IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2026, No. 2, pp. 73–105. DOI:10.46586/tches.v2026.i2.73-105

Loricae: Upgrading and Optimizing Multi-Party Computation Protocols with Filmy Hardware Enclaves

Xiang Li[1,2], Baiting Jiang[2], Xiaoyu Fan[2], Weijie Liu[4], Yifan Song[2] and Mingyu Gao[2,3]

[1] China Telecom eSurfing Cloud (State Cloud), Beijing, China, lix333@chinatelecom.cn
[2] Tsinghua University, Beijing, China, {jbt24,fxy23}@mails.tsinghua.edu.cn, yfsong@mail.tsinghua.edu.cn, gaomy@tsinghua.edu.cn
[3] Shanghai Qi Zhi Institute, Shanghai, China
[4] Nankai University, Tianjin, China, weijieliu@nankai.edu.cn

Abstract. Privacy-preserving computing techniques have been recently developed in two complementary directions. Cryptography-based multi-party computation (MPC) is provably secure but not performant, while hardware-based trusted execution environments (TEEs) trade security provability for high performance. Although some previous works have combined MPC and TEEs, they obscure a key question: *If TEEs are fully trusted, what is the necessity of such combination which simply hurts performance? Or, if TEEs are not trusted, to what extent are they untrusted?* We answer this question by formalizing a filmy enclave model to capture practical TEE features, where only the integrity of TEE content and the confidentiality of cryptographic procedures are guaranteed. With such TEEs, we can upgrade semi-honest MPC protocols to defend against malicious attackers. Data confidentiality is ensured through MPC protocols, while execution integrity is guaranteed by the code integrity within TEEs and our novel security designs. These designs safeguard all external inputs, including pre-generated correlated random numbers for MPC protocols and enclave state checkpoints for fault-tolerant execution. Besides protocol upgrade, we show filmy enclave models can also optimize communication for semi-honest protocols. Our design exhibits significant performance advantages over existing actively secure protocols.

Keywords: multi-party computation · trusted execution environment · enclave model formalization · secret sharing protocol · crash recovery

1 Introduction

The seminal paradigm of secure multi-party computation (MPC) was first established by Yao [Yao82]. Since then, it has drawn much attention from both academia and industry and become one of the mainstream paradigms to achieve privacy-preserving data processing [Yao82, Yao86, KS08, PSSW09, ZRE15, RR21, Sha79, CCD88, AFL+16, ABF+17, BOGW88, LX19, PS20, GSZ20, KPPS21, FCW+25, FCY+25]. MPC allows multiple parties to collaboratively derive a result without revealing their own inputs to others. Given different threat models, there is a large design space for MPC protocols [PGFW14]. The two most commonly assumed adversarial scenarios are semi-honest (a.k.a., passive) and malicious (a.k.a., active), where the former always sticks to the protocol but tries to steal secret data, and the latter could do arbitrary actions to compromise security. Protocols defending against malicious adversaries may incur much more overheads (e.g., as high as

Received: 2025-10-15 Accepted: 2025-12-15 Published: 2026-04-23

16×) than passively secure ones [Kel20a]. Therefore, although malicious adversaries are a more practical assumption, most real-world applications only adopt passively secure MPC protocols [FWC+21, RRG+21, RBS+22, HLC+22, GJM+23, JGB+23, LKFV23].

Another prevalent paradigm for privacy-preserving computing is to leverage trusted execution environments (TEEs) provided by the underlying hardware processors [Int18, Int20, Alv04, ARM21, Kap17], which enables high-performance processing that approaches the speed of insecure execution while offering confidentiality and integrity enhancements inside hardware provided enclaves [ATG+16, LLP+19, STC+20, EHZH+22, WLL+22, LLG23]. However, various TEE-related side channels are disclosed [LSG+17, XCP15, VBWK+17, GESM17, SWG+17, VBMW+18, CCX+19, MRR+15, ZS18, GMO01] and are shown to seriously compromise the confidentiality guarantee. Although software-level protections can be incorporated [DSC+15, ZDB+17, SLKP17], comprehensive elimination of side-channel vulnerabilities seems impossible, considering the trends of increasingly complicated processor microarchitectures and enormous stealthy physical channels.

Consequently, there are efforts *combining MPC protocols and TEEs* [BBB+17, FKSW19, CTH+19, LZZ+21, WNS+22, DW23]. However, in their "combination", TEEs are either alternated with MPC based on different security requirements for different computation phases, or act as a trusted third party to *implement* MPC. Instead, we *tightly* integrate TEEs into MPC protocols to design new protocols, achieving the best of both worlds, with provable security against malicious adversaries *and* high performance.

The first challenge of designing the integration and proving its security lies in accurately determining the TEE security boundary. Some previous work [PST17] overestimated the TEE's capabilities by ignoring side-channel vulnerabilities; others undervalued the TEE's strength as guaranteeing only integrity [TZL+17], thus missing out on opportunities to leverage TEEs in boosting MPC performance. To properly capture the practical behaviors of TEEs, we *formally* propose a *filmy enclave model*, striking a more practical intermediate position between the two extreme models. Concretely, compared with a fully secure enclave with both confidentiality and integrity protection, in a filmy enclave, *all user programs except for cryptographic operations may reveal states* to attackers. Here we leverage the opportunity that cryptographic libraries are usually well-examined, less vulnerable to side channels. The formalization of filmy enclaves provides the theoretical foundation for using TEEs to enhance MPC protocols. Specifically, due to the lack of confidentiality for general data, filmy enclaves cannot directly process plaintext data, and must use encrypted data formats such as secret-sharing MPC protocols.

On the other hand, the integrity guarantee of filmy enclaves seems to be able to prevent adversaries from deviating from the protocol. Unfortunately, similar to the conclusion that TEEs cannot directly upgrade crash fault-tolerant (CFT) coordination protocols to Byzantine fault-tolerant (BFT) [WDN+22], we observe that putting passively secure MPC protocols into enclaves cannot achieve active security against malicious adversaries. TEEs can only protect in-enclave processing and inter-enclave communication, but *as long as the adversary in the untrusted domain has opportunities to provide external inputs for enclaves, it has the potential to compromise integrity.* In practical system implementations, such external inputs broadly exist. The pre-generated correlated random numbers required by the protocol need to be dynamically loaded by the enclave as they are progressively consumed during execution. Besides, real-world MPC applications (e.g., machine learning [MZ17, DEK20, ZDC+21, WWP22]) may take quite a long time, during which the enclave or the underlying computer node may crash intentionally (i.e., maliciously) or unexpectedly. Simply recomputing from the beginning would incur unacceptable costs, thus appealing for fast crash recovery, which requires saving the intermediate states (a.k.a., a checkpoint) and then loading the states into the renewed enclave. Many enclave applications are designed to support checkpoints [PLD+11, MAK+17, BCLK17, KPW+19, ZCP+21]. As a result, the checkpoint states become another source of external input to the enclaves.

To eliminate external input attacks, we design, formalize, and prove a generic paradigm in the generalized universal composition (GUC) model, in order to upgrade MPC protocols to active security under the practical filmy enclave assumption.

Apart from upgrading the security of MPC protocols, we further use the filmy enclave model to optimize the communication performance of passively secure MPC protocols by concentrating on essential primitives such as oblivious transfer (OT) and oblivious product evaluation (OPE). Our key insight is that, given the enclave's integrity protection, we can ensure that a party can only decrypt the values designated by the protocol, but no more than these allowed ones. Consequently, we reduce the communication cost of semi-homomorphic-encryption-based OPE into a single round, and render random OT almost devoid of communication overheads.

We combine the above protocol upgrades and optimizations into a new TEE-enhanced, actively-secure MPC design named Loricae. We implement Loricae into the MP-SPDZ framework [Kel20a] to fairly compare with existing passively-secure and actively-secure MPC protocols. Our results demonstrate that with our security upgrade paradigm, Loricae is 1.2× to 71.4× and 1.2× to 32.0× faster than state-of-the-art MPC protocols in the LAN and WAN settings, respectively, while both achieving active security. The offline phase can be further accelerated by 1.1× to 2.3× using our performance optimizations.

To summarize, we make the following contributions:

- We formalize the filmy enclave model to capture a more natural usage of TEEs with more practical security guarantees than previous TEE models.
- We find that simply incorporating TEEs into passively-secure MPC protocols does not directly result in actively-secure designs, and illustrate the corresponding attacks.
- We design a novel framework that uses filmy enclaves to upgrade passively secure MPC protocols to be resistant to malicious adversaries, with formal security proofs.
- We describe several optimizations for the communication performance of passively secure MPC protocols under the filmy enclave model.
- We implement and evaluate the above designs as a new MPC system named Loricae, and show it achieves significant speedups over previous designs. We open source it at https://github.com/tsinghua-ideal/loricae.

2 Preliminaries

2.1 Secure Multi-Party Computation

Many multi-party computation (MPC) protocols have been designed based on garbled circuits [Yao82] or secret sharing [Sha79]. They are used to evaluate a function represented as a *circuit*, comprising a collection of *gates* connected through *wires*. The gates represent operations including addition, multiplication, comparison, and so on, while the wires denote dependencies among gates. Threat models mainly fall into two categories based on adversary behaviors. *Semi-honest* adversaries are curious to learn information about other parties' inputs while strictly following the instructions of the protocol. In contrast, *malicious* adversaries are more powerful and are allowed to arbitrarily deviate from the protocol in any ways [Can00, Lin20]. MPC protocols defending against malicious adversaries typically incur several times more overheads than those only tolerating semi-honest adversaries [KOS16, Kel20a, AFO+21, FCW+22]. For example, the actively-secure version of [DEF+19] has 16× cost than the passively-secure version [Kel20a].

The security of MPC protocols among n participating parties holds until the number of corrupted parties exceeds a threshold t. For example, protocols based on degree-t Shamir secret sharing [Sha79] (denoted as $[\cdot]_t$) tolerate $t < n/2$ corrupted parties (i.e., the *honest majority* setting). A well-known protocol of such is BGW [GBOW88, BOGW88]. Suppose each party $P_i, i \in \{0, \ldots, n-1\}$ holds a share x_i of $[x]_t$ and y_i of $[y]_t$. For the addition

operation, each party locally computes $z_i = x_i + y_i$. For multiplication, each party first locally computes $z_i = x_i \cdot y_i$, where z_i is a share of $[z]_{2t}$ instead of $[z]_t$. Then each party shares z_i as $[z_i]_t$. Finally, all parties locally compute $[z]_t = c_1[z_1]_t + \cdots + c_n[z_n]_t$ where $\{c_i\}_{i\in\{0,\ldots,n-1\}}$ are predetermined Lagrange coefficients. On the other hand, protocols based on additive secret sharing [DPSZ12] (denoted as $[\cdot]$) allow at most $t = n - 1$ corrupted parties (i.e., the *dishonest majority* setting) [Can00]. For example, the Semi protocol [Kel20a] removes all steps required for malicious security from MASCOT [KOS16] (optimized version of SPDZ [DPSZ12]). Its processing relies on *offline* pre-generated correlated randomness, such as Beaver triples [Bea92]. More specifically, offline Beaver triple generation requires oblivious product evaluation (OPE), where a party holding x and the other party holding y can cooperatively compute $z = x \cdot y$, and each party possesses a share of z, without knowing the other's secret [Gil99, KOS16]. OPE can be realized by other cryptographic primitives, such as oblivious transfer (OT) and additive homomorphic encryption (AHE) [KOS16, KPR18]. For the random OT variant where the input messages are randomly sampled [CO15], we formalize it as $\mathcal{F}_{\text{ROT}}^{1,k} : (\perp, b) \mapsto ((r_0, r_1), r_b)$.

As malicious behaviors affect the availability of correct protocol outputs, the output guarantees for honest parties have different levels. A common guarantee is *security with abort*: if an honest party detects malicious behaviors, it may simply abort the protocol.

2.2 Trusted Execution Environments

Trusted execution environments (TEEs), such as Intel SGX [Int18], Intel TDX [Int20], ARM TrustZone [Alv04], ARM CCA [ARM21], and AMD SEV [Kap17], create a potentially more efficient avenue for achieving data security, using hardware-based protection to avoid excessive cryptographic computation cost. They split the execution environment on a hardware processor into a trusted domain (we refer to it as *enclave* in this paper) and an untrusted domain. The hardware guarantees the *isolation* between the two domains.

Apart from hardware-based isolation, *attestation* and *sealing* are two common features provided by TEEs. Consider a case where there is a client and a server. The client wants to outsource the computation on her secret data to the untrusted server, so she creates an enclave on the server. Since the enclave is under the control of the untrusted server, the untrusted server could load arbitrary code into the enclave, such as computing a wrong result, or even directly dumping the secret data out of the enclave. Therefore, attestation [AGJS13] is needed to ensure the code legality in the enclave. During the attestation, a private and authenticated channel is built between the client and the enclave.

Since all the states in the enclave are volatile, they need to be persisted in the non-volatile storage (e.g., disks) before the enclave closes. However, the non-volatile storage is in the untrusted domain, therefore the to-be-persisted data should be manually encrypted and authenticated with an application key. Given that the application key also belongs to the volatile states, storing the key requires another key. To solve this recursive issue, the enclave can generate an enclave-platform-specific sealing key by hardware [AGJS13]. This key is used as a root key to protect the volatile states inside the enclave (including the application key). The sealing key is determined by the enclave identity and the platform, so the hardware can re-generate the same key to decrypt previously sealed states, but malicious code cannot generate the same sealing key.

However, existing TEEs are not a panacea. Their security has been questioned due to various *side channels*, including access-pattern-based attacks [LSG+17, XCP15, VBWK+17, GESM17, SWG+17], transient execution attacks [VBMW+18, CCX+19], and power/electromagnetic signal analysis [MRR+15, ZS18, GMO01]. Although additional programming efforts and/or software-level protections can be incorporated [DSC+15, ZDB+17, SLKP17], it seems impossible to fundamentally solve the side channel issues. As a result, in this work we recognize the existence of side channels and model such non-ideal phenomena in TEEs to enhance security (Section 3).

