathcal{P}_c$ be the set of corrupted parties.
Initially, set $\mathcal{P}_c = \emptyset$.

Compute:

- Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_i)$ from party $P_i \in \mathcal{P}$:
  - If $P_i \in \mathcal{P}_c$, send a notification $(\textsc{ComputeNotify}, \mathsf{sid}, x_i, P_i)$ to $\mathcal{S}$; Otherwise, send a notification $(\textsc{ComputeNotify}, \mathsf{sid}, |x_i|, P_i)$ to $\mathcal{S}$;
  - If it has received x_1 from P_1 and x_2 from P_2:
    - * Compute $y \leftarrow f(x_1, x_2)$;
  - Send $(\textsc{Output}, \mathsf{sid}, P_2)$ to adversary $\mathcal{S}$:
    - * Upon receiving $(\textsc{Deliver}, \mathsf{sid}, P_2)$ from $\mathcal{S}$, it sends $(\textsc{Compute}, \mathsf{sid}, y)$ to P_2;

Corruption handling:

- Upon receiving $(\textsc{Corrupt}, \mathsf{sid}, P_i)$ from the adversary $\mathcal{S}$, if $P_i \in \mathcal{P}$:
  - Set $\mathcal{P}_c := \mathcal{P}_c \cup \{P_i\}$;
  - Send $(\textsc{Input}, \mathsf{sid}, x_i, P_i)$ to $\mathcal{S}$ if x_i is already defined;

Figure 1: Functionality $\mathcal{F}^{f}_{\mathsf{2pc}}$

Semi-trusted Hardware Functionality $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}]$

It interacts with parties $\mathcal{P} := \{P_1, P_2\}$ and the adversary $\mathcal{A}$. It is parameterized with a PPT ITM M, a state Ψ, and a Boolean flag $\mathsf{corrupted}$.
Initially, set $\Psi := \emptyset$ and $\mathsf{corrupted} := \mathsf{false}$;

- Upon receiving $(\textsc{Corrupt}, \mathsf{sid}, \mathsf{M}^*)$ from $\mathcal{A}$:
  - Set $\mathsf{corrupted} := \mathsf{true}$;
  - If $\mathsf{M}^* \neq \emptyset$, replace $\mathsf{M} := \mathsf{M}^*$;
- Upon receiving $(\textsc{Run}, \mathsf{sid}, x_i)$ from party $P_i \in \mathcal{P}$:
  - If $\mathsf{corrupted} = \mathsf{true}$:
    - * Send leakage message $(\textsc{RunNotify}, \mathsf{sid}, x_i, P_i)$ to $\mathcal{A}$;
  - If $\mathsf{corrupted} = \mathsf{false}$:
    - * Send notification message $(\textsc{RunNotify}, \mathsf{sid}, P_i)$ to $\mathcal{A}$;
  - When $(\textsc{Run}, \mathsf{sid}, x_1)$ and $(\textsc{Run}, \mathsf{sid}, x_2)$ are both received:
    - * Run $(y_1, y_2; \Psi) \leftarrow \mathsf{M}(x_1, x_2; \Psi)$;
    - * For $i \in \{1, 2\}$, send $(\mathsf{Run}, \mathsf{sid}, y_i)$ to P_i;

Figure 2: The semi-trusted hardware functionality $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}]$

an anonymous group signature scheme called Intel Enhanced Privacy ID (EPID) [7]. The enclave to be attested first invokes the EREPORT instruction to create a locally verifiable report of its attributes and measurement, and sends this report to a special enclave named Quoting Enclave (QE). The QE verifies the report and produces a remotely verifiable quote by signing the quote with the group signature. The enclave then forwards the quote to the challenge party, and the party can contact with the remote Intel Attestation Service (IAS) server for verification. The IAS will first verify the group signature and then create a attestation verification report as a response.

3. Security Model

Universal Composibility. Our security model is based on the Universal Composibility (UC) framework [8], which lays down a solid foundation for designing and analyzing protocols secure against attacks in an arbitrary *network* execution environment (therefore it is also known as *network aware security model*). Roughly speaking, in the UC framework, protocols are carried out over multiple interconnected machines; to capture attacks, a network adversary $\mathcal{A}$ is introduced, which is allowed to corrupt some machines (i.e., have the full control of all physical parts of some machines); in addition, $\mathcal{A}$ is allowed to partially control the communication tapes of all uncorrupted machines, that is, it sees all the messages sent from and to the uncorrupted machines and controls the sequence in which they are delivered. Then, a protocol ρ is a UC-secure implementation of a functionality $\mathcal{F}$, if it satisfies that for every network adversary $\mathcal{A}$ attacking an execution of ρ, there is another adversary $\mathcal{S}$—known as the simulator—attacking the ideal process that uses $\mathcal{F}$ (by corrupting the same set of machines), such that, the executions of ρ with $\mathcal{A}$ and that of $\mathcal{F}$ with $\mathcal{S}$ makes no difference to any network execution environment.

The ideal world execution. In the ideal world, P_1 and P_2 only communicate with an ideal functionality $\mathcal{F}^{f}_{\mathsf{2pc}}$ during the execution. As depicted in Fig. 1, party $P_i \in \mathcal{P}$ sends $(\textsc{Compute}, \mathsf{sid}, x_i)$ to the functionality $\mathcal{F}^{f}_{\mathsf{2pc}}$, and $\mathcal{F}^{f}_{\mathsf{2pc}}$ sends a notification $(\textsc{ComputeNotify}, \mathsf{sid}, x_i, P_i)$ to the adversary $\mathcal{S}$ if P_i is corrupted; Otherwise, $\mathcal{F}^{f}_{\mathsf{2pc}}$ leaks the input size $(\textsc{ComputeNotify}, \mathsf{sid}, |x_i|, P_i)$ to $\mathcal{S}$. When both parties' inputs are received, $\mathcal{F}^{f}_{\mathsf{2pc}}$ computes $y \leftarrow f(x_1, x_2)$. It then sends $(\textsc{Compute}, \mathsf{sid}, y)$ to P_2 if the adversary $\mathcal{S}$ allows. For corruption handling, if the adversary $\mathcal{S}$ corrupts party $P_i \in \mathcal{P}$, $\mathcal{F}^{f}_{\mathsf{2pc}}$ adds P_i to the set of corrupted parties, $\mathcal{P}_c$, and leaks P_i's input x_i to $\mathcal{S}$ if it is already defined.

The real world execution. The real/hybrid world protocol Π uses a semi-trusted hardware components, which are modeled as the ideal functionality $\mathcal{F}_{\mathrm{HW}}$. Later, we will discuss how $\mathcal{F}_{\mathrm{HW}}$ is instantiated by Intel SGX in practice. For notation simplicity, we define $\mathcal{F}_{\mathrm{HW}}$ as a template, and specify the required functionalities in the description of a PPT Turing machine M. We use $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ in our semi-honest/malicious setting protocol $\Pi^{\mathsf{OT\text{-}GC}}_{\mathsf{2pc}}$.

3.1. Semi-trusted Hardware Model

We introduce a new notion, called *semi-trusted hardware model*. Unlike the conventional trusted hardware model, the semi-trusted hardware functionality $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}]$ shown in Fig. 2 can be corrupted by the adversary $\mathcal{A}$. The functionality $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}]$ is parameterized with a PPT ITM M, a state Ψ and a Boolean flag $\mathsf{corrupted}$ to indicate whether the hardware is corrupted. P_1 and P_2 can invoke $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}]$ to compute $(y_1, y_2; \Psi) \leftarrow \mathsf{M}(x_1, x_2; \Psi)$ by sending the input x_1 and x_2 respectively to $\mathcal{F}_{\mathrm{HW}}$.

However, the adversary $\mathcal{A}$ is allowed to corrupt $\mathcal{F}_{\mathrm{HW}}$ via the $(\textsc{Corrupt}, \mathsf{sid}, \mathsf{M}^*)$ command. When $\mathcal{A}$ is a semi-honest adversary, it sets $\mathsf{M}^* = \emptyset$. In execution, if $\mathcal{F}_{\mathrm{HW}}$ is corrupted, it will leak each party's input to $\mathcal{A}$. When $\mathcal{A}$ is a malicious adversary, M^* can be arbitrarily defined by $\mathcal{A}$ (not necessarily PPT), and $\mathcal{F}_{\mathrm{HW}}$ computes $(y_1, y_2; \Psi) \leftarrow \mathsf{M}^*(x_1, x_2; \Psi)$ instead. After the computation, $\mathcal{F}_{\mathrm{HW}}$ sends the output y_1 to P_1 and y_2 to P_2.

Description of $\mathsf{M}^{\mathsf{OT\text{-}GC}}$. We now define the Turing machine $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ for $\mathcal{F}_{\mathrm{HW}}$ that will be used for our 2PC protocol in the

```
Description of $\mathsf{M}^{\mathsf{OT\text{-}GC}}$
$\mathsf{M}^{\mathsf{OT\text{-}GC}}(x_1, x_2; \Psi)$ :
  • Parse $x_1 = \langle \mathsf{CMD}_1, x_1' \rangle$ and $x_2 = \langle \mathsf{CMD}_2, x_2' \rangle$;
  • If $\mathsf{CMD}_1 = \mathsf{CMD}_2 = \mathsf{ROT}$:
      – Parse $x_1' = \langle k_1^0, \ell_1 \rangle$ and $x_2' = \langle k_2^0, \ell_2 \rangle$;
      – Assert $\ell_1 = \ell_2$;
      – Generate $(b_1, \ldots, b_{\ell_1}) \leftarrow \mathsf{PRG}(k_2^0)$;
      – For $i \in [\ell_1]$: generate $R_i^{b_i} \leftarrow \mathsf{PRF}_{k_1^0}(i, b_i)$;
      – Return $y_1 := \emptyset$, $y_2 := \{R_i^{b_i}\}_{i \in [\ell_1]}$, and $\Psi := k_1^0$;
  • If $\mathsf{CMD}_1 = \mathsf{CMD}_2 = \mathsf{GC}$:
      – Parse $x_1' = \langle k_1^1, f_1 \rangle$ and $x_2' = f_2$;
      – Assert $f_1 = f_2$;
      – Generate $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f_1; k_1^1)$;
      – [M] Parse $e = \{(X_i^0, X_i^1)\}_{i \in [f_1.n]}$; /* Malicious */
      – [M] For $i \in [f_1.n_2]$: /* Malicious */
          ∗ [M] Generate $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i, 0)$, $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i, 1)$;
          ∗ [M] Set $\sigma_i^0 := H(R_i^0 \oplus X_{i+f_1.n_1}^0, R_i^1 \oplus X_{i+f_1.n_1}^1)$ and
            $\sigma_i^1 := H(R_i^1 \oplus X_{i+f_1.n_1}^0, R_i^0 \oplus X_{i+f_1.n_1}^1)$;
      – [S] Return $y_1 := \emptyset$ and $y_2 := (F, d)$; /* Semi-honest */
      – [M] Return $y_1 := \emptyset$ and $y_2 := (F, d, \{\sigma_i^0, \sigma_i^1\}_{i \in [f_1.n_2]})$; /* Mal
        */
```

Figure 3: Description of $\mathsf{M}^{\mathsf{OT\text{-}GC}}$

semi-honest/malicious adversarial setting. As depicted in Fig. 3, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ generates garbled circuit and random OT's. We use $[S]$ label to indicate instructions only included in the machine used in the semi-honest setting protocol, and $[M]$ label to indicate instructions only included in the machine used in the malicious setting protocol. Unlabeled instructions are perfomed in both settings.

When P_1 sends $\langle \mathsf{ROT}, \langle k_1^0, \ell_1 \rangle \rangle$ and P_2 sends $\langle \mathsf{ROT}, \langle k_2^0, \ell_2 \rangle \rangle$, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ parses their inputs to obtain their ROT seeds and the number of ROT instances, it then asserts P_1 and P_2 send the same ROT number $\ell_1 = \ell_2$.

