Auditable, Available and Resilient Private Computation on the Blockchain via MPC*

Christopher Cordi[1], Michael P. Frank[1], Kasimir Gabert[1], Carollan Helinski[1], Ryan C. Kao[1], Vladimir Kolesnikov[2], Abrahim Ladha[2], and Nicholas Pattengale[1]

[1] Sandia National Laboratories, Albuquerque, NM, USA
[2] Georgia Institute of Technology, Atlanta, GA, USA

Abstract. Simple but mission-critical internet-based applications that require extremely high reliability, availability, and verifiability (e.g., auditability) could benefit from running on robust public programmable blockchain platforms such as Ethereum. Unfortunately, program code running on such blockchains is normally publicly viewable, rendering these platforms unsuitable for applications requiring strict privacy of application code, data, and results.

In this work, we investigate using MPC techniques to protect the privacy of a blockchain computation. While our main goal is to hide both the data and the computed function itself, we also consider the standard MPC setting where the function is public.

We describe GABLE (Garbled Autonomous Bots Leveraging Ethereum), a blockchain MPC architecture and system. The GABLE architecture specifies the roles and capabilities of the players. GABLE includes two approaches for implementing MPC over blockchain: Garbled Circuits (GC), evaluating universal circuits, and Garbled Finite State Automata (GFSA).

We **formally model and prove** the security of GABLE implemented over garbling schemes, a popular abstraction of GC and GFSA from (Bellare et al, CCS 2012).

We analyze in detail the performance (including Ethereum gas costs) of both approaches and discuss the trade-offs. We implement a simple prototype of GABLE and report on the implementation issues and experience.

* This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsi-diary of Honeywell International, Inc., for NNSA under contract DE-NA0003525. This report describes objective technical results and analysis. Any subjective views or opinions that might be expressed in this report do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Approved for public release SAND2022-3529 O.

1 Introduction

Current programmable blockchain platforms such as Ethereum provide a "global computer," an always-on public computing resource. They guarantee reliability, availability and auditability for computations implemented as smart contracts, which are posted to the blockchain and subsequently executed. Even in the event of a widespread disaster, any still-functioning part of the Ethereum network renders this computing resource *available*. Organizations can take advantage of such a robustly available computing facility to execute particularly mission-critical computational tasks, if this computation can be done *privately*.

We note that the performance of secure multiparty computation (MPC) has been steadily improving and is practical today even for large functions/inputs.

In this work, we explore the motivation, use cases, architectures, and constructions for securing (i.e. ensuring privacy of) a sensitive computation done on sensitive inputs on a public blockchain network using MPC techniques.

We present GABLE (Garbled Autonomous Bots Leveraging Ethereum), an architecture and a system for running MPC on the Ethereum blockchain. We consider two different base approaches, garbled finite state machines/automata (GFSA), and garbled circuits (GC); we present both in a unified manner as garbling schemes. We show how functions can be computed privately. At a high level, in our architecture there are four types of players, or participant roles:

1. *Garbler* is a player who generates encrypted function and input encodings, publishes the former on Ethereum, and distributes the latter to other players. Garbler may be a single trusted entity or run distributedly, e.g., by an MPC over a private chain.
2. *Input Provider* or *Writer* is a player who contributes (encrypted) inputs to the computation.
3. *Input Unlocker* or just *Unlocker* is a designated player that manages encrypted inputs, preventing input providers from learning unauthorized information about the computation and adapting their input based on it.
4. *Output Recipient* or *Reader* is a player who may obtain a designated output from the computation.

We emphasize that players may play several of these logical roles simultaneously (with some exceptions; cf trust model Section 2.3). For example, an input provider may also be an output recipient.

1.1 Motivation

Existing public programmable smart-contract blockchains such as Ethereum provide a highly robust and accessible computing platform. An entity whose operations may require execution of business or strategic logic that needs to function (using inputs from various sources) with a high degree of assurance of its reliability and availability may implement that functionality as a smart contract running on a blockchain. Another desirable blockchain property is auditability, due to the generally immutable nature of data committed to a consensus chain.

However, the deploying entity may wish to keep private the following computation data and metadata:

- Input values provided to the computation, including their semantics.
- The nature of the computation that is being performed on the inputs. Most generally, we may wish for all information about the structure and function of the computation to be obscured.
- Any internal data within the computation.
- Semantics of any intermediate or final outputs produced by the computation.

Most of these features are implied by standard MPC properties, while hiding the computed function is the goal of Private Function Evaluation (PFE).

Use cases. We outline two use-case scenarios which illustrate the potential usefulness and practicality of our system. Of course, many others are possible.

Sealed-bid auctions: auditability and security. Consider a simple sealed-bid auction with simultaneous bidding, where the soundness of the computation may be fully audited after the fact, if needed. This may be useful, e.g., in public-interest auctions, such as auctioning airwaves to cellular providers, or electricity delivery contracts to utilities. Other cases (e.g., auctioning of a privately-held company to a select set of bidders) may demand strict privacy.

While such auctions may be easily run via an MPC, running them on a blockchain offers a much higher level of transparency and trust (and user engagement) than more traditional means of execution. We report some analysis of costs for our methods on simple multi-bidder auctions in Sec. 5.

Transactive energy refers to market-based mechanisms for managing the exchange of energy in a wide-area electrical grid [30]. Participants in such a market, e.g., utility companies, may wish to engage in automatic electronic negotiation without revealing their strategies. A capability such as GABLE could provide a solution. Each participant deploys their own garbled bot expressing their private negotiation strategy; these bots then negotiate with each other on the public blockchain, while hiding their internal negotiation strategies.

Private-public chain integration. The encrypted function may be generated (garbled) by an MPC executed over a private network. This assures trust in the garbling, facilitates future opening of the secure computation (auditing) if needed (cf Section 2.3), and integrates public and private chains.

1.2 Contribution

Blockchain and MPC technologies provide essentially complementary features to a computation. A blockchain assures availability (ability to provide input, compute, and retrieve the output when needed) and reliability (including assurance that only authorized players can submit input and that the output of the computation is correct), even if direct communication channels among the parties

are cut. MPC ensures privacy and correctness, guaranteeing partial reliability (output correctness) but nothing about availability.

Our system, GABLE (Garbled Autonomous Bots Leveraging Ethereum) combines the best of the two technologies. In our security model, we require not only the data, but, optionally, also the computed function to be private (standard MPC evaluates publicly known functions). In this work, we:

- Design a general architecture for available and resilient MPC over the Ethereum blockchain. It allows authorized players to submit inputs at any time, and to retrieve outputs as soon as they are available. Our design specifies the roles and capabilities of the players.
- Design two approaches for implementing MPC on the blockchain: Garbled Circuits (GC), including GC evaluating universal circuits, and Garbled Finite State Automata (GFSA).
- *Formally state and prove* the security of our system against malicious players. Note, we *do not formally define* the properties of auditability, availability and resilience of the computation. They are derived from the corresponding intuitive properties of the underlying Ethereum blockchain. We specifically discuss achieving auditability in Section 2.3.
- Analyze in detail the performance characteristics (including gas costs) of both approaches and discuss the trade-offs.
- Implement a simple prototype and several demonstration applications, and report on the implementation issues and experience. (A reference implementation of our initial prototype has already been publicly released, and wider licensing/availability of a more complete code base is under consideration.)

1.3 Results and evaluation

We prototyped GABLE using GFSA and experimented on an Ethereum testnet. See Section 5 for details of our experiments and results; here, we summarize our findings. First, a simple provenance-tracking application scenario was modeled using 5 cycles of reactive functionality in a simple 6-state FSA; the cost to run this demo on the Ethereum mainnet would have been less than US$3.00 on the day of the test. Next, we demonstrated a 5-state machine implementing MPC for the classic 2-party "Millionaire's Problem" for configurable input lengths based on a bit-serial comparison algorithm; the cost of that demo would have been about $0.50 per bit of input, and can be further reduced. Finally, we compared costs for GFSA versus a garbled universal circuit (GUC) implementation on a multi-bit auction problem, showing that the cost of GUC grows only modestly (polylogarithmically) with the number of bidders, as expected, and becomes less expensive than GFSA for $B > 7$ bidders, at which point the cost is about $20 per bit of price data. Note, GC *without* functional privacy would cost far less.