3 Filmy Enclave Model

To capture the effect of side channels and give a precise description about the security guarantee provided by TEEs, we should first formalize the model of TEEs. Previously, [PST17] formalized TEEs to have a trusted clock and protect confidentiality and integrity simultaneously, completely ignoring side channels. On the other hand, [TZL+17] proposed *transparent enclave*, based on the observation that various side channels mainly compromise the confidentiality guarantee of TEEs. So this model assumes maximum leakage — all the in-enclave program states, except the states for attestation (e.g., master signing key), can be leaked to the untrusted domain. Thus the enclave only guarantees integrity.

However, we find that security for the attestation procedure (assumed in the transparent model) almost implies security for other cryptographic procedures. In practice, the attestation is also implemented in a software module [Cos16, BWS19], which is not fundamentally different from other in-enclave cryptographic procedures. Besides, the same module involves other cryptographic keys. For example, Intel platforms persist the attestation key outside enclaves with a sealing key [Cos16]. The sealing key must also be secure if security is assumed for the attestation key. Therefore, we propose the *filmy enclave model*, striking a more practical intermediate position between the fully-trusted enclave model [PST17] and the transparent enclave model [TZL+17]. The key observation is that *cryptographic keys are typically better protected than data considering side channels*, justified by the following reasons: (1) Mature cryptographic libraries used in enclave SDKs [Int18] are developed and audited by experts and possess good properties, such as constant-time and oblivious execution [BCS13, BBC+14, JXJ+22]; (2) With careful library implementations, there may still be software-exploitable attacks like transient execution attacks [CCX+19, Int22] and CPU architectural vulnerabilities [BKS+22]. Those attacks can be mitigated by meticulously placing memory barriers after security checks to thwart speculative execution, by avoiding caching keys through `movnti` instructions (x86) and control register settings (ARM), and by others [HGS25]. However, it is generally difficult to apply those techniques to arbitrary user programs; (3) Other hardware mechanisms [ML05, BGG+13, NCR+23] or dedicated co-processors [Cai19, FG13] further enhance cryptographic procedures. For example, Intel TME-MK engine maintains a key table by hardware [Int17, ACE+23].

Hence, we model that *all user programs except for cryptographic operations may reveal states to the untrusted domain, while the integrity of all states inside enclaves is preserved.* For existing TEEs, we need to incorporate the aforementioned techniques to approximate the model as closely as possible, summarized as the following advice for future TEEs.

Advice for future TEEs. To fully realize the filmy enclave model, especially against side-channel attacks (SCA), hardware designers need to carefully harden cryptographic operations inside TEEs, e.g., through the use of SCA-resistant co-processors and register-based key pinning during cryptographic execution. Besides, designers also need to enhance in-TEE integrity guarantees by, e.g., redundant computations or validation metadata.

Model formalization. We adopt the universal composition (UC) paradigm [Can01, CR03], specifically the generalized UC (GUC) model [CDPW07], to precisely describe the filmy enclave model $\mathcal{G}_{\text{TEE}}$ as summarized in Functionality 1. Therefore, $\mathcal{G}_{\text{TEE}}$ is a globally shared functionality, which the environment $\mathcal{Z}$ directly interacts with. If a $\mathcal{G}_{\text{TEE}}$-hybrid protocol GUC-realizes some functionality, any other programs executing simultaneously will not affect its security. We defer the UC model details to the supplementary material.

The hardware manufacturer M would maintain the set of processors equipped with TEEs in a registry reg, i.e., for a processor P with TEE support, we denote $P \in \mathsf{reg}$. Besides, during the manufacturing process, a signing key pair denoted by $(\mathsf{msk}, \mathsf{mpk})$ is initialized for the signature scheme Σ, and the private key msk is never exposed outside the enclave. The attestation report will be signed using msk, while the challenger uses the public key mpk to verify the report. $\mathcal{G}_{\text{TEE}}$ captures the anonymous attestation

Functionality 1 $\mathcal{G}_{\text{TEE}}[\Sigma, \mathsf{reg}, M]$

Init: $(\mathsf{mpk}, \mathsf{msk}) := \Sigma.\mathsf{KGen}(1^\lambda)$, ▷ *signing key pair*
$T = \varnothing$, ▷ *abstract mapping of eid, P → prog, mem*
$R = \varnothing$ ▷ *secure communication channel info*

On receive* install(prog) from some $P \in \mathsf{reg}$:
1: $eid \leftarrow\!\$\ \{0,1\}^{\lambda'}$, store $T[eid, P] := (\mathsf{prog}, \vec{0})$
2: **return** eid ▷ *λ' is significantly larger than λ*

On receive* attest(eid, uid) from some P' for some $P \in \mathsf{reg}$:
3: $(\mathsf{prog}, \_) := T[eid, P]$; $y \leftarrow\!\$\ \mathbb{Z}_p$ ▷ *modulus p and base g are predetermined*
4: $R[eid, P, uid] := (y, \bot, \bot)$ ▷ *set the private key of entity eid in R*
5: $\sigma := \Sigma.\mathsf{Sig}_{\mathsf{msk}}(eid, \mathsf{prog}, g^y, uid)$
6: **return** g^y, σ

On receive* verify($eid, eid', \mathsf{prog}', g^x, \sigma'$) from some $P \in \mathsf{reg}$:
7: assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((eid', \mathsf{prog}', g^x, eid), \sigma')$
8: $y, \mathsf{pk}_0, \_ := R[eid, P, eid']$; assert $\mathsf{pk}_0 = \bot$ ▷ *no overwriting*
9: $R[eid, P, eid'] := (y, g^x, \sigma')$ ▷ *set the public key of entity eid'*

On receive* recvpk(eid, uid, g^x) from some $P \in \mathsf{reg}$:
10: $y, \mathsf{pk}_0, \_ := R[eid, P, uid]$; assert $\mathsf{pk}_0 = \bot$ ▷ *no overwriting*
11: $R[eid, P, uid] := (y, g^x, \bot)$ ▷ *set the public key of entity uid*

On receive* resume(eid, inp) from some $P \in \mathsf{reg}$:
12: $(\mathsf{prog}, \mathsf{mem}) := T[eid, P]$, abort if not found; $(\mathsf{outp}, \mathsf{mem}) := \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$ ▷ *execute*
13: $T[eid, P] := (\mathsf{prog}, \mathsf{mem})$
14: **return** outp

On receive* destroy(eid) from some $P \in \mathsf{reg}$:
15: remove entry $T[eid, P]$; remove entry $R[eid, P, uid]$ for all uid

On receive* reveal(eid) from some $P \in \mathsf{reg}$:
16: $(\mathsf{prog}, \mathsf{mem}) := T[eid, P]$, abort if not found
17: **return** $\mathsf{prog}, \mathsf{mem}$

On receive* authenc(uid, x, b) from enclave eid on P:
18: $y, g^x, \_ := R[eid, P, uid]$; $\mathsf{sk} := g^{xy}$; if $b = 1$, $y := \mathsf{AE.Enc}_{\mathsf{sk}}(x)$; else, $y := \mathsf{AE.Dec}_{\mathsf{sk}}(x)$
19: **return** y

On receive* seal($salt, x, b$) from enclave eid on P:
20: $\mathsf{prog}, \_ := T[eid, P]$; $\mathsf{sk} := \mathsf{KDF}(M[P], \mathsf{prog} \| salt)$ ▷ *derive the enclave-specific key*
21: if $b = 1$, $y := \mathsf{AE.Enc}_{\mathsf{sk}}(x)$; else, $y := \mathsf{AE.Dec}_{\mathsf{sk}}(x)$
22: **return** y

On receive* randomgen(l) from enclave eid on P:
23: $r \leftarrow\!\$\ \{0,1\}^l$
24: **return** r

feature [AGJS13, MAB+13, SJBZ18] by using the sole key (msk) for signing all attestation reports of all TEE instances. The feature indicates that for certain code, the challenger in the attestation can only verify whether the enclave runs the exact code on a real hardware environment, but cannot distinguish different instances on the same processor or even instances on different processors. We use T as an abstract structure recording the mapping from an enclave identity eid on certain processor P to the loaded program prog and the memory content mem. R stores the information for establishing secure channels between enclaves. The content of R, which includes cryptographic key management, is specially protected in our model, and does not belong to mem. For concreteness and clear elaboration, $\mathcal{G}_{\text{TEE}}$ utilizes the Decisional Diffie-Hellman (DDH) assumption, authenticated encryption (AE) with INT-CTXT security, digital signature scheme (Σ) with UF-CMA

security, and key derivation function (KDF).

After a processor P invokes install to launch an enclave and install the program, an enclave ID *eid* is returned. All other routines relevant to the enclave should be invoked with the specific *eid*. Then, remote attestation is done to validate code legality. The routine attest proves the states in enclave *eid* to a remote entity *uid* with processor P' and starts the key negotiation. The remote entity may be another enclave (in case of mutual attestation) or a non-enclave entity (e.g., a remote client). In the former case, the remote enclave will also attest to the local enclave, which calls the routine verify to check whether the remote enclave loads the expected program before building the secure channel. In the latter case, the local enclave just calls recvpk to receive g^x from entity *uid* and builds a secure channel with it. Other key exchange schemes can also be used here. Note that P is used internally but not provided as function parameters, restricting the enclaves on one processor from accessing states on other processors. After attestation and verification, P can invoke resume to run a program inside the enclave. Note that the program may comprise several functions, each invocation executes one function by designating the function name as part of the input. Finally, the routine destroy will be invoked explicitly if the main program executes to the end, or implicitly if a crash occurs. It removes all the states related to the specific enclave.

To formalize the leakage from TEEs, we define a routine reveal to allow the untrusted domain to obtain in-enclave states of the user-defined program. All the variables in a program including input arguments, intermediate results, and outputs are exposed. Furthermore, apart from the routines called from outside the enclave, we model three more routines authenc, seal, and randomgen, which are internally invoked and executed inside the enclave. The first two routines securely use cryptographic keys and do not expose internal states by definition. The routine authenc represents the authenticated encryption and decryption, and the routine seal implements the sealing feature of TEEs (Section 2.2), which involves a key embedded inside the processor. This key is different for each processor, thus denoted by $M[P]$. The routine randomgen models the capability of the hardware random number generator inside the enclave. Note that all the routines in Functionality 1 should be called locally. For example, a remote party cannot use reveal to steal the information from the enclave on another node.

Functionality 2 Simplified routines in $\mathcal{G}_{\text{TEE}}[\Sigma, \mathsf{reg}, M]$

$E = \varnothing$ ▷ *newly added abstract mapping of* $P \to eids$

On receive* install(prog) from some $P \in \mathsf{reg}$:

1: $eid \leftarrow\$ \{0,1\}^{\lambda'}$; store $T[eid, P] := (\mathsf{prog}, \vec{0})$; put eid into $E[P]$
2: **return** eid

On receive* attest(eid, uid) from some P' for some $P \in \mathsf{reg}$:

3: $\mathsf{sk}_0 \leftarrow\$ \{0,1\}^{\lambda}$; $R[eid, uid] := (\text{true}, \mathsf{sk}_0)$
4: $(\mathsf{prog}, \_) := T[eid, P]$, $\sigma := \Sigma.\mathsf{Sig}_{\mathsf{msk}}(eid, \mathsf{prog}, uid)$ ▷ *implicit assert* $eid \in E[P]$ *through* T
5: **return** σ

On receive* verify($eid, eid', \mathsf{prog}', \sigma'$) from some $P \in \mathsf{reg}$:

6: assert $\Sigma.\mathsf{Vf}_{\mathsf{mpk}}((eid', \mathsf{prog}', eid), \sigma')$ and $eid \in E[P]$; $\_, \mathsf{sk}_1 := R[eid', eid]$
7: $(\text{needKey}, \mathsf{sk}_0) := R[eid, eid']$; assert needKey = true; $R[eid, eid'] := (\text{false}, \mathsf{sk}_0 + \mathsf{sk}_1)$

Necessity for incorporating cryptography into $\mathcal{G}_{\mathsf{TEE}}$ formalization. Functionality 1 seems cumbersome due to the used cryptography like the DDH assumption, AE, and Σ. Indeed, if we only consider establishing secure channels between enclaves, the formalization can be simplified. For example, Functionality 2 excludes P from the index key of R for succinct key exchange abstraction and avoids using the DDH assumption. However, this formalization cannot capture the case where an enclave needs to build a secure channel

with a remote non-enclave party, since the party without a trusted processor does not have direct access to $\mathcal{G}_{\text{TEE}}$ and any messages sent to $\mathcal{G}_{\text{TEE}}$ are delivered by the party owning the enclave. Cryptography should be used to protect the messages as the remote party interacts with $\mathcal{G}_{\text{TEE}}$. Similarly, we can also apply the same technique on structure T to eliminate the need for Σ, but it will become unable to capture the attestation to a non-enclave party. Previous work [PST17, TZL+17] also incorporated digital signature schemes into the TEE functionality. We additionally integrate DH key exchange into the functionality instead of the in-enclave program, which allows us to exhibit the protection of cryptographic keys. Therefore, the cryptography used in $\mathcal{G}_{\text{TEE}}$ is necessary for (1) generality and (2) clear exhibition of key protection.

Comparison with [DMMQ23]. We note that [TZL+17] roughly describes a similar model in Section 4.2 without formalization, while [DMMQ23] formalizes a similar model, which differs from ours as follows.

First, [DMMQ23] assumes all processors are equipped with TEEs, based on which the formalization of secure channel establishment is much simplified. Only one party invokes a key-exchange routine in her enclave and then both enclaves return the same key handler without invocation from the other party. However, it can neither capture the case where an enclave needs to build a secure channel with a remote non-enclave party (which necessitates explicit use of some cryptography), nor differentiate one-side attestation from mutual attestation. So our model is more general and self-contained.