To generate the ROT instances, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ first generates the ROT choice bits $(b_1, \ldots, b_{\ell_1}) \leftarrow \mathsf{PRG}(k_2^0)$, and for $i \in [\ell_1]$, it computes $R_i^{b_i} \leftarrow \mathsf{PRF}_{k_1^0}(i, b_i)$. After that, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ returns $\{R_i^{b_i}\}_{i \in [\ell_1]}$ to P_2 and stores $\Psi := k_1^0$.

When P_1 sends $\langle \mathsf{GC}, \langle k_1^1, f_1 \rangle \rangle$ and P_2 sends $\langle \mathsf{GC}, f_2 \rangle$, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ parses their input to obtain the GC seed k_1^1 and the circuit to be computed, and it asserts P_1 and P_2 send the same circuit $f_1 = f_2$. $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ then generates the garbled circuit by $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f; k_1^1)$. In the semi-honest setting, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ can simply returns (F, d) to P_2.

In the malicious setting, in addition to generate the GC copy, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ needs to produce some verification messages. More specifically, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ first parses the encoding information $e = \{(X_i^0, X_i^1)\}_{i \in [f_1.n]}$, and it fetches the seed k_1^0 from the state Ψ. Next, for $i \in [f_1.n_2]$, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ generates $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i, 0)$ and $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i, 1)$. It then sets $\sigma_i^0 := H(R_i^0 \oplus X_{i+f_1.n_1}^0, R_i^1 \oplus X_{i+f_1.n_1}^1)$ and $\sigma_i^1 := H(R_i^1 \oplus X_{i+f_1.n_1}^0, R_i^0 \oplus X_{i+f_1.n_1}^1)$. These hash values $\{\sigma_i^0, \sigma_i^1\}_{i \in [f_1.n_2]}$ can help P_2 verify that P_1 honestly transfer the input wire labels in the OT process. In the end, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ returns $(F, d, \{\sigma_i^0, \sigma_i^1\}_{i \in [f_1.n_2]})$ to P_2.

Instantiation of $\mathsf{M}^{\mathsf{OT\text{-}GC}}$. In practice, $\mathsf{M}^{\mathsf{OT\text{-}GC}}$ can be instantiated by just running an SGX enclave on the P_2 side. P_1 will remotely interact with P_2's SGX enclave via a secure channel established by remote attestation. We use 128-bit AES-NI to implement the PRF and PRG algorithms.

As introduced in Sec. 2, we adopt three GC optimizations, respectively are point-and-permute, GRR3 and free-XOR. For the point-and-permute, we set the least significant bits of the wire labels as the select bits, and arrange the garbled table according to these bits. For the GRR3 optimization, we set the 0-label of the output wire as the first row of the garbled table, and XOR each row with this 0-label, then the first row becomes an all 0 string and thus can be eliminated. And the free-XOR optimization is implemented as described.

4. Silent 2PC Protocols

In this section, we present our silent 2PC protocols in the semi-honest setting and malicious setting. A straightforward approach is to split the protocol into an offline phase and an online phase. In the offline phase, the parties don't know their inputs, and they interact with $\mathcal{F}_{\mathrm{HW}}$ to obtain the GC tables and sufficiently many ROT instances; in the online phase, the parties use their inputs to exchange the GC labels and evaluate the function f. Unfortunately, only by the simulatable private GC definition (cf. Def. 2), this approach won't result in a UC simulatable secure protocol. The subtle issue is as follows. When P_2 is corrupted, the simulator $\mathcal{S}$ needs to extract its input so it can send P_2's input to the ideal functionality $\mathcal{F}_{\mathsf{2pc}}^f$ and learn the function output from it. $\mathcal{S}$ has to simulate (fake) GC tables according to the function output and the function f, and it then sends the simulated (fake) garbled circuit and the simulated (fake) decoding information to P_2 on behalf of $\mathcal{F}_{\mathrm{HW}}$, so the corrupted P_2 will see a right output after evaluating the garbled circuit. Obviously, if the GC tables are generated in an offline phase, it is impossible for $\mathcal{S}$ to invoke the GC simulator, because the inputs of the function haven't been determined, let along the function output.

Description of $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$. We depict our semi-honest/malicious setting protocol in Fig. 4, where f is the function that P_1 and P_2 want to jointly compute, as described in Sec. 2, n_1, n_2 and n are the input size of P_1, the input size of P_2 and the overall input size, respectively. In Fig. 4, we use $[S]$ label to indicate instructions only included in the semi-honest setting protocol, and $[M]$ label to indicate instructions only included in the malicious setting protocol, we also comment on these special line's end to emphasize this difference. Other instructions not labeled should be included in both the semi-honest setting protocol and the malicious setting protocol.

The semi-honest setting protocol. In the semi-honest setting protocol, P_2 first picks a random $k_2^0 \leftarrow \{0,1\}^\lambda$ as its ROT seed, and sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2 \rangle \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to generate n_2 ROT instances. Meanwhile, P_1 also picks a random $k_1^0 \leftarrow \{0,1\}^\lambda$, and for $i \in [n_2]$, it generates the ROT instances $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i, 0)$, $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i, 1)$ by itself. P_1 then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2 \rangle \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.

After that, P_2 receives the ROT instances $\{R_i^{b_i}\}_{i \in [n_2]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, it then invokes PRG with its ROT seed to generate $(b_1, \ldots, b_{n_2})$. Next, P_2 computes and sends the choice bits $\{c_i := b_i \oplus x_{2,i}\}_{i \in [n_2]}$ to P_1, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to generate the garbled circuit.

```
Protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$

Protocol description:
• Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_2 := (x_{2,1}, \ldots, x_{2,n_2}))$ from the environment $\mathcal{Z}$, $P_2$:
  – Pick random $k_2^0 \leftarrow \{0,1\}^\lambda$;
  – Send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2 \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$;
• Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_1 := (x_{1,1}, \ldots, x_{1,n_1}))$ from the environment $\mathcal{Z}$, $P_1$:
  – Pick random $k_1^0 \leftarrow \{0,1\}^\lambda$;
  – For $i \in [n_2]$:
      ∗ Generate $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i, 0)$, $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i, 1)$;
      ∗ [M] Set $\sigma_{1,i}^0 := H(R_i^0), \sigma_{1,i}^1 := H(R_i^1)$; /* Malicious */
  – Send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2 \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$;
  – [M] Send $\{\sigma_{1,i}^0, \sigma_{1,i}^1\}_{i \in [n_2]}$ to $P_2$; /* Malicious */
• Upon receiving $(\mathsf{Run}, \mathsf{sid}, \{R_i^{b_i}\}_{i \in [n_2]})$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ (and [M] $\{\sigma_{1,i}^0, \sigma_{1,i}^1\}_{i \in [n_2]}$ from $P_1$), $P_2$:
  – Generate $(b_1, \ldots, b_{n_2}) \leftarrow \mathsf{PRG}(k_2^0)$;
  – For $i \in [n_2]$:
      ∗ [M] Set $\hat{\sigma}_{1,i} := H(R_i^{b_i})$, and assert $\hat{\sigma}_{1,i} = \sigma_{1,i}^{b_i}$; /* Malicious */
      ∗ Set $c_i := b_i \oplus x_{2,i}$;
  – Send $\{c_i\}_{i \in [n_2]}$ to $P_1$;
  – Send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$;
• Upon receiving $\{c_i\}_{i \in [n_2]}$ from $P_2$, $P_1$:
  – Pick random $k_1^1 \leftarrow \{0,1\}^\lambda$;
  – Generate $(\hat{F}, \hat{e}, \hat{d}) \leftarrow \mathsf{Gb}(1^\lambda, f; k_1^1)$;
  – [M] Set $\sigma_2 := H(\hat{F}, \hat{d})$; /* Malicious */
  – Parse $\hat{e} = \{(X_i^0, X_i^1)\}_{i \in [n]}$;
  – For $i \in [n_2]$:
      ∗ Compute $W_i^0 := R_i^{c_i} \oplus X_{n_1+i}^0$ and $W_i^1 := R_i^{c_i \oplus 1} \oplus X_{n_1+i}^1$;
      ∗ [M] Compute $\hat{\sigma}_{3,i}^{c_i \oplus 1} = H(R_i^{c_i \oplus 1} \oplus X_{n_1+i}^0, R_i^{c_i} \oplus X_{n_1+i}^1)$; /* Malicious */
  – Send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, \langle k_1^1, f \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$;
  – Send $\{Z_i := X_i^{x_{1,i}}\}_{i \in [n_1]}$, $\{W_i^0, W_i^1\}_{i \in [n_2]}$ (and [M] $\sigma_2$, $\{\hat{\sigma}_{3,i}^{c_i \oplus 1}\}_{i \in [n_2]}$) to $P_2$;
• Upon receiving $\{Z_i\}_{i \in [n_1]}$, $\{W_i^0, W_i^1\}_{i \in [n_2]}$ (and [M] $\sigma_2$, $\{\hat{\sigma}_{3,i}^{c_i \oplus 1}\}_{i \in [n_2]}$) from $P_1$ and $(\mathsf{Run}, \mathsf{sid}, (F, d))$ (and [M] $\{\sigma_{3,i}^0, \sigma_{3,i}^1\}_{i \in [n_2]}$) from
  $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $P_2$:
  – [M] Set $\hat{\sigma}_2 := H(F, d)$ and assert $\hat{\sigma}_2 = \sigma_2$; /* Malicious */
  – For $i \in [n_2]$:
      ∗ [M] Set $\hat{\sigma}_{3,i}^{c_i} := H(W_i^0, W_i^1)$, and assert $\hat{\sigma}_{3,i}^0 = \sigma_{3,i}^0$ and $\hat{\sigma}_{3,i}^1 = \sigma_{3,i}^1$; /* Malicious */
      ∗ Compute $Z_{n_1+i} := W_i^{x_{2,i}} \oplus R_i^{b_i}$;
  – Evaluate $Y \leftarrow \mathsf{GC.Ev}(F, (Z_1, \ldots, Z_n))$;
  – Decode $y \leftarrow \mathsf{GC.De}(d, Y)$;
  – Return $(\textsc{Compute}, \mathsf{sid}, y)$ to the environment $\mathcal{Z}$;
```

Figure 4: The semi-honest/malicious setting protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ in the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$-hybrid model

Subsequently, P_1 picks a random $k_1^1 \leftarrow \{0,1\}^\lambda$ as the GC seed and generates a GC copy by $(\hat{F}, \hat{e}, \hat{d}) \leftarrow \mathsf{GC.Gb}(1^\lambda, f; k_1^1)$,[3] it then parses the encoding information to obtain the input wire labels $\{(X_i^0, X_i^1)\}_{i \in [n]}$. For $i \in [n_2]$, P_1 computes the OT responses $W_i^0 := R_i^{c_i} \oplus X_{n_1+i}^0$ and $W_i^1 := R_i^{c_i \oplus 1} \oplus X_{n_1+i}^1$. Next, P_1 sends the GC seed k_1^1 to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, and it sends the wire labels corresponding to its input value $\{Z_i := X_i^{x_{1,i}}\}_{i \in [n_1]}$ and the OT responses $\{W_i^0, W_i^1\}_{i \in [n_2]}$ to P_2.

After that, P_2 receives the garbled tables and the decoding information (F, d) from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, and it receives P_1's input wire labels and the OT responses from P_1. Then, for $i \in [n_2]$, P_2 computes $Z_{n_1+i} := W_i^{x_{2,i}} \oplus R_i^{b_i}$ to obtain its input wire label. At the end, P_2 evaluates $Y \leftarrow \mathsf{GC.Ev}(F, (Z_1, \ldots, Z_{n_1+n_2}))$, and decodes $y \leftarrow \mathsf{GC.De}(d, Y)$.

The malicious setting protocol. As depicted in Fig. 4, P_2 first picks a random $k_2^0 \leftarrow \{0,1\}^\lambda$ as its ROT seed, and sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2 \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to generate n_2 ROT instances. Meanwhile, P_1 also picks a random $k_1^0 \leftarrow \{0,1\}^\lambda$. For $i \in [n_2]$, P_1 generates the ROT instances $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i, 0)$, $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i, 1)$ and computes their hash values by $\sigma_{1,i}^0 := H(R_i^0), \sigma_{1,i}^1 := H(R_i^1)$. P_1 then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2 \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{\sigma_{1,i}^0, \sigma_{1,i}^1\}_{i \in [n_2]}$ to P_2.

