The GABLE Report: Garbled Autonomous Bots Leveraging Ethereum

Michael P. Frank
Christopher N. Cordi
Kasimir G. Gabert
Carollan B. Helinski
Ryan C. Kao
Vladimir Kolesnikov
Abraham K. Ladha
Nicholas D. Pattengale
ABSTRACT
Simple but mission-critical internet-based applications that require extremely high reliability and availability could potentially benefit from running on robust public programmable blockchain platforms such as Ethereum. Unfortunately, program code running on such blockchains is ordinarily publicly viewable, rendering these platforms unsuitable for applications requiring strict privacy of application code, data, and results. However, might it be possible to encode an application’s business logic and data for these platforms in such a way that it becomes impossible for unauthorized parties to infer any meaningful information whatsoever about the semantics of the data, and the operations being performed on that data? In this report, we describe GABLE (Garbled Autonomous Bots Leveraging Ethereum), a system concept developed at Sandia that achieves this security goal in a limited, but still useful range of circumstances. GABLE uses simple but effective algorithms to permit secure private execution of garbled state machines (and more efficient garbled circuits) on public computing resources. We give an example working implementation for garbled state machines, written using the Python and Solidity programming languages, and outline how our methods can be extended to support a more powerful garbled universal circuit model of computation. The capability embodied by the GABLE system has significant potential applications, a few of which we discuss in this report.
ACKNOWLEDGEMENTS

Thanks are owed to Tan Thai of Sandia, for early discussions about a problem statement which inspired the development of the design concepts described in this document.

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for NNSA under contract DE-NA0003525.

NOTE: 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.
CONTENTS

1. Document Description ......................................................................................................................................... 25
   1.1. Organization of this Document .................................................................................................................. 25

2. Motivations and Broad Use-Case Scenarios ..................................................................................................... 26

3. Requirements for the GABLE System ............................................................................................................. 28
   3.1. Assumptions & Limitations ......................................................................................................................... 30

4. Example Applications ........................................................................................................................................... 31
   4.1. Toy Problems .............................................................................................................................................. 31
      4.1.1. Millionaires’ Problem ................................................................................................................................. 31
      4.1.2. “Dungeon Race” game .............................................................................................................................. 32
   4.2. Serious Applications .................................................................................................................................... 33
      4.2.1. Supply Chain Provenance Tracking ......................................................................................................... 33
      4.2.2. Sealed-Bid Auctions ................................................................................................................................. 33
      4.2.3. Transactive Energy ....................................................................................................................................... 34
      4.2.4. Data Peering ............................................................................................................................................... 34

5. Technical Outline of the GABLE System Design ........................................................................................... 35
   5.1. State-Machine Formalism ............................................................................................................................ 35
   5.2. Garbled State Machine Encoding .............................................................................................................. 38
   5.3. Garbled State Machine Execution ............................................................................................................. 43
   5.4. Risks and Vulnerabilities ............................................................................................................................. 47
      5.4.1. Lookahead (a.k.a. Fairness) Problem ....................................................................................................... 47
      5.4.2. Reconvergent Arcs Problem .................................................................................................................. 49
      5.4.3. Reconvergent Paths Problem .............................................................................................................. 49
      5.4.4. Collusion Problem .................................................................................................................................... 50

6. Technical Details of the Prototype Implementation .......................................................................................... 51
   6.1. Prototype Implementation of a Garbler in Python ..................................................................................... 51
   6.2. Prototype Implementation of an Executor in Solidity ............................................................................. 52
   6.3. Testing the Prototype Implementation ..................................................................................................... 53

7. Demo Applications ............................................................................................................................................... 55
   7.1. Supply-Chain Provenance Tracking Demo ................................................................................................. 55
   7.2. Millionaires’ Problem Demo ..................................................................................................................... 58
   7.3. Remarks on Red-Teaming ......................................................................................................................... 60

8. State-Machine Compiler Concept ..................................................................................................................... 64
   8.1. Compilation Stages ....................................................................................................................................... 64
   8.2. Software Architecture ................................................................................................................................. 65
      8.2.1. Access Package ....................................................................................................................................... 65
      8.2.2. Crypto Package ....................................................................................................................................... 66
      8.2.3. Functionality Package ............................................................................................................................ 66
      8.2.4. Garbler Package ..................................................................................................................................... 67
      8.2.5. I/O Package .............................................................................................................................................. 67
      8.2.6. Machine Package ................................................................................................................................... 68
      8.2.7. Participants Package ............................................................................................................................. 69
      8.2.8. Stagers Package ...................................................................................................................................... 69
      8.2.9. Stages Package ....................................................................................................................................... 70
      8.2.10. Testing Package ................................................................................................................................... 70
Figure 8-2. Compiler packages.....................................................................................................................66
Figure 9-1. A circuit computation operating for up to $\ell$ layers of logic / time steps. .........................71
Figure 9-2. Generalized connector network, and a routing element in this interconnect fabric ....72
Figure 9-3. Algorithm for garbling selector gates in the first layer of the interconnect fabric........74
Figure 9-4. Simplified algorithm for garbling selector gates in first layer of interconnect fabric. .....76
Figure 9-5. Profile of circuit width versus logic layer for the SHA-256 circuit .....................................78
Figure 9-6. Profile of circuit width versus logic layer for the AES example circuit...........................78
Figure 9-7. Pseudocode for bitwise multi-bidder auction algorithm.......................................................80
Figure 9-8. Application circuit example for multi-bidder auction algorithm for $B = 3$ bidders.......82
Figure 9-9. Semi-logarithmic chart comparing costs of FSM vs. GUC approaches for auctions. .....83

LIST OF TABLES
Table 7-1. Gas Cost for Vendor Demo Transactions (in order of occurrence)...........................................57
Table 9-1. Deployment Cost Estimates for Circuit Examples........................................................................77
Table 9-2. Deployment Cost Estimates for Circuit Examples without Functional Privacy....................79
Table 9-3. Complexity and Cost of Monolithic State Machines for Simple Auctions...........................81
Table 9-4. Complexity and Cost of Garbled Universal Circuits for Small Auctions...............................82
Table 9-5. Complexity and Cost of Garbled Universal Circuits for Large Auctions...............................83
Table A-1. Table of Notations (in order of appearance)............................................................................96
Table A-2. Table of Notations (in alphabetical order) .............................................................................101
EXECUTIVE SUMMARY

Contemporary programmable blockchain platforms such as Ethereum are touted as providing a sort of “world computer,” in the sense of, an always-on public computing resource that uses the distributed consensus protocol underlying its blockchain technology to provide guaranteed reliability, availability and auditability for computations implemented as smart contracts, which are (for a fee) posted to the blockchain and subsequently executed. Even in the event of a widespread disaster, if one can access a connected component of the Ethereum network, this computing resource will remain online. Some organizations may wish to be able to take advantage of such a robustly available computing facility to execute particularly mission-critical computational tasks, but only if they can do so without revealing openly (including to potential adversaries or competitors) the nature of the computation performed.

This, then, raises an interesting question: Namely, is it possible to run a computation on an existing public blockchain such as Ethereum while publicly revealing no information at all about the data being operated on, the computation being performed on that data, or the computed results? If so, then exactly how, and under what conditions, can this be accomplished?

Although the above may, at first, sound like an impossible goal, we show in this report that it can in fact be achieved, and in a reasonably practical way, subject to certain assumptions and limitations; and we describe an example system called GABLE (Garbled Autonomous Bots Leveraging Ethereum) developed at Sandia that illustrates this capability. The complete source code for a reference implementation of a simple working prototype of GABLE is given in Appendices B & C, and results from tests of more complete demonstration applications are shown in §7.

The design of GABLE works by applying, in (what we find) an interesting way, some basic cryptographic techniques that have already been well known in the literature for some time. The key technique that we leverage is something we call garbled lookup tables, which provide a generic means by which encrypted inputs can be used to look up encrypted results, while not revealing the meaning of either the input, the result, or the function encoded by the lookup table. By chaining such garbled table lookups together in specific ways, we can easily implement garbled state machines (§5) and garbled universal circuits (§9), either of which can encode a desired computation in a way that meets the desired goal of completely inscrutable operation, under certain assumptions. Both methods incur certain complexity overheads, but for general (worst-case) computations, the overhead of the latter approach is much smaller—namely, a logarithmic factor, rather than exponential for the former. Our techniques also touch upon aspects of several popular secure computing paradigms such as secure multiparty computation, fully homomorphic encryption and indistinguishability obfuscation, as discussed in §10.

Results of experiments on our existing demos indicate that very simple applications can be performed in a totally garbled fashion on the public Ethereum network at a modest cost, that is, with fees equivalent to a few U.S. dollars. Somewhat more complex applications would of course cost more to deploy and execute in this way, but some are still feasible (§9.3). Extremely complex applications may not yet be feasible on today’s Ethereum network using the techniques we describe at present, but may become increasingly feasible over time given further development of more efficient programmable blockchain platforms and further refinement of our techniques.

Overall, our conclusion in this report is that secure, completely inscrutable execution of usefully complex application logic on programmable blockchain platforms such as Ethereum is a capability that is in fact already straightforward to achieve today, and the techniques available for doing this will only become more efficient and more capable in the future.
One important implication of this result is that, if and when these kinds of methods become widely utilized, it will then become highly unwise in general to assume that one can necessarily infer anything meaningful at all about what a given smart contract is actually doing merely by inspection of its code or its execution trace. Whether this development might eventually impact the design of, or governance policies for programmable blockchain platforms themselves remains an interesting open question.
# Glossary of Acronyms and Definitions

