gcVM: Publicly Auditable MPC via Garbled Circuits with Applications to Private EVM-Compatible Computation

Avishay Yanai, Meital Levy, Hila Dahari-Garbian, Mike Rosulek
Soda Labs

Abstract—**Blockchains have achieved substantial progress in scalability and fault tolerance, yet confidentiality remains an important challenge. Existing zero-knowledge (ZK) solutions provide partial privacy guarantees but have poor performance and composability, especially for computations involving the private state of many participants. In this work, we introduce gcVM, a novel extension to the Ethereum Virtual Machine (EVM) that integrates garbled-circuit-based secure multi-party computation to enable general-purpose, privacy-preserving computation on-chain. gcVM allows transactional interactions between untrusted parties, balancing the transparency of public blockchains with strong confidentiality. Our implementation demonstrates up to 83 confidential transactions per second (cTPS) on standard cloud instances, with projected enhancements expected to scale throughput to approximately 500 cTPS—two to three orders of magnitude faster than comparable FHE-based solutions. gcVM is compatible with existing EVM tooling, provides public auditability, and requires no trusted hardware, offering a practical and efficient platform for privacy-centric blockchain applications across finance, governance, and decentralized services.**

1 Introduction

Blockchains, the Ethereum Virtual Machine, and the problem of confidentiality. Blockchains were originally designed to prioritize availability and transparency, with significant progress made in scalability through innovations like rollups and advancements in Byzantine Fault Tolerance (BFT) protocols.

Our focus on this work is the Ethereum Virtual Machine (EVM). Unlike Bitcoin, the EVM provides a Turing complete, stateful execution environment that enables complex programmable interactions between users and smart contracts. This expressiveness makes the EVM the natural setting for exploring advanced cryptographic techniques that go beyond payment anonymity, such as privacy-preserving computation, anonymous credentials, and secure state updates.

An important fact about EVM for our purposes is that the entire state of all smart contracts (including financial and social information) is **public.** On the one hand, this fact played a key role in contributing to the *decentralization* of Ethereum (who is the first to deploy an EVM), as it allows anyone to contribute computational resources and to verify that the EVM state is the correct result of past transactions. On the other hand, the lack of confidentiality presents a barrier to the deployment of many useful applications of blockchain technology. For example, elections, sealed-bid auctions, lending/borrowing, deposit management, and OTC markets all involve information that must crucially remain confidential.

1.1. Existing Approaches & Limitations

Several cryptographic technologies have been proposed to address the challenge of confidentiality for blockchains.

Zero-knowledge proofs (ZK). ZK-based systems achieve privacy by committing to data on-chain and proving correctness without revealing the underlying inputs. This technique is highly efficient and provides public verifiability and succinctness, but its main drawbacks are that:

- The prover must know all inputs in plaintext, so the technique is limited to computations that involve private inputs from *just one* party. This makes it difficult for different smart contracts to interact.
- They typically rely on a trusted setup.

For example: ZCash [BCG$^+$14] and Monero [Mon23] use zk-SNARKs and ring signatures, respectively, to provide transactional anonymity but lack general smart-contract capability. Zexe [BCG$^+$20] and VeriZexe [XCZ$^+$23] extend this paradigm to decentralized private computation (DPC), supporting off-chain computation with succinct on-chain proofs. Hawk [KMS$^+$16] introduces privacy-preserving contracts via a trusted manager, later replaced by MPC in zkHawk [BCT21] and V-zkHawk [BT22].

Fully homomorphic encryption (FHE). FHE-based [Gen09] solutions allow parties to carry out arbitrary computations on encrypted data. Examples in the blockchain space include fheVM [Zam], Zether [BAZB20], Zkay [SBG$^+$19], SmartFHE [SWA23], and PESCA [Dai22]. FHE offers strong theoretical guarantees, but suffers from several limitations:

- Even with significant engineering improvements, FHE is slow. For example, fheVM supports roughly 2–3 confidential transactions per second on AWS's most powerful instances. [1]

- FHE-based solutions generally do not provide cryptographic *agility*, which is defined as the ability to easily replace cryptographic components if some are found to have weaknesses. FHE solutions and their correpsonding optimizations are intrinsically tied to a highly specific choice of cryptographic algorithm, at a specific security level. For example, fhEVM is based on the TFHE construction.

Trusted execution environments (TEE). TEE-based systems perform computation within secure enclaves that hold decryption keys, storing only encrypted data on-chain. Examples include Oasis [Fou23] and Phala [Net23]. Unlike cryptographic solutions, TEEs have almost no computational overhead. However,

- Trust in hardware is riskier than trust in cryptography. All known TEEs are vulnerable to side-channel attacks (SCAs).
- If secrets are stored in a single TEE, there is a single point of failure for availability. If secrets are stored in several TEEs, there are many points of failure for privacy.

Secure multiparty computation (MPC). MPC-based systems enable multiple parties to jointly compute contract logic over private data, via an interactive protocol. Solutions like zkHawk [BCT21] and V-zkHawk [BT22] eliminate the trusted manager in Hawk by using MPC protocols. Eagle [ByCDF23] further adds identifiable abort and public verifiability These approaches provide strong privacy even under dishonest-majority settings and are well-suited for decentralized input ownership. However, all previous approaches use secret-sharing-based MPC techniques, which have certain limitations.

- They require many sequential rounds of computation (proportional to the circuit-depth of the computation). This results in high latency for deep computations, as well as challenges inherent in having many protocol synchronization points.
- Existing MPC-based solutions have their network structure "baked in," meaning that clients must know the identities of the computing nodes.

1.2. Our Contributions

In this work, we introduce **gcVM**, a novel extension to the Ethereum Virtual Machine (EVM) that adds the ability to store private data on-chain and enable privacy-preserving smart contracts. gcVM is the first such system based on garbled circuits [BHR12], a classic technique from secure multi-party computation.

The current stable version of the gcVM has been deployed on a public Testnet and is scheduled for Mainnet deployment following a security audit and an expansion of the MPC node network. Even without some of the optimizations introduced in this work, the system demonstrates notable performance, achieving up to 83 confidential transactions per second (cTPS) on basic Amazon EC2 instances under moderate bandwidth conditions using single-threaded MPC nodes. This result indicates substantial room for further performance gains. Notably, one major enhancement described in this work, referred to as "offline soldering," is expected to scale the gcVM to approximately 500 cTPS, with additional improvements such as parallel evaluation projected to increase throughput even further. For context, a comparable commercial system based on Fully Homomorphic Encryption (FHE) achieves only 2–3 cTPS on Amazon's most powerful machines, implying that gcVM can offer two to three orders of magnitude higher performance in similar environments.

Our approach enjoys the following features:

Modularity. Our garbled-circuit-based solution consists of two distinct and independent phases. The Garbling phase involves significant computation by the network nodes (the 'garblers') and is conducted offline in a pre-processing stage, producing a garbled circuit—a secure one-time container for data processing. This phase continually generates garbled circuits for subsequent use during the Evaluation phase, where actual transactions are processed. The Evaluation phase is executed by the network nodes in an efficient manner. The Garbling phase can be instantiated in a variety of ways, leading to a modular 'privacy supply chain' that the Evaluation phase can consume.

Security and Cryptographic Agility. One design principle behind gcVM is to align with industry cryptographic standards right from the start, rather than introducing a proprietary, yet-to-be-standardized encryption and zero-knowledge schemes. In gcVM, private transaction data can be encrypted by any standard, symmetric-key encryption scheme (e.g., AES-GCM). Thus, we inherit the high assurance that these heavily scrutinized algorithms have enjoyed. Our approach is relatively agnostic to the specific choice of encryption. Instantiations are available in many key lengths and block lengths, positioning them as post-quantum ready.

Apart from conventional, off-the-shelf encryption schemes, the gcVM requires only an abstract garbling scheme [BHR12], which in turn can be instantiated from any standard block cipher (e.g., AES-128). An advantage of encrypting transaction data using standard symmetric-key encryption is that it is relatively practical to incorporate encryption/decryption logic within a garbled circuit (because standard symmetric-key algorithms have relatively small boolean circuits).

Arbitrary transaction logic. By using general-purpose MPC techniques, applied to data encrypted with conventional symmetric-key techniques, our approach can handle computations over private data of any users. Unlike ZK-based approaches, there is no requirement that a prover knows all of the private data for a single transaction. This feature is vital for a range of blockchain applications, from dynamic identity systems and DeFi applications like AMM to portfolio management, social trading, auctions, governance, and more.

Public auditability. A special requirement that may be crucial in many settings is ensuring that computations—even on ciphertexts—are publicly auditable, thereby mitigating the risk of manipulation or theft

1. Based on numbers provided in https://docs.zama.org/tfhe-rs/get-started/benchmarks/cpu/cpu-integer-operations, assuming a modest transaction involving the following atomic operations: Comparison, for comparing the payer's balance with the payout amount; addition, to increase the payee's balance; subtraction, to decrease the payer's balance; and a ZKPoK verification to authenticate the payer.

through 'silent collusion' among all parties. In practical terms, any state-transition is deemed invalid if it fails public audit. Garbled circuits pose a natural candidate to satisfy that requirement, as the garbled circuits themselves are public, and it only remains for the auditor to ensure that garbled inputs are obtained correctly.

Performance. We achieve high performance by separating the steps involved in garbled-circuit-based MPC into two phases. Garblers continuously produce a supply of garbled circuits for a selection of atomic operations (e.g., `ADD64`, `MULT64`, `LEQ`). This is the offline garbling phase. Then, when a transaction arrives, evaluators quickly assemble these already-garbled circuits together into one unified circuit representing the entire transaction. In this way, we maximize the amount of effort that can be done before the transaction is known.

Only the garbled evaluation needs to happen at the time of a transaction. Both the evaluation step and the "stitching" (soldering) steps require a small constant number of communication rounds among nodes, which does not depend on the number of parties involved or the complexity of the transaction.

User and developer experience. Users interact with the gcVM system in an extremely simple way. Private data is provided by simply encrypting it under a standard symmetric-key scheme (e.g., AES-GCM) and sending the resulting ciphertext to the network. Transactions then simply refer to [the contents of] these ciphertexts. All other cryptography (i.e., the garbled circuits and other primitives) is handled exclusively by the network nodes.

As a result, a gcVM client can be implemented using only cryptography that is included in `openssl`. This makes seamless deployment possible on a wider range of devices and platforms. As a point of comparison, clients in fhEVM must compute and send a zero-knowledge proof of knowledge.

Additionally, gcVM is built on EVM, the most popular environment for smart contracts. We can therefore take advantage of all the existing tooling and workflows for EVM contracts. The vast majority of FHE-based solutions, for example, rely on alternative, proprietary execution environments [Par21], [Sec], [Oas].