After that, P_2 receives the ROT instances $\{R_i^{b_i}\}_{i \in [n_2]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{\sigma_{1,i}^0, \sigma_{1,i}^1\}_{i \in [n_2]}$ from P_1, it then invoke PRG with its ROT seed to generate $(b_1, \ldots, b_{n_2})$. Next, for $i \in [n_2]$, P_2 computes $\hat{\sigma}_{1,i} := H(R_i^{b_i})$ and compares it with $\sigma_{1,i}^{b_i}$ to check $R_i^{b_i}$'s correctness. If no check fails, P_2 computes and sends the choice bits $\{c_i := b_i \oplus x_{2,i}\}_{i \in [n_2]}$ to P_1, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to generate the garbled circuit.

Subsequently, P_1 picks a random $k_1^1 \leftarrow \{0,1\}^\lambda$ as the GC seed and generates a GC copy by $(\hat{F}, \hat{e}, \hat{d}) \leftarrow \mathsf{GC.Gb}(1^\lambda, f; k_1^1)$. P_1 computes the hash of the garbled tables and decoding

[3] In practice, P_1 only needs to generate the encoding information of the GC, e, i.e., the input wire labels; therefore, $\mathsf{GC.Gb}$ is only partially executed for efficiency.

information by $\sigma_2 := H(\hat{F}, \hat{d})$, and it parses the encoding information to obtain the input wire labels $\{(X_i^0, X_i^1)\}_{i\in[n]}$. For $i \in [n_2]$, P_1 generates the OT responses $W_i^0 := R_i^{c_i} \oplus X_{n_1+i}^0$ and $W_i^1 := R_i^{c_i\oplus 1} \oplus X_{n_1+i}^1$, and a hash value $\hat{\sigma}_{3,i}^{c_i\oplus 1} = H(R_i^{c_i\oplus 1} \oplus X_{n_1+i}^0, R_i^{c_i} \oplus X_{n_1+i}^1)$. Next, P_1 sends the GC seed k_1^1 to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, and it sends the wire labels corresponding to its input value $\{Z_i := X_i^{x_{1,i}}\}_{i\in[n_1]}$, the hash values σ_2 and $\{\hat{\sigma}_{3,i}^{c_i\oplus 1}\}_{i\in[n_2]}$, and the OT responses $\{W_i^0, W_i^1\}_{i\in[n_2]}$ to P_2.

After that, P_2 receives the garbled tables F, the decoding information d and the hash values of the OT responses $\{\sigma_{3,i}^0, \sigma_{3,i}^1\}_{i\in[n_2]}$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. P_2 sets $\hat{\sigma}_2 := H(F, d)$ and asserts $\hat{\sigma}_2 = \sigma_2$. Then, for $i \in [n_2]$, P_2 sets $\hat{\sigma}_{1,i}^{c_i} := H(W_i^0, W_i^1)$, it then verifies that P_1 is honest in the OT process by checking $\hat{\sigma}_{3,i}^0 = \sigma_{3,i}^0$ and $\hat{\sigma}_{3,i}^1 = \sigma_{3,i}^1$; after these checks are finished, P_2 computes $Z_{n_1+i} := W_i^{x_{2,i}} \oplus R_i^{b_i}$ to obtain its input wire label. At the end, P_2 evaluates $Y \leftarrow \mathsf{GC.Ev}(F, (Z_1, \ldots, Z_{n_1+n_2}))$, and decodes $y \leftarrow \mathsf{GC.De}(d, Y)$.

5. Security

In this section, we examine the security of our schemes. We assume the hardware manufacturer will not collude with the MPC players; otherwise, when P_1 (or P_2) is colluding with $\mathcal{F}_{\text{HW}}$, no input privacy can be guaranteed.[4]

We first examine why our schemes are secure at the high level, and then formally state the security of our semi-honest/malicious setting protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$in Thm. 1/Thm. 2, respectively, where we restrict the adversary $\mathcal{A}$ to only corrupt either the semi-trusted hardware functionality $\mathcal{F}_{\text{HW}}$ or the player(s) P_1 (and/or P_2).

In the semi-honest setting, the view of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ is the MPC function f, the length of P_2's input n_2 and some random seeds, f and n_2 are already known to the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$, and the seeds are true random numbers; therefore, no additional information would be leaked to the adversary $\mathcal{A}$. Since $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ could only be passively corrupted, the correctness of garbled circuit and ROT instances are preserved. The input privacy of protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ is guaranteed by the simulatable privacy property of the underlying garbling scheme GC.

In the malicious setting, $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, P_1, and P_2 may be maliciously corrupted. The main design principle is as follows. In P_1's point of view, either $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ or P_2 could be corrupted. Similarly, in P_2's point of view, either $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ or P_1 could be corrupted. Note that our protocol does not provide accountability, i.e., when the protocol abort, we are not required to identify which party is guilty. Thus, P_2 can use messages generated by $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and messages sent by P_1 to carry out a mutual verification, and it aborts if any inconsistency is detected.

More specifically, the ROT instances $\{R_i^{b_i}\}_{i\in[n_2]}$ produced by $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ are checked using hash values $\sigma_{1,i}^0 := H(R_i^0)$ and $\sigma_{1,i}^1 := H(R_i^1)$ sent by P_1. This check has to be done before P_1 receives the OT choice bits $\{c_i\}_{i\in[n_2]}$, otherwise P_1 can carry out a selective failure attack to learn the ROT bits $\{b_i\}_{i\in[n_2]}$ and extract P_2's private input. The garbled tables and decoding information (F, d) are also validated via a hash digest provided by P_1. To verify that P_1 is honest in the OT process, P_2 uses hash values of all possible OT responses from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. Since a malicious $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ may provide incorrect values, when P_1 sends a OT response $R_i^{c_i} \oplus X_{n_1+i}^0, R_i^{c_i\oplus 1} \oplus X_{n_1+i}^1$, it also sends the hash of the other message $R_i^{c_i\oplus 1} \oplus X_{n_1+i}^0, R_i^{c_i} \oplus X_{n_1+i}^1$. Then $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ can't distinguish between aborts due to incorrect OT responses and aborts due to incorrect hash values.

Theorem 1. *If* $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \{0,1\}^\lambda$ *is a secure PRF function,* $\mathsf{PRG} : \{0,1\}^\lambda \mapsto \{0,1\}^{\ell(\lambda)}$ *is a secure PRG function, and* $\mathsf{GC} := (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is a secure simulatable private garbling scheme, protocol* $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ *(semi-honest setting) described in Fig. 4 UC-realizes* $\mathcal{F}_{\mathsf{2pc}}^f$ *as described in Fig. 1 in the* $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$*-hybrid model against any PPT semi-honest adversaries who can corrupt either* $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ *or the player(s)* P_1 *(and/or* P_2*) with static corruption.*

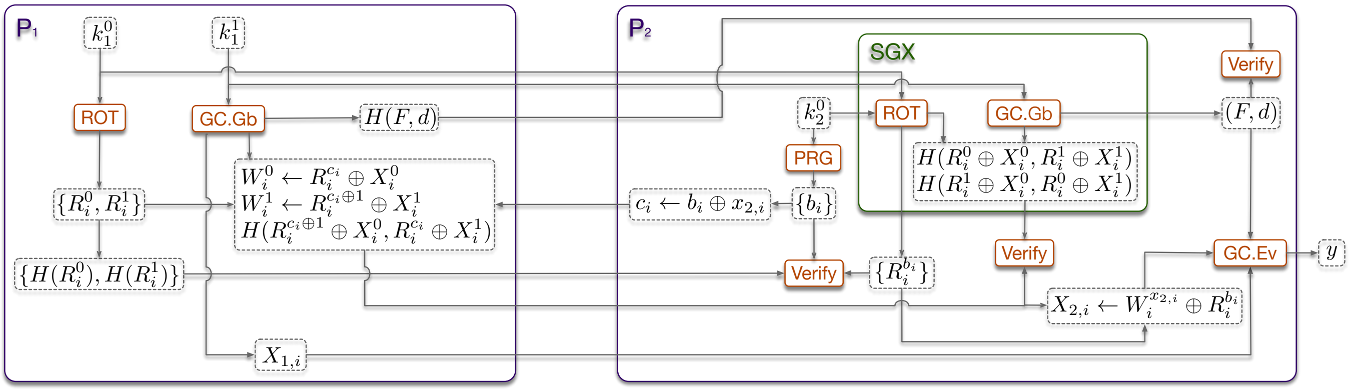
Proof. To prove Thm. 1, we construct a simulator $\mathcal{S}$ such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between (i) the real execution $\textsc{Exec}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}$ where the parties $\mathcal{P} := \{P_1, P_2\}$ run protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ in the $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$-hybrid model and the corrupted parties are controlled by a dummy adversary $\mathcal{A}$ who simply forwards messages from/to $\mathcal{Z}$, and (ii) the ideal execution $\textsc{Exec}_{\mathcal{F}_{\mathsf{2pc}}^f, \mathcal{S}, \mathcal{Z}}$ where the parties P_1 and P_2 interact with functionality $\mathcal{F}_{\mathsf{2pc}}^f$ in the ideal world, and corrupted parties are controlled by the simulator $\mathcal{S}$. We consider following cases.

Case 1: $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ is corrupted; P_1 and P_2 are honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ as well as honest parties P_1 and P_2. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon receiving $(\textsc{ComputeNotify}, \mathsf{sid}, |x_i|, P_i)$ for an honest party P_i from the external $\mathcal{F}_{\mathsf{2pc}}^f$, the simulator $\mathcal{S}$ picks random $k_i^0 \leftarrow \{0,1\}^\lambda$ and then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_i^0, n_2\rangle\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ on behave of P_i.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, Q_i)$ from $P_i \in \mathcal{P}$ via the interface of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ acts as $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to send $(\textsc{RunNotify}, \mathsf{sid}, Q_i, P_i)$ to $\mathcal{A}$. $\mathcal{S}$ then simulates the $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ functionality as defined.
- When the simulated party P_2 receive $\{R_i^{b_i}\}_{i\in[n_2]}$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ acts as P_2 to compute $(b_1, \ldots, b_{n_2}) \leftarrow \mathsf{PRG}(k_2^0)$, and then it sends $\{c_i := b_i\}_{i\in[n_2]}$ to the simulated party P_1 and send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.
- $\mathcal{S}$ then simulates the rest communication between P_1 and P_2 according to the protocol description as if both P_1 and P_2 receive $(\textsc{Compute}, \mathsf{sid}, 0)$ from $\mathcal{Z}$.

[4] We note that our schemes can be easily modified to ensure security in the scenario where P_1 and $\mathcal{F}_{\text{HW}}$ are colluding; while we don't have an efficient solution to handle the collusion between P_2 and $\mathcal{F}_{\text{HW}}$ yet, as this setting could be reduced to standard two-party computation without assistance of $\mathcal{F}_{\text{HW}}$ in practice.



P1
k_1^0
k_1^1
ROT
GC.Gb
$H(F,d)$
$\{R_i^0, R_i^1\}$
$W_i^0 \leftarrow R_i^{c_i} \oplus X_i^0$
$W_i^1 \leftarrow R_i^{c_i \oplus 1} \oplus X_i^1$
$H(R_i^{c_i \oplus 1} \oplus X_i^0, R_i^{c_i} \oplus X_i^1)$
$\{H(R_i^0), H(R_i^1)\}$
$X_{1,i}$
P2
SGX
k_2^0
ROT
GC.Gb
PRG
$H(R_i^0 \oplus X_i^0, R_i^1 \oplus X_i^1)$
$H(R_i^1 \oplus X_i^0, R_i^0 \oplus X_i^1)$
$c_i \leftarrow b_i \oplus x_{2,i}$
$\{b_i\}$
Verify
$\{R_i^{b_i}\}$
Verify
Verify
(F,d)
GC.Ev
y
$X_{2,i} \leftarrow W_i^{x_{2,i}} \oplus R_i^{b_i}$

Figure 5: Malicious silent-Yao protocol

Indistinguishability. Assume the communication between P_1 and P_2 is via the secure channel functionality $\mathcal{F}_{\text{SC}}$, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\text{EXEC}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]}_{\Pi^{\text{OT-GC}}_{\text{2pc}},\mathcal{A},\mathcal{Z}}$ and $\text{EXEC}_{\mathcal{F}^f_{\text{2pc}},\mathcal{S},\mathcal{Z}}$ are identical. Therefore, it is perfectly indistinguishable.

