g EVM v.0.1

Sod L bs

1 Introduction

Ethereum virtual machine (EVM) is the leading blockchain-native virtual machine, interleaving computer architecture aspects with incentive mechanisms, which made it the first state-transition engine for a decentralized and permissionless general system. One of the main obstacles to the mass adoption of the EVM is its lack of confidentiality, leaving all information public, which is obviously not in par with what businesses, communities, and individual expect from a system. To this end, we introduce the gcEVM, an extension to the EVM that support confidentiality by offering a series of functionalities for keeping private data and performing operation on it without ever exposing it (unless required by the execution itself). Because of the nature of the EVM, where everything is visible, handling ciphertexts must be meticulously done , in order to protect those ciphertexts from theft, replication, etc. We demonstrate our solution via a cryptographic primitive from the field of secure computation, called *garbled circuit*, hence the extension is called gcEVM.

In the rest of this section we give the necessary background on the EVM and argue about the importance of confidentiality in the EVM for a real impact. We briefly describe the notion of garbled circuits and garbling protocols in Section 2. Then, in Section 4 we describe our EVM extension that relies on a garbling protocol.

1.1 Background on the EVM

The EVM's ccounts and State. In a very high-level, the EVM takes an ordered list of valid transactions and execute them. Execution of transactions may change the state of the machine and so, in an abstract way, we refer to this execution process as the state-transition function of the machine. The state of the EVM is composed of many sub-states, each is associated with an *account* (also known as *address*); and these accounts may be either *external* or *internal*. n external account (also known as externally owned account, or EO) is an account that may initiate a transaction whereas an internal account (also known as a smart contract) only reacts to requests that were initiated by EO s. In the following we use the terms 'internal account' and 'smart contract' interchangeably. The sub-state associated with every account contains its *balance* and *nonce*, where balance refers to the number of coins that account 'owns' and the nonce refers to the number of transactions that account has issued so far, such that each newly issued transaction increment that number by exactly one; the latter is used as a protection from replay attacks – so that no transaction may by launched twice. Since internal accounts (smart contract) cannot really initiate transactions, their nonce is incremented only when they trigger the creation of a new internal account, thus, the nonce of a smart contract simply refers to the number of other smart contracts that they have created. While the sub-state associated with an EO consists of its balance and nonce only, the sub-state of a smart contract may additionally contain an arbitrary data structure along with a set of interfaces (or methods) that can change its sub-state (either its balance, nonce, or the additional data structure).

Transactions may be simple or complex; in a simple transaction an EO transfers some amount of the native coin to another account (external or internal); in a complex transaction an EO may either create a new smart contract (also referred as 'smart contract deployment) or call a method of an already existing smart contract. In the latter, the reaction of the smart contract to that call may involve further calls to methods of (potentially other) smart contracts, and so on. This way, a complex transaction initiated by an EO may trigger a chain-reaction that causes changes in multiple sub-states. On the other hand, a complex

transaction may also cause no change to any sub-state at all. When a method is said to be a *view-method* it is guaranteed that it never changes any sub-state, which implies that it can call only other view-methods.

The EVM's Execution, Memory and Storage. Execution in the EVM is fueled by gas, which is paid in the native coin (Ether in Ethereum) and serves as a measure of computational effort. This gas is paid from the balance of the EO that initiates the transaction. Each operation within the EVM, from arithmetic calculations to data storage, consumes a certain amount of gas, incentivizing efficiency and preventing network abuse (e.g., mounting a denial-of-service attack by sending an infinite stream of transactions).

Memory management in the EVM is unique; it maintains a volatile memory store (simply called 'memory' hereafter) during execution but does not retain it after the execution completes. This memory is linear (instantiated with a stack data structure) and expands as needed by a contract's execution but is wiped clean after the process ends. For persistent storage, the EVM utilizes a key-value store known as 'storage', which persists between transactions but is significantly more costly in terms of gas to utilize. This design encourages developers to optimize their use of storage and memory, balancing the need for persistent data against the gas costs of operations, ensuring that the EVM remains scalable and efficient. The storage is the one that manages the EVM's state, and so each sub-state is maintained as an isolated key-value store.

Let us describe a typical execution process: When a smart contract method is called, and it needs to read data from its storage (e.g., a variable value), the EVM fetches this data from the contract's storage (a sub-state) and loads it into memory for quick access during this specific execution. This is done through specific EVM opcodes such as `sload`, which reads data from the variable's location at the storage. The data read is then available in memory for processing or computation during the contract execution. fter the smart contract performs computations or manipulates data within the memory, there may be a need to persist some of this data back to storage. To store data back from memory to storage, the contract uses opcodes like `sstore`. This opcode takes the data from memory and writes it to the specified location in the contract's storage.

Data Passing Between Smart Contracts. The EVM facilitates data passing from one contract to another via a function call, it employs a mechanism that allows contracts to interact and invoke functions on each other. This process is foundational to the composability and interoperability of smart contracts on the Ethereum platform.

When a contract (caller) wants to invoke a function on another contract (callee), it specifies the callee's address and the function to be called, along with any arguments required by that function. This can be done using opcodes like `call` and `delegatecall`. The data passed to the callee includes information about the function to be executed (identified by its signature) and the arguments for that function. The callee contract then executes the specified function using the provided arguments. This execution can read from or write to the callee's storage, depending on the function's logic, and eventually gets back to the caller contract using the opcode `ret`. run-time variable, called `depth`, is incremented on every function call (to a different contract) and decremented when the function returns to the caller.

Error Handling. The notion of *reverting* a transaction or a contract call is a fundamental concept designed to ensure the integrity and security of smart contract operations. revert operation undoes all changes made to the state during a transaction or call, except for the consumption of gas, and returns an error message to the caller. This mechanism is crucial for handling errors and ensuring that failed transactions do not alter the blockchain state in an unintended manner. revert error is triggered intentionally by using the `revert` statement in Solidity or when conditions specified by `require` statements are not met. It's used to handle logical errors in contract execution, where a certain condition or prerequisite is not satisfied. The gas consumed up to the point of revert is not refunded, but any remaining gas is.

n `assert` function is used in Solidity to handle internal errors and to check for invariants within the code. If an assert statement fails, it indicates a serious bug in the contract code, leading to a 'panic' error. Unlike revert, assert failures consume *all* the gas provided with the transaction, signaling a more severe form of error that should not occur during normal operation.

Other types of error are possible in the EVM, like out-of-gas, stack overflow or underflow, invalid opcode, and more, each has a different cause and consequences on the EVM.

1.2 The EVM's (lack of) Confidentiality

One of the most controversial property of the EVM is that *everything is public*, meaning that the sub-state of smart contracts, which may include financial, social (and practically any kind of) information, is completely visible to all. s we argue below, this property of the EVM is a double-edged sword, which, for a long time, formed a dichotomy between decentralization and adoption.

On the one hand, this fact played as key-contributor to the *decentralization* of Ethereum (who is the first to deploy an EVM), as it allows anyone with a computer to run it and take part in the execution of agreed upon lists of transactions (with an adequate financial incentive mechanism); thereby increasing the validity of the system's state, or in other words, increasing the trust that the information laying in the state is the product of correct execution of the past transactions.