1.4 Outline of the Paper

We presented the motivation and several use cases above in Section 1.1, and previewed our contribution and results in Section 1.2 and Section 1.3. We present

preliminaries in Section 1.5 and discuss related work in Section 1.6. We provide a high-level overview of our approach in Section 2, including defining the logical players, the trust model, and security intuition. A generic security statement and proof are outlined in Section 3, and specialized for our main GC-based protocol in Section 4. We discuss our prototype and demo implementations and present results and cost analysis in Section 5 and conclude in Section 6. Appendix presents some technical details of the GFSA approach used in our early implementations.

A much more detailed report on the implementation and demos, including reference source code for the prototype, is available as a Sandia National Laboratories technical report [19].

1.5 Preliminaries

Blockchain technology. Bitcoin [26] is a stunningly successful technology, which generalizes public timestamping (ledger) and smart contracts work and uses proof-of-work, concepts considered before [22,17,27]. Bitcoin has limited support for programming digital smart contracts, i.e. it is not Turing-complete. This is in part due to the possibility of intentional or accidental resource overuse or even exhaustion as a result of programming issues. The need for a rich programming language for contracts was nevertheless recognized, and in 2013, the Ethereum blockchain was proposed [13]. Ethereum addressed the issues stemming from language Turing-completeness by putting the onus on the programmer/contract creator, and requiring payment per storage unit and execution step. We implement our system for Ethereum; our higher-level design is general and will fit most natural blockchain architectures.

MPC (including Two-Party Computation, 2PC, and the general p-Party Computation), is an area of cryptography that studies protocols which compute functions over players' private inputs while revealing nothing except the function output. MPC has improved dramatically over the past 15 years. The first proof-of-concept 2PC implementation, Fairplay [25], evaluated only 200 Boolean gates per second. Today, 2PC implementations can process up to 2–5 million gates/sec [34]. Improvements in the malicious model and in 3PC and MPC are even more impressive. Recent work reports 3PC techniques that can evaluate as many as *seven billion* Boolean gates/second [6]. Research on algorithms and implementations has firmly transitioned MPC from a theoretical curiosity to a subject of practical engineering.

We note that 2PC protocols, and specifically Yao's garbed circuit (GC) techniques [32], are most suitable for use in our setting, because the blockchain itself can naturally serve as the GC evaluator.

FSA (finite-state automata) comprise a standard but simple model of computation that differs from Boolean circuits. The FSA model is weaker (some functions require exponentially larger representation in FSA as compared to circuits). The primary benefit of GFSA, in the case of sufficiently low-complexity applications,

is simply that obscuring the structure of the computation becomes relatively trivial, since all computations reduce to a linear sequence of state transition table lookups, and relatively simple techniques suffice to obscure the topology of the state graph. As a result, the overhead to achieve functional privacy in GFSA becomes less than that of garbled (universal) circuits for computations operating on sufficiently small numbers of bits.

Garbling schemes and Garbled Functions (GF). We build our approach around GC and GFSA. They are special cases of garbling schemes, as defined by Bellare et al. [9]. Informally, we will refer to garbling schemes as *garbled functions* (GF), similarly to how "GC" refers to both the GC garbling scheme and the GC approach. We will use the terms "garbled" and "encrypted" function interchangeably in this work. The BHR framework defines a garbling scheme as a tuple of algorithms $\mathcal{G} = (\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$[3] and requires that they satisfy the following properties:

In addition to the correctness property, BHR define relevant notions of security for garbled functions: privacy, obliviousness and authenticity. We refer the reader to [9] for precise definitions of these standard notions. Here, we informally summarize these notions.

GF *correctness* guarantees correct evaluation if all players behave honestly.

GF *privacy* guarantees that an adversary `Adv` who sees the garbled function (e.g. the GC), the encoded inputs and the output decoding information, does not learn anything beyond the result of the computation.

GF *obliviousness* guarantees that an `Adv` who sees the garbled function and the encoded inputs, does not learn anything. This notion is different from privacy, which gives `Adv` the decoding information and allows it to obtain the output of the computation (and nothing else).

GF *authenticity* captures `Adv`'s inability to create from a GF F and its garbled input X a garbled output $Y \neq F(X)$ that will be deemed authentic.

Note, we only need correctness and privacy. The authenticity requirement can be avoided if we choose to rely on the blockchain to honestly evaluate GF. Obliviousness becomes unnecessary if the output decoding information d is always published (e.g., d is provided to output receivers, who may be corrupted by `Adv`), in which case we are never in the setting without d available to `Adv`.

MPC from GF. Bellare et al. [9] do not systematize the ways one can obtain an MPC protocol from GF. However, the following (informally presented) natural 2PC construction works, and is proven secure against semi-honest adversaries in [9], assuming GF is private:

Construction 1 (2PC from GF, informal)**.** `Gen` *generates GF* F*, encoding information* e*, and decoding information* d *by running* Gb*.* `Gen` *sends* F, d *to* `Eval`*.* `Gen` *and* `Eval` *securely (e.g. via Oblivious Transfer (OT)), deliver to* `Eval` *the*

[3] In the BHR notation, ev is a reference evaluator for plaintext functions, Gb is the Garbler, En is the input encoder, Ev is the garbled function evaluator, and De is the output decoder.

labels of e corresponding to players' inputs. `Eval` *evaluates GF by running* Ev*, and obtains the plaintext output from garbled output labels and d by running* De*.*

We note that the above construction is secure against malicious `Eval`, as long as label delivery remains secure against a malicious `Eval`. We will use this property in our security argument.

1.6 Related Work

We briefly discussed relevant MPC and FSA preliminaries in Section 1.5. In this section, we review several systems addressing privacy on the blockchain and compare them to our approach.

MPC+blockchain. As we discuss next, many works explore the interplay of blockchain and MPC. To our knowledge, only YOSO (You Only Speak Once) MPC [10,20] formally models a *public* blockchain executing MPC. YOSO deviates from the typical blockchain architecture (e.g., of Ethereum) of all nodes sharing the same view. Instead, YOSO nodes have private data and are selected to perform MPC subtasks. If sufficiently large fraction of selected players are honest, MPC is secure. To protect against adversarial corruption, these players are hidden: they are unpredictably self-selected (e.g., via mining-like process), and each MPC subtask consists of computing and sending a *single* message, after which they erase their relevant private state. The main technical challenge of YOSO MPC is sending encrypted messages (e.g. containing internal state and subtask computation output) to *unidentified* players who are self-selected in the future. While YOSO MPC has attractive asymptotic complexity, unfortunately, it is concretely prohibitively expensive due to the cost of its building blocks. Our solution, at the cost of much stronger corruption and trust model (e.g., we only handle non-adaptive corruptions, while YOSO supports adaptive), is far more efficient and aligns with Ethereum architecture.

On the other hand, permissioned networks, such as Hyperledger, may be run by a small number of semi-trusted servers, and MPC can naturally be executed among the servers to achieve full privacy of transactions and contracts. This direction is explored in [11]. Their approach does not extend to public blockchains, since an arbitrary number of adversarial nodes may participate in the public network. Our work can be seen as general MPC on a public ledger for a restricted use case, where the encrypted function is generated by an organization trusted by the participants (and whose honesty can be later audited).

Hawk [24] is an architecture for a blockchain that can support private data. It handles private data using a trusted *manager*, realized using trusted hardware, such as Intel SGX. The trusted enclave *may* be implemented via MPC. It is not clear who would be the MPC principals to achieve a reasonable trust model; further (and [24] acknowledges this) this would cause an impractical overhead.

The Enigma system [35] uses MPC protocols to implement support for private data on a blockchain. They use MPC off-chain to perform computation on shares

of data. We aim to run MPC on-chain for resilience, availability and auditability; Enigma's techniques will not achieve these properties.

A line of work explores the interplay of blockchain and (separately executed) MPC to achieve fairness in MPC or connect MPC to financial mechanisms directly [5,12,16]. Works such as [31] use blockchain to manage encrypted inputs to MPC perfomed by a separate trusted network. Ref. [15] considers a blockchain-hybrid MPC model (plain model with available ledger), and addresses foundational issues of MPC, such as concurrent composability, in this model. In contrast, in our work, the blockchain itself executes MPC.