Case 2: P_1 is corrupted; P_2 and $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ are honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ as well as honest P_2. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon receiving $(\text{COMPUTENOTIFY}, \mathsf{sid}, |x_2|, P_2)$ from the external $\mathcal{F}^f_{\text{2pc}}$, the simulator $\mathcal{S}$ picks random $k_2^0 \leftarrow \{0,1\}^\lambda$ and then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2\rangle\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ on behave of P_2.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2\rangle\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2\rangle\rangle)$ from P_2, for $i \in [n_2]$, $\mathcal{S}$ acts as $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ to compute $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i,0)$, $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i,1)$, and it picks a random $b_i \leftarrow \{0,1\}$. $\mathcal{S}$ then sends $\{R_i^{b_i}\}_{i\in[n_2]}$ to the simulated party P_2 on behave of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$.
- When the simulated party P_2 receive $\{R_i^{b_i}\}_{i\in[n_2]}$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$, $\mathcal{S}$ acts as P_2 to send $\{c_i := b_i\}_{i\in[n_2]}$ to the simulated party P_1. $\mathcal{S}$ then acts as P_2 to send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$.
- When P_2 receives $\{Z_i\}_{i\in[n_1]}$ and $\{W_i^0, W_i^1\}_{i\in[n_2]}$ from P_1, $\mathcal{S}$ uses the internal GC label information (F, e, d) of the simulated $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ to extract P_1's input x_1^*. It then sends $(\text{COMPUTE}, \mathsf{sid}, x_1^*)$ to the external $\mathcal{F}^f_{\text{2pc}}$ on behave of P_1.
- Upon receiving $(\text{OUTPUT}, \mathsf{sid}, P_2)$ from the external $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$, if $\mathcal{A}$ allows P_2 to finish the protocol execution and obtains y, $\mathcal{S}$ sends $(\text{DELIVER}, \mathsf{sid}, P_2)$ to the external $\mathcal{F}^f_{\text{2pc}}$.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \ldots, \mathcal{H}_2$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\text{EXEC}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]}_{\Pi^{\text{OT-GC}}_{\text{2pc}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, the ROT choice bits $b_1, \ldots, b_{n_2}$ are true random bits instead of computing from $(b_1, \ldots, b_{n_2}) \leftarrow \mathsf{PRG}(k_2^0)$.

Claim 1. *If* $\mathsf{PRG} : \{0,1\}^\lambda \mapsto \{0,1\}^{n_2}$ *is a secure PRG function with adversarial distinguishing advantage* $\mathsf{Adv}_{\mathsf{PRG}}(\mathcal{A}, \lambda)$, *then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with distinguishing advantage* $\mathsf{Adv}_{\mathsf{PRG}}(\mathcal{A}, \lambda)$.

Proof. It is a straightforward reduction; namely, by hybrid argument, we can show that if there exists an adversary $\mathcal{A}$ who can distinguish $\mathcal{H}_1$ from $\mathcal{H}_0$, then we can construct an adversary $\mathcal{B}$ who can break the PRG. □

Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, P_2 sends $\{c_i' := b_i\}_{i\in[n_2]}$ to P_1, instead of $\{c_i := b_i \oplus x_{2,i}\}_{i\in[n_2]}$.

Claim 2. *$\mathcal{H}_2$ and $\mathcal{H}_1$ are perfectly indistinguishable.*

Proof. Since b_i are the ROT select bits randomly picked by $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$, the distribution of $\{c_i'\}_{i\in[n_2]}$ and $\{c_i\}_{i\in[n_2]}$ are identical. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_1$ are perfectly indistinguishable. □

The adversary's view of $\mathcal{H}_2$ is identical to the simulated view $\text{EXEC}_{\mathcal{F}^f_{\text{2pc}},\mathcal{S},\mathcal{Z}}$. Therefore, the overall distinguishing advantage is $\mathsf{Adv}_{\mathsf{PRG}}(\mathcal{A}, \lambda)$.

Case 3: P_2 is corrupted; P_1 and $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ are honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ as well as honest P_1. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon receiving $(\text{COMPUTENOTIFY}, \mathsf{sid}, |x_1|, P_1)$ from the external $\mathcal{F}^f_{\text{2pc}}$, the simulator $\mathcal{S}$ picks random $k_1^0 \leftarrow \{0,1\}^\lambda$ and $R_i^0 \leftarrow \{0,1\}^\lambda$, $R_i^1 \leftarrow \{0,1\}^\lambda$. It then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2\rangle\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ on behave of P_1.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2\rangle\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2\rangle\rangle)$ from P_2, $\mathcal{S}$ acts as $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$ to compute $(b_1, \ldots, b_{n_2}) \leftarrow \mathsf{PRG}(k_2^0)$. $\mathcal{S}$ then fetches $\{R_i^0, R_i^1\}_{i\in[n_2]}$ from the simulated P_1 and sends $\{R_i^{b_i}\}_{i\in[n_2]}$ to P_2.
- When P_1 receives $\{c_i\}_{i\in[n_2]}$ from P_2, $\mathcal{S}$ fetches $\{b_i\}_{i\in[n_2]}$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\text{OT-GC}}]$'s internal state. $\mathcal{S}$ then extracts P_2's input $x_{2,i}^* := c_i \oplus b_i$. After that, it sends $(\text{COMPUTE}, \mathsf{sid}, x_2^*)$ to the external $\mathcal{F}^f_{\text{2pc}}$ on behave of P_2.

- Upon receiving $(\textsc{Compute}, \mathsf{sid}, y)$ from the external $\mathcal{F}_{\mathsf{2pc}}^{f}$ for P_2, the simulator $\mathcal{S}$ uses the GC simulator to generate $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f))$.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle \mathrm{GC}, \langle k_1^1, f\rangle\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathrm{GC}, f\rangle)$ from P_2 to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ sends (F', d') as the GC tables and decode information to P_2 on behave of $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. $\mathcal{S}$ then uses X' as the wire labels to generate $\{Z_i\}_{i\in[n_1]}$ and $\{W_i^0, W_i^1\}_{i\in[n_2]}$ as follows:
  1. For $i \in [n_1]$, set $Z_i := X'_i$;
  2. For $i \in [n_2]$: set $W_i^{x_{2,i}} := X'_{n_1+i} \oplus R_i^{b_i}$ and $W_i^{x_{2,i}\oplus 1} := R_i^{b_i \oplus 1}$;

  $\mathcal{S}$ then acts as P_1 to send those messages to P_2.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \ldots, \mathcal{H}_2$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\textsc{Exec}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that $\mathcal{H}_1$ uses true random numbers $R_i^0, R_i^1 \leftarrow \{0,1\}^\lambda$ instead of $R_i^b \leftarrow \mathsf{PRF}_{k_1^0}(i, b)$, $b \in \{0,1\}$.

Claim 3. *If* $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \{0,1\}^\lambda$ *is a secure PRF function with adversarial distinguishing advantage* $\mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \lambda)$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with distinguishing advantage* $2n_2 \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \lambda)$.

Proof. It is a straightforward reduction; namely, by hybrid argument, we can show that if there exists an adversary $\mathcal{A}$ who can distinguish $\mathcal{H}_1$ from $\mathcal{H}_0$, then we can construct an adversary $\mathcal{B}$ who can break the PRF. □

Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that $\mathcal{H}_2$ generates $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f))$, and then it uses X' as the wire labels to generate $\{Z_i\}_{i\in[n_1]}$ and $\{W_i^{x_{2,i}}\}_{i\in[n_2]}$. $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ also sends (F', d') as the GC tables and decoding information to P_2.

Claim 4. *If* GC *is simulatable private with adversarial distinguishing advantage* $\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{A}, \lambda)$*, then* $\mathcal{H}_2$ *and* $\mathcal{H}_1$ *are indistinguishable with distinguishing advantage* $\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{A}, \lambda)$.

Proof. First of all, by the requirement of simulatable privacy in Def. 2, $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f))$ should be indistinguishable from the real one. Moreover, since P_2 does not know $R_i^{b_i\oplus 1}$, if there is an adversary $\mathcal{A}$ who can distinguish the distribution of $\{W_i^0, W_i^1\}_{i\in[n_2]}$ from the real one with probability ε, then we can construct an adversary $\mathcal{B}$ who has the same distinguishing advantage $\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{B}, \lambda) = \varepsilon$. □

The adversary's view of $\mathcal{H}_2$ is identical to the simulated view $\textsc{Exec}_{\mathcal{F}_{\mathsf{2pc}}^{f}, \mathcal{S}, \mathcal{Z}}$. Therefore, if GC is simulatable private, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\textsc{Exec}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}$ and $\textsc{Exec}_{\mathcal{F}_{\mathsf{2pc}}^{f}, \mathcal{S}, \mathcal{Z}}$ are indistinguishable with distinguishing advantage

$$2n_2 \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \lambda) + \mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda) \; .$$

Case 4: P_1 and P_2 are corrupted; $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ is honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. The simulator $\mathcal{S}$ simulates the functionality $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.

Indistinguishability. This is a trivial case. Since both P_1 and P_2 are controlled by the adversary $\mathcal{A}$, no message is simulated by $\mathcal{S}$.

This concludes the proof. □

Theorem 2. *If* $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \{0,1\}^\lambda$ *is a secure PRF function,* $\mathsf{PRG} : \{0,1\}^\lambda \mapsto \{0,1\}^{\ell(\lambda)}$ *is a secure PRG function,* $H : \{0,1\}^* \mapsto \{0,1\}^\lambda$ *is a collision resistant hash function, and* $\mathsf{GC} := (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is a secure simulatable private garbling scheme, protocol* $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ *(malicious setting) described in Fig. 4 UC-realizes* $\mathcal{F}_{\mathsf{2pc}}^{f}$ *as described in Fig. 1 in the* $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$*-hybrid model against any PPT malicious adversaries who can corrupt either* $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ *or the player(s)* P_1 *(and/or* P_2*) with static corruption.*

The proof is provided in App. B.

6. Further reducing communication

In this section, we show how to further reduce the online communication complexity in the semi-honest setting. The main observation is as follows. P_2 can use some masks to hide its inputs, and directly send these masked inputs to P_1 to fetch the corresponding input wire labels; while in our original semi-honest setting protocol as presented in Fig. 4, P_2 has to use OT to obtain its wire labels. Note that using OT to transfer 1 label needs $2\lambda + 1$ bits communication, and directly fetch one only needs $\lambda + 1$ bits.

The idea of using masks in garbled circuit can trace back to the point-and-permute technique [12] – Beaver *et al.* append a select bit to each wire label, and arrange each garbled table according to the select bits of the input wire labels. Before presenting our protocol, we first specify the $\mathcal{F}_{\mathrm{HW}}$ functionality as $\mathsf{M}^{\mathsf{mask\text{-}GC}}$. It performs two tasks: (i) generate pseudo random input masks, send P_1 and P_2 the masks of their inputs; (ii) generate masked garbled circuit, send the input wire labels to P_1 and the garbled tables and decoding information to P_2. During the protocol $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ execution, both parties first query the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$ for their input masks; then, P_2 hides its input with masks, and sends the masked input to P_1; thereafter, both parties use $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$ to generate the garbled circuit; after receiving the input wire labels, P_1 sends the wire labels corresponding to both parties' masked inputs to P_2; at the end, P_2 evaluates the garbled circuit and decode the output. Due to space limitation, we put the detailed description of $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ and the protocol $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ in Fig. 6 and Fig. 7 in Appendix A.

