Correlated Randomness Teleportation via Semi-trusted Hardware — Enabling Silent Multi-party Computation

Yibiao Lu[1], Bingsheng Zhang[1], Hong-Sheng Zhou[2], Weiran Liu[3], Lei Zhang[3], and Kui Ren[1,4]

[1] Zhejiang University, {luyibiao,bingsheng,kuiren}@zju.edu.cn
[2] Virginia Commonwealth University, hszhou@vcu.edu
[3] Alibaba Group, weiran.lwr@alibaba-inc.com,zongchao.zl@taobao.com
[4] Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province

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, 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 or maliciously corrupt the hardware. Therefore, she can learn the input of the hardware component and might also tamper its output. We then show how to utilize such semi-trusted hardwares for *correlated randomness teleportation*. 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. When SGX is used to teleport garbled circuits, the resulting two-party computation protocol is 5.3-5.7X and 43-47X faster than the EMP-SH2PC in the LAN and WAN setting, respectively, for the AES-128, SHA-256, and SHA-512 evaluation. We also show how to achieve malicious security with little overhead.

Keywords: MPC · semi-trusted hardware model · garbled circuit.

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 [5] raises a security concern on possible backdoors contained in its design. When the hardware manufacturer is not fully trusted, sensitive information may be leaked through backdoors, 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) [11]. 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 4096-bit RSA signatures.) That means the input of SGX might be revealed to the adversary.

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? 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 $\mathcal{F}_{\text{HW}}$. $\mathcal{F}_{\text{HW}}$ is parameterized with a probabilistic polynomial time (PPT) interactive Turing machine (ITM) M, which specifies its functionality. When the hardware functionality $\mathcal{F}_{\text{HW}}$ is passively corrupted, the adversary $\mathcal{A}$ can learn all the incoming messages received by $\mathcal{F}_{\text{HW}}$; when $\mathcal{F}_{\text{HW}}$ 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 $\mathcal{F}_{\text{HW}}$.

We note that the existing remote attestation model [18], tamper-proof hardware token models [9, 12], and *server-aided* model [15] are different from our model. When hardware is fully trusted, unlike the remote attestation model, our $\mathcal{F}_{\text{HW}}$ does not sign its output. Moreover, the existing model does not address hardware leakage as well as malicious corruptions.

Our constructions. We show semi-trusted hardware can still be used to significantly improve the efficiency of an MPC protocol by reducing the communication. 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. We propose a new notion called *correlated randomness teleportation*, where the sender can teleport a large amount of correlated randomness to the receiver with little communication. 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 to the Receiver's SGX enclave via a secure channel, and the Receiver also sends a random seed k_2 to the enclave locally. Both parties can then generate polynomially

many ROT copies without any further communication. Namely, for a ROT copy, the Sender locally computes $R^0_{\mathsf{ctr}} \leftarrow \mathsf{PRF}_{k_1}(\mathsf{ctr}, 0)$ and $R^1_{\mathsf{ctr}} \leftarrow \mathsf{PRF}_{k_1}(\mathsf{ctr}, 1)$ from the seed k_1 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 using some pseudo-random number generator PRG, and then it computes $R^{b_{\mathsf{ctr}}}_{\mathsf{ctr}} \leftarrow \mathsf{PRF}_{k_1}(\mathsf{ctr}, b_{\mathsf{ctr}})$. The SGX locally outputs $\{R^{b_{\mathsf{ctr}}}_{\mathsf{ctr}}\}_{\mathsf{ctr}}$ to the Receiver.

Garbled circuit (GC) can also be viewed as a type of correlated randomness. With our technique, the communication between the 2PC players can also be dramatically reduced. 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 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. The overall communication is linear to 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 GC in an offline phase won't result in a 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 in the real/hybrid world. The protocol should invoke the secure hardware component at the right moment along with the 2PC protocol execution.

Efficiency. We mainly compare the performance of our protocols with the well-known EMP-toolkit maintained by Wang *et al.* [20]. Table. 1 shows the performance comparison between the passively secure IKNP OT extension protocol [8] implemented in EMP-toolkit [20] 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 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 [20]. 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.

Table. 2 shows the performance comparison between EMP-SH2PC [20] 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 [22] 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, as for the Garbler in our protocol, the garbling time is the time to generate input wire labels. We omit the time of transmitting seeds and wire labels in both protocols. 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

Table 1: Performance comparison of the ROT protocols. Result obtained from Dell OptiPlex 7080 (Intel Core 8700 CPU @ 3.20 GHz, 32 GB RAM, OS: Ubuntu 18.04 LTS). LAN: 1Gbps bandwidth, 0.1ms delay. WAN: 100Mbps bandwidth, 25ms delay.

# ROT	Network	Sender's running time (in ms)		Receiver's running time (in ms)	
		EMP-IKNP-ROT [20]	Our ROT	EMP-IKNP-ROT [20]	Our ROT
1×10^4	LAN	2.889	0.074	3.908	0.162
	WAN	26.331	0.079	76.358	0.169
1×10^5	LAN	17.790	0.780	19.355	1.575
	WAN	150.502	0.795	200.030	1.477
1×10^6	LAN	154.373	6.182	150.621	15.910
	WAN	1451.043	6.402	1495.294	16.032
1×10^7	LAN	1507.961	51.616	1451.562	103.937
	WAN	13859.934	51.280	13963.502	103.435
1×10^8	LAN	15030.832	505.289	14470.057	995.987
	WAN	138028.607	501.757	137034.187	980.795

Table 2: Performance comparison of the generation, transmission and evaluation process of the garbled circuit in the 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	EMP-SH2PC [20] time (in ms)			Our 2PC protocol time (in ms)		
		Garbler	Comm.	Evaluator	Garbler	Comm.	Evaluator (SGX+PC)
AES-128	LAN	246.557	1742.094	229.339	10.171	≈ 0	243.730 + 174.916
	WAN	265.919	18335.009	234.264	9.875	≈ 0	255.275 + 177.637
SHA-256	LAN	829.398	6135.087	776.880	26.310	≈ 0	805.893 + 583.828
	WAN	839.626	64433.208	777.284	28.981	≈ 0	804.904 + 581.166
SHA-512	LAN	2434.915	15745.170	2388.890	52.110	≈ 0	2061.712 + 1549.076
	WAN	2303.479	163362.579	2418.025	52.373	≈ 0	2072.215 + 1551.586

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	EMP-AG2PC [20] running time (in ms)				Ours (in ms)	
		Garb. offline	Garb. online	Eval. offline	Eval. online	Garbler	Evaluator
AES-128	LAN	94.744	5.185	92.055	5.193	3.100	6.311
	WAN	1345.708	53.440	1240.956	53.385	30.124	61.457
SHA-256	LAN	210.676	6.303	201.701	6.272	10.373	15.633
	WAN	2299.404	52.474	2196.297	52.440	47.756	86.059
SHA-512	LAN	435.581	9.634	423.302	9.593	25.756	34.944
	WAN	4095.115	56.471	4044.428	56.426	70.139	112.336

(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 5.3-5.7X faster than the

EMP-SH2PC [20]. In the WAN setting, our silent 2PC protocol is 43-47X faster than the EMP-SH2PC.