Please note that each instance of a glossary term is lightly underlined throughout this document.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>51% attack</td>
<td>In a blockchain-based system, a 51% attack occurs when more than half of the voting authority (e.g. hashing power) conspires to roll back transactions.</td>
</tr>
<tr>
<td>abstract (finite) state machine</td>
<td>In the GABLE compiler (in progress), an abstract state machine means a finite state machine prior to concretization transformations to prepare it for garbling.</td>
</tr>
<tr>
<td>(access) authorization</td>
<td>A key to access a specific capability relating to a given garbled Machine. Such authorizations may be delegated by the Garbler to other protocol participants.</td>
</tr>
<tr>
<td>access information</td>
<td>A body of information from the Garbler that is distributed to authorized Providers and Spectators that enables them to provide inputs to and/or interpret outputs from the given Machine. Essentially, this comprises a set of keys.</td>
</tr>
<tr>
<td>activation key</td>
<td>A special key $a_i$ that is needed to unlock the first-layer routing element for a given time step and combination of input values to that routing element; see §9.</td>
</tr>
<tr>
<td>activation phase</td>
<td>A phase during the execution cycle for the circuit model, after the input-gathering phase and before the evaluation phase, in which an Unlocker provides activation keys or activated/decrypted input keys, which then allow evaluation of the garbled universal circuit to proceed.</td>
</tr>
<tr>
<td>Accessor</td>
<td>In general, an Accessor is any entity that may possess Reader and/or Writer capabilities with respect to a given Machine.</td>
</tr>
<tr>
<td>AES</td>
<td>The Advanced Encryption Standard is a widely-used symmetric cryptosystem.</td>
</tr>
<tr>
<td>AND</td>
<td>A Boolean logical gate whose output is True if and only if all its inputs are True.</td>
</tr>
<tr>
<td>application cycle</td>
<td>A period of Machine execution during which the machine first gathers some input values from its external environment, and then updates its internal state in a process that may, in general, involve a sequence of multiple computation steps. In most of this document, we assume that there is only one computation step per application cycle and prefer to use the phrase time step in such cases. However, in parts of §9, we consider models that allow multiple computation steps to take place per application cycle; for example, the circuit in Figure 9-8 carries out one application cycle over a series of several computation steps.</td>
</tr>
<tr>
<td>application time step</td>
<td>See application cycle.</td>
</tr>
<tr>
<td>arc</td>
<td>A connection between two nodes in a graph. May be directed or undirected. The arcs found in GABLE’s state transition graphs are directed, and are labeled with transition conditions, and they represent conditional state transitions.</td>
</tr>
<tr>
<td>arc identifier</td>
<td>In GABLE, an arc identifier is a hash code that identifies a particular arc in the machine’s state transition graph (for a particular time step). Knowing an arc’s identifier enables one to locate and decrypt its arc data.</td>
</tr>
<tr>
<td>arrow</td>
<td>Often used as a graphical representation of an arc in a graph. Undirected arcs may be represented by double-headed arrows.</td>
</tr>
<tr>
<td>ASCII</td>
<td>The American Standard Code for Information Interchange is a standard encoding for English text characters encoded in 7 or 8 bits.</td>
</tr>
<tr>
<td>asymmetric cryptosystem</td>
<td>See public-key cryptosystem.</td>
</tr>
<tr>
<td>authority</td>
<td>A specific set of access authorizations associated with a given Machine.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>big-endian</td>
<td>A bit-ordering convention in which the most significant bit in a binary (base-2) number is ordered first.</td>
</tr>
<tr>
<td>bit</td>
<td>Historically a portmanteau of “binary digit,” this refers to a variable with two possible values or states, or to a specific value of such a variable.</td>
</tr>
<tr>
<td>bit string/vector</td>
<td>A specific type of protocol and data structure that provides a distributed ledger capability. Blockchain organizes transactions into a sequence of blocks.</td>
</tr>
<tr>
<td>blockchain</td>
<td>A specific type of protocol and data structure that provides a distributed ledger capability. Blockchain organizes transactions into a sequence of blocks.</td>
</tr>
<tr>
<td>Bloom filter</td>
<td>A Bloom filter is a data structure that supports fast probabilistic testing for set membership. The method supports a nonzero but negligible probability of false positive results but verifies non-membership with certainty.</td>
</tr>
<tr>
<td>Boolean (logical) gate</td>
<td>A special type of computational gate in which the inputs and outputs are all bits (two-state variables) which can be interpreted as truth values. George Boole famously studied examples such as AND, OR, NOT. See also logic gate.</td>
</tr>
<tr>
<td>Boolean variable</td>
<td>A variable that takes on a truth value. Named after George Boole.</td>
</tr>
<tr>
<td>bot</td>
<td>Derived from “robot;” an application that runs autonomously on a network.</td>
</tr>
<tr>
<td>bytecode</td>
<td>In general, a bytecode could refer to any encoding of data in terms of 8-bit octets or bytes; in this document, the word may also sometimes refer more specifically to compiled machine code for the Ethereum virtual machine.</td>
</tr>
<tr>
<td>cardinality</td>
<td>The cardinality of a finite set is just the number of elements in the set.</td>
</tr>
<tr>
<td>chain</td>
<td>Short for blockchain in this document.</td>
</tr>
<tr>
<td>check all subsets</td>
<td>An input method for the state machine model in which any subset of the set of provision keys received so far can potentially trigger a state transition.</td>
</tr>
<tr>
<td>(digital) circuit</td>
<td>See computational circuit. The state-updating block of Figure 3-1 can also be considered as a monolithic circuit consisting of a single large gate.</td>
</tr>
<tr>
<td>circuit layer</td>
<td>See logic layer.</td>
</tr>
<tr>
<td>(computational) circuit (model)</td>
<td>An efficient model of computation that can be described in terms of a directed acyclic network (or circuit) connecting primitive units called gates, which can be thought of as operations for assigning to specific internal state variables (ISVs) values that are functions of other ISVs.</td>
</tr>
<tr>
<td>clock cycle</td>
<td>See time step, application cycle.</td>
</tr>
<tr>
<td>collusion (problem)</td>
<td>This refers to a situation in which multiple protocol participants combine their access information in ways that allow them to infer properties of the Machine that they could not otherwise discover. In general, GABLE’s privacy properties are only guaranteed to be maintained assuming that collusion does not occur (and that requisite access information is not stolen from participants).</td>
</tr>
<tr>
<td>Combinator</td>
<td>Any standardized associative, commutative binary operator for combining bit strings that exhibits a low probability of generating collisions on random data. Some choices include binary addition, bitwise XOR, and sorted concatenation.</td>
</tr>
<tr>
<td>commitment</td>
<td>A cryptographic commitment to some data refers to some encoded information that commits to the identity of that data, where the data itself may not yet be revealed. Examples can include a cryptographic hash of, or digital signature for the data. To unlock a commitment means to reveal the original matching data.</td>
</tr>
<tr>
<td>Company, the</td>
<td>Whatever entity creates/deploy a given GABLE application.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>computational circuit</td>
<td>Any particular circuit in the circuit model of computation.</td>
</tr>
<tr>
<td>computational gate</td>
<td>See gate below.</td>
</tr>
<tr>
<td>computation step</td>
<td>In §9 of this document, this phrase refers to the execution of a single layer of application logic gates within a multi-layer computational circuit whose evaluation comprises the state-updating phase of an application cycle.</td>
</tr>
<tr>
<td>concrete (finite) state machine</td>
<td>In the GABLE compiler, a concrete state machine is a version of the application's finite state machine that has had concretization transformations applied.</td>
</tr>
<tr>
<td>concretization</td>
<td>A process of transforming a finite state machine to prepare it for garbling by unrolling the state sequence, splitting reconvergent arcs, and delaying outputs.</td>
</tr>
<tr>
<td>condition</td>
<td>See transition condition.</td>
</tr>
<tr>
<td>conditional state transition</td>
<td>This refers to a directed arc in a state transition graph that is labeled with a set of transition conditions, which must all be satisfied in order for the transition to occur. See also state transition.  (All state transitions in GABLE are considered to be conditional.)</td>
</tr>
<tr>
<td>contract</td>
<td>See smart contract.</td>
</tr>
<tr>
<td>cryptographic hash (function)</td>
<td>This is a hash function that satisfies certain standard cryptographically desirable properties, such as preimage resistance and collision resistance; and its I/O relation (although deterministic) should be effectively random.</td>
</tr>
<tr>
<td>current (machine) state</td>
<td>The state $s(t)$ of the state machine during the current time step $t$, before the state-updating phase has completed.</td>
</tr>
<tr>
<td>cyber-physical system</td>
<td>A cyber-physical system (CPS) [2] denotes a real-world system, typically networked, that is subject to some form of external (e.g., network-based) control.</td>
</tr>
<tr>
<td>cycle</td>
<td>In a directed graph, a cycle would be a sequence of directed arcs that are connected head-to-tail to form a closed loop in the graph. In this document, we sometimes also use the word cycle as a synonym for time step. Finally, in §9 we occasionally use cycle to refer specifically to an application cycle, as distinct from a computation step.</td>
</tr>
<tr>
<td>DAG</td>
<td>This stands for directed acyclic graph. This refers to a graph in which nodes are connected by directed arcs, but where there are no cycles.</td>
</tr>
<tr>
<td>dApp</td>
<td>Decentralized application. A user-facing (e.g., web-based) application in which the back-end support runs on a blockchain or other distributed system.</td>
</tr>
<tr>
<td>data entry</td>
<td>Refers to a particular field within a particular instance of a given data structure.</td>
</tr>
<tr>
<td>data source</td>
<td>See Source.</td>
</tr>
<tr>
<td>destination state</td>
<td>For an arc in the state transition diagram for a given state machine, the destination state is the state at the head of the arrow.</td>
</tr>
<tr>
<td>deterministic</td>
<td>A given Machine is called deterministic if and only if, at any time step, and in any machine state, and for any set of input values that may be received in that cycle, there is at most one possible immediate successor state. If a given Machine is not deterministic, we call it nondeterministic or stochastic.</td>
</tr>
<tr>
<td>deterministic finite automaton (DFA)</td>
<td>Another term for a (deterministic) finite state machine.</td>
</tr>
<tr>
<td>DFA</td>
<td>See deterministic finite automaton.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>digital signature</td>
<td>Digital data that authenticates the identity of the sender of a given message. A digital signature can be generated using a public-key cryptosystem if the sender encrypts the message with his or her private key.</td>
</tr>
<tr>
<td>directed arc/edge</td>
<td>An arc in a graph represented by a (single-headed) arrow. We say that the arc is directed from the node at the tail of the arrow to the node at its head.</td>
</tr>
<tr>
<td>directed graph</td>
<td>This is a graph in which all arcs are directed arcs.</td>
</tr>
<tr>
<td>distributed consensus</td>
<td>This phrase refers generally to methods by which the various entities making up a distributed system may reliably reach a consensus as to the answer to some question. Examples of distributed consensus mechanisms include voting systems as well as blockchain-based distributed ledger technologies.</td>
</tr>
<tr>
<td>distributed ledger</td>
<td>This phrase refers to a capability provided by a distributed system to maintain a (nominally consistent and reliable) ledger, or in other words an accounting system for some type of data.</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System, the Internet's hierarchical naming system.</td>
</tr>
<tr>
<td>DOE</td>
<td>Abbreviation for the (United States) Department of Energy.</td>
</tr>
<tr>
<td>DOS (attack)</td>
<td>A Denial-of-Service attack refers to a scenario in which an adversary attempts to reduce the availability of a networked computing resource to its legitimate users, typically by overwhelming the resource with illegitimate requests (spam).</td>
</tr>
<tr>
<td>edge</td>
<td>See arc.</td>
</tr>
<tr>
<td>end symbol</td>
<td>See Finish symbol.</td>
</tr>
<tr>
<td>entry identifier</td>
<td>A key, derived from the arc identifier, for decrypting a specific data entry or field of a given instance of an encrypted arc data structure.</td>
</tr>
<tr>
<td>ETH</td>
<td>Abbreviation for 1 Ether.</td>
</tr>
<tr>
<td>Ether</td>
<td>The basic monetary unit in Ethereum, it is abbreviated ETH. It is occasionally also called 1 buterin to honor Vitalik Buterin, Ethereum’s creator.</td>
</tr>
<tr>
<td>Ethereum</td>
<td>A popular programmable blockchain platform; see <a href="http://www.ethereum.org">www.ethereum.org</a>.</td>
</tr>
<tr>
<td>Ethereum Virtual Machine</td>
<td>The Ethereum Virtual Machine (EVM) defines the abstract computing platform on which smart contracts run in the Ethereum programmable blockchain.</td>
</tr>
<tr>
<td>evaluation phase</td>
<td>A phase of the execution cycle in the circuit model, during which one or more computation steps are carried out to evaluate corresponding layers of the application circuit. At the end of the evaluation phase, the machine’s internal state variables are updated with the outputs from the last circuit layer. See also state-updating phase.</td>
</tr>
<tr>
<td>exclusive OR</td>
<td>See XOR.</td>
</tr>
<tr>
<td>execution cycle</td>
<td>See application cycle.</td>
</tr>
<tr>
<td>Executor</td>
<td>A generic interpreter “Exec” for garbled machines with a specific I/O model.</td>
</tr>
<tr>
<td>executable machine</td>
<td>A self-interpreting smart contract Exec[G] in which a generic Executor is applied to the garbled machine data G.</td>
</tr>
<tr>
<td>fairness problem</td>
<td>See lookahead problem, and also the discussion in §5.4.1. More specifically, the fairness problem refers to situations in which a player may gain a tactical advantage by looking ahead at immediate outputs.</td>
</tr>
<tr>
<td>False</td>
<td>Together with True, this is one of the two Boolean truth values.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FHE</td>
<td>Fully Homomorphic Encryption; a strong form of homomorphic encryption (HE).</td>
</tr>
<tr>
<td>field</td>
<td>A location within a data structure, or a column in a table.  See also input field.</td>
</tr>
<tr>
<td>final state / halting state</td>
<td>A state in a state machine that has no successor states.</td>
</tr>
<tr>
<td>finite state machine</td>
<td>A state machine that has a finite (as opposed to infinite) number of states.  Also known in some circles as a deterministic finite automaton (DFA).</td>
</tr>
<tr>
<td>Finisher</td>
<td>A participant whose role is to send a Finish symbol to the Machine to reveal the Machine’s final output after all other computation is complete.</td>
</tr>
<tr>
<td>Finish symbol/token</td>
<td>A special input symbol (which we denote with Unicode “⊝”, or ASCII “F,” or possibly the nonprintable ASCII character DC4/control-T) which means roughly, “produce final output and halt,” which may be used to trigger the completion of Machine execution in some application scenarios.</td>
</tr>
<tr>
<td>FSM</td>
<td>See finite state machine, above.</td>
</tr>
<tr>
<td>functional encryption</td>
<td>A paradigm for secure computing; see §10.4.</td>
</tr>
<tr>
<td>(computational) functionality</td>
<td>An abstract description $F$ of an application’s functional behavior, in a form that can be translated into a state machine or computational circuit.</td>
</tr>
<tr>
<td>functional privacy</td>
<td>This refers to the property that (under requisite assumptions) all possible aspects of a given computational process are obscured from (inscrutable to) outside parties.  This is a key security goal achieved by the design of GABLE.</td>
</tr>
<tr>
<td>GABLE</td>
<td>Garbled Autonomous Bots Leveraging Ethereum – The name of the system concept described in this document.</td>
</tr>
<tr>
<td>garble</td>
<td>For our purposes, to garble an element of a computational process (all or part of the program code or data) means to encrypt (scramble or encode) it in such a way that it is impossible to infer its meaning, for all entities not possessing the required decryption key(s).</td>
</tr>
<tr>
<td>garbled (lookup) table</td>
<td>A garbled data structure that supports looking up one or more result fields, which are indexed by combining values of one or more input fields, but where the meaning of the input values, result fields, and the function computed by the lookup table are completely obscured by the garbling process.  The state-machine version of GABLE (§§5–6) uses garbled lookup tables to encode state transition functions.  A future universal-circuits version of GABLE (§9) will use garbled lookup tables to encode routing elements and application gates.</td>
</tr>
<tr>
<td>garbled (state) machine</td>
<td>A computational process (maybe using the state machine model), encoded in a garbled but still-executable form.  Also referred to abstractly with the symbol $G$.</td>
</tr>
<tr>
<td>garbled (universal) circuit (GUC)</td>
<td>A computational process using the circuit model, represented in a garbled but still-executable form $G$.  When we emphasize that a garbled circuit is universal, this means that its function cannot be inferred by inspecting its structure.  Occasionally, we also use the phrase “garbled circuit” to refer to the garbled state-updating block in the state machine model, which can be considered to be a sort of monolithic circuit.</td>
</tr>
<tr>
<td>Garbler</td>
<td>The entity that produces the garbled representation $G$ of a given application-specific computational functionality $F$.</td>
</tr>
<tr>
<td>gas</td>
<td>A measure of computational cost used on the Ethereum platform.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>(computational) gate</td>
<td>A function that computes the value of an output variable given the values of a small fixed number of input variables, or, an instance of the application of some such function to compute a new value of a state variable in terms of old values of other state variables.</td>
</tr>
<tr>
<td>gather all inputs</td>
<td>A simple input method under the multiple source input model in which (for state machines) garbled input provision keys are accepted from authorized participants until an arc is matched. Not resilient to malicious participants.</td>
</tr>
<tr>
<td>gather $N$ out of $M$ inputs</td>
<td>An input method under the multiple source input model in which only $N$ out of $M$ input variables need to be assigned values before the state can update.</td>
</tr>
<tr>
<td>GCN</td>
<td>See generalized connection network below.</td>
</tr>
<tr>
<td>generalized connection network</td>
<td>Described in Thompson 1977 [3], this is a configurable interconnect network that can be configured to assign each of $N$ outputs from any of $N$ inputs.</td>
</tr>
<tr>
<td>go symbol/token</td>
<td>See Proceed symbol.</td>
</tr>
<tr>
<td>graph</td>
<td>An abstract model of a system of nodes connected by arcs; graphs may be used to represent a very wide variety of different structures, including abstract relations, networks, polyhedra, and state machines.</td>
</tr>
<tr>
<td>GUC</td>
<td>See garbled universal circuit.</td>
</tr>
<tr>
<td>Gwei</td>
<td>1 Gwei, also known as 1 shannon, is equal to 1 billion ($10^9$) wei or 1 nanoether.</td>
</tr>
<tr>
<td>halting state</td>
<td>See final state / halting state.</td>
</tr>
<tr>
<td>hash (function/value)</td>
<td>A hash function takes an object as input and produces a derived pseudo-random number within some range (the hash or hash value) as output. See also cryptographic hash.</td>
</tr>
<tr>
<td>hash table</td>
<td>A hash table is a data structure that supports fast lookup of elements by utilizing a hash (not necessarily cryptographically secure) of the item to suggest a storage location for the item. In Solidity, mapping types implement hash tables.</td>
</tr>
<tr>
<td>HE</td>
<td>Abbreviation for homomorphic encryption.</td>
</tr>
<tr>
<td>homomorphic encryption</td>
<td>A paradigm for secure computing; see §10.2.</td>
</tr>
<tr>
<td>indistinguishability obfuscation</td>
<td>Abbreviated IO; this is a paradigm for secure computing. See §10.3.</td>
</tr>
<tr>
<td>initial state</td>
<td>This refers to the state that a given state machine occupies initially, before it has performed any computation steps. In some variations of the GABLE protocol, a separate protocol participant of a special type called a Starter may supply a (garbled) initial state key $K(s_{init})$ to start the machine’s execution in a particular state after its smart contract has been deployed to the blockchain.</td>
</tr>
<tr>
<td>IO</td>
<td>See indistinguishability obfuscation.</td>
</tr>
<tr>
<td>I/O model</td>
<td>A particular combination of an input model and an output model.</td>
</tr>
<tr>
<td>information source</td>
<td>See Source.</td>
</tr>
<tr>
<td>Initializer</td>
<td>See Starter.</td>
</tr>
<tr>
<td>input cycle</td>
<td>See application cycle.</td>
</tr>
<tr>
<td>input field</td>
<td>In a lookup table, for our purposes, an input field refers to a particular column of the table such that, together with any other input fields, the input values in those</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>Term</td>
<td>fields define a <em>key</em> that is used to select a matching subset of table rows and identify the selected result value(s).</td>
</tr>
<tr>
<td>input-gathering phase</td>
<td>This refers to the first portion of each application cycle, during which the Executor gathers input values from providers until the machine state can be updated. How this phase works in detail is specified by the input model.</td>
</tr>
<tr>
<td>input (value) key</td>
<td>This refers to a random $n$-bit key designated $K(v_i^j, t)$ generated by the Garbler, which authorizes an entity to supply the value $v_i^j$ to input line $V_i$ on time step $t$.</td>
</tr>
<tr>
<td>(external) input line / input variable</td>
<td>Usually these phrases refer to a particular designated input channel $V_i$ coming into the Machine from its external environment. Often these two terms are used interchangeably in this document, but we may sometimes prefer to say that a given <em>input variable</em> $V_i$ is an input <em>channel</em> whose existence persists over time, while its <em>value</em> $v_i(t)$ is a function of time, and varies across time steps; while (in a circuit picture) we may prefer to say that there is a separate <em>input line</em> $V_i^j = V_i(t)$ that feeds into each layer of the Machine’s state-updating circuitry, corresponding to the circuit layer for computing each time step $t$. However, occasionally in this document, these phrases are instead used more generically to refer to any conceptual channel coming into a given <em>circuit</em>, <em>gate</em>, or <em>garbled lookup table</em>, not necessarily an external one. So, for example, the word “input” in such contexts could refer to a line carrying internal state information into the element, and not necessarily to externally supplied input.</td>
</tr>
<tr>
<td>input (gathering) method</td>
<td>A protocol that specifies the rules determining (1) the process by which input providers provide input values to a machine, and (2) how those input values are used to determine how the machine’s state will be updated. The phrase <em>input method</em> typically refers to a more concrete/specific example of an <em>input model</em>.</td>
</tr>
<tr>
<td>input model</td>
<td>See also <em>input method</em>. An <em>input model</em> is the same general idea, but at a more abstract level. Examples of input models include <em>single source</em> and <em>multiple source</em>.</td>
</tr>
<tr>
<td>input provider/source</td>
<td>See <em>Provider</em>.</td>
</tr>
<tr>
<td>input provision key</td>
<td>A state-dependent key that is used to supply a given input value to the Machine on a given time step, when the Machine is in a particular state. Requiring a state-dependent key prevents certain replay attacks (that could otherwise compromise privacy) in which input keys already seen previously are applied to an alternate sequence of states. See the <em>Reconvergent Paths Problem</em> discussed in §5.4.3 for more information.</td>
</tr>
<tr>
<td>input symbol/token</td>
<td>A generic symbolic meaning associated with a particular <em>input value</em>. For example, an <em>input variable</em> $A$ could take on the <em>input value</em> $1_A$ representing the symbolic token ‘1.’</td>
</tr>
<tr>
<td>input value</td>
<td>A specific possible value (designated $v_i^j$) that may be associated to a specific <em>input variable</em> $V_i$. May represent a (more abstract) <em>input symbol</em>.</td>
</tr>
<tr>
<td>input variable</td>
<td>See the entry for <em>input line / input variable</em>, above.</td>
</tr>
<tr>
<td>interconnect fabric</td>
<td>A configurable network that can be used to route data in a circuit. See also <em>routing network</em>.</td>
</tr>
<tr>
<td>internal state</td>
<td>In the context of finite state machines, this refers to the <em>machine state</em>.</td>
</tr>
<tr>
<td>internal (state) line/variable</td>
<td>In the circuit model of computation, instead of having a single monolithic machine state, we have multiple independent <em>internal state variable</em> or ISVs (often binary), corresponding to wires crossing a bisection of the circuit.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ISV / i.s.v.</td>
<td>See internal state variable above.</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation is a simple format for representing structured data.</td>
</tr>
<tr>
<td>Keccak</td>
<td>A family of cryptographic hash functions that includes SHA-3. Note that the specific function in the Keccak family that is supported by Ethereum, called keccak, is not in fact identical to the final SHA-3 standard.</td>
</tr>
<tr>
<td>(cryptographic) key</td>
<td>In this document, the word “key” is used generally to refer to any information that is needed to decrypt or unlock some other information or process. Usually for us, keys will be random bit vectors of some prespecified length. Examples of specific types of keys in GABLE that we discuss in this document include activation keys, arc identifiers, entry identifiers, input provision keys, input value keys, output keys, participant keys, state keys, and time step keys. See also the individual glossary entries for these phrases.</td>
</tr>
<tr>
<td>layer</td>
<td>See logic layer below.</td>
</tr>
<tr>
<td>LDRD</td>
<td>Laboratory-Directed Research and Development – The internally-funded research programs at the DOE National Laboratories.</td>
</tr>
<tr>
<td>line</td>
<td>A conceptual wire or communication channel in a circuit picture of a Machine. Lines in GABLE include input lines coming into a given time step, state lines or internal state variable lines running in between time steps, and various other internal bit lines in the universal circuit implementations discussed in §9.</td>
</tr>
<tr>
<td>logic gate</td>
<td>A gate that operates on binary (two-state) inputs and produces a binary output. See also Boolean gate.</td>
</tr>
<tr>
<td>logic layer/level</td>
<td>Any computational circuit can be organized into layers (or levels), where all of the gates in each layer can operate simultaneously with each other, and they all feed their outputs to the inputs of gates in the next layer and/or subsequent layers. Layers can be evaluated in sequential order.</td>
</tr>
<tr>
<td>lookahead (problem)</td>
<td>By lookahead, we mean a situation where an input provider privately considers the effect, on resulting Machine execution, of providing one or more alternate input values to the current (or a past) state of the Machine, but without actually providing it. In some circumstances, this could allow the provider to infer some information about the Machine’s function or output that we might not want it to obtain, and to modify its own behavior accordingly. We call this the lookahead problem, and identifying methods to prevent this problem from arising was one of the challenges that was overcome in the design of the GABLE protocol. See also the related reconvergent arcs and reconvergent paths problems, which can be considered as special cases of lookahead.</td>
</tr>
<tr>
<td>lookup table</td>
<td>For our purposes, a lookup table refers generally to a data structure that can be visualized as a two-dimensional array, whose columns can be divided into input fields and result fields. One operation supported by a lookup table is to query it by specifying values for all of the input fields and retrieve the corresponding value(s) for the result fields. GABLE’s technology relies heavily on garbled versions of lookup tables; see the garbled lookup table entry.</td>
</tr>
<tr>
<td>Machine</td>
<td>Used here occasionally as shorthand for garbled machine. The word “machine” (not capitalized) may also refer to an ordinary state machine, before garbling.</td>
</tr>
<tr>
<td>machine state</td>
<td>A state of a state machine.</td>
</tr>
<tr>
<td>magic cookie</td>
<td>An easily recognizable constant bit string that renders a data object identifiable.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Mealy machine</td>
<td>In a <em>Mealy</em>-type finite state machine, the output from each time step depends on the arc or transition that was taken. In other words, it depends on both the old state, and the input that was provided (since these determine the transition). Contrast <em>Moore machine</em>. There are a number of ways that Mealy machine type behavior could be facilitated in GABLE, but this has not been our focus.</td>
</tr>
<tr>
<td>Millionaires’ Problem</td>
<td>A classic problem of MPC in which two parties A and B wish to jointly determine which of them is wealthier without disclosing any other information.</td>
</tr>
<tr>
<td>Moore machine</td>
<td>In a <em>Moore</em>-type finite state machine, the output from each time step depends only on the new machine state. Contrast <em>Mealy machine</em>. Our existing GABLE demos have generally been Moore machines, but not out of necessity.</td>
</tr>
<tr>
<td>move</td>
<td>If one thinks of a GABLE computation as a game, then a <em>move</em> by a player in the game (input provider) refers to the player’s action in supplying values for one or more input variables during a given application cycle.</td>
</tr>
<tr>
<td>MPC</td>
<td>(Secure) Multi-Party Computation; see <em>SMC</em>.</td>
</tr>
<tr>
<td>multiple-source input</td>
<td>This refers to input models in which, on a given cycle, multiple inputs must be received from providers before the machine state is updated. Several subtypes of multiple-source input methods are described in §5.3.</td>
</tr>
<tr>
<td>NAND</td>
<td>A Boolean logical gate whose output is False if all its inputs are True.</td>
</tr>
<tr>
<td>NNSA</td>
<td><em>National Nuclear Security Administration</em>, a sub-agency within <em>DOE</em>.</td>
</tr>
<tr>
<td>node</td>
<td>In an abstract graph, nodes are connected by arcs. In this document, the word “node” may also sometimes refer to a host, meaning a specific independently operating computer on a computer network.</td>
</tr>
<tr>
<td>NOR</td>
<td>A Boolean logical gate whose output is False if any of its inputs is True.</td>
</tr>
<tr>
<td>NOT</td>
<td>A Boolean logical gate whose output is True if its (sole) input is False.</td>
</tr>
<tr>
<td>nondeterministic</td>
<td>For our purposes in this report, a given machine is called <em>nondeterministic</em> if and only if it is not <em>deterministic</em>, that is, if there are possible situations in which it could immediately transition to any of multiple different next states given its current state and the inputs for the current cycle. This report does not specify a particular scheme for operation of nondeterministic machines.</td>
</tr>
<tr>
<td>Observer</td>
<td>See <em>Spectator</em>.</td>
</tr>
<tr>
<td>one-time pad</td>
<td>A cryptographic technique based on bitwise-XOR‘ing the message to be encrypted with a random key that is the same length as the message and that is not reused. This encoding is theoretically unbreakable when the key is completely unknown. GABLE’s encryption methods can be considered to be based on one-time pads using the values of cryptographic hashes as keys.</td>
</tr>
<tr>
<td>OR</td>
<td>A Boolean logical gate whose output is True if any of its inputs is True.</td>
</tr>
<tr>
<td>origin state</td>
<td>For an arc in the state transition diagram for a given state machine, the origin state is the state at the tail of the arrow.</td>
</tr>
<tr>
<td>out-conditions</td>
<td>When we speak of the <em>out-conditions</em> of a given state, we are referring to the set of condition sets associated with that state’s outgoing arcs.</td>
</tr>
<tr>
<td>output key</td>
<td>A key that is required for decrypting (extracting meaningful information from) one or more possible outputs from a given Machine.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>output message</td>
<td>An encrypted message string embedded in the contract that may be decrypted by spectators holding a certain output key when the machine decodes a certain transition (for Mealy machines) or a certain state (for Moore machines).</td>
</tr>
<tr>
<td>output model</td>
<td>A protocol specifying the rules determining the process by which authorized spectators may view specified outputs from machine states and/or transitions.</td>
</tr>
<tr>
<td>output variable</td>
<td>Although not utilized in the present implementations of GABLE, this phrase could be used to refer to a designated output channel from a given machine. This phrase may also be used to refer to the output fields in a lookup table.</td>
</tr>
<tr>
<td>output viewer</td>
<td>See Observer.</td>
</tr>
<tr>
<td>P2P</td>
<td>Abbreviation for Peer-to-Peer. Refers to a distributed network with no central server. Blockchain-based systems typically run on top of P2P networks.</td>
</tr>
<tr>
<td>participant</td>
<td>Any entity participating in a GABLE protocol. Specific participant roles that we mention in this document include the Garbler, Starter, Provider, Stepper, Unlocker, Spectator, and Finisher roles.</td>
</tr>
<tr>
<td>participant key</td>
<td>In an optimized key-distribution scheme, a single randomly generated participant key $K_i$ is securely distributed to the $i$th protocol participant; this then effectively becomes the seed for deterministically (but pseudo-randomly) deriving all the various access keys delegated to that participant.</td>
</tr>
<tr>
<td>PKI</td>
<td>Public Key Infrastructure, a system for managing keys for a public-key (i.e., asymmetric) cryptosystem.</td>
</tr>
<tr>
<td>player</td>
<td>If one thinks of a GABLE computation as a game, then the input providers (with or without Spectator abilities) could be thought of as players who are making moves in the game. (Other types of protocol participants may or may not also be thought of as players, depending on what story you want to tell.)</td>
</tr>
<tr>
<td>point-and-permute</td>
<td>An optimization of garbled lookup tables, introduced in [4], in which randomly assigned but unique point bits associated with each possible value of a given input variable are concatenated together over the variables and used to form a table row index. A disadvantage of using this method in GABLE is that it reveals an upper bound on the number of possible values of each variable.</td>
</tr>
<tr>
<td>private key</td>
<td>In a public-key cryptosystem, the private key is a key that is held privately by its owner, which can be used to decrypt messages encrypted using the corresponding public key, or generate digital signatures that can be verified using the public key. Also in this document we use the phrase “private key” more generally to refer to any privately-held encryption key (including for symmetric cryptosystems).</td>
</tr>
<tr>
<td>Proceed symbol/token</td>
<td>A special input symbol or token (which we can denote with Unicode “⊚”, or possibly the nonprintable ASCII character DC1/control-Q) meaning “proceed forwards to the next state,” which is necessary to advance Machine execution forwards in some input models. In some variants of GABLE, there may be several different proceed symbols $⊚_v$ depending on a previous input $v$.</td>
</tr>
<tr>
<td>Provider</td>
<td>Short for “input provider;” an entity providing (garbled) input data to a Machine.</td>
</tr>
<tr>
<td>provision key</td>
<td>See input provision key.</td>
</tr>
<tr>
<td>public key</td>
<td>In a public-key cryptosystem, the public key is a key that is openly released, which can be used to encrypt messages to (or verify signatures from) the owner of the corresponding private key.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>public-key cryptosystem</td>
<td>An asymmetric cryptographic framework in which there are two different, complementary keys, each of which can be used to decrypt messages that the other one encrypts. One of the two keys can be released publicly.</td>
</tr>
<tr>
<td>Python</td>
<td>A popular programming language which was used to implement our prototype Garbler.</td>
</tr>
<tr>
<td>Reader Synonym for Spectator.</td>
<td></td>
</tr>
<tr>
<td>reader key</td>
<td>A key for read access to a Machine, associated to a specific reading authority. See also output key.</td>
</tr>
<tr>
<td>reader/reading authority</td>
<td>Associated with a body of <em>access information</em> that enables the holder of that information to specify on specific outputs from the Machine at specific times.</td>
</tr>
<tr>
<td>reconvergent arcs (problem)</td>
<td>This refers to a situation in which there are two or more arcs from a given origin state $O$ to a given destination state $D$ which can be activated by the actions of a given input provider which is the last one to move on a given time step. This would weaken the privacy properties of the garbled state machine, so we avoid it using a process of reconvergent arc elimination, discussed in §5.4.2.</td>
</tr>
<tr>
<td>reconvergent paths (problem)</td>
<td>This refers to a scenario in which there are two or more paths in the state graph from a given origin state $O$ to some state (multiple cycles later) that are distinguished only by the input value provided by a given input provider in state $O$. The possibility of this scenario would weaken the privacy properties of the garbled state machine, if not for our use of input provision keys.</td>
</tr>
<tr>
<td>replay attack</td>
<td>In computer security, a replay attack [5] refers generally to a class of exploits in which some previously seen data is maliciously re-injected into the system by an adversary, with the effect of compromising the security goals of the system. In this document, we extend the scope of this term to include scenarios where the adversary only mentally replays previously seen data (in a sort of imagined alternate timeline) in a way that would allow them to infer information about the Machine that would violate our privacy goals. GABLE is designed to resist this class of exploits (assuming no collusion between participants).</td>
</tr>
<tr>
<td>result field</td>
<td>An output field of a lookup table.</td>
</tr>
<tr>
<td>result state</td>
<td>A final state whose output is readable by some parties (other than the Garbler).</td>
</tr>
<tr>
<td>Python</td>
<td>A popular programming language, used to code our example Garbler (App. B).</td>
</tr>
<tr>
<td>round-robin input method/model</td>
<td>A variant of the single-shot update method in which input providers take turns providing input values on subsequent application cycles in a predefined order.</td>
</tr>
<tr>
<td>routing element/unit</td>
<td>A unit cell making up an interconnect fabric. Generically, its function is to assign each of its outputs from a specific one of its inputs. An illustration of a two-input, two-output routing unit is shown in Figure 9-2 on p. 72.</td>
</tr>
<tr>
<td>routing network</td>
<td>See interconnect fabric. Note that a GCN is one type of routing network.</td>
</tr>
<tr>
<td>SAND Report</td>
<td>A technical report produced at Sandia National Laboratories, such as this one.</td>
</tr>
<tr>
<td>Sandia (National Laboratories)</td>
<td><em>Sandia National Laboratories</em> (SNL) is the flagship US DOE national laboratory; its contract is issued/overseen by NNSA.</td>
</tr>
<tr>
<td>secure computing / secure computation</td>
<td>For our purposes in this document, this phrase refers generally to any advanced computing method that provides a strong guarantee of properties such as the privacy, integrity, reliability, availability, and/or auditability of the computational process and/or results.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>selector gate</td>
<td>A simple type of computational gate that simply assigns to its output from a predetermined one of its inputs.</td>
</tr>
<tr>
<td>SHA-2, SHA-3</td>
<td>Families of cryptographic hash functions defined by two (very different) releases of the <em>Secure Hash Algorithm</em> standard endorsed by the U.S. government. See also Keccak.</td>
</tr>
<tr>
<td>SHA-256</td>
<td>A hash function that is a 256-bit instance of either SHA-2 or SHA-3.</td>
</tr>
<tr>
<td>shannon</td>
<td>Claude Shannon invented both Boolean digital circuits and information theory, and his name is sometimes used for the Gwei or nanoether in his honor.</td>
</tr>
<tr>
<td>signature</td>
<td>See <em>digital signature</em>.</td>
</tr>
<tr>
<td>single-shot update</td>
<td>This refers to an input method that implements the single-source input model.</td>
</tr>
<tr>
<td>single-source input</td>
<td>This refers to an input model in which, on any given input cycle, only a single input value from a single input provider is received before the machine state is updated.</td>
</tr>
<tr>
<td>smart contract</td>
<td>A program for executing transactions that enforces its own contractual terms.</td>
</tr>
<tr>
<td>SMC</td>
<td><em>Secure Multiparty Computation</em>, a paradigm for secure computing. See §10.1.</td>
</tr>
<tr>
<td>Solidity</td>
<td>A programming language for writing <em>smart contracts</em> for the <em>Ethereum</em> platform.</td>
</tr>
<tr>
<td>Source</td>
<td>In this document, short for “<em>information source</em>” or “<em>input source.</em>” See <em>Provider</em>.</td>
</tr>
<tr>
<td>Spectator</td>
<td>An entity that can view and interpret output from a (garbled) Machine.</td>
</tr>
<tr>
<td>spectator key</td>
<td>See <em>output key</em>.</td>
</tr>
<tr>
<td>Starter</td>
<td>This is a special type of protocol participant that could exist in some variants of GABLE. A <em>Starter</em> holds a special type of access authority that enables it to set the initial state of a deployed Machine that has not yet been initialized. (This capability is not yet implemented in our present prototypes and demos.)</td>
</tr>
<tr>
<td>Stepper</td>
<td>A special type of protocol participant that could exist in some versions of GABLE. The role of a Stepper is to send special <em>Proceed symbols</em> that cause the Machine to advance to the next state after ordinary inputs have been received. This can provide a way to avoid issues with lookahead or reconvergent arcs.</td>
</tr>
<tr>
<td>state</td>
<td>A particular configuration of a <em>state machine</em>, or a value of a <em>state variable</em>.</td>
</tr>
<tr>
<td>state (transition) diagram/graph</td>
<td>A representation of the functionality of a <em>finite state machine</em> using a <em>graph</em>. See <em>state transition diagram/graph</em> below.</td>
</tr>
<tr>
<td>state key</td>
<td>A randomly generated key for designating that a given <em>state line</em> within the Machine (for a given time step) is assigned to a given state.</td>
</tr>
<tr>
<td>state line</td>
<td>In the state-machine model, this refers to a conceptual wire $S_t$ that runs between the state-updating circuits for time step $t$ and time step $(t+1)$ respectively, except that, conceptually, $S_{-1}$ feeds the initial state into the Machine, and $S_{\ell-1}$ feeds the final state after time step $(\ell-1)$ out of the Machine. In the circuit model (§9), wires associated with individual internal state variables (as well as internal wires within the routing network) can also be considered to comprise types of state lines.</td>
</tr>
<tr>
<td>(explicit / monolithic) state machine (model)</td>
<td>A simple type of computational model, specifying an abstract machine that can change its <em>state</em> according to specified <em>transitions</em> activated by the input data. All state machines dealt with in this project qualify as <em>finite state machines</em>.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>State machine behavior</td>
<td>State machine behavior can be implemented using more efficient computational circuits rather than by state transition graphs; we would not refer to such an implementation as an &quot;explicit&quot; or &quot;monolithic&quot; state machine, however.</td>
</tr>
<tr>
<td>state transition diagram/graph</td>
<td>A visual depiction of an FSM using a directed graph representation. In a state transition diagram, states are represented by nodes and possible transitions are represented by directed arcs labeled with transition conditions.</td>
</tr>
<tr>
<td>state-updating phase</td>
<td>The last part of each application cycle, during which the new machine state is computed. This can be done all at once, as in the monolithic state machine model, or over several computation steps in the circuit model (see evaluation phase).</td>
</tr>
<tr>
<td>state variable</td>
<td>See internal state variable.</td>
</tr>
<tr>
<td>step</td>
<td>See time step.</td>
</tr>
<tr>
<td>Stepper</td>
<td>A special type of protocol participant that, in some input models, is required to send a special proceed token &quot;◉&quot; to the Machine to tell it to go ahead and proceed forwards to the next state, after any other inputs have been received.</td>
</tr>
<tr>
<td>stochastic</td>
<td>A given state machine is called stochastic if its behavior is (pseudo-) random. Such a machine is not deterministic. Stochastic machines are not currently supported in GABLE but could easily be added. See also nondeterministic.</td>
</tr>
<tr>
<td>successor state</td>
<td>In a finite state machine, state $s_j$ is a successor of state $s_i$ if and only if the machine's state transition graph includes an arc directed from $s_i$ to $s_j$.</td>
</tr>
<tr>
<td>symbol / token</td>
<td>See input symbol.</td>
</tr>
<tr>
<td>symmetric cryptosystem</td>
<td>A cryptographic system in which the same key is used both to encrypt and decrypt messages. Contrast asymmetric cryptosystem.</td>
</tr>
<tr>
<td>table</td>
<td>See lookup table.</td>
</tr>
<tr>
<td>Thompson (interconnect) network</td>
<td>A particular construction for GCNs described in Thompson, 1977 [3].</td>
</tr>
<tr>
<td>time step</td>
<td>Refers to a time interval $t$ during which a Machine gathers external inputs and then updates its internal state. See also application cycle, computation step.</td>
</tr>
<tr>
<td>time step (unlock) key</td>
<td>A random key $k_t$ that is needed to unlock the evaluation of the application circuit for time step $#t$ in a (future) universal-circuit-based version of GABLE.</td>
</tr>
<tr>
<td>token</td>
<td>See symbol.</td>
</tr>
<tr>
<td>transition</td>
<td>A directed arc (arrow, edge) between states in the state graph of a finite state machine, labeled with a set $C$ of transition conditions. The transition conditions must all be met in order for the machine to follow a given arc to its destination.</td>
</tr>
<tr>
<td>transition condition</td>
<td>Specifies a condition that must be satisfied in order for a given transition between states in a state graph to occur. The nature of such a condition in GABLE is to require that some specific input value $v_{i'}$ must be assigned to a specific input variable $V_i$.</td>
</tr>
<tr>
<td>True</td>
<td>Together with False, this is one of the two Boolean truth values.</td>
</tr>
<tr>
<td>truth table</td>
<td>A lookup table that tabulates the value of the output variable of a given logic gate for each possible combination of values for its input variables.</td>
</tr>
<tr>
<td>truth value</td>
<td>A variable that represents a truth value, a.k.a. a Boolean variable, can take on the value True or False, often equated with 1 or 0, respectively.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>turn</td>
<td>If one thinks of a GABLE computation as a game, then a turn in the game would refer to a given application cycle; if there is only a single player “A” that may move on a given turn, then we can refer to that turn as “player A’s turn.”</td>
</tr>
<tr>
<td>UC or u.c.</td>
<td>See universal circuit, below.</td>
</tr>
<tr>
<td>Unassigned/ Undefined symbol</td>
<td>For a given input line $V_i$, this is a special symbol $\bot_i$ meaning roughly, “no values were assigned to input variable $V_i$ on this cycle.” If the encoded representation of $\bot_i$ is hard-coded into the contract, care should be taken in the protocol to ensure that this does not allow lookahead to occur.</td>
</tr>
<tr>
<td>undirected arc</td>
<td>An arc between two nodes in a graph that has no associated directionality. Could be modeled with a pair of directed arcs going in opposite directions.</td>
</tr>
<tr>
<td>universal circuit</td>
<td>A programmable computational circuit which can be configured to emulate any computational circuit up to a certain size, given appropriate configuration data.</td>
</tr>
<tr>
<td>Unlocker</td>
<td>In the efficient universal-circuit-based garbled computation protocol described in §9, an Unlocker is a special participant that unlocks (activates, decrypts) inputs to the first layer of the u.c. for computing a given time step.</td>
</tr>
<tr>
<td>value</td>
<td>A value is any conceptual entity that can be assigned to a variable.</td>
</tr>
<tr>
<td>variable</td>
<td>An abstract quantity that varies, that is, takes on different values in different circumstances. Important types of variables that are dealt with explicitly in GABLE include input variables and state variables.</td>
</tr>
<tr>
<td>vendor client</td>
<td>In the supply-chain provenance tracking demo (§7.1), a vendor client is an instance of a script that performs actions to emulate the behavior of an input provider called a vendor, which interacts with the garbled Machine to provide for on-blockchain provenance tracking with strict privacy controls.</td>
</tr>
<tr>
<td>verifiable computation</td>
<td>A paradigm for secure computing; see §10.4.</td>
</tr>
<tr>
<td>vertex</td>
<td>See node.</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network, a secure virtual network between hosts, implemented as a cryptographic overlay on top of an underlying network that is not private.</td>
</tr>
<tr>
<td>wei</td>
<td>1 wei is equal to $10^{-18}$ ether or 1 attoether. It is the smallest unit of Ether that can be transferred on the Ethereum blockchain. It is named after Wei Dai, who described an early concept for digital currency which he called “b-money.”</td>
</tr>
<tr>
<td>width</td>
<td>The width of a circuit or memory refers to the number of (usually binary-valued) lines bisecting the circuit, or to the number of internal state variables (usually bits) in the memory.</td>
</tr>
<tr>
<td>wire</td>
<td>In a computational circuit, a wire is a connection between gates that conceptually carries a data value. In abstract terms, it can be represented by an internal state variable. See also line.</td>
</tr>
<tr>
<td>Writer</td>
<td>Synonym for Provider.</td>
</tr>
<tr>
<td>writing authority</td>
<td>Associated with a body of access information that enables the holder of that information to supply particular values to specific input variables on specific time steps.</td>
</tr>
<tr>
<td>XOR</td>
<td>A Boolean logical gate whose output is True if an odd number of its inputs are True.</td>
</tr>
</tbody>
</table>
1. DOCUMENT DESCRIPTION

This document is intended for public release. It comprises the primary final report from the 3-year “Blockchain Derived Secure Computing” LDRD project at Sandia National Laboratories, which ran from 2017 to 2020. The primary purpose of this report is to outline the technical design specifications for a system named GABLE (an acronym for Garbled Autonomous Bots Leveraging Ethereum), which was developed at Sandia (in collaboration with the Georgia Institute of Technology) in fulfillment of project goals, in which garbled or encoded representations of simple state-machine circuits can be generated and then executed, with strong privacy properties, on robust smart-contract execution platforms, such as (as an initial target) the public Ethereum network.

1.1. Organization of this Document

The rest of this document is organized as follows:

Section 2 motivates the work by summarizing a few general categories of use-case scenarios that could benefit from the availability of a system such as GABLE, and Section 3 outlines some general requirements on the capabilities of the GABLE system that arise out of consideration of those scenarios. Section 4 describes some more specific example applications. Section 5 describes, at a conceptual level, some technical aspects of how present versions of GABLE function, and/or (in cases where not-yet-implemented features are described) how future versions are envisioned to function. Section 6 gives more detailed, concrete specifications of particular representations/encodings that are utilized by actual deployed GABLE instances, and discusses prototype implementation and testing. Section 7 describes several complete multi-node demonstration applications that have been developed and successfully tested at Sandia (in sandboxed environments) to exercise the new GABLE capability. Section 8 describes a concept for a state machine compiler (not yet implemented). Section 9 describes one way that the computation model used in GABLE can be made much more efficient without sacrificing privacy, and does some cost analysis. Section 10 surveys some of the related work in the literature, and compares and contrasts our paradigm for secure computation with several more standard ones. Section 11 outlines some possible directions for future work and concludes.

There are also several appendices: Appendix A tabulates the mathematical notation used in this document. Appendices B & C provide source code (in Python and Solidity) for a simple prototype implementation of the concepts described in this document. Appendix D describes how to exercise the GABLE prototype in a development environment.

A detailed table of contents may be found on pp. 5–6.
2. MOTIVATIONS AND BROAD USE-CASE SCENARIOS

We begin by noting that existing public programmable smart-contract blockchains such as Ethereum provide a computing platform that is highly robust and widely accessible. Similar but smaller private blockchains may also provide these characteristics, albeit to a lesser extent.

Any entity (which we’ll generally call “the Company” to be concise, although it may or may not be a business entity in a traditional sense) whose operations may sometimes require the execution of business or strategic logic that needs to function (using inputs from various providers or sources) with an extremely high degree of assurance of its reliability and availability may therefore wish to consider implementing that functionality in the form of a smart contract running on top of a blockchain.

However, depending on the nature of the Company’s operations, a very high degree of privacy for the required computational process (which we’ll generically refer to as “the Machine”) may also be necessary. Information that the Company may wish to keep private (not decipherable by unauthorized parties) may include the following:

- The real identities of any information sources that are providing inputs to the computation, and in some cases, even the number of distinct input sources that exist.
- The meanings of the actual input values that are provided to the computation, and in some cases, even the number of possible alternative values of each input that is provided to the computation (within very large bounds).
- 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.
- The meanings of any intermediate or final outputs produced by the computation.

Altogether, we consider obscuring as much of the above information as possible (where this turns out to mean all, or almost all of it) to constitute what we mean by attaining functional privacy.

One of the well-studied related notions from cryptography and computer science that is at play here is that of Secure Multiparty Computation (SMC or MPC) (cf. [6], [7], [8]) (see also §10.1), in which various entities supply inputs to a computational process, which is carried out via a protocol involving those entities, where a function of all those inputs gets computed by the process (and thence becomes common knowledge to the participants), but without revealing any other information about the values of the inputs publicly, or even to other participants. Ideally, a protocol for SMC should preserve these security guarantees even if no individual entity participating in the protocol is trusted. That is, each individual input provider is at most able to openly reveal the inputs that they themselves provided to the computation, but not the inputs provided by other participants. And, no trusted central computation server is assumed to be available, either.

In our scenario, we further specify that only the entity responsible for originally deploying the system (i.e., the Company), or its delegates, and not necessarily any of the individual input providers, will necessarily be able to inspect the output of the computation. The Company will also be able to inspect the inputs to and all intermediate and final results from the computation. (A pre-selected subset of outputs, may, however, be made available to other authorized parties.) Further, all participants, other than the Company itself, including any authorized input providers and output viewers, need not even be aware of what is the nature of the computation that is being performed by the Machine.

The net effect, therefore, in our scenario, is roughly equivalent (aside from some verifiability properties which we’ll discuss in Sec. 10.4) to one in which the entire computation was being performed
in private by the Company itself, after receiving encrypted inputs from the input providers—but the difference here is that we wish for the computational mechanism itself to operate autonomously, with very high reliability and availability. Ideally, the Company could simply deploy the computing system to a blockchain at any time (e.g., shortly before it is about to be needed), and then disappear from the scene indefinitely, leaving the Machine to operate autonomously, and then (optionally) come back to retrieve the final result(s) of the computation at a later time. (Or alternatively, other authorized parties may utilize its outputs.)

Besides SMC, some other notions of secure computing that relate to our security model are those of (a) homomorphic encryption [9] (§10.2), which involves computation on encrypted data without decrypting it, (b) indistinguishability obfuscation [10] (§10.3), which involves making it impossible to tell what program for computing a given function is being executed, or to predict (without running the program) what the program’s output would be for input cases that have not yet been tested, (c) functional encryption [11] (§10.4), which allows obtaining the value of a function of some encrypted data, but not the data itself, and (d) verifiable computation [12] (§10.5), in which the fact that a given computation was performed correctly by another party can be independently verified. Although these notions are not precisely the same ones that we are invoking in our particular scenario, our setup does provide certain aspects of these capabilities as well; we’ll review some relevant comparisons in Sec. 10.

The envisioned need, as outlined above, for an extremely high degree of privacy may, at first, appear to be inherently incompatible with the very public nature of the largest existing programmable blockchains, such as Ethereum. However, as we will see, these requirements may nevertheless be met using some surprisingly simple techniques, which we will present in section 5 (with further technical elaboration in §6 & §9). But first, in the next section, we’ll describe the requirements in more detail.
3. REQUIREMENTS FOR THE GABLE SYSTEM

Based on the general motivating scenario from Sec. 2, the following lists the high-level technical requirements that we wish to be met by our design for the GABLE system (see Figure 3-1):

1. For high robustness and availability, the computing machine should run on top of an existing programmable blockchain platform. For our initial prototype, we chose to target the Ethereum platform due to its being the oldest and most well-established of these systems, although a number of other programmable blockchains could have been used instead.

2. Whenever the Company wishes to deploy a particular operational function \( F \) (representable as a state machine) on the target blockchain platform, it runs a specific (finite state machine or circuit-based) representation of \( F \) through an internal subsystem called the Garbler, which translates \( F \) into the form of an encoded (“garbled”) representation \( G \) (a.k.a. the “Machine”). \( G \) is then embedded within an executable smart contract \( \text{Exec}[G] \) that is digitally signed by the Company and published on the blockchain. In our scheme, the only openly meaningful information about \( F \) that the garbled representation \( G \) of the computation is allowed to openly reveal (through its plaintext representation on the blockchain) is:
   
   a. An upper bound \( \hat{\ell} \geq \ell \) on the number \( \ell \) of steps (cycles of execution, or layers of gates in a circuit) that are supported by \( G \). (Steps are taken when encoded input symbols are provided by one or more input providers.) It is only an upper bound since the representation of \( G \) could always be padded with extra dummy cycles.