Security. The security analysis of the improved protocol $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ is analogous to semi-honest setting $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ under the same assumption. The proof is much similar to the one of Thm. 1, so we only provide a high level description here. As in the semi-honest setting protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$, we isolate the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$ from any private inputs, so even when $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$ is corrupted, the input privacy can be guaranteed;

TABLE 4: Performance comparison of the computation process of the semi-honest setting protocols $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ and $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$. Result obtained from SGX-enabled Dell OptiPlex 7080 (Intel Core 8700 CPU @ 3.20 GHz, 32 GB RAM, OS: Ubuntu 18.04 LTS). It shows the running time (in ms) for evaluating AES-128, SHA-256, and SHA-512 circuits once, respectively.

Circuit	Network setting	Garbler running time (in ms)		Evaluator running time (in ms)	
		$\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$	$\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$	$\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$	$\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$
AES-128	LAN (1Gbps, 1ms)	4.653	2.255	7.607	5.240
	WAN (100Mbps, 25ms)	101.905	50.567	155.027	103.970
SHA-256	LAN (1Gbps, 1ms)	4.729	2.286	9.392	6.828
	WAN (100Mbps, 25ms)	102.131	50.577	160.933	110.272
SHA-512	LAN (1Gbps, 1ms)	4.934	2.291	12.221	10.448
	WAN (100Mbps, 25ms)	103.404	50.650	176.338	124.731

and the correctness of $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$'s outputs are preserved since we only consider passive corruption. For P_2's privacy, since P_2's input is masked by pseudo random bits, P_1 cannot learn P_2's real input. When P_2 is corrupted, we can extract its inputs from the masked input received by P_1.

Efficiency. Table. 4 shows the performance comparison our two semi-honest setting protocols $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$ and $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$. We take the AES-128, SHA-256, and SHA-512 circuit evaluation as benchmarks. In the LAN setting, $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ is 2.06-2.15X faster w.r.t. the Garbler's running time and 1.16-1.45X faster w.r.t. the Evaluator's running time than $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$. In the WAN setting, $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ is 2.01-2.04X faster w.r.t. the Garbler's running time and 1.41-1.49X faster w.r.t. the Evaluator's running time than $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$.

7. Implementation and Benchmarks

Our protocols are implemented in C++ using Intel SGX SDK on Linux. We use AES-NI for the PRF and PRG algorithms. To efficiently generate ROT's in the SGX enclave, we carefully analyze the performance bottleneck and notice that if we just generate one instance of ROT per batch, then the Receiver needs to enter the enclave for many times, and the enter/exit may cause performance loss. Because this, we group multiple ROT's into a batch, and choose the optimal batch size according to the test result.

We already explained our choice of GC optimizations in Sec. 2, and here we provide more details. Denote the seed of the garbled circuit as k, to generate the wire labels, we first compute the $\mathsf{PRF}_k(0)$ and force its least significant bit to be 1, and the result is the Δ in the free-XOR optimization. Subsequently, we invoke the PRF for n times in the form $\mathsf{PRF}_k(i)$ to generate the 0-label of the i-th input, then we compute $\Delta \oplus \mathsf{PRF}_k(i)$ to get the 1-label of the i-th input. After obtaining all the wire labels, we computes $k^{'} := \mathsf{PRF}_k(n+1)$ as the seed for generation of garbled tables.

With regard to the generation of the garbled circuits, we assume the order of the gates in the circuit description is layer-designed such that, for a gate to be garbled, it's input wire won't be the output wire of a gate that hasn't been garbled. Hence, we can garble the gates as this order. For a XOR gate, since free-XOR is used, its garbled tables is eliminated, and we simply XOR the two input wires' 0-label to obtain the 0-label of the output wire. For each non-XOR gate, we generate 4 ciphertexts for different input wire label combinations. After that, we can determine each ciphertext's place in the garbled table according to the select bit, i.e., the least significant bit of the wire labels, as desribed in the point-and-permute optimization. Denote the first input wire label's select bit as s_a and the second input wire label's select bit as s_b, the ciphertext derived from these two wire labels will be placed in the $(s_a + 2 * s_b + 1)$-th row of the garbled table. Furthermore, since we adopt the GRR3 optimization, we compute the output wire's label in this way: if the first row is generated from two 1-label, we set the value of output wire's 1-label as the first row's value; otherwise, we set the value of the output wire's 0-label as the first row's value. Once we know the value of the output wire's 0-label/1-label, the other label can be computed by simply XOR Δ with it. Next, we XOR each row with appropriate output wire label, then the first row becomes an all 0 string and thus can be eliminated.

The evaluation process use the same seed as the garbling process, since the SGX enclave runs on the Evaluator's machine, this seed can be locally transferred. The evaluation order is also as in the circuit description. For each XOR gate, the Evaluator only has two wire labels and we XOR these two label to obtain the result. For each non-XOR gate, we invoke H with the input wire labels, and decrypt one row in the garbled table, whose place can be computed according to the input wire labels' select bits.

We use two types of hash functions in our construction. To efficiently generate the GC table, we instantiate the hash function H using block ciphers.This idea can trace back to the JustGarble system [20], where Bellare *et al.* modeled a public fixed-key AES as a random permutation π, and construct the garbled circuit bashed on π. Motivated by JustGarble, Zahur *et al.* [15] use a random permutation π to construct the hash function H, and π is instantiated by a public fixed-key AES in a particular way. However, in recent work of Guo *et al.* [21], an attack that can completely break the security of garbling scheme instantiated by public fixed-key AES is found, and they claim that constructing the hash function in a different way can prevent this attack. Their implementation is also based on AES, while the hash function is evaluated on both the input and a tweak, and it involves re-keying AES rather than fixed-key AES. In our implementation, we construct the hash function H based on the work of Guo *et al.* [21]. On the other hand, in our malicious setting, we mostly need the compression property of the underlying hash function for efficient verification; therefore, SHA256 is adopted.

We perform the experiments on an SGX-enabled Dell OptiPlex 7080 equipped with an Intel Core 8700 CPU @ 3.20 GHz with 32.0 GB RAM, running Ubuntu 18.04 LTS.

We evaluate all protocols in two simulated network settings: (i) a LAN setting with 1Gbps bandwidth and 0.1ms delay and (ii) a WAN setting with 100Mbps bandwidth and 25ms delay.

To test the performance of our ROT generation protocol, we compared our protocol with the implementation of the IKNP OT extension protocol [13] and Ferret OT extension protocol [14] in EMP-ROT [11]. Table. 1 shows the performance comparison for generating 10^4 to 10^8 instances of ROT, where the result is the average of 10 tests.

To test the performance of the 2PC protocols, our benchmarks use three Bristol Fashion format circuits [22], which consists of only AND gates, XOR gates and inverters; namely, AES-128 circuit, SHA-256 circuit and SHA-512 circuit. The AES-128 circuit contains 36919 wires and 36663 gates, including 6400 AND gates; in this circuit, P_1's input size, P_2's input size and the output size are all 128 bits. The SHA-256 circuit contains 135841 wires and 135073 gates, including 22573 AND gates; in this circuit, P_1's input size is 512 bits, P_2's input size and the output size are both 256 bits. The SHA-512 circuit contains 351153 wires and 349617 gates, including 57947 AND gates; in this circuit, P_1's input size is 1024 bits, P_2's input size and the output size are both 512 bits. For the semi-honest setting protocol, we compared our protocol with EMP-SH2PC [11] (EMP-SH2PC provides an efficient semi-honest 2PC implementation based on Yao's GC protocol with half-gates [15] optimization); for the malicious setting protocol, we compared our protocol with EMP-AG2PC [11] (EMP-AG2PC implements an efficient maliciously secure two-party computation protocol, authenticated garbling [10]).

Table. 2 shows the performance comparison for evaluating the aforementioned benchmark circuits for 1000 times using the semi-honest setting protocols, and the results are the average of 10 tests. Table. 3 shows the performance comparison for evaluating the aforementioned benchmark circuits once using the malicious setting protocols, and the results are the average of 100 tests. All the one-time expenses are omitted, e.g., creating enclave in our protocol and initialize $\mathcal{F}_{pre}$ in EMP-AG2PC.

Alternative Deployment Setting. Alternative TEE techniques can be adopted to realize the $\mathcal{F}_{\mathrm{HW}}$ functionality in case the side-channel leakages of SGX is a concern. For instance, we could deploy a TEE server aside of P_2 with fast local connection.

8. Related Work

As a closely related work, Mohassel *et al.* [9] proposed a scheme that enables efficient secure computation on mobile phones. Their protocol is constructed in a *Server-Aided* setting, where a semi-honest (covert) server who does not collude with protocol players is used to accelerate computation. Their protocol is based on the GMW protocol [23] and the Beaver triples [24], and is secure against malicious adversary. Although looks similar, there are many differences between this work and theirs. Our intention is to reduce the communication in the protocol execution, so we use trusted hardware that can be deployed locally to assist computation, while they focus on light-weight schemes that can be implemented on mobile phones. Moreover, unlike [9], the semi-trusted hardware can be maliciously corrupted in our model.

Wang *et al.* [10] proposed an efficient framework for maliciously secure two-party computation, which is known as *Authenticated Garbling*. Later, Katz *et al.* [25] make authenticated garbling compatible with the half-gates [15] optimization, resulting in a protocol for malicious secure 2PC in which the communication complexity of the online phase is essentially equivalent to that of state-of-the-art semi-honest secure 2PC.

Gupta *et al.* [26] proposed a protocol using Intel SGX for SFE problem which is secure in the semi-honest model, they also show how to improve their protocol's security. However, no implementation is provided in their work due to the lack of equipments. They also notice the problem that the developers need to trust hardware and hard supplier when using Intel SGX, but don't propose a feasible solution.

Bahmani *et al.* [27] proposed an intuitive approach in which the program in an isolated execution environment (IEE) plays the role of a trusted third party and the major part of computational load is left to the untrusted machine. In this way, they reach a minimum communication complexity that only depends on number of inputs and outputs. Obviously, the trust to IEE and hardware manufacturer is crucial. They introduced a novel notion of labelled attested computation (LAC) and give a LAC-based solution with rigorous security guarantees. They implement Intel SGX-based version of their protocol and compare it with the ABY framework, and their solution is hundreds of times faster than ABY.

Felsen *et al.* [28] proposed an Intel SGX-based secure function evaluation (SFE) approach in which private inputs are sent to enclave. In their protocol, only the inputs and the outputs need to be transferred, the communication complexity of their protocol is optimal up to an additive constant. They evaluate the Boolean circuit representation of the function in enclave to provide security with regards to software side-channel attacks. In addition, they reduce the problem of private function evaluation (PFE) to the problem of SFE by using universal circuits and are the first to address PFE problem via TEEs. They give a prototype implementation of their protocol and compare its performance with state-of-the-art implementations of Yao's GC and the GMW protocols.

Choi *et al.* [29] consider the possibility of SGX being compromised and want to protect the most sensitive data in any case. They propose a hybrid SFE-SGX protocol which consists of calculation in SGX enclave and standard cryptographic techniques. The function to be evaluated is partitioned into several round functions, the odd rounds are executed in enclave and the even rounds are done using a scheme based on garbled circuit. They claim that, if the partition scheme is proper, which means no private inputs is leaked by intermediate values, their hybrid approach ensures security against semi-hones adversary. They also notice that there are numerous side-channel attacks against SGX that can extract information from enclave, so they deploy corresponding mitigation techniques to protect privacy. They present how to use this hybrid protocol to solve privacy-preserving retrieval and privacy-preserving navigation. In Choi's work, the enclave gets part of the private input, while we ensure the enclave is isolated with any private data and only produce information independent of inputs, which guarantees privacy even if the enclave is compromised.

Chakraborty *et al.* [30] use the trusted hardware to enable intellectual property protection. They propose an obfuscation framework called Hardware Protected Neural Network (HPNN) in which a deep neural network is trained as a function of a secret key and then, the obfuscated deep learning model is hosted on a public model sharing platform.