Table. 3 shows the performance comparison between EMP-AG2PC [20] and our malicious setting silent 2PC protocol. (EMP-AG2PC implements an efficient maliciously secure two-party computation protocol, authenticated garbling [21].) We perform the experiments on this same machine as above. We take the AES-128, SHA-256, and SHA-512 circuit evaluations 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 17-32X faster w.r.t. the garbler's running time and 12-15X faster w.r.t. the evaluator's running time than the EMP-AG2PC [20]. In the WAN setting, our silent PC protocol is 46-59X faster w.r.t. the garbler's running time and 21-36X 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. We abbreviate *probabilistic polynomial time* as PPT, and *interactive Turing machine* as ITM. 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 [3], 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 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 [3]). *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 [3]). *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:*

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

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.* [22].

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.* [2] 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.* [14] 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.

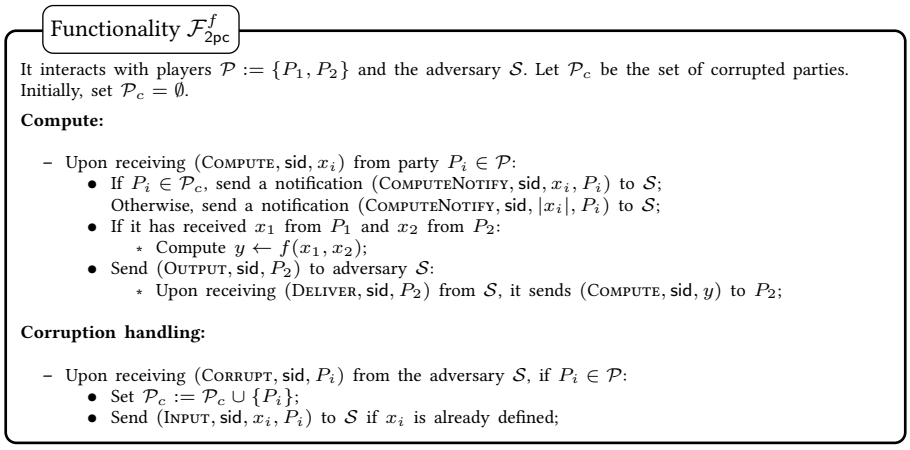

Fig. 1: Functionality $\mathcal{F}_{\mathsf{2pc}}^{f}$

We note that, in a conventional 2PC setting, the other optimization techniques, such as *GRR2* [19] and *half-gates* [22], 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.

3 Security Model

Simulation-based Security. Our security model follows the simulation paradigm, 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 a simulation-based security model, 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 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.

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

It interacts with players $\mathcal{P} := \{P_1, P_2\}$ and the adversary $\mathcal{A}$. It is parameterized with a PPT ITM M and a Boolean flag corrupted.
Initially, set corrupted := false.

- Upon receiving $(\textsc{Corrupt}, \mathsf{sid}, \mathsf{M}^*)$ from $\mathcal{A}$:
  - Set corrupted := 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 corrupted = true:
    - Send leakage message $(\textsc{RunNotify}, \mathsf{sid}, x_i, P_i)$ to $\mathcal{A}$;
  - If corrupted = 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 $(y_1, y_2) \leftarrow \mathsf{M}(x_1, x_2)$;
    - For $i \in \{1, 2\}$, send $(\mathsf{Run}, \mathsf{sid}, y_i)$ to P_i;

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

The ideal world execution. In the ideal world, P_1 and P_2 only communicate with an ideal functionality $\mathcal{F}_{\mathsf{2pc}}^f$ during the execution. As depicted in Fig. 1, party $P_i \in \mathcal{P}$ sends $(\textsc{Compute}, \mathsf{sid}, x_i)$ to the functionality $\mathcal{F}_{\mathsf{2pc}}^f$, and $\mathcal{F}_{\mathsf{2pc}}^f$ sends a notification $(\textsc{ComputeNotify}, \mathsf{sid}, x_i, P_i)$ to the adversary $\mathcal{S}$ if P_i is corrupted; Otherwise, $\mathcal{F}_{\mathsf{2pc}}^f$ leaks the input size $(\textsc{ComputeNotify}, \mathsf{sid}, |x_i|, P_i)$ to $\mathcal{S}$. When both parties' inputs are received, $\mathcal{F}_{\mathsf{2pc}}^f$ 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 $P_i \in \mathcal{P}$, $\mathcal{F}_{\mathsf{2pc}}^f$ 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 $\mathcal{F}_{\text{HW}}$. Later, we will discuss how $\mathcal{F}_{\text{HW}}$ is instantiated by Intel SGX in practice. For notation simplicity, we define $\mathcal{F}_{\text{HW}}$ as a template, and specify the required functionalities in the description of a PPT Turing machine M. We use $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ in our semi-honest/malicious setting protocol $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$.

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 $\mathcal{F}_{\text{HW}}[\mathsf{M}]$ shown in Fig. 2 can be corrupted by the adversary $\mathcal{A}$. The functionality $\mathcal{F}_{\text{HW}}[\mathsf{M}]$ is parameterized with a PPT ITM M and a Boolean flag corrupted to indicate whether the hardware is corrupted. The parties P_1 and P_2 can invoke $\mathcal{F}_{\text{HW}}[\mathsf{M}]$ to compute $(y_1, y_2) \leftarrow \mathsf{M}(x_1, x_2)$ by sending the input x_1 and x_2 respectively to $\mathcal{F}_{\text{HW}}$.

However, the adversary $\mathcal{A}$ is allowed to corrupt $\mathcal{F}_{\text{HW}}$ 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 $\mathcal{F}_{\text{HW}}$ 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 $\mathcal{F}_{\text{HW}}$ computes $(y_1, y_2) \leftarrow$

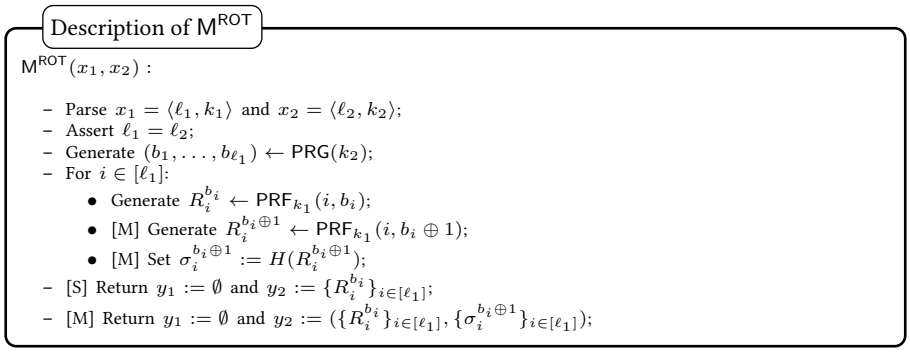

Fig. 3: Description of $\mathsf{M}^{\mathsf{ROT}}$

$\mathsf{M}^*(x_1, x_2)$ instead. After the computation, $\mathcal{F}_{\text{HW}}$ sends the output y_1 to the party P_1 and y_2 to the party P_2.