![Figure 3-1. Overall architecture of the GABLE system.](image)

The Company wishing to run the application function \( F \) deploys the garbled Machine \( G \) as an executable smart contract on a public blockchain. The Company also distributes corresponding access information (keys) to authorized input providers and spectators, which allows them to provide inputs to, and/or interpret outputs from, the deployed Machine.
b. For each potential step of the computation, an upper bound \( \tilde{q} \geq q \) on the number \( q \) of alternative conditional state transitions ("arcs") that are supported by \( G \) for that step, or, in the circuit model (§9), an upper bound \( \tilde{\omega} \geq w \) on the circuit width \( w \).

We will see later that, with a little more work, even the values \( \tilde{\ell}, \tilde{q} \) can be further obfuscated, albeit while still revealing an upper bound \( \tilde{\ell}q \geq \ell q \) on the product \( \ell q \).

3. In addition to \( G \), the Garbler also generates some corresponding access information (e.g. a set of private keys), which gets distributed via secure channels, both to selected entities (called “Sources” or “Providers”) that it wants to enable to serve as information sources, as well as to any entities (“Spectators”, “Observers”) that it wishes to enable to inspect and interpret specified outputs of the computation. So far, the conceptual design of GABLE does not yet explicitly constrain how the access information is to be distributed to the various Sources and Spectators, but in general, any available secure method could be used.

4. During each cycle of execution of the garbled machine \( G \), there are two general classes of options that we primarily consider for how the input model may work:

a. **Single-source input:** Any single Source (which could be any entity on the Internet holding the requisite access credentials) submits an encoded input symbol \( \sigma \) to the machine by posting a corresponding message to the smart contract (while paying any required transaction fee); the machine \( G \) then (if authorization succeeds) updates its internal state accordingly and (optionally) produces an intermediate output, which generally would be interpretable only by entities holding the required key(s).

b. **Multiple-source input:** In this type of input model, the machine waits to asynchronously receive some number \( N \) of inputs satisfying stated criteria—for example, this may include 1 message from each of \( N \) distinct authorizations, or, more generally, an “\( N \) out of \( M \)” criterion (with \( M \geq N \)). After the required number \( N \) of appropriate input symbols are obtained, regardless of the order in which they were received, the machine then updates its state based on these inputs (and optionally produces an output).

Other input models are possible, and we will describe a few more of them in section 5.3, but many (if not all) cases could ultimately be reduced, if needed, to just these two categories—and even the multiple-source input case could, if necessary, be implemented via multiple steps of a state machine or circuit that takes just a single-source input on each cycle.

5. While the garbled machine \( G \) is running on the blockchain, inspection of its public execution trace necessarily reveals the following information, at any given time:

a. How many encoded input messages have been provided to the machine from outside sources;

b. Which specific encoded messages triggered the machine to immediately update its state after being received. E.g., in the multiple-source input model, the machine might always be seen to update its state after receiving some number \( N \) of properly authorized messages providing values for different inputs. However, the observation that the state was updated after \( N \) messages were received in general implies only that it was

\[ \text{Also, although we do not discuss it further in this document, our scheme could be modified to generate the Machine dynamically as the computation runs, in a piecemeal fashion as a sequence of contracts, in which case there would not necessarily be any a priori discernible upper bound on its ultimate complexity.} \]
possible for that number of messages to cause the state to be updated on that cycle. (Since, depending on the input model, some machines \( G \) might be able to update themselves after receiving varying numbers of input messages.)

6. At any time, of course the Company itself may inspect the present state of \( G \) (which is publicly visible on the blockchain), and use its own private knowledge (obtained previously from its Garbler) to determine (a) the precise (decrypted) input data that was provided to the computation, the past and present states visited in the computation, (b) whether the computation has reached a halting state, and (c) any other intermediate or final outputs. The Company could also provide (in advance, or at will) appropriate keys to obtain some selected part of this information about the computation to some other selected entity (a “Spectator”), to use for some arbitrary intended purpose (e.g., some other cyber-physical system could have been deployed, somewhere out in the world, that uses the output of the computation to automatically trigger some real-world action).

3.1. Assumptions & Limitations

Here, we briefly summarize some of the key assumptions that we will need to make, in our system design, in order to be able to say that we can meet the above requirements, as well as the key limitations that will apply to our solution for meeting these requirements.

**Major Assumptions:**

1. Some entity (i.e., the Company) exists that can be trusted to create the garbled machine.
2. Some method exists to securely distribute access credentials to authorized participants.
3. The various protocol participants (e.g., input providers, spectators) will not share information or otherwise collude with each other in such a way as to compromise the privacy guarantees of the system.
4. The underlying blockchain will remain reliably accessible and will not be compromised (e.g. by a 51% attack or a network split) in a way that subverts the reliability guarantees of the system.

**Major Limitations:**

1. Application functionalities must be expressible either as a relatively simple finite state machine, or as a computational circuit operating on a moderately small number of bits.
2. Any given garbled machine instance may only be executed at most once (without potentially weakening its privacy properties).
3. Effectively, all time steps in the machine’s operation must be garbled separately (e.g., all loops must be fully unrolled) to avoid weakening the privacy properties.
4. Our available methods for implementing garbled universal circuits (§9) impose a minimum space and time complexity overhead factor that is order \( \log w \), where \( w \) is circuit width.

This concludes our discussion of the overall system-level requirements, including the high-level picture of the system architecture and the overall security model, as well as the key assumptions and limitations that are inherent to our approach. In the next section, we survey a variety of example applications that the above requirements enable us to support.
4. **EXAMPLE APPLICATIONS**

In this section, we briefly describe some more specific example applications for the GABLE system.

4.1. **Toy Problems**

The following examples are not serious applications, but they illustrate some of the most basic secure computing capabilities provided by the GABLE framework.

4.1.1. *Millionaires' Problem*

The “Millionaires’ Problem” is the original simple “toy” example of an application of secure multiparty computation that was given in the seminal paper by Yao [8]. The scenario is as follows: Two wealthy people, who we may refer to as A and B, wish to cooperate to determine whether A is wealthier than B, without sharing any information with each other, other than the answer to this question.

There is a simple 5-state Moore-type finite state machine (Figure 4-1) that provides adequate computational functionality for solving this problem, given prior mutual knowledge between the parties of some absolute upper bound $L$ on the number of bits in their possible wealth values. (For example, taking the total wealth on Earth to be roughly 1 quadrillion dollars, we know *a priori* that $L = 50$ bits ought to be more than adequate to express any given Earthling’s wealth value in dollars.)

Over each of $L$ time steps, both parties provide (in either order) consecutive bits of their wealth values to this machine, least-significant bit first. This machine keeps track of whether the $t$-bit partial numbers seen so far are equal (state $S_{\text{Eq}}$), or A’s is larger (state $S_A$), or B’s is larger (state $S_B$).

![Figure 4-1. State machine for the Millionaires’ Problem.](image)

Arens are labeled with respective bit values $ab$ from the parties AB, provided least-significant bit first over $L$ time steps. A third party provides a special final “finish” symbol $\bigoplus$ after all $L$ prior steps have completed. The size of this state machine could be further reduced to 4 or (in one scenario) even 2 states, as discussed in the text.
Finally, after all $L$ time steps have passed, some outside party (called a Finisher) provides a special input token “⊝” (taken to mean “end”) that transitions the machine to a result state, which is either $s_{\text{yes}}$ (meaning yes, $A$ is wealthier than $B$) or $s_{\text{no}}$ (meaning no, $A$ is not wealthier than $B$). Both of the result states (red) are visible (interpretable) by both $A$ and $B$, but none of the machine’s prior states (green) are visible at all to either $A$ or $B$. The GABLE system can easily provide the capability to run this machine with these security properties, assuming that there is a trusted Company that can generate and deploy the garbled machine.

Note that the distinct state $s_{\text{Eq}}$ is, strictly speaking, unnecessary for solving the given problem; it could be absorbed into state $s_B$, which would be redefined to mean that $B$’s partial wealth value is greater than or equal to $A$’s. With this done, it turns out that the extra states $s_{\text{yes}}, s_{\text{no}}$ could also be eliminated, if an outside party instead provides both $A$ and $B$, after the $L$’th time step, with access information that enables them to interpret the meaning of the Machine’s current state ($s_A$ or $s_B$). If $A$ and $B$ had this information initially, it would raise a “fairness problem,” discussed in Sec. 5.4.1.

We implemented a complete demo for the Millionaire’s Problem in GABLE; see §7.2.

### 4.1.2. “Dungeon Race” game

Suppose we are a gaming company that wishes to create competitive games with monetary payouts that can be played via smart contracts on the blockchain, so players can be confident that they will get paid if they win, without having to trust us or their opponent(s). Specifically, suppose that we wish to create an “Adventure” type game in which the players navigate a virtual dungeon-type maze, and whoever finds the hidden treasure first, wins. The nature of the desired game experience is that each player can only (virtually, through text descriptions) perceive their immediate surroundings in the maze. We wish for it to be mathematically impossible for any player (without colluding with other players) to analyze the structure of the smart contract to “look ahead” at the results of their possible actions to figure out anything at all about the dungeon, besides what they have already seen. We imagine that the company also wants the game to remain playable once started, even if the company goes defunct, so the game must operate entirely autonomously. (Recall how, in Ready Player One [13], the brilliant but eccentric James Halliday created a game that would be played after his death to determine who would inherit his fortune.)

In our case, the payout to the winner is funded by the participating players themselves. The overall protocol for the Dungeon Race game is as follows: Some number $m$ (say $m = 2$) of players who wish to compete against each other register their public keys with the company, which then generates a random dungeon layout, maps it to a corresponding state machine (or circuit, see §9), and uses a GABLE Garbler internally to generate a smart contract implementing an executable garbled machine for the game, which is then deployed on a programmable public blockchain, such as Ethereum, say. Requisite access information is distributed to the players. The players then start the game by posting a previously agreed stake to the contract. The rules are that whoever finds the treasure first receives the entire payout from all $m$ players, minus a small fee which goes to the company. The players take turns making moves. There is a “time limit” of some maximum total number $\ell$ of moves that can be made. If nobody finds the treasure before the time limit expires, then all the stakes revert back to the players.

In an explicit state-machine-based implementation, to keep the size of the state machine manageable, we could have a small number $L$ of locations (rooms) in the dungeon that the players can be located at. With $m$ players, there are $L^m$ configurations, so e.g. a 20-room dungeon with 2 players
would need 400 states. We may also expand on this slightly with a small number of additional bits of state, specifying e.g., whether a weak floor has collapsed, a found key has been picked up, a switch that opens a secret door has been flipped, etc. (With 3 extra bits, we’re now at 2400 states.)

It’s feasible to implement this kind of slightly complicated application using GABLE; although its implementation in the explicit state-machine model would be significantly facilitated by the existence of a state machine compiler like the one described in §8.

Alternatively, if a circuit model (§9) rather than a state-machine model is used, then applications like this could be made more complex, while reducing their implementation cost.

4.2. Serious Applications

The following examples are potentially realistic, more serious applications for the platform that may be of interest to enterprise customers (or that may inspire ideas for additional related applications).

4.2.1. Supply Chain Provenance Tracking

In this example, the Company wishes to track the progress of an item being produced down a supply chain, as different vendors consecutively perform their respective value-add processing steps on the item. We wish for the supply-chain tracking system to be able to continue to function autonomously even if the Company goes offline, and its history should be fully auditable. However, we assume that (apart from auditing) there is a confidentiality requirement: We do not want any outside parties to have the ability to interpret the state of the system, and even individual vendors may only be enabled to see local supply-chain status information that pertains specifically to them.

This application functionality can be easily represented in state machine form, as suggested by Figure 4-2. Each vendor could be restricted to only being able to interpret the machine states that concern them and supply input symbols representing the actions that are available to them to perform and report. Additional states and transitions may be added to the state machine design as needed to expand the flexibility of the model, for example to provide for contingencies, such as the return of the item to a prior vendor in the supply chain for repair or additional processing.

We implemented a complete demonstration application for this simple example; see §7.1.

4.2.2. Sealed-Bid Auctions

The Millionaire’s Problem that we considered in §4.1.1, although it was itself frivolous in nature, is in fact functionally equivalent to a more practical problem of selecting winning bid(s) in a simple sealed-bid auction with simultaneous bidding by semi-anonymous parties, and wherein the soundness of the procedure for selecting the winning bid(s) can be fully audited after the fact if needed. The problem can be generalized to larger numbers of bidders, although handing this cost-effectively requires using the more efficient circuit-based computation model; this application is analyzed in detail in §9.3.

4.2.3. Transactive Energy

Transactive energy refers to market-based mechanisms for managing the exchange of energy in a wide-area electrical grid [14]. Hypothetically, participants in such a market (such as utility companies) may wish to engage in automatic electronic negotiation with other participants using bots implementing custom negotiation strategies, without revealing their strategies to other participants. Additionally, it may be in the public interest to provide a level playing field for the execution of these negotiation
bots, to prevent larger participants from gaining an unfair advantage through execution of sophisticated, computationally intensive negotiation strategies. In this scenario, 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 negotiation strategies. Alternatively, a central clearinghouse could accept bots from all participants and roll them together into a single garbled computation that outputs the results of negotiations. In either case, the required protocols remain to be worked out in more detail.

### 4.2.4. Data Peering

This is similar to the previous example, but for the case of peering [15] arrangements between participants in a data network, such as autonomous systems (ASs) [16]. In cases where a market is set up for automated negotiation between ASs about peering policies, garbled bots may implement negotiation strategies on a level playing field without revealing details about their strategies to other parties, as in the transactive energy case.
5. TECHNICAL OUTLINE OF THE GABLE SYSTEM DESIGN

In this section, we outline, at a conceptual level, some key technical aspects of how GABLE works, under our present conception for how the system will be designed. More detailed, concrete encodings that we used in our prototype implementation will be described in the next section.

This section (and the next) focus on our initial version of GABLE, which is based on a simple explicit state-machine model of computation. However, the techniques developed here can also easily be extended to support a more efficient circuit model of computation, as described in section 9.

5.1. State-Machine Formalism

Our initial approach to GABLE uses the simple finite state machine (FSM) model of computation, in the sense of an explicitly represented state transition graph (with nodes representing states, and arcs representing transitions). In this subsection, we briefly present some formal notation for describing finite state machines to be garbled.

The state machine operates in discrete time steps; we index these time steps with non-negative integers \( t \in \{0, 1, \ldots, \ell - 1\} \), where \( \ell \) is the maximum total number of time steps supported by the machine. We can visualize the state machine as comprising a sequence of digital circuits, where each circuit executes a given time step, and the output from each circuit (its “resulting state”) feeds into the input of the next circuit in the sequence. Generally, it would also be possible to utilize a different circuit on each time step, but for now, we only consider machines in which the circuit, or the state update rule, is the same at each time step.

There is also a set \( S = \{s_1, \ldots, s_p\} \) of \( p \) possible states that the machine may be in at a given time, corresponding to the possible output values that each of these state-update circuits passes to the next one in the sequence. The full set of states may or may not be reachable at each time step.

For now, we assume that the state-updating circuitry, including the set of input variables (input lines) that feed into it, is the same at each time step \( t \in \{0, 1, \ldots, \ell - 1\} \). However, the actual values \( v_i(t) \in V_i \) provided for the input variables may in general differ on different time steps. In addition to the externally provided input lines, there are also state lines \( S_t \) (state variables) passing the current state information from each copy of the circuit to the next. The copy of the circuit for time step \( t \) takes the value on state line \( S_{t-1} \) as input and produces the value on state line \( S_t \) as output. The initial state line \( S_{-1} \) could either be a fixed constant \( s_{\text{init}} \), or it may be supplied externally at the time of machine initialization.

![Figure 5-1. Overall picture of a state machine operating for up to \( \ell \) time steps.](image)

For now, we assume that the state-updating circuitry, including the set of input variables (input lines) that feed into it, is the same at each time step \( t \in \{0, 1, \ldots, \ell - 1\} \). However, the actual values \( v_i(t) \in V_i \) provided for the input variables may in general differ on different time steps. In addition to the externally provided input lines, there are also state lines \( S_t \) (state variables) passing the current state information from each copy of the circuit to the next. The copy of the circuit for time step \( t \) takes the value on state line \( S_{t-1} \) as input and produces the value on state line \( S_t \) as output. The initial state line \( S_{-1} \) could either be a fixed constant \( s_{\text{init}} \), or it may be supplied externally at the time of machine initialization.
For each time step \( t \in \{0, 1, ..., \ell - 1\} \), we can define the output state line \( S_t \) as a state variable representing the state information that is output from the circuit carrying out step \( t \) of the computation. For times \( t < \ell - 1 \), this state line feeds into the circuit for time step \( t + 1 \); for time \( t = \ell - 1 \), this state line \( S_{\ell-1} = S_{\text{fin}} \) can be considered the final output of the entire machine. See Figure 5-1. We use \( s(t) \in S \) to refer to the actual machine state resulting from time step \( t \) on a particular run.

Further, we can imagine there is an input state line \( S_{-1} \) representing the initial state of the machine, which feeds into the first stage of the circuit. Without loss of generality, we can assume that this variable always takes on the value \( S_{-1} = s_{\text{init}} = s_1 \); i.e., there is some fixed initial state \( s_{\text{init}} = s_1 \in S \). However, in some applications, it may be desirable to allow \( s_{\text{init}} \) to be provided dynamically, at machine initialization time.

Next, assume there is a set \( V = \{V_i\} \) (for \( i \in \{1, ..., m\} \)) of \( m \) different input variables \( V_i \) (or input lines) that feed in to the state machine (from its external environment) on each time step. Again, although in general, if the circuits at each time step were different, so too could the set \( V \) of input variables at each time step be different, but currently we are not considering that case, and so we assume that the set \( V \) of input variables is always the same. For each of these input variables \( V_i \), there is a corresponding set \( V_i = \{V_i^1, ..., V_i^m\} \) (note here the superscript represents an additional index, rather than exponentiation). The (assumed unique) value assigned to input variable \( V_i \) on time step \( t \) can be denoted \( v_i(t) \). The input line delivering the value of input variable \( V_i \) for time step \( t \) to the machine can be denoted \( V_i(t) \).

The state machine can now be defined by a set \( A = \{a_0, ..., a_{q-1}\} \) of \( q \) possible (directed) arcs (a.k.a. arrows, directed edges) specifying how each state may transition to the next. Each arc \( a_k \) (for \( k \in \{0, ..., q - 1\} \)) can be described as an ordered triple \( a_k = (O_k, C_k, D_k) \), where \( O_k \in S \) is the arc's origin state, \( D_k \) is the arc's destination state, and \( C_k \) describes the arc's set of transition conditions, which can be described as a set \( C_k = \{c_g\} \) of condition objects \( c_g \), where the size (cardinality) of \( C_k \) always satisfies \( 1 \leq |C_k| \leq |V| \) (i.e., in any given condition set \( C_k \), there is always at least one condition, and there are no more conditions than there are state variables).

Each specific condition \( c = c_g \) can be described as an ordered pair \( c_g = (i_g, j_g) \) of indices, where \( i_g = i \in \{1, ..., m\} \) is the index of some input variable \( V_i \in V \), and \( j_g = j \in \{1, ..., n_i\} \) indexes a possible value \( V_i^j \) of that variable. The meaning of the condition \( c \) is that input variable \( V_i \) must take the value \( V_i^j \) in order for this arrow to be followed; i.e., it represents a requirement that \( V_i = V_i^j \). If all the conditions \( c_g \in C_k \) in the entire condition set are met, then the arrow will be followed, and the resulting state will be \( D_k \); otherwise this particular arrow will not apply. See Figure 5-2.

We generally assume that for each arc, its condition set \( C_k \) is internally consistent, meaning that no variable index \( i \) appears in two different conditions \( c_{g_1}, c_{g_2} \in C_k \) (i.e., with different value indices \( j_{g_1} \neq j_{g_2} \); since these two conditions would then not normally be considered to be simultaneously satisfiable, and the arc would be useless to include).\(^2\) We may also want to require that each condition set \( C_k \) be complete, meaning that \( |C_k| = |V| \); i.e., the condition set includes a condition for every possible value of each input variable.

\(^2\) Interestingly, however, it would actually be possible in our scheme to allow multiple different input values to be fed into a given input variable on a single time step and have this trigger associated arcs in a meaningful way, depending on the input model. However, we will not pursue that option in detail in this document.
variable. (However, as we will show later, it’s still possible to utilize a given condition set under some state-machine execution methods, even if this is not the case.)

Further, we may optionally want to require that the machine be deterministic, meaning that no two possible complete assignments of values to variables could ever satisfy the condition sets \( C_{k_1}, C_{k_2} \) of two different arcs \( a_{k_1}, a_{k_2} \) having the same origin state \( O_{k_1} = O_{k_2} \) but different destination states \( D_{k_1} \neq D_{k_2} \), since then the resulting state could not be uniquely determined from the set of satisfied transition conditions.

In addition, we may also want to require that, for any given state \( s \in S \), the set of all condition sets on arcs outgoing from \( s \), which we may refer to in aggregate as the out-conditions of \( s \), and denote

\[
OC(s) = \{ C \mid \exists a = (O,C,D) \in A: O = s \}.
\]

must be a complete set of condition sets, which are each themselves individually complete as well, so that every possible assignment of values to input variables appears as the condition set for some arc outgoing from \( s \). In this case, the total number of outgoing arcs from each state, in a deterministic machine, would be given by

\[
|OC(s)| = \prod_{i=1}^{m} n_i,
\]

that is, the product, over all the input variables, of their numbers of alternate values. So, for example, given that there are \( m \) different input variables, if each of them has 2 possible values, then this would imply that there must be exactly \( 2^m \) different arcs outgoing from each state, one for each possible assignment of values to all of the variables. However, incomplete out-conditions are not always a fatal problem; for example, in some applications, the condition sets that are not explicitly included in \( OC(s) \) may be ones that ought to never arise. (As a contingency, the semantics of the machine operation can be defined in such a way that it halts, or remains in the same state, or transitions automatically to some explicitly-specified exception state, in case none of the current state’s out-conditions apply at the time that the next step is taken.)

In the example state machine in Figure 5-3, states \( s_{\text{Init}} \) and \( s_{\text{Reset}} \) have complete out-conditions, while states \( s_{\text{Pass}} \) and \( s_{\text{Fail}} \) have no out-conditions—this can serve as one way of representing that they are final (or halting) states.

Figure 5-2. An arc in a state machine.

Each arc or arrow \( a_k \) in the state machine can be described as an ordered triple \( a_k = (O_k, C_k, D_k) \), where \( O_k, D_k \in S \) are states called the origin state and the destination state of the arc, together with a set \( C_k = \{ c_1, c_2, \ldots \} \) of up to \( |V| \) transition conditions, each of which specifies that some particular input variable \( V \) must be assigned to some given input value \( v \), that is, \( V = v \).
For some applications, it may also be desirable to allow some of the input variables to optionally be left unassigned on a given cycle of operation (e.g., if no value has been provided by a given input source yet, by the time at which the step will be executed). This may be accomplished by including a special value ⊥_𝑖𝑖 meaning unassigned in the value set 𝑣𝑣_𝑖𝑖 for input variable 𝑉𝑉_𝑖𝑖, and utilizing that value by default on time steps when no other value has been provided for 𝑉𝑉_𝑖𝑖. Alternatively, if arcs are allowed to have incomplete condition sets, it may sometimes be possible to determine that a given arc’s traversal conditions have been met even if not all of the input variables in the full set 𝑉𝑉 have been assigned.

We could include additional output lines in the above model, but for the time being, we will refrain from doing so, to improve the simplicity of the model. This is not a fatal restriction, however, since, if an additional output is needed for any of the time steps, in principle it could always just be included in the resulting state. Later, in sec. 5.3, we will see some ways to do this.

5.2. Garbled State Machine Encoding

In this subsection, we describe a particular method for encoding or garbling an arbitrary state machine, as defined above, in such a way that the requirements described in section 3 may be met (apart from a few caveats addressed in sec. 5.4). That is, the machine can be deployed on a programmable blockchain, and executed publicly on that blockchain, yet without publicly revealing any information about the input values that were provided by the data sources on each cycle, or about the machine’s function or output (either its resulting state, or any other output) on each clock cycle.

The specific garbling technique that we will describe below depends on the ready availability (within the target programmable blockchain’s smart-contract language) of some cryptographic hash function, denoted 𝑡𝑡(·). Fortunately, a number of existing programmable blockchain systems (including the popular Ethereum system) do already include hashes as built-in primitive functions; specifically, both the 256-bit SHA-2 hash function, and also a hash function in the Keccak family (a family of hash functions that also includes SHA-3) are available primitives on the Ethereum platform.
This technique also depends on the Garbler’s ability (within the Company’s security domain) to produce cryptographic keys consisting of (securely generated and cryptographically strong) random bit strings (which will be the same length as the output of the hash function \( h \)) and distribute them securely to the various input sources and spectators that may participate in the computation.

Specifically, at some point prior to the start of the computation, for each possible time step index \( t \in \{0, 1, \ldots, \ell - 1\} \), where \( \ell \) is the maximum length of the garbled computation in time steps that we wish to support (as the Garbler), we do the following: For each possible value \( \nu_i^j \in V_i \) of each input variable \( V_i \), we generate an \( n \)-bit random key, denoted \( K(\nu_i^j, t) \), where \( n \) is the number of bits in the output of \( h \). We may then (securely) distribute this key to whichever entity or entities upon which we wish to confer the specific capability to provide the value \( \nu_i^j \) for input variable \( V_i \) to the Machine on time step \( t \).

An optimization of this key-distribution procedure which requires much less overhead to send and store the input keys, which applies when the input variables \( V_i \) are associated with corresponding input providers, is to generate and distribute a single random \( n \)-bit participant key \( K_i \) to that provider, and then we can let each input key \( K(\nu_i^j, t) \) be generated deterministically from the participant key \( K_i \) and the parameters \( j, t \) using the hash function \( h(\cdot) \), e.g. as

\[
K(\nu_i^j, t) = h(h(K_i + j) + t),
\]

(3)

(where “+” here could be byte-string concatenation, or any non-information-losing function) so that each input key will be unpredictable, and the value \( \nu_i^j \) that it represents will be indecipherable, to any entity not possessing the associated participant key \( K_i \).

In addition to these input keys, for each time step \( t \in \{-1, 0, 1, \ldots, \ell - 1\} \), and for each possible state \( s \in S \), the garbler also generates an \( n \)-bit random key, denoted \( K(s, t) \), which will constitute our garbled representation of the assertion \( S_t = s \), i.e., denoting that the resulting state of the machine output from time step \( t \) is state \( s \). (Except, recall, \( S_{-1} \) has a special meaning; it is the initial state.) Throughout any given time step \( t \) of the Machine’s execution, the garbled representation of the machine’s state resulting from the previous time step, \( K_{\text{cur}} = K(s(t - 1), t - 1) \), is publicly viewable.

Then, the protocol for providing the input value \( \nu_i^j \) for input variable \( V_i \) as an input to the machine on time step \( t \in \{0, \ldots, \ell - 1\} \) will be as follows:

1. Assume, to start, that there is some entity (input source) that possesses the key \( K(\nu_i^j, t) \) (however it was generated and distributed). It is possible that this same key could be held by multiple parties, in which case any of them could do the following.
2. The entity should wait until the Machine has begun gathering inputs for time step \#\( t \). This can be determined by inspecting a publicly readable current time step index maintained by the Machine, or (if such is not provided) by counting state changes. (However, the latter approach requires more effort to avoid possible concurrency issues.)
3. The entity reads the publicly viewable current-state key \( K_{\text{cur}} \) from the Machine.
4. The entity privately (within its own security domain) computes a provision key $p_{t_i}^{i_j}$ specifically for the purpose of providing the data value $v_i^j$ to update the machine in step $t$ given that it was in state $s(t - 1)$ at the start of the time step; this key is computed as follows:

$$p_{t_i}^{i_j} = h(K_{\text{cur}} \oplus K(v_i^j, t)).$$  \hspace{1cm} (4)

This transformation is necessary to prevent certain types of replay attacks; see §5.4.3.

5. (Optional) If the deployed Machine instance requires/allows PKI-based authentication of input providers, then the entity should/may digitally sign this provision key using a private key held by the provider, where the corresponding public key is known to the Machine as identifying an entity authorized to supply keys for the present input variable and time step.\(^3\) Note that this signature effectively comprises an optional second-factor form of authentication, since it is not strictly necessary if keys are only distributed to authorized parties in the first place, and if key management is sound. However, requiring this additional step opens the possibility that individual entities’ PKI certificates could be updated or revoked sometime after the original set of keys for the garbled machine are distributed.

6. The entity then transmits the (signed or unsigned) provision key to the Machine by invoking a suitable method in the smart contract’s API. For example, in the prototype implementation discussed in §6 and Appendices C & D, the provideInput() method is called.

So long as the above procedure is followed, and an appropriate PKI signature is provided if required, and the message is received at a time when the Machine is indeed still waiting to receive a value (or values) for input variable $V_i$ for step $t$ of its execution, then this event (carrying out this procedure) constitutes this input source providing the input value $v_i^j$ on input line $V_i$ for time step $t$.

With the above model of input provision in place, the state machine itself can now be encoded by producing a garbled representation of its set $\mathcal{A} \in \{a_0, \ldots, a_{q-1}\}$ of arcs, as follows. This encoding is designed in such a way that it will not publicly reveal any information whatsoever about the meanings of the input values provided, or of the resulting output state. At most, general viewers of the computation can only see that some encoded inputs were provided by some input sources, and that the machine arrived at some encoded new state as a result. Since this same behavior applies on every time step, it provides no meaningful information about the progress of the computation, other than the number of steps that have been executed.

For each time step $t \in \{0, \ldots, \ell - 1\}$, we will produce an independent garbled representation of the state-updating circuit that applies at time $t$, even when our assumption holds that the actual circuit for each time step is the same. This prevents adversaries from correlating patterns of execution of the machine across multiple time steps.

First, each individual arc $a = (O, C, D) \in \mathcal{A}$ will be encoded as follows (see Figure 5-4). Let $\mathcal{C} = \{c_1, c_2, \ldots, c_r\}$ be the arc’s set of transition conditions, with $r = \vert \mathcal{C} \vert$. First, we assemble an arc identifier

\[^3\] In Ethereum, as in typical blockchains, all message transactions are signed by default using the public key associated with the sender’s account. In Solidity, authentication only takes checking to see if msg.sender is on a given whitelist.
\[ I = I_{a} = K(O, t - 1) + \sum_{g=1}^{r} p^{i_{g}j_{g}}, \]  

where

Figure 5-4. Overall illustration of arc-garbling algorithm.

The Garbler associates cryptographically strong random keys to all the states and input assignments contributing to given arc on a given time step. The origin state and input provision keys are combined by XOR (or other Combinator) and then hashed to XOR-mask the arc’s destination state key.
\[ p^{i_{p/g}}_t = h \left( K(O, t-1) \oplus K^{i_g}_t \right) \] (6)

is the corresponding input provision key for providing input \( v^{i_g}_t \) on time step \( t \) when the current state is the origin state \( O \), and where addition (and summation) in eq. 5 stand for some pre-selected associative, commutative, information-preserving method of combining bit strings, which we will call the Combinator.

The Combinator could be implemented in several possible ways. E.g., the bit strings could first be sorted lexicographically, or placed into a standard order based on the variable indices \( i_g \), in cases when these are available, and then simply concatenated together. Alternatively, they could be interpreted as integers and then added together numerically (or XOR'd); although this technically loses some information (in the rare case when two different subsets of keys happen to have the same sum/XOR), it will only yield collisions with an extremely small probability. The important thing is that \( I \) should utilize enough of the random information contained in the full set of keys that was used to construct it so that it has an extremely low probability of being conflated with an arc identifier constructed without knowing all of that data.

Now, using this arc identifier \( I \), we will produce two encrypted data entries \( E_{\text{next}} \) and \( E_{\text{valid}} \), which will be keyed off of respective entry identifiers \( I_{\text{next}} \) and \( I_{\text{valid}} \). Each of these entry identifiers is derived from the arc identifier \( I \) in some fairly arbitrary (but non-information-losing) fashion, for example, by combining an extra field identifier (e.g., the ASCII codes for ‘n’ or ‘v’, or some alternative, more obfuscatory constants) with the arc identifier \( I \).