Second, [PST17, TZL+17, DMMQ23] assume the existence of the globally determined and unique session ID [Can01] for install, requiring a trusted party to assign identifiers to each execution, which seems impractical [LLR06, GL05]. Our model avoids this assumption. In addition, our model strips out the attestation process to capture the natural usage of TEEs, instead of consolidating the attestation into the routine resume by accompanying each returned value with a signature [PST17, TZL+17, DMMQ23], which leads to technical problems in proofs [PST17].

Finally, we additionally model seal and randomgen, which are also important TEE features. Especially for crash recovery, enclaves should securely persist states with seal.

4 Protocol Upgrade with TEEs

In Section 3 we formally model TEEs considering side channels. Now we explore how such filmy enclaves should be utilized. With the filmy enclave model, TEEs cannot ensure confidentiality for program states. Therefore, for general computation, we cannot directly process plaintext data in the enclaves. Instead, we need to incorporate cryptographic methods, using MPC protocols in this paper, to compensate for confidentiality.

Combining MPC protocols with TEEs is not only necessary for TEEs under filmy enclave models, but also profitable for MPC protocols. Recall that MPC protocols can be categorized into passively secure ones and actively secure ones. To upgrade the former to the latter, previous designs mainly used zero-knowledge proofs (ZKP) automatically generated through the GMW compiler [GMW87], to force the honest behaviors of each party [IKOS07]. Fortunately, the same effect could be achieved by TEEs with lower overheads, since the integrity guarantee of TEEs "seems to be sufficient" to prevent a malicious adversary from deviating from the protocol and achieve active security. The prover only needs to send an attestation report from the TEE showing it runs the correct program, without sending sensitive information to the verifier for proof.

However, our deeper investigation discovers that this is not the case. The enclaves can only ensure integrity for the in-enclave execution. *The adversary can still exploit the opportunities of providing inputs for the enclaves from the untrusted domain to compromise integrity.* Besides the inter-enclave communication channels that are already protected

by authenticated encryption, we show that there remain potential attack points: (1) manipulating checkpointing states between crash and recovery and (2) changing offline pre-generated data. We first illustrate the threat model and design goals when combining TEEs and MPC protocols in Section 4.1. Next, we discuss the concrete attacks and our corresponding defenses in Sections 4.2 and 4.3. Then we integrate the solutions into a specific protocol Semi [Kel20a], formally summarizing the complete protocol in Section 4.4, and proving its security in Section 4.5.

4.1 Threat Model and Design Goals

Functionality 3 $\mathcal{F}_{\mathcal{C}}\left[\{P_i\}_{i\in\{0,\ldots,n-1\}}\right]$

1: $\mathcal{F}_{\mathcal{C}}$ receives $\{\mathsf{inp}_i\}_{i\in\{0,\ldots,n-1\}}$ from all parties $\{P_i\}_{i\in\{0,\ldots,n-1\}}$.
2: If all parties have sent inp_i data, then $\mathcal{F}_{\mathcal{C}}$ computes $\mathsf{outp} := \mathcal{C}(\mathsf{inp}_0, \mathsf{inp}_1, \cdots, \mathsf{inp}_{n-1})$. Otherwise, $\mathcal{F}_{\mathcal{C}}$ sends $\perp$ to all parties and returns.
3: $\mathcal{F}_{\mathcal{C}}$ sends outp to the ideal adversary $\mathcal{S}$.
4: For each party P_i, $\mathcal{F}_{\mathcal{C}}$ receives an instruction from the ideal adversary $\mathcal{S}$. If it is "proceed", $\mathcal{F}_{\mathcal{C}}$ sends outp to P_i. Otherwise, $\mathcal{F}_{\mathcal{C}}$ sends $\perp$ to P_i.

In this section, we consider a case where n parties jointly compute a function represented as a circuit $\mathcal{C}$, and each party controls one server equipped with a TEE, modeled by Functionality 1 in Section 3. A malicious adversary could potentially compromise $t < n$ parties. If a party is corrupted, the adversary controls the whole server, including all privileged software including the virtual machine manager (VMM) and operating system (OS). The adversary can also manipulate the network originating or terminated at corrupted parties, including eavesdropping, modifying, delaying, and dropping packets. However, we assume every message between honest parties is delivered within a bounded time. Instead of simply assuming a synchronous network between the enclaves, our assumption captures the fact that the network traffic between two enclaves goes through the underlying machines, which may be controlled by the adversary.

We aim to achieve the following goals: (1) The upgraded protocols achieve *security with abort* in the dishonest majority setting. (2) The implementation of the paradigm should be general, i.e., agnostic to specific protocols. (3) The upgrade paradigm should be efficient and practical considering real-world settings. In particular, we consider *crash recovery* support, an essential feature in modern cloud computing as computation tasks become increasingly complicated and long-running. We also consider practical hardware constraints, such as limited memory capacity.

We mainly discuss the dishonest majority setting, in which case the security goals can be formalized as Functionality 3. Nevertheless, our methods should be compatible with the honest majority setting.

4.2 Manipulated Checkpoints

The circuit evaluation phase in MPC seems secure if put into TEEs, since no input is provided from the untrusted domain. However, the commonly desired crash recovery feature would periodically save intermediate computation states (a.k.a., a checkpoint), which are then loaded into restarted enclaves. Supporting recovery opens up new opportunities to the adversary, who could arbitrarily cause a crash and manipulate the checkpoint as input to the (recovered) enclave.

Using a message authentication code (MAC) involving a secret cryptographic key helps detect the manipulation. However, the key should not be provided by the (corrupted) party owning the enclave, otherwise, the party can regenerate a MAC for the tampered value.

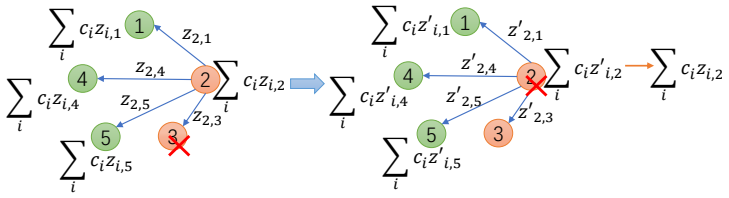


Figure 1: An attack on crash recovery in the BGW protocol. Green parties are honest and red parties are corrupted. For clarity, we only show the arrows related to the share distribution of party 2.

Therefore, *the key should be internally generated and hidden from the party owning the enclave, and the TEE sealing mechanism satisfies this requirement* [AGJS13]. Only when the specific code binding to the key is loaded into the enclave, can the enclave internally (re)generate the key (Section 2.2). Besides, for the freshness of dumped values against shuffle or rollback attacks, we need to *assign each intermediate wire value share in the MPC circuit with a unique identifier*. Specifically, the MAC binds both the value and the identifier. During crash recovery, the enclave loads the authenticated value and its identifier, checks the MAC, and compares the loaded identifier with an expected identifier. Expected identifiers cannot be internally generated, requiring negotiation with other enclaves according to the current execution progress. Negotiation among enclaves may lead to additional $\mathcal{O}(n)$ communication overhead per gate, where n is the number of parties. Therefore, besides uniqueness, identifiers should also be *locally derivable*.

We design the identifier to encompass the following information: session ID, enclave ID, and gate ID. A session includes a set of enclave instances, each belonging to one party. An enclave ID identifies an enclave instance, which means that even if two enclaves load the same program, their IDs are different. The gate ID uniquely refers to a gate in a circuit whose output is the corresponding wire value. Note that the parties compute only one circuit in a session, therefore there are no two identical gate IDs in a session. At the beginning of a session, all enclaves verify the consistency of the session ID as well as the next gate ID to be evaluated, and broadcast their enclave IDs to verify they are in the same set, thus agreeing on a committee. A committee is composed of enclave instances that are pairwise interconnected. No enclave instance connects to entities outside the committee. Therefore, one enclave instance never belongs to two committees simultaneously. No further agreement is needed until a crash occurs, and expected gate IDs are locally updated to match the progress. A restarted enclave has a new enclave ID, while the enclave ID in the expected identifier is stale and corresponds to the crashed enclave. Therefore, loading values requires negotiating the stale identifier.

A subtle point is that, at the time of evaluating gate g, an enclave may crash before its states are saved, while others have already finished checkpointing. To achieve liveness, the other parties should legally roll back to the previous gate $g-1$, and compute gate g along with the recovered enclave again. The allowance of legal rollback is unproblematic if the protocol always runs deterministically. Unfortunately, this feature can be exploited by the attacker if the protocol generates and consumes random numbers on the fly. Considering the multiplication procedure in the BGW protocol, each party P_i generates $[z_i]_t = \{z_{i,j}\}_{j=1}^n$ with randomness. Figure 1 shows an attack by utilizing legal rollback. As a gate completes, the corrupted party 3 intentionally crashes while the other party $j, j \neq 3$ derives and holds $\sum_i c_i z_{i,j}$. After party 3 recovers, the gate is computed again and at this time all other parties randomly generate and distribute a different set of shares. Then the corrupted party 2 crashes after it has done a checkpoint, while the other party $j, j \neq 2$ holds $\sum_i c_i z'_{i,j}$. As party 2 recovers, it has two different checkpoints that are both valid for the same gate. It may load the stale one $\sum_i c_i z_{i,j}$ instead of the correct one $\sum_i c_i z'_{i,j}$, causing inconsistent

shares with others and resulting in wrong results.

From the theoretical perspective, fixing the random number sequence at the beginning seems to be sufficient to solve the issue. However, it is generally inefficient and impractical to store the whole sequence of random numbers inside the limited enclave memory. Moreover, doing so still requires extra mechanisms of agreeing on and committing all parties' random number sequences, as well as making other specific modifications to the protocol implementation. Our key insight is that *once an enclave recovers from the crash, a new session ID should be negotiated among all enclaves.* Therefore, as an enclave restarts, all enclaves need to synchronize both the previous session ID and the new session ID. The former contributes to the expected identifier to load the checkpoint, while the latter is used to generate new checkpoints. In the previous attack, the two checkpoints of party 2 would have different session IDs, and only the newer one can be used.

4.3 Inconsistent Correlated Randomness

Then we consider another type of input data to enclaves, the pre-generated correlated randomness, such as the Beaver triples in the Semi protocol. We take Beaver triples as an example for further illustration, while the same idea also applies to other correlated randomness such as the ones used in the DN protocol [DN07] and in the bit decomposition protocol [DFK+06].

Given that the offline and online phases are separate in time, one party probably uses one enclave to run the preprocessing and another enclave for online circuit evaluation. Thus the generated randomness generally needs to be stored outside the enclave for being loaded later. Similar to the issues of manipulated checkpoints, solely using MAC cannot guarantee that all enclaves consume consistent shares of Beaver triples. For a specific gate u, an honest party P_i provides Beaver triple shares $(a_{u,i}, b_{u,i}, c_{u,i})$, while a corrupted party P_j may provide any triple shares that are inconsistent with the shares of honest parties, e.g. $(a_{v,j}, b_{v,j}, c_{v,j})$, $v \neq u$, leading to incorrect results.

A straightforward method is to attach a unique identifier to each generated Beaver triple. As each enclave consumes Beaver triple shares in the online phase, it communicates with other enclaves to check the consistency of identifiers, causing excessive communication costs. Alternatively, one may generate all triples and authenticate them as a whole, load all triples into the enclave, and verify them before computation. However, the limited enclave memory space may not be sufficient to hold all triples. Unpredictable crashes at runtime may further require additional triples beyond these initially loaded ones.

To improve performance, *an enclave running in the online phase of the protocol should dynamically and locally determine the expected identifier of Beaver triple.* It requires that as each Beaver triple is generated, it attaches an identifier similar to that of checkpointing, i.e., a composed identifier containing a session ID, an enclave ID, and a triple ID. Note that Beaver triples are generated to be used by a set of multiplication gates in the circuit, in the order of their gate IDs. At the beginning of the online phase, in addition to the (online) session ID, the gate ID, and the set of enclave IDs (Section 4.2), all enclaves should also synchronize the (offline) session ID and the enclave IDs for Beaver triples, as well as the first unconsumed triple ID. When a multiplication gate needs a Beaver triple, each enclave loads the authenticated shares and checks whether the composed identifier matches the agreed one. The expected triple ID can be derived by gate ID.

4.4 Full Protocol

We now combine the aforementioned techniques to upgrade semi-honest protocols to defend against malicious adversaries, using Semi [Kel20a] as an example. We formalize the protocol of the online phase as Protocol 1, which installs and invokes Program 1 into each party's local filmy enclave as modeled in Functionality 1. Program 1 embodies the

Functionality 4 $\mathcal{G}_{\text{offline}}\left[\{P_i\}_{i\in\{0,\ldots,n-1\}}, \mathsf{reg}, M\right]$

$\mathcal{G}_{\text{offline}}$ receives the identities of corrupted parties Corr from the ideal adversary $\mathcal{S}$.
$\mathcal{G}_{\text{offline}}$ gets inputs: the desired # of Beaver triples m and a program prog, and sets $s := 0$.
while $\mathcal{G}_{\text{offline}}$ receives t from $\mathcal{S}$, $t \neq \perp$ and $s < m$ **do**
    $\mathcal{G}_{\text{offline}}$ generates t Beaver triples $\{([a_k], [b_k], [c_k])\}_{k\in\{0,\ldots,t-1\}}$ where $c_k = a_k \cdot b_k$; $s := s + t$.
    $\mathcal{G}_{\text{offline}}$ asserts all $P_i \in \mathsf{reg}$, generates $eid^i_{\text{pre}} \leftarrow\!\$\ \{0,1\}^{\lambda'}$ for each P_i, and distributes $eids_{\text{pre}} := \{eid^j_{\text{pre}}\}_{j\in\{0,\ldots,n-1\}}$ to each P_i.
    $\mathcal{G}_{\text{offline}}$ derives the sealing key $\mathsf{sk}_i := \mathsf{KDF}(M[P], \mathsf{prog} \| eid^i_{\text{pre}})$ for each P_i, and encrypts the shares of each party with the key $sealed\_triple_{k,i} := \mathsf{AE.Enc}_{\mathsf{sk}_i}(((a_{k,i}, b_{k,i}, c_{k,i}), k, eids_{\text{pre}}))$ for all $k \in \{0, \ldots, t-1\}$.
    $\mathcal{G}_{\text{offline}}$ sends $\{sealed\_triple_{k,i}\}_{k\in\{0,\ldots,t-1\}}$ to each P_i.