4 Correlated Randomness Teleportation

Correlated randomness is widely used in the MPC offline protocols to achieve better online efficiency. In practice, correlated randomness can be generated and distributed by a trusted server. However, this approach still needs huge communication between the trusted server and the players to deliver those correlated random copies. In this section, we show it is possible to utilize a semi-trusted hardware to teleport correlated randomness with little ($O(\lambda)$) communication. Take two-party computation as an example. Without loss of generality, suppose $\mathcal{F}_{\text{HW}}$ is located at P_2's side with fast local connections, e.g., $\mathcal{F}_{\text{HW}}$ is instantiated with P_2's SGX. In the following, we provide Random OT teleportation and GC teleportation protocols to illustrate our idea.

4.1 Random OT Teleportation

Description of $\mathsf{M}^{\mathsf{ROT}}$. We now define the Turing machine $\mathsf{M}^{\mathsf{ROT}}$ for $\mathcal{F}_{\text{HW}}$ in Fig. 3. We use [S] (or [M]) labels to indicate instructions only included in the machine used in the semi-honest (or malicious) setting protocol. Unlabeled instructions are performed in both settings.

When P_1 sends $\langle \ell_1, k_1 \rangle$ and P_2 sends $\langle \ell_2, k_2 \rangle$, $\mathsf{M}^{\mathsf{ROT}}$ parses their inputs to obtain the ROT seeds k_1, k_2 and the number of ROT to be generated ℓ_1, ℓ_2, and it asserts P_1 and P_2 send the same number $\ell_1 = \ell_2$. Subsequently, $\mathsf{M}^{\mathsf{ROT}}$ use k_2 to generate the ROT select bits by $(b_1, \ldots, b_{\ell_1}) \leftarrow \mathsf{PRG}(k_2)$. Then, $\mathsf{M}^{\mathsf{ROT}}$ computes $R_i^{b_i} \leftarrow \mathsf{PRF}_{k_1}(i, b_i)$, for $i \in [\ell_1]$. In the semi-honest setting, $\mathsf{M}^{\mathsf{ROT}}$ can simply returns the ROT copies $\{R_i^{b_i}\}_{i \in [\ell_1]}$ to P_2.

In the malicious setting, in addition to generate the ROT copies, $\mathsf{M}^{\mathsf{ROT}}$ needs to produce some verification messages. More specifically, after generating a ROT copy $R_i^{b_i}$, $\mathsf{M}^{\mathsf{ROT}}$ also generates $R_i^{b_i \oplus 1} \leftarrow \mathsf{PRF}_{k_1}(i, b_i \oplus 1)$, and it sets $\sigma_i^{b_i \oplus 1} := H(R_i^{b_i \oplus 1})$

Protocol Π_{ROT}

Protocol description:

- Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_2 := \ell)$ from the environment $\mathcal{Z}$, the party P_2:
  - Pick random $k_2 \leftarrow \{0,1\}^\lambda$;
  - Generate $(b_1, \ldots, b_\ell) \leftarrow \mathsf{PRG}(k_2)$;
  - Send $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_2 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$;
- Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_1 := \ell)$ from the environment $\mathcal{Z}$, the party P_1:
  - Pick random $k_1 \leftarrow \{0,1\}^\lambda$;
  - For $i \in [\ell]$:
    - Generate $R_i^0 \leftarrow \mathsf{PRF}_{k_1}(i, 0)$ and $R_i^1 \leftarrow \mathsf{PRF}_{k_1}(i, 1)$;
    - [M] Set $\sigma_i^0 := H(R_i^0)$ and $\sigma_i^1 := H(R_i^1)$;
  - [M] Set $\tau := H(\{\sigma_i^0, \sigma_i^1\}_{i \in [\ell]})$;
  - [S] Send $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_1 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$;
  - [M] Send $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_1 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$ and send τ to P_2;
  - Return $(\textsc{Compute}, \mathsf{sid}, \{R_i^0, R_i^1\}_{i \in [\ell]})$ to the environment $\mathcal{Z}$;
- [S] Upon receiving $(\mathsf{Run}, \mathsf{sid}, \{R_i^{b_i}\}_{i \in [\ell]})$, the party P_2:
  - Return $(\textsc{Compute}, \mathsf{sid}, \{R_i^{b_i}\}_{i \in [\ell]})$ to the environment $\mathcal{Z}$;
- [M] Upon receiving $(\mathsf{Run}, \mathsf{sid}, (\{R_i^{b_i}\}_{i \in [\ell]}, \{\hat{\sigma}_i^{b_i \oplus 1}\}_{i \in [\ell]}))$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$ and receiving τ from P_1, the party P_2:
  - For $i \in [\ell]$, set $\hat{\sigma}_i^{b_i} := H(R_i^{b_i})$;
  - Set $\hat{\tau} := H(\{\hat{\sigma}_i^0, \hat{\sigma}_i^1\}_{i \in [\ell]})$;
  - Assert $\hat{\tau} = \tau$;
  - Return $(\textsc{Compute}, \mathsf{sid}, \{R_i^{b_i}\}_{i \in [\ell]})$ to the environment $\mathcal{Z}$;

Fig. 4: The semi-honest/malicious setting Π_{ROT} in the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$-hybrid model

as the verification message. In the end, $\mathsf{M}^{\mathsf{ROT}}$ returns the ROT messages $\{R_i^{b_i}\}_{i \in [\ell_1]}$ and the verification messages $\{\sigma_i^{b_i \oplus 1}\}_{i \in [\ell_1]}$ to P_2.

Description of Π_{ROT}. We depict our semi-honest/malicious setting protocol in Fig. 4, where ℓ is the number of ROT copies P_1 and P_2 want to generate. We use [S] (or [M]) labels to indicate instructions only included in the semi-honest (or malicious) setting protocol. Other instructions not labeled should be included in both settings.

The semi-honest setting. In the semi-honest setting protocol, the party P_2 first picks a random $k_2 \leftarrow \{0,1\}^\lambda$ as its ROT seed, and it uses this seed to generate $(b_1, \ldots, b_\ell) \leftarrow \mathsf{PRG}(k_2)$ as the ROT select bits. Then, P_2 sends $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_2 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$. The party P_1 also picks a random $k_1 \leftarrow \{0,1\}^\lambda$ as its ROT seed, and it uses this seed to generate $R_i^0 \leftarrow \mathsf{PRF}_{k_1}(i, 0)$ and $R_i^1 \leftarrow \mathsf{PRF}_{k_1}(i, 1)$, for $i \in [\ell]$. Subsequently, P_1 sends $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_1 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$, and it returns $(\textsc{Compute}, \mathsf{sid}, \{R_i^0, R_i^1\}_{i \in [\ell]})$ to the environment $\mathcal{Z}$. After that, P_2 receives the ROT copies $\{R_i^{b_i}\}_{i \in [\ell]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$.