The encryption method for both data entries will then be the same; given any entry identifier \( I_e \) (where \( e \) specifies either the ‘next’ or ‘valid’ data entry of a specific arc \( a \)), and any \( n \)-bit “plaintext” value \( x \) to be encrypted, the corresponding ciphertext \( y \) to be used for the data entry is given by the following encoding:

\[ y = \text{enc}[I_e, x] = h(I_e) \oplus x, \] (7)

where \( h(\cdot) \) is our cryptographic hash function (which produces an \( n \)-bit hash) and \( \oplus \) represents a bitwise exclusive-OR operation. Note that effectively, this is a “one-time pad” [17] encryption of \( x \), where the random bit string used for the pad is obtained by hashing the entry identifier \( I_e \)—assuming here that the given identifier \( I_e \) will not be used more than once. Decryption of \( y \) to obtain \( x \) can only be accomplished if that exact entry identifier \( I_e \) can be obtained, which is only possible for an entity that possesses all the keys which go into the calculation of the arc identifier \( I \).

Now, this encryption function \( \text{enc}[\cdot, \cdot] \) (eq. 7) is then used to compute both encrypted data entries for the arc:

\[ E_{\text{next}} = \text{enc}[I_{\text{next}}, K(D, t)], \] (8)
\[ E_{\text{valid}} = \text{enc}[I_{\text{valid}}, v], \] (9)

where \( D \), recall, denotes this arc’s destination state, \( K(D, t) \) is that state’s garbled encoding (a.k.a. its “key”) at the present time step, and \( v \) represents some arbitrary fixed (or at least, recognizable) \( n \)-bit
“magic cookie” string. For example, $v = 0^n$ (a string simply consisting of $n$ zero bits) can be used, or we could use a (padded) ASCII representation of the English word “valid.” Alternatively, for slightly more obscurity, we could instead use a variable, arc-dependent value, such as $v = h(I)$.

The encoding $E(a, t)$ of the entire arc $a$ for time step $t$ is then simply the pair of encoded data entries:

$$E(a, t) = (E_{\text{next}}, E_{\text{valid}}). \quad (10)$$

Alternatively, the encrypted pair of entries could also be produced in a single step by simply XOR-masking both entries $K(D, t), v$ at once with a suitably longer hash of $I_a$. (Shown in Figure 5-4.)

To encode the entire set $A = \{a_0, ..., a_{q-1}\}$ of arcs for the time step $t$, we then simply provide the full set $E(t) = \{E(a_0, t), ..., E(a_{q-1}, t)\}$ of the corresponding encoded arcs, but expressed in a randomized order, to ensure that no meaningful information about the machine’s structure remains implicit in the arc order.

The complete description of the garbled finite state machine $G$ over all $\ell$ of the possible time steps $t \in \{0, 1, ..., \ell - 1\}$ then consists of the sequence $G = (E(0), E(1), ..., E(\ell - 1))$ of the garbled descriptions of the individual time steps, in temporal order. It is possible to randomize the order of the time steps as well, which we’ll discuss in more detail in the next subsection (§5.3) below, but this is not strictly necessary to meet our security requirements.

### 5.3. Garbled State Machine Execution

Executing a garbled finite state machine $G$ which has been assembled via the above construction, in a way that meets our privacy requirements (except, see §5.4), is now extremely simple. The entire garbled machine $G$ is contained within some smart contract/executable program, which also includes a generic garbled machine interpreter $\text{Exec}$ (called an $\text{Executor}$). This smart contract, which we denote $\text{Exec}[G]$, is then (at some point before its execution will need to start) published on a programmable blockchain (such as Ethereum’s).

First, we assume that the encoded form $K(s_{\text{init}}, -1)$ of the initial state $s_{\text{init}}$ of the machine before the start of time step 0 is already available, as part of $\text{Exec}[G]$. Alternatively, it can be provided, in a separate, special initialization step, by an entity called a $\text{Starter}$ that has been authorized to start $G$ running in some particular state $s \in S$. (Different entities could be authorized to start the machine up in different initial states.)

Then, to execute each of the normal (i.e., post-initialization) time steps $t \in \{0, 1, ..., \ell - 1\}$, in numerical order, we can carry out the following protocol.

For each time step $t$, during the input-gathering phase of that time step, some number of external entities (input sources) holding input keys $K(v_i^j, t)$ for some of the possible values $v_i^j \in V_i$ of some of the possible input variables $V_i \in V$ for that time step asynchronously transmit (possibly digitally signed) messages to the $\text{Exec}[G]$ smart contract, containing corresponding provision keys $p_i^{j,t}$ for the input values $v_i^j$ that they wish to supply. The Executor can ignore such messages if they were not signed by the private key holder for some appropriately authorized public key (the set of which can be hard-coded into the contract). Alternatively, we might not require a signature.
Depending on the input model being used, we can then use one of the following methods (among others) to decide when to go to the next, state-updating phase of the Executor:

1. **Single-shot update.** This method only works for machines all of whose arcs’ transition condition sets contain only a single condition (that is, a single variable=value assignment). In this method, as soon as we receive any single (properly-signed) input value, we look for an arc that matches it (as per the arc-matching procedure below), and if it is found, we go to the state-updating phase; otherwise, we discard the input value and either halt permanently, or continue waiting for valid inputs (the Executor’s behavior could be defined either way). Additional specializations of single-shot updating are possible; for example, we could require input providers to provide inputs in round-robin order or at least forbid any single input provider from providing inputs two steps in a row, to help prevent lookahead.

2. **Gather all inputs.** This method only works reliably when all the input providers will, together, during each time step, reliably provide all the input values that are needed to satisfy the condition set for some arc. In this method, we continue gathering encoded input values, and adding them, as they are received (but in an order-insensitive way), to a candidate arc identifier $i'$. As soon as this candidate arc identifier matches any of the encoded arcs of $G$, according to the arc-matching procedure below, we go to the state-updating phase. With this method, if some of the input providers decline (or are unable) to cooperate and fail to provide some needed input values as expected, it would be possible for the Executor to hang indefinitely. However, this potential weakness can be mitigated via a number of methods, discussed below.

3. **Gather all inputs, with timeout.** This method is like gather all inputs above, except that the smart contract can be programmed to halt (terminating with no hope of recovery) if one of the arcs is not matched within some specified timeout. Similarly, a timeout can also be applied to the single-shot update method. (Note that, depending on the smart contract platform, implementing this feature may require trusted external parties to supply time messages.)

4. **Gather all inputs, with timeout and infill.** This method requires that $\text{Exec}[G]$ publicly shows the number $m$ of input variables, and that input providers publicly reveal which variables they are providing values for. This therefore violates our security requirements from Sec. 3, but can be acceptable in contexts with looser requirements. In this method, we keep track (publicly) within the Executor of which variable indices $i \in \{1, \ldots, m\}$ have had values provided for them so far, and if no value is provided for some variables before the timeout, we assign a special value $\perp_i$ (meaning “unassigned”), whose encoding is shown publicly within the Executor code, to each of the as-yet-unassigned variables $V_i$. We then look for an arc matching the arc identifier including the infilled values. More complicated variations on this method could hide a larger amount of this information up front, by requiring some external parties (such as other input sources) to supply these default values for the missing variables.

5. **Gather N out of M inputs.** In this method, which is somewhat similar to the previous one, as soon as a prespecified number $N$ of distinct input values has been provided, where $N$ is less than the total number of variables, $N < M = m$, we can either fill in the remaining variables with default “unassigned” values (as above), or just try to find an arc that matches just the subset of input values that have been received so far.

6. **Check all subsets.** In this method, we maintain a set $K$ of all the input provision keys $p^{ij}_t$ that have been received so far. When receiving each new provision key $p_t = p^{ij}_t$, we first try combining it with each possible subset $k \subseteq K$ of the already-received keys, looking for a matching
Arc. If we find a match with an encrypted arc, we go to the state-updating phase. If we are unsuccessful, then we update the set $K$ by adding $p$ into it, $K := K \cup \{p\}$, and continue waiting to receive more input values. This way, if there is any arc whose condition set matches any subset of the input values received so far, that arc will be taken, as soon as possible.

Arc-matching procedure. Regardless of which method is used in the input-gathering phase, a candidate arc identifier $I'$ is constructed, as in eq. 5 above, from the encoded input state $K(S_{t-1})$, in combination with some subset $k \subseteq K$ of the input provision keys $p^k_{t\downarrow}$ that have been received so far. (The choice of input-gathering method only determines which of the constructible candidate arc identifiers are considered for matching against the encoded arcs for the given time step $t$)

To determine whether a given arc identifier $I'$ matches any of the encoded arcs $E \in E(t)$ in the set $E(t)$ of encrypted arcs for the current time step, we conceptually attempt the following for each of the encoded arcs $E$, broken down as $E = (E_{\text{next}}, E_{\text{valid}})$.

First, we derive a candidate entry identifier $I'_{\text{valid}}$ for the ‘valid’ field from the arc identifier $I'$ in the usual way (e.g., combining a byte encoding ASCII(‘v’) with it). Then, using that as our decryption key, we call $\text{enc}[I'_{\text{valid}}, E_{\text{valid}}]$, which does a trial decryption of $E_{\text{valid}}$ (this same $\text{enc}$ function works for decryption as well as encryption if the entry identifier is correct, because $I_e \oplus I_e = 0$ for all $I_e$). If the result is the designated special constant ‘valid’ identifier string v (= 00, say), then the decryption worked, which means (with probability $1 - 2^{-n}$) that the arc identifier $I'$ matches that of the specific encoded arc $E = (E_{\text{next}}, E_{\text{valid}})$ that we are currently looking at.

At this point, we can then validly decrypt the encrypted ‘next state’ field $E_{\text{next}}$ by generating the appropriate entry identifier $I'_{\text{next}}$ for it, e.g., with $I'_{\text{next}} = I' + \text{ASCII(’n’)}$, and then use this $I'_{\text{next}}$ as our decryption key and call $\text{enc}[I'_{\text{next}}, E_{\text{next}}]$; the result of this is then the “plaintext” (albeit still garbled) version of the encoded destination state, namely $K(D, t)$. (Note that this still does not reveal any information about $D$ itself.) At this point in time (meaning, any time after the most recent input was supplied), we can say that this particular encrypted arc $E$, including its garbled destination state $K(D, t)$, have been unlocked, since their unencrypted (albeit still garbled) representations can now be inferred by anyone inspecting the blockchain, by following the above procedure.

If, in a specific input situation, two different encrypted arcs $E_1, E_2 \in E(t)$ (or more) can be matched against available candidate arc identifier(s), and these two arcs yield multiple different garbled destination states, e.g., two different values $K(D_1, t) \neq K(D_2, t)$, then the machine is nondeterministic, and the semantics can either be that the machine halts (treating this as an error), or alternatively, that it selects one of the indicated destination states at random (which would require further defining a method for assigning probabilities to the different destinations and then pseudo-randomly selecting a particular outcome). For now, we will set aside this possibility of nondeterministic/stochastic machine behavior and assume that all the matching arcs in any given (non-error) input situation consistently identify just a single encrypted destination state $K(D, t)$.

---

4 The average time to successfully match an arc that is present in $E(t)$ can easily be reduced from $\Theta(q)$ to $\Theta(1)$ by storing arcs in a hash table, indexed by hashing the arc identifier. Similarly, Bloom filters [1] can be utilized to reduce the expected time cost for failed arc lookups to a negligible level. (The more standard “point-and-permute” method for fast garbled table lookups, also mentioned in footnote 5 on p. 74, cannot be applied here without protocol changes, since our input keys are hashed, and further, even with protocol changes, the number of point bits utilized would reveal an upper bound on the number of alternative values for each input field.)
Now, in the state-updating phase, we simply take the garbled destination state $K(D, t)$ of the arc that was matched, and use it as the garbled resulting state $K(s(t))$ that is produced (output) on line $S_t$ in time step $t$ of the computation. At this point, we can then increment the time step counter, $t := t + 1$, and repeat the above procedure to execute the encrypted circuit $E(t)$ for the new value of the time step $t$.

If we wish to obfuscate the fact that there even exists a time-step variable $t$ in the machine whose value is progressively increasing, and/or the maximum number $\ell$ of time steps supported within $G$, or the complexity of the garbled circuits for individual time steps (in cases where they differ), we can do this by (for example) simply combining all the encrypted-arc sets $E(0), E(1), \ldots, E(\ell - 1)$ for the different steps of the computation together into one large encrypted-arc set $E$:

$$E = \bigcup_{t=0}^{\ell-1} E(t),$$

which is then stored as part of the smart contract $\text{Exec}[G]$ in a completely randomized order. This same set $E$ of encrypted arcs can then be used on every execution cycle, and then no explicit time-step identifier is needed at all. This works because the fact that the states and input values were garbled differently for each time step ensures that on any step, we can only match the arcs in $E$ that came from the set $E(t)$ for the current time step.

Naively, one might expect that combining arc sets in this way would make the execution of each step $\ell \times$ more time-consuming, since we have $\ell$ times as many arcs to try to match. However, arcs can easily be stored in a hash table configuration keyed off the arc identifier (as was noted in footnote 4 on p. 45), and if this is done, then there is no loss in performance from combining arc sets.

An advantage of combining arc sets is that it hides some information about the number $\ell$ of time steps, as well as the number $q = |A|$ of arcs in the state machine, revealing only an upper bound $\ell q$ on their product $\ell q$ (namely, the total number of encrypted arcs in $E$). (It’s only an upper bound since the set $E$ of arcs could be padded with extra dummy arcs that will never match anything.)

In any event, the only necessarily publicly visible information resulting from the execution of each time step is then the garbled value $K(s, t)$ of the current state $s$ (and of states previously visited). This code will be meaningless to outside observers, while having perfectly clear implications to any entity upon which the Company (which originally generated all the state keys, in its Garbler) has conferred the ability to (partially or fully) decrypt that encrypted state, using any of several methods which we will now discuss. Using these methods, the Company can easily provide different entities with the ability to infer different specific output information from the machine’s public behavior.

More specifically, consider any meaningful logical proposition $P(s, t)$, which may be any arbitrary truth-valued function of the machine state $s \in S$ and the time-step index $t \in \{0, \ldots, \ell - 1\}$. This proposition is then satisfied in some corresponding subset $P \subseteq S \times \{0, \ldots, \ell - 1\}$ of the full set $S \times \{0, \ldots, \ell - 1\}$ of all ordered pairs $(s, t)$ consisting of a possible machine state together with a time-step index. The set $P$ can then be rendered in a somewhat privacy-preserving way by replacing each of the ordered pairs $(s, t)$ with a hash of its encoded form, $h(K(s, t))$, and then randomizing the order of the set elements; let $H$ denote the resulting set of all hashed, encoded elements of $P$. Then, if we wish to confer upon some Spectator the ability to recognize when the machine’s state after some

$$\text{46}$$
time step has come to satisfy the proposition $P(s, t)$, we can simply deliver to them the set $H$, and they can simply check, whenever the machine arrives at a new garbled state $K(s(t))$, whether $h(K(s(t))) \in H$. If so, then the proposition $P(s(t), t)$ holds for the current machine state $s(t)$.

This method can be straightforwardly (if inefficiently) extended to also confer the ability to infer any given arbitrary function $f(s, t)$ of the current state and current time upon any spectator, through binary representation of the function’s output, and rendering of each bit of the result as a proposition, in the above manner.

Much more efficient output encodings are also possible; for example, the spectator can combine the current encrypted state $K(s(t))$ together with a “reader key” $K_R$ identifying a reading authority $R$ and then hashed, then the result can be XOR’d together with each of a set $M = \{m_b\}$ of coded output messages $m_b$ which are distributed to the spectator and which optionally (for nonrepudiability) can be made publicly available as part of a digitally-signed package (e.g., in the contract itself). The coded message(s) $m_b$ that correspond(s) to the current state can then yield plaintext data with any desired content:

$$h(K(s(t)) + K_R) \oplus m_b = \text{(Plaintext output from } S_t \text{ viewable by reader } R)$$  \hspace{1cm} (12)

The correct message(s) $m_b$ can be identified by, e.g., inclusion of a magic cookie value, or by a lack of nonprintable ASCII characters. The output messages can furthermore be any desired length, if an arbitrary-length hash function is used here, or if more than one coded message corresponds to the current state. Note that malleability of these coded output messages is not a concern as long as they are distributed via an authenticated channel (for example, by including them in the digitally signed smart contract).

Thus, arbitrary Moore-type finite state machines (in which the output depends only on the current state), with specific output lines feeding to specific Spectators, can be constructed via methods like the above.

For a Mealy-type finite state machine, in which output information is associated with transitions, so that not all of the machine’s output information needs to be implied by the current state (which can be useful, since it can allow the number of states to be made smaller), we would need to modify our machine encoding to represent the additional outputs for each arc separately. Doing this is very straightforward but will be addressed in detail at a later time.

5.4. Risks and Vulnerabilities

An account of some known or potential risks, vulnerabilities, and weaknesses associated with the above system follows, along with a discussion of some known or potential strategies for ameliorating or mitigating these concerns.

5.4.1. Lookahead (a.k.a. Fairness) Problem

One clear flaw in the protocol, as presently envisioned, is that the input provider who provides the last input needed to trigger the execution of a given time step can inspect the state of the smart contract to “look ahead” and see what the result would be (to the extent that it can be interpreted by them), and accordingly withhold or modify their provided input. This problem arises quite generally in all MPC protocols, whenever the parties to the protocol might not be considered completely trustworthy.
Since it confers what might be considered an unfair advantage to the last participant in any such protocol, it is referred to as the “fairness problem” in the MPC literature.

**Amelioration/mitigation strategies:** One approach towards ameliorating this risk would be to have a trusted external input provider who provides a coded “proceed” symbol ⊙ (similar to the “end” symbol ⊝ that we used in §4.1.1) that “clocks in” the previously-received inputs from the other providers, and yields the transition details. For robustness, several different entities can redundantly all have this capability delegated to them. A remaining vulnerability is that, if these “clocking” providers or Steppers are not, in fact, completely trustworthy, they might choose to change their behavior depending on the outcome that would be revealed as well. On the other hand, if these third-party participants are completely trusted, then why is the entire system needed at all? One could just let a fully trusted third-party participant execute the entire desired computation internally to themselves.

We do not yet know a general, completely satisfactory solution to this dilemma. It’s an example of the general Byzantine fault tolerance problem, but the usual blockchain-based solutions to that problem don’t preserve privacy.

In fact, for a computation such as ours that is taking place on a public blockchain, it seems that the only way to avoid the lookahead problem in general, without using fully-trusted third parties, is to simply enforce the following constraint: For each input provider, for each state-updating event which can be triggered by that provider, involving an arc \( A \) and destination state \( S \), we explicitly preclude that specific input provider from obtaining any interpretable information whatsoever from \( A \) or \( S \). This can be done by simply not giving that provider any of the output keys that would be needed to spectate on those outputs. Note that this implies that the general public also cannot spectate on those outputs. However, it’s always possible to have some outputs which are public, others which can only be seen by input providers, and others which can only be seen by designated spectators other than the most recent input provider.

Note that it’s still possible, under this constraint, to have extended computations with multiple input providers in which at least some of the outputs that are generated as the computation progresses are (eventually) readable by all providers: For each provider, you can simply delay the readability of that output information from the viewpoint of that provider until after one of the other providers has subsequently made a move (supplied an input). As long as you don’t have a long string of consecutive moves made by a single provider, that provider will be able to see some output from his moves in a fairly short period of time (as long as all the other providers have not stopped providing inputs in the meantime). In any case, this seems to be the only general class of solutions not requiring any fully trusted providers. Note that it can, however, be defeated if the providers collude with each other to look ahead, but there’s no way to prevent that, anyway—the best we can ever hope for, in a public Machine, if the providers are not completely controlled by the Company, is only to prevent lookahead if the providers don’t collude with each other.

The only remaining limitation of the above approach would be if we have a scenario where, say, the very last step in the computation is required to reveal, publicly, some information that all of the input providers have some disincentive to publicly reveal, for some reason. In this case, it becomes necessary to still invoke a third party which is trusted to “pull the lever,” metaphorically speaking, as would be needed to publicly reveal the answer. One means of doing this (besides the “Finisher” approach taken in §4.1.1) would be if that party supplies a key in a message to the smart contract that unlocks a sealed commitment embedded in the code, revealing publicly the output key that is required in order for anyone to be able to interpret the final state. This only requires trusting the third party to
send that “final reveal” message at the requisite time and does not require any parties to trust the message’s content (since it is verified by successfully unlocking the commitment).

Finally, we should point out that, even in cases when outputs are not visible and there is no fairness issue, an ability to look ahead (even retroactively, from a past state) and examine even the garbled states that would result from different inputs can, in some cases, result in a reduction of functional privacy; this issue and methods to address it are addressed in §§5.4.2–5.4.3 below.

5.4.2. **Reconvergent Arcs Problem**

We saw in our original requirements in §3 that in general we wish for the participants in the protocol to be able to infer nothing about the structure of the original state machine \( F \) from the garbled machine \( G \), except for a limit on the size of \( F \). However, there is one case in which the garbling method described in §5.2 falls short of meeting this requirement, and that is in the case where two different arcs from a given origin state could be immediately triggered by two different input values that could both be provided by a given input provider at the current (or a past) time step that would both lead to the same destination state \( s \). In this case, the randomized representations \( K(s, t) \) of that state resulting from decrypting both arcs would be the same, and so, the input provider, inspecting the garbled machine representation \( G \) any time after the origin state and the prior inputs on step \( t \) have been unlocked, could infer that both inputs yielded the same state in the original machine \( F \).

However, there are some simple ways in which this problem can be avoided, in the case of single-step reconvergence. One way would be to simply duplicate each state in the machine that has convergent arcs impinging on it from a given origin state, so each of those arcs goes to a different copy of the state. Another way would be to perform each state transition in two steps: First, the arc to be traversed is selected as usual, and the machine transitions to a unique intermediate state (which can be visualized as sitting in the middle of the arc). Then, another participant (a Stepper) provides a coded “go” token \( \odot_v \) which varies depending on the set \( v \) of this step’s input values (this is necessary to prevent the reconvergence from being retrospectively inferred); once this token is received, the machine then completes the transition to the ultimate destination state. These two methods impose different complexity overheads on the garbled machine representation and on the execution protocol. Which method has the least overhead depends on the details of the situation.

5.4.3. **Reconvergent Paths Problem**

In §5.4.2 above, we alluded to the fact that an input provider could reconsider his past inputs and look at what would have happened if he had changed one of his inputs on a past time step, but all other inputs (by himself and other parties) on that and subsequent time steps (up to the present) stayed the same. This could be viewed as a form of replay attack that would allow multiple input sequences that eventually reconverge onto the same state to be identified, if not for the fact that the state-dependence of the input provision keys in our algorithm ensures that other input providers’ inputs can’t be replayed for any state sequence except for the one that occurred in the actual run of the Machine. Another solution, which we adopt in the circuit-based version of our algorithm in §9, would be to require a separate Unlocker participant to decode each actual input key provided before it can be used. (This is similar to the Stepper approach discussed in §5.4.2 above.) However, we avoided adopting that particular approach in our implementations of the state-machine model, because it adds complexity (in terms of extra rounds of interaction) to the protocol.
5.4.4. Collusion Problem

Of course, the primary weakness of the entire class of methods relating to the approach described in this report is that, if multiple different participants in the protocol should happen to collude with and/or share keys with each other (or with common outside parties), then they (or any entities that end up obtaining sufficient keys from multiple participants) can infer additional properties of the function being computed, and thereby compromise the functional privacy guarantees of the system. The main method by which the Company could try to mitigate this weakness would be if it takes care not to facilitate participants’ even finding out each others’ identities, locations, or contact information.

However, given that protocol participants could always advertise their roles to each other surreptitiously using unknown or “black market” forums, it seems difficult in general to guarantee that collusion between participants could never possibly arise without the knowledge of the Company. Thus, these methods may primarily be useful in contexts where there is at least limited trust in or control of the participants by the Company, or where an absolutely perfect guarantee of functional privacy is not strictly required.
6. TECHNICAL DETAILS OF THE PROTOTYPE IMPLEMENTATION

In this section, we provide concrete technical details of the various representations, encodings, and protocol messages utilized in the existing prototype implementation of GABLE (based on Ethereum).

6.1. Prototype Implementation of a Garbler in Python

Appendix B gives the complete Python 3 code (both without and with comments) for a simple reference implementation garbler.py of a Garbler, which translates a given finite state machine $F$ into a garbled description $G$ of that machine, suitable for execution in the context of an appropriate Executor, such as the one described in the next subsection (6.2) below.

In this implementation, the initial finite state machine $F$ to be garbled (the one from Figure 5-3) is specified in a JSON-format file that appears as follows:

```
[["SInit", {"A": "1", "B": "1"}, "SPass"],
 ["SInit", {"A": "0", "B": "0"}, "SFail"],
 ["SInit", {"A": "0", "B": "1"}, "SReset"],
 ["SInit", {"A": "1", "B": "0"}, "SReset"],
 ["SReset", {"A": "0", "B": "1"}, "SInit"],
 ["SReset", {"A": "1", "B": "0"}, "SInit"],
 ["SReset", {"A": "0", "B": "0"}, "SFail"],
 ["SReset", {"A": "1", "B": "1"}, "SFail"]]
```

This gives a list $A$ of arcs, where each arc $a \in A$ is represented as a list $(O, C, D)$ of three elements:

1. A string which is a label identifying the arc's origin state $O$.
2. A representation of the arc's condition set $C$, consisting of a map from strings labeling input variables $V$ to strings labeling their assigned values $v$. Thus, "A": "1" represents $A = 1_A$ for some input variable $A$.
3. A string which labels the arc's destination state $D$.

In this version of the Garbler, the machine's initial state $s_{init}$ is always the one labeled $SInit$. The garbler is invoked using the following command line template:

```
$ ./garbler.py [-h|--help] [-seed seed] [-time_steps nsteps] circuit
```

where the options enclosed in brackets [ ] are optional command-line arguments. The -h or --help option displays a help message; next, --seed seed sets the seed of the pseudo-random number generator to the given string seed to allow repeatability (if not provided, results will be nondeterministic); next, --time_steps nsteps sets the maximum number $\ell$ of supported time steps to the given integer nsteps, which defaults to 10. Finally, the sole required argument circuit is the name of the input file, in the format above.

The output of the Garbler then consists of the following files:

- $init.gc$ – The random 256-bit code for the initial state $s_{init}$. It is in JSON format, consisting of a bracketed list containing one string of 64 hex digits.
- $t.gc$ – For each time step number $t \in \{0, \ldots, \ell - 1\}$, this is the garbled circuit $E(t)$ for that time step, in JSON format as a list of encrypted arcs, each of which is a list of two elements,
which are 64 hex digit strings for the encrypted fields $E_{next}, E_{valid}$ of the arc. For example, for a machine with 8 arcs such as the one in Figure 5-3, we might have:

$$
[\text{"4aa23a389f984ec6b8150c42df256c64182c42b2eeab69ae664bd18c1879d8",}
\text{"df3e363587e9630c9ed54556f09de3327424c397a57245e9f8980c8c12"}],
[\text{"ea056d860d8318f677ae2f9823b805ae631013b4a2b4e67ad532074d85bf8",}
\text{"5e1a65bfde9c1f82d1233b465a7a1de88fc402eb44e4d84f7747679685ff9a"}],
[\text{"737f3ca364c68a472b2d2cded8a48896a9759f0321903ebcc7e18d837cb",}
\text{"b086ddffdc996ee26f6115a070b3764305cc6aa873e9f03105b5df08136a44"}],
[\text{"a546f18a70f4ef1e1100ea92e1d361ee649eda4218fa59b27897d9f5b8b85",}
\text{"768973a8680364dd97cac09a9ec626fad335bb38097ad4c136d7fba9eb"}],
[\text{"b580f99bb56c2df2e5eb68b1894ed5252af2f9e7076d7b129d8e9e6b631",}
\text{"148885ebd0c6b9c9633689c750536c6c168aa888d713368f1a5cb2cb0f6cf4"}],
[\text{"adc27bae5346ead211f5e8a46f6f6ad9b66e4f8d3629e50980eaa3375a1941f6d",}
\text{"55bd3550456ae59687806a1c89369f3a23c8ebbb59e223e4b11000e29946"}],
[\text{"95ccc0158e47e0a708a104fca8667333cd7e9738108d416713420590897e",}
\text{"f0b8f4e9b20da553364c45b5d4b278256e0434e2926aba27630a1eae53f1c6"}],
[\text{"03c0373f748f0df96d453f47e14688a655c04015f88e16d46c246af3560d1e",}
\text{"f9d9feca073589c643a06d224861329c45e02626a1b2e5e782ee625dcaaa272d2"}]
$$

(This is the actual content of file $0.gc$ for the present Garbler on the above input with seed ‘xyz’.)

- **t-in.keys** – For each time step number $t \in \{0, ..., \ell - 1\}$, and each line name $ln$, this file (in JSON format) gives the map from value labels to 64-hex-digit value codes (“keys”) for that line. The line names include the labels of all the input variables $V$, as well as two special line names _in_states and _out_states, which denote the input and output state variables $S_{t-1}$ and $S_t$ respectively. For $0 \leq t \leq \ell - 2$, the file $t-out_states.keys$ is identical to the file $(t+1)-in_states.keys$, since both reflect the set of keys $\{K(s,t) \mid s \in S\}$. For example, the following is an instance of the file $0-in_states.keys$ for the Figure 5-3 machine, specifying the possible coded initial states $K(s,-1)$:

```json
{"SFail": "5c8b28ab72dfe4ad5dd643f8d9eaf9d841f6bc3617fa335387a737438e3b1fe",
"SInit": "7128b98ce5a2a8cc5f8db6f52bf6c1e7b20b612e816650b677d1d9194f8",
"SPass": "2f7214f79f2ce8b9517b6977b40cd1ba42d0c011idd1b6f2026ff82ac99e09",
"SReset": "38b947a283b90b69a7ce1682e0d9e19c45119cf0b1dbf64e2a56e5a1cd8694"}
```

In a real deployment scenario, the keys for the input variables would be distributed to input Providers, and hashed versions of the keys needed to interpret the state variables (not generated in the prototype Garbler implementation) would be distributed to any Spectators as appropriate.

Meanwhile, the garbled circuit files *.gc (representing the garbled Machine $G$) would be processed into an executable form Exec[$G$] (such as the one discussed in the next section), which is then deployed on a suitable blockchain.

### 6.2. Prototype Implementation of an Executor in Solidity

Appendix C gives the complete Solidity code (both without and with comments) for a simple example of a smart contract **ExecutableMachine** comprising an executable implementation Exec[$G$] of an Executor that is interpreting a hard-coded representation of a garbled version $G$ of the state-machine function $F$ represented in Figure 5-3. In this (very simple) implementation, we use a variation on the gather-all-inputs method but with no authentication of input Providers (leaving us open to a spam-based DOS attack). However, those limitations could be straightforwardly ameliorated; we are just keeping this example simple for expository purposes.
In this code, an encrypted arc $E = (E_{next}, E_{valid})$ is represented with a simple data structure:

```solidity
class EncryptedArc {
    uint256 encNext;
    uint256 encValid;
}
```

containing the $n = 256$-bit encoded fields $E_{next} = encNext$ and $E_{valid} = encValid$ of a given arc.

For a given time step $t$ and (randomized) arc index $k \in \{0, \ldots, q - 1\}$, we use the procedure described in Sec. 6.1 above to generate its encrypted representation $E(a_k, t)$, and then simply hard-code that data into a corresponding statement within `ExecutableMachine`'s constructor, for example as follows:

```solidity
arcs[t][k] = EncryptedArc(0x4aa23a389f984ece6b8150c42df2565c64182c42b2eeab69ae664bd18c1879d8,
                          0xdf8e363587e796300c9ed5456d556f0d9de3327424397a57245ef98d0c8cc12);
```

where $t, k$ here stand in for the corresponding literal integer constants. See subappendices C.1 and C.2. (Example data for only one time step, $t = 0$, is included in the first listing, for brevity.)

Similarly, the 256-bit code for the initial state $s_{init}$ can be provided in a corresponding initializer for one of the contract’s constant state variables, like so:

```solidity
uint256 constant sInit = 0x7128b98ce5a2a8cce5f8db6fc52cbf6c1e7b20b6122e8168650b067d1d194fb6;
```

and then another, publicly accessible state variable `curState`, for keeping track of the coded current state $K_{cur} = K(S_{t-1}, t-1)$, is initialized to `sInit`.