Protocol 1 $\mathsf{Prot}\left[\mathcal{C}, \{P_i\}_{i\in\{0,\ldots,n-1\}}\right]$

procedure ATTEST(Others, i, eid_i, prog)
    for all $j \in$ Others **do**
        send eid_i to P_j, await eid_j from P_j; $x^j_i, \sigma^j_i := \mathcal{G}_{\text{TEE}}.\text{attest}(eid_i, eid_j)$
        send x^j_i, σ^j_i to P_j; await x^i_j, σ^i_j from P_j; $\mathcal{G}_{\text{TEE}}.\text{verify}(eid_i, eid_j, \mathsf{prog}, x^i_j, \sigma^i_j)$
    return $\{eid_j\}_{j\in\{0,\ldots,n-1\}}$

For party P_i: on input "setup" inp_i from environment $\mathcal{Z}$:
obtain $eids_{\text{pre}}$ and $\{sealed\_triple_{k,i}\}_{k\in\{0,\ldots,m-1\}}$ by invoking $\mathcal{G}_{\text{offline}}$; and set $base_{\text{pre}} := 0$ as the first unconsumed triple ID
$eid_i := \mathcal{G}_{\text{TEE}}.\text{install}(\mathsf{prog}[\mathcal{C}, \{P_i\}_i])$; $eids := \text{ATTEST}(\{0, \ldots, n-1\} \setminus \{i\}, i, eid_i, \mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
$y, g := \mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"init"}, i, eids, eids_{\text{pre}}, base_{\text{pre}}, \mathsf{inp}_i)$
record enclave eid_i computes the sealed output y of gate g in an object $\mathcal{C}$ ▷ *We abuse $\mathcal{C}$ here*
persist $\mathcal{C}$, $eids_{\text{pre}}$, $base_{\text{pre}}$, $\{sealed\_triple_{k,i}\}_{k\in\{0,\ldots,m-1\}}$

For party P_i: on input "recover" $\{P_k\}_{k\in K}$ from environment $\mathcal{Z}$:
if $i \in K$ **then** ▷ *K is the set of crashed parties*
    restore $\mathcal{C}$, $eids_{\text{pre}}$, $base_{\text{pre}}$; $eid_i := \mathcal{G}_{\text{TEE}}.\text{install}(\mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
    $eids := \text{ATTEST}(\{0, \ldots, n-1\} \setminus \{i\}, i, eid_i, \mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
else
    $eids := \text{ATTEST}(K, i, eid_i, \mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
get the maximum layer number $layer$ in which all gates have outputs ▷ *current progress in $\mathcal{C}$*
exchange $layer$ with others to derive the common latest finished $layer$, exchange $eids_{\text{pre}}$ and $base_{\text{pre}}$ with others to derive the largest $base_{\text{pre}}$, and persist $eids_{\text{pre}}$, $base_{\text{pre}}$
$s :=$ output values of all gates in $layer$ of $\mathcal{C}$ ▷ *$s = \{(eid^i_g, y_g)\}_g$, set of gates*
$\mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"recover"}, i, eids, eids_{\text{pre}}, base_{\text{pre}}, s)$

For party P_i: on input "compute" from environment $\mathcal{Z}$:
$g :=$ a gate in $\mathcal{C}$ who has inputs ready but does not finish computing ▷ *in topological order*
restore $base_{\text{pre}}$, and the sealed triple, i.e., $k = base_{\text{pre}} + g$
$res_i, g, \_ := \mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"compute"}, sealed\_triple_{k,i})$
record enclave eid_i computes the sealed output res_i of gate g in the object $\mathcal{C}$, persist $\mathcal{C}$

For party P_i: on input "output" from environment $\mathcal{Z}$:
$res := \mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"output"}, \{eid_j\}_{j\in\{0,\ldots,n-1\}})$

minimization of code placed within the TEE. We only show the procedures for addition and multiplication. For simplicity, we omit the scenario where one of the two operands is not a secret value, in which case the computation can be trivially done locally. We also omit the implementation details of defenses against network manipulation (e.g., replay attacks), which can be solved by adding counters at both ends. In cases where a party or its enclave does not receive expected messages in time, it just aborts the protocol. The

Program 1 $\mathsf{prog}[\mathcal{C}, \{P_i\}_{i\in\{0,\dots,n-1\}}]$

procedure ExDataWith(i, j, *data*, *validation*)
  $y^i_j := \mathcal{G}_{\text{TEE}}.\text{authenc}(eid_j, (data, validation), 1)$ ▷ *encrypt*
  send y^i_j to enclave eid_j; await y^j_i from enclave eid_j
  $(data', validation') := \mathcal{G}_{\text{TEE}}.\text{authenc}(eid_j, y^j_i, 0)$; assert $validation = validation'$ ▷ *decrypt*
  return $data'$

On input ("init", i, *eids*, $eids_{\text{pre}}$, $base_{\text{pre}}$, inp):
keep i, *eids*, $eids_{\text{pre}}$, $base_{\text{pre}}$ in enclave; initialize $eids^{\text{tmp}} := \{eid_i\}$, $eids^{\text{tmp}}_{\text{pre}} := \{eid^i_{\text{pre}}\}$
for $eid_j \in eids, j \neq i$ **do**
  $(j, eid_j, eid^j_{\text{pre}}) :=$ ExDataWith$(i, j, (i, eid_i, eid^i_{\text{pre}}), (eids, eids_{\text{pre}}, base_{\text{pre}}))$
  insert $eid_j, eid^j_{\text{pre}}$ into $eids^{\text{tmp}}, eids^{\text{tmp}}_{\text{pre}}$
assert all j's appear, $eids = eids^{\text{tmp}}$, $eids_{\text{pre}} = eids^{\text{tmp}}_{\text{pre}}$
$(s_{i,1}, \cdots, s_{i,n}) :=$ Share(inp)
for $eid_j \in eids, j \neq i$ **do**
  $s_{j,i} :=$ ExDataWith$(i, j, s_{i,j}, _)$
$\mathcal{C}_e[0] = (s_{1,i}, \cdots, s_{n,i})$, $y := \mathcal{G}_{\text{TEE}}.\text{seal}(eid_i, (\mathcal{C}_e[0], 0, eids), 1)$ ▷ *$\mathcal{C}_e$ is an object for circuit $\mathcal{C}$*
return y, 0

On input ("recover", i, *eids*, $eids_{\text{pre}}$, $base_{\text{pre}}$, *sealed_pairs*):
for (eid^i_g, y_g) in *sealed_pairs* **do**
  $(s_g, gate_g, eids_g) := \mathcal{G}_{\text{TEE}}.\text{seal}(eid^i_g, y_g, 0)$, $\mathcal{C}_e[gate_g] := s_g$
assert all $eids_g$ and eid^i_g are equal; keep i, $\mathcal{C}_e$, *eids*, $eids_{\text{pre}}$, $base_{\text{pre}}$ in enclave
$gates := \{gate_g\}_g$; init $eids^{\text{tmp}} := \{eid_i\}$, $eids^{\text{tmp}}_g := \{eid^i_g\}$, $eids^{\text{tmp}}_{\text{pre}} := \{eid^i_{\text{pre}}\}$
for $eid_j \in eids, j \neq i$ **do**
  $(j', eid_j, eid^j_g, eid^j_{\text{pre}}) :=$ ExDataWith$(i, j, (i, eid_i, eid^i_g, eid^i_{\text{pre}}),$
    $(eids, eids_g, gates, eids_{\text{pre}}, base_{\text{pre}}))$; assert $j' = j$
  insert $eid_j, eid^j_g, eid^j_{\text{pre}}$ into $eids^{\text{tmp}}, eids^{\text{tmp}}_g, eids^{\text{tmp}}_{\text{pre}}$
assert $eids = eids^{\text{tmp}}$, $eids_g = eids^{\text{tmp}}_g$, $eids_{\text{pre}} = eids^{\text{tmp}}_{\text{pre}}$

On input ("compute", *seal_tri*):
get the next gate g to compute, $(x, y) := \mathcal{C}_e[g]$
if *g.isMult* **then** ▷ *g is a multiplication gate*
  retrieve $eids_{\text{pre}}$, $base_{\text{pre}}$ in enclave
  $((a, b, c), offset_{\text{pre}}, eids'_{\text{pre}}) := \mathcal{G}_{\text{TEE}}.\text{seal}(eid^i_{\text{pre}}, seal\_tri, 0)$
  assert $offset_{\text{pre}} = base_{\text{pre}} + g$, $eids_{\text{pre}} = eids'_{\text{pre}}$
  if $i \neq 0$ **then**
    $(\alpha, \beta) :=$ ExDataWith$(i, 0, (x + a, y + b), _)$; $z := \alpha \cdot \beta - \alpha \cdot b - \beta \cdot a + c$
  else ▷ *$i = 0$*
    await res^j_0 from each enclave eid_j; then $(\alpha_j, \beta_j) := \mathcal{G}_{\text{TEE}}.\text{authenc}(eid_j, res^j_0, 0)$
    $(\alpha, \beta) := \sum_{j=1}^{n} (\alpha_j, \beta_j)$; broadcast $res^0_j := \mathcal{G}_{\text{TEE}}.\text{authenc}(eid_j, (\alpha, \beta), 1)$
    $z := \alpha \cdot \beta - \alpha \cdot b - \beta \cdot a + c$
  $\mathcal{C}_e[g] = z$; keep $\mathcal{C}_e$ in enclave ▷ *gate output*
else
  $\mathcal{C}_e[g] = x + y$; keep $\mathcal{C}_e$ in enclave ▷ *gate output*
$res := \mathcal{G}_{\text{TEE}}.\text{seal}(eid_i, (\mathcal{C}_e[g], g, eids), 1)$
return *res*, g, *isMult*

On input ("output"):
$z_i :=$ the output of a final gate in topology order
if $z_i = \perp$ **then** abort
if $i \neq 0$ **then**
  $z :=$ ExDataWith$(i, 0, z_i, _)$
else ▷ *$i = 0$*
  await res^j_0 from each enclave eid_j; $z_j := \mathcal{G}_{\text{TEE}}.\text{authenc}(eid_j, res^j_0, 0)$
  $z := \sum_{j=1}^{n} (z_j)$; broadcast $res^0_j := \mathcal{G}_{\text{TEE}}.\text{authenc}(eid_j, z, 1)$
return z

offline phase for the Beaver triple generation follows ideas similar to those of the online phase. For brevity, we include the details of the offline phase in the supplementary material and instead abstract a global functionality in Functionality 4. The offline phase should generate the same sealing keys as the online phase so the Beaver triples can be used. We combine the online (Program 1) and offline enclave programs into one, so the two phases have the same platform and the same enclave program. We assume the offline phase has prepared enough Beaver triples in one iteration for the online processing.

We briefly describe the flow of the protocol below. During initialization, all the parties attest each others' enclaves (Protocol 1 Lines 7 to 8), and eventually agree on the current enclave committee *eids*, the offline phase committee $eids_{\text{pre}}$, and the first unconsumed Beaver triple $base_{\text{pre}}$ in the init enclave function (Program 1 Lines 6 to 10). Here we use the enclave committee *eids* as the session ID. To avoid ID collision with high probability, λ' is significantly larger than λ. For instance, with $\lambda = 128$, setting $\lambda' = 384$ ensures a collision probability of 10^{-18} even for 8.9×10^{48} samples. Therefore this agreement satisfies the requirements in Sections 4.2 and 4.3. After that, each party secretly shares its respective input with others (Program 1 Lines 11 to 13).

If a node crashes, all its volatile states, including channels with other enclaves, would be lost. The restarted enclave rebuilds the secure channels through attestation (Protocol 1 Lines 11 to 15). Each party should get the current progress of circuit evaluation, and synchronizes the progress(Lines 16 to 17). This may cause the lived parties to recompute some finished gates, but this has no security issues (Section 4.2). The recover enclave function needs to verify the consistency of the new committee *eids* and the pre-crash committee $eids_g$ (Section 4.2).

In the normal compute function, we evaluate circuit $\mathcal{C}$ gate by gate, following the protocol [DPSZ12]. For each gate, the enclave records the progress and seals the information including the gate ID, the gate output, and the enclave committee (Program 1 Line 38). The sealed states are persisted to storage outside the enclave (Protocol 1 Line 23). Finally, output returns the final result.

Comparison with the transparent enclave model [TZL+17]. In their model, the keys for authenticating user messages are exposed, so the adversary can arbitrarily change the messages between enclaves. All other mechanisms, such as sealing and the optimizations in Section 5, are invalidated due to the leakage of relevant keys. Therefore transparent enclaves can hardly upgrade MPC protocols.

4.5 Security Proof

We now prove that Protocol 1 UC-realizes $\mathcal{F}_{\mathcal{C}}$ (Functionality 3). We assume the number of corrupted parties $t = n - 1$, and only P_i is honest. We also assume the parties' inputs and the function output have fixed lengths (otherwise using padding) for simplicity. We point out that, since the adversary may call $\mathcal{G}_{\text{TEE}}$.reveal at any time to get the in-enclave state, we cannot directly use previous techniques (e.g., [PST17]) to allow the simulator to program the output using the backdoor; otherwise the environment can distinguish the real world and ideal world as the parameter for output programming is vacant in real world but provided in ideal world. On the other hand, since our model separately captures the TEE attestation feature, each $\mathcal{G}_{\text{TEE}}$.resume call does not need to return a signed signature along with the output any longer. We directly utilize the fact that the simulator can intercept the adversary's communication with $\mathcal{G}_{\text{TEE}}$ and modify the output of $\mathcal{G}_{\text{TEE}}$ to make the views of the two worlds indistinguishable.