The malicious setting. In the malicious setting protocol, the party P_2 first picks a random $k_2 \leftarrow \{0,1\}^\lambda$ as its ROT seed, and it uses this seed to generate $(b_1, \ldots, b_\ell) \leftarrow \mathsf{PRG}(k_2)$ as the ROT select bits. Then, P_2 sends $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_2 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$. The party P_1 also picks a random $k_1 \leftarrow \{0,1\}^\lambda$ as its ROT seed. For $i \in [\ell]$, P_1 generates $R_i^0 \leftarrow \mathsf{PRF}_{k_1}(i, 0)$ and $R_i^1 \leftarrow \mathsf{PRF}_{k_1}(i, 1)$, and it sets $\sigma_i^0 := H(R_i^0)$ and $\sigma_i^1 := H(R_i^1)$. Subsequently, it sets a hash value of all these hash values $\tau := H(\{\sigma_i^0, \sigma_i^1\}_{i \in [\ell]})$. P_1 then sends $(\mathsf{Run}, \mathsf{sid}, \langle \ell, k_1 \rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$ and sends τ to P_2, and it returns $(\textsc{Compute}, \mathsf{sid}, \{R_i^0, R_i^1\}_{i \in [\ell]})$ to the environment $\mathcal{Z}$. After that, P_2 receives the ROT

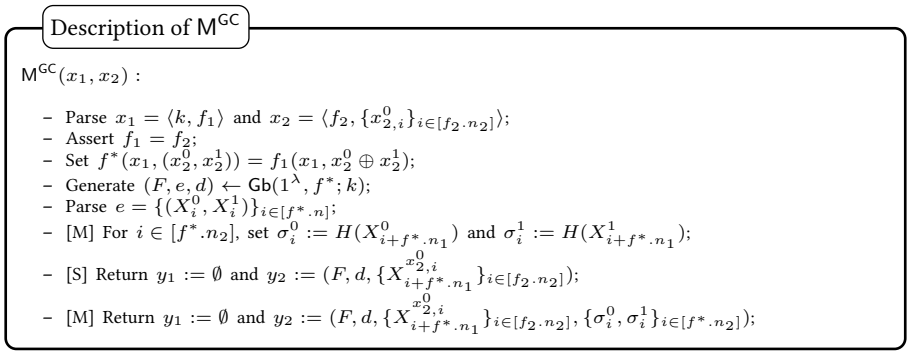

Fig. 5: Description of M^{GC}

copies $\{R^{b_i}_i\}_{i\in[\ell]}$ and hash values $\{\hat{\sigma}^{b_i\oplus 1}_i\}_{i\in[\ell]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{ROT}}]$ and τ from P_2. For $i \in [\ell]$, P_2 sets $\hat{\sigma}^{b_i}_i := H(R^{b_i}_i)$. At last, P_2 sets $\hat{\tau} := H(\{\hat{\sigma}^0_i, \hat{\sigma}^1_i\}_{i\in[\ell]})$ and asserts $\hat{\tau} = \tau$ to check these hash values.

Security. When SGX is malicious, it may produce incorrect $R^{b_i}_i$. To check the correctness of $R^{b_i}_i$ at a low communication cost while preventing P_1 from learning b_i, we let P_1 and SGX collaboratively generate verification messages. More specifically, SGX will send hash values of R^0_i and R^1_i to P_2 (since P_2 can generate $H(R^{b_i}_i)$ by itself, only $H(R^{b_i\oplus 1}_i)$ is needed). Meanwhile, P_1 computes and sends $\tau = H(\{H(R^{b_i}_i), H(R^{b_i\oplus 1}_i)\}_{i\in[\ell]})$ to P_2. This hash value τ can be used to verify the validity of SGX's outputs later. Due to space limitation, the full proof can be found in the full version.

4.2 GC Teleportation with applications to silent 2PC

Description of M^{GC}. We now define the Turing machine M^{GC} for $\mathcal{F}_{\mathrm{HW}}$ that will be used for our 2PC protocol in the semi-honest/malicious adversarial setting (cf. Fig. 5). We use [S] (or [M]) labels to indicate instructions only included in the machine used in the semi-honest (or malicious) setting protocol. Unlabeled instructions are performed in both settings.

When P_1 sends $\langle k, f_1\rangle$ and P_2 sends $\langle f_2, \{x^0_{2,i}\}_{i\in[f_2.n_2]}\rangle$, M^{GC} parses their inputs to obtain the GC seed k, the circuit to be computed and P_2's secret-shared input x^0_2. M^{GC} asserts P_1 and P_2 send the same circuit $f_1 = f_2$, and use f_1 to generate a function $f^*(x_1, (x^0_2, x^1_2)) = f_1(x_1, x^0_2 \oplus x^1_2)$. M^{GC} then generates the garbled circuit by $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f^*; k)$, and it parses the encoding information $e = \{(X^0_i, X^1_i)\}_{i\in[f^*.n]}$ to get the input wire labels. In the semi-honest setting, M^{GC} can simply returns (F, d) and the wire label of x^0_2 to P_2.

In the malicious setting, in addition to generate the GC copy, M^{GC} needs to produce some verification messages. More specifically, after parsing the encoding information, M^{GC} sets $\sigma^0_i := H(X^0_{i+f^*.n_1})$ and $\sigma^1_i := H(X^1_{i+f^*.n_1})$, for $i \in [f^*.n_2]$. These hash values $\{\sigma^0_i, \sigma^1_i\}_{i\in[f^*.n_2]}$ can help P_2 to verify that it receives the cor-

Protocol $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$

We define $f^*(x_1, (x_2^0, x_2^1)) = f(x_1, x_2^0 \oplus x_2^1)$, both x_2^0 and x_2^1 are P_2's inputs, so $n_1^* = n_1, n_2^* = 2n_2, n^* = n_1^* + n_2^*$.

- Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_2 := (x_{2,1}, \ldots, x_{2,n_2}))$ from $\mathcal{Z}$, the party P_2:
  - For $i \in [n_2]$, pick random $x_{2,i}^0 \leftarrow \{0,1\}$, and set $x_{2,i}^1 = x_{2,i} \oplus x_{2,i}^0$;
  - Send $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x_{2,i}^0\}_{i\in[n_2]}\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and $\{x_{2,i}^1\}_{i\in[n_2]}$ to P_1;
- Upon receiving $(\textsc{Compute}, \mathsf{sid}, x_1 := (x_{1,1}, \ldots, x_{1,n_1}))$ from the environment $\mathcal{Z}$ and $\{x_{2,i}^1\}_{i\in[n_2]}$ from P_2, P_1:
  - Pick random $k \leftarrow \{0,1\}^\lambda$;
  - Generate $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f^*; k)$;
  - Parse $e = \{(X_i^0, X_i^1)\}_{i\in[n^*]}$;
  - [M] For $i \in [n_2^*]$, set $\sigma_i^0 := H(X_{i+n_1}^0)$ and $\sigma_i^1 := H(X_{i+n_1}^1)$;
  - [M] Send $\tau := H(F, d, \{\sigma_i^0, \sigma_i^1\}_{i\in[n_2^*]})$ to P_2;
  - Send $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and send $\{Z_i = X_i^{x_{1,i}}\}_{i\in[n_1]}, \{Z_{i+n_1+n_2} = X_{i+n_1+n_2}^{x_{2,i}^1}\}_{i\in[n_2]}$ to P_2;
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, (\hat{F}, \hat{d}, \{Z_{i+n_1} = X_{i+n_1}^{x_{2,i}^0}\}_{i\in[n_2]}))$ (and [M] $\{\hat{\sigma}_i^0, \hat{\sigma}_i^1\}_{i\in[n_2^*]}$) from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and receiving $\{Z_i\}_{i\in[n_1]}, \{Z_{i+n_1+n_2}\}_{i\in[n_2]}$ (and [M] τ) from P_1, party P_2:
  - [M] Set $\hat{\tau} := H(\hat{F}, \hat{d}, \{\hat{\sigma}_{0,i}^0, \hat{\sigma}_{0,i}^1\}_{i\in[n_2^*]})$;
  - [M] Assert $\hat{\tau} = \tau$;
  - [M] For $i \in [n_2]$, assert $\hat{\sigma}_i^{x_{2,i}^0} = H(Z_{i+n_1})$ and $\hat{\sigma}_{i+n_2}^{x_{2,i}^1} = H(Z_{i+n_1+n_2})$;
  - Evaluate $Y \leftarrow \mathsf{GC.Ev}(\hat{F}, (Z_1, \ldots, Z_{n^*}))$;
  - Decode $y \leftarrow \mathsf{GC.De}(\hat{d}, Y)$;
  - Return $(\textsc{Compute}, \mathsf{sid}, y)$ to the environment $\mathcal{Z}$;