Zero-knowledge proofs (ZKP) are widely used both in MPC and in blockchain. We note that public ledger nodes never prove anything (indeed, the underlying secret would then be known to everyone). Instead, ZKPs are used by off-chain entities, such as wallets, to prove correctness of their actions. Several ledgers, such as ZCash, provide transaction privacy based on ZKPs. This line of work is ortohogonal to the privacy protection work we consider.

Solidus [14] uses a publicly verifiable ORAM machine to generalize and scale up the ZKPs for the use case where financial institutions representing many accounts interact with a ledger.

Blockstream CA [29] use simple ZKPs in conjunction with additive homomorphic commitments to manipulate secret data on the ledger. Partial privacy can be achieved for very simple functionalities (for efficiency, we are constrained by additively-homomorphic encryption).

In contrast to the above approaches, our solution is general MPC.

Trusted enclaves. As in the Hawk example above, privacy can be achieved if one is willing to entrust hardware enclaves, such as SGX. Nodes of the blockchain network may be equipped with enclaves, which would execute encrypted contracts on encrypted data. Several other systems, such as Secret Network [4], also implement this approach. We note that enclave security is a cat-and-mouse game; in this work, we do not rely on secure enclaves.

2 Overview: Approach and Trust Model

As discussed in Sec. 1, we wish to add privacy of both computations and data to the process of contract execution on the Ethereum network. Data and function privacy is normally achieved using an appropriate secure computation protocol. However, in the public blockchain setting, the number of network nodes is unspecified, and MPC privacy guarantees cannot be achieved. Instead, we take the following approach:

2.1 Logical players and evaluation pattern

We consider several logical players:

– The *Contract creator* or *Garbler* sets up encryptions of functions and inputs. It initializes the contract and sends encrypted labels to corresponding input providers. Garbler can be run by an MPC, e.g. over a private chain.
– *Input provider* or *writer*. This player is authorized to interact with a published contract (which implements a GF) and provide (garbled) input into the contract based on the plaintext input it has.
– *(Input) unlocker*. This player facilitates secure input provision by establishing an extra decryption step (performed by the unlocker) of the submitted garbled input. This prevents input providers, who posses *both* input labels on each input wire, from decrypting the internals of the encrypted computation. Effectively, use of the unlocker (who we assume does not collude with input providers) implements a secure OT of the input label based on the input.
– *Evaluator*. This player (implemented by the blockchain itself) evaluates the GF by executing the contract created by the contract creator on garbled inputs provided by the input providers. (By its nature, the blockchain also generates an indelible public archive of the contract's execution, including garbled inputs and outputs.)
– *Output recipient* or *reader*. This player is authorized to receive the output of the computation. It is also possible to make the output available to all.

2.2 Approach

In our approach, the blockchain network itself plays the role of the *Evaluator* `Eval` of the GF (either a garbled circuit or a garbled FSA, in this work).

GF generation and contract publishing. The computed function is first represented as a Boolean circuit or FSA. Then it is garbled within the BHR framework [9], resulting in a GF (e.g., GC or GFSA).

The GF is assumed to be *honestly* generated by an agent of a contract creator, `Gen`. We note that `Gen` possesses all secrets of the encrypted function and therefore is able to infer the internal state of the (plaintext) computation, should it ever gain access to the encrypted evaluation. Therefore we assume that all the secrets of the (small and self-contained) computation performed by `Gen` are *securely deleted*[4]. That is, we assume that `Gen` produces GF F, encoding information e and decoding information d. Upon delivery (as we discuss next) of e and d to the blockchain network players, and of F to the contract, `Gen` securely erases all its state (perhaps except F and d). We note that secure deletion of `Gen`'s state is not needed if audit may be desired or it is allowable for `Gen` to inspect the details of the evaluation, such as inputs, intermediate states, etc.

Input provision. As plaintext input becomes available to input providers, they may enter the corresponding garbled input labels into the contract. To do this, in the GC case, they must have access to *both* garbled labels for each Boolean input. This would present a serious security problem if not addressed. A player who

[4] If we require auditability of MPC, this information must be securely stored instead of being deleted. Then, upon audit, the generated GC can be reconstructed, and its correctness and correctness of MPC execution verified. See Section 2.3 for details.

knows more than one label of a wire may infer unallowed plaintext information. In addition to passively learning private information, the attacker may adaptively substitute its input, thereby affecting the correctness of the computation as well.

We address this issue by introducing and using *unlockers*, logical players who help manage input labels. Thus, the process of input provision proceeds as follows (we specify it for the case of GC; the GFSA case is analogous):

1. `Gen` generates GF and corresponding input labels, w_i^0 and w_i^1, representing two labels for each Boolean input wire W_i. `Gen` encrypts these labels with unlocker key k_u. For each input wire W_i, `Gen` gives the two encryptions $Enc_{k_u}(w_i^0), Enc_{k_u}(w_i^1)$ to the input provider responsible for the wire W_i. `Gen` gives the unlocker key k_u to the unlocker associated with W_i.
2. When the input provider is ready to submit the (encrypted) input $b \in \{0,1\}$ on wire W_i, it publishes to the contract $Enc_{k_u}(w_i^b)$, the encrypted label corresponding to its input b, received from `Gen`.
3. When notified (e.g., off-chain or by the blockchain, or in response to monitoring the blockchain), the unlocker retrieves $Enc_{k_u}(w_i^b)$ from the contract, decrypts it with the key k_u received from `Gen`, and publishes w_i^b to the contract.

Secure evaluation and output delivery. Once all inputs are provided to the contract, the contract is evaluated by the blockchain and the (encrypted) output is produced. Anyone may inspect the encrypted output, and only authorized players (those who received d, or corresponding portions of d from `Gen`) may decrypt and obtain the plaintext output.

Reactive functionalities. We stress that the computation need not be one-shot. It is natural to consider multi-staged evaluation, where intermediate outputs may be provided to output recipients, and function state propagated across the stages. This is easy to achieve with obvious variations of GF evaluation. One approach to this is illustrated in Fig. 1. We prove security only for one-shot functionalities. Proofs can be naturally extended to the reactive case.

2.3 Trust model

After having described the players and their actions, we are now ready to specify the trust model. There are two main assumptions:

- We assume that the contract generator acts honestly and *securely erases* its state after completion of its task. Note, this is immediately achieved if garbler is implemented as MPC e.g., run on a private chain.
- Input providers *do not collude* with corresponding unlockers. That is, we allow arbitrary collusions of players, but a set of colluding parties may not include an input provider and an unlocker for the same wire/GFSA step).

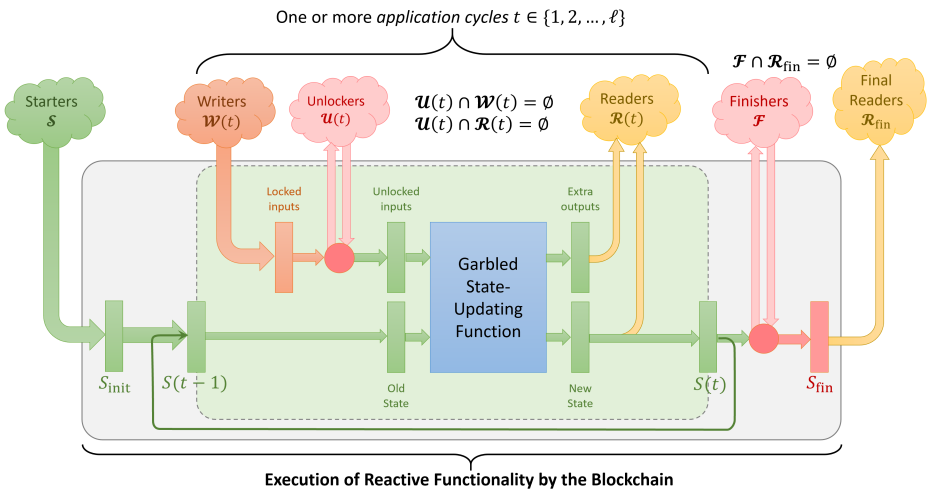

Fig. 1. ***Reactive execution of garbled machines by blockchain contracts.*** The gray region represents the full machine execution, which may require one or several contracts. The green sub-region represents operation within one *application cycle*, each of which may accept new inputs and produce new intermediate outputs. The blue rectangle represents a garbled state-update function that maps (old state, input) to (new state, output); this block can be implemented either using a (monolithic) garbled FSA transition table, or as a traditional GC (the latter requiring a *projective* or bit-vectorized state encoding). It must be garbled separately for each cycle. In this conception, $\mathcal{S}$ denotes a set of players called *Starters* authorized to configure the initial state, and each cycle t may have its own sets of authorized input providers $\mathcal{W}(t)$, unlockers $\mathcal{U}(t)$, and output recipients $\mathcal{R}(t)$. The Finishers $\mathcal{F}$ and Final Readers $\mathcal{R}_{\text{fin}}$ are only required if there is a final output that is supposed to be visible to a broad audience including the other players, but where the other players may have a disincentive to reveal the specific output value to that audience.