1.3. Other Related Work

Public verifiability / covert security in MPC. The notion of *public verifiability* and *covert security* in secure multi-party computation (MPC) has been studied extensively in recent years. Damgård, Orlandi, and Simkin [DOS20] introduced a general transformation from arbitrary passively secure preprocessing protocols into protocols that achieve covert security with public verifiability, while maintaining the same corruption threshold. Their construction employs time-lock puzzles to enable delayed opening of protocol commitments, thereby allowing external verification of protocol correctness. Although their primary focus is on the two-party setting, they also outline how their approach can be extended to multi-party computation. Following this line of work, Fischlin et al. [FHKS21] presented a generic compiler that converts covertly secure MPC protocols into publicly verifiable ones. Their compiler leverages time-lock encryption to ensure that the probability of detecting cheating (often referred to as the deterrence factor) remains high and independent of the number of participating parties.

A related strand of research studies publicly verifiable two-party computation (2PC). Works such as [KM15], [HKK+19], [ZDH19], [DOS20] propose publicly checkable cut-and-choose techniques for 2PC based on garbled circuits, enabling third parties to verify correctness without compromising privacy.

Beyond covert and two-party settings, several works have explored *auditable* and *publicly verifiable MPC*. Public verifiability, as introduced in [BDO14], [SV15], allows any external observer to verify the correctness of outputs based solely on publicly available data, such as values posted to a public bulletin board. Baum et al. [BDO14] and Schoenmakers and Veeningen [SV15] developed frameworks for publicly auditable MPC, highlighting the importance of external accountability in distributed computation. More recently, Baum, Orlandi, Scholl, and Simkin [BOSS20] presented the first concretely efficient, constant-round MPC protocols achieving identifiable abort and public verifiability in the dishonest-majority setting. Their protocols assume static corruptions and rely on broadcast and bulletin board functionalities. In both the identifiable abort and publicly verifiable variants, their constructions achieve constant-round communication, counting broadcast or bulletin-board access as a single round.

Related work on garbling. The garbled circuits technique was first proposed by Yao [Yao86] and later formalized by Lindell and Pinkas [LP09] and Bellare, Hoang and Rogaway [BHR12]. We use a technique called *soldering*, whereby components are garbled before the final circuit (transaction) is known, and then later assembled into a unified garbled circuit. The technique was pioneered by Nielsen and Orlandi [NO09] and refined in a series of works [FJN+13], [FJNT15], [KNR+17], [ZH17].

Paper structure. We provide an overview of the network architecture, the threat model, and the technical concepts behind the garbled circuit-based MPC framework in Section 2; we provide a formal treatment of the MPC framework and a security analysis in Sections 3-4 and 5, respectively; and finally report about the current integration with the go-ethereum client and the performance of the network in Section 6.

2 Technical Overview and Threat Model

2.1. Architecture Overview

The network is depicted in Figure 1. Transactions originate from clients (aka EOAs) and pass through the *sequencing* sub-network, which produces a *non-canonical block* (appears in red). A non-canonical block is simply a container of transactions intended to be used internally only and does not change the blockchain's state. Transactions in that block only pass through basic validation like signature verification, gas consumption limit, etc.

The non-canonical block is then passed to the *execution* sub-network, which iterates over the transactions in the non-canonical block, executes them one-by-one, and produces a *canonical* block. The canonical block is then published and declared as the new blockchain 'head', and contains claims about the changes in the blockchain's state. The execution sub-network is composed of the usual EVM execution engine as well as an integration with garbled circuits evaluator nodes to handle privacy preserving computation workloads. These evaluator nodes continuously receive garbled circuit materials from the garblers. Separating sequencing from execution is inherent to privacy-preserving chains, which stems from a 'chicken-and-the-egg' problem, as explained in Appendix A.

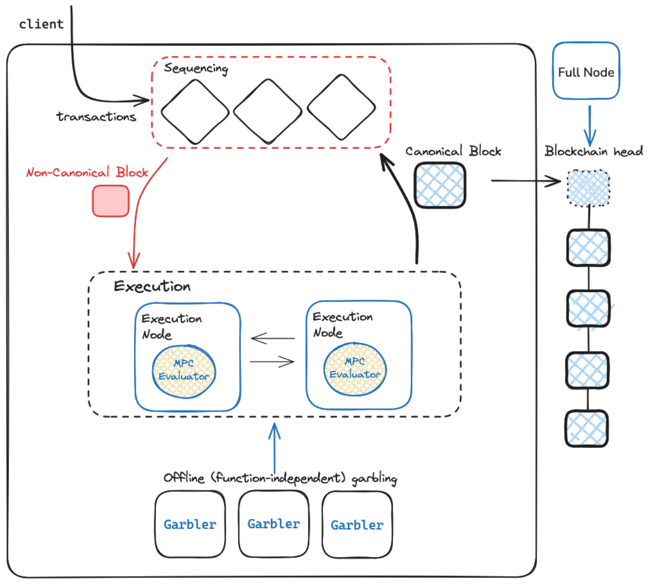


Figure 1. High-level architecture of the gcVM Network.

The gcVM extends the instruction set of the EVM with an analogous set of instructions for secure computation. These instructions receive encryptions as input and produce an encryption as an output, namely, an instruction for securely computing the function $y = f(x_1, x_2, \dots, x_n)$ receives n ciphertexts $\mathsf{ct}_1, \dots, \mathsf{ct}_n$ and outputs a ciphertext ct_y, s.t. decryption of ct_y is equal $f(\mathsf{pt}_1, \dots, \mathsf{pt}_n)$ where pt_i is the decryption of ct_i. To achieve that without having to trust any single party, the execution sub-network runs an MPC protocol that ensures that plaintexts are not revealed (unless this was the intention of the client).

The evaluation nodes collectively manage one global symmetric encryption 'network key' and one symmetric encryption 'user key' per user (EOA), such that all keys are secret shared among themselves. The user key is distributively generated on demand by the evaluation nodes, upon a user-onboard transaction. The network key is used to encrypt the blockchain's private shared state whereas the user key is used to bring in new arguments to the gcVM and to return results to specific users, according to the smart contract logic.

Users can bring in new ciphertexts as well as invoke confidential computing on existing ones by sending a transaction to the gcVM. The lifetime of a transaction involves handling different forms of encrypted values, as depicted in Figure 2 and explained below.

A client can send a transaction that potentially contains encrypted arguments. Such argument is encrypted using the client's symmetric encryption key, and is individually signed. The signature is applied to the encryption as well as other metadata, consisting of the destination contract and method. This facilitates a protection against ciphertext theft attacks by ensuring that only the destination contract and method may process that ciphertext. Collectively, the ciphertext and the signature are referred to as 'input-text'.

At execution time, each input-text's authenticity is first validated, meaning that the gcVM verifies that the input-text is signed by the same transaction sender's signing key, and the current contract and method match those it signed on. The transaction reverts if that validation fails.

After validation, the ciphertext is passed through a decryption garbled circuit, which results in a garbled-text. That circuit is given as inputs the ciphertext and the appropriate user symmetric key, and performs the decryption operation. To that end, both the ciphertext and the user symmetric key are transformed, by the evaluator MPC nodes, into labels that suit the decryption garbled circuit. This is done in a sub-protocol that resembles an oblivious transfer (OT), except that here the evaluators simulate both the sender and the receiver. At the end of that sub-protocol, the evaluators have a single label for each input wire of the decryption garbled circuit, and are ready to locally evaluate.

Note that the plaintext encrypted under the input-text and the garbled-text is the same, except that garbled-text encryption format makes the encrypted data amenable for secure computation by subsequent garbled circuits. The garble-text represents a plaintext by a set of wire labels, in a way that the labels do not disclose anything about that plaintext.

The required computation, which is specified by the contract's logic, is conducted inside an environment called 'Garbled Execution Environment' (GEE), which ensures that private data cannot leak from one transaction execution sandbox to another. The GEE can ensure such isolation thanks to the unpredictability of garbled-texts; that is, the fact that as long as the labels for the input wires of a garbled circuit are not disclosed, the labels for its output wires are unknown too. Therefore, one cannot 'inject' a valid garbled-text to the execution of the GEE.

When execution is completed, depending on the contract logic, it can either decrypt a garbled-text, encrypt it to a certain user (EOA) or to the network. While decrypting a garbled-text only requires revealing the true meaning of the labels obtained for the output wires, encrypting it to a user or the network requires passing that garbled-text through an encryption garbled circuit (with the user's or the network's key, respectively). If encrypted to the network, that data joins the private shared state and can be reused in future transactions, in case that contract is being called again.

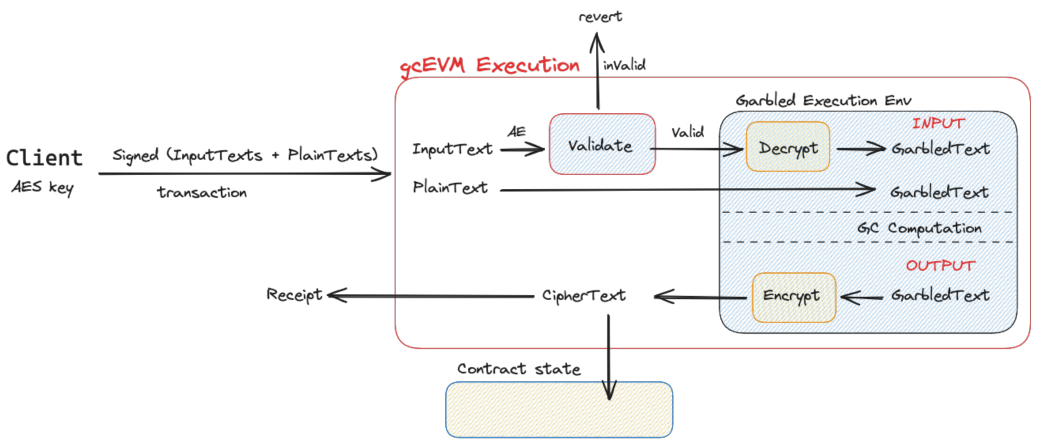


Figure 2. Transaction flow in the gcVM.

2.2. The gcVM's threat model

In the heart of our MPC protocol lies a *garbling scheme*. With great simplicity, this is a cryptographic scheme that is given a boolean circuit f[2], and outputs a tuple F, e, d where F is the 'garbled' version of f, the encoding e maps input wire w_i to two possible labels $L_{i,0}$ and $L_{i,1}$ representing the bits 0 and 1; similarly, the decoding d maps the labels $L_{j,0}$ and $L_{j,1}$ to the bits 0 and 1, for every output wire w_j. In an analogy to f, which carries one out of two bits over each of its wires, F carries one out of two possible labels over each wire. *Correctness* of a garbling scheme requires F to 'mimic' f; that is, if f has n and m input and output wires (resp.), evaluating F on labels $L_{1,b_1}, \ldots, L_{n,b_n}$ (one label per input wire) results in labels $L_{1,b'_1}, \ldots, L_{m,b'_m}$ (one label per output wire) s.t. $f(b_1, \ldots, b_n) = (b'_1, \ldots, b'_m)$. *Privacy* of a garbling scheme requires that the labels used in the evaluation of F do not leak the bits they represent.