Fig. 6: The semi-honest/malicious setting protocol $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$ in the $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$-hybrid model

rect input wire labels from P_1 in the subsequent execution. In the end, M^{GC} returns $(F, d, \{X_{i+f^*.n_1}^{x_{2,i}^0}\}_{i\in[f_2.n_2]}, \{\sigma_i^0, \sigma_i^1\}_{i\in[f^*.n_2]})$ to P_2.

Instantiation of M^{GC}. In practice, M^{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.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.

Description of $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$. We depict our semi-honest/malicious setting protocol in Fig. 6, 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 addition, we define a modified function $f^*(x_1, (x_2^0, x_2^1)) = f(x_1, x_2^0 \oplus x_2^1)$, in which x_2^0 and x_2^1 are the additive secret shares of P_2's original input x_2. This idea of splitting P_2's inputs is from the work of Mohassel *et al.* [16], in their setting, there are two garblers and one evaluator, and the evaluator secret-shares its inputs and sends shares to the garblers. We use [S] (or [M]) labels to indicate instructions

only included in the semi-honest (or malicious) setting protocol. Other instruction not labeled should be included in both the semi-honest setting protocol and the malicious setting protocol.

The semi-honest setting. In the semi-honest setting protocol, the party P_2 first secret shares its input $x_{2,i}$ as $x_{2,i} = x^0_{2,i} \oplus x^1_{2,i}$, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x^0_{2,i}\}_{i\in[n_2]}\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and $\{x^1_{2,i}\}_{i\in[n_2]}$ to P_1. After receiving the secret shares of P_2's inputs $\{x^1_{2,i}\}_{i\in[n_2]}$, P_1 picks a random $k \leftarrow \{0,1\}^\lambda$ as the seed of GC, it generates a GC with this seed by $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f^*; k)$ and it parses the input wire labels by $e = \{(X^0_i, X^1_i)\}_{i\in[n^*]}$. Then, P_1 sends $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and it sends the input wire labels of its own inputs $\{Z_i = X_i^{x_{1,i}}\}_{i\in[n_1]}$ and P_2's input shares $\{Z_{i+n_1+n_2} = X^{x^1_{2,i}}_{i+n_1+n_2}\}_{i\in[n_2]}$ to P_2. Subsequently, P_2 receives the garbled tables F, the decoding information d and the input wire labels of $\{x^0_{2,i}\}_{i\in[n_2]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and it receives P_1's input wire labels and the input wire labels of $\{x^1_{2,i}\}_{i\in[n_2]}$, it evaluates the garbled circuit by $Y \leftarrow \mathsf{GC.Ev}(F, (Z_1, \ldots, Z_{n_1+2n_2}))$, and decodes the output value by $y \leftarrow \mathsf{GC.De}(d, Y)$.

The malicious setting. In the malicious setting protocol, the party P_2 first secret shares its input $x_{2,i}$ as $x_{2,i} = x^0_{2,i} \oplus x^1_{2,i}$, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x^0_{2,i}\}_{i\in[n_2]}\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and $\{x^1_{2,i}\}_{i\in[n_2]}$ to P_1. After receiving the secret shares of P_2's inputs $\{x^1_{2,i}\}_{i\in[n_2]}$, P_1 picks a random $k \leftarrow \{0,1\}^\lambda$ as the seed of GC, it generates a GC with this seed by $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f^*; k)$ and it parses the input wire labels by $e = \{(X^0_i, X^1_i)\}_{i\in[n^*]}$. Then, P_1 computes the hash values of all P_2's input wire labels, $\sigma^0_i := H(X^0_{i+n_1})$ and $\sigma^1_i := H(X^1_{i+n_1})$, for $i \in [n^*_2]$, in addition, it computes another hash value of these all hash values and the garbled circuit by $\tau = H(F, d, \{\sigma^0_i, \sigma^1_i\}_{i\in[n^*_2]})$. After that, P_1 sends $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ to $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and it sends the hash value τ, the input wire labels of its own inputs $\{Z_i = X_i^{x_{1,i}}\}_{i\in[n_1]}$ and P_2's input shares $\{Z_{i+n_1+n_2} = X^{x^1_{2,i}}_{i+n_1+n_2}\}_{i\in[n_2]}$ to P_2. Subsequently, P_2 receives the garbled tables F, the decoding information d, the input wire labels of $\{x^0_{2,i}\}_{i\in[n_2]}$ and the hash valuse of all its input wire labels $\{\hat{\sigma}^0_i, \hat{\sigma}^1_i\}_{i\in[n^*_2]}$ from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and it receives τ, P_1's input wire labels and the input wire labels of $\{x^1_{2,i}\}_{i\in[n_2]}$. Then, P_2 checks the message sent by $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$ with the hash value τ, and it verifies that $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and P_1 sends the correct input wire labels using the hash values from $\mathcal{F}_{\mathrm{HW}}[\mathsf{M}^{\mathsf{GC}}]$. At last, P_2 evaluates the garbled circuit by $Y \leftarrow \mathsf{GC.Ev}(F, (Z_1, \ldots, Z_{n_1+2n_2}))$, and decodes the output value by $y \leftarrow \mathsf{GC.De}(d, Y)$.

5 Security

In this section, 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 $\Pi^{\mathsf{GC}}_{\mathsf{2pc}}$ in Thm. 1/Thm. 2, respectively, where we restrict the adversary $\mathcal{A}$ to only corrupt one of the following entities (i) the semi-trusted hardware functionality, (ii) player P_1 and (iii) player P_2.

In our protocols, P_2's input x_2 is secretly shared as $x_2 = x_2^0 \oplus x_2^1$, and P_2 sends x_2^0 to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and x_2^1 to P_1. $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and P_1 will not be corrupted simultaneously, so the adversary can not learn P_2's input value.

In the semi-honest setting, the view of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ is the MPC function f, a random input share of x_2 and the seed of the garbled circuit, f is already known to the environment $\mathcal{Z}$ and the adversary $\mathcal{A}$; therefore, no additional information would be leaked to the adversary $\mathcal{A}$. Since $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ could only be passively corrupted, the correctness of the garbled circuit and the wire labels of P_2's secret shared input are preserved. The input privacy of protocol $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$ is guaranteed by the simulatable privacy property of the underlying garbling scheme GC. In the malicious setting, $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$, P_1, and P_2 may be maliciously corrupted. The main design principle is as follows. In P_2's point of view, either $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and messages sent by P_1 to carry out a mutual verification, and it aborts if any inconsistency is detected. More specifically, the wire labels of P_2's secret shared input x_2^0 and x_2^1 are checked using hash values $\sigma_i^0 := H(X_{i+n_1}^0)$ and $\sigma_i^1 := H(X_{i+n_1}^1)$ generated by $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$. The correctness of the garbled circuit and the hash values are ensured by another hash value $\tau = H(F, d, \{\sigma_i^0, \sigma_i^1\}_{i\in[n_2^*]})$.