On the other hand, the fact that everything is visible to all poses a significant setback to the *usability* and *adoption* of the EVM. To date, the EVM's prominent use-case is de-fi, which paved the way to collaborative liquidity pools and automatic movement of funds. In many cases however, de-fi is used by bad actors for scam (e.g., 'rug pools'), fraud (e.g., money laundering), and many times is considered as 'funds streaming in a loop' with no real impact. rguably, one of the reasons for this is the *lack of confidentiality*, leaving honest actors (smart contracts, D Os and users) unable to prove and verify each other's identity, thereby being more vulnerable to exploits and manipulations. Furthermore, the lack of confidentiality sets an obstacle to real social impact, as most of real world social activities deal with private information that must be treated adequately. For example, an election requires the independence of votes and the freedom to vote one's opinion without fearing any consequences. Sealed bid auction, as another example, requires an independence of bids as well as their confidentiality, since public bids exposes the bidder's sensitive financial state.

Privacy Enhancing Technologies. Privacy has always been a pivotal aspect of human life, cherished across all ages. However, in the current era, marked by the internet revolution, its significance has magnified. Society is racing to keep pace with technological advancements to ensure personal information remains confidential. In today's digital world, where nearly every action leaves a trace, safeguarding our privacy is not just important—it is essential. We find ourselves constantly navigating the fine line between harnessing technology's benefits and protecting our private lives from intrusion.

The mid-20th century's introduction of sophisticated cryptography signaled a crucial shift towards secure communication. This period was defined by the urgent need to transmit messages safely across potentially compromised mediums—be it physical documents carried by unreliable couriers or digital data transmitted through the airwaves or wires vulnerable to interception. The burgeoning field of cryptography focused on devising methods that guaranteed not only secure communication but also the efficiency and resilience of these communications against eavesdropping or tampering. This era witnessed cryptography's evolution from an arcane practice to a fundamental, science-driven toolkit essential for military units, governments, and eventually the general public, shaping the complex digital security landscape we navigate today.

By the late 20th century and in subsequent decades, the cryptographic community experienced a revolutionary shift in its research focus, heralding the era of secure multiparty computation (MPC). This cryptographic breakthrough allowed multiple parties to collaboratively compute functions over their private data without revealing the actual data to one another. These advancements extended beyond theoretical significance, offering profound practical implications. They facilitated the secure exchange and processing of information in a manner that preserved both privacy and confidentiality, addressing a growing concern in an increasingly interconnected world. From enabling confidential electronic voting systems to secure data sharing among organizations, multiparty computation marked a considerable advancement. This period of research broadened the horizons of cryptography beyond traditional encryption and decryption, catering to the nuanced requirements of a society ever more reliant on digital interactions and the perpetual challenge of balancing technological convenience with the imperative to preserve privacy.

Opting for a garbled-circuit-based MPC to achieve on-chain privacy aligns with several critical metrics:

- **Modularity**. The garbled-circuit-based solution is structured into two independent phases, termed 'Garbling' and 'Evaluation'. The Garbling phase, involving significant computation by the network nodes, is conducted 'offline' in a pre-processing stage, producing a garbled circuit—a secure 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 exceedingly efficient manner. This modular approach is elegant and establishes a 'privacy supply chain'.

- **Security**. midst various encryption schemes, our solution aspires to align with industry standards right from the start, rather than introducing a proprietary, untested encryption scheme or waiting for a lengthy standardization process. We employ encryption schemes already widely adopted by the world's most secure systems, including those managed by governments and large banks. This approach involves standardized symmetric-key schemes for encryption (e.g., ES-CTR) and standardized asymmetric-key schemes for key distribution (e.g., RS), enhancing adoption by eliminating the need for additional, unverified security assumptions. Contrary to other MPC methods, garbled-circuit-based solutions facilitate an efficient integration of these encryption schemes within a circuit that can be securely evaluated in a distributed manner.

  Privacy. In recent years, numerous initiatives have sought to enhance on-chain privacy via the powerful cryptographic tool known as zero-knowledge proof (ZKP). While ZKP allows data owners to verify the correctness of statements about their data without revealing the data itself, it falls short in scenarios involving multiple data owners who wish to collaborate based on their private data. This is vital for a range of blockchain applications, from dynamic identity systems and DeFi applications like MM to portfolio management, social trading, peer-to-peer messaging, auctions, and governance. Our approach to on-chain privacy is driven by a secure MPC protocol, where users' data is sent to a private data pool. Here, any process, public or private, can be applied to the data without disclosing anything but the process result as intended by the application developer, and only with user consent.

- **Performance**. With the objective of optimizing real-time transaction processing involving private data, having pre-prepared garbled circuits enables nodes participating in the Evaluation phase to achieve a high transaction throughput. The low-latency characteristic of garbled-circuit-based MPC ensures that the number of communication rounds between nodes is constant and does not depend on the number of parties involved or the complexity of the transaction. Crucially, the technologies underpinning both the Garbling and Evaluation phases are ready for implementation on current devices (including smartphones), without the need for specialized hardware or awaiting significant advancements in research.

- **End-user experience**. Maintaining an unaltered security experience (as highlighted in the security discussion), submitting private data to the network resembles sending data over a TLS channel, employing a symmetric-key encryption scheme. This means devices today are already equipped to interact with the network using standard protocols and widely known software libraries. This compatibility extends to software operating on personal computers, web browsers, smartphones, smart sensors, and potentially any IoT device.

This holistic approach to on-chain privacy underscores our dedication to ensuring security, privacy, efficiency, and a user-centric experience, thereby setting a new standard for privacy in the digital domain.

2 Garbled Circuits

In this section we use a variation of the notation and definitions from '*Foundations of Garbled Circuits*' by Bellare, Hoang and Rogaway [BHR12].

2.1 Circuits

For simplicity, we consider circuits with fan-in of two, even though our solution is not limited to those.

circuit is a six-tuple $f = (n, m, q, A, B, G)$, where $n \geq 2$ is the number of *inputs*, $m \geq 1$ is the number of *outputs*, and $q \geq 1$ is the number of *gates*. The inputs, wires, outputs, and gates sets are indexed by $\mathsf{Inputs} = \{1, \ldots, n\}$, $\mathsf{Wires} = \{1, \ldots, n+q\}$, $\mathsf{Outputs} = \{n+q \quad m+1, \ldots, n+q\}$, and $\mathsf{G\ tes} = \{n+1, \ldots, n+q\}$, respectively. Then, the functions A and B are of the form $\mathsf{G\ tes} \to \mathsf{Wires} \setminus \mathsf{Outputs}$, where $A(g)$ (resp. $B(g)$) returns the first, or left, (resp. second, or right) incoming wire of gate g. Finally, G is a function of the form $\mathsf{G\ tes} \times V^2 \to V$, where V is the domain of values that wires can take. Here V is defined abstractly while typically it is defined by finite group, ring or field. For instance, many times V is instantiated by the domain $V = \ _2 = \{0, 1\}$ and the function G, which define a binary (Boolean) gates; alternatively, it can be defined by $V =$ for some finite field and G, which define arithmetic gates over .

The above embodies the following. Gates have fan-in of two (two inputs), arbitrary functionality, and arbitrary fan-out (an output wire may serve as an incoming wire to unlimited number of gates). The wires are numbered 1 to $n + q$. Every non-input wire is the outgoing wire of some gate. The i-th value of an input is presented along wire i. The i-th value of an output is collected off wire $n + q \quad m + i$. The outgoing wire of each gate serves as the name of that gate. Output wires may not be input wires and may not be incoming wires to gates. No output wire may be twice used in the output. Requiring $A(g) < B(g) < g$ ensures that the directed graph corresponding to circuit f is *acyclic*, and that no wire twice feeds a gate; the numbering of gates comprise a topological sort.