We then allow input providers to provide inputs to `ExecutableMachine` through a public function

```solidity
provideInput(uint8 varIndex, uint256 value)
```

where `varIndex = i`, the index of the input variable $V_i$ whose value is being provided, and `value` is the $n = 256$-bit input provision key $p_{t}^{ij}$ that represents value $v_{t}^{ij}$ being provided for input variable $V_i$ for the current time step $t$ and current garbled machine state `curState = K_{cur}`. Provision keys for different input assignments are then combined together (using $\oplus$, bitwise XOR, as the Combinator function), and (once values for all of them have been provided) are then used, in an internal function `executeStep()`, to compute the new value of the encoded current machine state `curState` in the manner that was described in Sec. 5.3 above. Spectators can invoke the `curState()` public getter function to obtain the new current state, from which they can then infer authorized output information as per the procedure we discussed in Sec. 5.3. See the listings in Appendix C for further details.

### 6.3. Testing the Prototype Implementation

To test the Solidity code for the prototype Executor on a development blockchain, we used the Truffle suite of Ethereum development tools [18]. Appendix D gives a complete procedure and test script for taking the garbled version of the machine from Figure 5-3 through the following example sequence of execution steps:

- **Time step #0:** Input assignments $A = 0$, $B = 1$ take the machine to state $S_{Reset}$ ($S_{Reset}$).
- **Time step #1:** Input assignments $A = 1$, $B = 0$ take the machine back to state $S_{Init}$.
- **Time step #2:** Input assignments $A = 1$, $B = 1$ take the machine to state $S_{Pass}$ ($S_{Pass}$), which is a final state, since it has no successors.
• At this point, the test script queries the machine for its (coded) current state, to make sure that it has arrived in the correct state.

We carried out this test within a simple sandboxed environment (a development chain running on a single firewalled host), and it was successful.

More extensive demos are discussed in the next section.
7. DEMO APPLICATIONS

Since the start of calendar year 2019, we successfully implemented two different simple but complete demonstration applications, which were deployed on a test network within a virtual sandbox (emulatory-type) environment for blockchain experiments developed at Sandia called FIREWHEEL [19]. Both demos were fully tested and work precisely as intended.

The two demo applications that have been developed so far are:

1. Supply-Chain Provenance Tracking Demo (based on the state machine in §4.2.1).
2. Millionaires’ Problem Demo (based on a state machine similar to the one in §4.1.1).

We also briefly considered developing a demo for the Dungeon Race game example discussed in §4.1.2, but that one was deemed too complex for hand-development, which motivated the preliminary exploration of a state machine compiler concept, which will be discussed in §8.

However, our current plan, moving forward, is to first develop a circuit-based implementation of GABLE such as is described in §9, and then a variety of much more ambitious applications can be developed much more feasibly on top of that platform.

A working name for the early circuit-based version of GABLE is GOOSE (which stands for Garbled Online Obfuscated Secure Evaluation), and a preliminary implementation of GOOSE is being prototyped by our collaborators at Georgia Tech, and is currently undergoing early testing and further development. A separate report will describe the GOOSE prototype once it has been completed.

Some additional details regarding the two demo applications already implemented follows. How to set up and run the demos is documented in much more detail in a separate document whose most recent version, as of this writing, has the filename “GABLE demo instructions 0v12.docx.”

7.1. Supply-Chain Provenance Tracking Demo

As mentioned above, this demo is based on the state machine that is illustrated in Figure 4-2 in §4.2.1. A variety of scripts are used to set up, build and deploy the demo in our test environment, which is a sandboxed virtual network including 4 main virtual host nodes, each running an Ethereum client (geth) and configured as peers on an emulated private Ethereum test net:

- Node #0: Master node. Deploys garbled smart contract to network.
- Node #1: Runs a vendor client for Vendor #1 (of 3).
- Node #2: Runs a vendor client for Vendor #2 (of 3).
- Node #3: Runs a vendor client for Vendor #3 (of 3).

There are also two additional virtual nodes running a local DNS server and a blockchain explorer. Presently the demo just displays diagnostic text output on each node summarizing what is happening at that node. As an example, Figure 7-1 below shows the diagnostic output from Node #1 in a sample run. Note that, for purposes of this demo, appropriate spectator authorities were distributed to the respective vendors that allow them to interpret only the machine states pertaining to them specifically.
The gas cost to run this demo was measured and is shown in Table 7-1, and pie-charted in Figure 7-2. As of March 13, 2020, given a sample average gas price of 9 gwei/gas on the mainnet, and an Ether price of US$128.09/ETH, the total cost comes to US$2.78. Note that of the total, approxim-
ately 15% is Truffle framework overhead (relating to the Migrations contract), ~73% is the deployment of the ExecutableMachine contract containing all of the garbled arc data, and on average, each of the 5 vendor transactions (updating the machine state) takes only about 2.6% of the total.

Table 7-1. Gas Cost for Vendor Demo Transactions (in order of occurrence)

<table>
<thead>
<tr>
<th>Description</th>
<th>Block Height</th>
<th>Gas Required</th>
<th>Cost in ETH @ 9 Gwei/gas (avg. as of 3/13/20)</th>
<th>Cost in USD @ US$128.09 / ETH (market as of 3/13/20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deploy Truffle Migrations contract</td>
<td>687</td>
<td>284,908</td>
<td>2.5642 mETH</td>
<td>$0.3284</td>
</tr>
<tr>
<td>Initial transaction to Migrations contract</td>
<td>689</td>
<td>42,034</td>
<td>0.3783 mETH</td>
<td>$0.0485</td>
</tr>
<tr>
<td>ExecutableMachine contract deployed</td>
<td>691</td>
<td>1,756,030</td>
<td>15.8043 mETH</td>
<td>$2.0244</td>
</tr>
<tr>
<td>Record deployment to Migrations contract</td>
<td>693</td>
<td>27,034</td>
<td>0.2433 mETH</td>
<td>$0.0312</td>
</tr>
<tr>
<td>Vendor 1 sends “R” symbol</td>
<td>773</td>
<td>73,351</td>
<td>0.6602 mETH</td>
<td>$0.0846</td>
</tr>
<tr>
<td>Vendor 1 sends “T” symbol</td>
<td>786</td>
<td>57,466</td>
<td>0.5172 mETH</td>
<td>$0.0602</td>
</tr>
<tr>
<td>Vendor 2 sends “R” symbol</td>
<td>796</td>
<td>60,121</td>
<td>0.5411 mETH</td>
<td>$0.0693</td>
</tr>
<tr>
<td>Vendor 2 sends “T” symbol</td>
<td>809</td>
<td>60,057</td>
<td>0.5405 mETH</td>
<td>$0.0692</td>
</tr>
<tr>
<td>Vendor 3 sends “R” symbol</td>
<td>819</td>
<td>58,287</td>
<td>0.5246 mETH</td>
<td>$0.0672</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>2,419,288</td>
<td>21.7736 mETH</td>
<td>$2.7890</td>
</tr>
</tbody>
</table>

Figure 7-2. Pie chart of gas cost distribution for the supply-chain demo. Data from Table 7-1 above.
7.2. Millionaires’ Problem Demo

This demo is based loosely on the toy application discussed in §4.1.1, except that, instead of using the state machine displayed in Figure 4-1, we used a different state machine designed for an input model in which input providers take turns providing consecutive bits of their respective binary inputs. This was done to simplify the process of reconvergent arc elimination discussed in §5.4.2.

To explain further: With only one input provider moving per time step, elimination of reconvergent arcs requires at most splitting a given destination state into 2 states; whereas, with two input providers who may move in either order, splitting a destination state into 3 states would be required in some cases. See Figure 7-3 for the new state machine.

For purposes of this demo, each participant is given a limited reader authority that allows them to interpret (before the final state is reached) only whether it is their turn to move in the current state or not. On its own turn, each participant provides 1 bit at a time of its wealth value, as shown in Figure 7-4 and Figure 7-5. In this example, wealth values are 4 bits long, so after 8 turns (4 for each player), input collection is complete, and the Finisher sends the “finish” symbol “$\square$” (represented as “F” in this demo, see Figure 7-6) which causes a transition to a final state, $S_{ag}$ (player A is richer than B) or $S_{al}$ (player A is not richer than B); the final state is then readable by all participants.
An experiment was done to characterize how the bytecode size and gas cost for the Millionaire’s Problem demo varies as a function of the number of bits in the wealth values (Figure 7-7), for numbers of bits ranging from 1 to 5, except that the contract for the 5-bit case could not actually be deployed as a single contract due to a transaction size limit of 32,768 bytes in our local geth, as presently configured.

To enable larger application instances to be deployed, regardless of any contract size limits, we confirmed that a given ExecutableMachine smart contract can successfully be broken into several contracts, a master contract along with any number of additional contracts which store the garbled
We used this method to successfully test Millionaire’s Problem instances up to 32 bits, which would have cost 75,185,098 gas (~677 mETH, US$86.67) to deploy on the mainnet as of 3/13/’20.

7.3. Remarks on Red-Teaming

After implementing some working demos, some thought was given as to whether it would be worthwhile to have a “red team” at Sandia attempt to compromise the protocol. Several technical discussions occurred with Sandia red team experts, discussing various tradeoffs, and thinking through a
meaningful assessment scenario whereby the red team would attempt to reverse-engineer the functional purpose of an executing GABLE contract given varying amounts of key/label information.

The decision was made to not proceed with actually performing this assessment due to a variety of factors, the most important of which being that real GABLE deployments will be done in the context of additional infrastructure, i.e., the process by which the Garbler distributes access keys, the endpoint security of the nodes participating in the protocol, and any supporting dApp infrastructure. A red team assessment of a real deployment would likely prioritize weaknesses in this surrounding technology over the core protocol itself, which has a relatively minimal surface area.

An intermediate approach was ultimately taken for anecdotally characterizing the functional privacy of deployed GABLE contracts. Specifically, we analyzed GABLE-produced smart contracts with the ERAYS Ethereum reverse-engineering toolkit [20]. We observed that (as expected per our design), contracts for differing state machines are structurally identical (compare Figs. 7-8 & 7-9). We have not yet taken the logical next step of observing dynamic program traces of different state machines to ensure that the state/transition sequences do not exhibit obvious artifacts of their functional purpose—but they should not, per the design of our protocol.

![Figure 7-7. Bytecode size and deployment gas cost for the Millionaire's Problem demo.](image)

The chart on the left shows the size of the compiled bytecode for the ExecutableMachine smart contract in bytes as a function of the number of input bits; the chart on the right shows the gas cost to deploy the contract. As expected, the scaling relation for both functions is exactly linear. At current gas and ETH prices as of this writing (March 20, 2020) the 9,526,903 gas needed to deploy the 4-bit demo would cost about US$2.56.
Figure 7-8. **ERAYS analysis of the GABLE contract for the supply-chain demo.**

(Left) State transition diagram for the supply-chain demo, from Figure 4-2; (Right) Output from ERAYS analysis of the corresponding smart contract bytecode. Compare with Figure 7-9.
Figure 7-9. ERAYS analysis of the GABLE contract for the Millionaire’s Problem demo. (Left) State transition diagram for the Millionaire’s Problem demo, from Figure 7-3; (Right) Output from ERAYS analysis of the corresponding smart contract bytecode. Compare with Figure 7-8.
In early 2019, a fair bit of time was spent doing some exploratory development work pursuant to the goal of implementing a state machine compiler, which would be capable of automatically transforming high-level state machine descriptions written in the Python programming language into corresponding executable garbled contracts. The original vision for this compiler was that it would be flexible enough to support many of the variations on the protocol discussed in this document (different input models, output models, etc.) while also providing automatic mechanisms to mitigate the weaknesses previously discussed in §5.4. Some of the concepts behind the compiler are discussed below.

At present, development work on this compiler remains incomplete, due to resource constraints, and also because a higher priority is now being placed in our project on the development of a different, more powerful computation model, such as the one described in §9—which, if that effort is successful, could render our present, state-machine-based approach somewhat obsolete. However, for archival purposes, we nevertheless describe the present state of the compiler concept below.

8.1. Compilation Stages

Figure 8-1 depicts the sequence of stages in the compilation and deployment process that takes an application from the source code into a garbled machine running on a blockchain. The various steps in this process are described below.

We start out by writing out a functional description of the application’s behavior as a Python script or module (.py file) (orange) utilizing a certain library of sub-packages collected under the top-level package name app_compiler. This library provides a variety of base classes and compiler components that are used in the compilation process. When executed, the application module (or a separate top-level script) kicks off the compilation process.

The concept behind this compiler is that the source code for the application defines the core functionality of the application in a state-updating method that provides the logic to update the machine state in terms of ordinary state variables. In the first step of the compilation process, called functional compilation, the compiler traverses the entire tree of possible state transitions (starting from the initial state), identifying recurring nodes and merging them as it goes along to form the state machine graph. This is called the abstract state machine (light blue). It is in a raw form, not yet ready for garbling.

The next step in the compilation procedure is the process of concretization, in which we take the abstract state machine produced in the previous step and transform it in several respects to produce a concrete state machine (gray) that is suitable for garbling. Important functions accomplished as part of the concretization process include the following:

1. **Automatic unrolling.** This refers to a state machine transformation that ensures that no individual state can be encountered more than once in any possible run of the machine. One straightforward way to do this is to include a state variable that counts the number
of steps of execution. A method utilizing fewer states in some cases would split off mutated versions of only those states that would otherwise create a graph cycle.

2. **Reconvergent arc splitting.** This refers to a state machine transformation in which we look for cases where two (or more) outgoing arcs from a given node point to the same destination node—this is the reconvergent arc problem mentioned in §5.4.2. In such cases, we can simply create new, mutated version(s) of the destination node.

3. **Output delaying.** This refers to a state machine transformation in which we take an output that would normally be visible right away (as an output from the state transition, or the destination state), and instead store it in a hidden part of the state, and arrange for it to be visible as an output from the next transition or state. In certain input models, this transformation mitigates the lookahead problem discussed in §5.4.1, by preventing the input provider from being able to see the result of their input until after another participant has made a move.

In the third major step of the compilation process, *garbling*, we simply garble the concrete state machine, as per an appropriate procedure such as the one described in §5.2. The result of this process is generally a set of files (red), including ones that contain appropriate access authorities for distribution to application participants. Also included is a file that describes the garbled machine $G$.

The fourth step, *packaging*, involves transforming the garbled machine description $G$ into an executable form (dark blue) $\text{Exec}[G]$ which is ready to be deployed on a (public or permissioned) blockchain. In this step, we could alternatively target a variety of other back-end smart-contract platforms (besides Ethereum).

Finally, the fifth step is *deployment*, in which the smart contract for the executable machine is actually deployed to a specific distributed smart-contract blockchain, and meanwhile, the keys to access the machine are securely distributed to authorized participants. At this point, the machine is ready to use.

### 8.2. Software Architecture

At present, the preliminary (and very incomplete) experimental code base for the compiler is organized into a number of packages, which are sub-packages of the top-level `app_compiler` package. The list of packages is shown in Figure 8-2. Some further description of their content and status follows.

#### 8.2.1. Access Package

The `access/` package collects modules having to do with access information and accessor authorities. Currently, it includes the following modules:

- `accessInformation.py` – Defines a class for objects that gather together all authorizations needed to access inputs and outputs of a given garbled machine.

- `authority.py` – Defines a base class for authorities (sets of access authorizations), and subclasses for special classes of authorities (e.g., ones for use by the general public, the Garbler, or the union of all defined protocol participants).

- `readerAuthority.py` – Defines subclasses of the `Authority` class for various types of read-only access to the garbled machine.
• **starterAuthority.py** – Defines subclasses of Authority for enabling an entity to
initially start up the garbled machine in any of a designated set of initial states.

• **writerAuthority.py** – Defines subclasses of Authority for enabling various types of
write-only access to the garbled machine.

### 8.2.2. Crypto Package

The crypto/ package collects modules defining various types of cryptographic primitives to be used
in the expanded GABLE system, including hash functions, commitments, and encrypted data entries.
It is presently intended to include the following modules:

• **commitment.py** – Defines classes for cryptographic commitments.

• **entryEncrypter.py** – Generates encrypted representations of data entries (identified by
entry identifiers).

• **hashFunction.py** – Classes for hash functions which may be used in the garbling pro-
cess.

### 8.2.3. Functionality Package

For our purposes, a complete functionality characterizes an application’s functional behavior, I/O meth-
ods, access authorities and participants. The functionality/ package collects associated modules,
including the following:

• **application.py** – An application object gathers together all information associated with
a particular GABLE application throughout the compilation process.

• **behavior.py** – Defines an application’s functional behavior, through specification of as-
associated information such as its initial abstract state.
• **functionality.py** – Base class for defining an application’s overall functionality. This includes a specification of the application’s I/O variables & methods, access authorities, participants, and functional behavior.

• **inputMethod.py** – Defines classes for defining input methods, which specify the protocol to be used for providing inputs to the application and updating its state.

• **outputMethod.py** – Defines classes for defining output methods, which specify the protocol for obtaining outputs from the application.

### 8.2.4. Garbler Package

The **garbler/** package collects modules defining various aspects of the Garbler stage of compilation, including the following:

• **garbledArc.py** – Defines classes for garbled representations of a *TransitionArc* in a finite state machine.

• **garbledFunctionality.py** – Defines a class for encapsulating a garbled version of an entire application *Functionality*.

• **garbledInputValue.py** – Defines classes for objects representing garbled versions of *InputValue* objects. This includes input value keys as well as input provision keys.

• **garbledMachine.py** – Defines classes for representing a garbled version of a particular (concrete) finite state machine.

• **garbledOutputKey.py** – Defines classes for representing a garbled version of an output key that can be used to un-garble garbled output messages.

• **garbledOutputMessage.py** – Defines classes for representing garbled versions of output messages.

• **garbledState.py** – Defines classes for garbled representations of an individual state in a finite state machine.

### 8.2.5. I/O Package

The **io/** package collects modules associated with machine input and output, including the following:

• **bitVariable.py** – Defines classes for working with binary-valued input variables.

• **bitVariableType.py** – Defines a type object for binary-valued input variables.

• **inputSymbol.py** – Defines a class for symbolic tokens representing specific input values.

• **inputValue.py** – Defines a class for “input value” objects, which assign a specific input token to a specific input variable.

• **inputValueSet.py** – Defines classes for representing partial (or complete) variable assignments. Note that some kinds of input value sets could actually assign multiple different symbols to a single input variable, as alluded to in footnote 2 on p. 36.

• **inputVariable.py** – Defines classes for representing individual input variables for a given application.
• **inputVariableType.py** – Defines classes for representing the type of a given input variable. A type specifies the set of allowed input symbols, and whether multiple symbols may be supplied for a given input variable within a given machine cycle.

• **lineVariable.py** – Defines classes for line variables; here a “line” is a concept that includes both input and output variables, and possibly also a “previous state” variable.

• **lineVariableType.py** – Defines classes for specifying the type of a line variable. These are base classes from which to derive input variable types, output variable types, etc.

• **outputMessage.py** – Defines classes for representing plaintext messages to be produced as output from a given machine.

• **outputs.py** – Defines classes for gathering together all of the output messages that may be produced by a given machine.

• **outputVariable.py** – Defines classes for representing an individual output channel from a machine.

• **outputVariableType.py** – Defines classes for representing the type of a given output variable.

• **standardSymbols.py** – Defines objects for a number of predefined standard symbolic tokens, such as the ones designated 0 (binary zero), 1 (binary one), ■ (end), ⊥ (undefined), ⊚ (go to next state), and ⊙ (produce final output & halt).

### 8.2.6. **Machine Package**

The **machine/** package collects modules for representing finite state machines, both abstract and concrete versions. This includes the following modules:

• **abstractMachineArc.py** – Defines classes for representing state-transition arcs in an abstract finite state machine.

• **abstractMachineState.py** – Defines classes for representing state-machine states in an abstract finite state machine.

• **concreteMachineArc.py** – Defines classes for representing state-transition arcs in a concrete finite state machine.

• **concreteMachineState.py** – Defines classes for representing state-machine states in a concrete state machine.

• **condition.py** – Defines classes for representing a transition condition that is attached to an arc in a finite state machine.

• **internalVariable.py** – Defines classes for representing individual *internal* state variables within a state machine.

• **machineState.py** – Defines base classes for representing individual machine states in a finite state machine; both abstract and concrete machine states are derived from these.

• **reconvArcSet.py** – Defines classes for representing a specific reconvergent set of arcs within a given abstract finite state machine. (These get resolved in the concretization stage.)
- **stateMachine.py** – Defines base classes for representing finite state machines; both abstract and concrete state machines are derived from these.

- **stateNode.py** – Defines base classes for representing individual nodes in a state transition graph, representing a given state of a given finite state machine. Both abstract and concrete machine states are derived from these.

- **transitionArc.py** – Defines base classes for representing an individual arc in a state transition graph, representing a given transition within a given finite state machine. Both abstract and concrete machine arcs are derived from these.

### 8.2.7. Participants Package

The **participants/** package collects modules for representing various types of entities that may play a role in applications of the GABLE system. This includes the following:

- **accessor.py** – Defines base classes for participants that may access (read from and/or write to) the machine.

- **bitWriter.py** – Defines a class for writers that are associated to a specific binary input variable.

- **finisher.py** – Defines a class for special type of writer upon which is conferred the authority to end the machine’s operation, causing it to produce its final outputs (if any) and halt.

- **participant.py** – Defines base classes for describing all protocol participants. Also defines subclasses for a variety of special types of participants.

- **reader.py** – Defines a class of participant that has read-only access to a given machine.

- **starter.py** – Defines classes for a special type of participant that is authorized to start up the machine in any of one or more designated initial states.

- **writer.py** – Defines a class of participant that has write-only access to the machine.

### 8.2.8. Stagers Package

The **stagers/** package collects modules implementing entities that execute the various steps in the compilation process. These include:

- **deployer.py** – Implements an entity that can deploy a given `ExecutableMachine` to a designated distributed computing platform.

- **functionalCompiler.py** – Implements an entity that can convert a given application functionality from its source form to an abstract finite state machine representation.

- **garbler.py** – Implements an entity that can generate the garbled representation of a given application functionality, given its concrete state machine representation.

- **packager.py** – Implements an entity that can package a garbled machine into the form of an executable machine that is ready to deploy to a given type of distributed computing platform.

- **concretizer.py** – Implements an entity that can transform an abstract finite state machine into a concrete finite state machine that is ready for garbling.
8.2.9. **Stages Package**

The `stages/` package collects modules defining classes that are used for representing applications in various stages of compilation. These include:

- **deployedMachine.py** – Defines classes that represent a given deployed machine instance, and that can track the information necessary to access it.

- **executableMachine.py** – Defines classes that can be used to represent a garbled machine together with an Executor (garbled-machine interpreter) targeted to operate on a specific set of distributed computing platforms.

The other 4 stages (abstract machine, concrete machine, garbled machine, functionality) are presently defined in other packages.

At some point, it may be desirable to implement a master compiler application that is capable of taking the target application from any stage of compilation to any specified later stage.

8.2.10. **Testing Package**

The `_testing/` package is intended to collect together modules defining a set of automated unit tests which may be used for regression testing, etc. At present, this package contains just a single module called `_runUnitTests.py`, which runs all of the unit tests.
9. A MORE EFFICIENT COMPUTATION MODEL

As we mention elsewhere in this document, the original state-machine model of computation utilized in GABLE is not efficient for carrying out general computational tasks. This is true for the simple reason that, for an arbitrary computation operating on a \( w \)-bit memory, there are in general up to \( 2^w \) possible states of the machine (at least—note, this ignores any additional control state that may exist separately from the memory). Thus, even for a fairly simple computation, the size of an explicit state-machine representation of a desired algorithm can very quickly become completely intractable.

A more efficient model of computation is the circuit model. In the circuit model, rather than having just one monolithic internal state, you have an array of internal state variables, each with some limited number of allowed values; a simple approach that suffices is to use only binary (2-state) state variables, or bits. Then, each step of the computation consists of computing new values for some subset of the state variables in the system, each as a function of some limited number of state variables on the previous time step. The functions computing new values of state variables can generically be called gates; special cases, for binary state variables, are the classic Boolean gates (such as NOT, AND, OR, XOR, NAND, NOR, etc.). Even just 2-input NAND gates by themselves (or NOR gates) are sufficient for universal computing. In a generic picture of the circuit model, a computation can be represented as shown in Figure 9-1 (compare with Figure 5-1). This can also be viewed as one giant state machine, or, alternatively, as \( w \) small state machines in parallel, with interconnects between them.

![Figure 9-1](image_url)

*Figure 9-1. A circuit computation operating for up to \( \ell \) layers of logic / time steps.*

At each time step we have at most \( m \leq w \) input variables; the gate inputs get assigned from these and the previous step’s internal state variables by the interconnect network. Unlike in Figure 5-1, now we assume that the state-updating circuitry (including the interconnect configuration, and the identities of the gates) may be different on each time step. Any desired circuit computation can be embedded into this structure, assuming the depth \( \ell \) and width \( w \) are sufficient. In the text, we discuss how the entire circuit structure may be obfuscated.
The structure of Figure 9-1 already hints at how a circuit-based computation may be obfuscated. The evaluation of any given target gate can be treated exactly like the evaluation of a state update step in §5, except that the input lines are now all internal state variables, rather than all but one of them being external input variables. Apart from this, each gate can be garbled in exactly the same manner as a state transition table in §5; in both cases, we are using the same underlying technology of garbled lookup tables. Gates can be evaluated (by any party) as soon as their inputs are known.

However, note that one important difference is that, in this new picture, the input variables and internal state variables will get routed to gate inputs by the interconnect network, which can be composed generically as (for example) a Thompson generalized connector network [3] as shown in Figure 9-2, composed of 2-input, 2-output routing elements. Each of the internal blocks (blue) shown in the routing element (bottom of figure) is a garbled gate, like any other, except that, in the very first layer of routing elements only, there is a special third input (red) to each routing unit called “activate” that

![Diagram](image)

**Figure 9-2.** Generalized connector network, and a routing element in this interconnect fabric.

(Top) An \( N = 2^n \)-input Thompson network [3], with \( 4n - 3 \) layers of \( N/2 \) routing units each, is capable of assigning each of its \( N \) outputs from any of its \( N \) input lines; half of this network is used for signal splitting (fanout), and the other half is used for permuting the order of signals. (Bottom) Each unit’s function is simply to assign each output from one or the other input; however, logic could be incorporated within here as well. Each block within the unit is a selector gate which can be evaluated in the same way as the gates in the top-level diagram (Figure 9-1), with one exception: In the first layer of routing units only, a third “activate” key \( a_i \) is needed, unique to each routing element, which is computed by a protocol participant (called the “Unlocker”) which was not involved in providing normal inputs for the current time step. The activate key is computed by the Unlocker using the formula \( a_i = h(K(L_{t-1}^{2i-1}) \oplus K(L_{t-1}^{2i}) \oplus k_t) \), where \( L_{t-1}^{i} \) denotes one of the \( N \) input lines, which may be either an externally-provided input variable, or an internal state variable (with these input lines, all together, being indexed by an integer \( j \), which may be written as \( j = 2i - 1 \) or \( j = 2i \) for some integer \( i \), where \( 1 \leq i \leq w = N/2 \)), and \( k_t \) is a key needed to start time step \( t \). See Figure 9-3 below for some additional discussion of this algorithm.

The structure of Figure 9-1 already hints at how a circuit-based computation may be obfuscated. The evaluation of any given target gate can be treated exactly like the evaluation of a state update step in §5, except that the input lines are now all internal state variables, rather than all but one of them being external input variables. Apart from this, each gate can be garbled in exactly the same manner as a state transition table in §5; in both cases, we are using the same underlying technology of garbled lookup tables. Gates can be evaluated (by any party) as soon as their inputs are known.
must be fed with a special key $a_i$ which is computed by hashing the two inputs to the element together with a special key $k_t$ needed to “unlock” the evaluation of the present time step. The purpose of this key is solely to (effectively) decrypt the input values provided, and thereby prevent input providers from looking ahead at the response of the circuit to different input values. Ideally, the time-step key should be held (and the corresponding activate keys provided) by an input provider that was not involved in providing inputs to the current time step, and that is not able to spectate on outputs from the present time step either, in order to prevent any possible lookahead issues.

The idea here is that all that this particular protocol participant (called the “Unlocker”) can possibly do wrong is to simply fail to provide the proper activate keys needed to initiate evaluation of this time step. To provide redundancy, there can be multiple unlocker entities upon whom are conferred the ability to provide the first-layer “activate” inputs for any given time step, to reduce the chance that the failure of any one such entity to fulfill its designated role will compromise the ability of the whole computation to function.

Alternatively, if we wish for only designated general input providers to be able to act as unlockers, while still preventing lookahead, this can be accomplished as follows: At each time step $t$, only a certain pre-designated subset of the available input providers are permitted to provide inputs for that time step, and only (all or a subset of) the remaining input providers are given the unlock key for that time step. If input providers do not collude, this protocol then suffices to prevent lookahead.

The above construction effectively comprises an implementation of universal circuits (UC), that is, circuits that can embed arbitrary Boolean circuits, and further, our garbled execution method totally obscures the underlying target circuit’s structure and functionality. Please note that in this implementation of UC, no separate programming input is required at all; rather, the circuit is configured at compile time, through the choice of truth tables. This is sufficient for privacy due to the general unintelligibility of the truth tables and data values in the garbled circuit, given that lookahead is being prevented.

To outline a complete protocol for execution of these garbled universal circuits:

1. As usual, the Garbler precomputes the smart contract that encodes the garbled machine and distributes the necessary access keys to input providers and output spectators. Initial garbled states for the $w$ internal state variables can be provided in the smart contract itself, or in a separate step by a special Initializer participant.

2. For each time step $t \in 0, 1, \ldots, \ell - 1$, we carry out the following three execution phases:
   a. Input-gathering phase. Input keys are gathered from general input providers for the current time step according to the input model being used. Input lines that are not needed can be left at some hard-coded default value such as $\perp_i$ (“undefined”). At some point, input gathering is deemed complete, and we progress to the…
   b. Activation phase. At this point, no more inputs may be accepted. Any unlockers on the network that hold the unlock key $k_t$ for the current time step may, at this time, compute the $w$ activation keys $a_i$ for all of the gates in the first layer of the interconnect fabric for the current time step. See Figure 9-3 below. Note that once all these keys have been computed, the entire circuit for the present time step may be evaluated (even privately by the unlocker), which is why it’s important, to prevent lookahead, that the unlocker not be allowed to spectate on any outputs from the current time step. In any case, if any unlocker is complying with the protocol, it will provide all $w$ of the required activation keys to the machine.
c. Evaluation phase. Now, the smart contract evaluates the entire circuit for the current time step; the final effect of this is to update the values of all \( w = \frac{N}{2} \) of its internal state variables, with their new values being the outputs from the final level of \( w \) two-input gates. These variables can then be potentially inspected by spectators as per their conferred reader authorities. However, note that the evaluation of the circuit itself is completely inscrutable to all participants in the protocol (except the Garbler), and reveals nothing whatsoever about the structure or function of the current logic layer. When the update phase is complete, we increment the time step counter, and commence the input gathering phase for the next time step.

A more detailed picture of the algorithm for garbling the first layer of routing elements is shown in Figure 9-3 above. Here, we illustrate (a) an arbitrary routing element in the first (leftmost) layer of routing elements shown in the top part of Figure 9-2; each selector gate within this element may be configured to select either the top (indexed \( 2i - 1 \)) or bottom (indexed \( 2i \)) input line from a pair of adjacent input lines, each of which could be either an output from the previous time step or an externally supplied input line. Either selection implies a particular logical truth table (b) corresponding to the routing element’s function. The entries of this truth table can be encoded by random keys in the usual fashion (c), except that, to prevent lookahead, we also require an additional, third input key \( a_i \) in each row, which, during the activation phase, is computed from the routing element’s inputs and the time-step unlock key \( k_t \) held by the unlocker, and provided to the online executor. Once the
activation key has been provided, the usual algorithm (d) unlocks the encrypted truth-table row and the encoded output symbols can be retrieved. The same procedure also suffices to then cascade results through the entire $4n - 2$ levels of the universal circuit (consisting of the $4n - 3$ level deep Thompson network, followed by 1 level of general application logic—although actually that one could also be merged into the last layer of the Thompson network), except that the activation keys are no longer needed for any layer after the first. Note that the response of the universal circuit to alternative input values cannot be probed by any participant, since the corresponding activation keys are not available. Thus, the output of every gate in the universal circuit simply looks like a random bit vector and cannot be further interpreted by any party (except by participants holding appropriate spectator keys). As with state machine arcs, the stored order of the truth table rows for each application gate or routing element should also be randomized.