We construct a simulator $\mathcal{S}$, which behaves like honest parties and local functionalities (if any) to interact with the real-world adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. Unless otherwise stated later, $\mathcal{S}$ passes through messages between corrupted parties and global functionalities $\mathcal{G}_{\text{TEE}}$, $\mathcal{G}_{\text{offline}}$ (Functionalities 1 and 4). Besides, $\mathcal{S}$ also passes through

messages between corrupted parties and the environment $\mathcal{Z}$.
On input "setup":

- $\mathcal{S}$ calls $\mathcal{G}_{\text{offline}}$ and obtain results. Besides, $\mathcal{S}$ records the results as others calls $\mathcal{G}_{\text{offline}}$.
- $\mathcal{S}$ calls $eid_i := \mathcal{G}_{\text{TEE}}.\text{install}(\mathsf{prog}[\mathcal{C}, \{P_i\}_{i\in\{0,\dots,n-1\}}])$, sends eid_i to every P_j and receives eid_j from every P_j. If the received eid_j is not the answer to the previous install call from P_j, or P_j does not install the specific program, $\mathcal{S}$ jumps to the exception handler denoted **except**.
- $\mathcal{S}$ calls $x_i^j, \sigma_i^j := \mathcal{G}_{\text{TEE}}.\text{attest}(eid_i, eid_j)$, sends x_i^j, σ_i^j to P_j, and receives x_j^i, σ_j^i from P_j for every j. If the received x_j^i, σ_j^i is not the answer to the previous attest call by P_j, $\mathcal{S}$ jumps to **except**. Besides, $\mathcal{S}$ checks whether $P_j, j \neq i$ calls attest with eid_j and eid_k previously got by P_k for each k. If not, $\mathcal{S}$ also jumps to **except**.
- $\mathcal{S}$ calls $\mathcal{G}_{\text{TEE}}.\text{verify}(eid_i, eid_j, \mathsf{prog}[\mathcal{C}, \{P_i\}_{i\in\{0,\dots,n-1\}}], x_j^i, \sigma_j^i)$ for every j. Besides, $\mathcal{S}$ checks whether $P_j, j \neq i$ calls verify with the exact program and other parameters previously observed by $\mathcal{S}$.
- $\mathcal{S}$ calls $y, gate := \mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"init"}, i, \{eid_j\}_j, eids_{\text{pre}}, base_{\text{pre}}, \vec{0})$. On the other hand, as each $P_{j(\neq i)}$ calls $\mathcal{G}_{\text{TEE}}.\text{resume}(eid_j, \text{"init"}, j, \{eid_k\}_k, eids_{\text{pre}}, base_{\text{pre}}, \mathsf{inp}_j)$ for the first time, $\mathcal{S}$ extracts and sends inp_j to $\mathcal{F}_{\mathcal{C}}$ (Functionality 3). Besides, $\mathcal{S}$ checks whether the parameters in "init" except j, eid_{pre}^j, and inp for each P_j are identical, whether all j's appear, and whether all eid_{pre}^j (including eid_{pre}^i) constitute $eids_{\text{pre}}$. Otherwise $\mathcal{S}$ jumps to **except**. $\mathcal{S}$ also keeps a copy of $\mathcal{C}$ for each P_j and updates the copy as the protocol indicates.

On input "recover":

- For invocations to $\mathcal{G}_{\text{TEE}}$.install, $\mathcal{G}_{\text{TEE}}$.attest, and $\mathcal{G}_{\text{TEE}}$.verify, $\mathcal{S}$ does similar actions as in "setup". Note that if P_i crashes, $\mathcal{S}$ naturally collects *eids* of the current committee. If P_i survives, $\mathcal{S}$ detects the timeout for connection to crashed parties and maintains new *eids*.
- $\mathcal{S}$ calls $\mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"recover"}, i, \{eid_j\}_j, eids_{\text{pre}}, base_{\text{pre}}, \{(eid_g^i, y_g^i)\}_g)$. On the other hand, as every $P_j, j \neq i$ calls $\mathcal{G}_{\text{TEE}}.\text{resume}(eid_j, \text{"recover"}, j, \{eid_k\}_k, eids_{\text{pre}}, base_{\text{pre}}, \{(eid_g^j, y_g^j)\}_g)$, $\mathcal{S}$ checks whether $eids_{\text{pre}}$, $base_{\text{pre}}$, and $\{eid_k\}_k$ are the same to the ones set in $\mathcal{S}$'s invocation, whether all j's appear, and whether all eid_{pre}^j constitute $eids_{\text{pre}}$. If not, $\mathcal{S}$ jumps to **except**. Since $\mathcal{S}$ holds and updates a copy of $\mathcal{C}$ for each party, $\mathcal{S}$ is able to get $s := \{(eid_g^j, y_g^j)\}_g$ for each $P_j, j \neq i$ and checks whether the returned values are equal to the last argument of "recover"; if not, go to **except**.

On input "compute":

- $\mathcal{S}$ calls $\mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"compute"}, sealed\_triple_{k,i})$. As P_j calls the function, $\mathcal{S}$ checks whether the $sealed\_triple_{k,j}$ has the same k and whether it is consistent with the record in the first step of "setup". Besides, $\mathcal{S}$ checks whether each *sealed_triple* is used only once by each P_j if the returned $isMult = true$; otherwise $\mathcal{S}$ jumps to **except**. $\mathcal{S}$ updates the copy of $\mathcal{C}$ for each P_j.

On input "output":

- As $P_j, j \neq i$ calls $res := \mathcal{G}_{\text{TEE}}.\text{resume}(eid_j, \text{"output"})$. If $res \neq \perp$, $\mathcal{S}$ sends "proceed" to $\mathcal{F}_{\mathcal{C}}$, receives outp from $\mathcal{F}_{\mathcal{C}}$, replaces res with outp. Otherwise, $\mathcal{S}$ sends "abort".
- As corrupted P_j and $j = 0$ calls $\mathcal{G}_{\text{TEE}}$.reveal, $\mathcal{S}$ modifies the returned states. Specifically, if any z_i should be revealed, $\mathcal{S}$ intercepts z_i and replaces it with $\mathsf{outp} - \sum_{k\neq i} z_k$. Further, if z should be revealed, $\mathcal{S}$ replaces it with outp.

Except: if any exception was triggered before $\mathcal{S}$ calls the enclave function init, $\mathcal{S}$ refuses to provide input to $\mathcal{F}_{\mathcal{C}}$. Otherwise, if any exception was triggered before $\mathcal{S}$ calls the enclave function output, $\mathcal{S}$ provides the instruction "abort" to $\mathcal{F}_{\mathcal{C}}$.

Now we use the hybrid argument to prove the validity of $\mathcal{S}$, starting from a real-world execution of Protocol 1. Due to lack of space, we only highlight the essential steps.

Hybrid H_1 proceeds as the real protocol, except that whenever the enclave of the

honest party uses the random a and b to compute $xpa = x + a$ and $ypb = y + b$, $\mathcal{S}$ first samples xpa and ypb and then compute $a = xpa - x$, $b = ypb - y$. We only change the way of preparing the (decrypted) messages that would be seen by corrupted parties; the output distributions are identical.

Hybrid H_2 proceeds as H_1, except that during the output phase, if no abort occurs, $\mathcal{S}$ first computes the circuit output z and reconstructs $[z]$ by z and the shares of the corrupted parties. Since $[z]$ is uniquely determined by the $n-1$ shares of the corrupted parties and z, the two distributions are identical.

Hybrid H_3 proceeds as H_2, except that during the output phase, if no abort occurs, $\mathcal{S}$ just takes the output $\mathsf{outp} = z$ from Functionality 3, and reconstructs $[z]$. Since in H_2, the input of the honest party is only used to compute the function outputs, while in H_3 the outputs are computed by the ideal functionality Functionality 3, it makes no difference.

Hybrid H_4 proceeds as H_3, except that the input of the honest party is replaced with $\vec{0}$. Since the shares of Beaver triples are random numbers, and AE ensures the adversary can only distinguish two plaintexts from the ciphertext with negligible probability, the distribution should be the same.

Hybrid H_5 proceeds as H_4, except that all the aborts in the protocol (including the abort of in-enclave program) are removed while $\mathcal{S}$ adds the exception handler **except**. Since all the aborts in the protocol can be reduced to the violence of rules about exceptions, given that the signature scheme Σ is secure and that AE has INT-CTXT security, the distribution should be the same.

5 Protocol Optimization with TEEs

Apart from MPC protocol upgrade, in this section, we will show TEEs also help optimize their performance and lead to new designs. We investigate two primitives: oblivious transfer (OT) [Rab05, Bea96, IKNP03, KOS15] and oblivious product evaluation (OPE) [Gil99, KOS16]. These primitives could be used in both the offline phase (e.g., for Beaver triple generation) and the online phase (e.g., boolean to arithmetic sharing [DSZ15]) of MPC. We use TEEs to build communication-round-optimal protocols for these primitives.

5.1 Oblivious Transfer

Incorporating TEEs, [DMMQ23] have optimized OT and random OT protocols using the same method: the sender sends both encrypted messages to the receiver, and the receiver is forced only to decrypt one of them. It is optimal for OT but not for random OT.

We further propose an optimized random OT protocol shown in Program 2 realizing $\mathcal{F}_{\mathrm{ROT}}^{l,k}$. After seed negotiation, the pseudo-random generator yields a sequence of r_i for both enclaves. The enclave generates a pair of random numbers by "encrypting" $r_i||0$ and $r_i||1$ for the sender in the i-th round. For the receiver, it derives one of the random numbers according to its choice in the i-th round. Compared to [DMMQ23], we involve nearly no communication except for the negotiation of the random seed. In each round, instead of transmitting two encrypted random numbers to the receiver to choose [DMMQ23], the receiver's enclave *locally generates* the chosen random number without communication. Program 2 is secure in the filmy enclave model. S is oblivious to b, while R cannot derive $a^i_{1-b_i}$ by computing $\mathcal{G}_{\mathrm{TEE}}$.authenc without sk.

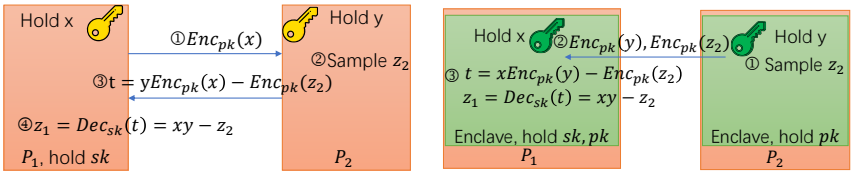
5.2 Oblivious Product Evaluation

To construct OPE for Beaver triple generation, besides using OT, semi-homomorphic encryption can also be used as the building block [KPR18]. Figure 2a shows the OPE protocol defending against semi-honest adversaries, which requires two rounds of communication.

Program 2 $\mathsf{prog}_{\mathrm{ROT}^{l,k}}[S,R]$: l iterations with k-bit messages

For sender S: on input ("run-rot", eid_R):
1: $seed \leftarrow\$ \{0,1\}^\lambda$; send $y := \mathcal{G}_{\mathrm{TEE}}.\mathrm{authenc}(eid_R, seed, 1)$ to eid_R; $(r_0, \cdots, r_{l-1}) := \mathsf{PRG}(seed)$
2: for $i \in \{0, \ldots, l-1\}$: $a_0^i := \mathcal{G}_{\mathrm{TEE}}.\mathrm{authenc}(eid_R, r_i||0, 1)$, $a_1^i := \mathcal{G}_{\mathrm{TEE}}.\mathrm{authenc}(eid_R, r_i||1, 1)$
3: **return** $\{a_0^i, a_1^i\}_{i\in\{0,\ldots,l-1\}}$

For receiver R: on input ("run-rot", eid_S, $\{b_i\}_{i\in\{0,\ldots,l-1\}}$):
4: await y from eid_S; set $seed = \mathcal{G}_{\mathrm{TEE}}.\mathrm{authenc}(eid_R, y, 0)$; $(r_0, \cdots, r_{l-1}) := \mathsf{PRG}(seed)$
5: for $i \in \{0, \ldots, l-1\}$: $a_{b_i}^i := \mathcal{G}_{\mathrm{TEE}}.\mathrm{authenc}(eid_S, r_i||b_i, 1)$
6: **return** $\{a_{b_i}^i\}_{i\in\{0,\ldots,l-1\}}$



(a) Protocol in [KPR18] **(b)** Our protocol

Figure 2: Optimizing OPE performance using TEEs.

To optimize the protocol using TEEs, we still utilize the feature that TEEs protect cryptographic procedures. However, in this case, we require the TEE to protect the encryption/decryption procedure of semi-homomorphic encryption, which is not formalized in Functionality 1. Our optimized protocol is shown in Figure 2b. Owing to code integrity enforcement and key protection, the enclave for P_1 cannot decrypt $\mathsf{Enc}_{pk}(y)$ immediately at the time of receiving it, preventing P_1 from learning the secret of P_2. Although the amount of communication is the same as the basic one, it only needs one round of communication.

6 Evaluation

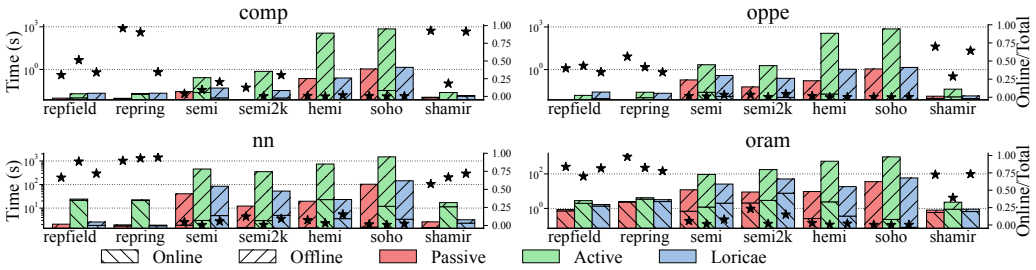
6.1 Implementation

We integrate all of our proposed techniques into MP-SPDZ [Kel20a]. Note that our techniques are not specific to any protocol, so our implementation naturally upgrades the original passively-secure protocols in MP-SPDZ to defend against malicious adversaries using enclaves. Such an integration also allows for fair comparisons with the native actively-secure protocols in MP-SPDZ. We insert checkpoints between basic blocks, inside which MP-SPDZ rearranges instructions for round-minimizing optimizations [Kel20a]. The frequency of checkpointing can be adjusted by users. To avoid slow disk accesses during checkpointing to block the main protocol execution, we copy the states to another memory area and asynchronously persist the copied data using another thread.