Circuit evaluation. The canonical evaluation function $\mathsf{ev_{circ}}$ takes a circuit f and a list of inputs $x = x_1, x_2, \ldots, x_n$ and returns a list of outputs $x_{n \ q \ m \ 1}, \ldots, x_{n \ q}$. See Lisging 1 for a formal description:

lgorithm 1 Canonical Evaluation $\mathsf{ev_{circ}}(f, x)$

1: $(n, m, q, A, B, G) \leftarrow f$.
2: **for** $g \leftarrow n + 1$ to $g \leftarrow n + q$ **do**
3: $\quad a \leftarrow A(g)$ and $b \leftarrow B(g)$
4: $\quad x_g \leftarrow G(g, x_a, x_b)$
5: **end for**
6: **return** $x_{n \ q \ m \ 1}, \ldots, x_{n \ q}$

2.2 Garbling Schemes

garbling scheme is a five-tuple of algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$, where Gb is probabilistic and the rest are deterministic. Let $f = (n, m, q, A, B, G)$ be a circuit that we wish to garble. Recall that f represent a function of the form $V^n \to V^m$. On input f and a security parameter $\kappa \in \mathbb{N}$, the garbling algorithm Gb returns the triple $(F, e, d) \leftarrow \mathsf{Gb}(1^{\ }, f)$, where e describes an encoding function, $\mathsf{En}(e, \cdot)$, that maps an initial input $x \in V^n$ to a *garbled input* $X = \mathsf{En}(e, x)$; F describes a *garbled circuit*, $\mathsf{Ev}(F, \cdot)$, that maps each garbled input X to a *garbled output* $Y = \mathsf{Ev}(F, X)$; and d describes a decoding function, $\mathsf{De}(d, \cdot)$ that maps a garbled output Y to a final output $y = \mathsf{De}(d, Y) \in V^m$.

Projective garbling schemes. common approach in existing garbling schemes is for e to consist of a vector of sets of *labels*, such that a set of labels L_i is associated with the i-th input or output wire ($i \in \{1, \ldots, n\} \cup \{n+q \quad m+1, \ldots, n+q\}$). For example, if the circuit is Boolean then we have $V = \{0, 1\}$ and there are two labels for the i-th input wire, namely, $L_i = \{L_i^0, L_i^1\}$. The encoding function $\mathsf{En}(e, \cdot)$ then uses the values $x = x_1, \ldots, x_n$ to select from $e = (L_1, \ldots, L_n)$ the subvector $X = (X_1, \ldots, X_n) = (L_1^{x\ }, \ldots, L_n^{x_n})$.

2.2.1 Example: Secure Two-Party Computation (2PC) via Garbled Circuits

Garbling schemes were originally designed as a solution for secure two-party computation, which work as follows. Suppose lice has n_A inputs, denoted $x_A = (x_1, \ldots, x_{n_A})$ and Bob has n_B inputs, denoted $x_{n_A\ 1}, \ldots, x_{n_A\ n_B}$, where $x_i \in \{0,1\}$ for all i, and $n = n_A + n_B$. lice and Bob wish to securely compute the circuit $f = (n, m, q, A, B, G)$ over their joint inputs $x_1, \ldots, x_n$. lice and Bob can use a garbling scheme to do so, as shown in lgorithm 2 in which lice and Bob take the roles of the Garbler and Evaluator, respectively. The garbler (lice) generates the garbled circuit F and its own garbled input X_A, and sends both to the evaluator (Bob). Then, the evaluator obtains its own garbled input X_B via a cryptographic protocol called oblivious transfer (OT). Finally, using the garbled circuit and the garbled inputs, the evaluator can obtain the garbled output Y and using the decoding information d it can obtain the actual (plaintext) output y.

lgorithm 2 Secure Two-Party Computation $f(x_A, x_B)$

1: Interpret $(n, m, q, A, B, G) \leftarrow f$.
2: lice (the garbler) computes $(F, e, d) \leftarrow \mathsf{Gb}(1\ , f)$ (κ is the security parameter).
3: lice computes her garbled input $X_A = \mathsf{En}(e, x_A)$.
4: lice sends F, d and X_A to Bob.
5: lice and Bob invoke a two-party protocol called *oblivious transfer* (OT). In this protocol lice privately has e and Bob privately has x_B, and Bob (and only Bob) obtains the output $X_B = \mathsf{En}(e, x_B)$.
6: Bob computes $Y = \mathsf{Ev}(F, X)$ where X is the concatenation of X_A and X_B.
7: Bob computes $y = \mathsf{De}(d, Y)$ which is the output of the computation. t this stage Bob may share that output with lice; in case Bob is suspicious of being malicious other security measurements are in place.

Security notions of garbling schemes. Garbling schemes may be associated with different security guarantees, like *privacy*, *obliviousness*, *authenticity*, and more; in our context we are mostly interested in privacy and authenticity. In a high level, *privacy* refers to the fact that the evaluator is unable to tell which actual (plaintext) value passes through wires of the circuit, even after having the garbled circuit and the garbled inputs (unless it has prior information about those values). uthenticity means that the evaluator cannot produce garbled output Y' that is different than the correct garbled output Y that is obtained through honest evaluation of the garbled circuit.

Extension to Secure Multiparty Computation (MPC). The above is an example of how to distribute the computation of f between two parties where each party holds part of the inputs. In our context, we are interested in secure computation that can be executed by many parties where the private inputs themselves might not even be theirs, e.g., private input may contain some identity information of a client (who is not necessarily running an execution node). In a very high level, this is solved by two techniques, the first is *cryptographic secret sharing* and the second is *multiparty garbling protocol*. In such a solution, an external client is the one who holds the private input and the execution parties are responsible for holding it securely and performing secure computation over it when needed.[1]. The client's private data is stored by the execution parties in a way that requires many of them to behave maliciously in order to disclose that data, otherwise, no information can be inferred about that data. With regard to computation over such secret shared data, the parties invoke a garbling protocol that can be run by many parties. Various protocols for multiparty garbling have been proposed in the literature since the '90, optimizing for various metrics like communication rounds, bandwidth and computation complexity. The protocol that we use for the gcEVM (see next section) is fundamentally different than those protocols, although building on similar techniques.

[1] Comput tion over client's priv te d t must be under the client's consent, constr int we de l with in the next sections.

3 General Secure Multiparty Computation via Garbling

Given a projective garbling scheme as described above, in the following we show how to construct a general secure computation for multiple parties, that can capture a high-level programming language. Denote the parties involved in the computation by $\mathcal{P} = \{P_1, \ldots, P_N\}$ and define two (potentially overlapping) sets $\mathsf{G\ rblers} \subseteq \mathcal{P}$ and $\mathsf{Ev\ lu\ tors} \subseteq \mathcal{P}$. We rely on the fact that the adversary may corrupt at most $t_G < |\mathsf{G\ rblers}|$ parties of the $\mathsf{G\ rblers}$ and at most $t_E < |\mathsf{Ev\ lu\ tors}|$ parties of the evaluators. Note that a corrupted P_i who is both garbler and evaluator contributes to both t_G and t_E. We begin with a description of some necessary functionalities that our protocol relies on, and an MPC protocol for a stand-alone computation of some function f that is known to the parties ahead of time. Then, we present a stateful protocol that can be capture a runtime execution environment for any machine, and can capture computation of functions/programs to be known only in 'real time'.