The above design suffices to achieve our goal (garbled universal circuits with only logarithmic overhead), but it can be further simplified. First, although the above procedure subjects all line variables $t_{l-1}^j$ coming into a given step of the computation to an “activation” process (carried out by the unlocker) before they can be utilized in evaluating the garbled universal circuit, actually it is only necessary to activate the externally-supplied input values to this process, to prevent input providers from looking ahead (“trying out” different values for a given variable, as it were) and thereby learning something about the u.c.’s routing topology or the application layer function. Figure 9-4 illustrates a satisfactory procedure for this more limited activation. Here, the $m$ externally-supplied input variables, where $m \leq w$, are fed to the upper inputs of corresponding first-layer routing elements, after subjecting them to “activation” or decryption by the Unlocker for purposes of evaluating the garbled truth tables for the selector gates in the routing element.

The above construction assumes that, in the most general class of applications, meaningful new input values that were a priori unknown could impinge on the circuit at any time during its evaluation (i.e., at the time step corresponding to any layer of target logic). This provides a capability for participants to react to outputs (that they have spectator keys for) from previous time steps, providing a possibility for interactive computations to take place. However, for many applications, this feature may not be required, and the only meaningful inputs could all arrive at time $t = 0$. In this case, there are different options for how the remainder of the circuit layers could be evaluated:

1. We could have special external input providers called Steppers provide special Proceed (“⊚”) tokens to a designated input line (e.g., $V_t^1$) on each time step $t > 0$, with this token then decrypted by an Unlocker in the usual fashion, kicking off the evaluation of the u.c. for time step $#t$, with any actual logical value corresponding to the token (say “1”) ignored by the circuit. All other input lines can just be assigned hard-coded default values in the contract such as $⊥i$ (“undefined”). Thus, all $\ell - 1$ stages of execution after the first are driven forwards by an alternating sequence of “proceed, decode” actions by the Steppers/Unlockers.

2. Alternatively, given just a simple change to the Executor, we can simply skip input-gathering phases altogether for every time step $t > 0$, and run through all $\ell$ logic layers / time steps in one long continuous pass after the inputs to time step $t = 0$ are supplied. This is fine and it incurs no loss of privacy, since the application of the Unlockers to the originally

---

5 There is a further optimization to this algorithm wherein the final bits of the line encodings for 0 and 1 logical values are ensured to be different and are used together to select the appropriate truth table row; the “valid” field is then not required. This “point-and-permute” technique was introduced by Kolesnikov and Schneider in 2008 [4].

75
supplied inputs ensures that no alternate evaluation of the circuit can be probed; thus, every layer in the circuit remains unintelligible throughout the evaluation process.

The above ideas, of course, barely scratch the surface of the very wide variety of universal-circuit-based secure computing schemes that are possible; for example, see [21] for a more sophisticated scheme which may provide simpler constructions in practical cases, and which can likely be adapted for use in our framework.

Figure 9-4. **Simplified algorithm for garbling selector gates in first layer of interconnect fabric.**

This version takes advantage of the fact that only the externally-supplied inputs, not the internal states, need to be activated by the Unlocker. (a) Here we see a detail of the interface between two adjacent time steps, step #\((t-1)\) and step \(t\). Internal state lines \(S_{t-1}\), where \(1 \leq i \leq w\), carry the garbled internal state variables output from time step \(t-1\) to the lower input lines \(L_{t-1}^{2i}\) of the first layer of selector gates. Meanwhile, the input variables \(V_t^i\) for the current time step, \(1 \leq i \leq m \leq w\), feed the upper input lines \(L_{t-1}^{2i-1}\), except that (b) we pass their garbled values through an Unlocker, which effectively "decrypts" or activates them for use in the selector gates. This is done using the formula \(d_t^{ij} = h(k_t^{ij} \oplus k_t)\), where \(k_t^{ij} = K(V_t^{i}\oplus t)\), and \(k_t\) is a time-step unlock key as before. (c) In this example, the plain-text truth table for both selector gates’ function is the same as in Figure 9-3, and the garbled version (d) is also the same, except that the separate activation input \(a_i\) is no longer needed since the external input is already decoded.
9.1. Cost Analysis

A simple analysis was undertaken to estimate what the cost would be to deploy garbled universal circuits on Ethereum for evaluating several arbitrary examples of application circuits using the method described above. The EMP (Efficient Multi-Party) computation toolkit (EMP-TOOL)\(^6\) was used to synthesize a circuit for the two-bit Millionaire’s Problem, and additional circuits were obtained from Nigel Smart’s repository\(^7\) at Katholieke Universiteit Leuven. Results are shown in the table below.

<table>
<thead>
<tr>
<th>Netlist Name</th>
<th>2bit-millionaire</th>
<th>AES-non-expanded</th>
<th>adder_32bit</th>
<th>and_gate</th>
<th>sha-1</th>
<th>sha-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data for Original Circuit</td>
<td># gates</td>
<td>8</td>
<td>33,616</td>
<td>375</td>
<td>3</td>
<td>106,601</td>
</tr>
<tr>
<td></td>
<td># wires</td>
<td>12</td>
<td>33,872</td>
<td>439</td>
<td>5</td>
<td>107,113</td>
</tr>
<tr>
<td></td>
<td># inputs</td>
<td>4</td>
<td>256</td>
<td>64</td>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td># outputs</td>
<td>1</td>
<td>128</td>
<td>33</td>
<td>1</td>
<td>160</td>
</tr>
<tr>
<td>Layered Circuit Data</td>
<td>Depth (Logic Layers)</td>
<td>6</td>
<td>243</td>
<td>127</td>
<td>2</td>
<td>10,599</td>
</tr>
<tr>
<td></td>
<td>Width (Max Gates/Layer)</td>
<td>4</td>
<td>693</td>
<td>153</td>
<td>2</td>
<td>4,991</td>
</tr>
<tr>
<td></td>
<td>Gate Locs. (Depth x Width)</td>
<td>24</td>
<td>168,399</td>
<td>19,431</td>
<td>4</td>
<td>52,899,609</td>
</tr>
<tr>
<td></td>
<td>L.C. overhead (locs/gate)</td>
<td>3.00</td>
<td>5.01</td>
<td>51.82</td>
<td>1.33</td>
<td>496.24</td>
</tr>
<tr>
<td>Universal Circuit Data</td>
<td>Adjusted circuit width</td>
<td>4</td>
<td>1,024</td>
<td>256</td>
<td>2</td>
<td>8,192</td>
</tr>
<tr>
<td></td>
<td>Thompson network layers</td>
<td>9</td>
<td>41</td>
<td>33</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Routing elements/network</td>
<td>36</td>
<td>41,984</td>
<td>8,448</td>
<td>10</td>
<td>434,176</td>
</tr>
<tr>
<td></td>
<td>Total routing elements</td>
<td>216</td>
<td>10,202,112</td>
<td>1,072,896</td>
<td>20</td>
<td>4,601,831,424</td>
</tr>
<tr>
<td></td>
<td>U.C. overhead factor</td>
<td>54</td>
<td>607</td>
<td>5,722</td>
<td>13</td>
<td>86,337</td>
</tr>
<tr>
<td>Cost Estimates</td>
<td>Est. gas to deploy (million)</td>
<td>43</td>
<td>2,040,422</td>
<td>214,579</td>
<td>4</td>
<td>920,366,285</td>
</tr>
<tr>
<td></td>
<td>Est. ETH cost to deploy</td>
<td>0.043</td>
<td>2,040.42</td>
<td>214.58</td>
<td>0.004</td>
<td>920,366.28</td>
</tr>
<tr>
<td></td>
<td>Est. USD cost to deploy</td>
<td>$ 7.98</td>
<td>$ 376,702.78</td>
<td>$ 39,615.61</td>
<td>$ 0.74</td>
<td>$ 169,918,023.50</td>
</tr>
</tbody>
</table>

The first section of the table, “Data for Original Circuit,” gives some raw data such as logic gate counts for the original circuit netlists. (These circuits are not necessarily well optimized, but this is just to give us a starting point for analysis.)

To generate the data in the next table section, “Layered Circuit Data,” we wrote a simple tool to map each gate to the earliest possible logic layer and report the resulting circuit depth and maximum circuit width (in gates). No attempt was made to apply any logic transformations or other optimizations of the layer assignment to help minimize circuit width or depth. The charts in Figure 9-6 and Figure 9-5 outline the circuit structure for a couple of the examples (AES and SHA-256).

The “Universal Circuit Data” part of Table 9-1 was calculated as follows. The adjusted circuit width \(w\) rounds up the width to the nearest power of 2. The number of Thompson network layers is \(4 \log_2 w + 3\), and the number of routing elements per Thompson network is that times the adjusted width \(w\). The total number of routing elements is simply that times the depth of the application circuit. The UC overhead factor shows the factor of increase in the total number of gates (assuming here that all application gates are absorbed into the final routing layer for each time step).

---

\(^7\) See [https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html](https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html).
Finally, “Cost Estimates” are obtained as follows. If point-and-permute [4] is used to index truth table rows, each routing element requires storing an array of eight 256-bit words, which requires approximately 200,000 gas units, according to our empirical results from our test deployments of state machines for various Millionaire’s Problem examples as reported in §7.2. The estimated gas cost in the table is that required to deploy just the storage contracts (in millions of gas units) and ignores the cost for the Executor code itself, which is generally small in comparison (only around 1 million gas). Finally, the gas cost was converted to ETH and US dollars using price data sampled on April 23rd, 2020 (specifically, assuming paying only 1 Gwei/gas, at an Ether price of US$184.62 per ETH).

![Figure 9-6](image)

**Figure 9-6.** Profile of circuit width versus logic layer for the AES example circuit.
The traces show (blue) the number of regular logic gates (NOT/AND/XOR), (orange) the number of buffer gates (i.e., unchanging internal state variables), and (gray) the total number of circuit elements of both types, as a function of the logic-layer depth in the circuit. Simple circuit transformations might reduce the cost of this example somewhat, but probably not by more than a factor of ~2–3\(\times\).

![Figure 9-5](image)

**Figure 9-5.** Profile of circuit width versus logic layer for the SHA-256 circuit.
This particular circuit algorithm is memory-intensive, with the number of active gates at each logic layer being only a small fraction of the total circuit width. Simple transformations probably cannot reduce the cost of this example by much more than ~2\(\times\).
Although these are just rough calculations, and the circuits are not well optimized, a preliminary conclusion from this analysis is that the practical cost of this method, while at least not exponential, remains substantial for applications of a complexity requiring, say, general 32-bit integer arithmetic. However, a number of further optimizations can be explored which would improve the situation:

1. For some application contexts, the original target circuits could be much better optimized to minimize cost. For example, supposing that the input bits for a 32-bit add were supplied one at a time (like in our state machine for the Millionaire's Problem), and the addition is done serially, and if output bits are read out one at a time as well, then the circuit width, and therefore the overhead factor for the Thompson network, could be made much smaller.

2. The mapping of the circuit onto layers could also be designed to minimize cost.\(^8\)

3. It should also be possible (although it would likely require a substantial research effort) to migrate a greater amount of application logic (\textit{i.e.}, more than one layer) into the Thompson network, thereby reducing the number of computation steps required to embed the original application circuit.

4. Finally, more efficient universal circuit constructions could be explored.

It should be noted that the kinds of applications represented by the above circuits are not really ones that would be appropriate to run on GABLE to begin with. See §9.3 for a more realistic scenario.

### 9.2. Cost Comparison vs. Garbled Circuits without Functional Privacy

It's important to point out that almost all the cost of the examples above comes from our requirement for functional privacy, which forces us to take pains to obscure the circuit itself (gates and topology), as well as the data. This incurs substantial overhead for our universal circuit embedding. For an application that only requires data privacy, we can do much better. If the techniques described in \cite{4} are used to eliminate NOT and XOR gates, and the number of AND gates is minimized, then the estimated costs to deploy just the garbled AND gate data are shown in Table 9-2 (OR gates are not shown because they were converted to ANDs). Note that by abandoning the functional privacy requirement, both the size of the garbled representations of these example circuits and the cost of deploying them to the blockchain are reduced by factors ranging from 40× to over 300,000×.

<table>
<thead>
<tr>
<th>Netlist Name</th>
<th>2bit_millionaire</th>
<th>AES-non-expanded</th>
<th>adder_32bit</th>
<th>and_gate</th>
<th>sha-1</th>
<th>sha-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of AND gates in circuit</td>
<td>1</td>
<td>6,800</td>
<td>127</td>
<td>1</td>
<td>29,084</td>
<td>90,825</td>
</tr>
<tr>
<td>32-byte words to store garbled gates</td>
<td>4</td>
<td>27,200</td>
<td>508</td>
<td>4</td>
<td>116,336</td>
<td>363,300</td>
</tr>
<tr>
<td>Estimated gas required to deploy data</td>
<td>100,000</td>
<td>680,000,000</td>
<td>12,700,000</td>
<td>100,000</td>
<td>2,908,400,000</td>
<td>9,082,500,000</td>
</tr>
<tr>
<td>Estimated Ether cost to deploy (mETH)</td>
<td>0.10</td>
<td>680.00</td>
<td>12.70</td>
<td>0.10</td>
<td>2,908.40</td>
<td>9,082.50</td>
</tr>
<tr>
<td>Estimated USD cost to deploy data</td>
<td>$0.02</td>
<td>$125.34</td>
<td>$2.34</td>
<td>$0.02</td>
<td>$336.95</td>
<td>$1,676.81</td>
</tr>
<tr>
<td>Functional privacy overhead factor</td>
<td>432</td>
<td>3,001</td>
<td>16,896</td>
<td>40</td>
<td>316,451</td>
<td>35,697</td>
</tr>
</tbody>
</table>

\(^8\) For example, one can iterate through circuit widths (above the minimum) that are powers of 2, then minimize circuit depth for the given width, then choose the solution that offers the lowest overall cost.
9.3. Cost Comparison vs. State Machines for Auction Problem

One application that is simple enough (in relation to its utility) for blockchain-based garbled universal circuits to be viable is that of determining the winner(s) in a simple auction scenario (which we briefly mentioned in §4.2.2) with $B$ participants (bidders). Whoever submits the highest bid wins, and there can be multiple winners in case of a tie.

If all bid values are $L$ bits long, and if each participant supplies individual bits of their bid one at a time, most-significant bit (MSB) first, on consecutive time steps, then this can be accomplished using a state-updating circuit having $B$ bits worth of internal state (i.e. a circuit of width $w = B$ binary state variables between stages, or a monolithic state machine with $p = 2^B - 1$ states, where the $-1$ is there because a state with no winners is impossible, and can be omitted), using the following algorithm:

Call the internal state variables $W_i$, for $i = 0, ..., B - 1$. Variable $W_i$ will indicate the truth value of the proposition, “Bidder #i could still be a winner, given the partial bids seen so far.” Initially, we assign $W_i := 1$ (i.e., True) for all $i$; before we have seen any input bits, any player could still win.

On each time step $t$, for $t = 0, ..., L - 1$, each participant supplies input value $v_i(t)$ comprising bit $(L - 1) - t$ of their bid; that is, the bit in the $2^{(L-1)-t}$s place. At this point, one of two things may happen:

1. If there is any participant $i$ such that both $W_i = 1$ and $v_i(t) = 1$, then the state bits of all participants $i$ get updated as per $W_i := W_i \cdot v_i(t)$, where “$\cdot$” accomplishes logical AND.
2. Otherwise, the state is unchanged.

In either case, we then proceed to the next time step. See Figure 9-7 below for pseudocode.

This algorithm works because, since we are processing bits in big-endian order, as soon as any player neglects to supply a “1” bit on some cycle when some other potential winner has, that player is immediately eliminated from the set of possible winners (since even if they supplied “1” for all remaining bits, it wouldn’t make up for not supplying “1” on this round).

At the end of the algorithm, a Finisher can advance us to a new state in which we reveal to each participant only whether they themselves specifically were among the set of winners or not.

\begin{itemize}
  \item (Minimum) number of time steps $\ell = L$. (One per bit-position.)
  \item May need more time steps in GUC if we can’t fit the update circuit in one step.
  \item For time step index $t := 0, 1, ..., \ell - 1$, do:
    \begin{itemize}
      \item Let bit position $p := (L - 1) - t$; // MSB first
      \item For each bidder index $i := 1, 2, ..., B$, do:
        \begin{itemize}
          \item Get the value $b^i_p$ of the bit in position $#p$ for bidder #i.
          \item Let Boolean update flag $update? := \bigvee_{i=1}^B (W_i \land b^i_p)$; // State-updating phase
        \end{itemize}
      \item If $update? = 1$ (True) then:
        \begin{itemize}
          \item For each bidder index $i := 1, 2, ..., B$, do:
            $W_i \land= b^i_p$;
        \end{itemize}
    \end{itemize}
  \item Final outputs are \{W\}.
\end{itemize}

\textbf{Figure 9-7.} Pseudocode for bitwise multi-bidder auction algorithm. See text for discussion.
An explicit state-machine implementation of this algorithm will necessarily have high complexity due to the need for at least $2^B - 1$ internal states; further, each state will have $2^B$ outgoing arcs, to reflect the number of possible input configurations on each cycle. Thus, there will be $2^{2B} - 2^B$ arcs for each time step. Note this ignores any additional complexity overhead required for reconvergent arc elimination, although that problem can be avoided easily by adding Unlockers to the protocol as we described for the circuit model earlier in §9.

### Table 9-3. Complexity and Cost of Monolithic State Machines for Simple Auctions


<table>
<thead>
<tr>
<th>Bidders</th>
<th>States/cycle</th>
<th>Arcs per state</th>
<th>Arcs per cycle</th>
<th>Contract data words per cycle</th>
<th>Gas/cycle (millions)</th>
<th>Cost/cycle (USD)</th>
<th>Cost/ bidder/ bit (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>24</td>
<td>0.6</td>
<td>$0.10</td>
<td>$0.05</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>56</td>
<td>112</td>
<td>2.8</td>
<td>$0.48</td>
<td>$0.16</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>16</td>
<td>240</td>
<td>480</td>
<td>12.0</td>
<td>$2.07</td>
<td>$0.52</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>32</td>
<td>992</td>
<td>1,984</td>
<td>49.6</td>
<td>$8.58</td>
<td>$1.72</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
<td>64</td>
<td>4,032</td>
<td>8,064</td>
<td>201.6</td>
<td>$34.85</td>
<td>$5.81</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
<td>128</td>
<td>16,256</td>
<td>32,512</td>
<td>812.8</td>
<td>$140.52</td>
<td>$20.07</td>
</tr>
<tr>
<td>8</td>
<td>255</td>
<td>256</td>
<td>65,280</td>
<td>130,560</td>
<td>3,264.0</td>
<td>$564.31</td>
<td>$70.54</td>
</tr>
<tr>
<td>9</td>
<td>511</td>
<td>512</td>
<td>261,632</td>
<td>523,264</td>
<td>13,081.6</td>
<td>$2,261.68</td>
<td>$251.30</td>
</tr>
<tr>
<td>10</td>
<td>1,023</td>
<td>1,024</td>
<td>1,047,552</td>
<td>2,095,104</td>
<td>52,377.6</td>
<td>$9,055.56</td>
<td>$905.56</td>
</tr>
<tr>
<td>11</td>
<td>2,047</td>
<td>2,048</td>
<td>4,192,256</td>
<td>8,384,512</td>
<td>209,612.8</td>
<td>$36,239.96</td>
<td>$3,294.54</td>
</tr>
<tr>
<td>12</td>
<td>4,095</td>
<td>4,096</td>
<td>16,773,120</td>
<td>33,546,240</td>
<td>838,656.0</td>
<td>$144,995.24</td>
<td>$12,082.94</td>
</tr>
<tr>
<td>13</td>
<td>8,191</td>
<td>8,192</td>
<td>67,100,672</td>
<td>134,201,344</td>
<td>3,355,033.6</td>
<td>$580,051.76</td>
<td>$44,619.37</td>
</tr>
<tr>
<td>14</td>
<td>16,383</td>
<td>16,384</td>
<td>268,419,072</td>
<td>536,838,144</td>
<td>13,420,953.6</td>
<td>$2,320,348.67</td>
<td>$165,739.19</td>
</tr>
<tr>
<td>15</td>
<td>32,767</td>
<td>32,768</td>
<td>1,073,709,056</td>
<td>2,147,418,112</td>
<td>53,685,452.8</td>
<td>$9,281,677.93</td>
<td>$618,778.53</td>
</tr>
<tr>
<td>16</td>
<td>65,535</td>
<td>65,536</td>
<td>4,294,901,760</td>
<td>8,589,803,520</td>
<td>214,745,088.0</td>
<td>$37,127,278.26</td>
<td>$2,320,454.89</td>
</tr>
</tbody>
</table>

In contrast, a GUC (Garbled Universal Circuit) implementation of the same state machine can be much more compact. Each step of the algorithm requires the following logic operations:

1. $B$ AND gates, to check each participant for the condition $W_i = 1$ and $v_i(t) = 1$.
2. $(B - 1)$ OR gates, to see if the above condition was true for any participant.
3. Another $B$ AND gates (with one input negated), to allow each participant’s state bit $W_i$ to pass through unchanged if none of the possible winners supplied a ‘1’ bit.
4. Another $B$ OR gates, to combine results from 1 and 3.

See Figure 9-8 for a sketch of the circuit for the 3-bidder case. Thus, we need a total of $4B - 1$ two-input gates. It seems unlikely that these can all be moved into the Thompson network. For now, we will just assume conservatively that each layer of the circuit will require a separate GUC step (Thompson net). Table 9-4 below shows the cost to deploy the garbled data for this machine. Note that the cost per bidder per bit is less than US$30 in all cases up to 16 participants.

We can compare results between Table 9-3 and Table 9-4. See the chart in Figure 9-9 on p. 83.

For perspective, Table 9-5 estimates the cost for running auctions on garbled universal circuits for larger numbers of bidders. Note that the cost per bidder per bit scales up only polylogarithmically, and even for very large auctions with thousands of bidders is still under US$150.00.
In summary, for certain problems such as simple simultaneous auctions (as in §4.2.2) in which the cost to run the computation can be amortized over potentially large numbers of users, the universal garbled circuits approach can be reasonably feasible.

**Figure 9-8. Application circuit example for multi-bidder auction algorithm for \( B = 3 \) bidders.**

The \( B \)-input NOR gate at center can be transformed into a depth-\( \lceil \lg B \rceil \) binary tree of \( B - 1 \) two-input OR gates, with the output feeding bubbled inputs of the AND gates in the next layer.

In summary, for certain problems such as simple simultaneous auctions (as in §4.2.2) in which the cost to run the computation can be amortized over potentially large numbers of users, the universal garbled circuits approach can be reasonably feasible.

**Table 9-4. Complexity and Cost of Garbled Universal Circuits for Small Auctions**


<table>
<thead>
<tr>
<th>Bidders</th>
<th>B-way NOR comp. steps</th>
<th>NOR circ. width (gates)</th>
<th>Tot. circuit width (gates)</th>
<th>Adj. circ. width (gates)</th>
<th>Rout'g layers/comp. step</th>
<th>Rout’g elems./comp. step</th>
<th>Comp. steps/app. cycle</th>
<th>256-bit words/app. cycle</th>
<th>Gas/app. cycle (millions)</th>
<th>Cost/app. cycle (USD)</th>
<th>Cost/bidder/bit (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>104</td>
<td>4</td>
<td>3,328</td>
<td>83.2</td>
<td>15.36</td>
<td>7.68</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>104</td>
<td>5</td>
<td>4,160</td>
<td>104.0</td>
<td>19.20</td>
<td>6.40</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
<td>16</td>
<td>17</td>
<td>272</td>
<td>5</td>
<td>10,880</td>
<td>272.0</td>
<td>50.22</td>
<td>12.55</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>272</td>
<td>6</td>
<td>13,056</td>
<td>326.4</td>
<td>60.26</td>
<td>12.05</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>272</td>
<td>6</td>
<td>13,056</td>
<td>326.4</td>
<td>60.26</td>
<td>10.04</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>17</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>6</td>
<td>32,256</td>
<td>806.4</td>
<td>148.88</td>
<td>21.27</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>20</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>6</td>
<td>32,256</td>
<td>806.4</td>
<td>148.88</td>
<td>18.61</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>22</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>7</td>
<td>37,632</td>
<td>940.8</td>
<td>173.69</td>
<td>19.30</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>25</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>7</td>
<td>37,632</td>
<td>940.8</td>
<td>173.69</td>
<td>17.37</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>27</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>7</td>
<td>37,632</td>
<td>940.8</td>
<td>173.69</td>
<td>15.79</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>30</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>7</td>
<td>37,632</td>
<td>940.8</td>
<td>173.69</td>
<td>14.47</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>32</td>
<td>32</td>
<td>21</td>
<td>672</td>
<td>7</td>
<td>37,632</td>
<td>940.8</td>
<td>173.69</td>
<td>13.36</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>35</td>
<td>64</td>
<td>25</td>
<td>1,600</td>
<td>7</td>
<td>89,600</td>
<td>2,240.0</td>
<td>413.55</td>
<td>29.54</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>37</td>
<td>64</td>
<td>25</td>
<td>1,600</td>
<td>7</td>
<td>89,600</td>
<td>2,240.0</td>
<td>413.55</td>
<td>27.57</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>40</td>
<td>64</td>
<td>25</td>
<td>1,600</td>
<td>7</td>
<td>89,600</td>
<td>2,240.0</td>
<td>413.55</td>
<td>25.85</td>
<td></td>
</tr>
</tbody>
</table>
Table 9-5. Complexity and Cost of Garbled Universal Circuits for Large Auctions


<table>
<thead>
<tr>
<th>Bidders</th>
<th>Total circuit width (gates)</th>
<th>Adj. circ. width (gates)</th>
<th>Rout’g layers/ comp. step</th>
<th>Rout’g elems./ comp. step</th>
<th>Comp. steps/ app. cycle</th>
<th>256-bit words/ app. cycle</th>
<th>Gas/ app. cycle (millions)</th>
<th>Cost/ app. cycle (USD)</th>
<th>Cost/ bidder/ bit (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>80</td>
<td>128</td>
<td>29</td>
<td>3,712</td>
<td>8</td>
<td>237,568</td>
<td>5,939.2</td>
<td>$1,096.50</td>
<td>$34.27</td>
</tr>
<tr>
<td>64</td>
<td>160</td>
<td>256</td>
<td>33</td>
<td>8,448</td>
<td>9</td>
<td>608,256</td>
<td>15,206.4</td>
<td>$2,807.41</td>
<td>$43.87</td>
</tr>
<tr>
<td>128</td>
<td>320</td>
<td>512</td>
<td>37</td>
<td>18,944</td>
<td>10</td>
<td>1,515,520</td>
<td>37,888.0</td>
<td>$6,994.88</td>
<td>$54.65</td>
</tr>
<tr>
<td>256</td>
<td>640</td>
<td>1,024</td>
<td>41</td>
<td>41,984</td>
<td>11</td>
<td>3,694,592</td>
<td>92,364.8</td>
<td>$17,052.39</td>
<td>$66.61</td>
</tr>
<tr>
<td>512</td>
<td>1,280</td>
<td>2,048</td>
<td>45</td>
<td>92,160</td>
<td>12</td>
<td>8,847,360</td>
<td>221,184.0</td>
<td>$40,834.99</td>
<td>$79.76</td>
</tr>
<tr>
<td>1024</td>
<td>2,560</td>
<td>4,096</td>
<td>49</td>
<td>200,704</td>
<td>13</td>
<td>20,873,216</td>
<td>521,830.4</td>
<td>$96,340.33</td>
<td>$94.08</td>
</tr>
<tr>
<td>2048</td>
<td>5,120</td>
<td>8,192</td>
<td>53</td>
<td>434,176</td>
<td>14</td>
<td>48,627,712</td>
<td>1,215,692.8</td>
<td>$224,441.20</td>
<td>$109.59</td>
</tr>
<tr>
<td>4096</td>
<td>10,240</td>
<td>16,384</td>
<td>57</td>
<td>933,888</td>
<td>15</td>
<td>112,066,560</td>
<td>2,801,664.0</td>
<td>$517,243.21</td>
<td>$126.28</td>
</tr>
<tr>
<td>8192</td>
<td>20,480</td>
<td>32,768</td>
<td>61</td>
<td>1,998,848</td>
<td>16</td>
<td>255,852,544</td>
<td>6,396,313.6</td>
<td>$1,180,887.42</td>
<td>$144.15</td>
</tr>
</tbody>
</table>

9.4. Conclusions on Cost

Some final remarks to conclude our overall discussion of cost in this section: We acknowledge that of course, the expected real-world Ethereum gas cost for some of the examples illustrated in §9.1 & §9.3 above are exorbitantly large. Our presentation of these estimated costs should not be taken to

![Finite State Machine vs. Garbled Universal Circuit Cost on Auction Problem](image)

Figure 9-9. Semi-logarithmic chart comparing costs of FSM vs. GUC approaches for auctions.

The horizontal axis shows the number of bidders \( B \); the log-scale vertical axis shows the estimated cost in dollars to deploy the auction contract, per bit in the length of the bid values. For example, if we wanted to support bid prices up to $1,000,000, with a $1,000 price granularity, we would need 10 bits in the bid values, and the total cost to operate the auction would be 10× the level shown. E.g., if there were 10 bidders, the total cost to run the auction in garbled fashion on Ethereum would be $96,700 if we had to use the state machine model, but only $1,737 in the universal circuit model, for a cost savings of 98.2%.
suggest or imply that corresponding deployments would necessarily be recommended. Rather, for practicality considerations, we propose that potential users engage in the following thought process: For any multi-party application requiring functional privacy, one should consider the overhead cost of deploying (agreeing upon, purchasing, networking, maintaining, etc.) a dedicated, resilient population of compute nodes (whether cloud-based, or physical) upon which to perform the desired computation, and weigh that against the cost of using an approach like GABLE’s.

For cases in which SMC features are desired but our approach is exorbitantly expensive, a more prudent near-term choice (in lieu of possible further improvements in SMC and garbled smart contract efficiency if R&D in this area continues) will likely rather be to procure and deploy physical compute nodes running a state-of-the-art, non-blockchain-based SMC protocol, assuming that the “per-cycle cost” of traditional SMC (largely consisting of power/cooling) is significantly lower than the cycle cost of blockchain-based deployments, even when accounting for the amortized cost of any initial capital investment in the compute infrastructure.

However, our approach may nevertheless have advantages over the “deploy your own infrastructure” approach in areas such as disaster resilience/survivability (e.g., Ethereum is extremely survivable, whereas the survivability of “deploy your own” depends on the resources invested in redundant datacenters, etc.) and transparency (our approach should make certain post-haste verifiability approaches more believable, due to the immutability of the Ethereum blockchain). As further efficiency improvements are applied to our, and related, approaches, a wider range of applications under such schemes may eventually become more competitive with “deploy your own” SMC infrastructure, and may come to look increasingly attractive under requirements for sufficiently high survivability, etc.
10. RELATED WORK

In this section, we briefly review a sample of the relevant work in several subfields of cryptography that relate to secure computation, and compare and contrast the overall setup (in terms of the overall security model and requirements) and features achieved by our framework with those of the existing work in those fields. Please note that we are not here attempting to carry out a comprehensive review of each subfield that we mention.

10.1. Secure Multi-Party Computation

The study of Secure Multi-party Computation (SMC, or more often MPC) began with the seminal work of Yao [8] in 1982, in which he considered the example of two parties indexed \( i = 1, 2 \), each of which knows a number \( x_i \), and the parties wish to jointly determine whether \( x_1 < x_2 \) without revealing to each other any other information about the numbers held. This is the “Millionaires’ Problem,” which we already saw a version of in Sec. 4.1.1. It is straightforwardly generalizable to a case with \( N \geq 2 \) parties, and any arbitrary computable function \( f(x_1, \ldots, x_N) \) of the data held by each party.