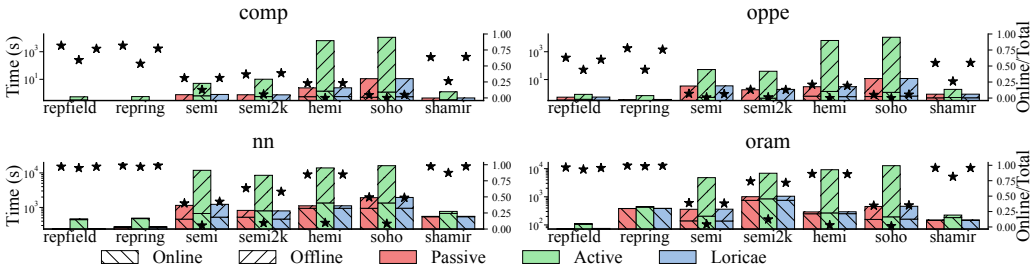
6.2 Experimental Setup

Platform. we use several `ecs.g7t.4xlarge` instances on Alibaba Cloud, with Intel SGX providing 32 GB trusted memory and up to 25 Gbps inter-machine network bandwidth for LAN. We emulate WAN by using the kernel component `netem` [The23], setting to 40 ms round-trip latency and 40 Mbps throughput. We use Occlum v0.30.0 [STC+20] as a SGX libOS to run MP-SPDZ inside enclaves.

Workloads. We use typical workloads [MR18, LX19, PSSY21] from the open-sourced MP-SPDZ library [Kel20b] for performance evaluation. (1) `comp` does greater-than compar-



(a) LAN results, with time breakdown.



(b) WAN results, with time breakdown.

Figure 3: Overall performance comparison among passive, active, and Loricae designs.

ison between two 10-sized arrays of integers; (2) `oppe` uses oblivious piecewise polynomial evaluation for the sigmoid function; (3) `oram` uses a *distributed oblivious RAM* (DORAM) protocol [KS14] to access one element from a 10,000-sized array. Unlike the traditional client-server ORAM, DORAM is implemented as a secret-sharing-based MPC protocol, in which both the data and the indices are in secret shares, and the data access procedure is oblivious. (4) `nn` applies a neural network with 1 million input samples. Among them, `comp` and `oppe` are small benchmarks, while the others are larger ones.

Baselines. We evaluate 7 passively secure protocols `repring`, `repfield`, `semi2k`, `semi`, `hemi`, `soho`, and `shamir`. For each protocol, we compare three versions, namely the *passive* baseline, the native version with *active* security (`psrepring`, `psrepfield`, `spdz2k`, `mascot`, `lowgear`, `highgear`, and `malicious-shamir`, respectively), and the Loricae upgraded one. The `repring`, `repfield`, and `shamir` protocols assume the honest majority setting, while others are in the dishonest majority setting. By default, We set 3 parties for the honest majority setting and 2 for the dishonest one.

6.3 Overall Performance and Breakdown

Figure 3 shows the overall performance comparison between Loricae and the baselines in both the LAN and WAN settings. Here we do not enable the optimizations in Section 5, in order to precisely identify the cost only from the security upgrade with TEEs. The execution latency is split into the online phase and the offline phase. As expected, the offline phase of the dishonest-majority protocols dominates the overall overheads. We compare the total latency across designs.

In the LAN setting, for small benchmarks `comp` and `oppe`, although the overheads of the SGX software stack cause Loricae to run slightly slower than the actively secure ones of `repfield` and `repfield`, Loricae achieves 1.2× speedups over *active* for honest-majority protocols on average. For protocols in the dishonest-majority setting, the communication overheads for computing verification are much larger, and Loricae can achieve significant speedups over the actively secure versions, by 71.4× on average. On the other hand,

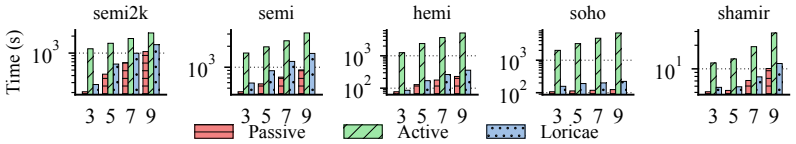


Figure 4: Performance comparison as the number of parties increases.

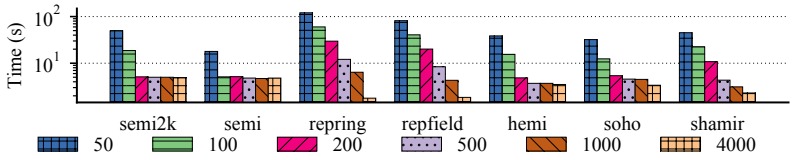


Figure 5: Online phase time as the checkpoint period varies.

Loricae incurs an average 2.2× overhead compared to the passively secure schemes. For larger benchmarks, the speedups of Loricae over *active* range from 1.2× to 32.0×, while the slowdown compared to *passive* is 1.7× on average. The overheads of Loricae over *passive* mainly result from the libOS in TEEs, where the syscalls and I/O may take significant execution time that is orthogonal to our work. Constant-time execution incurs less than 10% overheads. Overheads from libOS may be reduced by accelerating I/O [TUP+21] or manually partitioning the MP-SPDZ and putting partial codes into the enclave. We leave them for future work.

We notice that for `nn` on `repring` and `repfield`, Loricae is even faster than *passive*. We suspect this is caused by the Occlum libOS, which replaces the standard glibc library. To verify it, we run Loricae and the baselines on an AMD EPYC 7543P processor that supports SEV-SNP [SS20], with the same OS support in the VM TEE. In that case, Loricae is 1.3× and 1.7× slower than *passive* on the same two benchmarks.

In the WAN setting, Loricae incurs negligible slowdown compared to *passive*, since the extra overheads of SGX and Occlum on local execution are overwhelmed by the high communication cost of WAN. On the other hand, Loricae outperforms *active* by 1.1× to 2619.9×, and averages at 10.8×.

6.4 Varying Numbers of Parties

Figure 4 shows the execution time when the number of parties increases from 3, 5, 7, to 9. We exclude `repring` and `repfield` as they only support the 3-party setting. As the committee size grows, the communication overheads increase and the overall performance degrades in all designs. The slowdown of Loricae over *passive* is relatively stable, while the downturn in *active* over *passive* is exacerbated with more parties. For example, as the number of parties increases, the slowdown of *active* is amplified from 18× to 55×.

6.5 Checkpoint Overheads

The checkpoint overheads mainly come from (1) temporarily locking and copying the memory states; (2) persisting the states with a single thread in our current implementation. Despite implemented asynchronously, the frequency of checkpointing still has large performance impact. The default checkpoint period is 4000 basic blocks in the previous evaluation. Figure 5 shows the performance impact as the checkpoint period varies in the LAN setting. The performance naturally improves as the period gets longer. We observe that the main bottleneck is the single-thread persisting operation. In contrast, in the WAN setting, as the period also varies from 50 to 4000 basic blocks, the performance

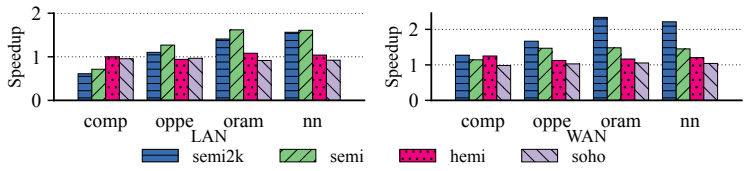


Figure 6: Offline phase speedups from the optimizations.

has almost no observable changes, since the local persisting latency is well hidden by the remote communication in the main thread.

6.6 Optimizations on OT and OPE

Next we focus on the offline phase. We apply the OT optimizations (Section 5.1) to generate Beaver triples in the offline phase of `semi` and `semi2k` and OPE optimizations (Section 5.2) in `hemi` and `soho`. Figure 6 shows the offline phase performance comparison with the original triple generation approaches *without TEEs*. With the LAN setting, for small benchmarks like `comp`, optimizing OT with TEEs even slows down the execution due to the use of SGX and Occlum. For large benchmarks, it achieves a 1.5× speedup for OT-based Beaver triple generation. Although the communication overhead of random OT is nearly reduced to zero, the whole triple-generation procedure still needs other communication, limiting the overall latency improvement. Similarly, with the relatively low latency in LAN, the round reduction for HE-based OPE also has an insignificant speedup. In the WAN setting, the improvements from both optimizations become more substantial. Optimizations on OT and HE-based OPE lead to 1.5× and 1.1× speedups on average.

7 Related Work

Some previous designs [PST17, TZL+17, BBB+17, CTH+19, ZXWG22, DW23] also combined cryptographic algorithms and TEEs. Pass et al. [PST17] formalized the TEE model and used it to circumvent some impossibility results in MPC protocols. Tramèr et al. [TZL+17] proposed the transparent enclave model and designed ZKP with it. Choi et al. [CTH+19] allowed the protocol designer to choose a component of function to run either inside the enclave or via standard cryptographic techniques. Bahmani et al. [BBB+17] simply used TEEs as a trusted third party to achieve the goal of MPC, while Felsen et al. [FKSW19] additionally crafted software implementation to avoid some side-channel issues. PPMLAC [ZXWG22] designed a specialized trusted hardware chip (similar to TEEs) to reduce the communication overhead of MPC protocols. STAMP [HHL+22] used a dedicated but less powerful trusted chip to accelerate privacy-preserving machine learning tasks. Lu et al. [LZZ+21] and Dong et al. [DW23] used TEEs to prepare correlated randomness for MPC protocols of the online phase. In specific scenarios, Wu et al. [WNS+22] utilized different trust levels to apply TEEs or MPC. For machine learning inference, Chex-Mix [NLDD23] protected the confidentiality of clients with homomorphic encryption, and the confidentiality and integrity of model providers with TEEs.

8 Conclusions

Loricae realizes secure multi-party computation against malicious adversaries with significantly better performance than existing designs, which is enabled by combining efficient semi-honest MPC protocols and hardware TEEs. The security of Loricae

can be formally proved even in the presence of various side-channel vulnerabilities in TEEs. Essentially, the semi-honest MPC data format ensures data confidentiality, while TEE program attestation and our proposed protection for external input data together guarantee execution integrity. LORICAE represents a novel direction that symphonizes the complementary advantages of cryptographic and hardware-based approaches for privacy-preserving computing.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable suggestions, as well as Xin Yang, Liang Zhang, Fan Zhang, and the Tsinghua IDEAL group members, for constructive discussion. Mingyu Gao is the corresponding author.

References

[ABF+17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries—Breaking the 1 Billion-Gate per Second Barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, 2017.

[ACE+23] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Security Review. Technical report, Google technical report, 2023.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS 2016*, pages 805–817, 2016.

[AFO+21] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure Graph Analysis at Scale. In *ACM CCS 2021*, pages 610–629, 2021.

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *2nd HASP*, volume 13, 2013.

[Alv04] Tiago Alves. TrustZone: Integrated Hardware and Software Security. *White paper*, 2004.

[ARM21] ARM. Arm Confidential Compute Architecture. https://developer.arm.com/documentation/den0125/0200/?lang=en, 2021.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI 16*, pages 689–703, 2016.

[BBB+17] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure Multiparty Computation from SGX. In *FC 2017*, pages 477–497, 2017.

[BBC+14] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level Non-interference for Constant-time Cryptography. In *ACM CCS 2014*, pages 1267–1279, 2014.

[BCLK17] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory. In *DSN 17*, pages 157–168, 2017.

[BCS13] Daniel J Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast Constant-Time Code-Based Cryptography. In *CHES 2013*, pages 250–272, 2013.

[Bea92] Donald Beaver. Efficient Multiparty Protocols using Circuit Randomization. In *CRYPTO'91*, pages 420–432, 1992.

[Bea96] Donald Beaver. Correlated Pseudorandomness and the Complexity of Private Computations. In *28th ACM STOC*, pages 479–488, 1996.

[BGG+13] Lilian Bossuet, Michael Grand, Lubos Gaspar, Viktor Fischer, and Guy Gogniat. Architectures of Flexible Symmetric Key Crypto Engines—A Survey: From Hardware Coprocessor to Multi-Crypto-Processor System on Chip. *ACM Computing Surveys*, 45(4):1–32, 2013.

[BKS+22] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security 2022*, pages 3917–3934, 2022.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *20th ACM STOC*, pages 1–10, 1988.

[BWS19] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *ACM CCS 2019*, pages 1087–1099, 2019.

[Cai19] Liang Cai. Guard Your Data with the Qualcomm Snapdragon Mobile Platform. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf, 2019.

[Can00] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13:143–202, 2000.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*, pages 136–145, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty Unconditionally Secure Protocols. In *20th ACM STOC*, pages 11–19, 1988.

[CCX+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy*, pages 142–157, 2019.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally Composable Security with Global Setup. In *TCC 2007*, pages 61–85, 2007.

[CDS13] Chai Wen Chuah, Ed Dawson, and Leonie Simpson. Key Derivation Function: The SCKDF Scheme. In *IFIP Advances in Information and Communication Technology*, volume 405, pages 125–138, 07 2013.

[CF85] Josh D Cohen and Michael J Fischer. A Robust and Verifiable Cryptographically Secure Election Scheme (Extended Abstract). In *26th FOCS*, pages 372–382, 1985.