A typical garbled circuit-based secure computation protocol consists of one garbler and one evaluator. Given a circuit f, the garbler produces a suitable (F, e, d) as above, and hands the evaluator F and d, as well as one label per input wire, that is, the label that represents the actual input bit.[3]

In our setting, we abstract out the two roles and end up with a set of garblers G and a set of evaluators E, where $1 \leqslant |G|$ and $2 \leqslant |E|$. Similarly, we define two threshold parameters $0 \leqslant t_G < |G|$ and $0 \leqslant t_E < |G|$ that specify the maximal number of garblers and evaluators that the adversary is assumed to corrupt. With that abstraction we can achieve the following guarantees:

- *Privacy* holds as long as the adversary corrupts less than t_E evaluators and less than t_G garblers.
- *Public auditability* guarantees that correctness *always* holds, even when all parties collude.

The latter property is crucial for a blockchain-based solution, as the computation is typically delegated to a set of nodes as a service. Public auditability enables 3rd parties to make sure that, even in extreme cases where privacy breaks, that service could not steal funds from its users.

The new threat model provides more flexibility in deployments. The idea is that the garblers can, in principle, use a one-directional communication channel to the internet, thereby significantly reducing their attack surface. This is because the garblers only need to send out the garbled circuits they produce, without receiving any input from the evaluators (recall that the evaluators simulate the OT protocol among themselves).

2.3. Key concepts of the MPC protocol

The above allows for various possible instantiations of a GC-based MPC protocol, depending on the number of garblers, evaluators, and the threshold parameters. In the following we list the key concepts that are common to most settings.

Transaction-independent garbling via online and offline soldering. In contrast to the two-party protocol described above, in our setting where $2 \leqslant |E|$, the garblers' work is *completely independent of the online computation* – the tuple (F, e, d) is produced and handed over to the evaluators such that F is handed in the clear whereas e and d are secret shared among them. This way the evaluators do not need to interact with the garbler(s) anymore and can simulate the OT protocol among themselves (in order to obtain the correct label for each input wire of F). That is, the two labels $L_{i,0}$ and $L_{i,1}$ for every input (and output) wire w_i are secretly shared between evaluators. When it is time to evaluate that circuit, the evaluators conduct a lightweight 2-rounds protocol in which they obtain L_{i,b_i}, where b_i is the real input bit to enter w_i. The evaluators can do that even when b_i is held in a secret shared form.

To enable meaningful computation by the evaluators, the garbler(s) continuously produce garbled circuits for each of the EVM's instructions (`ADD64, MUL64, EQ64,` etc.) and hand them over to the evaluators, as described above. Upon receiving a transaction, described as a sequence of EVM instructions, the evaluators consume the precomputed garbled circuits. However, in order to pass value from the output wire w_i of one circuit to the input wire w_j of the subsequent circuit, the evaluators

2. We focus on boolean circuits here but the same holds for arithmetic circuits.

3. When this bit is a private input to the evaluator, an oblivious transfer (OT) protocol enables the evaluator to obtain the appropriate label without disclosing to the garbler the actual input bit.

conduct an *online soldering* protocol. That is, this enables the evaluators to translate label L_{i,b_i} obtained on output wire w_i to label L_{j,b_j} on input wire w_j, such that $b_i = b_j$. Note that L_{i,b_i} and L_{j,b_j} encode the same value b and so the computation continues as if both garbled circuits were connected in the first place.

The online soldering incurs communication rounds between the execution of different circuits. An optimized version, we call *offline soldering*, performs all needed soldering *in one shot*, thereby, spending a fixed number of communication rounds to solder many circuits at the same time instead of per circuit, which accounts to a huge performance improvement as the whole transaction (or even the whole block) can be securely computed following a fixed number of communication rounds. See Appendix B for an illustration.

Encryption scheme and key-management. The system relies on symmetric key encryption for secrecy (and on digital signatures for authenticity). That means that the network maintains a secret key k_N as well as a secret key k_U per user U, where each key is secret shared among the evaluators. To input an encrypted argument $\texttt{ct} = \texttt{Enc}(k_U, m)$, the user U associates a signature σ on ct and some metadata that specifies the exact contract and method to which this argument is intended (to prevent a 'ciphertext theft'). Then, when executed, the signature is verified against the sender's public key as well as the contract and method being executed, and then ct and k_U are entered as inputs to a garbled circuit for 'secure decryption'; meaning that the result of the decryption $m = \texttt{Dec}(k_U, \texttt{ct})$ remains in labels format, also called 'garbled-text'. This garbled-text is readily available to be soldered to other garbled circuit for further computation.

Public auditability. For public auditability, the protocol has to ensure the following:

1) The garbled circuits produced by the garbler(s) are correct, namely, they compute the intended function and nothing else.
2) Given correct garbled circuits, the evaluators evaluate them correctly. That is, they reveal and use the correct labels for all input wires of the gabrbled circuits they evaluate.

We now explain how the above items are addressed:

1) The first requirement can be satisfied cryptographically by a cut-and-choose-based protocol [KM15], [HKK+19], [ZDH19], [DOS20], where some of the produced garbled circuits are completely revealed for verification, and the rest are used for actual computation. In our setting the cut-and-choose overhead can be significantly reduced, taking into account that (i) this is a repeating rather than a one-time game, a setting not yet addressed in previous works [KM15], [HKK+19], [ZDH19], [DOS20] and left for future work; (ii) in our context, an incorrect garbled circuit can be reported and verified on-chain (in a technique similar to Arbitrum's bisection-based fraud-proof algorithm [Arb23]). This means that an arbitrator smart contract will need to verify the correctness of a single garbled gate, incurring a constant on-chain cost. If the arbitrator smart contract indeed finds an incorrect garbled gate then the garbler(s) will be slashed. Alternatively, the garbling itself can be computed using a publicly auditable protocol, for which there are various approaches [BDO14], [SV15]. In the case of $|G| = 1$, this can be done with a simple zero-knowledge proof that the produced circuit was computed correctly [ASH+20]. To satisfy a greater throughput demands, multiple garbling committees can be formed such that each committee works independently.
As a second non-cryptographic alternative, garbler(s) can run inside a TEE, and so even a leak of the hardware secret keys will not cause an incorrect garbled circuits.
In reality, both methods can be combined in order to achieve two layers of security.
2) Let us address the second requirement. A useful property of garbled circuits is that, given the input wires' labels used to evaluate a garbled circuit, anyone will obtain the exact same labels on the output wires. These labels do not reveal anything to a 3rd party auditor (from the privacy guaranteed by the garbling scheme), meaning that if the labels on the input wires are correct, anyone can verify the resulting labels on the output wires, as reported by the evaluators.
It remains to make sure that the translation from inputs to labels, as well as the soldering (which translates from labels to labels) is publicly verifiable. This is done via a novel technique that utilizes homomorphic commitments applied by the garbler to the permutation bits of the input and output wires' labels of every garbled circuit.
Publicly auditable soldering is achieved by extending the garbling scheme to contain more information about the labels. The following explanation builds on the Half-Gates garbling scheme [ZRE15], but can be extended to any garbling scheme, as discussed later.
In the Half-Gates garbling scheme each input/output wire w_i is associated with two labels L_i^0 and L_i^1 with signal bit (their lsb) being 0 and 1 respectively. The signal bit does not leak information about the real bit that the label represents; rather, the real bit is dictated by a random permutation bit p_i that is also associated with the wire: if $p_i = 0$ then L_i^0 and L_i^1 represent 0 and 1 (respectively), otherwise (if $p_i = 1$) then they represent 1 and 0 (respectively). In our protocol, all labels and permutation bits are secret shared by the evaluators and only one label per wire is revealed right before evaluation. In addition, we extend the garbling scheme so the evaluators also obtain (from the garbler(s)) a commitment to the permutation bit p_i, denoted $c_i = \text{Com}(p_i; r_i)$ as well as a secret sharing to the decommitment r_i, where Com is an additively homomorphic commitment. The evaluators also obtain a single global commitment to the value 1, namely $c^* = \text{Com}(1; r^*)$.
Now, suppose that the evaluators want to connect the output wire w_i of one circuit, to the input wire w_j of another circuit. The evaluators first reveal

$p_{ij} = p_i \oplus p_j$; this tells them whether the permutation bits are the same (i.e., $p_{ij} = 0$) or not (i.e., $p_{ij} = 1$). The evaluators do as follows:

- Case $p_{ij} = 0$. The evaluators need to reveal the map $\{L_i^0 \to L_j^0,\ L_i^1 \to L_j^1\}$ and so they reveal the soldering information $\delta_{ij,0} = L_i^0 \oplus L_j^0$ and $\delta_{ij,1} = L_i^1 \oplus L_j^1$.
- Case $p_{ij} = 1$. The evaluators need to reveal the map $\{L_i^0 \to L_j^1,\ L_i^1 \to L_j^0\}$ and so they reveal the soldering information $\delta_{ij,0} = L_i^0 \oplus L_j^1$ and $\delta_{ij,1} = L_i^1 \oplus L_j^0$.

When it is time to evaluate a complex set of garbled circuits, in which output wire w_i is connected to input wire w_j, the evaluators first obtain the active label on w_i, say L_i^b (for some $b \in \{0,1\}$) and translate it to the appropriate active label on w_j by locally computing $L_j^{b'} = L_i^b \oplus \delta_{ij,b}$. Note that $b' = b$ if $p_{ij} = 0$ and $b' = 1 - b$ if $p_{ij} = 1$. Note that given a label L_i^b, the signal bit b is known by $\mathrm{lsb}(L_i^b)$. Therefore, given p_{ij}, a public auditor need only to make sure that $\mathrm{lsb}(L_i^b) \oplus \mathrm{lsb}(L_j^{b'}) = p_{ij}$ and also verify that the reported p_{ij} is computed correctly.
To enable auditors making sure that the reported p_{ij} is computed correctly, the evaluators also reveal the decommitment for a commitment to zero, as follows:

- Case $p_{ij} = 0$. Reveal $r_{ij} = r_i - r_j$. Then the auditor verifies that r_{ij} is indeed a decommitment to $c_i - c_j$ and so this is a commitment to 0.
- Case $p_{ij} = 1$. Reveal $r_{ij} = r^* - (r_i + r_j)$. Then the auditor verifies that r_{ij} is indeed a decommitment to $c^* - (c_i + c_j)$ and so this is a commitment to 0.

In the above, note that commmitment arithmetics is done over the appropriate module defined by the commitment scheme.