Theorem 1. *If* $\mathsf{GC} := (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is a secure simulatable private garbling scheme, protocol* $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$ *(semi-honest setting) described in Fig. 6 securely realizes* $\mathcal{F}_{\mathsf{2pc}}^f$ *as described in Fig. 1 in the* $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$*-hybrid model against any PPT semi-honest adversaries who can corrupt one of the following entities: (i)* $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$*, (ii)* P_1*, or (iii)* P_2 *with static corruption.*

Theorem 2. *If* $H : \{0,1\}^* \mapsto \{0,1\}^\lambda$ *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* $\Pi_{\mathsf{2pc}}^{\mathsf{GC}}$ *(malicious setting) described in Fig. 6 securely realizes* $\mathcal{F}_{\mathsf{2pc}}^f$ *as described in Fig. 1 in the* $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$*-hybrid model against any PPT malicious adversaries who can corrupt one of the following entities: (i)* $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$*, (ii)* P_1*, or (iii)* P_2 *with static corruption.*

The proofs are provided in Appendix A.1.

6 Implementation and Benchmarks

Our protocol is implemented in C++ using Intel SGX SDK on Linux. We use AES-NI for the PRF algorithm. 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 semi-honest ROT generation protocol, we compared our protocol with the implementation of the IKNP OT extension protocol [8] in EMP-OT [20]. Table. 1 shows the performance comparison for generating 10^4 to 10^8 copies of ROT, where the result is the average of 10 tests.

Table 4: Details of the benchmark Bristol Fashion circuit

Circuit	# wire	# gate	# AND gate	# P_1's input	# P_2's input	# output
AES-128	36919	36663	6400	128	128	128
SHA-256	135841	135073	22573	256	256	256
SHA-512	351153	349617	57947	512	512	512

To test the performance of the 2PC protocols, our benchmarks use three Bristol Fashion format circuits [1], and the details are provided in Table. 4. For the semi-honest setting protocol, we compared our protocol with EMP-SH2PC [20] (EMP-SH2PC provides an efficient semi-honest 2PC implementation based on Yao's GC protocol with half-gates [22] optimization); for the malicious setting protocol, we compared our protocol with EMP-AG2PC [20] (EMP-AG2PC implements an efficient maliciously secure two-party computation protocol, authenticated garbling [21]). 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 benchmark circuits once using the malicious version, and the results are the average of 100 tests.

7 Related Work

As mentioned above, there are several hardware models proposed in the literature, such as the remote attestation model [18] and the tamper-proof hardware token models [9, 12]. However, the existing model does not address hardware leakage as well as malicious corruptions. Mohassel *et al.* [15] 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. However, their objective is to save computation, while our goal is to reduce communication. Moreover, in our model, the hardware can be maliciously corrupted. Järvinen *et al.* [10] used hardware token to reduce the cost of the OT process in standard GC protocols. In their protocol, a sender generates a garbled circuit and it uses hardware tokens, e.g. One-Time Memory (OTM) tokens, to store the GC encoding information, the garbled circuit and the hardware tokens are collectively called One-Time Program (OTP), which is a non-interactive version of GC protocol. In our work, we also remove the OT process, but to keep sensitive information away from the enclave, we secret-share P_2's input and sends the shares to P_1 and SGX. A similar idea can be found in Mohassel *et al.* [16]. Kolesnikov [13] used hardware tokens to construct an efficient OT protocol. This work considers the client-server setting where the server is the sender and the client is the receiver. The server can deploy a hardware token in the client side, and the client can obtain messages by querying the token. Our work provides a more efficient malicious setting protocol, instead of the cut-and-choose technique.

There have been some Intel SGX-based MPC solutions. Gupta *et al.* [7] proposed protocols using Intel SGX for SFE problem which is secure in the semi-honest model,

and show how to improve their protocol's security.. The naive solution is to let the players enter their inputs to the enclave, and they reduce the data leakage problem by using SGX to convert plaintexts to ciphertexts (e.g. wire labels) and vice versa, but the enclave still knows the input values. They notice the problem that the players need to trust hardware supplier when using Intel SGX, but don't give a feasible solution. Felsen *et al.* [6] 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. Choi *et al.* [4] 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, in the odd rounds, the computation is executed in the enclave and the player Bob (the remote party) only provide less sensitive inputs, in the even rounds, a scheme based on garbled circuit is used and Bob provides more sensitive data. These works focus on the efficiency of the Intel SGX-based solutions, and the main security concern is the side-channel attack problem. Providing private information to enclave is an inevitable step of their protocols; therefore, private information may be leaked in our setting.

8 Conclusion

In this work, we investigate the problem where the trusted hardware manufacturer is not fully trusted, and the hardware components may leak sensitive information to the remote servers. In our model, the adversary is allowed to passively or maliciously corrupt the hardware component. We present several correlated randomness teleportation protocols, such as ROT and GC generation with applications to silent MPC, where the communication only depends on the input size regardless the circuit size. The resulting protocols are significantly faster than the EMP-IKNP-ROT, EMP-SH2PC and EMP-AG2PC.

References

1. Archer, D., Abril, V.A., Lu, S., Maene, P., Mertens, N., Sijacic, D., Smart, N.: 'Bristol Fashion' MPC Circuits (2020), https://homes.esat.kuleuven.be/~nsmart/MPC/ Accessed January 5th, 2021
2. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: Proceedings of the twenty-second annual ACM symposium on Theory of computing. pp. 503–513 (1990)
3. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 784–796 (2012)
4. Choi, J.I., Tian, D., Hernandez, G., Patton, C., Mood, B., Shrimpton, T., Butler, K.R., Traynor, P.: A hybrid approach to secure function evaluation using sgx. In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. pp. 100–113 (2019)
5. Dan, G., Jim, S.: 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
6. Felsen, S., Kiss, Á., Schneider, T., Weinert, C.: Secure and private function evaluation with intel sgx. In: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop. pp. 165–181 (2019)
7. Gupta, D., Mood, B., Feigenbaum, J., Butler, K., Traynor, P.: Using intel software guard extensions for efficient two-party secure function evaluation. In: International Conference on Financial Cryptography and Data Security. pp. 302–318. Springer (2016)
8. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Annual International Cryptology Conference. pp. 145–161. Springer (2003)
9. Järvinen, K., Kolesnikov, V., Sadeghi, A.R., Schneider, T.: Embedded sfe: Offloading server and network using hardware tokens. In: International Conference on Financial Cryptography and Data Security. pp. 207–221. Springer (2010)
10. Järvinen, K., Kolesnikov, V., Sadeghi, A.R., Schneider, T.: Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 383–397. Springer (2010)
11. Johnson, S., Scarlata, V., Rozas, C., Brickell, E., Mckeen, F.: Intel® software guard extensions: Epid provisioning and attestation services. White Paper **1**(1-10), 119 (2016)
12. Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: Naor, M. (ed.) Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4515, pp. 115–128. Springer (2007). https://doi.org/10.1007/978-3-540-72540-4_7, https://doi.org/10.1007/978-3-540-72540-4_7
13. Kolesnikov, V.: Truly efficient string oblivious transfer using resettable tamper-proof tokens. In: Theory of Cryptography Conference. pp. 327–342. Springer (2010)
14. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free xor gates and applications. In: International Colloquium on Automata, Languages, and Programming. pp. 486–498. Springer (2008)
15. Mohassel, P., Orobets, O., Riva, B.: Efficient server-aided 2pc for mobile phones. Proceedings on Privacy Enhancing Technologies **2016**(2), 82–99 (2016)
16. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 591–602 (2015)

17. Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: Proceedings of the 1st ACM conference on Electronic commerce. pp. 129–139 (1999)
18. Pass, R., Shi, E., Tramer, F.: Formal abstractions for attested execution secure processors. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 260–289. Springer (2017)
19. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: International conference on the theory and application of cryptology and information security. pp. 250–267. Springer (2009)
20. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit (2016), https://github.com/emp-toolkit/ Accessed January 5th, 2021
21. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 21–37 (2017)
22. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 220–250. Springer (2015)

A Appendix

A.1 Security Proof of Our Main Theorems

Due to space limitation, we only provide the security proof for malicious setting.

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 $\text{EXEC}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]}_{\Pi^{\mathsf{GC}}_{\mathsf{2pc}},\mathcal{A},\mathcal{Z}}$ where the parties $\mathcal{P} := \{P_1, P_2\}$ run protocol $\Pi^{\mathsf{GC}}_{\mathsf{2pc}}$ in the $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$-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 $\text{EXEC}_{\mathcal{F}^f_{\mathsf{2pc}},\mathcal{S},\mathcal{Z}}$ 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: $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $(\text{ComputeNotify}, \mathsf{sid}, |x_2|, P_2)$ for an honest party P_2 from the external $\mathcal{F}^f_{\mathsf{2pc}}$, the simulator $\mathcal{S}$ picks random $x^0_{2,i} \leftarrow \{0,1\}$, for $i \in [n_2]$, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x^0_{2,i}\}_{i\in[n_2]}\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ on behave of P_2.
- Upon receiving $(\text{ComputeNotify}, \mathsf{sid}, |x_1|, P_1)$ for an honest party P_1 from the external $\mathcal{F}^f_{\mathsf{2pc}}$, the simulator $\mathcal{S}$ picks random $k \leftarrow \{0,1\}^\lambda$, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ on behave of P_1. $\mathcal{S}$ then generate $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f^*; k)$ and parse $e = \{(X^0_i, X^1_i)\}_{i\in[n^*]}$. Subsequently, for $i \in [n^*_2]$, $\mathcal{S}$ sets $\sigma^0_i := H(X^0_{i+n_1})$ and $\sigma^1_i := H(X^1_{i+n_1})$, and it sets $\tau = H(F, d, \{\sigma^0_i, \sigma^1_i\}_{i\in[n^*_2]})$. $\mathcal{S}$ then sends τ to the simulated party 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 $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$, $\mathcal{S}$ acts as $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ to send $(\textsc{RunNotify}, \mathsf{sid}, Q_i, P_i)$ to $\mathcal{A}$. $\mathcal{S}$ then simulates the $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ functionality as defined.
- When the simulated party P_2 receives $(\hat{F}, \hat{d}, \{X^{x^0_{2,i}}_{i+n_1}\}_{i\in[n_2]}, \{\hat{\sigma}^0_i, \hat{\sigma}^1_i\}_{i\in[n^*_2]})$ from $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ and receives τ from the simulated P_1, P_2 computes $\hat{\tau} = H(\hat{F}, \hat{d}, \{\hat{\sigma}^0_{0,i}, \hat{\sigma}^1_{0,i}\}_{i\in[n^*_2]})$ and asserts $\hat{\tau} = \tau$. Thereafter, $\mathcal{S}$ fetches the internal GC label information (F, e, d) from the simulated P_1. For $i \in [n_2]$, $\mathcal{S}$ acts as P_2 to assert $Z_{i+n_1} = X^{x^0_{2,i}}_{i+n_1}$.
- 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 $\mathcal{F}_{\text{SC}}$, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\textsc{exec}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]}_{\Pi^{\mathsf{GC}}_{\mathsf{2pc}},\mathcal{A},\mathcal{Z}}$ and $\textsc{exec}_{\mathcal{F}^f_{\mathsf{2pc}},\mathcal{S},\mathcal{Z}}$ are identical except the scenario where the real-world output y is different from the ideal-world output y'. This happens when the malicious $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ provides inconsistent information, 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 $\textsc{exec}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]}_{\Pi^{\mathsf{GC}}_{\mathsf{2pc}},\mathcal{A},\mathcal{Z}}$ and $\textsc{exec}_{\mathcal{F}^f_{\mathsf{2pc}},\mathcal{S},\mathcal{Z}}$ are indistinguishable.

Case 2: P_1 is corrupted; P_2 and $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $\mathcal{F}^f_{\mathsf{2pc}}$, the simulator $\mathcal{S}$ picks random $x^0_{2,i} \leftarrow \{0,1\}$, for $i \in [n_2]$, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x^0_{2,i}\}_{i\in[n_2]}\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ on behave of P_2. For $i \in [n_2]$, $\mathcal{S}$ sends random $\hat{x}^1_{2,i} \leftarrow \{0,1\}$ to P_1 on behave of P_2.
- Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x^0_{2,i}\}_{i\in[n_2]}\rangle)$ from P_2, $\mathcal{S}$ acts as $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ to set $f^*(x_1, (x^0_2, x^1_2)) = f_1(x_1, x^0_2 \oplus x^1_2)$ and generate the garbled circuit by $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f^*; k)$. $\mathcal{S}$ then parse $e = \{(X^0_i, X^1_i)\}_{i\in[n_1+2n_2]}$ and sends $(F, d, \{X^{x^0_{2,i}}_{i+n_1}\}_{i\in[n_2]}, \{\sigma^0_i, \sigma^1_i\}_{i\in[n^*_2]})$ to the simulated party P_2 on behave of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$.
- When the simulated party P_2 receives $\{Z_i\}_{i\in[n_1]}$, $\{Z_{i+n_1+n_2}\}_{i\in[n_2]}$ and τ from P_1, $\mathcal{S}$ acts as P_2 to compute $\hat{\tau} = H(F, d, \{\sigma^0_i, \sigma^1_i\}_{i\in[n^*_2]})$ and assert $\hat{\tau} = \tau$. Thereafter, $\mathcal{S}$ fetches the internal GC label information (F, e, d) from the simulated $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$. For $i \in [n_2]$, $\mathcal{S}$ acts as P_2 to assert $Z_{i+n_1+n_2} = X^{x^1_{2,i}}_{i+n_1+n_2}$. In addition, $\mathcal{S}$ uses the internal GC label information (F, e, d) and $\{Z_i\}_{i\in[n_1]}$ to extract P_1's input x^*_1, and it sends $(\textsc{Compute}, \mathsf{sid}, x^*_1)$ to the external $\mathcal{F}^f_{\mathsf{2pc}}$ 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 $\mathcal{H}_0, \ldots, \mathcal{H}_2$.