[CO15] Tung Chou and Claudio Orlandi. The Simplest Protocol for Oblivious Transfer. In *LATINCRYPT 2015*, pages 40–58, 2015.

[Cos16] Victor Costan. Intel SGX Explained, 2016.

[CR03] Ran Canetti and Tal Rabin. Universal Composition with Joint State. In *CRYPTO 2003*, pages 265–281, 2003.

[CTH+19] Joseph I Choi, Dave Tian, Grant Hernandez, Christopher Patton, Benjamin Mood, Thomas Shrimpton, Kevin RB Butler, and Patrick Traynor. A Hybrid Approach to Secure Function Evaluation using SGX. In *ASIACCS 19*, pages 100–113, 2019.

[DEF+19] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120, 2019.

[DEK20] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure Evaluation of Quantized Neural Networks. *PoPETs*, 2020(4):355–375, 2020.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC 2006*, pages 285–304, 2006.

[DMMQ23] Felix Dörre, Jeremias Mechler, and Jörn Müller-Quade. Practically Efficient Private Set Intersection from Trusted Hardware with Side-Channels. In *ASIACRYPT 2023, Part IV*, pages 268–301, 2023.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and Unconditionally Secure Multiparty Computation. In *CRYPTO 2007*, pages 572–590, 2007.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*, pages 643–662, 2012.

[DSC+15] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security 2015*, pages 447–462, 2015.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*, 2015.

[DW23] Wentao Dong and Cong Wang. Poster: Towards Lightweight TEE-Assisted MPC. In *ACM CCS 2023*, pages 3609–3611, 2023.

[EHZH+22] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. Benchmarking the Second Generation of Intel SGX Hardware. In *DaMoN 22*, pages 1–8, 2022.

[FCW+22] Xiaoyu Fan, Kun Chen, Guosai Wang, Mingchun Zhuang, Yi Li, and Wei Xu. NFGen: Automatic Non-linear Function Evaluation Code Generator for General-Purpose MPC Platforms. In *ACM CCS 2022*, pages 995–1008, 2022.

[FCW+25] Xiaoyu Fan, Kun Chen, Guosai Wang, Xiaowei Zhu, Haoqing He, Xie Yong, Xiaofeng Jia, Yidong Li, and Wei Xu. Pair-Then-Aggregate: Simplified and Efficient Parallel Programming Paradigm for Secure Multi-Party Computation. In *IPDPS 25*, 2025.

[FCY+25] Xiaoyu Fan, Kun Chen, Jiping Yu, Xiaowei Zhu, Yunyi Chen, Huanchen Zhang, and Wei Xu. GORAM: Graph-Oriented ORAM for Efficient Ego-Centric Queries on Federated Graphs. *Proceedings of the VLDB Endowment*, 18(10), 2025.

[FG13] William Futral and James Greene. *Intel Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters*. Springer Nature, 2013.

[FKSW19] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and Private Function Evaluation with Intel SGX. In *CCSW 19*, pages 165–181, 2019.

[FWC+21] Xiaoyu Fan, Guosai Wang, Kun Chen, Xu He, and Wei Xu. PPCA: Privacy-Preserving Principal Component Analysis using Secure Multiparty Computation (MPC). *arXiv preprint arXiv:2105.07612*, 2021.

[GBOW88] S Goldwasser, M Ben-Or, and A Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computing. In *20th ACM STOC*, pages 1–10, 1988.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *10th EuroSec*, pages 1–6, 2017.

[Gil99] Niv Gilboa. Two Party RSA Key Generation. In *CRYPTO'99*, pages 116–129, 1999.

[GJM+23] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. SIGMA: Secure GPT Inference with Function Secret Sharing. *Cryptology ePrint Archive, Report 2023/1269*, 2023.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure Multi-Party Computation without Agreement. *Journal of Cryptology*, 18:247–287, 2005.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *CHES 2001*, pages 251–261, 2001.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play Any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*, pages 218–229, 1987.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In *CRYPTO 2020, Part II*, pages 618–646, 2020.

[HGS25] Tristan Hornetz, Lukas Gerlach, and Michael Schwarz. Lixom: Protecting Encryption Keys with Execute-Only Memory. In *FC 2025*, pages 385–401, 2025.

[HHL+22] Pengzhi Huang, Thang Hoang, Yueying Li, Elaine Shi, and G Edward Suh. STAMP: Lightweight TEE-Assisted MPC for Efficient Privacy-Preserving Machine Learning. *arXiv preprint arXiv:2210.10133*, 2022.

[HLC+22] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private Inference on Transformers. *Advances in Neural Information Processing Systems*, 35:15718–15731, 2022.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO 2003*, pages 145–161, 2003.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-Knowledge from Secure Multiparty Computation. In *39th ACM STOC*, page 21–30, 2007.

[Int17] Intel. Memory Encryption Technologies Specification. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf, 2017.

[Int18] Intel. Intel Software Guard Extensions (Intel SGX) Developer Guide. https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html, 2018.

[Int20] Intel. Intel TDX. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html, 2020.

[Int22] Intel. Refined Speculative Execution Terminology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html, 2022.

[Jac22] Hakon Jacobsen. TEK4500: Authenticated Encryption. https://www.uio.no/studier/emner/matnat/its/TEK4500/h22/notes/authenticated_encryption.pdf, 2022.

[JGB+23] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: FSS-based Secure Training and Inference with GPUs. In *2023 IEEE Symposium on Security and Privacy*, pages 63–63, 2023.

[JXJ+22] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *2022 IEEE Symposium on Security and Privacy*, pages 650–665, 2022.

[Kap17] David Kaplan. Protecting VM Register State with SEV-ES. *White paper, Feb*, 2017.

[Kel20a] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS 2020*, pages 1575–1590, 2020.

[Kel20b] Marcel Keller. MP-SPDZ Library. https://github.com/data61/MP-SPDZ, 2020.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively Secure OT Extension with Optimal Overhead. In *CRYPTO 2015, Part I*, pages 724–741, 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS 2016*, pages 830–842, 2016.

[KPPS21] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-Fast and Robust Privacy-Preserving Machine Learning. In *usenix21name*, pages 2651–2668, 2021.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. In *EUROCRYPT 2018, Part III*, pages 158–189, 2018.

[KPW+19] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *14th EuroSys*, pages 1–15, 2019.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP 08*, page 486–498, 2008.

[KS14] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. In *ASIACRYPT 2014, Part II*, pages 506–525, 2014.

[Lin20] Yehuda Lindell. Secure Multiparty Computation. *Communications of the ACM*, 64(1):86–96, 2020.

[LKFV23] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. SECRECY: Secure Collaborative Analytics in Untrusted Clouds. In *NSDI 23*, pages 1031–1056, 2023.

[LLG23] Xiang Li, Fabing Li, and Mingyu Gao. Flare: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework. *Proceedings of the VLDB Endowment*, 16(6):1439–1452, 2023.

[LLP+19] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-Preserving Remote Deep-Learning Inference using SGX. In *MobiCom 19*, pages 1–17, 2019.

[LLR06] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. On the Composition of Authenticated Byzantine Agreement. *Journal of the ACM*, 53(6):881–917, 2006.

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-Grained Control Flow inside SGX Enclaves with Branch Shadowing. In *USENIX Security 2017*, pages 557–574, 2017.

[LX19] Yi Li and Wei Xu. PrivPy: General and Scalable Privacy-Preserving Data Mining. In *25th KDD*, pages 1299–1307, 2019.

[LZZ+21] Yibiao Lu, Bingsheng Zhang, Hong-Sheng Zhou, Weiran Liu, Lei Zhang, and Kui Ren. Correlated Randomness Teleportation via Semi-trusted Hardware - Enabling Silent Multi-party Computation. In *ESORICS 2021, Part II*, pages 699–720, 2021.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *2nd HASP*, volume 10, 2013.

[MAK+17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security 2017*, pages 1289–1306, 2017.

[ML05] John P McGregor and Ruby B Lee. Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing. *ACM SIGARCH Computer Architecture News*, 33(1):16–26, 2005.

[MR18] Payman Mohassel and Peter Rindal. ABY3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS 2018*, pages 35–52, 2018.

[MRR+15] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security 2015*, pages 865–880, 2015.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, 2017.

[NCR+23] Vikram Narayanan, Claudio Carvalho, Angelo Ruocco, Gheorghe Almási, James Bottomley, Mengmei Ye, Tobin Feldman-Fitzthum, Daniele Buono, Hubertus Franke, and Anton Burtsev. Remote Attestation of Confidential VMs using Ephemeral vTPMs. In *ACSAC 23*, pages 732–743, 2023.

[NLDD23] Deepika Natarajan, Andrew Loveless, Wei Dai, and Ronald Dreslinski. Chex-Mix: Combining Homomorphic Encryption with Trusted Execution Environments for Oblivious Inference in the Cloud. In *2023 IEEE European Symposium on Security and Privacy*, pages 73–91, 2023.

[Pai99] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT'99*, pages 223–238, 1999.

[PGFW14] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N Wright. Systematizing Secure Computation for Research and Decision Support. In *SCN 14*, pages 380–397, 2014.

[PLD+11] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical State Continuity for Protected Modules. In *2011 IEEE Symposium on Security and Privacy*, pages 379–394, 2011.

[PS20] Arpita Patra and Ajith Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning, 2020.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT 2009*, pages 250–267, 2009.

[PSSY21] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security 2021*, pages 2165–2182, 2021.

[PST17] Rafael Pass, Elaine Shi, and Florian Tramer. Formal Abstractions for Attested Execution Secure Processors. In *EUROCRYPT 2017, Part I*, pages 260–289, 2017.

[Rab05] Michael O Rabin. How To Exchange Secrets with Oblivious Transfer, 2005.

[RBS+22] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. SecFloat: Accurate Floating-Point Meets Secure 2-Party Computation. In *2022 IEEE Symposium on Security and Privacy*, pages 576–595, 2022.

[RR21] Mike Rosulek and Lawrence Roy. Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits. In *CRYPTO 2021, Part I*, pages 94–124, 2021.

[RRG+21] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SIRNN: A Math Library for Secure RNN Inference. In *2021 IEEE Symposium on Security and Privacy*, pages 1003–1020, 2021.

[Sha79] Adi Shamir. How To Share a Secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, 1979.

[SJBZ18] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives. *White paper*, page 12, 2018.

[SLKP17] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS 2017*, 2017.

[SS20] AMD Sev-Snp. Strengthening VM Isolation with Integrity Protection and More. *White Paper, January*, 53:1450–1465, 2020.

[STC+20] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking inside a Single Enclave of Intel SGX. In *ASPLOS 20*, pages 955–970, 2020.

[SWG+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *14th DIMVA*, pages 3–24, 2017.

[The23] The Linux Community. tc-netem(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc-netem.8.html, 2023.

[TUP+21] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. Rkt-io: A Direct I/O Stack for Shielded Execution. In *EuroSys 21*, pages 490–506, 2021.

[TZL+17] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *2017 IEEE European Symposium on Security and Privacy*, pages 19–34, 2017.

[VBMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security 2018*, pages 991–1008, 2018.

[VBWK+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security 2017*, pages 1041–1056, 2017.

[WDN+22] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. Engraft: Enclave-Guarded Raft on Byzantine Faulty Nodes. In *ACM CCS 2022*, pages 2841–2855, 2022.

[WLL+22] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. Operon: An Encrypted Database for Ownership-Preserving Data Management. *Proceedings of the VLDB Endowment*, 15(12):3332–3345, 2022.

[WNS+22] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. Hybrid trust multi-party computation with trusted execution environment. In *NDSS 2022*, 2022.

[WWP22] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU Platform for Secure Computation. In *USENIX Security 2022*, pages 827–844, 2022.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[Yao82] Andrew Chi-Chih Yao. Protocols for Secure Computations (Extended Abstract). In *23rd FOCS*, pages 160–164, 1982.

[Yao86] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets (Extended Abstract). In *27th FOCS*, pages 162–167, 1986.

[ZCP+21] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. VeriDB: An SGX-Based Verifiable Database. In *SIGMOD 21*, pages 2182–2194, 2021.

[ZDB+17] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI 17*, pages 283–298, 2017.

[ZDC+21] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. CEREBRO: A Platform for Multi-Party Cryptographic Collaborative Learning. In *USENIX Security 2021*, pages 2723–2740, 2021.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates. In *EUROCRYPT 2015, Part II*, pages 220–250, 2015.

[ZS18] Mark Zhao and G. Edward Suh. FPGA-Based Remote Power Side-Channel Attacks. In *2018 IEEE Symposium on Security and Privacy*, pages 229–244, 2018.

[ZXWG22] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. PPMLAC: High Performance Chipset Architecture for Secure Multi-party Computation. In *ISCA 22*, pages 87–101, 2022.