In general, MPC refers to any such secure computation setup in which a group of parties who do not necessarily fully trust each other are nevertheless able to compute such a function \( f \) of their respective inputs and mutually learn the output, without learning anything else about each other’s inputs.

Secure Multi-party Computation does not require a trusted central authority to collect inputs, perform calculations, or distribute outputs. Computation is performed by one or more of the cooperating parties. Thus, the influence of dishonest parties is a concern. A “secure” MPC scheme ensures that a dishonest actor cannot affect the outcome of the scheme in a way which cannot be done in a supervised scenario with a trusted central authority [22]. This security model is known as the ideal/real simulation paradigm.

Yao’s solutions to the Millionaires’ Problem [8] are similar to key exchange schemes, where two parties exchange information dependent on private values (but not revealing any information about those values) in order to end up with a shared secret. One of these solutions depends on the use of one-way functions, functions which are easy to compute but whose inputs are difficult to recover. One-way functions, also used in key exchange schemes, provide for privacy between parties as well as defense against unauthorized parties intercepting transactions. Standard cryptographic hash functions, such as the \( h(\cdot) \) function invoked in our GABLE framework (§5.2) are examples of functions that are, at least, widely believed to be one-way.

Since [8], Yao also introduced garbled circuits [23], which provide a way to address SMC for general functions, not just specific computations like whether \( x_1 < x_2 \) as in the Millionaires’ Problem. In one version, Alice acts as the executor of the scheme. For a given gate, she generates two keys for each input and output wire, and builds a garbled computation table, where the keys associated with the possible outputs are encrypted with the keys for the corresponding inputs. To share information and mutually calculate the outcome, a key-exchange-like transaction is performed. This can be facilitated by oblivious transfer [24]. Our technique described in this report (§§5.2-5.3) for generating and interpreting garbled state machines is similar to, and descended from Yao’s garbled circuit methods. Our technique for garbling universal circuits, described in §9, is even more closely related.

Additional methods for computing arbitrary functions using SMC have been developed over the years, attempting to address security against malicious adversaries (e.g., [25], [26]). SMC can be applied in situations where the user of a computational tool does not want to reveal personal data to the service provider (this essentially the use case that we wish to be solved with all of the techniques
described in this section). Research in SMC is being used in a variety of applications, including secure databases [27], machine learning on cloud services [28], and data mining [29].

The similarities between our setup and the standard ones for SMC include:

1. The result of the computation can be arranged to become common knowledge among all the input providers, while not also becoming available to arbitrary outside parties. In our approach, this can be done, for example, by giving each of the input providers a key that unlocks a commitment in the smart contract as to what is the unique spectator key that allows obtaining the mutually visible output of the computation.

2. In the process of running the Machine, no information whatsoever about the input values provided by each participant is revealed to any of the other participants, other than what is directly implied by whatever result from the computation is revealed to each participant.

There are a few differences, however, between our problem definition, and traditional ones for SMC:

1. In our setup, the input providers are not necessarily enabled to learn the result of the computation (but meanwhile, other parties can be, and the Company itself, at minimum, is). Normally in SMC, the purpose of the computation is solely to provide mutual knowledge of the computational result to the participants. In our setup, we can, if we wish, enable the input providers to also learn the result of the computation, but we do not have to do so.

2. In our setup, the input providers are not necessarily even made aware of the nature of the computation that is being performed. Normally in SMC, the input providers are cooperating to compute a function that is known to them. In our setup, the input providers are indeed still cooperating to compute a function, but (in our base protocol, where they only receive a subset of keys) it is impossible for them to know exactly what function is being computed. They must just trust that the overall function being computed has whatever properties they expect—although it’s possible for the computation to be verified later; see §10.5. (However, they can, at least, obtain mutual information about whatever computational result they are all given.)

Note that, however, it would be possible to allow the input providers to learn what the computation is by simply giving them the entire garbled circuit including all I/O keys, but in that case, they could infer the other providers’ input values as well, and so the whole system would then no longer be providing an SMC setup, per se. In this case, the whole system would then effectively be more closely akin to a sort of authenticated VPN between the input providers, one that allows them to privately execute whatever protocol the Machine facilitates without arbitrary outside parties snooping.

3. In our setup, the entity that originally ran the Garbler (the Company) is enabled to read all inputs and outputs from the computation (at least, if that entity can view the protocol messages exchanged), and it also is the only entity that knows for certain what computation the Machine actually does (assuming that the full garbled circuit is not given out to the other participants). Normally in typical SMC setups, only the input providers can know what their individual inputs were, but they all clearly know what computation they are cooperating to carry out. Other parties not involved in the protocol would not necessarily be enabled to learn the inputs, the outputs, or what computation is being performed.

10.2. Homomorphic Encryption

Homomorphic encryption (HE) is a paradigm for secure computation in which a computation can be performed on encrypted data to produce the correct encrypted result, without having to decrypt the
input data. Its most general and powerful variant is fully homomorphic encryption (FHE), in which the computation that is performed may be any arbitrary computation. Craig Gentry proposed a scheme for FHE based on lattices in 2009 [9], which we describe in more detail below.

10.2.1. Brief Review of FHE

The term homomorphic here comes from the field of abstract algebra, in which a homomorphism is a function \( f : X \rightarrow Y \) which preserves a binary operation. For example, if addition is the binary operation of interest in both \( X \) and \( Y \), then \( f(a + b) = f(a) + f(b) \) for \( a, b \) in \( X \). In the context of encryption, it can be beneficial to have an encryption function which is homomorphic with respect to arithmetic operations. If this is the case, then we can perform operations on encrypted data, which when decrypted, will result in the same operations on plaintext data. A classic use case for this kind of encryption is cloud computing. A client storing encrypted data on the cloud can request the server to perform a computation on the data without revealing the plaintext. The client can then decrypt the manipulated data and receive the plaintext with the same operations applied.

RSA encryption [30], a popular public-key cryptosystem, provides one example of a function which is homomorphic with respect to multiplication. Say we are encrypting messages \( a, b \) with the RSA encryption function \( E \), defined by \( E(m) = m^e \mod N \) for arbitrary message \( m \), public key \( e \), and modulus \( N \). Then

\[
E(a \cdot b) = (a \cdot b)^e \mod N = a^e \cdot b^e \mod N = (a^e \mod N) \cdot (b^e \mod N) = E(a) \cdot E(b).
\]

Similarly, decryption is also homomorphic with respect to multiplication. Thus, one can perform multiplication on encrypted data which corresponds to multiplication on the plaintext, without needing to decrypt the data.

There are other encryption functions which are homomorphic with respect to addition (e.g., [31]). The “holy grail” of cryptography [32] is described by some to be a “fully homomorphic” encryption function which is homomorphic with respect to both multiplication and addition. This would allow for arbitrary mathematical operations on encrypted data (evaluation of arbitrary circuits) to preserve the structure of operations on plaintexts.

Initial research produced cryptographic functions homomorphic with respect to both addition and multiplication, but which do not allow for an arbitrary number of each operation (e.g., [33]). This kind of scheme is known as somewhat homomorphic encryption (SWHE). Gentry [9] introduced a framework for FHE with arbitrary operations using lattice-based cryptography, and a technique called bootstrapping. Here we give a broad overview of Gentry’s method.

In lattice-based encryption, a ciphertext is an element of a polynomial ring of the form \( c = as + e \), where \( e \) is an “error vector” with small coefficients. Encryption is homomorphic with respect to multiplication and addition, which take place in this polynomial ring. However, lattice-based encryption is SWHE, because the error vector \( e \) gets larger with each additional addition and multiplication on encrypted data, which leads to decryption errors (the error term must be small in order for approximation of the plaintext to succeed with high probability). Gentry aims to “refresh” the ciphertext periodically to create a new ciphertext which has a shorter error vector but still decrypts correctly. The basic setup for this is as follows. Here, the notation \( E(k, m) \) indicates encryption of a message \( m \) using key \( k \), while \( D(k, c) \) is decryption of a ciphertext \( c \) using key \( k \). In homomorphic encryption, if a scheme can evaluate a circuit on ciphertexts homomorphically, the result will decrypt to the same circuit applied to plaintext data. In other words, if evaluating a circuit \( C \) on ciphertext vector \( \psi = \)
\(E(pk, m)\) produces \(\psi'\), then \(C(m) = D(sk, \psi')\). In bootstrapping, we evaluate the (somewhat modified) decryption circuit on ciphertexts. We start with a public-secret key pair \(pk_1, sk_1\), a plaintext \(\pi\), and ciphertext \(\psi_1 = E(pk_1, \pi)\). This ciphertext has some error vector \(x_1\), which we want to refresh. With a newly generated public-private key pair \(pk_2, sk_2\), calculate \(sk'_1 = E(pk_2, sk_1)\) and \(\psi'_1 = E(pk_2, \psi_1)\). Then evaluate the decryption circuit \(D\) on ciphertexts \((sk'_1, \psi'_1)\) to produce another ciphertext \(\psi_2\). This has its own error vector \(x_2\). Then, by the properties of homomorphic encryption, the decryption circuit applied to \((sk_1, \psi_1)\) (i.e., the plaintext \(\pi\)) should be equal to \(D(sk_2, \psi_2)\). As long as the error vector \(x_2\) on the “refreshed” ciphertext \(\psi_2\) is smaller than the original \(x_1\), we have removed some noise from the ciphertext.

Research since the publication of Gentry’s theoretically practical FHE scheme has focused on increasing the efficiency and improving practical implementations of FHE.

10.2.2. Comparison with FHE

Our approach in GABLE can be considered to provide a method, albeit (in the state-machine model) a very inefficient one, for satisfying some of the security requirements of FHE. The inefficiency arises from the fact that the explicit state-machine model of computation requires an exponentially large amount of space to represent the state graph, compared to computation models of standard power, such as RAM machines [34]. Thus, that approach would not be feasible for homomorphically encrypting general computations of larger sizes. However, for very simple computations, it does indeed provide full encryption of inputs, intermediate states, and outputs; and also, at no point in the computation process does a “plain-text” representation of any meaningful information about the computational state need to be revealed (to protocol nodes that are actually performing the computation). So, our method provides a security feature similar to FHE, in that respect.

Further, in a universal circuits (u.c.) based implementation of GABLE such as described in §9, the exponential inefficiency goes away, and is replaced by a merely logarithmic multiplicative overhead factor. This level of overhead may be considered competitive with other methods for some applications of FHE.

However, our setup also differs from FHE in the sense that we have an extra entity, separate from the one that provided the input (namely, the “Company” entity which originally ran the Garbler) that would have the ability to decrypt the input, intermediate states, and output of the computational process. It can also delegate the ability to decrypt subsets of this information to other parties. Thus, the overall security model here is rather different from the one usually considered in FHE, in which only the entity supplying the input would have the ability to decrypt the input, the intermediate states and the output.

10.3. Indistinguishability Obfuscation

Indistinguishability obfuscation (IO) (cf., e.g., [10]) is a secure-computation paradigm in which executable program representations are obfuscated, or transformed unintelligibly to such an extreme extent that it becomes impossible to tell which of the similarly-sized circuits for computing a given function was the one that was obfuscated—in other words, the representation reveals nothing about the algorithm other than its size, and the overall input-output relation that it computes (which can be sampled by running the obfuscated program). Furthermore, given that the only information that can be inferred about the original program or circuit is its input-output relation, and that the I/O relation itself can only be determined by running the program on sample inputs, it therefore becomes impossible to predict what the output of the program would be on inputs that it has not been run on yet—at least,
to the extent that there exists some program of the given size that could produce a given possible output. (As we consider more complex functions, at some point it would become impossible to represent them with an obfuscated circuit of a given size—although verifying whether this is the case would generally be uncomputable.)

In more detail: In IO, an obfuscator takes a circuit or program and produces a functionally equivalent circuit which is “unintelligible.” Functionally equivalent means that the obfuscator computes the same function as the original circuit, producing identical input-output pairs. It is the details of the computation which are hidden. IO could be used to protect software, for example, by obscuring the program at work, and preventing unauthorized functional modification of the software. (Since the obfuscated version of the program is unintelligible, any attempt to modify it will break it completely.)

A related concept is that of reusable garbled circuits [35], which allows for the use of a garbled circuit on more than one input. In the traditional garbled circuit model, which we mentioned in §10.1, encoding more than one input vector for a garbled circuit weakens its security. (The same is true if multiple input sequences are provided to the garbled state machines we describe in §5.) In contrast, if a multi-use garbled circuit is constructed, it can act as an obfuscated circuit, where the inputs are encoded by the data owner (these encoded inputs are known as “tokens”).

IO started to become more practical when [36], [10] introduced a possible method of tying together functional encryption and obfuscation to obtain reachable security assumptions. Since then, research has included applying IO to problems in cryptography [37], such as deniable encryption [38].

Our method, as with the classic (non-reusable) garbled circuits from which it is derived, does not, strictly speaking, implement IO. One reason for this is that it would be theoretically possible (albeit infeasibly time-consuming) for an attacker to simply enumerate all possible input keys, run them through the Machine to find all valid state sequences, and thereby map out the complete garbled state graph. This could then be directly distinguished from other similarly sized but topologically distinct state graphs that compute the same function, i.e., that happen to be equivalent in the sense that they arrive at the same final output state for all possible input streams.

Even on a more practical level, if input providers collude with each other (e.g., by sharing input keys), they could, relatively quickly, map out the possible behaviors of the Machine over the set of valid input values that they possess. In such a scenario, determining the full machine behavior would be relatively feasible, requiring a number of runs only on the order of the complexity of the state machine. (At most only one run is needed per arc that exists in the machine, to traverse that arc.)

However, if input providers do not collude, there is no practical way that a single input provider can, by themselves, map out the possible behaviors of the machine. Even after having seen sample inputs from another provider for a single run, there is no way to infer what the behavior of the machine would have been if any of the other providers’ inputs had been different. So, this property of our scheme can be considered to provide a (very limited) form of IO. However, if multiple runs with different input sequences can be observed, then it becomes possible again for an input provider to infer structural information about the state machine (e.g., that a given machine state can be reached by either of two different paths).

Unfortunately, there is, in general, no way that the Company can know that the input providers (if they are arbitrary entities who could communicate with each other) are not colluding with each other secretly to de-obfuscate the Machine, so, the above observation only provides a very weak and limited form of IO, which only applies if the input providers are under strict control and/or can be completely blocked from communicating with each other via any channels other than by sending messages to a
(single) actual deployed instance of the Machine on a specific published blockchain—since, only in that case, if the providers did attempt to collude to de-obfuscate the Machine behavior (by, for example, providing inputs to additional copies of the Machine which they themselves re-deployed on the same blockchain), the Company would, at least, be able to see this happening on that chain.

10.4. Functional Encryption

Functional encryption (FE) [39], [40], [11] is a paradigm for secure computation in which a participant is enabled to learn (in plaintext) only a selected function of some encrypted data, but nothing else about the unencrypted form of the data.

10.4.1. Brief Review of FE

In a typical functional encryption scheme, a central authority (which could be an authorized user) holds a master secret key \( m_k \) and public key \( p_k \) pair. The authority can use \( m_k \) to generate a secret key \( s_k_F \) which corresponds to some function \( F \). This function can be applied to a plaintext message \( m \). Using \( p_k \), the authority encrypts the plaintext to produce a ciphertext \( c \). Any entity which holds \( s_k_F \) can decrypt \( c \) to reveal the result \( F(m) \) of the function applied to \( m \).

For example, public-key encryption can be considered a very simple example of FE. The secret key corresponding to \( p_k \) can decrypt \( c \) to reveal the full plaintext message. In this case, the secret key corresponds to the function \( F(x) = x \). Another example is in cloud computing. A client stores encrypted data on a cloud service and wishes the cloud to perform some calculation on the data without decrypting it (this is a typical use case for homomorphic encryption). Using FE, the client may distribute a key \( s_k_F \) to the cloud which corresponds to a function \( F(x) \) of all data \( x \) which satisfies a certain condition (example referenced in [41]).

The functionality of an FE scheme can be randomized, where a random sample from an output distribution is selected [42], [43]. As an example, this can be useful if a financial institution wants to give an auditor a random sample of financial records, without revealing all customers' data and without cherry-picking the accounts to present for audit.

The notion of FE has been extended to allow for functions of \( n \) plaintexts given \( n \) ciphertexts. This is called multi-input functional encryption (MIFE) [44]. The output in this scheme may utilize the same secret key. A similar notion is multi-client functional encryption (MCFE), in which multiple parties may each produce a ciphertext, and the set of ciphertexts are decrypted to a function of the plaintexts [44]. Recently, research has taken place to extend MCFE to a decentralized model, where there is not a single authority holding the master secret key [45].

10.4.2. Comparison with FE

Our scheme does allow for a form of functional encryption, at least of small datasets in multi-party settings. For example, with two input providers, who are given respective sets of input keys, and who take turns providing inputs to the Machine over the course of a single run, an arbitrary function of the final state can be made available to one of the providers, by simply giving that provider a spectator key, which they can hash together with the final state ID to unmask an arbitrary output string associated with the final state, which can be provided (in XOR-masked form) in the \( \text{Exec}[G] \) code. The main limitation of this approach to functional encryption, in the state-machine approach, is simply our usual limitation, namely that the worst-case space complexity of that particular method grows exponentially with the number of bits of machine state, so that only very simple functions of very small.
amounts of encrypted input data are feasible to compute in that way. However, if a circuit model is used instead, as we described in §9, then this limitation goes away, and more complex functions on larger encrypted datasets become feasible.

10.5. Verifiable Computation

Verifiable computation or verifiable computing [12] refers to computation for which the correctness of the computation can be verified by a party other than the one that originally performed the computation. Any blockchain-based computation is inherently verifiable, in the trivial sense that any node that accepts a block that includes a given smart contract transaction is supposed to be able to verify that the transaction was properly executed, as per the instructions in the literal bytecode for the contract—this can, in general, be done by simply re-executing the transaction code in the node's own local space.

In our case, this trivial verifiability property holds as well, although part of the point of our scheme is that the meaning of the executed computation need not be discernable to parties other than the Company that ran the Garbler originally. However, that party can indeed interpret the computation recorded on the blockchain and verify that it correctly reflects the intended execution of the original state machine $F$. Further, the Company can give any other party the ability to carry out this same verification process, by, for example, giving them the complete output of the Garbler.

Alternatively, if the originally-posted smart contract that was executed was digitally signed by the Company, together with commitments to public output keys which are unsealed (by the Company or their delegate, such as a Finisher) upon program completion, then this gives a way for users to at least verify that the result of the computation indeed correctly reflects the Company's prior intent, without necessarily giving away the state machine's detailed function.
11. CONCLUSIONS AND FUTURE WORK

In this document, we have described the conceptual technical principles behind, and a detailed, working prototype implementation and multiple demos of, the GABLE system for autonomously executing garbled versions of simple finite state machines (and in the future, more complex circuits) on public programmable blockchains, such as Ethereum, to obtain a high degree of reliability, availability, and trust. Further, the design of GABLE provides extremely strong privacy guarantees: Assuming only that (1) participants do not release their delegated keys, or otherwise collude to defeat the system, and (2) that the hashes and random keys utilized satisfy standard cryptographic assumptions, it is impossible for any unauthorized parties to feasibly obtain any meaningful prior information whatsoever about the target application’s functionality, apart from an upper bound on its complexity, and no meaningful information during or after its execution either, except that its state was updated at certain times in response to (undecipherable) inputs from parties which may be associated with certain public keys (if such are used to sign input messages). Nevertheless, authorized parties may obtain any outputs from the machine (at any point in its execution) that the Machine’s creator wishes to enable them to obtain.

Our framework combines certain security features of several related secure computing paradigms, such as homomorphic encryption (operating on encrypted data), indistinguishability obfuscation (hiding of algorithmic details), and secure multiparty computation (mutual privacy of inputs between participants, no need for a trusted third party after setup, ability to produce shared knowledge of output). In addition, it is reasonably computationally efficient both to generate and to interpret the garbled machine, either as a function of the explicit state machine’s complexity, or, in a future circuit-based version, as a function of circuit complexity. The main limitation of the existing state-machine based prototype is simply that explicit state machines, themselves, do not comprise a space-efficient representation of computation to begin with, since a general computation operating on an $n$-bit memory requires $\Theta(2^n)$ states to represent in a fully explicit state-machine form, in the worst case. However, for computations with very limited memory requirements (operating on a small handful of bits, say), the existing method is quite feasible, and further, it is straightforward to extend our methods to handle somewhat more complex computations using a circuit-based approach, as described in §9.

Some useful and/or interesting directions for future work include:

1. Prototyping the circuit-based version of the system described in §9, or some related approach offering similar features. (The latter has already begun in the GOOSE effort mentioned in §7.)

2. Exploring a wider variety of input models in prototype implementations.

3. Creating more sophisticated compilation frameworks to facilitate translating higher-level application codes into garbled machine instances (beyond what had been envisioned in §8).

4. Implementing some more substantial example applications (such as the Dungeon Race game example that we discussed briefly in Sec. 4.1.2).

We welcome public feedback on the material in this report, and we look forward to continuing to share the ideas explored in this project with the wider research community in future publications.
REFERENCES


APPENDIX A  TABLES OF NOTATIONS

The following tables list the mathematical symbols and notations used in §§3–5 and §9 of this document, roughly in order of first appearance in subappendix A.1, and alphabetically in subappendix A.2.

The following typographic conventions are used throughout §§3–5: A boldface symbol, like $S$, or $C$, denotes a set. If a symbol (like $S$ or $v$) is in roman (upright) font, it connotes that this particular object is assumed to have a constant value (throughout a given machine, as well as over time during a given machine execution). Italic symbols such as $S$ or $t$ more generally connote variables, or quantities that may be taken to vary, such as over time, or across the various parts of a machine.

A.1 Table of Notations—In Order of Appearance

Table A-1. Table of Notations (in order of appearance)

<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>A desired abstract computational functionality, expressible as a finite state machine.</td>
</tr>
<tr>
<td>$G$</td>
<td>A garbled representation of some specific finite state machine implementation of $F$.</td>
</tr>
<tr>
<td>Exec[$G$]</td>
<td>A smart contract in which a generic Executor is applied to the garbled state machine $G$.</td>
</tr>
<tr>
<td>$\ell$</td>
<td>Maximum length (in cycles) of the state-machine execution supported by $G$. Also used to refer to the number of layers/levels of logic gates in the computational circuit model.</td>
</tr>
<tr>
<td>$\hat{\ell}$</td>
<td>An upper bound on $\ell$ which may be publicly inferred through inspection of Exec[$G$].</td>
</tr>
<tr>
<td>$q$</td>
<td>The quantity (number) of alternative arcs (conditional state transitions) supported on each cycle.</td>
</tr>
<tr>
<td>$\hat{q}$</td>
<td>An upper bound on $q$ which may be publicly inferred through inspection of Exec[$G$].</td>
</tr>
<tr>
<td>$w$</td>
<td>The width, in bit lines, of an explicit computational circuit for computing the next state, in a version of GABLE based on the circuit model of computation (see §9).</td>
</tr>
<tr>
<td>$\hat{w}$</td>
<td>An upper bound on $w$ which may be publicly inferred through inspection of Exec[$G$].</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>An encoded input symbol to a machine $G$ using a single-source input model.</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of required inputs to a machine $G$ using a multiple-source input model.</td>
</tr>
<tr>
<td>$M$</td>
<td>Number of alternative inputs from which the $N$ inputs in an “$N$ out of $M$” input model are selected. $M \geq N$.</td>
</tr>
</tbody>
</table>

Symbols first used in sec. 4:

<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B$</td>
<td>A party to the Millionaires’ Problem example computation (see §4.1.1).</td>
</tr>
<tr>
<td>$L$</td>
<td>Maximum possible length of a party’s wealth number in bits (see §4.1.1).</td>
</tr>
<tr>
<td>$\ominus$</td>
<td>Special “end” symbol, which directs a machine to yield its final output and halt (§4.1.1).</td>
</tr>
</tbody>
</table>

Symbols first used in sec. 5.1:

<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>Index of a “time step” (state-machine cycle, or circuit layer). $t \in {0, \ldots, \ell - 1}$.</td>
</tr>
<tr>
<td>$S$</td>
<td>Set of supported state-machine states. $S = {s_1, \ldots, s_p}$.</td>
</tr>
<tr>
<td>Symbol/Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of supported states for the state machine. $p =</td>
</tr>
<tr>
<td>$l$</td>
<td>Index of a specific state. (This one is not used in the document, except below.)</td>
</tr>
<tr>
<td>$s, s_t$</td>
<td>A specific (machine) state, $s = s_t \in S$.</td>
</tr>
<tr>
<td>$S_t$</td>
<td>Line (variable) for the state output from step $t$ of the state machine’s execution.</td>
</tr>
<tr>
<td>$S_{-1}$</td>
<td>Line (variable) for the initial state input to step 0 of the state machine’s execution.</td>
</tr>
<tr>
<td>$s_{init}$</td>
<td>Initial state, $s_{init} \in S$. This could be hardcoded into $G$, or provided in an initialization step.</td>
</tr>
<tr>
<td>$s_{fin}$</td>
<td>Final state variable, $s_{fin} = S_{t-1}$. This determines the final result of the computation.</td>
</tr>
<tr>
<td>$s(t)$</td>
<td>The actual machine state $s(t) \in S$ resulting after step $t$ of a particular run.</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of input variables supported for the state machine. $V = {V_1, ..., V_m}$.</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of input variables. $m =</td>
</tr>
<tr>
<td>$i$</td>
<td>Index of a specific input variable, $V_i \in V$.</td>
</tr>
<tr>
<td>$V, V_i$</td>
<td>A specific input variable, $V = V_i \in V$.</td>
</tr>
<tr>
<td>$V_i(t)$</td>
<td>Input line supplying the value assigned to input variable $V_i$ for time step $t$.</td>
</tr>
<tr>
<td>$v_i$</td>
<td>Set of possible values of input variable $V_i$. We write $v_i = {v_i^1, ..., v_i^n}$.</td>
</tr>
<tr>
<td>$v_i(t)$</td>
<td>Denotes the value assigned to input variable $V_i$ on time step $t$. $v_i(t) \in v_i$.</td>
</tr>
<tr>
<td>$n_i$</td>
<td>Number of alternative values of input variable $V_i$. $n_i =</td>
</tr>
<tr>
<td>$j$</td>
<td>Index of a particular value of some particular input variable $V_i$. $j \in {1, ..., n_i}$.</td>
</tr>
<tr>
<td>$v_i^j$</td>
<td>A particular value of a particular input variable $V_i$. Note $v_i^j \in v_i$.</td>
</tr>
<tr>
<td>$A$</td>
<td>Set of arcs (a.k.a., arrows, directed edges, conditional state transitions) in the directed graph that defines the state machine. $A = {a_0, ..., a_{q-1}}$.</td>
</tr>
<tr>
<td>$q$</td>
<td>Number of arcs in the state machine. $q =</td>
</tr>
<tr>
<td>$k$</td>
<td>Index of some specific arc, $a_k \in A$.</td>
</tr>
<tr>
<td>$a, a_k$</td>
<td>A specific arc, $a = a_k$. Identifiable with an ordered triple, $a = (O, C, D)$.</td>
</tr>
<tr>
<td>$O, O_k$</td>
<td>The origin state $O = O_k \in S$ of some specific arc $a_k$.</td>
</tr>
<tr>
<td>$D, D_k$</td>
<td>The destination state $D = D_k \in S$ of some specific arc $a_k$.</td>
</tr>
<tr>
<td>$C, C_k$</td>
<td>The set $C = C_k$ of transition conditions for some specific arc $a_k$. $</td>
</tr>
<tr>
<td>$r$</td>
<td>The number of transition conditions in some specific set $C$ of transition conditions, $r =</td>
</tr>
<tr>
<td>$g$</td>
<td>Index of some specific condition $c_g$ within some specific set $C$ of transition conditions, $g \in {1, ...,</td>
</tr>
<tr>
<td>$c, c_g$</td>
<td>Some specific condition $c = c_g$ within some specific set $C$ of transition conditions. Identifiable with an ordered pair $c = (i, j)$ of a variable index and a value index $j \in {1, ..., n_i}$.</td>
</tr>
<tr>
<td>$OC(s)$</td>
<td>The out-conditions of state $s$ are the set of condition sets $C$ on arcs $a = (s, C, D) \in A$.</td>
</tr>
<tr>
<td>$\perp_i$</td>
<td>A special value for input line $V_i$ meaning “Input variable $V_i$ is unassigned on this cycle.”</td>
</tr>
<tr>
<td>Symbol/Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>( h(\cdot) )</td>
<td>A selected cryptographic hash function, with an ( n )-bit output.</td>
</tr>
<tr>
<td>( n )</td>
<td>The fixed length, in bits, of the random keys, hash function outputs, and encrypted values.</td>
</tr>
<tr>
<td>( K, \overline{K}(v^t_i, t) )</td>
<td>A randomly generated ( n )-bit input key for enabling input value ( v^t_i ) to be provided on input line ( V_i ) for use in time step ( t ).</td>
</tr>
<tr>
<td>( K_i )</td>
<td>A randomly-generated ( n )-bit participant key which an authorized provider may use to (pseudo-)randomly generate the input keys ( K(v^t_i, t) ) for all possible values ( v^t_i ) of input variable ( V_i ) for all time steps ( t ).</td>
</tr>
<tr>
<td>+</td>
<td>A generic Combinator method for combining inputs to a hash function; the Combinator might be lossless (like byte-string concatenation) or associative (like ( \oplus )). See text for details.</td>
</tr>
<tr>
<td>( K(s, t) )</td>
<td>A randomly generated ( n )-bit state key for representing that the state resulting from time step ( t ) of the machine’s execution (which will be used as input to step ( t + 1 ) of the machine’s execution), is ( s ). (Also, if ( t = -1 ), this indicates that ( s ) is the initial state.)</td>
</tr>
<tr>
<td>( K_{\text{cur}} )</td>
<td>A variable representing the most-recent state key ( K(S_{t-1}, t-1) ) as it varies over time ( t ).</td>
</tr>
<tr>
<td>( p_t^{i/j} )</td>
<td>The input provision key for providing the input value ( v^t_i ) to the machine on time step ( t ), given that its current state ( s ) leading in has a particular key ( K_{\text{cur}}^{i/j} = K(s, t-1) ).</td>
</tr>
<tr>
<td>( I, I(a) )</td>
<td>The arc identifier for some given specific arc ( a \in A ). Defined in eq. 5.</td>
</tr>
<tr>
<td>( I_{\text{next}} )</td>
<td>This entry identifier, derived from ( a )'s identifier ( I ), allows encrypting ( a )'s next-state field.</td>
</tr>
<tr>
<td>( I_{\text{valid}} )</td>
<td>This entry identifier, derived from ( a )'s identifier ( I ), allows encrypting ( a )'s valid field.</td>
</tr>
<tr>
<td>( I_e )</td>
<td>A generic entry identifier; may be either ( I_{\text{next}} ) or ( I_{\text{valid}} ), depending on the selected field.</td>
</tr>
<tr>
<td>( e )</td>
<td>A data entry, meaning a particular field of (encrypted) data associated to a given arc ( a ).</td>
</tr>
<tr>
<td>( x )</td>
<td>An arbitrary ( n )-bit plaintext to be encrypted.</td>
</tr>
<tr>
<td>( \oplus )</td>
<td>Bitwise exclusive-OR operation, applicable to pairs of bit-strings of equal length.</td>
</tr>
<tr>
<td>( y, \text{enc}[I_e, x] )</td>
<td>Encrypted version of the ( n )-bit plaintext ( x ) to be used within a specific data entry ( e ).</td>
</tr>
<tr>
<td>( E_{\text{next}} )</td>
<td>Encrypted version of the next-state field of arc ( a ) at time ( t ). ( E_{\text{next}} = \text{enc}[I_{\text{next}}, K(D, t)] ).</td>
</tr>
<tr>
<td>( E_{\text{valid}} )</td>
<td>Encrypted version of the 'valid' field of arc ( a ) at time ( t ). ( E_{\text{valid}} = \text{enc}[I_{\text{valid}}, v] ).</td>
</tr>
<tr>
<td>( v )</td>
<td>A constant ( n )-bit code indicating that a given arc has been validly matched. E.g., ( v = 0^n ).</td>
</tr>
<tr>
<td>( v )</td>
<td>A variable (arc-dependent) valid indicator, for slightly more obscurity. E.g., ( v = h(I) ).</td>
</tr>
<tr>
<td>( E(a, t) )</td>
<td>The encoded representation of some specific arc ( a ) at time ( t ). ( E(a, t) = (E_{\text{next}}, E_{\text{valid}}) ).</td>
</tr>
<tr>
<td>( E(t) )</td>
<td>The encoded representation of the entire arc set ( A ) at time ( t ). ( E(t) = \bigcup_{a \in A} (e(a, t)) ).</td>
</tr>
</tbody>
</table>