Security against cheating `Gen`*: audits, covert [8] and PVC [7,23] security.* We assume that `Gen` behaves honestly and, further, erases its state. In some scenarios, it may be desired to open the computation at a later stage, e.g., for audit purposes. This can increase trust in `Gen` and the transparency of the process. Of course, the auditor (or the public, if the computation is opened to the public) will learn the inputs of all players. Release of this information may be acceptable, e.g., in situations where inputs are sensitive only for a certain duration of time.

Auditing of `Gen` is easily achieved by requiring `Gen` to generate everything from a PRG seed and to securely store the seed. During audit, the seed is revealed and the auditor verifies that all actions of `Gen` are consistent with the seed (this may require participation of unlockers and input providers). Because GC is secure against a malicious evaluator, honest generation of GC implies correctness and security against malicious players in the collusion model described above.

In the case when the function is public, we can also easily achieve covert [8] or even publicly verifiable covert (PVC) [7,23] security. Following the ideas on [8],

covert security can be achieved by requiring `Gen` to produce two GFs and sets of inputs; the blockchain network, e.g. via a randomness beacon, challenges to open one of them, verifies its correctness, and evaluates the unopened GF. PVC, a strengthening of the covert model introduced by [7], requires the ability to prove cheating, in case a cheater was caught. Because the GF and all inputs are published on the chain, it is easy to collect evidence of cheating. Firstly, we can require `Gen` to publish a seed (failure to do so will automatically imply guilt). Further, it is easy to verify that `Gen`'s actions are consistent with the seed and punish it (e.g. via funds slashing) if a violation is detected.

3 Generic Security Statement and Proof

We state the general security theorem for functions implemented as garbled functions and present the proof. The security of our specific construction presented next in Section 4 is an immediate corollary of this general theorem.

Let $\mathcal{G} = (\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a garbling scheme, satisfying correctness and privacy as defined by [9] (as noted in Section 1.5, obliviousness and authenticity are not needed). We additionally require that the decoding information d is projective[5], and decoding each bit calls a hash function, modeled as a *Random Oracle* (RO). Note, standard GC constructions in fact do implement d this way: output wire's plaintext value, for example, can be obtained by computing $\text{low\_bit}[H(w_i)]$, where w_i is the output label[6]. Similarly, other garbling schemes, such as GFSA, can have a decoding function d incorporate a call to RO. We will use the RO programmability [18] in our simulation.

Theorem 1. *Let $\mathcal{G} = (\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ be a garbling scheme as above. Let $(y_0, ..., y_p) = f(x_0, ..., x_q)$ be the function desired to be computed, such that each bit of the function output depends on all inputs[7]. Let `Gen` be the contract generator, $IP_1, ..., IP_n$ be the input providers, $U_1, ..., U_m$ be the unlockers, and $R_1, ..., R_\ell$ be the output receivers. Assume `Gen` is honest and generates $(F, e, d) = \mathcal{G}.\mathsf{Gb}$ and distributes (F, e, d) to players as described above. Let $I \subset \{IP_i, U_j, R_k\}$ be the set of colluding malicious players, such that for no input wire W_i both its input provider and unlocker are in I.*

[5] As defined in [9], in a projective garbling scheme, the encoding information is represented as a list of tokens, one denoting 0, and one denoting 1, for each bit of the input; an encoding of a player's input is a collection of the tokens corresponding to its plaintext input. Similarly, for the output decoding, we say it is projective if the plaintext output is decoded bitwise in a similar manner.

[6] To use $\text{low\_bit}[H(w_i)]$ as the decoding function, `Gen` needs to ensure that $\text{low\_bit}[H(w_i^b)] = b$. This is easy to do by choosing the output labels from corresponding domains. We stress that this is but one way of implementing d with these properties.

[7] While some functions of interest do not meet this requirement, the functions we consider in this work will: indeed, universal circuit and FSA function outputs depend on all their inputs.

Then blockchain evaluation of f which computes $\mathcal{G}.\mathsf{ev}$ as described above, is secure against a malicous adversary corrupting I.

Proof. For lack of space, we present the full proof in Appendix E.

Remark 1. If an unlocker U_j colludes with a reader R_k, together they can learn the output of the computation and abort based on it. This is not a vulnerability in the standard notion of simulation-based security. Note, if we wish to avoid such adaptive abort, we can require that no unlocker colludes with any reader.

4 Instantiations and Security Proofs

Construction 2 (UC GC-based)**.** *Our main construction is the instantiation of the generic GF-based construction described above in Section 2 based on the following choice of underlying primitives/schemes:*

Let f be a function to be computed on the blockchain. Let C be a Universal Circuit computing f. Let $\mathcal{G}$ be the classic Yao GC garbling scheme with point-and-permute and projective decoding function as specified in assumptions of Theorem 1.

Having proven a generic security theorem (Theorem 1) for computing functions represented by arbitrary garbled functions, the proof of security of our main protocol, which is GC-based, is an immediate corollary of Theorem 1.

Theorem 2. *Assume all assumptuions of Theorem 1 hold, including the collusion assumptions. Then Construction 2 is secure in the malicious model against collusions specified in Theorem 1.*

Proof. Proof is an immediate corollary of Theorem 1 and the fact that the underlying GC scheme used in 2 satisfies the required assumptions of Theorem 1. □

Other instantiations and proofs are analogous. In particular, GC-based instantiation is the same as UC GC with the exception of garbling the circuit C, and not necessarily a UC. A garbled FSA offers a reasonable performance for simple functions compared to UC GC. In Appendix C we cast a one-shot evaluation of GFSA as a GF in the [9] notation. A GFSA-based GF satisfying privacy and correctness can be used as a basis of our general construction.

5 Prototype Implementations and Test Results

To illustrate our approach and assess its real-world cost, we implemented a simple prototype and several demonstration applications. Specifically, an implementation of the GFSA approach (see Appendix) was developed, for simplicity, and applied to several demo applications represented as finite state automata.

The prototype Garbler, implemented in Python, takes a simple JSON-format description of an FSA transition function and translates it to a sequence of garbled tables, one for each state update cycle (time step). After garbling, another

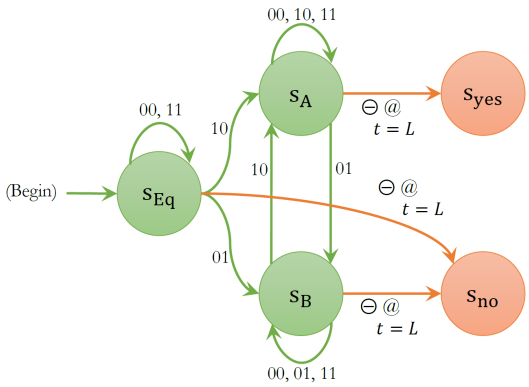

Fig. 2. ***Base FSA used in Millionaire's Problem demo.*** In this version, players AB supply bits of their input values simultaneously, least-significant bit first, on successive cycles. In the final cycle, after L time steps have passed, a Finisher (as in Fig. 1) supplies a special "Finish" symbol $\ominus$ which makes the final result readable by both parties.