Optimized soldering. While the above works well, it may be inefficient in cases the input/output length is large (e.g., for circuits that operate on 128/256 bit integers). This is because of the reliance on homomorphic commitment, which are typically built on public-key primitives (e.g., Pedersen commitment [Ped92]).
In the following we provide a high level description of a novel publicly auditable optimized soldering mechanism, that builds on the above principles, but require much less invocations of the homomorphic commitment tool. The new mechanism does not rely on the internal instantiation of the garbling scheme. Surprisingly, when instantiated with the Half-Gates (or similar) scheme, it allows a complete reveal of all permutation bits (even though in a typical usage of it they serve as key role for privacy).
The new mechanism enhances each circuit with a 'header' and 'footer' sub-circuits. For example, the circuit that computes `MULT256` that takes two 256-bit integers a, b and returns a 256-bit integer c is enhanced as follows:

- Instead of having only two inputs a, b, the circuit now has two additional inputs δ_a, δ_b.
- The header sub-circuit is hard-coded with two random secrets α_a and α_b, picked and known only to the garbler(s).
- The footer sub-circuit is hard-coded with a random secret α_c, picked and known only to the garbler(s).

The header computes $a^* = a - \alpha_a + \delta_a$ and $b^* = b - \alpha_b + \delta_b$. Then the circuit computes the intended operation, in this example, this is $c^* = a^* + b^*$ and passes the result to the footer, which computes and outputs $c = c^* + \alpha_c$. Overall, this incurs a circuit increase by three addition/subtraction.
The idea is that input and output to garbled circuits are public, but are always masked by a one-time pad that is unknown to the evaluators. The goal of the header is to remove that one-time pad in order to get the plaintext arguments a^* and b^*, and the goal of the footer is to re-mask the output c^*, resulting in c.
The one-time pads $\alpha_a, \alpha_b, \alpha_c$ are secret shared by the evaluators, which allows them perform the soldering, thereby building a complex garbled circuit out of many atomic ones. This is done as follows, suppose the output y^* of circuit C_1 is to be connected to the input x of another circuit C_2. Recall that C_1 doesn't output y^*, rather, it outputs $y = y^* + \alpha_y$ where α_y is random and secret shared. To input y^*, the evaluators simply connect the output wires of y to the input wires of x (this is trivial now because x, y and the permutation bits associated with them are public), and reveal $\delta_x = \alpha_x - \alpha_y$, which is also entered as a public input to C_2. The header of C_2 now computes $x^* = x - \alpha_x + \delta_x = x - \alpha_x + \alpha_x - \alpha_y = y^* + \alpha_y - \alpha_x + \alpha_x - \alpha_y = y^*$. At this point, the value $x^* = y^*$ is ready for the actual computation that the circuit is intended to perform, as required.
To facilitate public auditability of the soldering, the evaluators are also equipped with additively homomorphic commitments to all masking values, as well as a secret sharing to their decommitments. In the above example, the evaluators are equipped with $c_y = \mathrm{Com}(\alpha_y, r_y)$ and $c_x = \mathrm{Com}(\alpha_x, r_x)$ we well as secret shares of r_y and r_x. The evaluators now have to prove that the revealed δ_x indeed equals $\alpha_x - \alpha_y$. This can be done by revealing $r_x - r_y$, in order for the auditor to verify that $c_x - c_y = \mathrm{Com}(\delta_x, r_x - r_y)$.

3 Cryptographic Building Blocks

3.1. Garbling

A **garbling scheme** consists of the following algorithms: (we slightly modify the standard definitions from [BHR12])

- $\mathrm{Garble}(f) \to (F, e, d)$: on input a circuit f, outputs a garbled circuit F, encoding information e, and decoding information d.
- $\mathrm{Eval}(F, X) \to Y$: on input a garbled circuit F and garbled input X, outputs a garbled output Y.

We define a "multiplexer" function Encode as follows. If w is a $n \times 2$ array of wire labels, and $x \in \{0,1\}^n$, then

$$\mathrm{Encode}(W, x) = \big(W[1, x_1], W[2, x_2], \ldots, W[n, x_n]\big).$$

We insist that a garbling scheme is **projective** for both garbled inputs and outputs, meaning that e and d are such 2D arrays of wire labels. Then a garbling scheme is **correct** if, whenever $(F, e, d) \leftarrow \text{Garble}(f)$, we have the following with probability 1:

$$\text{Eval}(F, \text{Encode}(e, x)) = \text{Encode}(d, f(x))$$

Adaptive (circuit-)privacy: Intuitively, the view of the evaluator (garbled circuit plus a garbled input) leaks no more than the circuit input/output, even when inputs are chosen after seeing the garbled circuit. In particular, they leak nothing about the garbler's choice of circuit within some class of functions $\mathcal{F}$.

Formally, there is a two-stage simulator $(\text{Sim}_1, \text{Sim}_2)$ such that the following two games are indistinguishable:

```
receive $f \in \mathcal{F}$ from adversary
$(F, e, d) \leftarrow \text{Garble}(f)$
give $F$ to adversary
receive $x$ from adversary
$X := \text{Encode}(e, x)$
give $X$ to adversary
```

```
receive $f \in \mathcal{F}$ from adversary
$(F, \sigma) \leftarrow \text{Sim}_1()$
give $F$ to adversary
receive $x$ from adversary
$X \leftarrow \text{Sim}_2(\sigma, x, f(x))$
give $X$ to adversary
```

Note: the simulator does not get to see the choice of f, so the adversary's view is independent of f (except via $f(x)$) in the game on the right.

Strong (real-or-random) output-label authenticity: When evaluating a garbled circuit, one obtains a garbled representation of the output y with respect to the output labels d, which we write as $\text{Encode}(d, y)$. It should be hard to guess wire labels for the *complementary* bits (i.e., those that represent the bits of $\overline{y}$). We require a strong flavor of this authenticity property, that the complementary wire labels are indistinguishable from random. More formally, the following two games are indistinguishable:

```
receive $f$ from adversary
$(F, e, d) \leftarrow \text{Garble}(f)$
give $F$ to adversary
receive $x$ from adversary
$X := \text{Encode}(e, x)$
// real labels encoding
// complementary output:
$\tilde{Y} := \text{Encode}(d, \overline{f(x)})$
give $(X, \tilde{Y})$ to adversary
```

```
receive $f$ from adversary
$(F, e, d) \leftarrow \text{Garble}(f)$
give $F$ to adversary
receive $x$ from adversary
$X := \text{Encode}(e, x)$
// dummy/random labels:
$\tilde{Y} \leftarrow (\{0,1\}^\lambda)^*$
give $(X, \tilde{Y})$ to adversary
```

Input label authenticity: Intuitively, the evaluation algorithm should detect the presence of erroneous garbled inputs. More formally, the adversary has negligible probability of winning this game:

```
receive $f$ from adversary
$(F, e, d) \leftarrow \text{Garble}(f)$
give $F$ to adversary
receive $x$ and $\vec{\epsilon}$ from adversary
$X := \text{Encode}(e, x) \oplus \vec{\epsilon}$
adversary wins if $\text{Eval}(F, X) \neq \bot$ and $\vec{\epsilon} \neq 0^*$
```

Instantiating garbled circuits: The state of the art schemes for boolean garbling are those of Rosulek & Roy [RR21] and Zahur, Rosulek, and Evans [ZRE15]. The security definitions we have presented here, for privacy and output authenticity, are compatible with standard garbling schemes without modification. Our requirement for input-label authenticity is non-standard, and would require slightly modifying a standard scheme as follows: For every input wire w, with labels W_0 and W_1, the garbled circuit must include additional hashes $H(w, W_0), H(w, W_1)$. To validate a garbled input, the evaluator can hash the given input labels and check for their presence in the garbled circuit information. When the hash function is the one used in the garbling scheme, a standard argument shows that there is no harm to security by including these additional wire label hashes.

3.2. Setup Functionalities

Our protocol requires two setup functionalities: $\mathcal{F}_{\text{a-com}}$ and $\mathcal{F}_{\text{u-com}}$. Both are homomorphic secret-sharing functionalities: senders (one or many, if they have consensus) can share a value among the receivers. Receivers (if they have consensus) can open any linear combination of shared values. In $\mathcal{F}_{\text{a-com}}$, the results are authenticated, meaning that receivers will obtain the correct value (or else abort). In $\mathcal{F}_{\text{u-com}}$, the results are unauthenticated, meaning that a corrupt receiver can inject an additive error into the opened value.

```
$\mathcal{F}_{\text{u-com}}, \mathcal{F}_{\text{a-com}}$

on input $(\texttt{store}, k, v)$ from **all** senders:
set $V[k] := v$
give $(\texttt{store}, k)$ to all receivers

on input $(\texttt{add-const}, k, c; k')$ from **all** receivers:
set $V[k'] = V[k] + c$
give $(\texttt{add-const}, k, c; k')$ to all receivers

on input $(\texttt{reveal}, k_1 + k_2 + \cdots)$ from **all** receivers:
$v = V[k_1] + V[k_2] + \cdots$
in the $\mathcal{F}_{\text{u-com}}$ functionality only:
    if any receiver is corrupt, await $\epsilon$ from simulator
    $v := v + \epsilon$
give $(\texttt{reveal}, k, v)$ to all receivers
```

Instantiating the setup functionalities. Our protocol requires protocols for such functionalities, secure in the corruption settings we consider. Namely, they should be secure against a dishonest majority of senders, *or* a dishonest majority of receivers.

One potential instantiation of $\mathcal{F}_{\text{u-com}}$ is through simple additive secret sharing. The senders initially choose a common seed s, known only to them. To implement the $(\texttt{store}, k, v)$ command, they generate an additive secret sharing of v, with randomness derived pseudorandomly from s. In this way, they compute exactly the same individual shares. Each sender distribute shares to the corresponding receivers, who then ensure that they have received identical shares from all senders. The additive homomorphic feature of additive sharing is standard. The receivers open a shared value by first committing to their shares and then opening. There is no guarantee that corrupt receivers commit to their correct shares, but by using a round of commitments, the effect of incorrect shares is that of adding a known error ϵ to the opened value.

$\mathcal{F}_{\text{a-com}}$ can be instantiated with authenticated secret shares, using either the BDOZ method [BDOZ11] or the SPDZ method [DPSZ12]. The senders act as the dealer in these secret sharing schemes, and values are opened using the secure opening protocols of these schemes.

We require the sharing to be additively homomorphic with respect to some group. Our protocol uses $\mathcal{F}_{\text{u-com}}$ for wire labels, so the group can be strings of length λ with respect to the XOR operation. We need $\mathcal{F}_{\text{a-com}}$ to have homomorphism with respect to a group $\mathcal{G}$ (which may be $(\{0,1\}^\lambda, \oplus)$ but need not be) which is discussed below.

Public verifiability: The public verifiability of our main protocol rests on the public verifiability of the $\mathcal{F}_{\text{a-com}}$ functionality. That is, the commitments should be binding from the perspective of an external judge, even if *all* receivers are corrupt.

BDOZ and SPDZ authenticated sharings are not binding in this way; if all receivers are corrupt, then they can easily falsify the transcript of an opening. To achieve public verifiability, then, the senders should commit to the value using a Pedersen commitment (which it publishes, e.g., in the ledger), then additively share the opening/decommitment value among the receivers. Then, cheating among receivers is irrelevant because binding is inherited directly from the Pedersen commitment, and not from the corruption threshold. We are not aware of any Pedersen-like commitment supporting XOR homomorphism, but only homomorphism on $(\mathbb{Z}_n, +)$. We also note that Pedersen commitments operate homomorphically on both the committed value and the decommitment values, so evaluators are able to homomorphically compute the necessary sharings of the decommitment values.