3.1 Secure Computation for Function

Given a projective garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$, our MPC protocol makes use of two 'internal' MPC protocol; these internal protocols may be instantiated with any MPC construction from the literature (e.g., based on secret-sharing techniques, other garbling schemes, fully-homomorphic encryption, or any other cryptographic technique). The functionalities to be realized by these internal protocols are $\mathcal{F}_{\mathsf{G}}$, the *garbling functionality*, $\mathcal{F}_{\mathsf{En}}$, the *encoding functionality*, and $\mathcal{F}_{\mathsf{De}}$, the *decoding functionality*, described in Listings 3,4 and 5, respectively. For some data structure x, that can be a vector, list, key-value store, etc., we denote by $[x]$ the result of a threshold secret sharing of x, where each receiver obtains only a share of x.

lgorithm 3 Garbling Functionality $\mathcal{F}_{\mathsf{G}}{}^{,\mathcal{G}}$

Parties: the functionality interacts with $\mathsf{G\ rblers}$ and $\mathsf{Ev\ lu\ tors}$.
Parameters: a computational security parameter κ and a garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$.

1: Upon receiving $(\mathbf{g\ rble}, f)$ from the $\mathsf{G\ rblers}$:
   - Compute $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$.
2: **return** $F, [d]$ and $[e]$ to the $\mathsf{Ev\ lu\ tors}$.

lgorithm 4 Encoding Functionality $\mathcal{F}_{\mathsf{En}}{}^{,\mathcal{G}}$

Parties: the functionality interacts with the $\mathsf{Ev\ lu\ tors}$.
Inputs: For each input wire $i \in \{1, \ldots, n\}$, the parties either know the public input x_i or have a sharing of it, namely $[x_i]$ is held by the $\mathsf{Ev\ lu\ tors}$. Let $\mathsf{PublicIdx}, \mathsf{Priv\ teIdx} \subseteq \{1, \ldots, n\}$ be the indices of the wires for which the input is public and private, respectively, then $\mathsf{PublicIdx} \cap \mathsf{Priv\ teIdx} = \emptyset$. Let $x = (x_1, \ldots, x_n)$.
Parameters: $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$.

1: Upon receiving $(\mathbf{encode}, [e], (x'_1, \ldots, x'_n))$ from the $\mathsf{Ev\ lu\ tors}$, where $x'_i = x_i$ if $i \in \mathsf{PublicIdx}$ and $x'_i = [x_i]$ if $i \in \mathsf{Priv\ teIdx}$, reconstruct e and x_i for all $i \in \mathsf{Priv\ teIdx}$ and compute $X = \mathsf{En}(e, x)$.
2: **return** X to the $\mathsf{Ev\ lu\ tors}$.

Note that in the functionalities' description we omit some details, like whether the **garble** command can be triggered by only a subset of $\mathsf{G\ rblers}$, and how is the functionality interacts with the adversary.

The MPC protocol that uses the above functionalities is given in Listing 6.

3.2 Secure Computation of a Future Function

Secure computation of a future function refers to the mode of operation in which the function f to be computed is not known at the time of garbling, yet, we want to enable the $\mathsf{Ev\ lu\ tors}$ run a secure computation

lgorithm 5 Decoding Functionality $\mathcal{F}_{\mathsf{De}}^{,\mathcal{G}}$

Parties: the functionality interacts with the Ev lu tors.

Inputs: The parties have a garbled output Y. The parties also hold a sharing of a decoding information $[d]$.

Parameters: $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$.

1: Upon receiving $(\textbf{decode}, [d], Y)$ from the Ev lu tors, reconstruct d and compute $y = \mathsf{De}(d, Y)$.

2: **return** y to the Ev lu tors.

lgorithm 6 MPC Protocol $^{,\mathcal{G}}$

Parties: the functionality interacts with all parties $P_1, \ldots, P_N$ (which consists of G rblers $\cup$ Ev lu tors).

Inputs: For each input wire $i \in \{1, \ldots, n\}$, if $i \in \mathsf{PublicIdx}$ or $i \in \mathsf{Priv\ teIdx}_j$. In the latter case it means that x_i is a private input of party P_j.

Parameters: κ and $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$.

1: Upon receiving $(\textbf{compute}, f)$ from the parties:

- The G rblers send $(\textbf{g rble}, f)$ to $\mathcal{F}_{\mathsf{G}}^{,\mathcal{G}}$, upon which the Ev lu tors obtain $F, [d]$ and $[e]$.
- Party P_j shares $[x_i]$ to the Ev lu tors for every $i \in \mathsf{Priv\ teIdx}_j$.
- The Ev lu tors send $(\textbf{encode}, [e], x'_1, \ldots, x'_n)$ to $\mathcal{F}_{\mathsf{En}}^{\mathcal{G}}$ where $x'_i = x_i$ if $i \in \mathsf{PublicIdx}$ and $x'_i = [x_i]$ otherwise, and obtain X.
- The Ev lu tors locally compute $Y = \mathsf{Ev}(F, X)$
- The Ev lu tors send $(\textbf{decode}, [d], Y)$ to $\mathcal{F}_{\mathsf{De}}^{\mathcal{G}}$ and obtain y.

2: **return** y to the Ev lu tors.

of an arbitrary function whenever needed.

In stateful computation the parties are 'aware' to some instruction-set (or opcodes) to be supported, e.g., `add, mult`, etc.

To enable the parties to run computation that is unknown in the offline (or preprocessing) phase, which is when G rblers produce the garbled circuits in the form of $F, [e], [d]$, we use an additional functionality called 'soldering', denoted $\mathcal{F}_{\mathsf{Solder}}^{\mathcal{G}}$. The soldering functionality allows the composition of small garbled circuits into one larger garbled circuit. The functionality takes as input two garbled circuits $(F_1, [e_1], [d_1])$ and $(F_2, [e_2], [d_2])$, and a set of indices I_1 and I_2, such that output wires with indices I_1 of the first circuit should be 'soldered' to input wires with indices I_2 of the second circuit. When wire w_1 is soldered to wire w_2, it means that the same (secret) value will pass through them at time of evaluation. The soldering functionality enables the Ev lu tors (who receive the two garbled circuits) to solder the wires without further intervention of the G rblers.

s an example, recall that each wire in the garbled circuit is associated with a vector of labels. For instance, consider the Boolean case in which the wires are associated with two labels, one that represents the value 0 and another that represents the value 1. Consider garbled circuits F_1 and F_2 such that output wire w_1 in F_1 should be soldered with input wire w_2 in F_2. The soldering functionality is given the two garbled circuits (along with their encoding and decoding information $[e]$ and $[d]$), and returns some (public) information that tells the parties how to, at the time of evaluation, translate from the label received on w_1 to the correct label on w_2. Let the labels associated with w_1 and w_2 be $(L^0_{w\ }, L^1_{w\ })$ and $(L^0_{w_2}, L^1_{w_2})$, respectively, then the soldering functionality returns $_{w\ \ 2}$ such that $\mathsf{Solder}(L^0_{w\ }, {}_{w\ \ 2}) = L^0_{w_2}$ and $\mathsf{Solder}(L^1_{w\ }, {}_{w\ \ 2}) = L^1_{w_2}$, where Solder is the translation procedure. Note that whether the Ev lu tors will have $L^0_{w\ }$ or $L^1_{w\ }$