Python script translates the garbled machine data to source code in the Solidity programming language for a smart contract for the Ethereum platform; this contract includes the garbled tables as static data, together with a generic *Executor* which accepts garbled input values from input providers and evaluates the garbled machine, producing garbled outputs which can then be interpreted by authorized output recipients.

We used the popular Truffle tool suite, which provides a framework for Ethereum development, to develop, test and deploy (on a private test network, and later on the Ethereum mainnet) several prototypes and demonstrations, which we now discuss.

We implemented a provenance tracking demo (presented in detail in Appendix F for lack of space). Next we discuss our implementation of the millionaires problem.

5.1 Millionaires' Problem Demo

This demo executed a 5-state machine (Fig. 2) implementing MPC for Yao's classic 2-party "Millionaires' Problem" [33] with bit-serial inputs. Here, to achieve MPC fairness (the last player to move possesses an informational advantage due to his ability to look ahead at final outputs), we invoke a special extra player "Finisher" (separate from the 2 normal parties) that acts to reveal the result. (See also Fig. 1.)

Extending this line of argument, we observe that every step (input provision and corresponding GFSA state update) of the GFSA execution may exhibit the following similar vulnerability: the input provider may see the immediate effect of its input, such as such as whether the next FSA state depends on its input.

This issue can be resolved by state-graph transformations, which increase the size of the state machine. In one version that we tested, each additional bit of input length cost almost exactly 2 million gas units to store the additional garbled machine data (since each time step has to be garbled separately), which was about $0.50 worth of Ethereum on the day of that test. With some overhead, the total cost to run an FSA for a 32-bit, two-party Millionaire's Problem was 75 million gas, corresponding to roughly US$75 or so at typical prices. That demo required spreading out the GFSA data over multiple smart contracts, due to Ethereum contract size limits.[8] Reimplementing this same demo using Unlockers for each input and an optimized 2-state FSA allowed us to reduce the cost to ∼$12.

5.2 Configurable Garbled Universal Circuit (GUC) Method

To let us handle applications of a complexity beyond the reach of the GFSA approach in future implementations of GABLE, a simple approach was designed to implement a garbled circuit (GC) for (configurable) universal circuits (UCs). Fig. 3 illustrates our basic UC approach. Input values here are activated using Unlockers (not shown), as we described earlier in the paper.

Although implementation of this method is still in progress, careful analysis of the approach allowed us to already compare its costs to those of the existing GFSA technique for an example problem, a multi-party auction (generalized from the Millionaire's Problem). Fig. 4 shows comparison results. As we expected, cost scales up exponentially with the number of bidders B for GFSA, but only as $\Theta(B \log^2 B)$ for the GUC. (Circuit width scales as $\Theta(B)$, circuit depth scales as $\Theta(\log B)$, and the depth of the Thompson network for each application circuit layer also scales as $\Theta(\log B)$.) The break-even point with our implementation falls at $B = 7$ bidders, where the cost of both approaches is roughly $20 per input bit.

6 Conclusion

In this paper, we described a novel approach to performing secure computation (including functional privacy) on a blockchain. The general approach has two basic embodiments that we discuss, based on the garbling of finite-state automata (FSA) and Boolean circuits, respectively. We gave an overview of the basic structure of the approach, including its participant roles and high-level procedures, outlined a proof of its basic security properties, and discussed early implementations and test results.

We found that simple FSA-based applications can be executed privately at moderate dollar costs on the Ethereum blockchain. For more complex applications, a simple approach based on a construction we call configurable Garbled

[8] Per EIP-170 (`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md`), a contract's deployed bytecode size cannot exceed 24,576 bytes.

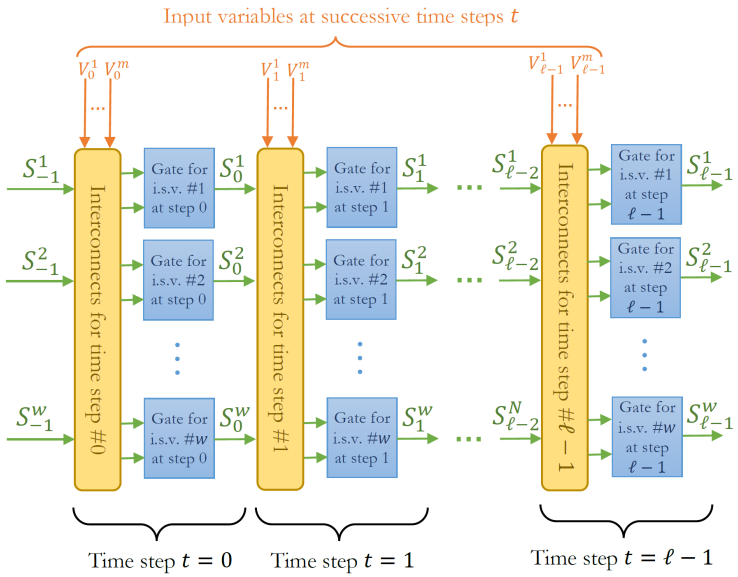

Fig. 3. ***Concept for configurable universal circuits in multi-step computations.*** In this approach, for each layer of application logic, a generalized connection network such as a Thompson network [28] obscures the interconnect topology. The elements of that network, together with generic application gates for computing new values of internal state variables (i.s.v.'s), are configured via truth tables during the garbling process. Thus, no separate programming input is required for this type of UC, yet the function of the network remains obscured.

Universal Circuits (GUC) achieves complete functional privacy with costs that scale as $\Theta(wd \log w)$ in the width w and depth d of the application circuit, with reasonable constant factors. We carried out a detailed cost comparison for a multi-bidder auction application, for which GUC outperforms garbled FSA for $B > 7$ bidders, and remains arguably feasible to perform on-chain with full functional privacy for up to hundreds or even thousands of bidders.

We deployed and executed two GABLE demos on the Ethereum mainnet in late July and mid-September of 2020. The purpose of these tests was to 1) ensure that there were no unforeseen difficulties with real-world deployment, and 2) validate our cost estimation methodology. Both purposes were realized, with no surprises. The first deployment [2,3] (July) was a very simple four-state machine, similar to the supply-chain example of Fig. 6. The second deployment ([1], Sep.) was for a GFSA implementation of a two-party auction as in Fig. 4.

References

1. Auction contract. Ethereum address `0x98ccd7e190ac28a36d4f065a4f14dc5e0b67-f5c7`.

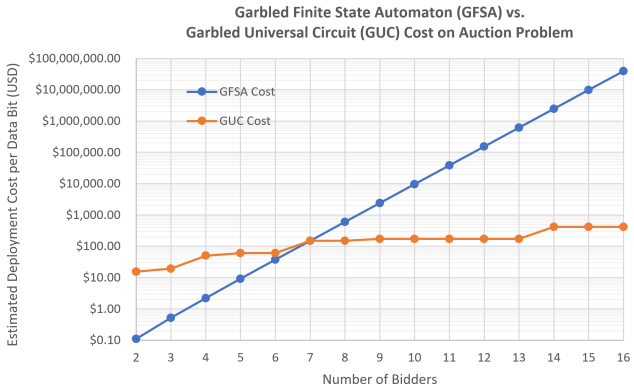

Fig. 4. ***Cost comparison for GFSA vs. configurable universal GCs for multi-party auctions.*** The break-even point occurs for $B = 7$ bidders, where the cost of both techniques is about 800 million gas per bit of input length. Prices here assumed optimistically that we are paying only 1 mETH (or in the ballpark of $0.20) per million gas; however, in recent months, the average gas price has been substantially higher.

2. Simple executor contract. Ethereum address `0xc8a54a72f187ec444ed0896890128-4bbd6d2ec06`.
3. Simple storage contract. Ethereum address `0x57f1c190982d0a9ecdf7c4703e134d-9eaf347de0`.
4. The Secret Network. `https://scrt.network/`. Retrieved June 25, 2020.
5. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.
6. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, Oct. 2016.
7. G. Asharov and C. Orlandi. Calling out cheaters: Covert security with public verifiability. In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 681–698. Springer, Heidelberg, Dec. 2012.
8. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 137–156. Springer, Heidelberg, Feb. 2007.
9. M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, Oct. 2012.
10. F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin. Can a public blockchain keep a secret? In R. Pass and K. Pietrzak, editors, *TCC 2020, Part I*, volume 12550 of *LNCS*, pages 260–290. Springer, Heidelberg, Nov. 2020.
11. F. Benhamouda, S. Halevi, and T. Halevi. Supporting private data on hyperledger fabric with secure multiparty computation. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 357–363, 2018.

12. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, Aug. 2014.
13. V. Buterin. Ethereum Whitepaper. `ethereum.org/en/whitepaper`, 2013.
14. E. Cecchetti, F. Zhang, Y. Ji, A. E. Kosba, A. Juels, and E. Shi. Solidus: Confidential distributed ledger transactions via PVORM. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 701–717. ACM Press, Oct. / Nov. 2017.
15. A. R. Choudhuri, V. Goyal, and A. Jain. Founding secure computation on blockchains. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 351–380. Springer, Heidelberg, May 2019.
16. A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 719–728. ACM Press, Oct. / Nov. 2017.
17. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147. Springer, Heidelberg, Aug. 1993.
18. M. Fischlin, A. Lehmann, T. Ristenpart, T. Shrimpton, M. Stam, and S. Tessaro. Random oracles with(out) programmability. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 303–320. Springer, Heidelberg, Dec. 2010.
19. M. P. Frank, C. N. Cordi, K. G. Gabert, C. B. Helinski, R. C. Kao, V. Kolesnikov, A. K. Ladha, and N. D. Pattengale. The GABLE report: Garbled autonomous bots leveraging Ethereum. Technical report SAND2020-5413, Sandia National Laboratories, 2020. `https://www.osti.gov/biblio/1763537`.
20. C. Gentry, S. Halevi, H. Krawczyk, B. Magri, J. B. Nielsen, T. Rabin, and S. Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 64–93, Virtual Event, Aug. 2021. Springer, Heidelberg.
21. O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
22. S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, Jan. 1991.
23. V. Kolesnikov and A. J. Malozemoff. Public verifiability in the covert model (almost) for free. In T. Iwata and J. H. Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 210–235. Springer, Heidelberg, Nov. / Dec. 2015.
24. A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. 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.
25. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In M. Blaze, editor, *USENIX Security 2004*, pages 287–302. USENIX Association, Aug. 2004.
26. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008. Retrieved June 25, 2020.
27. N. Szabo. Secure Property Titles with Owner Authority. `nakamotoinstitute.org/secure-property-titles/`, 1998. Retrieved 06/25/2020.
28. C. Thompson. Generalized connection networks for parallel processor intercommunication. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1977.
29. A. van Wirdum. "Confidential assets" brings privacy to all blockchain assets: Blockstream. Bitcoin Magazine, April 2017, 2017. Retrieved June 25, 2020.

30. Wikipedia. Transactive Energy. `en.wikipedia.org/wiki/Transactive_energy`.
31. Y. Yang, L. Wei, J. Wu, and C. Long. Block-smpc: A blockchain-based secure multi-party computation for privacy-protected data sharing. In *Proceedings of the 2020 The 2nd International Conference on Blockchain Technology*, ICBCT'20, page 46–51, New York, NY, USA, 2020. Association for Computing Machinery.
32. A. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.
33. A. C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, pages 160–164, Chicago, IL, USA, 1982. doi:10.1109/SFCS.1982.38.
34. S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, Apr. 2015.
35. G. Zyskind, O. Nathan, and A. Pentland. Decentralizing privacy: Using blockchain to protect personal data. In *IEEE Symposium on Security and Privacy Workshops,*, pages 180–184, 2015.

Appendix

Here we describe in additional detail the methods, algorithms and encodings used in our GFSA-based prototype and demo implementations. We cast GFSA as a BHR GF, and sketch the intuition behind proofs of correctness, privacy and obliviousness. We also give additional details of our prototype implementations and early test results.

A FSA Formalism

First, we briefly introduce some formalism for discussing finite state machines, or finite state automata (abbreviated FSA in this document). For our purposes, a *finite state automaton* $\mathcal{A}$ can be defined as a tuple

$$\mathcal{A} = (\mathbf{\Sigma}, \mathbf{S}, \mathfrak{T}) \tag{1}$$

where $\mathbf{\Sigma}$ is an *alphabet* or set of possible input *symbols*, $\mathbf{S}$ is a set of (monolithic) *states*, and $\mathfrak{T}$ is a *transition function*, defined (for Moore-type state machines) as a function

$$\mathfrak{T} : \mathbf{S} \times \mathbf{\Sigma} \to \mathbf{S}, \tag{2}$$

that is, mapping a pair (s_o, σ) of an *origin state* $s_\mathrm{o} \in \mathbf{S}$ and input symbol σ to a *destination state* $s_\mathrm{d} \in \mathbf{S}$. Note that in this definition, we are not explicitly describing outputs; they can be inferred from the states in a separate construction.

To facilitate multi-party computation, we can write the input symbol alphabet as a Cartesian product of separate symbol alphabets $\mathbf{\Sigma}_i$ for each input provider i; thus, an input symbol σ is actually a tuple $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ of symbols provided by different input providers. In such a case we write $\sigma = \vec{v}$ and call it an *input vector*.

B State Machine Garbling

A simple method for garbling an FSA, as defined above, is as follows. For simplicity, we present the method without unlockers. (Unlockers are easily added in a manner analogous to GC: `Gen` encrypts and sends FSA garbled labels to input providers, and unlockers are given the corresponding decryption keys.)

We will garble each time step (state-update cycle) separately. We assume that a prespecified maximum number ℓ of state-update cycles will be supported. Individual time steps or state-update cycles are indexed $t \in \{1, \ldots, \ell\}$. Variable $S_t \in \mathbf{S}$ identifies the state resulting from time step t (with S_0 being the initial state), and variable $\vec{V}(t) \in \mathbf{\Sigma}$ gives the vector of input symbols supplied to step t.

Garbling each time step works as follows. For each time step t, the (composite) input variable $\vec{V}_t$ and origin state variable S_{t-1} can each be identified with a corresponding multi-valued signal line L_i, with an associated set of possible line

values l_i^j, corresponding to possible input vectors $\vec{v} \in \mathbf{\Sigma}$ or states $s \in \mathbf{S}$ respectively. These lines can be considered akin to the bit wires of a GC construction, but are multi-valued. Line values l_i^j can be encoded using corresponding encrypted labels e_i^j. In our implementation, in the case of input variables V_t^i for individual providers, and state variables S_t, we assign their labels using (cryptographically securely generated) 256-bit random strings; labels for composite input variables $\vec{V}_t$ are computed from those for their component variables V_t^i. For a composite input vector $\vec{v}$ and state s, we write $e(\vec{v})$ and $e(s)$ for the corresponding encoded labels e_i^j. The label $e(\vec{v})$ is computed by combining the labels $e(\sigma_i)$ supplied by individual input providers as follows,

$$e(\vec{v}) = \sum_{i=1}^{n} e(\sigma_i), \tag{3}$$

where recall σ_i is the plain-text input value (symbol) whose encoding is being provided by an individual input provider. The sum operation is implicitly modulo 2^{256}, or may alternatively be replaced by bitwise exclusive-or ($\oplus$).

The garbled representation of the state-transition function for time step t, $\mathfrak{G} = \mathfrak{G}_t$, is then constructed as follows. It suffices to store it in a hash table or mapping structure $\mathfrak{G}[\cdot]$, supported by most programming languages. For each pair $(\vec{v}, s)$ of a possible input vector and origin state, we construct an *arc identifier* I by combining their labels, for example, using $I = e(\vec{v}) \oplus e(s)$. Now, assume we have available an indexed family $h_i(\cdot)$ of hash functions, which may be derived from a base hash function $h(\cdot)$ by, for example, $h_i(x) = h(x + i)$. Now, we can encode the particular state transition $(s, \vec{v}) \to \mathfrak{T}(s, \vec{v})$ by extending the mapping structure $\mathfrak{G}[\cdot]$ by assigning:

$$\mathfrak{G}[h_0(I)] := h_1(I) \oplus e(\mathfrak{T}(s, \vec{v})). \tag{4}$$

(Here, square brackets denote accessing an entry of the map structure $\mathfrak{G}$.) The right-hand side is simply encrypting the next-state label $e(\mathfrak{T}(s, \vec{v}))$ using an encryption function that consists of simply XOR-masking it with $h_1(I)$. Meanwhile, $h_0(I)$ is just being used here as a key that indexes into the mapping, which will allow the entry to be rapidly retrieved later. (This method for hash table keying can be considered a generalization of the point-and-permute method that allows fast lookup of garbled table values.) This therefore comprises a fairly prosaic implementation of the well-known garbled table abstraction.