4 Our Main Protocol

4.1. Opcodes & Transactions

We begin by formalizing our notation for transactions. Let $(\mathcal{G}, +)$ be a group. An **opcode** is a deterministic function $f : \mathcal{G}^m \to \mathcal{G}^m$. (The purpose of requiring a group operation over these values will be explained later.) For ease of explanation, we assume all opcodes have the same number of inputs/outputs, but this is not a fundamental limitation of our protocol techniques.

Intuitively, a transaction is a directed acyclic graph (or simply a circuit) of opcodes, whose inputs are either public values or system key values. Formally, a transaction T consists of the following:

- T.ops is an ordered sequence of opcode ids.
- T.in(i, j) indicates where input j of opcode i comes from. There are three possibilities:
  - T.in$(i, j) = (\text{key}, P)$ means the input is $K[P]$
  - T.in$(i, j) = (\text{const}, v)$ means the input is public constant v
  - T.in$(i, j) = (\text{conn}, i', j')$ means the input comes from output j' of opcode i'.
- T.conn is the set of all (i', j', i, j) such that T.in$(i, j) = (\text{conn}, i', j')$. In other words, output j' of op i' becomes input j of op i.
- T.out is a list of (i, j) pairs, indicating that output j of op i is a transaction-level output.

Such a transaction must satisfy the following:

- Topological ordering: if T.in$(i, j) = (i', j')$ then i' comes before i in T.ops list.
- No fan-out: for all (i', j'), there is at most one tuple of the form $(i', j', \cdot, \cdot)$ in T.conn.

Running a transaction, with respect to a mapping f of ids $\mapsto$ functions, and a mapping K of key-names $\mapsto$ keys, refers to the following process:

```
RUN($T, f, K$):
  for each $i \in T$.ops:
    for each input $j$:
      if $T.\text{in}(i,j) = (\text{key}, P)$: $x[i,j] := K[P]$
      if $T.\text{in}(i,j) = (\text{const}, v)$: $x[i,j] := v$
      if $T.\text{in}(i,j) = (\text{conn}, i', j')$: $x[i,j] := y[i',j']$
    $y[i,\cdot] = f[i](x[i,\cdot])$

  for each $(i,j) \in T$.out:
    $out = out \| y[i,j]$
  return $out$
```

4.2. The Ideal Functionality

The ideal $\mathcal{F}_{\text{gcVM}}$ functionality is given below:

```
$\mathcal{F}_{\text{gcVM}}$

on command $(\text{keygen}, P)$ from all garblers:
// $P$ may be $\bot$, indicating network key
if $K[P]$ undefined: $K[P] \leftarrow \{0,1\}^\lambda$
give $(\text{keygen}, K[P])$ to party $P$

on command $(\text{new-op}, f)$ from all garblers:
give $(\text{new-op}, f)$ to the simulator; await response $i$
$f[i] := f$
give $(\text{new-op}, f, i)$ to all evaluators

on command $(\text{run}, T)$ from all evaluators:
for all $i \in T$.ops: assert $f[i]$ defined
$out := \text{RUN}(T, f, K)$
for all $i \in T$.ops: delete $f[i]$ // (single use)
give $(\text{run}, T, out)$ to all evaluators
```

4.3. Protocol Intuition & Concepts

For an opcode $f : \mathcal{G}^m \to \mathcal{G}^m$ and vectors $\alpha, \beta \in \mathcal{G}^m$, we define its **masked** version $f_{\alpha,\beta}$ as:

$$f_{\alpha,\beta}(\tilde{x}, \delta) = f(\tilde{x} - \alpha) + \beta + \delta.$$

Note that $\alpha, \beta, \tilde{x}, \delta$ are all vectors in $\mathcal{G}^m$. The class of all maskings of f is denoted:

$$\text{mask}(f) = \{f_{\alpha,\beta} \mid \alpha, \beta \in \mathcal{G}^m\}$$

Our protocol evaluates opcodes in a masked fashion, as follows:

- Garblers will generate a collection of (garbled) opcodes, with secret α, β masks hard-coded in, and hidden by the garbling.
- During run-time, if the plaintext input to opcode f is x, then the masked value $\tilde{x} = x + \alpha$ is assumed to be public to the evaluators.

- The purpose of δ inputs to allow a masked opcode circuit to mask its output using the input mask of a downstream opcode. In other words, the output of one opcode circuit is masked in exactly the manner that the next opcode circuit expects. Note that the masks for each garbled circuit are chosen independently, before the transaction's connection topology is known. This is why δ must be an input to the circuit.

Garbling masked opcode circuits: In the simplest case, we take $\mathcal{G} = (\{0,1\}^\lambda, \oplus)$. Standard boolean garbled circuits support masking in this $\mathcal{G}$, essentially "for free." In the standard point-and-permute paradigm, α and β can be the garbler's secret "permute bits" on the input/output wires. Then a garbled encoding of input x is already designed to reveal $\alpha \oplus x$.

But our protocol requires an $\mathcal{F}_{\text{a-com}}$ functionality that is homomorphic with respect to the same group $\mathcal{G}$ as the garbling masks. Not all instantiations of $\mathcal{F}_{\text{a-com}}$ support XOR homomorphism — specifically, instantiations using Pedersen commitments support only $\mathcal{G} = (\mathbb{Z}_n, +)$. In these cases, the garbled circuit must "manually" encode the masking/unmasking operations with extra circuitry.

Common notation: We use the following conventions:

- i = id of opcode instance
- j = logical input/output of an opcode (a group element)[4]
- k = particular bit in the binary encoding of a group element

So, (i,j,k) may refer to the kth bit in the encoding of the jth input to the ith opcode instance.

Garbling notation: We garble masked opcodes of the form $f_{\alpha,\beta}$, which have two input vectors: x and δ. We partition the garbled input information e (and garbled input labels X) into two parts: e_x and e_δ (resp., X_x and X_δ).

Thus, $e_x[j,k,b]$ denotes the wire label representing that the kth bit of $x[j]$ is b.

Overview of soldering: Suppose for simplicity each opcode has a single input and single output, and that a certain transaction requires the output of opcode f to become the input of opcode f'. The garblers will have already produced a garbled (masked) opcode circuits $f_{\alpha,\beta}$ and $f'_{\alpha',\beta'}$. They will also have distributed homomorphic commitments to the $\alpha,\beta,\alpha',\beta'$ masks.

During evaluation time, the evaluators will inductively obtain the masked input $\tilde{x} = x + \alpha$ for opcode f. To connect the f and f' circuits, they combine homomorphic commitments of α' and β to reveal the value $\delta = \alpha' - \beta$. Now the plaintext values $\tilde{x}$ and δ will be the inputs to the masked $f_{\alpha,\beta}$ circuit. As a result, this masked circuit will compute:

$$\begin{aligned} f_{\alpha,\beta}(\tilde{x},\delta) &= f(\tilde{x}-\alpha) + \beta + \delta \\ &= f\Big((x+\alpha)-\alpha\Big) + \beta + (\alpha' - \beta) = f(x) + \alpha' \end{aligned}$$

In this way, the output of this masked circuit is the opcode output $f(x)$, masked with the correct mask (α') for the next circuit.

4. j is relevant for the non-binary case, or if we have multiple output one for different circuit.

The benefit of having the output of one circuit *exactly equal* the input of the next circuit is that the garbled encodings of these values can be easily translated. In particular, suppose d represents all output labels of the $f_{\alpha,\beta}$ circuit, and e' represents all input labels (for the "x-input" but not "δ-input") of the $f'_{\alpha',\beta'}$ circuit. Then the evaluators can homomorphically reveal (essentially) $d \oplus e'$. Think of this as a $n \times 2$ matrix of one-time pad ciphertexts, where each output label of $f_{\alpha,\beta}$ is used to mask the matching input label of $f'_{\alpha',\beta'}$. (Our strong output authenticity property ensures that it is safe to use output labels as one-time pads in this way.) So after evaluating $f_{\alpha,\beta}$, the evaluators can (for each wire) decrypt the corresponding one-time pad ciphertext to learn (non-interactively) the garbled input for $f'_{\alpha',\beta'}$.

4.4. Protocol Description

Our protocol requires the following:

- an instance of $\mathcal{F}_{\text{a-com}}$ over the group $(\mathcal{G},+)$, which is the same group as for masking opcodes.
- an instance of $\mathcal{F}_{\text{u-com}}$, which can be over the group $(\{0,1\}^\lambda, \oplus)$

The formal description is given in Figure 3.

5 Security Analysis

Theorem 1. If (Garble, Eval) is a garbling scheme satisfying the properties listed in Section 3.1, then our protocol UC-securely realizes $\mathcal{F}_{\text{gcVM}}$ in the presence of an adversary who corrupts all but one garbler, or all but one evaluator.

Additionally, correctness (public verifiability) is guaranteed even against an adversary who corrupts all but one garbler and any number of evaluators, as long as $\mathcal{F}_{\text{a-com}}$ is binding in the presence of one honest sender.

We defer full proofs to the full version, and provide only high-level sketches in this version. We divide the analysis into three cases, depending on the corruption scenario.

Corrupt garblers only:

Note that $\mathcal{F}_{\text{a-com}}$ and $\mathcal{F}_{\text{u-com}}$ require consensus from *all* garblers for `store` commands, and evaluators (who are honest) also require consensus from garblers about the garbled circuits F. Thus, since we assume at least one honest garbler, any corrupt garbler who does not strictly follow the protocol will cause it to abort. This is straight-forward to simulate.

Corrupt evaluators only: We assume that at least one evaluator is honest. Then since $\mathcal{F}_{\text{a-com}}$ and $\mathcal{F}_{\text{u-com}}$ require consensus from *all* evaluators for `reveal` commands, the adversary cannot reveal any result besides what the protocol prescribes. However, in the case of $\mathcal{F}_{\text{u-com}}$, the adversary can contribute an additive error to the value that is opened. In our protocol, (we show that) honest parties abort if the adversary uses any nonzero error. Simulating this effect is relatively straightforward.