The protocol that enables the computation of an arbitrary function f follows: In the preprocessing phase, the G rblers send $(\textbf{g rble}, f_{\mathsf{opcode}})$ to $\mathcal{F}_{\mathsf{G}}^{,\mathcal{G}}$, upon which the Ev lu tors obtain $F_{\mathsf{opcode}}, [d_{\mathsf{opcode}}]$ and $[e_{\mathsf{opcode}}]$ many times for every opcode in the instruction set. The evaluators store those garbled circuits for a later use, when they know what is the function f. t that time, they represent the function f as a 'computation-tree' that

is composed of 'atomic' opcodes from the instruction set. For example, if the function has four variables, x_1, x_2, x_3, x_4, and it returns $(x_1 + x_2) \cdot (x_3 + x_4)$ then the computation-tree has four leaves (on level 0), x_1, x_2, x_3, x_4, the nodes in level 1 are `add` (that connects x_1, x_2 and `add` (that connects x_3, x_4, and the node in level 2 is `mult` that connects the results of the two nodes in level 1. For each connection, identify the output and input wires that should be connected and apply the soldering functionality on these wires to obtain the translation information. Finally, obtain the garbled input X for the inputs x_1, x_2, x_3, x_4 and begin evaluating the circuit as above. Whenever need to move from an output wire w_1 of one node (garbled circuit), whose obtained label is L_w , to the input wire w_2 of the next node (in a higher level), use the translation information $_{w\ ,w_2}$ obtained from the soldering functionality to get $L_{w_2} = \mathsf{Solder}(L_w\ ,\ _{w\ ,w_2})$, from which it is possible to continue evaluation of the next garbled circuit.

4 The gcEVM

The gcEVM involves extension of the EVM in multiple dimensions. First, we introduce new data types in order to capture the fact that information of this type must be kept secret; then, we introduce new operations that can perform manipulation on secret data types without disclosing the secrets; and finally, we must take extra care on the way we manage and protect these new data type against attackers who wish to mount some sort of a replay attack. We discuss all these topics in this section.

4.1 gcEVM-Related Data Types

Similar to other systems, all the information in the EVM sub-state, including the balance, nonce, and anything residing in the data structure maintained by smart contracts, is stored in atomic *typed* variables, namely, variables that capture a certain type of information, be it small or large, signed or unsigned integers, strings, or bytes. Let `Types` be the set of data types supported by the EVM.

For the purpose of supporting confidentiality, we introduce a new set of data types, denoted `STypes`, that is analogous to `Types`; each data type in `STypes` is basically the secure version of one data type from `Types`. For example, we have $\texttt{uint8}, \texttt{uint16}, \texttt{uint32}, \texttt{uint64} \in \texttt{Types}$, then their secure version are $\texttt{suint8}, \texttt{suint16}, \texttt{suint32}, \texttt{suint64} \in \texttt{STypes}$. Generally speaking, the secure version of an EVM data type will have the same name, prepended with the letter '`s`' (to indicate a `s`ecret). smart contract developer must use these types if it wishes the underlying information to remain secret.

These data types in `STypes` are all referred as an abstract data type, called `CT` (for 'ciphertext'), which is essentially a re-definition of `uint256`, whereas data types in `Types` are referred as `PT` (for 'plaintext'). Looking ahead, having a ciphertext in a smart contract state, or in the memory during an execution does not necessarily mean that it is *authenticated*; the gcEVM must make sure that a ciphertext is authenticated before entering it into any secure computation procedure.

We make a distinction between secret data types that are used for security 'at rest', 'in transit', or 'in use'. That is, while the *ciphertext* data type (denoted `CT`) are use to secure data at rest (be it the persistent storage or the volatile memory used in the course of an execution of a transaction), we use the *inputtext* data type (denoted `IT`) for protecting data in transit and the *garbledtext*data type (denoted `GT`) for protecting data in use.

Protecting data in transit means protecting the ciphertexts that a user wish to send to some smart contract function in a transaction. Specifically, the goal is that when a user incorporate some ciphertext in its transaction, this ciphertext will be used only in the context of this transaction and cannot be re-used in other transactions (by malicious actors). For example, if a user participates in a sealed bid auction and sends a ciphertext in its transaction that hides its plaintext bid, we must prevent an adversary from copying that user's ciphertext and submitting it as its own bid; furthermore, we must prevent an adversary from using that ciphertext in any way, so that the gcEVM will not agree to perform any secure operation on it.

Protecting data in use refers to the fact that even when data is secure on storage or on memory, its security might be broken when performing some operations on it, like using it within a secure computation protocol.

In the following we give a formal description of the three data types:

- **ciphertext (T).** This data type represents the result of a CP -secure encryption scheme and used for securing data at rest. It is the actual datatype visible in the system's state. Due to other security mechanisms employed in the system, like authenticated memory and storage, and the fact that decryption is performed to 'well formed' ciphertexts only, we do not need to use a CC -secure encryption scheme (the attacker does not get to choose the ciphertexts to be decrypted by the system).

  Formally, let $\mathsf{Enc} = (\mathsf{kgen}, \mathsf{enc}, \mathsf{dec})$ be a CP -secure encryption scheme, for a message $m = (\{0,1\}^{\ell})^*$ (i.e., the message length is a multiple of the encryption block length ℓ) we have:

  $$\mathtt{CT} = c \leftarrow \mathsf{enc}_k(m) \tag{1}$$

  where $(ek, dk) \leftarrow \mathsf{Enc.kgen}(1^{\kappa})$[2] and κ is the computational security parameter. Looking ahead, by default, instances of `CT` will be the result of encryption using the system's key, whereas some `CT`s will be the result of encryption using a client's key.

  The encryption scheme we use is `ES128` in the counter mode (CTR), thus, for a message $m = m_1 \| m_1 | \ldots$ (with $|m_i| = 128$) the encryption result is $c = c_0 \| c_1 \| \ldots$ where $c_0 = r$, $c_i = \mathtt{ES128}_k(r + i) \oplus m_i$, and r is chosen uniformly at random. CTR mode is advantageous as it allows a random access to a specific slot in the plaintext, and it only performs a forward evaluation of the underlying PRF (`ES`).

- **inputtext (`IT`).** This data type is a wrapper of `CT` only used to infiltrated data to the gcEVM from the outside world. The role of `IT` is to make sure that the wrapped `CT` is used only for the purpose it is intended to by the user who sent it. n `IT` may be formed of an authenticated encryption (e.g., using the encrypt-then-authenticate approach) or a signcryption;[3] in both cases the associated data being authenticated must contain the identities of the sender and the receiver. The gcEVM inherits the transaction format from the EVM and so every message is already signed, and the signature is applied on those identities, as required. The sender's identity (which is the user) is extracted from the signature itself, while the receiver's identity is combined of the contract address and the function within that contract to be invoked.