9. Conclusion

In this work, we investigate the problem where the trusted hardware manufacturer are not fully trusted, and the hardware components may leak sensitive information to the remote servers through backdoors, side-channels, steganography, and kleptography, etc. In our model, the adversary is allowed to passively and/or maliciously corrupt the hardware component. We present two efficient semi-honest setting 2PC protocols and one efficient malicious setting 2PC protocol. The communication of our protocols only depends on the input size regardless the circuit size. We implemented our protocols and compared it with the EMP-toolkit. When the semi-trusted hardware is instantiated by Intel SGX, our ROT protocol is several magnitude times faster than the EMP-IKNP-ROT and EMP-Ferret-ROT, and our semi-honest setting (and malicious setting) 2PC protocol is also significantly faster than the EMP-SH2PC (and EMP-AG2PC). We will generalize our technique to more multi-party computation scenarios, such as PSI and PPML, in the future.

References

[1] G. Dan and S. Jim, “More than 20gb of intel source code and proprietary data dumped online,” [EB/OL], https://arstechnica.com/information-technology/2020/08/intel-is-investigating-the-leak-of-20gb-of-its-source-code-and-private-data/ Accessed August 30, 2020.

[2] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, “Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[3] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: Sgx cache attacks are practical,” in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[4] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, “Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1041–1056.

[5] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[6] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[7] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, “Intel® software guard extensions: Epid provisioning and attestation services,” *White Paper*, vol. 1, no. 1-10, p. 119, 2016.

[8] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[9] P. Mohassel, O. Orobets, and B. Riva, “Efficient server-aided 2pc for mobile phones,” *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 82–99, 2016.

[10] X. Wang, S. Ranellucci, and J. Katz, “Authenticated garbling and efficient maliciously secure two-party computation,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 21–37.

[11] X. Wang, A. J. Malozemoff, and J. Katz, “EMP-toolkit: Efficient Multi-Party computation toolkit,” 2016, https://github.com/emp-toolkit/ Accessed January 5th, 2021.

[12] D. Beaver, S. Micali, and P. Rogaway, “The round complexity of secure protocols,” in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 503–513.

[13] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, “Extending oblivious transfers efficiently,” in *Annual International Cryptology Conference*. Springer, 2003, pp. 145–161.

[14] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, “Ferret: Fast extension for correlated ot with small communication,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1607–1626.

[15] S. Zahur, M. Rosulek, and D. Evans, “Two halves make a whole,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

[16] M. Bellare, V. T. Hoang, and P. Rogaway, “Foundations of garbled circuits,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 784–796.

[17] M. Naor, B. Pinkas, and R. Sumner, “Privacy preserving auctions and mechanism design,” in *Proceedings of the 1st ACM conference on Electronic commerce*, 1999, pp. 129–139.

[18] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 486–498.

[19] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *International conference on the theory and application of cryptology and information security*. Springer, 2009, pp. 250–267.

[20] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 478–492.

[21] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu, "Better concrete security for half-gates garbling (in the multi-instance setting)," in *Annual International Cryptology Conference*. Springer, 2020, pp. 793–822.

[22] D. Archer, V. A. Abril, S. Lu, P. Maene, N. Mertens, D. Sijacic, and N. Smart, "'Bristol Fashion' MPC Circuits," 2020, https://homes.esat.kuleuven.be/~nsmart/MPC/ Accessed January 5th, 2021.

[23] S. Micali, O. Goldreich, and A. Wigderson, "How to play any mental game," in *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, 1987, pp. 218–229.

[24] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Annual International Cryptology Conference*. Springer, 1991, pp. 420–432.

[25] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang, "Optimizing authenticated garbling for faster secure two-party computation," in *Annual International Cryptology Conference*. Springer, 2018, pp. 365–391.

[26] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor, "Using intel software guard extensions for efficient two-party secure function evaluation," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 302–318.

[27] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure multiparty computation from sgx," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 477–497.

[28] S. Felsen, Á. Kiss, T. Schneider, and C. Weinert, "Secure and private function evaluation with intel sgx," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 165–181.

[29] J. I. Choi, D. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. Butler, and P. Traynor, "A hybrid approach to secure function evaluation using sgx," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 100–113.

[30] A. Chakraborty, A. Mondai, and A. Srivastava, "Hardware-assisted intellectual property protection of deep learning models," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

Appendix

1. Descriptions of $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ and $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$

In this section, we provide a detailed description of $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ and $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$, the optimization techniques we adopted is the same as in the protocol $\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}$.

We first define the Turing machine $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ for $\mathcal{F}_{\mathrm{HW}}$. As depicted in Fig. 6, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ generates input masks and masked garbled circuit. When P_1 sends $\langle \textsc{mask}, \langle k_1^0, f_1 \rangle\rangle$ and P_2 sends $\langle \textsc{mask}, \langle k_2^0, f_2 \rangle\rangle$, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ asserts P_1 and P_2 send the same circuit $f_1 = f_2$, and it sets the seed by $k^0 := k_1^0 \oplus k_2^0$. $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ then generates the input masks $(\rho_1, \ldots, \rho_{f_1.n}) \leftarrow \mathsf{PRG}(k^0)$ and sends P_1 and P_2 their respective masks.

When P_1 sends $\langle \mathrm{GC}, \langle k_1^1, f_1 \rangle\rangle$ and P_2 sends $\langle \mathrm{GC}, \langle k_2^1, f_2 \rangle\rangle$, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ asserts P_1 and P_2 send the same circuit $f_1 = f_2$, and it sets the seed of the masked garbled circuit by $k^1 := k_1^1 \oplus k_2^1$. To generate the wire labels, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ sets $\Delta \leftarrow \mathsf{PRF}_{k^1}(0)$. Then for $i \in [f.n]$, it computes the 0-label $X_i^0 \leftarrow \mathsf{PRF}_{k^1}(i)$ and the 1-label is set by $X_i^1 := X_i^0 \oplus \Delta$. After that, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ generates the garbled tables in topological order. We use $G(i, j, k)$ to represent a gate: its input wires are the i-th wire and the j-wire, and its output wire is the k-th wire. For an XOR gate, we simply sets the mask and the 0-label of the k-th wire as $\rho_k = \rho_i \oplus \rho_j$ and $X_k^0 = X_i^0 \oplus X_j^0$; for an AND gate, we first computes $T := H(X_i^0, X_j^0)$, then set the output wire's mask $\rho_k := T_{[1]} \oplus (\rho_i \wedge \rho_j)$ and its $T_{[1]}$-label $X_k^{T_{[1]}} := T$, note that this label can either be 0-label or 1-label, and the other label has an offset Δ with it. After the mask bit of all three wires are known, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ computes a masked truth value, and generates the masked garbled table by $U^1 := H(X_i^0, X_j^1) \oplus X_k^{\hat{x}_k^{(0,1)}}$, $U^2 := H(X_i^1, X_j^0) \oplus X_k^{\hat{x}_k^{(1,0)}}$ and $U^3 := H(X_i^1, X_j^1) \oplus X_k^{\hat{x}_k^{(1,1)}}$. At the end, $\mathsf{M}^{\mathsf{mask\text{-}GC}}$ sends Δ and the input wires' 0-labels to P_1, and sends the masked garbled tables and the decoding information to P_2. The decoding information is the mask bits of the output wires.

We depict the protocol $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ in Fig. 7 , where f is the function that P_1 and P_2 want to jointly compute. In $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$, both parties first pick their random number $k_i^0 \leftarrow \{0,1\}^\lambda, i \in 1, 2$, and send $(\mathsf{Run}, \mathsf{sid}, \langle \textsc{mask}, \langle k_i^0, f \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ returns the masks ρ_i to P_1 and P_2 so they can use these masks to hide their inputs.

After that, P_2 computes $z_{i+n_1} := x_{2,i} \oplus \rho_{i+n_1}$, for $i \in [n_2]$ and picks another random number $k_2^1 \leftarrow \{0,1\}^\lambda$. It then sends $\{z_{i+n_1}\}_{i \in [n_2]}$ to P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathrm{GC}, \langle k_2^1, f \rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.

Subsequently, P_2 also picks random $k_1^1 \leftarrow \{0,1\}^\lambda$ and sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathrm{GC}, (k_1^1, f)\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, which will reply $(\mathsf{Run}, \mathsf{sid}, (\Delta, \{X_i^0\}_{i \in [n]}))$. It uses the masks to hide its input by computing $z_i := x_{1,i} \oplus \rho_i$, for $i \in [n_1]$. Next, it selects wire labels corresponding to both P_1 and P_2's masked input, if $z_i = 1$, it XOR Δ with X_i^0 to obtain X_i^1. It sends the input wire labels $\{Z_i := X_i^{\hat{x}_{1,i}}\}_{i \in [n_1]}$ and $\{Z_{i+n_1} := X_{i+n_1}^{\hat{x}_{2,i}}\}_{i \in [n_2]}$ and its masked input $\{z_i\}_{i \in [n_1]}$ to P_2.

After P_2 receiving P_1's masked inputs $\{z_i\}_{i \in [n_1]}$ and these input wire labels $\{Z_i\}_{i \in [n]}$ from P_1, and the garbled tables and

```
Description of $\mathsf{M}^{\mathsf{mask\text{-}GC}}$

$\mathsf{M}^{\mathsf{mask\text{-}GC}}(x_1, x_2; \Psi)$ :
  • Parse $x_1 = \langle \mathsf{CMD}_1, x'_1 \rangle$ and $x_2 = \langle \mathsf{CMD}_2, x'_2 \rangle$;
  • If $\mathsf{CMD}_1 = \mathsf{CMD}_2 = \textsc{mask}$:
    – Parse $x'_1 = \langle k^0_1, f_1 \rangle$ and $x'_2 = \langle k^0_2, f_2 \rangle$;
    – Assert $f_1 = f_2$;
    – Set $k^0 := k^0_1 \oplus k^0_2$;
    – Generate $(\rho_1, \ldots, \rho_{f_1.n}) \leftarrow \mathsf{PRG}(k^0)$;
    – Return $y_1 := \{\rho_i\}_{i \in [f_1.n_1]}$, $y_2 := \{\rho_i\}_{i \in [f_1.n_1+1, f_1.n]}$, and
      $\Psi := \{\rho_i\}_{i \in [f_1.n]}$;
  • If $\mathsf{CMD}_1 = \mathsf{CMD}_2 = \mathsf{GC}$:
    – Parse $x'_1 = \langle k^1_1, f_1 \rangle$ and $x'_2 = \langle k^1_2, f_2 \rangle$;
    – Assert $f_1 = f_2$;
    – Set $k^1 := k^1_1 \oplus k^1_2$;
    – Generate $\Delta \leftarrow \mathsf{PRF}_{k^1}(0)$;
    – For $i \in [f_1.n]$, generate $X^0_i \leftarrow \mathsf{PRF}_{k^1}(i)$ and $X^1_i := X^0_i \oplus \Delta$;
    – Generate the garbled circuit $F$ in topological order:
      For each gate $G(i, j, k)$:
        ∗ XOR gate: set $\rho_k = \rho_i \oplus \rho_j$ and $X^0_k = X^0_i \oplus X^0_j$;
        ∗ AND gate:
            · Set $T := H(X^0_i, X^0_j)$;
            · Set $\hat{x}^{(0,0)}_k = T_{[1]}$ and $X^{\hat{x}^{(0,0)}_k}_k := T$;
            · Set $\rho_k := T_{[1]} \oplus (\rho_i \wedge \rho_j)$;
            · Set $\hat{x}^{(0,1)}_k := (\rho_i \wedge \rho_j \oplus 1) \oplus \rho_k$;
            · Set $\hat{x}^{(1,0)}_k := (\rho_i \oplus 1 \wedge \rho_j) \oplus \rho_k$;
            · Set $\hat{x}^{(1,1)}_k := (\rho_i \oplus 1 \wedge \rho_j \oplus 1) \oplus \rho_k$;
            · Generate the garbled table as
              $U^1 := H(X^0_i, X^1_j) \oplus X^{\hat{x}^{(0,1)}_k}_k$,
              $U^2 := H(X^1_i, X^0_j) \oplus X^{\hat{x}^{(1,0)}_k}_k$, and
              $U^3 := H(X^1_i, X^1_j) \oplus X^{\hat{x}^{(1,1)}_k}_k$, where
              $X^{(0)}_k \oplus X^{(1)}_k = \Delta$;
    – Return $y_1 := (\Delta, \{X^0_i\}_{i \in [f_1.n]})$ and $y_2 := (F, d)$, where $F$
      consists of all the AND gates' garbled table and $d$ consists of the
      mask bit for the output wire;
```

Figure 6: Description of $\mathsf{M}^{\mathsf{mask\text{-}GC}}$

decoding information F, d from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$, it evaluates the garbled circuit in topological order according to the masked inputs and their corresponding labels, and it decodes the output using the mask bits.