Thus the main role of the simulator is to simulate the garbled circuits, garbled inputs, and the outputs of

```
initially:
garblers perform coin tossing to obtain secret $gk$
evaluators perform coin tossing to obtain secret $ek$

on command (keygen, $P$):
// $P$ may be $\perp$, indicating network key
every garbler does:
   $R := \mathsf{PRF}(gk, (\texttt{prekey}, P))$
   send $(\texttt{store}, (\texttt{pre-key}, P), R)$ to $\mathcal{F}_{\text{a-com}}$
   (if $P \neq \perp$) send $R$ to $P$ through a secure channel
every evaluator does:
   await $(\texttt{store}, (\texttt{pre-key}, P))$ from $\mathcal{F}_{\text{a-com}}$
   $R' := \mathsf{PRF}(ek, P)$
   send $(\texttt{add-const}, (\texttt{pre-key}, P), R'; (\texttt{key}, P))$
   (if $P \neq \perp$) send $R'$ to $P$ through a secure channel
party $P$ does:
   ensure identical $R$ received from every garbler
   ensure identical $R'$ received from every evaluator
   store $R + R'$ as gcEVM key

on command (new-op, $f$):
every garbler does:
   $counter := counter + 1$
   $\alpha, \beta := \mathsf{PRF}(gk, (\texttt{alpha-beta}, counter))$
   $r := \mathsf{PRF}(gk, (\texttt{garble-seed}, counter))$
   $(F, e, d) := \mathsf{Garble}(f_{\alpha,\beta}; r)$
   $i := \mathsf{CRHF}(F)$

   for each logical input $j$:
      send $(\texttt{store}, (\texttt{alpha}, i, j), \alpha[j])$ to $\mathcal{F}_{\text{a-com}}$
      send $(\texttt{store}, (\texttt{beta}, i, j), \beta[j])$ to $\mathcal{F}_{\text{a-com}}$
      for each encoding bit $k$ and $b \in \{0,1\}$:
         send the following to $\mathcal{F}_{\text{u-com}}$:
            $(\texttt{store}, (\texttt{in-label-x}, i, j, k, b), e_x[j,k,b])$
            $(\texttt{store}, (\texttt{in-label-delta}, i, j, k, b), e_\delta[j,k,b])$
            $(\texttt{store}, (\texttt{out-label}, i, j, k, b), d[j,k,b])$

   broadcast $(\texttt{new-op}, f, i, F)$ to all evaluators

every evaluator does:
   abort if any of the following are violated:
      all $(\texttt{store}, \cdots)$ outputs received from $\mathcal{F}_{\text{a-com}}/\mathcal{F}_{\text{u-com}}$,
         as expected
      identical $(\texttt{new-op}, f, i, F)$ received from all garblers
      $i = \mathsf{CRHF}(F)$
   store $F[i] := F$

on command (run, $T$):
every evaluator does:
   // compute $\delta$ inputs:
   for every $(i, j, i', j') \in T.\mathsf{conn}$:
      send $(\texttt{reveal}, (\texttt{alpha}, i', j') - (\texttt{beta}, i, j))$ to $\mathcal{F}_{\text{a-com}}$
      await response $(\texttt{reveal}, \cdots, \delta[i,j])$
   for every $(i, j) \in T.\mathsf{out}$:
      send $(\texttt{reveal}, -(\texttt{beta}, i, j))$ to $\mathcal{F}_{\text{a-com}}$
      await response $(\texttt{reveal}, \cdots, \delta[i,j])$

   // obtain garbled $\delta$ inputs:
   for every $(i, j)$ and every encoding bit $k$:
      write $\delta[i,j]$ in binary, so that $\delta_k$ is its $k$th bit
      send $(\texttt{reveal}, (\texttt{in-label-delta}, i, j, k, \delta_k))$ to $\mathcal{F}_{\text{u-com}}$
      await response $(\texttt{reveal}, \cdots, X_\delta[i,j,k])$

   // soldering
   for every $(i, j, i', j') \in T.\mathsf{conn}$ and $b \in \{0,1\}$:
      send $(\texttt{reveal}, (\texttt{out-label}, i, j, k, b) + (\texttt{in-label-x}, i', j', k, b))$
            to $\mathcal{F}_{\text{a-com}}$
      await response $(\texttt{reveal}, \cdots, \Delta[i, j, i', j', k, b])$

   // obtain (masked) transaction inputs:
   for every $(i, j)$ such that $T.\mathsf{in}(i,j) \neq (\mathsf{conn}, \cdots)$:
      if $T.\mathsf{in}(i,j) = (\mathsf{const}, v)$:
         send $(\texttt{reveal}, (\texttt{alpha}, i, j) + v)$ to $\mathcal{F}_{\text{a-com}}$
         await response $(\texttt{reveal}, \cdots, \tilde{x}[i,j])$
      else $T.\mathsf{in}(i,j) = (\mathsf{key}, P)$:
         send $(\texttt{reveal}, (\texttt{alpha}, i, j) + (\mathsf{key}, P))$ to $\mathcal{F}_{\text{a-com}}$
         await response $(\texttt{reveal}, \cdots, \tilde{x}[i,j])$
      write $\tilde{x}[i,j]$ in binary, so that its $k$th bit is $\tilde{x}[i,j]_k$
      for every encoding bit $k$:
         send $(\texttt{reveal}, (\texttt{in-label-x}, i, j, k, \tilde{x}[i,j]_k))$ to $\mathcal{F}_{\text{u-com}}$
         await response $(\texttt{reveal}, \cdots, X_x[i,j,k])$

   // evaluation:
   for every $i \in T.\mathsf{ops}$:
      for each input $j$ and encoding bit $k$:
         if $T.\mathsf{in}(i,j) = (\mathsf{conn}, i', j')$:
            // masked output $\tilde{y}[i', j']$ defined previously
            $\tilde{x}[i,j] := \tilde{y}[i', j']$
            $X_x[i,j,k] := Y[i', j', k] \oplus \Delta\big[i', j', i, j, k, \tilde{x}[i,j]_k\big]$
         else: ($X_x[i,j,k]$ already defined previously)

      $Y[i,\cdot] := \mathsf{Eval}(F[i], X_x[i,\cdot] \| X_\delta[i,\cdot])$ // abort if this step returns $\perp$
      $\tilde{y}[i,\cdot] := \mathsf{Decode}(Y[i,\cdot])$
      delete $F[i]$

   // output:
   for every $(i, j) \in T.\mathsf{out}$:
      $out = out \| \tilde{y}[i,j]$
   output $(\mathsf{run}, T, out)$
```

Figure 3. Formal protocol description.

the $\mathcal{F}_{\text{a-com}}$ and $\mathcal{F}_{\text{u-com}}$ functionalities. We describe the simulator below:

```
simulator
upon receiving $(\texttt{new-op}, f)$ from $\mathcal{F}_{\text{gcVM}}$:
simulate expected $(\texttt{store}, \cdots)$ messages from $\mathcal{F}_{\text{u-com}}/\mathcal{F}_{\text{a-com}}$,
      as prescribed in protocol description
generate simulated garbled circuit $F$ (for circuit class mask($f$))
reply to $\mathcal{F}_{\text{gcVM}}$ with $i = \text{CRHF}(F)$
store $F[i] := F$
simulate broadcast $(\texttt{new-op}, f, i, F)$ from all garblers

command $(\texttt{run}, T)$:
for every $(i, j)$:
  $\delta[i, j] \leftarrow \mathcal{G}$
  sample $\tilde{x}[i, j], \tilde{y}[i, j] \leftarrow \mathcal{G}$, subject to
        $\tilde{x}[i, j] = \tilde{y}[i', j']$ for all $(i', j', i, j) \in T.\text{conn}$
for every $i \in T.\text{ops}$:
  run the second phase of GC simulator for $F[i]$,
        with plaintext input $(\tilde{x}[i, \cdot], \delta[i, \cdot])$ and output $\tilde{y}[i, \cdot]$,
        obtaining (simulated) garbled input $X_x[i, \cdot]$ and $X_\delta[i, \cdot]$
  $Y[i, \cdot] = \text{Eval}(F[i], X_x[i, \cdot] \| X_\delta[i, \cdot])$
for every $(i, j, i', j') \in T.\text{conn}$:
  $b := \tilde{x}[i', j']_k$
  // $= \tilde{y}[i, j]_k$ = the bit that will actually be visible on this wire
  $\Delta[i, j, i', j', k, b] := Y[i, j, k] \oplus X_x[i', j', k]$
  $\Delta[i, j, i', j', k, 1 - b] \leftarrow \{0, 1\}^\lambda$

wait for all corrupt evaluators to send the prescribed
      reveal commands to $\mathcal{F}_{\text{a-com}}/\mathcal{F}_{\text{u-com}}$
simulate $\mathcal{F}_{\text{a-com}}/\mathcal{F}_{\text{u-com}}$ responses using variable
      names defined above, and adding the adversary-provided
      error for responses from $\mathcal{F}_{\text{u-com}}$

if nonzero error was added to any $X_x[i, j, k], X_\delta[i, j, k]$,
      or $\Delta[i, j, i', j', k, \tilde{y}[i, j]_k]$:
  simulate that the honest parties aborted
```

Below we summarize the sequence of hybrids that establish indistinguishability between the real and ideal worlds:

- (Real world:) simulation plays the role of all honest parties and setup functionalities.
- (Hybrid 1 introduces the following change:) If the adversary provides nonzero error for an $\mathcal{F}_{\text{u-com}}$-response that is actually used (either an X_x, X_δ value, or $\Delta[i, j, i', j', k, b]$ value for the correct b), then honest parties abort. This hybrid differs from the previous only in the bad event that a nonzero error would not have caused honest parties to abort. Using a reduction to the input authenticity property, this bad event can be shown to have negligible probability, and hence the change is indistinguishable.
- (Hybrid 2 introduces the following change:) Simulate $\Delta[i, j, i', j', k, 1 - b]$ values (where b is the "correct" value), as output by $\mathcal{F}_{\text{u-com}}$ in the soldering phase, as random strings. This change is indistinguishable by a straight-forward reduction to the real-or-random output authenticity property of garbling. Appealing to that security game, we are able to replace correct complementary output labels $Y[i, j, k, 1 - b]$ with random strings. But each such output label is used only in the computation of a single Δ value (because of our fan-out restriction), acting as a OTP.
- (Hybrid 3 introduces the following change:) Replace real garbled circuit $F[i]$ and garbled input $X[i, \cdot]$ with simulated ones. The garbled circuit is produced during `new-op`, and the garbled input produced after the simulator knows the input/output of each opcode circuit (the public, masked values). Thus we have a straight-forward reduction to the adaptive security of the garbling scheme. The simulation does not need the output labels $d[\cdot]$, only the visible garbled output, which can be computed by evaluating the garbled circuit. Note that after this change the simulator simulates the garbled circuit without using its α, β parameters.
- Now the α, β values are used *only* to compute the `reveal-output` values from $\mathcal{F}_{\text{u-com}}/\mathcal{F}_{\text{a-com}}$ (namely, δ values and $\tilde{x}$ values). Because of the fan-out restriction on the transaction, a uniform distribution over α's and $\beta's$ induces a uniform distribution over these $\mathcal{F}_{\text{u-com}}/\mathcal{F}_{\text{a-com}}$ outputs. Thus, this hybrid is identical to the ideal world with the simulator we have defined above.

Corrupt garblers and evaluators (public verifiability): In this setting we assume the presence of at least one honest garbler, although all evaluators may be corrupt. If the adversary corrupts a mixture of garblers and evaluators, then privacy is compromised. However, we claim that the protocol achieves guaranteed correctness / public verifiability.