  The above suggests that `IT` can be in the exact same format as `CT`, however, there are subtleties that require us taking some extra care. Specifically, instead of fully relying on the signature on the transaction as a whole, we ask the user to individually sign each `CT` (as well as the identities). This is important for security at least for the support of view functions in a setting of a single gcEVM node (i.e., the entire system consists of a single node). Since invocation of view functions do not trigger the verification of a signature on the transaction (in fact, calls to view functions do not have to be signed at all), it means that an attacker may 'steal' an honest user's ciphertext: the attacker's contract will have a function like `leakData(CT c, address sender)`, which onboards the ciphertext `c` to the system's memory using the `sender`'s key, and then decrypts it, so that the plaintext hidden by `c` is revealed to everyone (and to the attacker in particular). The attacker now takes some ciphertext `c` sent to the gcEVM earlier by an honest user of address `user_addr`, and calls the above function with `leakData(c, user_addr)`, which reveals the value that the honest user intended to keep private. Signing each ciphertext individually prevent such an attacks.

  Then, we formalize an inputtext as follows. Let ct_m be a ciphertext for message m and let $\mathsf{Sig} = (\mathsf{kgen}, \mathsf{sign}, \mathsf{verify})$ be an unforgeable signature scheme; the inputtext format for m is:

  $$\mathtt{IT} = (ct_m, \sigma) = (ct_m, \mathsf{sign}_{sk}(d \| ct_m)) \tag{2}$$

[2] Since we use symmetric encryption scheme we h ve k ek k, but symmetric schemes m y be used in the s me w y

[3] See [KL20] for discussion bout uthentic ted encryption nd signcryption, nd [BSW06, SPW07] for sign tures with strong security; in [BMP22] they rgue th t `ECDS` h s strong security (lso c lled enh nced unforge bility).

where $(sk, vk) \leftarrow \mathsf{Sig.kgen}(1^\kappa)$, κ is the computational security parameter, and d encapsulates the identities, namely,

$$d = \texttt{user\_addr} \| \texttt{contract\_addr} \| \texttt{func}. \tag{3}$$

Given $\texttt{IT} = (ct, \sigma)$, before the gcEVM agrees to work with ct it must first check the identity of the sender and then decrypt ct using that sender's key. Specifically, this is done by:

$$m = \begin{cases} \mathsf{dec}_k(c) & \text{if } \mathsf{verify}_{pk}(d \| c, \sigma) = 1 \\ \perp & \text{otherwise} \end{cases} \tag{4}$$

We instantiate Sig with the `ECDS` scheme over `secp256k1`.

- **garbledtext(`GT`).** This datatype is used to securely handle data while in use. Unlike inputtext and ciphertext, garbledtext is in a form that is readily available for manipulation (e.g., making arithmetics over the plaintext it hides) which is made inside the garbled execution environment (see below).[4] Using a binary-projective implementation of a garbling scheme the garbledtext version of a message $m = (m_1, \ldots, m_\ell)$ where $m_i \in \{0, 1\}$, is `gt` of type `GT`, such that

$$\texttt{gt} = \mathtt{L}_1, \ldots, \mathtt{L}_\ell \tag{5}$$

where $\mathtt{L}_\ell$ is a κ-bit label (typically $\kappa = 128$). Overall, a garbled-text expands the underlying data by a factor of κ. This expansion has no effect on the long-term storage requirement of the system, as the lifetime of garbledtexts is short (it is only valid during the execution time of a transaction). The garbledtexts themselves are not utilized by smart contracts and are not appear in their raw form in the gcEVM memory or storage, instead, their *handles* are being used, where a handle of a garbledtext is simply a hash on the list of its labels; namely, $h_{\texttt{gt}} = H(\texttt{gt})$ is the handle of `gt`, where H is a hash function. We instantiate H by `Keccak`, which is a collision resistant hash function, therefore, the probability of two garbledtexts having the same handle is negligible.

  The nodes who take part in evaluation or verification of garbled circuits do obtain the raw representation of garbledtext (the labels). Those nodes store a map of the form $h_{\texttt{gt}} \rightarrow \texttt{gt}$; whenever a secure operation is invoked on garbledtext(s) the evaluation nodes perform the actual computation, which mostly results with another garbledtext, whereas verification nodes have the result of the computation and verify its correctness.

  The fact that evaluation nodes obtain the raw form of garbledtexts forces the protocol design to assume the worst-case scenario, that the attacker obtains them too (e.g., an attacker who corrupts an evaluation node). Thus, the system must protect itself from *theft of garbledtexts*. For example, suppose that a corrupted evaluation node knows that `gt` is a garbledtext result of some secure operation in the next block, then it may inject $h_{\texttt{gt}}$ to a function of some contract in the next block, such that the function performs decryption of that garbledtext. Fortunately, the garbling scheme ensures that all garbledtexts are *unpredictable* and are only known to the evaluation nodes at the moment of evaluation and never before.

4.2 The gcEVM Data-Flow

Before delving into technical details, let us describe the data-flow at a high level, which is also depicted in Figure 1.

In order to preserve security in the course of the execution, the network and the users maintain multiple keys:

[4] We note th t in FHE-b sed solutions there is no distinction in the represent tion of hidden d t t rest nd in use.

- *Network key*: nk is the network symmetric key. This key is being distributively generated on the network's startup (via a key-generation protocol) which results with a key share nk_i to MPC node number i. refresh protocol is applied to the network key, which results with a new key share nk'_i to the nodes, but the secret key nk itself remains the same; such mechanism is intended to protect from adaptive adversaries, who can corrupt a dynamic subset of the parties at every given moment, and thwart accumulation of their power. In addition, a re-generation of the network key would take place from time to tiem, according to the network's policy (e.g., upon accumulation of additional 20% of staked funds); this re-generation protocol would generate a new network key nk^* and re-encrypt all ciphertexts under this new one.

- *User key*: part from the secret signing key that users usually maintain in their wallet, the user has a symmetric key uk with which it enters new data to the network. Like the network key nk, the user key uk is distributively generated and is secret shared among the network nodes (so node number i holds share uk_i). Thus, when a new data is to be entered (e.g., to a function of a smart contract) the user encrypts it and the function asks to decrypt and use it. s will be shortly explained, such a decryption does not reveal the plaintext to the function (or in public in any way), rather, it transform the data into a garbledtext, which enables the function to securely operate on it.

- *Key-retrieval key*: This is a *public-key* that is generated by a user in order to retrieve its symmetric key from the network. This is done by a generation of asymmetric encryption and decryption keys ek, dk by the user, and submitting ek to the network (via a transaction). Upon receiving the encryption key ek, the network encrypts the user's symmetric key uk under the temporary public encryption key ek, and returns the encryption result to the user. The user then uses the secret decryption key dk to decrypt that message and obtain the symmetric key uk that is also maintained by the network. The user can now use uk in order to enter encrypted data to the network.

The network maintains its own symmetric encryption key as well as a symmetric encryption key for every user that wants to benefit from data privacy in the system. In contrast to a signature key-pair, which is generated at the user and can be used without any on-boarding process, the encryption key for each user is generated by the network and can be used by the user only after on-boarding, which entails a simple query to the system to generate its key (or to retrieve it if it already exists) via the key-retrieval key. Note that the key may be generated prior to the user's query, in cases a smart contract already performed an encryption toward that user (meaning that the smart contract decided that some information should be decipherable by that user only).

For a user to bring encrypted data to the gcEVM, it has to encrypt it using its own symmetric key, and sign it using its own signing key. These two form an inputtext (as detailed above). Once this inputtext reaches a function of a contract, the function first has to verify its authenticity and that the inputtext has landed where the sender (user) really intended it to land; the result of a successful verification will be a garbledtext that is ready to work with (it can serve as an input for secure operations) inside the garbled execution environment (GEE). The verification procedure is according to Equation 4 above.