2. Security Proof in the Malicious Setting

Theorem 2. *If* $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \{0,1\}^\lambda$ *is a secure PRF function,* $\mathsf{PRG} : \{0,1\}^\lambda \mapsto \{0,1\}^{\ell(\lambda)}$ *is a secure PRG function,* $H : \{0,1\}^* \mapsto \{0,1\}^\lambda$ *is a collision resistant hash function, and* $\mathsf{GC} := (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is a secure simulatable private garbling scheme, protocol* $\Pi^{\mathsf{OT\text{-}GC}}_{\mathsf{2pc}}$ *(malicious setting) described in Fig. 4 UC-realizes* $\mathcal{F}^f_{\mathsf{2pc}}$ *as described in Fig. 1 in the* $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$*-hybrid model against any PPT malicious adversaries who can corrupt either* $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ *or the player(s)* P_1 *(and/or* P_2*) with static corruption.*

Proof. To prove Thm. 2, we construct a simulator $\mathcal{S}$ such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between (i) the real execution $\mathrm{EXEC}^{\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}_{\Pi^{\mathsf{OT\text{-}GC}}_{\mathsf{2pc}}, \mathcal{A}, \mathcal{Z}}$ where the parties $\mathcal{P} := \{P_1, P_2\}$ run protocol $\Pi^{\mathsf{OT\text{-}GC}}_{\mathsf{2pc}}$ in the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$-hybrid model and the corrupted parties are controlled by a dummy adversary $\mathcal{A}$ who simply forwards messages from/to $\mathcal{Z}$, and (ii) the ideal execution $\mathrm{EXEC}_{\mathcal{F}^f_{\mathsf{2pc}}, \mathcal{S}, \mathcal{Z}}$ where the parties P_1 and P_2 interact with functionality $\mathcal{F}^f_{\mathsf{2pc}}$ in the ideal world, and corrupted parties are controlled by the simulator $\mathcal{S}$. We consider following cases.

Case 1: $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ is corrupted; P_1 and P_2 are honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ as well as honest parties P_1 and P_2. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon receiving $(\textsc{ComputeNotify}, \mathsf{sid}, |x_2|, P_2)$ from the external $\mathcal{F}^f_{\mathsf{2pc}}$, the simulator $\mathcal{S}$ picks random $k^0_2 \leftarrow \{0,1\}^\lambda$ and then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k^0_2, n_2 \rangle \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ on behave of P_2.
- Upon receiving $(\textsc{ComputeNotify}, \mathsf{sid}, |x_1|, P_1)$ from the external $\mathcal{F}^f_{\mathsf{2pc}}$, the simulator $\mathcal{S}$ picks random $k^0_1 \leftarrow \{0,1\}^\lambda$. For $i \in [n_2]$, the simulator $\mathcal{S}$ generates $R^0_i \leftarrow \mathsf{PRF}_{k^0_1}(i, 0)$, $R^1_i \leftarrow \mathsf{PRF}_{k^0_1}(i, 1)$, and it computes $\sigma^0_{1,i} := H(R^0_i)$, $\sigma^1_{1,i} := H(R^1_i)$. $\mathcal{S}$ then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k^0_1, n_2 \rangle \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{\sigma^0_{1,i}, \sigma^1_{1,i}\}_{i \in [n_2]}$ to P_2 on behave of P_1.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, Q_i)$ from the party $P_i \in \mathcal{P}$ via the interface of $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ acts as $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to send $(\textsc{RunNotify}, \mathsf{sid}, Q_i, P_i)$ to $\mathcal{A}$. $\mathcal{S}$ then simulates the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ functionality as defined.
- When the simulated party P_2 receives $\{R^{b_i}_i\}_{i \in [n_2]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{\sigma^0_{1,i}, \sigma^1_{1,i}\}_{i \in [n_2]}$ from P_1, $\mathcal{S}$ acts as P_2 to compute $(b_1, \ldots, b_{n_2}) \leftarrow \mathsf{PRG}(k^0_2)$. For $i \in [n_2]$, $\mathcal{S}$ computes $\hat{\sigma}_{1,i} := H(R^{b_i}_i)$, and asserts $\hat{\sigma}_{1,i} = \sigma^{b_i}_{1,i}$. After that, it sends $\{c_i := b_i\}_{i \in [n_2]}$ to the simulated party P_1 and send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.
- When the simulated party P_1 receives $\{c_i\}_{i \in [n_2]}$ from the simulated party P_2, $\mathcal{S}$ acts as P_1 to pick random $k^1_1 \leftarrow \{0,1\}^\lambda$ and generate $(\hat{F}, \hat{e}, \hat{d}) \leftarrow \mathsf{Gb}(1^\lambda, f; k^1_1)$. It then sets $\sigma_2 := H(\hat{F}, \hat{d})$ and parses $\hat{e} = \{(X^0_i, X^1_i)\}_{i \in [n]}$. Thereafter, $\mathcal{S}$ acts as P_1 according to the protocol description as if $x_1 = \mathbf{0}$. More specifically, for $i \in [n_2]$: $\mathcal{S}$ computes $W^0_i := R^{c_i}_i \oplus X^0_{n_1+i}$ and $W^1_i := R^{c_i \oplus 1}_i \oplus X^1_{n_1+i}$; it then computes $\hat{\sigma}^{c_i \oplus 1}_{3,i} = H(R^{c_i \oplus 1}_i \oplus X^0_{n_1+i}, R^{c_i}_i \oplus X^1_{n_1+i})$. After that, $\mathcal{S}$ sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, \langle k^1_1, f \rangle \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{Z_i := X^0_i\}_{i \in [n_1]}$, σ_2, and $\{W^0_i, W^1_i, \hat{\sigma}^{c_i \oplus 1}_{3,i}\}_{i \in [n_2]}$ to P_2.
- When the simulated party P_2 receives $\{Z_i\}_{i \in [n_1]}$, σ_2, and $\{W^0_i, W^1_i, \hat{\sigma}^{c_i \oplus 1}_{3,i}\}_{i \in [n_2]}$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, (F, d))$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ computes $\hat{\sigma}_2 := H(F, d)$, and asserts $\hat{\sigma}_2 = \sigma_2$. Thereafter, for $i \in [n_2]$, $\mathcal{S}$ computes $\hat{\sigma}^{c_i}_{3,i} := H(W^0_i, W^1_i)$, and asserts $\hat{\sigma}^0_{3,i} = \sigma^0_{3,i}$ and $\hat{\sigma}^1_{3,i} = \sigma^1_{3,i}$.
- Upon receiving $(\textsc{Output}, \mathsf{sid}, P_2)$ from the external $\mathcal{F}^f_{\mathsf{2pc}}$, the simulator $\mathcal{S}$ returns $(\textsc{Deliver}, \mathsf{sid}, P_2)$ if and only if all the checks are valid.

Indistinguishability. Assume the communication between P_1 and P_2 is via the secure channel functionality $\mathcal{F}_{\mathrm{SC}}$, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\mathrm{EXEC}^{\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}_{\Pi^{\mathsf{OT\text{-}GC}}_{\mathsf{2pc}}, \mathcal{A}, \mathcal{Z}}$ and $\mathrm{EXEC}_{\mathcal{F}^f_{\mathsf{2pc}}, \mathcal{S}, \mathcal{Z}}$ are identical except the scenario where the real-world output y is different from the ideal-world output y'. This happens when the malicious $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ provides inconsistent information,

Protocol $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$

Protocol description:

- Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_i := (x_{i,1}, \dots, x_{i,n_i}))$ from the environment $\mathcal{Z}$, P_i:
  - Pick random $k_i^0 \leftarrow \{0,1\}^\lambda$;
  - Send $(\mathsf{Run}, \mathsf{sid}, \langle \textsc{mask}, \langle k_i^0, f\rangle\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$;
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \{\rho_i\}_{i\in[n_1+1,n]})$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$, P_2:
  - For $i \in [n_2]$: compute $z_{i+n_1} := x_{2,i} \oplus \rho_{i+n_1}$;
  - Pick random $k_2^1 \leftarrow \{0,1\}^\lambda$;
  - Send $\{z_{i+n_1}\}_{i\in[n_2]}$ to P_1;
  - Send $(\mathsf{Run}, \mathsf{sid}, \langle \text{GC}, \langle k_2^1, f\rangle\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$;
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \{\rho_i\}_{i\in[n_1]})$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$ and $\{z_i\}_{i\in[n_1+1,n]}$ from P_2, P_1:
  - Pick random $k_1^1 \leftarrow \{0,1\}^\lambda$;
  - Send $(\mathsf{Run}, \mathsf{sid}, \langle \text{GC}, (k_1^1, f)\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, which will reply $(\mathsf{Run}, \mathsf{sid}, (\Delta, \{X_i^0\}_{i\in[n]}))$;
  - For $i \in [n_1]$: compute $z_i := x_{1,i} \oplus \rho_i$;
  - Send $\{z_i\}_{i\in[n_1]}$, $\{Z_i := X_i^{\hat{x}_{1,i}}\}_{i\in[n_1]}$ and $\{Z_{i+n_1} := X_{i+n_1}^{\hat{x}_{2,i}}\}_{i\in[n_2]}$ to P_2;
- Upon receiving $\{z_i\}_{i\in[n_1]}$ and $\{Z_i\}_{i\in[n]}$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, (F, d))$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$, P_2:
  - Evaluate the garbled circuit in topological order. For each gate $G(i,j,k)$:
    - XOR gate, compute $z_k := z_i \oplus z_j$ and $Z_k := Z_i \oplus Z_j$;
    - AND gate, compute $T := H(Z_i, Z_j)$. If $2 * z_i + z_j = 0$, set $z_k := T_{[1]}$ and $Z_k := T$; else, decode the $(2 * z_i + z_j - 1)$-th row of the garbled table by computing $z_k := (T \oplus U^{2*z_i+z_j})_{[1]}$ and $Z_k := T \oplus U^{2*z_i+z_j}$;
  - For $i \in [m]$, decode the output by computing $y_i := z_{i+N-m} \oplus \lambda_{i+N-m}$;
  - Return $(\textsc{Compute}, \mathsf{sid}, \{y_i\}_{i\in[m]})$ to the environment $\mathcal{Z}$;

Figure 7: The semi-honest setting protocol $\Pi_{\mathsf{2pc}}^{\mathsf{mask\text{-}GC}}$ in the $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{mask\text{-}GC}}]$-hybrid model

yet she manages to pass all the hash validations. It means that the adversary provides at least one different hash preimage that would hashes to the same value as the original preimage. Therefore, the simulator and the adversary can jointly outputs two messages $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$. Assume H is a collision resistant cryptographic hash function, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\textsc{exec}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}},\mathcal{A},\mathcal{Z}}$ and $\textsc{exec}_{\mathcal{F}_{\mathsf{2pc}}^f,\mathcal{S},\mathcal{Z}}$ are indistinguishable.