The proof boils down to the following essential claims:

1) The protocol requires total consensus from the garblers about the garbled circuits and the homomorphic commitments. So if at least one garbler is honest, then all garbled circuits and homomorphic commitments are correct.
2) If $\mathcal{F}_{\text{a-com}}$ is binding even when all receivers are corrupt, then all of the $\mathcal{F}_{\text{a-com}}$ openings must be correct with respect to the underlying α, β masks and the network keys. In particular, the masked inputs and the δ values are correct and consistent with the α, β masks hard-coded into the opcode circuits.
3) Homomorphic openings in $\mathcal{F}_{\text{u-com}}$ need not be correct in this case. However, if these are not opened correctly, then the input authenticity property of the garbled circuit (which *is* correct) ensures that garbled evaluation will abort with overwhelming probability.

Thus, if evaluation does not abort, we may conclude that the correctly generated garbled circuits are being evaluated on the correct input labels that encode the correct transaction inputs.

6 Geth Integration & PoC Evaluation.

Integration with Go-Ethereum. We forked the Go-Ethereum repository [Eth25] and extended it in two aspects:

- Consensus-Execution Separation. As explained above, we separate the two sub-networks. The consensus sub-network can be permissionless and follow the Ethereum 2.0 Gasper PoS protocol (see [BG17]), however, in our PoC we used a version of Ethereum PoA, with some modification to facilitate the consensus-execution separation. The execution sub-network is permissioned (PoA) and benefits from synchronous decryption mechanism (in contrast to the new fhEVM version that transitioned to asynchronous decryption[5]). Synchronous decryption is important for smart contract developers, as they can utilize the same code-flow as they are used to from the usual (non privacy preserving) contract coding. The async decryption architecture forces developers using 'callbacks' for getting a decryption results, which complicates the code and makes it less understandable. We utilize the 'difficulty' parameter in order to work with two block types: canonical and non-canonical (see Figure 1) and yet have minimal code changes. The difficulty parameter is used in PoW and PoA protocols to determine the chain's head according to the different chains accumulated weight. We fix different difficulty values X and Y to non-canonical (NC) and canonical (CA) blocks, respectively, such that $X < Y$. This way, the next canonical block surely override its predecessor non-canonical one. The following is an example of the first few blocks of the chain:

```
Block #0 CA. Genesis
  Block #1 NC(parent=#0, difficulty=X)
  Block #1 CA (parent=#0, difficulty=Y)
     Block #2 NC(parent=#1, difficulty=X)
     Block #2 CA (parent=#1, difficulty=Y)
        Block #3 NC(parent=#2, difficulty=X)
        Block #3 CA (parent=#2, difficulty=Y)
          ...
```

- Instruction Set Extension. We use Ethereum's standard interface to extend the EVM with more functionality using precompiled contracts. In our case, we extend it with functionality that triggers secure computation. A solidity library `MpcCore.sol` facilitates a convenient way to trigger secure computation related functionalities. That library defines the necessary types (e.g., `itUint64, ctUint64, gtUint64` for 64-bit integer input-text, cipher-text and garbled-text; see Section 2.1 for more explanation) and the functionalities that can work on these types. [6]

The MPC instantiation. The MPC version evaluated in this PoC includes a protection from malicious adversaries, but does not provide public auditability. In addition, current soldering happens online rather than offline (see Section 2.3), meaning that the parties spend two communication rounds before each GC evaluation. This PoC consists of two replicated garblers (producing exactly the same GCs) so if one malicious garbler publishes incorrect GCs (which are different than the ones published by the honest garbler) the protocol halts until the attacker is eliminated. In addition, it consists of two evaluators. The concrete guarantees in this model are as follows: (i) corruption of at most one party (either garbler or evaluator) leaves both correctness and privacy intact; (i) corruption of both evaluators or both garblers breaks both correctness and privacy; and (iii) corruption of an evaluator and a garbler breaks privacy but not correctness. The garbling scheme used is the Half-Gates [ZRE15], and the authenticated secret sharing is instantiated with IT-MACs originated from the garblers. That is, a share p_1 of a permutation bit p held by the first evaluator is associated with a MAC $M[p_0] \in \{0,1\}^\lambda$. To reveal p_1 to the second evaluator, the first evaluator sends p_1 and $M[p_1]$, upon which the second evaluator verifies that $M[p_1] = P[p_1] \oplus p_1 \cdot K$, where K is the global secret MAC key held by the second evaluator, $M[p_1]$ is given to the first evaluator and $P[p_1]$ is a secret random pad given to the second evaluator. The garblers are responsible for distributing these values to the evaluators. It is easy to see that IT-MACs are XOR-homomorphic, which enables authentication XOR of secret shared bits. See more details on IT-MAC in Section C.

We note that public auditability, offline soldering, and increasing the number of parties are ongoing work on which we plan to report in the full paper.

gcVM configuration. The main advantage of the gcVM is expressed by the fact that the evaluators can store many GCs in a local inventory, so they can get ready for a burst event that requires a pick consumption. Specifically, the evaluators are configured with a capacity argument for the circuit of each opcode, which means that they keep accepting GCs from the garbler until that capacity is filled. For instance, given 1000 `Transfer` GCs, 3000 `Onboard` GCs and 2000 `Offboard` GCs in the evaluators' inventory, they can still process 1000 confidential ERC20 (CERC20) transfers even in an extreme cases when garblers are disconnected (See illustration in Figure 7). The gcVM version on which we report is configured with:

- Block interval of 5s, meaning that the sequencer sub-network publishes the next non-canonical block 5 seconds after the last canonical block is sealed.
- Capacities for `Transfer`, `Onboard` and `Offboard` is set to 2000, 6000, and 4000, respectively.
- Block gas limit is set to 310,000,000. This means that up to 1000 CERC20 can fit in a single block.

To increase ctps one can reduce the block interval time, increase the capacities and increase the block gas limit.

Deployment and benchmarks. The MPC nodes are deployed over a 5Gb network on AWS (all in North Virginia), on `r5n.2xlarge` on-demand instances. Metrics of interest to evaluated are:

- The gcVM throughput, which is measured by the number of CERC20 transfers per second, or *ctps*.
- The dollar cost burden on MPC nodes per CERC20 transfer.

We measure CERC20 throughput using the following setup:

5. See https://www.zama.ai/post/fhevm-v0-4
6. See https://anonymous.4open.science/r/gcvm-DDC5/MpcCore.sol for a full list of supported types and functionalities.

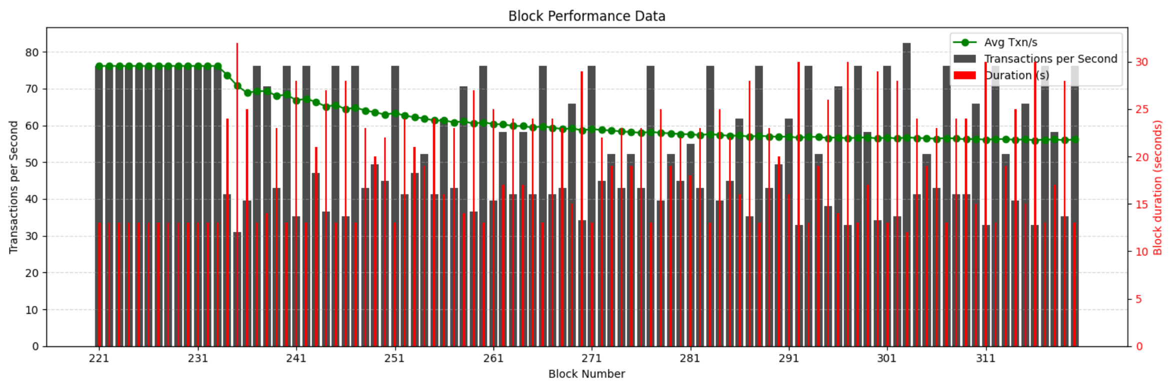


Figure 4. Throughput test summary. Note that avg. ctps drops after inventory is drained, but remains stable afterwards, implying that increasing the inventory can keep a high ctps for a longer period. Each gray bar represents a block; its height indicates the calculated ctps. The red line on top of a gray bar indicates the time to execute that block.

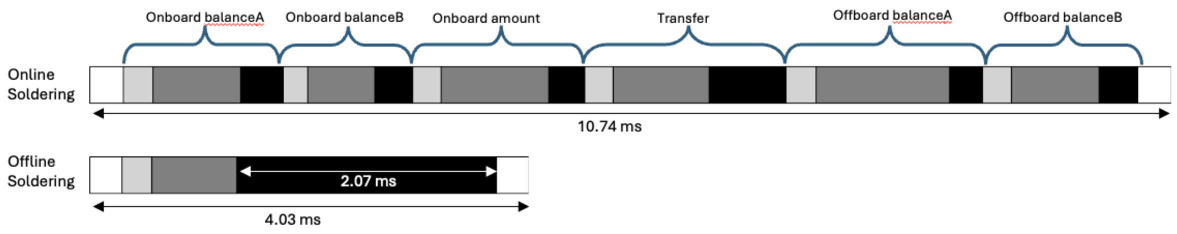


Figure 5. CERC20 time consumption breakdown on a single threaded evaluator. The left and right white margins represent Geth's tx preparation and finalization, and in between there are 6 segments: 3 for onboarding the `balanceA, balanceB, amount` ciphertexts (transforming them into garbledtext), then the `Transfer` operation, followed by 2 `Offboard operation` transforming the updated `balanceA, balanceB` back into ciphertexts. Each segment begins with a precompiled contract call preparation (light gray), followed by the online soldering rounds (dark gray), finally followed by the actual GC evaluation (black).

- We ran 10 EOA accounts in parallel. Each account was programmed to send 10,000 transfers, resulting in a burst of 100,000 transactions processed in total.
- All accounts sent transactions to a pre-deployed cERC20 contract.[7]
- The accounts were fully onboarded prior to the start of the test, so the users onboarding process and its associated transactions were excluded from the scope.

The results are summarized in Figure 4 and show the following: over 1930 seconds (32 minutes) and 102 blocks we get an average of 56.17 ctps, with minimum of 31 and maximum of 82.5 ctps.

A typical CERC20 time consumption breakdown of handling a CERC20 transfer is depicted in Figure 5. The top chart is the actual transaction breakdown in our implementation that currently utilizes online soldering, whereas the bottom chart demonstrates the anticipated improvement when optimizing with offline soldering per transaction. This offline soldering can be further pushed to reduce overall time if performed once for many transactions (and potentially the entire block). The core GC evaluation time takes only 2.07ms on a single threaded evaluator, which, given a client more optimized than Geth (e.g., Reth[8]), can take us to about 500 ctps, before parallelization.

The dollar cost per CERC20 transfer per evaluator is calculated based on (i) an hourly rate of the instances used (`$0.59`); (ii) an average throughput of 56 ctps (before any optimization); and (iii) the cross-region data transfer costs `$0.0050`/GB, considering each CERC20 incurs data transfer of 1MB (dominated by 5 garbled circuits for decryption/encryption, each of 200KB). We get that the cost per transfer is $\$0.000008 = 0.59/3600/50 + 0.0050/1024$, or 0.0008 cents.