The above is one path to the GEE; a second path to the GEE would be to 'onboard' a ciphertext, which turns a ciphertext into a garbledtext (both hiding the same plaintext). Ciphertexts reside in the state of each contract and 'belong' to the contract where they reside, meaning that a function on one contract cannot request the onboarding of a ciphertext in another contract, which is critical for the security and privacy of users' data.

4.3 uthentic Memory and Storage

It is crucial to ensure the authenticity of all three data types within the operation of the network, safeguarding them against any potential malicious manipulation. Given the transparency of the blockchain, *inputtext* and *ciphertext* are susceptible to malicious copying or unauthorized acquisition. Contrarily, we argue that *garbledtexts* cannot be predicted, copied or obtained outside the transaction's execution.

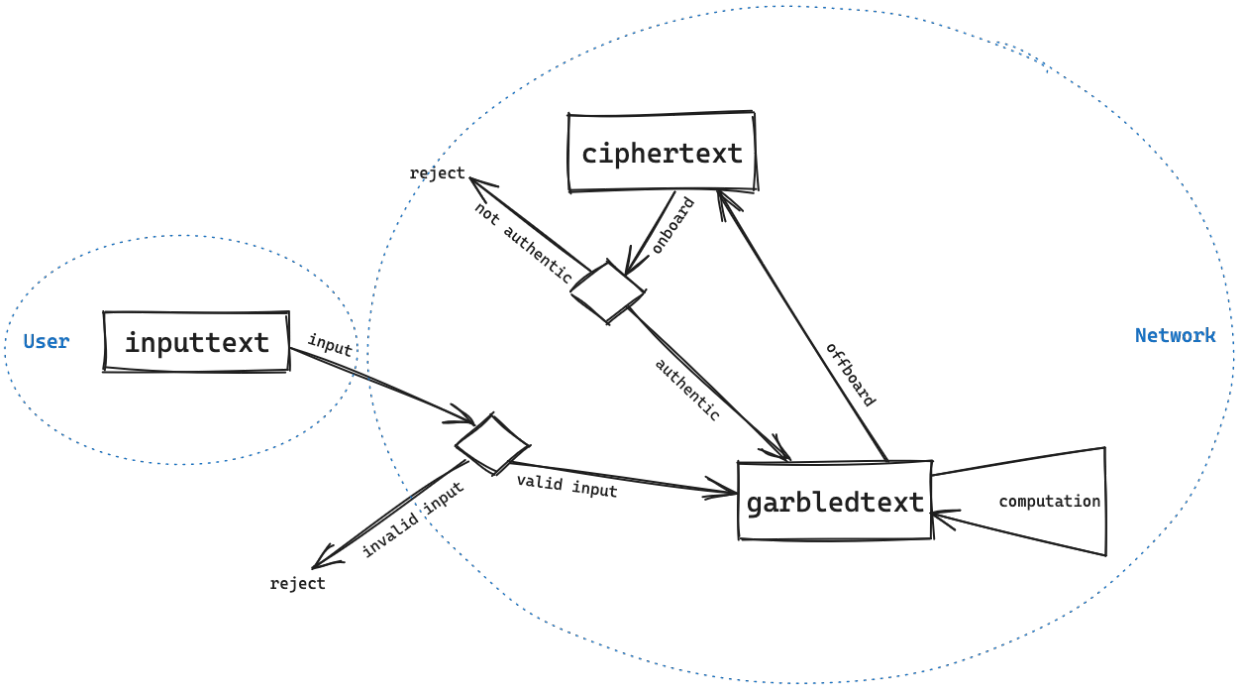

Figure 1: Transition between data types.

Generally speaking, the challenge of protecting private data in the context of blockchains mostly deals with ensuring an adequate and tight access control to those ciphertexts. The use of the plaintext behind an inputtext or a ciphertext must be permitted only if this usage complies with the intentions of the contract that is owning or receiving them, and the users who contributed those inputtexts.

In a high level, the system employs three types of protection mechanisms: *authenticated storage*, *authenticated memory* and *garbled execution environment* (GEE).

- uthenticated storage protects ciphertexts 'at rest'; each ciphertext is associated with one or more contract addresses in a way that permits onboarding of the ciphertext only in the execution of those contracts; this means that copying a ciphertext from one contract to another is futile. By copying a ciphertext *ct* we mean either hardcoding the content of *ct* into another contract, or transmitting it within a transaction as an argument.

- uthenticated memory is the vehicle between the storage and the execution environment. The function first loads the ciphertext from storage into the execution environment memory, from which it may turn into a garbledtext. On the other way around, to store the plaintext value behind a garbledtext to storage, it has to be offboarded first into a ciphertext that resides in memory, from which it is actually stored in the storage.

- Ciphertexts cannot be transferred between contracts, which means it is useless to pass a ciphertext as an argument between different contracts. The way to pass private information between contracts is for the caller contract to turn the ciphertext into a garbledtext first, which puts it in the garbled execution environment, and then pass the garbledtext to the callee contract, this way, the garbledtext is available for secure manipulation by the caller as well. Garbledtexts are protected by the fact that they are random and unpredictable values generated 'on-the-fly'.

uthenticated Memory & Storage. Memory refers to the EVM run-time memory, which is stack-based, and storage refers to the EVM persistent storage, which is maintained per address. Recall that only contracts' addresses (i.e., to exclude EO addresses) are associated with actual storage, whereas EO s are associated with the their balance and nonce only. For a function to perform some computation on a state variable, it has to first load it from the contract's storage and to know its exact location in the storage. To do that, the function provides the variable's location `loc` to the `sload` opcode, which triggers the EVM to execute it and push the variable's content to the memory stack. On the other way around, for a function to persistently save some value into some state variable, it provides the location `loc` of that variable as well as the value to the `sstore` opcode.

The EVM keeps track of the `depth` of the execution, that is, when an EO calls some function `func1` on a certain contract `contract1`, the execution of `func1` begins at `depth` $= 1$; if that function calls function, say `func2`, on another contract, say `contract2`, then `depth` changes to 2 (and changes back to 1 when `func2` returns to `func1`), and so on. Note that function calls within the same contract do not change `depth`.

The gcEVM provides an extension to the 'normal' EVM memory and storage operation described above, which we call *authenticated memory* and *authenticated storage*.

The authenticated memory maintains a map of the form

$$\mu : \mathbb{N} \to \mathtt{CT}^*,$$

that is, for each execution depth the map maintains all authenticated ciphertexts for that depth. To check whether a ciphertext ct is authenticated for depth d, we check if $ct \in \mu(d)$. In our context, a ciphertext may arrive in memory by either loading it from storage via the `sload` opcode or as a result of the `Offboard` mechanism (which, given a garbledtext, returns a ciphertext).

uthenticated storage is maintained in a per-contract basis; each contract, say on address `addr`, is associated with another contract at address `addr′` that contains only storage (and no bytecode). The relation between `addr` and `addr′` must be one-to-one, so that a malicious entity would not be able to associate another address `addr″` to neither `addr` or `addr′`. The associated contract at `addr′` forms the authenticated storage of the contract at `addr`; this authenticated storage is only accessible from the EVM and not by the contract developer. If a valid ciphertext ct resides at location `loc` of the storage of address `addr`, then the authenticated storage, of address `addr′`, contains ct at the same location `loc`. We must ensure that a user cannot cause this to happen on invalid ciphertexts.