It is clear that, at execution time, given the tables $\mathfrak{G}_1, \ldots, \mathfrak{G}_\ell$, the circuit evaluator can, as input labels are received for successive time steps, simply perform the necessary hash table lookups, decrypt the (outer layer of encryption on the doubly-) encrypted next-state labels, and proceed to correctly traverse the entire garbled state sequence (assuming there are no collisions between our 256-bit hash table keys).

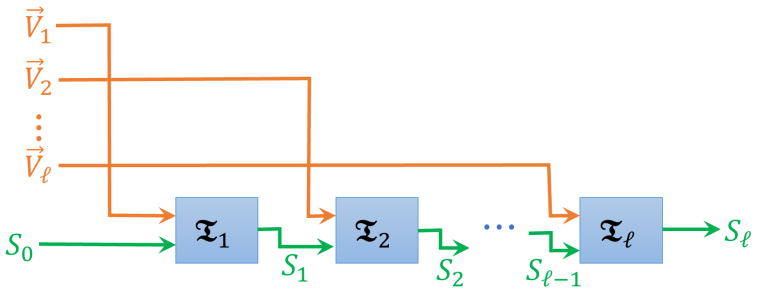

Fig. 5. ***An ℓ-step FSA computation as a circuit with multivalued lines.*** For each time step $t \in \{1, 2, \ldots \ell\}$ we have an input line $\vec{V}_t$ which will deliver the vector $\vec{v} = (\sigma_1, \ldots, \sigma_n)$ of symbols from input providers for time step t, and a state line S_t which holds the new state after update step t. S_0 is the initial state, which can be considered to be another input or held constant. Assuming the input alphabets $\mathbf{\Sigma}_i$ are all binary, $\mathbf{\Sigma}_i = \{0, 1\}$, we can think of each input line $\vec{V}_t$ as a single input line having 2^n possible values. Meanwhile, S_t can be viewed as an input line with $|\mathbf{S}|$ possible values. $\mathfrak{T}_t$ is simply the FSA transition function $\mathfrak{T}$, which is normally the same for each t but can also be allowed to vary. Each $\mathfrak{T}_t$ block can be thought of as a single large two-input gate with multi-valued input and output lines. The entire FSA computation can then be viewed as a single circuit, simply one whose 'wires' have many possible values, and (apart from its multi-valued-ness) is garbled in the same manner as a Boolean circuit.

C Casting GFSA as a BHR Garbled Function (GF)

This will follow the same general structure as for GCs. We must specify how the elements of the GFSA construction map to the components of the BHR model (see Fig. 1 of [9]), and prove that they have the required security properties.

First, ignoring reactive functionalities for now, we describe how to view an entire FSA-based computation (over any fixed number of steps ℓ) as a single circuit. This helps to clarify why GC-based security proofs apply to it.

The correspondence between BHR GF and the GFSA method goes as follows. We focus here on consideration of the entire state-machine execution over all ℓ update steps as a single garbled function. (As mentioned in the main text, extension of the proof construction to the reactive case is not covered in this paper.)

The following two lists refer specifically to the block diagram for a generic garbling scheme shown in Fig. 1 of [9], which should be inspected by the reader.

Correspondence for blocks of a generic garbling scheme:

- ev evaluates the circuit in Fig. 5 given a list $s_0, \vec{v}_1, \ldots, \vec{v}_\ell$ of input values for the input lines $(S_0, \vec{V}_1, \ldots, \vec{V}_\ell)$, respectively, and lookup tables for the functions $(\mathfrak{T}_1, \ldots, \mathfrak{T}_\ell)$. It returns a plaintext identifier for the final state s_ℓ.
- Gb: is the procedure for garbling FSAs, which produces (F, e, d), where F is a collection of garbled transition tables $\mathfrak{G}_t[\cdot]$ for each time step t.

– En generates the encoding of the entire FSA input x, given x and the encoding information e. En is evaluated by unlockers decrypting encrypted labels published by input providers.
– Ev evlauates GF F by evaluating the garbled gates $\mathfrak{G}_t$ in sequence using the process described above where we combine the labels on the two input lines $\vec{V}_t, S_{t-1}$ and use that as the key to retrieve and decode the garbled table row corresponding to the arc identifier I to retrieve each garbled state $e(s_t)$; the final one of these, $e(s_\ell)$, is then the garbled output.
– De decodes the output label. In our system, De applies a hash function H.

Correspondence for lines of a generic garbling scheme:

– Function f to be garbled: This is a representation of the transition function $\mathfrak{T}$ of the application FSA as an explicit state transition graph (list of arcs).
– Garbled function F: This is our list of garbled transition tables for all the time steps, $(\mathfrak{G}_1, \ldots, \mathfrak{G}_\ell)$.
– Encoding information e: This includes the garbled labels $e(\vec{v}_t)$ for all ℓ input lines, as well as all of the needed unlocker key(s).
– Plaintext input x: A list of plaintext input vectors $(\vec{v}_1, \ldots, \vec{v}_t)$, specifying the input values that all the of input providers want to provide for all of the time steps. Also, the initial state s_0 can optionally be part of the input (or it can be assigned to a predetermined constant).
– Encoded input X: This is a list $(e(\vec{v}_1), \ldots, e(\vec{v}_\ell))$ of encoded input identifiers obtained by unlocking and combining the input labels provided by individual input providers.
– Encoded output Y: This is just the result of evaluating the garbled circuit; i.e., it is the final state label $e(s_\ell)$.
– Output decryption information d: This could just be a table mapping hashes of garbled state labels to plaintext identifiers.
– Plaintext output y: This is a plaintext interpretation of the final state, readable by an output recipient.

D GFSA correctness and security properties

Correctness of the GFSA construction is immediate due to the straightforward encoding of the FSA transition function $\mathfrak{T}$ as a corresponding garbled table $\mathfrak{G}[\cdot]$.

Privacy, obliviousness and authenticity can be shown analogously to that of standard Yao GC, as above we presented our GFSA construction as a (slight generalization of) GC. We mention the following relevant observations.

1. Information needed to decode line values in the circuit picture of Fig. 5 is not available to any party (except the Garbler,[9] if this is allowed and the information is retained), except that the final (output) line may be decoded by the designated output recipient(s);

[9] Generally speaking, we assume that the Garbler is not an adversary.

2. Alternate input values cannot be probed by input providers, since the Unlocker(s) will only accept/unlock a single value for each input.
3. The encoding of the garbled table $\mathfrak{G}[\cdot]$ ensures that the only path through the state graph that can be decrypted is the one that gets decrypted (while still garbled) by the sequence of input vectors actually provided; thus, no information about the state graph topology can be inferred (other than that it includes at least one path of length ℓ).

E Proof of Security of Theorem 1

Proof. (Sketch.)

We prove malicious security by simulation using standard ideal-real simulation definition (see e.g., [21]). Let I be the set of corrupted players. We build a simulator Sim_I by interacting with players in I. Sim_I starts the game (i.e. starts all interactive Turing machines implementing players in I). Then Sim_I runs $\mathcal{G}.\mathsf{Gb}$ and distributes F, e, d to players in I appropriately. This includes adding F and d to the views of players in I, since F, d are published on the blockchain (the proof for the case when d is distributed only to the readers is analogous). Then Sim_I listens to the messages/events from players who are members of I, and responds to them. Sim_I also emulates the actions of the honest players, such as submitting encrypted input when a plaintext input becomes available, or unlocking another player's input (whether honest or malicious) as needed. Consider the following possibilities for player $P \in I$ that Sim_I needs to respond to:

1. P is input provider. The only message expected from P is the encrypted label. Sim_I records the encrypted label provided by P. Because, by assumption, the unlocker for that wire is not corrupted, Sim_I simulates the corresponding action of the unlocker by publishing the decrypted label (and notifying I appropriately). However, if the decryption fails, Sim_I follows the protocol specification (e.g. simply ignores the invalid input encryption or publishes it anyway.)
2. P is unlocker. Then P can only provide the unlocked wire label. Sim_I records the unlocked label.
3. P is output receiver. No messages are expected from P, and they are ignored by Sim_I.