Case 2: P_1 is corrupted; P_2 and $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ are honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ as well as honest P_2. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon receiving $(\textsc{ComputeNotify}, \mathsf{sid}, |x_2|, P_2)$ from the external $\mathcal{F}_{\mathsf{2pc}}^f$, the simulator $\mathcal{S}$ picks random $k_2^0 \leftarrow \{0,1\}^\lambda$ and then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2\rangle\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ on behave of P_2.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2\rangle\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2\rangle\rangle)$ from P_2, $\mathcal{S}$ acts as $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to compute $R_i^0 \leftarrow \mathsf{PRF}_{k_1^0}(i, 0)$, $R_i^1 \leftarrow \mathsf{PRF}_{k_1^0}(i, 1)$, and picks a random $b_i \leftarrow \{0,1\}$, for $i \in [n_2]$. $\mathcal{S}$ then sends $\{R_i^{b_i}\}_{i\in[n_2]}$ to the simulated party P_2 on behave of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.
- When the simulated party P_2 receives $\{R_i^{b_i}\}_{i\in[n_2]}$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{\sigma_{1,i}^0, \sigma_{1,i}^1\}_{i\in[n_2]}$ from P_1, for $i \in [n_2]$, $\mathcal{S}$ uses the internal b_i to computes $\hat{\sigma}_{1,i} := H(R_i^{b_i})$, and asserts $\hat{\sigma}_{1,i} = \sigma_{1,i}^{b_i}$. After that, it sends $\{c_i := b_i\}_{i\in[n_2]}$ to P_1 and send $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.
- When the simulated party P_2 receives $\{Z_i\}_{i\in[n_1]}$, σ_2, and $\{W_i^0, W_i^1, \hat{\sigma}_{3,i}^{c_i\oplus 1}\}_{i\in[n_2]}$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, (F, d))$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ computes $\hat{\sigma}_2 := H(F, d)$, and asserts $\hat{\sigma}_2 = \sigma_2$. Thereafter, $\mathcal{S}$ fetches the internal GC label information (F, e, d) and $\{R_i^0, R_i^1\}_{i\in[n_2]}$ from the simulated $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. For $i \in [n_2]$, $\mathcal{S}$ acts as P_2 to assert $W_i^0 = R_i^{c_i} \oplus X_{n_1+i}^0$, $W_i^1 = R_i^{c_i\oplus 1} \oplus X_{n_1+i}^1$ and $\hat{\sigma}_{3,i}^{c_i\oplus 1} = \sigma_{3,i}^{c_i\oplus 1}$. In addition, $\mathcal{S}$ uses the internal GC label information (F, e, d) of the simulated $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ together with $\{Z_i\}_{i\in[n_1]}$ to extract P_1's input x_1^*. It then sends $(\textsc{Compute}, \mathsf{sid}, x_1^*)$ to the external $\mathcal{F}_{\mathsf{2pc}}^f$ on behave of P_1.
- Upon receiving $(\textsc{Output}, \mathsf{sid}, P_2)$ from the external $\mathcal{F}_{\mathsf{2pc}}^f$, the simulator $\mathcal{S}$ returns $(\textsc{Deliver}, \mathsf{sid}, P_2)$ if and only if all the checks are valid and $\mathcal{A}$ allows P_2 to finish the protocol execution and obtains y.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \dots, \mathcal{H}_3$.

Hybrid $\mathcal{H}_0$**:** It is the real protocol execution $\textsc{exec}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$**:** $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, the ROT choice bits $b_1, \dots, b_{n_2}$ are true random bits instead of computing from $(b_1, \dots, b_{n_2}) \leftarrow \mathsf{PRG}(k_2^0)$.

Claim 5. *If* $\mathsf{PRG} : \{0,1\}^\lambda \mapsto \{0,1\}^{n_2}$ *is a secure PRG function with adversarial distinguishing advantage* $\mathsf{Adv}_{\mathsf{PRG}}(\mathcal{A}, \lambda)$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with distinguishing advantage* $\mathsf{Adv}_{\mathsf{PRG}}(\mathcal{A}, \lambda)$.

Proof. It is a straightforward reduction; namely, by hybrid argument, we can show that if there exists an adversary $\mathcal{A}$ who can distinguish $\mathcal{H}_1$ from $\mathcal{H}_0$, then we can construct an adversary $\mathcal{B}$ who can break the PRG. □

Hybrid $\mathcal{H}_2$**:** $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, P_2 sends $\{c_i' := b_i\}_{i\in[n_2]}$ to P_1, instead of $\{c_i := b_i \oplus x_{2,i}\}_{i\in[n_2]}$.

Claim 6. *$\mathcal{H}_2$ and $\mathcal{H}_1$ are perfectly indistinguishable.*

Proof. Since b_i are the ROT select bits randomly picked by $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, the distribution of $\{c_i'\}_{i\in[n_2]}$ and $\{c_i\}_{i\in[n_2]}$ are identical. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_1$ are perfectly indistinguishable. □

Hybrid $\mathcal{H}_3$: $\mathcal{H}_3$ is the same as $\mathcal{H}_2$ except that in $\mathcal{H}_3$, P_2 fetches the internal GC label information (F, e, d) and $\{R_i^0, R_i^1\}_{i\in[n_2]}$ from the simulated $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. After receiving $\{W_i^0, W_i^1\}_{i\in[n_2]}$ from P_1, it checks if $W_i^0 = R_i^{c_i} \oplus X_{n_1+i}^0$ and $W_i^1 = R_i^{c_i\oplus 1} \oplus X_{n_1+i}^1$; otherwise, $\mathcal{S}$ aborts.

Claim 7. *If H is a collision resistant cryptographic hash function, $\mathcal{H}_3$ and $\mathcal{H}_2$ are indistinguishable.*

Proof. The difference between $\mathcal{H}_2$ and $\mathcal{H}_3$ is that in $\mathcal{H}_2$, P_2 only checks $H(W_i^0, W_i^1)$ whereas in $\mathcal{H}_3$, P_2 directly checks if $W_i^0 = R_i^{c_i} \oplus X_{n_1+i}^0$ and $W_i^1 = R_i^{c_i\oplus 1} \oplus X_{n_1+i}^1$. It is easy to see when H is a collision resistant cryptographic hash function, $\mathcal{H}_3$ and $\mathcal{H}_2$ are indistinguishable. □

The adversary's view of $\mathcal{H}_3$ is identical to the simulated view $\mathrm{EXEC}_{\mathcal{F}_{\mathsf{2pc}}^f, \mathcal{S}, \mathcal{Z}}$. Therefore, the overall distinguishing advantage is $\mathsf{Adv}_{\mathsf{PRG}}(\mathcal{A}, \lambda)$.

Case 3: P_2 is corrupted; P_1 and $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ are honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ as well as honest P_1. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon receiving $(\textsc{ComputeNotify}, \mathsf{sid}, |x_1|, P_1)$ from the external $\mathcal{F}_{\mathsf{2pc}}^f$, the simulator $\mathcal{S}$ picks random $k_1^0 \leftarrow \{0,1\}^\lambda$. For $i \in [n_2]$, the simulator $\mathcal{S}$ picks random $R_i^0 \leftarrow \{0,1\}^\lambda$, $R_i^1 \leftarrow \{0,1\}^\lambda$, and it computes $\sigma_{1,i}^0 := H(R_i^0), \sigma_{1,i}^1 := H(R_i^1)$. $\mathcal{S}$ then sends $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2\rangle\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ and $\{\sigma_{1,i}^0, \sigma_{1,i}^1\}_{i\in[n_2]}$ to P_2 on behave of P_1.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_1^0, n_2\rangle\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{ROT}, \langle k_2^0, n_2\rangle\rangle)$ from P_2, $\mathcal{S}$ acts as $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ to compute $(b_1, \ldots, b_{n_2}) \leftarrow \mathsf{PRG}(k_2^0)$. $\mathcal{S}$ then fetches $\{R_i^0, R_i^1\}_{i\in[n_2]}$ from the simulated P_1 and sends $\{R_i^{b_i}\}_{i\in[n_2]}$ to P_2.
- When P_1 receives $\{c_i\}_{i\in[n_2]}$ from P_2, $\mathcal{S}$ fetches $\{b_i\}_{i\in[n_2]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$'s internal state. $\mathcal{S}$ then extracts P_2's input $x_{2,i}^* := c_i \oplus b_i$. After that, it sends $(\textsc{Compute}, \mathsf{sid}, x_2^*)$ to the external $\mathcal{F}_{\mathsf{2pc}}^f$ on behave of P_2.
- Upon receiving $(\textsc{Compute}, \mathsf{sid}, y)$ from the external $\mathcal{F}_{\mathsf{2pc}}^f$ for P_2, the simulator $\mathcal{S}$ uses the GC simulator to generate $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f))$.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, \langle k_1^1, f\rangle\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle \mathsf{GC}, f\rangle)$ from P_2 to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$, $\mathcal{S}$ sends (F', d') as the GC tables and decode information to P_2 on behave of $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$. $\mathcal{S}$ then uses X' as the wire labels to generate $\{Z_i\}_{i\in[n_1]}$ and $\{W_i^0, W_i^1, \hat{\sigma}_{3,i}^{c_i\oplus 1}\}_{i\in[n_2]}$ as follows:
  1. For $i \in [n_1]$, set $Z_i := X_i'$;
  2. For $i \in [n_2]$: set $W_i^{x_{2,i}} := X_{n_1+i}' \oplus R_i^{b_i}$, $W_i^{x_{2,i}\oplus 1} := R_i^{b_i\oplus 1}$, and $\hat{\sigma}_{3,i}^{c_i\oplus 1} := H(R_i^{c_i\oplus 1} \oplus X_{n_1+i}^0, R_i^{c_i} \oplus X_{n_1+i}^1)$.
  $\mathcal{S}$ then acts as P_1 to send those messages to P_2.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \ldots, \mathcal{H}_2$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\mathrm{EXEC}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that $\mathcal{H}_1$ uses true random numbers $R_i^0, R_i^1 \leftarrow \{0,1\}^\lambda$ instead of $R_i^b \leftarrow \mathsf{PRF}_{k_1^0}(i, b)$, $b \in \{0,1\}$.

Claim 8. *If $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \{0,1\}^\lambda$ is a secure PRF function with adversarial distinguishing advantage $\mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \lambda)$, then $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable with distinguishing advantage $2n_2 \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \lambda)$.*

Proof. It is a straightforward reduction; namely, by hybrid argument, we can show that if there exists an adversary $\mathcal{A}$ who can distinguish $\mathcal{H}_1$ from $\mathcal{H}_0$, then we can construct an adversary $\mathcal{B}$ who can break the PRF. □

Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that $\mathcal{H}_2$ generates $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f))$, and then it uses X' as the wire labels to generate $\{Z_i\}_{i\in[n_1]}$ and $\{W_i^{x_{2,i}}, \hat{\sigma}_{3,i}^{c_i\oplus 1}\}_{i\in[n_2]}$. $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ also sends (F', d') as the GC tables and decoding information to P_2.

Claim 9. *If GC is simulatable private with adversarial distinguishing advantage $\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{A}, \lambda)$, then $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable with distinguishing advantage $\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{A}, \lambda)$.*

Proof. First of all, by the requirement of simulatable privacy in Def. 2, $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f))$ should be indistinguishable from the real one. Moreover, since P_2 does not know $R_i^{b_i\oplus 1}$, if there is an adversary $\mathcal{A}$ who can distinguish the distribution of $\{W_i^0, W_i^1\}_{i\in[n_2]}$ from the real one with probability ε, then we can construct an adversary $\mathcal{B}$ who has the same distinguishing advantage $\mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{B}, \lambda) = \varepsilon$. □

The adversary's view of $\mathcal{H}_2$ is identical to the simulated view $\mathrm{EXEC}_{\mathcal{F}_{\mathsf{2pc}}^f, \mathcal{S}, \mathcal{Z}}$. Therefore, if GC is simulatable private, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\mathrm{EXEC}_{\Pi_{\mathsf{2pc}}^{\mathsf{OT\text{-}GC}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]}$ and $\mathrm{EXEC}_{\mathcal{F}_{\mathsf{2pc}}^f, \mathcal{S}, \mathcal{Z}}$ are indistinguishable with distinguishing advantage

$$2n_2 \cdot \mathsf{Adv}_{\mathsf{PRF}}(\mathcal{A}, \lambda) + \mathsf{Adv}_{\mathsf{GC}}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda) \ .$$

Case 4: P_1 and P_2 are corrupted; $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$ is honest.

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. The simulator $\mathcal{S}$ simulates the functionality $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{OT\text{-}GC}}]$.

Indistinguishability. This is a trivial case. Since both P_1 and P_2 are controlled by the adversary $\mathcal{A}$, no message is simulated by $\mathcal{S}$.

This concludes the proof.

□