The authenticated memory and storage adhere to the following rules, which are also depicted in Figure 2.

1. *Load from storage.* ciphertext ct resulting from `sload` applied to a storage at location `loc` that is performed in depth d is first checked against the authenticated storage. If the authenticated storage has ct in location `loc` as well then ct is added to the set $\mu(d)$, otherwise ct is *not* add to $\mu(d)$ (but it is pushed to the normal memory).

2. *Onboard.* When Onboard is invoked by a function at depth d on cipher ct, if ct is authenticated for depth d (i.e., $ct \notin \mu(d)$) then it is being translated into a garbledtext and that garbledtext is returned to the caller; otherwise, the execution is reverted.

3. *User input.* s explained above, a user input is encrypted by its own symmetric key and upon verification (of authenticity) it is turned directly into a garbledtext, readily available for secure operations.

4. *Public input.* contract might want to input some public input into the secure execution environment (GEE), which means that value has to be turned into a garbledtext. This is done via the special opcode 'SetPublic'.

5. *Offboard.* ciphertext ct resulting from the `Offboard` mechanism (applied to a garbledtext) that is performed by a function in `depth` $= d$ is added to the set $\mu(d)$, upon which we say that ct is *authenticated for depth d*.

Loc tion is lso known s 'key' in the context of the EVM stor ge, s the stor ge is simply key-v lue store.

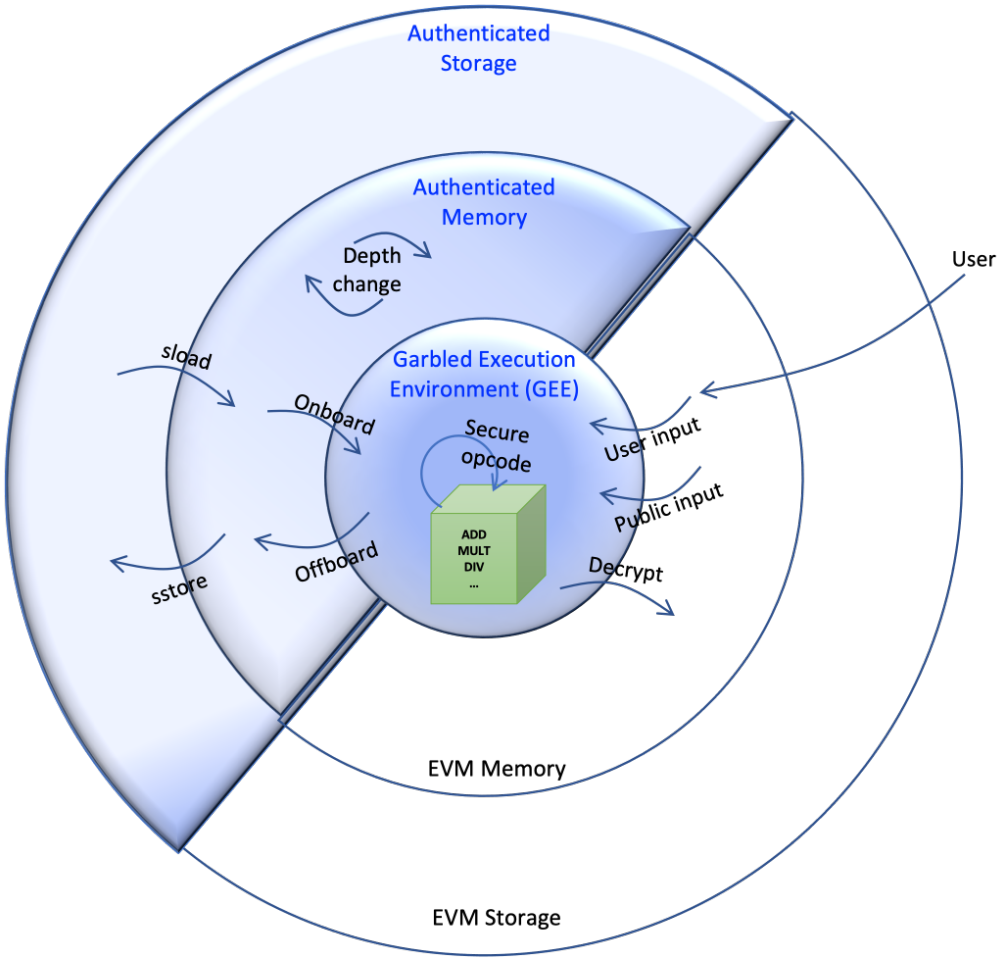

Figure 2: Overview of the gcEVM. The colored part represents the security and privacy extension to the normal EVM, which is represented by the white part.

6. *Offboard to user.* When a contract wishes to disclose some value only to a specific user, that value is turned from garbledtext into a ciphertext that is encrypted under that user's key (the symmetric key uk), and that ciphertext is not considered authenticated (it is placed in memory but is not added to $\mu(d)$).

7. *Decrypt.* When a garbledtext is decrypted, the plaintext value is returned directly to the normal (non-authenticated) memory.

8. *Store to storage.* When `sstore` is called by a function at `depth` $= d$, to store ct at location `loc`, if $ct \in \mu(d)$ then, in addition to writing ct to location `loc` of the normal storage, write it to location `loc`

in the authenticated storage as well, otherwise (if $ct \notin \mu(d)$) write it only to the normal storage, and make sure the authenticated storage at location `loc` is empty.

9. *Cleaning memory.* When `depth` is decreased from d to $d' < d$ (when a function returns, reverts, etc.) the set $\mu(d)$ in the authenticated memory is cleaned. Similarly, when `depth` is increased from d to $d+1$ (on a function call), we make sure that $\mu(d+1)$ is empty.

10. *Immobility of ciphertexts.* n authenticated ciphertext cannot transit between depths, namely, calling a function (on another contract) with an argument of type `CT` would deem that ciphertext invalid (i.e., it will not be considered as authenticated for the new depth); similarly, when returning an argument of type `CT` to a caller function from another contract the returned value would not be considered authenticated. The right way to move such private values is by using the `GT` type.

The Garbled Execution Environment (GEE). s explained, a garbledtexts may be generated from an inputtext or a ciphertext, *only* upon confirming their authenticity. It is crucial to note that garbledtexts bear significance only in the course of the execution of the transaction. dditionally, their inherent safety is derived from the negligible likelihood of predicting them, given that they are formed of random values that are revealed to the parties only at the very moment of execution. This property allows us to treat them with ease rather than keep tracking them across function calls. Since the next batch of transactions to run is fixed before the garbledtexts are revealed, it is not possible for a user or for a contract to hardcode a garbledtext in their transaction or state, as garbledtexts are unpredictable, furthermore, these garbledtexts become useless once the execution of the transaction is completed; the garbledtexts for the next transactions batch would be completely fresh.

References

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the CM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, US , October 16-18, 2012*, pages 784–796. CM, 2012.

[BMP22] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. Efficient asymmetric threshold ECDS for mpc-based cold storage. *ePrint rchive*, 2022.

[BSW06] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational diffie-hellman. In *PKC*, pages 229–240. Springer, 2006.

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2020.

[SPW07] Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. How to strengthen any weakly unforgeable signature into a strongly unforgeable signature. In *CT-RS* , pages 357–371. Springer, 2007.