Protocol 2 $\mathsf{Prot}_{\text{triple}}[\{P_i\}_{i\in\{0,\dots,n-1\}}]$

For each party P_i:

$a^i \leftarrow\$ \mathbb{F}$, $b^i \leftarrow\$ \mathbb{F}$

$\left(a^i_0, \dots, a^i_{k-1}\right) := \mathbf{g}^{-1}(a^i)$ ▷ *bit decomposition* $\mathbf{g}^{-1} : \mathbb{F} \mapsto \{0,1\}^k$

for each P_j, $j \neq i$ **do**

  call $\mathcal{F}^{k,k}_{\text{ROT}}$ with P_j, acting as the receiver with input $\left\{a^i_h\right\}_{h\in\{0,\dots,k-1\}}$, and receiving $\left\{q^{(j,i)}_{a^i_h,h}\right\}_{h\in\{0,\dots,k-1\}}$

  call $\mathcal{F}^{k,k}_{\text{ROT}}$ with P_j, acting as the sender with no input, and receiving $\left\{q^{(i,j)}_{0,h}, q^{(i,j)}_{1,h}\right\}_{h\in\{0,\dots,k-1\}}$

  $\mathbf{q}^{(i,j)} := \left(q^{(i,j)}_{0,0}, \dots, q^{(i,j)}_{0,k-1}\right)$

  for $h \in \{0, \dots, k-1\}$ **do**

    send $d^{(i,j)}_h := q^{(i,j)}_{0,h} - q^{(i,j)}_{1,h} + b^i$ to P_j, receive $d^{(j,i)}_h$ from P_j

    $t^{(j,i)}_h := q^{(j,i)}_{a^i_h,h} + a^i_h \cdot d^{(j,i)}_h = q^{(j,i)}_{0,h} + a^i_h \cdot b^j$

  $\mathbf{t}^{(j,i)} := \left(t^{(j,i)}_0, \dots, t^{(j,i)}_{k-1}\right)$

$c^i := a^i \cdot b^i + \sum_{j\neq i} \left\langle \mathbf{g}, \mathbf{t}^{(j,i)} \right\rangle + \left\langle \mathbf{g}, \mathbf{q}^{(i,j)} \right\rangle$

A MPC Protocols and Relevant Primitives

The Semi protocol [Kel20a] is secure in the dishonest majority setting. Each party $P_i, i \in \{0, \dots, n-1\}$ holds shares $[x]_i, [y]_i$ such that $x = \sum_{i=1}^n [x]_i$ and $y = \sum_{i=1}^n [y]_i$. The addition gate thus only involves local computation of $[x]_i + [y]_i$. For multiplications, it relies on pre-generated Beaver triples [Bea92]. Each Beaver triple is denoted as $([a], [b], [c])$ where $c = a \cdot b$. Each triple is consumed only once by a multiplication gate during the online phase. Specifically, for the kth multiplication, all parties locally compute $[x + a_k], [y + b_k]$ and send the shares to one party, who reconstructs $x + a_k$ and $y + b_k$ and broadcasts them to all parties. Then all parties can use them to compute $[x \cdot y] = (x + a_k)(y + b_k) - (x + a_k)[b_k] - (y + b_k)[a_k] + [c_k]$.

Oblivious transfer (OT) typically involves a sender and a receiver [Rab05, Bea96, IKNP03, KOS15], where for a set of the sender's messages, the receiver obtains one of them while the sender does not know which one. The functionality for 1-iteration 1-out-of-2 OT with k-bit sender's inputs s_i and 1-bit receiver's input b can be formalized as $\mathcal{F}^{1,k}_{\text{OT}} : ((s_0, s_1), b) \mapsto (\perp, s_b)$. For the random OT variant where the input messages are randomly sampled [CO15], we formalize it as $\mathcal{F}^{1,k}_{\text{ROT}} : (\perp, b) \mapsto ((r_0, r_1), r_b)$.

With additive homomorphic encryption (AHE) [CF85, Pai99], given only the public key pk and the encryption of messages m_0 and m_1 (i.e., $\mathsf{Enc}_{\mathsf{pk}}(m_0)$ and $\mathsf{Enc}_{\mathsf{pk}}(m_1)$), one can compute $\mathsf{Enc}_{\mathsf{pk}}(a \cdot m_0 + b \cdot m_1)$ without the private key sk, where a and b are public values.

Finally, Protocol 2 shows a protocol that relies on OT to generate Beaver triples [KOS16].

B Universal Composition Models

B.1 Plain Universal Composition (UC)

The Universal Composition (UC) framework enables modular security analysis of complex cryptographic systems. In this model, a protocol can be decomposed into subroutines, each of which can be analyzed independently by proving that it realizes an associated ideal functionality $\mathcal{F}$. The Universal Composability Theorem then guarantees—informally speaking—that if a higher-level protocol π makes subroutine calls to an ideal $\mathcal{F}$, its security

is preserved when $\mathcal{F}$ is replaced by any actual protocol ρ that securely realizes $\mathcal{F}$. This supports a modular, plug-and-play approach to protocol design.

Security in the UC framework is defined via the ideal–real paradigm. A protocol π is said to be secure if, for any adversary $\mathcal{A}$ attacking π in the real world, there exists a simulator $\mathcal{S}$ that can generate a simulated attack in the ideal world—where parties interact only with the trusted ideal functionality $\mathcal{F}$—such that no external observer can distinguish between the two executions.

To capture arbitrary external influences, including concurrent executions of other protocols, the model introduces an adversarial environment $\mathcal{Z}$. This environment provides inputs to all parties, observes their outputs, and may interact with the adversary to coordinate corruptions. The protocol π UC-realizes $\mathcal{F}$ if, for every polynomial-time adversary $\mathcal{A}$ in the real world, there exists a simulator $\mathcal{S}$ such that no probabilistic polynomial-time (p.p.t.) environment $\mathcal{Z}$ can distinguish between the execution of π under $\mathcal{A}$, and the ideal execution of $\mathcal{F}$ under $\mathcal{S}$. Formally:

$$\forall\, \mathcal{A},\ \exists\, \mathcal{S},\ \forall\, \mathcal{Z} : \mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$$

where $\approx$ denotes computational indistinguishability.

By the UC Composition Theorem [Can01], any protocol that uses $\mathcal{F}$ as a subroutine retains its security when $\mathcal{F}$ is replaced by a protocol ρ that UC-realizes it—even in arbitrary concurrent compositions.

B.2 Generalized Universal Composition (GUC)

Many useful cryptographic functionalities (e.g., public-key encryption, digital signatures, zero-knowledge proofs with trusted setup) cannot be realized in the plain UC model without additional assumptions. To address this, hybrid models are often employed, where protocols are allowed access to an ideal setup functionality $\mathcal{G}$ (e.g., a common reference string or a certificate authority).

In the standard UC framework, such setup functionalities are local to a specific protocol instance. The environment $\mathcal{Z}$ can only interact with $\mathcal{G}$ through the adversary $\mathcal{A}$ (or the simulator $\mathcal{S}$). This limitation prevents accurate modeling of globally available trusted infrastructure—such as Public Key Infrastructures (PKIs), trusted hardware (e.g., TPMs), or blockchain consensus—which are simultaneously accessible to multiple independent protocols across the system.

To address this shortcoming, Canetti et al. [CDPW07] introduced the Generalized Universal Composition (GUC) framework. GUC extends UC to support global setup functionalities $\mathcal{G}$ that are accessible to all protocol instances and parties in the system, including the environment $\mathcal{Z}$.

The key technical difference lies in the interaction model: in GUC, the environment $\mathcal{Z}$ can directly query $\mathcal{G}$ without going through the adversary or simulator. Consequently, the setup functionality must be non-programmable from the simulator's perspective: $\mathcal{S}$ cannot arbitrarily choose or manipulate the internal state of $\mathcal{G}$ (e.g., secret keys), as such choices would be observable by $\mathcal{Z}$ and break indistinguishability.

C Term Definitions

The Decisional Diffie-Hellman (DDH) problem: Given a group G, a generator g of G, and three elements $a, b, c \in G$, decide whether there exist integers x, y such that $a = g^x$, $b = g^y$, and $c = g^{xy}$.

The Decisional Diffie-Hellman (DDH) assumption: Any probabilistic polynomial time algorithm solves the DDH problem only with negligible probability.

Authenticated encryption scheme [Jac22]: A tuple of polynomial time algorithms $\mathsf{AE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, where:

- KGen is a probabilistic key generation algorithm that outputs a key in a key space $\mathcal{K}$.
- Enc: $\mathcal{K} \times \mathcal{M} \mapsto \mathcal{C}$ is a probabilistic encryption algorithm that takes as input a key in $\mathcal{K}$ and a message from a message space $\mathcal{M}$, and outputs a ciphertext in a ciphertext space $\mathcal{C}$.
- Dec: $\mathcal{K} \times \mathcal{C} \mapsto \mathcal{M} \cup \{\perp\}$ is a deterministic decryption algorithm which takes as input a key in $\mathcal{K}$ and a ciphertext in $\mathcal{C}$, and outputs either a message in $\mathcal{M}$ or an error symbol $\perp$.

Digital signature scheme: A triple of polynomial time algorithms, $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$, where:

- KGen (key-generator) generates a public key pk, and a corresponding private key sk, on input 1^n, where n is the security parameter.
- Sign (signing) takes as input the private key sk and a message $\mathbf{x}$, and returns a tag t.
- Vf (verifying) takes as input the public key pk, a message $\mathbf{x}$, and a tag t, and outputs "accepted" or "rejected".

Key derivation function [CDS13]: A cryptographic algorithm $K \leftarrow \mathsf{KDF}(p, s, c, n)$, where:

- p is a private string, whose length is chosen from the space of all possible private strings, `PSPACE`.
- s is a salt, a public random string chosen from the salt space `SSPACE`.
- c is a public context string chosen from a context space `CSPACE`.
- n is a positive integer that indicates the number of bits to be produced by KDF.
- K is the derived n-bit cryptographic key.

D The Offline Protocol with Filmy Enclaves

Protocol 3 is an OT-based implementation for Functionality 4. Compared to Protocol 1, Protocol 3 is simpler. The offline protocol does not need to load any prepared data. Moreover, since the goal is to generate a quantity of independent Beaver triples, the circuit $\mathcal{C}$ should consist of a set of multiplication gates and no intermediate results need to be recovered from. The procedure of crash recovery can be simplified. If a crash occurs, we only need to refresh the committee. Once the Beaver triples generated by the previous offline committee run out during the online phase, the online committee can negotiate to switch the $eids_{\text{pre}}$ and $base_{\text{pre}}$. The procedure is similar to "recover" in Program 1. For simplicity, we omit the details since we have assumed that the Beaver triples generated by one committee are enough. In Protocol 3 and Program 3, we reuse procedures such as `AttestAll` and `ExDataWith`. The main part of "compute" incorporates Protocol 2.

Protocol 3 $\mathsf{Prot}\left[\mathcal{C}, \{P_i\}_{i\in\{0,\ldots,n-1\}}\right]$

For party P_i: on input "setup" from environment $\mathcal{Z}$:

$eid_i := \mathcal{G}_{\text{TEE}}.\text{install}(\mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
$eids := \textsc{AttestAll}(\{0,\ldots,n-1\}\setminus\{i\}, i, eid_i, \mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
$\mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"init"}, i, eids)$

For party P_i: on input "recover" $\{P_k\}_{k\in K}$ from environment $\mathcal{Z}$:

if $i \in K$ **then**
  $eid_i := \mathcal{G}_{\text{TEE}}.\text{install}(\mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
  $eids := \textsc{AttestAll}(\{0,\ldots,n-1\}\setminus\{i\}, i, eid_i, \mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
else
  $eids := \textsc{AttestAll}(K, i, eid_i, \mathsf{prog}[\mathcal{C}, \{P_i\}_i])$
$\mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"init"}, i, eids)$

For party P_i: on input "compute" from environment $\mathcal{Z}$:

$res_i := \mathcal{G}_{\text{TEE}}.\text{resume}(eid_i, \text{"compute"})$
persist res_i ▷ *sealing Beaver triple*

Program 3 $\mathsf{prog}[\mathcal{C}, \{P_i\}_{i\in\{0,\ldots,n-1\}}]$

On input ("init", i, $eids$):

keep i, $eids$, $offset := 0$ in enclave
initialize $eids_{\text{tmp}} := \{eid_i\}$
for $eid_j \in eids, j \neq i$ **do**
  $(j, eid_j, eid^j_{\text{pre}}) := \textsc{ExDataWith}(i, j, (i, eid_i), eids))$
  $eids_{\text{tmp}}.\text{insert}(eid_j)$
assert all j's appear, $eids = eids_{\text{tmp}}$

On input ("compute"):

retrieve $eids$, $offset$ in enclave
$a^i := \mathcal{G}_{\text{TEE}}.\text{randomgen}(k)$, $b^i := \mathcal{G}_{\text{TEE}}.\text{randomgen}(k)$ ▷ *assume the online input has k bits*
$\left(a^i_0,\ldots,a^i_{k-1}\right) := \mathbf{g}^{-1}(a^i)$
for each eid_j, $j \neq i$ **do**
  call $\mathsf{prog}_{\text{ROT}^{k,k}}[eid_j, eid_i]$ with enclave eid_j, acting as the receiver with input $\left\{a^i_h\right\}_{h\in\{0,\ldots,k-1\}}$, and receiving $\left\{q^{(j,i)}_{a^i_h,h}\right\}_{h\in\{0,\ldots,k-1\}}$
  call $\mathsf{prog}_{\text{ROT}^{k,k}}[eid_i, eid_j]$ with enclave eid_j, acting as the sender with no input, and receiving $\left\{q^{(i,j)}_{0,h}, q^{(i,j)}_{1,h}\right\}_{h\in\{0,\ldots,k-1\}}$
  $\mathbf{q}^{(i,j)} := \left(q^{(i,j)}_{0,0},\ldots,q^{(i,j)}_{0,k-1}\right)$
  for $h \in \{0,\ldots,k-1\}$ **do**
    $d^{(i,j)}_h := q^{(i,j)}_{0,h} - q^{(i,j)}_{1,h} + b^i$
    $d^{(j,i)}_h := \textsc{ExDataWith}(i, j, d^{(i,j)}_h, \_)$
    $t^{(j,i)}_h := q^{(j,i)}_{a^i_h,h} + a^i_h \cdot d^{(j,i)}_h = q^{(j,i)}_{0,h} + a^i_h \cdot b^j$
  $\mathbf{t}^{(j,i)} := \left(t^{(j,i)}_0,\ldots,t^{(j,i)}_{k-1}\right)$
$c^i := a^i \cdot b^i + \sum_{j\neq i} \left\langle \mathbf{g}, \mathbf{t}^{(j,i)} \right\rangle + \left\langle \mathbf{g}, \mathbf{q}^{(i,j)} \right\rangle$
$offset := offset + 1$
$seal\_tri := \mathcal{G}_{\text{TEE}}.\text{seal}(eid_i, ((a^i, b^i, c^i), offset, eids), 1)$
return $seal\_tri$