7. The contract can be found here https://anonymous.4open.science/r/gcvm-DDC5/PrivateERC20Contract.sol
8. See https://github.com/paradigmxyz/reth

References

[Arb23] Arbitrum. Arbitrum (ARB) Deep Dive: Infrastructure, ARB Ecosystem and Competitors. https://zerocap.com/insights/research-lab/arbitrum-arb-deep-dive/, 2023.

[ASH+20] Jackson Abascal, Mohammad Hossein Faghihi Sereshgi, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Is the classical GMW paradigm practical? The case of non-interactive actively secure 2PC. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1591–1605. ACM Press, November 2020.

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 423–443. Springer, Cham, February 2020.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.

[BCT21] Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkhawk: Practical private smart contracts from mpc-based hawk. *CoRR*, abs/2104.09180, 2021.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 175–196. Springer, Cham, September 2014.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Berlin, Heidelberg, May 2011.

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BOSS20] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 562–592. Springer, Cham, August 2020.

[BT22] Aritra Banerjee and Hitesh Tewari. Multiverse of hawkness: A universally-composable mpc-based hawk variant. *Cryptogr.*, 6(3):39, 2022.

[ByCDF23] Carsten Baum, James Hsin yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023, Part I*, volume 13950 of *LNCS*, pages 270–288. Springer, Cham, May 2023.

[Dai22] Wei Dai. PESCA: A privacy-enhancing smart-contract architecture. Cryptology ePrint Archive, Report 2022/1119, 2022.

[DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 647–676. Springer, Cham, August 2020.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.

[Eth25] Ethereum Foundation and Contributors. go-ethereum: Go implementation of the Ethereum client. https://github.com/ethereum/go-ethereum/, 2025.

[FHKS21] Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic compiler for publicly verifiable covert multi-party computation. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 782–811. Springer, Cham, October 2021.

[FJN+13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 537–556. Springer, Berlin, Heidelberg, May 2013.

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

[Fou23] Oasis Protocol Foundation. Oasis network: Confidential smart contracts using tees. https://oasisprotocol.org/, 2023.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[HKK+19] Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wenjie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 97–121. Springer, Cham, May 2019.

[KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 210–235. Springer, Berlin, Heidelberg, November / December 2015.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016.

[KNR+17] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 3–20. ACM Press, October / November 2017.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[Mon23] The Monero Project. Monero: Secure, private, untraceable cryptocurrency. https://www.getmonero.org/, 2023.

[Net23] Phala Network. Phala: Confidential smart contracts with trusted execution environments. https://phala.network/, 2023.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, Berlin, Heidelberg, March 2009.

[Oas] The Oasis standard body. http://www.oasis-open.org.

[Par21] Partisia. Partisia blockchain: A web 3.0 public blockchain built with mpc for trust,transparency, privacy and speed of light finalization. 2021.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, August 1992.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 94–124, Virtual Event, August 2021. Springer, Cham.

[SBG+19] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1759–1776. ACM Press, November 2019.

[Sec] Secret. Secret network graypaper.

[SV15] Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 2015*, volume 9092 of *LNCS*, pages 3–22. Springer, Cham, June 2015.

[SWA23] Ravital Solomon, Rick Weber, and Ghada Almashaqbeh. smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption. In *2023 IEEE European Symposium on Security and Privacy*, pages 309–331. IEEE Computer Society Press, July 2023.

[XCZ+23] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VeriZexe: Decentralized private computation with universal setup. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 4445–4462. USENIX Association, August 2023.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[Zam] Zama. Zama fhevm: Fully homomorphic encryption for the ethereum virtual machine. https://github.com/zama-ai/fhevm.

[ZDH19] Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2PC over a blockchain with applications to financially-secure computations. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 633–650. ACM Press, November 2019.

[ZH17] Ruiyu Zhu and Yan Huang. JIMU: Faster LEGO-based secure computation using additive homomorphic hashes. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 529–572. Springer, Cham, December 2017.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Berlin, Heidelberg, April 2015.

Appendix A. Separating sequencing and execution in privacy preserving blockchains

Privacy-preserving blockchains bring an innovative edge by enabling the processing of encrypted data while selectively decrypting ciphertexts when dictated by the transaction logic. This includes the ability to re-encrypt data toward a user-owned key, ensuring that only the intended recipient can access sensitive information.

However, this powerful capability introduces a critical challenge to validators. In blockchains like Ethereum, validators maintain the integrity of the network by producing canonical blocks—-those that accurately record state changes from one block to the next. Without the ability to decrypt or re-encrypt, validators in a privacy-preserving blockchain lose this essential function.

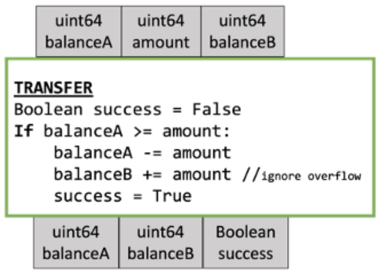
On the flip side, granting validators (collectively) decryption powers introduces a significant risk of privacy leakage. Validators could irreversibly decrypt sensitive transaction data before it is confirmed as part of a finalized block. This could expose confidential information from transactions that ultimately never make it into the canonical chain, compromising the privacy guarantees that the blockchain aims to uphold.

To address this, the gcVM introduces an innovative safeguard: non-canonical blocks must achieve finality before they are handed over to the execution sub-network. Only at this stage does decryption become permissible, ensuring that sensitive data is accessed only when its integrity and inclusion in the blockchain are fully confirmed.

This approach strikes a delicate balance between preserving user privacy and maintaining the network's integrity, setting a new standard for privacy-focused blockchain solutions.

Appendix B. Illustration of soldering

In Figure 6 we present the basic functionality of a token transfer in the confidential ERC20 standard. The functionality is given the sender's and receiver's balances $\texttt{balance}_A$ and $\texttt{balance}_B$ as well as the `amount` to be transferred from A to B. The functionality first verifies that $\texttt{balance}_A \geqslant \texttt{amount}$, and if successful, it transfers the amount, by decreasing $\texttt{balance}_A$ and increasing $\texttt{balance}_B$ by `amount`.



```
TRANSFER
Boolean success = False
If balanceA >= amount:
    balanceA -= amount
    balanceB += amount //ignore overflow
    success = True
```

Figure 6. The Confidential ERC20 transfer functionality.

When done in secure computation, we do not wish to leak to everyone whether $\texttt{balance}_A \geqslant \texttt{amount}$, therefore, both paths are executed, and the balances are updated according to the comparison result, using the `MUX` operation.

When executed in the gcVM, the evaluators hold garbled circuits for operations `GTE, ADD, SUB` and `MUX`, and they wish to pass values from the output of one circuit to the next; see Figure 7. In the online soldering technique, the evaluator conduct the soldering protocol between execution of every two circuits, whereas in the offline soldering technique they conduct the soldering protocol for all circuits at once, before evaluation begins, and then evaluate all circuits as if they were one big circuit.

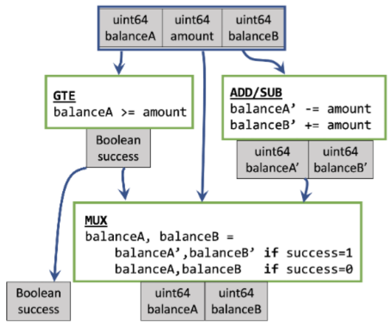


Figure 7. Confidential ERC20 transfer via soldering. CERC20 incurs three `Onboard` operations, to decrypt ciphertexts `balanceA, amount, balanceB`, one `Transfer` operation that performs the calculation insider the green boxes, and two `Offboard` operations to encrypt the updated `balanceA, balanceB`.

Appendix C. Bit Authentication via Information theoretic MAC

Consider a scenario in which a dealer $\mathcal{D}$ gives a bit b to a receiver $\mathcal{R}$ ('sending phase'), who later wants to prove to a verifier $\mathcal{V}$ that b is indeed the bit sent from $\mathcal{D}$ ('validating phase'). This can be easily solved by having $\mathcal{D}$ sending a digital signature on b which can be transferred to V at the validation phase. However, this can be done more efficiently if we allow some interaction between $\mathcal{D}$ and $\mathcal{V}$ in the sending phase. Specifically, we can use an information theoretic MAC (IT-MAC) as follows:

- Setup: $\mathcal{D}$ and $\mathcal{V}$ agree on a global random IT-MAC key K.
- Sending phase:
  - $\mathcal{D}$ picks a random one-time pad $P[b]$ and computes $M[b] = P[b] \oplus b \cdot K$.
  - $\mathcal{D}$ sends $P[b]$ to $\mathcal{V}$ and $(b, M[b])$ to $\mathcal{R}$.
- Validating phase:
  - $\mathcal{R}$ sends $M[b]$ to $\mathcal{V}$.
  - $\mathcal{V}$ learns the bit $b \in \{0,1\}$ for which $P[b] = M[b] \oplus b \cdot K$ (i.e., it computes both cases and checks which one holds). If this is not the case for neither $b = 0$ nor $b = 1$ then $\mathcal{V}$ rejects and aborts.

Since K is unknown to $\mathcal{R}$ and a new pad $P[b]$ is used each time a bit should be authenticated, the MAC value $M[b]$ does not leak information about K. Similarly, $P[b]$ is a random value independent of b, therefore, b remains secret to $\mathcal{V}$.

IT-MAC is XOR-homomorphic.

- Given pads $P[b], P[c]$ to $\mathcal{V}$ and MACs $M[b], M[c]$ to $\mathcal{R}$, $\mathcal{V}$ can receive and validate the authenticity of $d = b \oplus c$ without learning b or c individually. This can be done by having $\mathcal{R}$ send $M[d] = M[b] \oplus M[c]$ to $\mathcal{V}$, who checks if $P[b] \oplus P[c] = M[d] \oplus K$ for some $d \in \{0,1\}$.
- Given an authenticated bit b (i.e., $\mathcal{R}$ has $(b, M[b])$ and $\mathcal{V}$ has $P[b]$), it is possible to obtain the authenticated bit c where $c = b \oplus \mathsf{a}$ for some public bit $\mathsf{a} \in \{0,1\}$ without interaction:
  - If $\mathsf{a} = 0$, then $c = b$ and so everything remains the same, i.e., authentication of c is equal the authentication of b.
  - If $\mathsf{a} = 1$, then $c = b \oplus 1$. It is simply done by having $\mathcal{V}$ compute $P[c] = P[b] \oplus 1 \cdot K$ and $\mathcal{R}$ leave $M[c] = M[b]$. MAC verification go through because $P[c] = P[b] \oplus K = M[b] \oplus b \cdot K \oplus K = M[b] \oplus (b \oplus 1) \cdot K = M[b] \oplus c \cdot K$ as required.

In our 2-evaluators protocol. In the context of our protocol, evaluators E_1 and E_2 are associated with global MAC keys K_1 and K_2. The dealer is the garbler and each evaluator plays both as a receiver and a validator.