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

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, P_2 sends random $\{\hat{x}^1_{2,i}\}_{i\in[n_2]}$ to P_1, instead of $\{x^1_{2,i} := x^0_{2,i} \oplus x_{2,i}\}_{i\in[n_2]}$.

Claim. $\mathcal{H}_1$ and $\mathcal{H}_0$ are perfectly indistinguishable.

Proof. Since $\{x^0_{2,i}\}_{i\in[n_2]}$ are random bits picked by P_2, the distribution of $\{\hat{x}^1_{2,i}\}_{i\in[n_2]}$ and $\{x^1_{2,i}\}_{i\in[n_2]}$ are identical. Therefore, $\mathcal{H}_1$ and $\mathcal{H}_0$ are perfectly indistinguishable.

Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, P_2 fetches the internal GC label information (F, e, d) from the simulated $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and it checks if $Z_{i+n_1+n_2} = X^{x^1_{2,i}}_{i+n_1+n_2}$; otherwise, $\mathcal{S}$ aborts.

Claim. If H is a collision resistant cryptographic hash function, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable.

Proof. The difference between $\mathcal{H}_1$ and $\mathcal{H}_2$ is that in $\mathcal{H}_1$, P_2 only checks $H(Z_{i+n_1+n_2})$; whereas, in $\mathcal{H}_2$, P_2 directly checks if $Z_{i+n_1+n_2} = X^{x^1_{2,i}}_{i+n_1+n_2}$. It is easy to see when H is a collision resistant cryptographic hash function, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable.

The adversary's view of $\mathcal{H}_2$ is identical to the simulated view $\textsc{exec}_{\mathcal{F}^{f}_{\mathsf{2pc}},\mathcal{S},\mathcal{Z}}$. Therefore, it is perfectly indistinguishable.

Case 3: P_2 is corrupted; P_1 and $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ 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 $\mathcal{F}^{f}_{\mathsf{2pc}}$ and receiving $\{x^1_{2,i}\}_{i\in[n_2]}$ from P_2, the simulator $\mathcal{S}$ picks random $k \leftarrow \{0,1\}^\lambda$, and it sends $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ to $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ on behave of P_1.
– Upon receiving $(\mathsf{Run}, \mathsf{sid}, \langle k, f\rangle)$ from P_1 and $(\mathsf{Run}, \mathsf{sid}, \langle f, \{x^0_{2,i}\}_{i\in[n_2]}\rangle)$ from P_2, $\mathcal{S}$ computes P_2's input $x^*_{2,i} := x^0_{2,i} \oplus x^1_{2,i}$, for $i \in [n_2]$. After that, it sends $(\textsc{Compute}, \mathsf{sid}, x^*_2)$ to the external $\mathcal{F}^{f}_{\mathsf{2pc}}$ on behave of P_2.
– Upon receiving $(\textsc{Compute}, \mathsf{sid}, y)$ from the external $\mathcal{F}^{f}_{\mathsf{2pc}}$ for P_2, the simulator $\mathcal{S}$ sets $f^*(x_1, (x^0_2, x^1_2)) = f_1(x_1, x^0_2 \oplus x^1_2)$ and uses the GC simulator to generate $(F', X', d') \leftarrow \mathsf{Sim}(1^\lambda, y, \Phi(f^*))$. $\mathcal{S}$ then uses X' as the wire labels to generate $\{Z_i\}_{i\in[n_1+2n_2]}$ as $Z_i := X'_i$. $\mathcal{S}$ picks $2n_2$ random numbers $\hat{Z}_i \leftarrow \{0,1\}^\lambda$. For

$i \in [n_2]$, $\mathcal{S}$ sets $\sigma_i^{x^0_{2,i}} := H(Z_{i+n_1})$, $\sigma_i^{x^0_{2,i}\oplus 1} := H(\hat{Z}_i)$, $\sigma_{i+n_2}^{x^1_{2,i}} := H(Z_{i+n_1+n_2}$ and $\sigma_{i+n_2}^{x^1_{2,i}\oplus 1} := H(\hat{Z}_{i+n_2})$. Subsequently, $\mathcal{S}$ sets $\tau = H(F', d', \{\sigma_i^0, \sigma_i^1\}_{i\in[n_2^*]})$. At last, $\mathcal{S}$ sends $\{Z_{i+n_1}\}_{i\in[n_2]}$ as the wire label of x_2^0, (F', d') as the GC tables and decode information and $\{\sigma_i^0, \sigma_i^1\}_{i\in[n_2^*]}$ as the hash values of P_2's wire labels to P_2 on behave of $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$, and it sends $\{Z_i\}_{i\in[n_1]}$, $\{Z_{i+n_1+n_2}\}_{i\in[n_2]}$ and τ to P_2 on behave of P_1.

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 $\text{EXEC}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]}_{\Pi^{\mathsf{GC}}_{\text{2pc}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that $\mathcal{H}_1$ generates different hash values by $\sigma_i^{x^0_{2,i}\oplus 1} := H(\hat{Z}_i)$ and $\sigma_{i+n_2}^{x^1_{2,i}\oplus 1} := H(\hat{Z}_{i+n_2})$, for $i \in [n_2]$, where $\{\hat{Z}_i\}_{i\in[2n_2]}$ are random values.

Claim. If H is a collision resistant cryptographic hash function, $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable.

Proof. The difference between $\mathcal{H}_0$ and $\mathcal{H}_1$ is that in $\mathcal{H}_0$, $\sigma_i^{x^0_{2,i}\oplus 1} := H(X_{i+n_1}^{x^0_{2,i}\oplus 1})$ and $\sigma_{i+n_2}^{x^1_{2,i}\oplus 1} := H(X_{i+n_1+n_2}^{x^1_{2,i}\oplus 1})$; whereas, in $\mathcal{H}_1$, $\sigma_i^{x^0_{2,i}\oplus 1} := H(\hat{Z}_i)$ and $\sigma_{i+n_2}^{x^1_{2,i}\oplus 1} := H(\hat{Z}_{i+n_2})$. It is easy to see when H is a collision resistant cryptographic hash function, $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable.

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 $\{Z_i\}_{i\in[n_1+2n_2]}$. $\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]$ also sends (F', d') as the GC tables and decoding information to P_2.

Claim. If GC is simulatable private with adversarial distinguishing advantage $\mathsf{Adv}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}_{\mathsf{GC}}(\mathcal{A}, \lambda)$, then $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable with distinguishing advantage $\mathsf{Adv}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}_{\mathsf{GC}}(\mathcal{A}, \lambda)$.

Proof. 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 except for the adversarial distinguishing advantage $\mathsf{Adv}^{\mathsf{prv.sim},\Phi,\mathsf{Sim}}_{\mathsf{GC}}(\mathcal{A}, \lambda)$.

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, if GC is simulatable private, the views of $\mathcal{A}$ and $\mathcal{Z}$ in $\text{EXEC}^{\mathcal{F}_{\text{HW}}[\mathsf{M}^{\mathsf{GC}}]}_{\Pi^{\mathsf{GC}}_{\text{2pc}},\mathcal{A},\mathcal{Z}}$ and $\text{EXEC}_{\mathcal{F}^f_{\text{2pc}},\mathcal{S},\mathcal{Z}}$ are indistinguishable with distinguishing advantage

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