Further, Sim_I emulates the actions of honest players as follows:

1. Input submission by honest player. Sim_I publishes arbitrary encrypted label for the corresponding input (e.g. the zero label for the case of GC). Note, even though `Adv` may have the unlocking key for this, it won't be able to distinguish any two labels that may be submitted, by the privacy property of $\mathcal{G}$.
2. Unlocking by the honest player. Sim_I knows all the decryption keys as it generated them, and can perfectly emulate this action.

As the recorded inputs are submitted and published, no information is revealed to Adv (or to any player) *until the last input is submitted*. This is because even though Adv may have access to (unencrypted) labels corresponding to the inputs, by theorem assumption, every output bit depends on all function input bits, and hence cannot be computed. Further, because of the privacy guarantee of the underlying garbling scheme, Adv learns nothing from its view at this point.

Consider the submission of the last input and subsequent unlocks by all honest players. At this point, Adv is able to compute the output of the function, if it submitted valid inputs. Adv at this point may choose to:

1. abort (or, equivalently, submit at least one invalid unlock). In this case, Sim_I will simulate abort, which may include delivering the output of the computation to I. We discuss the output simulation below.
2. submit valid unlocks. In this case, the function output is obtained. We discuss the output simulation next.

Finalizing the simulation. In case Adv obtains the output by opening all input labels, we simulate this by querying the trusted party and giving it all the inputs of I. Because all communication is performed via the blockchain, Sim_I recorded all messages of players in I. Given that Sim_I generated the GF itself, it trivially extracts inputs of malicious players (including provisioning of invalid inputs or invalid decryptions), and submits the input to the Ideal Trusted Party. In response, Sim_I receives the true output. It then *programs* the RO such that the output decoding information d implements the correct output. Sim_I outputs the view it had so far provided to I, as well as whatever players in I output. This completes the simulation.

It is easy to see that this view is indistinguishable from the real view due to the privacy and correctness properties of the underlying garbling scheme and the programmability of RO.

□

F Prototype Implementations and Test Results: Provenance Tracking

F.1 Initial Prototype

A simple, arbitrary 4-state FSA (not shown) was used for an initial test of our method on a development chain.

After testing the prototype, we proceeded to implement more meaningful demonstration applications and test them on a private testnet to verify their Ethereum "gas" (resource) costs. Our early GFSA demos did not yet include unlockers, and utilized more expensive methods (not discussed in this paper) to attain functional privacy; thus, the measured costs reported below overestimate what the true costs of our present method would be in some cases. Selected demos were also tested on the Ethereum mainnet, and worked as expected.

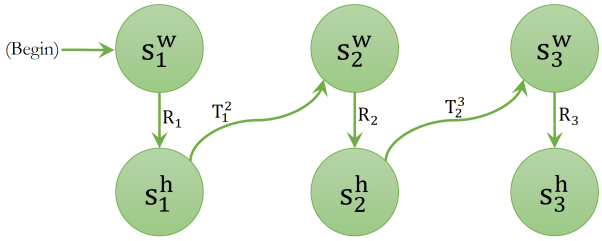

Fig. 6. ***Simple 6-state FSA used in supply-chain demo*** The states s_i^w, s_i^h here represent "waiting" and "holding" conditions for vendors $i = 1, 2, 3$ as an item is passed along a supply chain. Vendor i can input a respective symbol R_i to denote that they have received the item, and T_i^j to denote that they have shipped the item to vendor j. Output state visibilities here are arranged in such a way that each vendor only has visibility on its respective states.

F.2 Provenance Tracking Demo

Our first "real" application demonstration to run on an active, multi-node (albeit sandboxed) Ethereum test network was a simple 6-state FSA for a simple *supply-chain provenance tracking* application (Fig. 6). Although still a "toy" demonstration, this one is at least suggestive of potential real-world applications; also, it exercised our capability to have limited visibility of outputs.

In the demo, three different hosts each run a "vendor client" and a local Ethereum node. A fourth host runs the Garbler, deploys the garbled machine contract to the blockchain, and distributes selected input encryption/output decryption data e, d to respective vendor nodes. Each node's vendor client watches for its 'w' state (denoting that it should expect to receive the item), then simulates receiving the item, then transmits its 'R' symbol to notify the machine of this, then simulates processing the item and shipping it to the next vendor, then transmits its 'T' symbol to notify the machine of this, at which point the next vendor in the supply chain takes over. Fig. 7 shows a brief excerpt of demo output.

Fig. 8 shows a breakdown of costs, calculated based on "gas," which is the unit of computational resource usage in Ethereum. The real-world price of Ethereum gas varies substantially, but, during the period of our experiments (Mar.-Sep. 2020), frequently fell within a range of 10–50 mETH (milli-ether) per million gas units.[10] The price of ether (the native cryptocurrency of the Ethereum blockchain) also varies substantially, but fell in the range of US$0.10–$0.40 per mETH during that period.

We note that, of the total resource cost, the lion's share (73%) was the storage cost for the GFSA data. This test taught us that storage costs dominate the total cost of executing garbled computations on the Ethereum blockchain using our approach; thus, we focused primarily on storage costs in our later experiments.

[10] A historical chart of Ethereum gas prices can be found at `https://etherscan.io/chart/gasprice`.

```
CLIENT FOR VENDOR ID: 1
The current time step number is 0.
The current state is: 2f7214f79f2ceb951b7b6977b40cd11ba42dc00111ddd1bf62026ff82ac99e09
    I have an output key that decodes this state as meaning I am 'waiting'.
    This means, someone sent me the item and I should expect to receive it!
    Twiddling thumbs...
    I just received the item!  Now I'll report this to the machine...
    Providing symbol 'R' garbled as: 5b9c8b444992a0cb1815adc795d9de69f0cbca2e25094d199edb0de8eaf02210
Waiting for state to change...
    Detected a change in the machine state!
The current time step number is 1.
The current state is: c01a61b2a93afac09cc6fc8192686c4e9192f67d4eb19067666ff5492f0864cb
    I have an output key that decodes this state as meaning I am 'holding'.
    This means, I am currently holding the item for processing.
    Working hard on adding value to the item...
    Finished!  Shipped item to next vendor.  Now telling the machine that.
    Providing symbol 'T' garbled as: 4e61625743dd2f04f491d903cc28b63563ec31d130d738bf2f045665d5343739
Waiting for state to change...
    Detected a change in the machine state!
The current time step number is 2.
The current state is: 4669c9dd752b02830d971a3a000294aae11f05ddd9ca32e6d89d190bf85423d6
    I don't know the meaning of this state...  I'm ignoring it.
Waiting for state to change...
⋮
```

Fig. 7. ***Excerpt from one vendor client's diagnostic output in the supply chain demo.*** The actual garbled encodings (256-bit labels) for machine states and input symbols are shown. The narrative flavor of most of the text here is merely intended to be suggestive of a real application.

Transaction Category	No. of Txns.	Resource Usage ('gas' units)	% of total	Cost at Market Prices as of March 13, 2020	
				Cost in Ether @ 9 nETH/gas	Cost in USD@ $128.09/ETH
Deploy main contract	1	1,756,030	72.6%	15.8043 mETH	US$ 2.02
Provide garbled input	5	309,282	12.8%	2.7835 mETH	US$ 0.36
Truffle framework (overhead)	3	353,976	14.6%	3.1858 mETH	US$ 0.41
TOTALS:	**9**	**2,419,288**	**100%**	**21.7736 mETH**	**US$ 2.79**

Fig. 8. ***Breakdown of costs for the supply chain demo.*** Although real-world prices of Ethereum gas units vary substantially, one million gas (Mgas) typically corresponded to at least about US$1 worth of Ether (the base cryptocurrency of the Ethereum network) at the time of our experiments. However, at the time of this writing (Feb. 19, 2021) the cost per Mgas is far higher (nearly US$400).