**Symbols first used in sec. 5.2:**

<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(\cdot) )</td>
<td>A selected cryptographic hash function, with an ( n )-bit output.</td>
</tr>
<tr>
<td>( n )</td>
<td>The fixed length, in bits, of the random keys, hash function outputs, and encrypted values.</td>
</tr>
<tr>
<td>( K, \overline{K}(v^t_i, t) )</td>
<td>A randomly generated ( n )-bit input key for enabling input value ( v^t_i ) to be provided on input line ( V_i ) for use in time step ( t ).</td>
</tr>
<tr>
<td>( K_i )</td>
<td>A randomly-generated ( n )-bit participant key which an authorized provider may use to (pseudo-)randomly generate the input keys ( K(v^t_i, t) ) for all possible values ( v^t_i ) of input variable ( V_i ) for all time steps ( t ).</td>
</tr>
<tr>
<td>+</td>
<td>A generic Combinator method for combining inputs to a hash function; the Combinator might be lossless (like byte-string concatenation) or associative (like ( \oplus )). See text for details.</td>
</tr>
<tr>
<td>( K(s, t) )</td>
<td>A randomly generated ( n )-bit state key for representing that the state resulting from time step ( t ) of the machine’s execution (which will be used as input to step ( t + 1 ) of the machine’s execution), is ( s ). (Also, if ( t = -1 ), this indicates that ( s ) is the initial state.)</td>
</tr>
<tr>
<td>( K_{\text{cur}} )</td>
<td>A variable representing the most-recent state key ( K(S_{t-1}, t-1) ) as it varies over time ( t ).</td>
</tr>
<tr>
<td>( p_t^{i/j} )</td>
<td>The input provision key for providing the input value ( v^t_i ) to the machine on time step ( t ), given that its current state ( s ) leading in has a particular key ( K_{\text{cur}}^{i/j} = K(s, t-1) ).</td>
</tr>
<tr>
<td>( I, I(a) )</td>
<td>The arc identifier for some given specific arc ( a \in A ). Defined in eq. 5.</td>
</tr>
<tr>
<td>( I_{\text{next}} )</td>
<td>This entry identifier, derived from ( a )'s identifier ( I ), allows encrypting ( a )'s next-state field.</td>
</tr>
<tr>
<td>( I_{\text{valid}} )</td>
<td>This entry identifier, derived from ( a )'s identifier ( I ), allows encrypting ( a )'s valid field.</td>
</tr>
<tr>
<td>( I_e )</td>
<td>A generic entry identifier; may be either ( I_{\text{next}} ) or ( I_{\text{valid}} ), depending on the selected field.</td>
</tr>
<tr>
<td>( e )</td>
<td>A data entry, meaning a particular field of (encrypted) data associated to a given arc ( a ).</td>
</tr>
<tr>
<td>( x )</td>
<td>An arbitrary ( n )-bit plaintext to be encrypted.</td>
</tr>
<tr>
<td>( \oplus )</td>
<td>Bitwise exclusive-OR operation, applicable to pairs of bit-strings of equal length.</td>
</tr>
<tr>
<td>( y, \text{enc}[I_e, x] )</td>
<td>Encrypted version of the ( n )-bit plaintext ( x ) to be used within a specific data entry ( e ).</td>
</tr>
<tr>
<td>( E_{\text{next}} )</td>
<td>Encrypted version of the next-state field of arc ( a ) at time ( t ). ( E_{\text{next}} = \text{enc}[I_{\text{next}}, K(D, t)] ).</td>
</tr>
<tr>
<td>( E_{\text{valid}} )</td>
<td>Encrypted version of the 'valid' field of arc ( a ) at time ( t ). ( E_{\text{valid}} = \text{enc}[I_{\text{valid}}, v] ).</td>
</tr>
<tr>
<td>( v )</td>
<td>A constant ( n )-bit code indicating that a given arc has been validly matched. E.g., ( v = 0^n ).</td>
</tr>
<tr>
<td>( v )</td>
<td>A variable (arc-dependent) valid indicator, for slightly more obscurity. E.g., ( v = h(I) ).</td>
</tr>
<tr>
<td>( E(a, t) )</td>
<td>The encoded representation of some specific arc ( a ) at time ( t ). ( E(a, t) = (E_{\text{next}}, E_{\text{valid}}) ).</td>
</tr>
<tr>
<td>( E(t) )</td>
<td>The encoded representation of the entire arc set ( A ) at time ( t ). ( E(t) = \bigcup_{a \in A} (e(a, t)) ).</td>
</tr>
</tbody>
</table>

**Symbols first used in sec. 5.3:**

<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I' )</td>
<td>A candidate arc identifier that is constructed within the Executor.</td>
</tr>
<tr>
<td>( K )</td>
<td>A set which may be maintained within the Executor consisting of all of the input provision keys ( p_t^{i/j} ) that have been received so far during the current time step ( t ).</td>
</tr>
<tr>
<td>Symbol/Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td>$k$</td>
<td>An arbitrary subset of the present input provision key set $K$, which is combined with a new input provision key $p$ to produce a candidate arc identifier $I'$ for trial matching against arc conditions.</td>
</tr>
<tr>
<td>ASCII(ch)</td>
<td>Denotes the ASCII bytecode for a given text character $ch$.</td>
</tr>
<tr>
<td>$E$</td>
<td>Combined representation of the union of all the arc sets $E(t)$ for all times $t \in {1, ..., \ell - 1}$.</td>
</tr>
<tr>
<td>$\Theta(-)$</td>
<td>Standard notation from computational complexity theory for an exact asymptotic order of growth. Very roughly speaking, this refers to a quantity proportional to the given expression.</td>
</tr>
<tr>
<td>$P(s, t)$</td>
<td>An arbitrary propositional function of a state $s \in S$ and a time step index $t \in {1, ..., \ell}$.</td>
</tr>
<tr>
<td>$P$</td>
<td>Enumerative representation ${(s, t) \mid P(s, t)}$ of the proposition $P(s, t)$.</td>
</tr>
<tr>
<td>$H$</td>
<td>Shuffled, hashed, encrypted representation of $P$. The set $H = {h(K(s, t)) \mid (s, t) \in P}$.</td>
</tr>
<tr>
<td>$X \times Y$</td>
<td>The Cartesian product, ${(x, y) \mid x \in X, y \in Y}$ of the arbitrary sets $X$ and $Y$.</td>
</tr>
<tr>
<td>$f(s, t)$</td>
<td>An arbitrary function of a state $s \in S$ together with a time step index $t \in {1, ..., \ell}$.</td>
</tr>
<tr>
<td>$R$</td>
<td>A reading authority; designates the ability to read a certain function of the current machine state (and/or a taken transition, if any output is associated with these).</td>
</tr>
<tr>
<td>$K_R$</td>
<td>A randomly generated $n$-bit reader key which confers the ability to read a certain function of the machine state as authorized under the reading authority $R$.</td>
</tr>
<tr>
<td>$M$</td>
<td>A set of coded output messages $m_b$ which may be distributed to readers, and possibly digitally signed by the Company if their non-repudiable authentication is required.</td>
</tr>
<tr>
<td>$m_b$</td>
<td>A particular coded output message $m_b \in M$, containing the (encrypted) output information that is readable under a particular reading authority $R$ as a consequence of reaching some particular machine state $s$ during time step $t$.</td>
</tr>
<tr>
<td>$b$</td>
<td>Index of an output message $m_b$. Its range is $1 \leq b \leq</td>
</tr>
</tbody>
</table>

Symbols first used in sec. 5.4:

| ⊗           | A special input symbol to a machine meaning, “proceed to the next state.” Useful in some protocol variants to work around the fairness problem. |
| $\nu$       | The set of all input values received on the current time step. |
| ⊗$\nu$      | A variant of the Proceed symbol whose encoding varies depending on the set of input values received. In a variant protocol this provides a way to resolve the reconvergent arcs problem. |

Symbols first used in sec. 9:

<p>| $w$         | “Width” of a computational memory, in terms of a number of bits. For convenience, in our circuit model we take $w$ to be a power of 2, specifically, $w = 2^{n-1}$ for some integer $n \geq 1$. |
| $V_i^t$     | Input line (variable) $#i$ ($1 \leq i \leq m$) coming into time step $#t$ ($0 \leq t &lt; \ell$); note $m \leq w$. |
| $S_i^t$     | Internal state line (variable) $#i$ ($1 \leq i \leq w$) coming from time step $#t$ ($-1 \leq t &lt; \ell - 1$). |
| $a_i$       | The activation key required to operate the $i^{th}$ routing element in the first layer of the interconnect fabric of the logic layer for the current time step $t$. This is computed by an unlocker for step $t$ as $a_i = h(K(L_{t-1}^{2^{i-1}}) \oplus K(L_{t-1}^{2^i}) \oplus k_t)$. |</p>
<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{t-1}$</td>
<td>This is the $j^{th}$ line feeding into the width-$N$ circuit for evaluating logic layer $#t$ of a circuit-based computation. Note that in general some of these lines may be output state variables from the previous logic layer, and some may be external input variables for time step $t$.</td>
</tr>
<tr>
<td>$n$</td>
<td>An integer $n = \log_2 N$, where $N = 2^w$ is the total width (in bit-lines) of a circuit that updates a $w$-bit wide internal state, and which may also have up to $m \leq w$ bits of external input feeding into each layer of logic.</td>
</tr>
<tr>
<td>$N$</td>
<td>This integer is a power of two, $N = 2^n$, and is also $N = 2w$, and is the total width (in bit-lines) of a circuit that updates a $w$-bit wide internal state, and which may also have up to $m \leq w$ bits of external input feeding into each layer of logic.</td>
</tr>
<tr>
<td>$i$</td>
<td>This is an integer in the range $1, 2, \ldots, w$ which indexes adjacent pairs of input lines. It also can be used to index input variables and internal state variables.</td>
</tr>
<tr>
<td>$j$</td>
<td>This indexes a particular input line across the width of the u.c. and is given either by $j = 2i - 1$ or $j = 2i$ if $i$ is the pair index. Note that we have that $j$ is in the range $1, 2, \ldots, N$ where $N$ is the input width of the circuit.</td>
</tr>
<tr>
<td>$k_t$</td>
<td>A key needed to unlock evaluation of the circuit for logic layer $#t$. This key should be held by a protocol participant (the “unlocker”) that is not involved in the preparation of other inputs to layer $t$ (that is, new general inputs being provided in the input-gathering phase of the current time step).</td>
</tr>
<tr>
<td>$L_{t,u}$</td>
<td>This symbol refers to an internal line within the universal circuit for evaluating layer $t$ of the application circuit. The index $u$ indicates the logic level within the u.c. that this line is the output of, and (for Thompson interconnect networks) is bounded by $0 \leq u &lt; 4n - 3$, where $n = \log_2 N$, and $N$ is the width of the universal circuit. The index $j$ is bounded by $1 \leq j \leq N$, and refers to this line’s position across the width of the universal circuit.</td>
</tr>
<tr>
<td>$K(L_{t,u})$</td>
<td>Garbled (random bit-vector) encoding of the logical value of the line variable $L_{t,u}$ for layer $t$.</td>
</tr>
<tr>
<td>$k_t^{db}$, $k_t^{bu}$</td>
<td>These symbols refer to the random keys assigned to particular values of particular lines associated with the u.c. for evaluating a given time step (layer) of the target computational circuit. The index $b$ refers to the bit-value (0 or 1) being represented (we can also generalize this to non-binary line variables). The index $t$ refers to the time step that is producing this value, and the index $u$ (if present) refers to the logic layer within the u.c. for the present time step that is producing this value, if this is an internal line of that circuit. The index $j$ refers to this line’s position vertically across the width of the universal circuit.</td>
</tr>
<tr>
<td>$E$</td>
<td>Encrypted/garbled version of a given truth table row for a routing element or logic gate.</td>
</tr>
</tbody>
</table>

**Symbols first used in sec. 9.3:**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Length of each auction bid in bits.</td>
</tr>
<tr>
<td>$B$</td>
<td>Number of bidders in an auction.</td>
</tr>
<tr>
<td>$W_i$</td>
<td>An internal state variable meaning bidder $i$ could still win. $i \in {0, \ldots, B - 1}$</td>
</tr>
<tr>
<td>$p$</td>
<td>Bit-position index (LSB = position 0) within the binary (radix-2) bid value; $p = (L - 1) - t$.</td>
</tr>
<tr>
<td>$b^i_p$</td>
<td>The bit in bit-position $#p$ of bidder $i$’s bid. Given by input value $v_i(t)$ for $t = (L - 1) - p$.</td>
</tr>
<tr>
<td>update?</td>
<td>A Boolean variable that is True if any potential winner supplied a ‘1’ bit on this cycle.</td>
</tr>
</tbody>
</table>
### A.2 Table of Notations—In Alphabetical Order

**Table A-2. Table of Notations (in alphabetical order)**

<table>
<thead>
<tr>
<th>Symbol/Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Set of arcs (a.k.a., arrows, directed edges, conditional state transitions) in the directed graph that defines the state machine. $A = {a_0, ..., a_{q-1}}$.</td>
</tr>
<tr>
<td>a, a_k</td>
<td>A specific arc, $a = a_k$. Identifiable with an ordered triple, $a = (O, C, D)$.</td>
</tr>
<tr>
<td>a_i</td>
<td>In sec. 9, the activation key required to operate the $i^{th}$ routing element in the first layer of the interconnect fabric of the logic layer for the current time step $t$.</td>
</tr>
<tr>
<td>B</td>
<td>Second party to the Millionaires’ Problem example computation (see §4.1.1).</td>
</tr>
<tr>
<td>B</td>
<td>Number of bidders in an auction (§9.3).</td>
</tr>
<tr>
<td>b</td>
<td>Index of an output message $m_b$. Its range is $1 \leq b \leq</td>
</tr>
<tr>
<td>b_i_p</td>
<td>In §9.3, this denotes the bit in bit-position $#p$ of bidder $i$’s bid.</td>
</tr>
<tr>
<td>C, C_k</td>
<td>The set $C = C_k$ of transition conditions for some specific arc $a_k$. $</td>
</tr>
<tr>
<td>c, c_g</td>
<td>Some specific condition $c = c_g$ within some specific set $C$ of transition conditions. Identifiable with an ordered pair $c = (i, j)$ of a variable index and a value index $j \in {1, ..., n_i}$.</td>
</tr>
<tr>
<td>D, D_k</td>
<td>The destination state $D = D_k \in S$ of some specific arc $a_k$.</td>
</tr>
<tr>
<td>E</td>
<td>Combined representation of the union of all the arc sets $E(t)$ for all times $t \in {1, ..., \ell - 1}$</td>
</tr>
<tr>
<td>E(t)</td>
<td>The encoded representation of the entire arc set $A$ at time $t$. $E(t) = \bigcup_{a \in A} {e(a, t)}$.</td>
</tr>
<tr>
<td>E, E(a, t)</td>
<td>The encoded representation of some specific arc $a$ at time $t$. $E(a, t) = (E_{next}, E_{valid})$. Also, in §9, $E$ is the encoded representation of a truth table row for a gate in a u.c..</td>
</tr>
<tr>
<td>E_{next}</td>
<td>Encrypted version of the next-state field of arc $a$ at time $t$. $E_{next} = enc[I_{next}, K(D, t)]$.</td>
</tr>
<tr>
<td>E_{valid}</td>
<td>Encrypted version of the ‘valid’ field of arc $a$ at time $t$. $E_{valid} = enc[I_{valid}, v]$.</td>
</tr>
<tr>
<td>Exec[G]</td>
<td>A smart contract in which a generic Executor is applied to the garbled state machine $G$.</td>
</tr>
<tr>
<td>e</td>
<td>A data entry, meaning a particular field of (encrypted) data associated to a given arc $a$.</td>
</tr>
<tr>
<td>enc[I_e, x]</td>
<td>Encrypted version of the $n$-bit plaintext $x$ to be used within a specific data entry $e$.</td>
</tr>
<tr>
<td>F</td>
<td>A desired abstract computational functionality, expressible as a finite state machine.</td>
</tr>
<tr>
<td>f(s, t)</td>
<td>An arbitrary function of a state $s \in S$ together with a time step index $t \in {1, ..., \ell}$.</td>
</tr>
<tr>
<td>G</td>
<td>A garbled representation of some specific finite state machine implementation of $F$.</td>
</tr>
<tr>
<td>g</td>
<td>Index of some specific condition $c_g$ within some specific set $C$ of transition conditions, $g \in {1, ...,</td>
</tr>
<tr>
<td>H</td>
<td>Shuffled, hashed, encrypted representation of $P$. The set $H = {h(K(s, t)) \mid (s, t) \in P}$.</td>
</tr>
<tr>
<td>h(·)</td>
<td>A selected cryptographic hash function, with an $n$-bit output.</td>
</tr>
<tr>
<td>I, I(a)</td>
<td>The arc identifier for some given specific arc $a \in A$. Defined in eq. 5.</td>
</tr>
<tr>
<td>Symbol/Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>$I'$</td>
<td>A candidate arc identifier that is constructed within the Executor.</td>
</tr>
<tr>
<td>$I_e$</td>
<td>A generic entry identifier; may be either $I_{\text{next}}$ or $I_{\text{valid}}$, depending on the selected field.</td>
</tr>
<tr>
<td>$I_{\text{next}}$</td>
<td>This entry identifier, derived from $a$’s identifier $I$, allows encrypting $a$’s next-state field.</td>
</tr>
<tr>
<td>$I_{\text{valid}}$</td>
<td>This entry identifier, derived from $a$’s identifier $I$, allows encrypting $a$’s valid field.</td>
</tr>
<tr>
<td>$i$</td>
<td>Index of a specific input variable, $V_i \in V$, or (in sec. 9) the index of an adjacent pair of bit-lines across a universal circuit’s width, or the index of an internal state variable.</td>
</tr>
<tr>
<td>$j$</td>
<td>Index of a particular value of some particular input variable $V_i$. $j \in {1, \ldots, n_i}$. Or, in sec. 9, the index of a particular bit-line across a universal circuit’s width.</td>
</tr>
<tr>
<td>$K$</td>
<td>A set which may be maintained within the Executor consisting of all of the input provision keys $P_t^{L_t^j}$ that have been received so far during the current time step $t$.</td>
</tr>
<tr>
<td>$K, K(v_t^j, t)$</td>
<td>A randomly generated $n$-bit input key for enabling input value $v_t^j$ to be provided on input line $V_t$ for use in time step $t$.</td>
</tr>
<tr>
<td>$K_{\text{cur}}$</td>
<td>A variable representing the most-recent state key $K(S_{t-1}, t-1)$ as it varies over time $t$.</td>
</tr>
<tr>
<td>$K_i$</td>
<td>A randomly-generated $n$-bit participant key which an authorized provider may use to (pseudo-randomly) generate the input keys $K(v_t^j, t)$ for all possible values $v_t^j$ of input variable $V_t$ on all time steps $t$.</td>
</tr>
<tr>
<td>$K_R$</td>
<td>A randomly generated $n$-bit reader key which confers the ability to read a certain function of the machine state as authorized under the reading authority $R$.</td>
</tr>
<tr>
<td>$K(s, t)$</td>
<td>A randomly generated $n$-bit state key for designating that the state resulting from time step $t$ of the machine’s execution (which will be used as input to step $t + 1$ of the machine’s execution), is $s$. (Also, if $t = -1$, this indicates that $s$ is the initial state.)</td>
</tr>
<tr>
<td>$K(L_t^j), K(L_{t, u}^j)$</td>
<td>In §9, the garbled (random bit vector) encoding of the logical value of the line variable $L_t^j$ or $L_{t, u}^j$.</td>
</tr>
<tr>
<td>$k$</td>
<td>An arbitrary subset of the present input provision key set $K$, which is combined with a new input provision key $p$ to produce a candidate arc identifier $I'$ for trial matching against arc conditions.</td>
</tr>
<tr>
<td>$k$</td>
<td>Index of some specific arc, $a_k \in A$.</td>
</tr>
<tr>
<td>$k_t$</td>
<td>In sec. 9, a key needed to unlock evaluation of the universal circuit for logic layer #t of a target computation circuit.</td>
</tr>
<tr>
<td>$k_{t,b}, k_{t,u}$</td>
<td>In sec. 9, the random key assigned to bit-value $b$ of bit-line $L_t^j$ or $L_{t, u}^j$, respectively.</td>
</tr>
<tr>
<td>$L$</td>
<td>Maximum possible length of a party’s wealth number in bits (see §4.1.1). Also used in §9.3 for the length of an auction bid in bits.</td>
</tr>
<tr>
<td>$L_{t-1}^j$</td>
<td>In sec. 9, the $j^{\text{th}}$ bit-line feeding into the width-$N$ universal circuit for evaluating logic layer #t of a circuit-based computation.</td>
</tr>
<tr>
<td>$L_{t, u}^j$</td>
<td>In sec. 9, the $j^{\text{th}}$ bit-line output from layer $u$ of the universal circuit for evaluating target logic layer #t of a circuit-based computation.</td>
</tr>
<tr>
<td>$l$</td>
<td>Index of a specific state. (This one is not used in the document, except below.)</td>
</tr>
<tr>
<td>Symbol/Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>$\ell$</td>
<td>Maximum length (in cycles) of the state-machine execution supported by $G$.</td>
</tr>
<tr>
<td>$\hat{\ell}$</td>
<td>An upper bound on $\ell$ which may be publicly inferred through inspection of $G$.</td>
</tr>
<tr>
<td>M</td>
<td>A set of coded output messages $m_k$, which may be distributed to readers, and possibly digitally signed by the Company if their non-repudiable authentication is required.</td>
</tr>
<tr>
<td>$M$</td>
<td>Number of alternative inputs from which the $N$ inputs in an “$N$ out of $M$” input model are being selected. $M \geq N$.</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of input variables. $m =</td>
</tr>
<tr>
<td>$m_b$</td>
<td>A particular coded output message $m_b \in M$, containing the (encrypted) output information that is readable under a particular reading authority $R$ as a consequence of reaching some particular machine state $s$ during time step $t$.</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of required inputs to a machine $G$ using a multiple-source input model, or (in sec. 9), the total width in bit-lines of a universal circuit for updating the machine state.</td>
</tr>
<tr>
<td>$n$</td>
<td>The fixed length, in bits, of the random keys, hash function outputs, and encrypted values, or (in sec. 9) the logarithm base 2 of the width $N$ of a universal circuit.</td>
</tr>
<tr>
<td>$n_i$</td>
<td>Number of alternative values of input variable $V_i$. $n_i =</td>
</tr>
<tr>
<td>$O, O_k$</td>
<td>The origin state $O = O_k \in S$ of some specific arc $a_k$.</td>
</tr>
<tr>
<td>OC$(s)$</td>
<td>The out-conditions of state $s$ are the set of condition sets $C$ on arcs $a = (s, C, D) \in A$.</td>
</tr>
<tr>
<td>$P$</td>
<td>Enumerative representation ${ (s, t) \mid P(s, t) }$ of the proposition $P(s, t)$.</td>
</tr>
<tr>
<td>$P(s, t)$</td>
<td>An arbitrary propositional function of a state $s \in S$ and a time step index $t \in {1, \ldots, \ell}$.</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of supported states for the state machine. $p =</td>
</tr>
<tr>
<td>$p_t^{i_j}$</td>
<td>The input provision key for providing the input value $v_i^j$ to the machine on time step $t$, given that its current state $s$ leading in has a particular key $K_{cur} = K(s, t - 1)$.</td>
</tr>
<tr>
<td>$q$</td>
<td>Number of arcs in the state machine. $q =</td>
</tr>
<tr>
<td>$\hat{q}$</td>
<td>An upper bound on $q$ which may be publicly inferred through inspection of $G$.</td>
</tr>
<tr>
<td>$R$</td>
<td>A reading authority; designates the ability to read a certain function of the current machine state (and/or a taken transition, if any output is associated with these).</td>
</tr>
<tr>
<td>$r$</td>
<td>The number of transition conditions in some specific set $C$ of transition conditions, $r =</td>
</tr>
<tr>
<td>$S$</td>
<td>Set of supported state-machine states. $S = {s_1, \ldots, s_p}$.</td>
</tr>
<tr>
<td>$S_{-1}$</td>
<td>Line (variable) for the initial state input to step 0 of the state machine’s execution.</td>
</tr>
<tr>
<td>$S_{\text{fin}}$</td>
<td>Final state variable, $S_{\text{fin}} = S_{t-1}$. This determines the final result of the computation.</td>
</tr>
<tr>
<td>$S_t$</td>
<td>Line (variable) for the state output from step $t$ of the state machine’s execution.</td>
</tr>
<tr>
<td>$S_t^i$</td>
<td>In §9, internal state line (variable) $#i$ (1 ≤ $i$ ≤ $w$) from time step $#t$ (0 ≤ $t$ &lt; $\ell$).</td>
</tr>
<tr>
<td>$s, s_t$</td>
<td>A specific state, $s = s_t \in S$.</td>
</tr>
<tr>
<td>$S_{\text{init}}$</td>
<td>Initial state, $S_{\text{init}} \in S$. This could be hardcoded into $G$, or provided in an initialization step.</td>
</tr>
<tr>
<td>Symbol/Notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>$s(t)$</td>
<td>The actual machine state $s(t) \in S$ resulting after step $t$ of a particular run.</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>An encoded input symbol to a machine $G$ using a single-source input model.</td>
</tr>
<tr>
<td>$t$</td>
<td>Index of a “time step” (state-machine cycle). $t \in {1, \ldots, \ell}$.</td>
</tr>
<tr>
<td>$update$?</td>
<td>In §9.3, a Boolean variable that is True if any potential winner supplied a ‘1’ bit on this cycle.</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of input variables supported for the state machine. $V = {V_1, \ldots, V_m}$.</td>
</tr>
<tr>
<td>$v_i$</td>
<td>Set of possible values of input variable $V_i$. We write $v_i = {v_i^1, \ldots, v_i^n}$.</td>
</tr>
<tr>
<td>$V, V_i$</td>
<td>A specific input variable, $V = V_i \in V$.</td>
</tr>
<tr>
<td>$V_i(t)$</td>
<td>Input line supplying the value of input variable $V_i$ for time step $t$; we tend to use this notation when discussing the state-machine model.</td>
</tr>
<tr>
<td>$V_i^t$</td>
<td>Another notation for input line (variable) #i ($1 \leq i \leq m$) coming into time step #t ($0 \leq t &lt; \ell$); note that $m \leq w$ in the circuit model discussed in §9.</td>
</tr>
<tr>
<td>$v_i(t)$</td>
<td>Refers to the value assigned to input variable $V_i$ on time step $t$. $v_i(t) \in v_i$.</td>
</tr>
<tr>
<td>$\nu$</td>
<td>The set of all input values received on the current time step.</td>
</tr>
<tr>
<td>$\nu$</td>
<td>A constant $n$-bit code indicating that a given arc has been validly matched. E.g., $\nu = 0^n$.</td>
</tr>
<tr>
<td>$\nu$</td>
<td>A variable (arc-dependent) valid indicator, for slightly more obscurity. E.g., $\nu = h(i)$.</td>
</tr>
<tr>
<td>$v_i^t$</td>
<td>A particular value of a particular input variable $V_i$. $v_i^t \in v_i$.</td>
</tr>
<tr>
<td>$W_i$</td>
<td>In §9.3, an internal state variable meaning bidder $i$ could still win. $i \in {0, \ldots, B - 1}$.</td>
</tr>
<tr>
<td>$w$</td>
<td>The “width” of a computational memory, in terms of a number of bits. Or, the output width, in bit lines, of an explicit computational circuit for computing the next state, in a version of GABLE based on the circuit model of computation (see §9).</td>
</tr>
<tr>
<td>$X \times Y$</td>
<td>The Cartesian product, ${(x, y) \mid x \in X, y \in Y}$ of the arbitrary sets $X$ and $Y$.</td>
</tr>
<tr>
<td>$x$</td>
<td>An arbitrary $n$-bit plaintext to be encrypted.</td>
</tr>
<tr>
<td>$y$</td>
<td>Encrypted version of the $n$-bit plaintext $x$ to be used within a specific data entry $e$.</td>
</tr>
<tr>
<td>$\Theta(\cdot)$</td>
<td>Standard notation from computational complexity theory for an exact asymptotic order of growth. Roughly speaking, this refers to a quantity proportional to the given expression.</td>
</tr>
<tr>
<td>$\perp_i$</td>
<td>A special value for input line $V_i$ meaning “Input variable $V_i$ is unassigned on this cycle.”</td>
</tr>
<tr>
<td>$+$</td>
<td>A generic operator for combining inputs to a hash function; this operator might be lossless (like concatenation of self-delimiting byte strings) or associative (like $\odot$). See text for details.</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>Bitwise exclusive-OR operation, applicable to pairs of bit strings of equal length.</td>
</tr>
<tr>
<td>$\odot$</td>
<td>A special input symbol to a machine meaning, “proceed to the next state.”</td>
</tr>
<tr>
<td>$\odot_p$</td>
<td>A variant of the Proceed symbol whose encoding varies depending on the set of input values received. In a variant protocol this provides a way to resolve the reconvergent arcs problem.</td>
</tr>
<tr>
<td>$\ominus$</td>
<td>A special input symbol to a machine meaning, “halt” (yielding final outputs, if any).</td>
</tr>
</tbody>
</table>
In this appendix, we provide complete code for a reference implementation of a Garbler in the Python language, version 3.x.

B.1 Concise version of Garbler code (no comments)

The following listing of the reference Python code for the Garbler has all comments stripped out for conciseness. A much more verbose listing with extensive comments is provided in subappendix B.2.

```python
import argparse
import random
import json

KEY_LENGTH=256

def randBytes(length):
    return (bytes(random.getrandbits(8) for _ in range(length//8))

def hash256(seq):
    assert(type(seq) == type(b''))
    return keccak(seq)

def get_key_targets(mach, in_states):
    lineVals = {}
    states = set()
    for arc in mach:
        (origState, conditions, destState) = arc
        states.add(origState)
        states.add(destState)
        for lineName in conditions:
            assert(not lineName.startswith('_'))
            condVal = conditions[lineName]
            try:
                lineVals[lineName].add(condVal)
            except KeyError:
                lineVals[lineName] = {condVal}
    lineVals['_out_states'] = states
    if not in_states:
        lineVals['_in_states'] = states
    return lineVals

def gen_step_keys(mach, in_states=None):
    lineVals = get_key_targets(mach, in_states)
    keys = {}
    for target in sorted(lineVals):
        keys[target] = {}
        for val in sorted(lineVals[target]):
            keys[target][val] = randBytes(KEY_LENGTH)
    if in_states:
        keys['_in_states'] = in_states
    return keys

def save_keys(t, keys):
    for key in keys:
        with open('%s-%s.keys' % (t, key), 'w') as keyFile:
            json_keys = {k: v.hex() for k, v in keys[key].items()}
            json.dump(json_keys, keyFile)

def save_gc(t, gc):
    with open('%s.gc' % t, 'w') as gcFile:
        json.dump(json, gcFile)
```

105
def encdec(k, v):
    assert (len(v) == KEY_LENGTH//8)
    return bytes(map(lambda h,x: h^x, hash256(k), v))

def garble(circ, ks):
    encArcs = []
    for arc in circ:
        (origState, conditions, destState) = arc
        arcID = ks['_in_states'][origState]
        for lineName in filter(lambda varName: not varName.startswith('_'), ks.keys()):
            if lineName in conditions:
                Pij = hash256(bytes(map(lambda b1,b2: b1^b2, ks[lineName][conditions[lineName]], ks['_in_states'][origState])))
                nextID = bytes(arcID[:-1] + bytes([arcID[-1] ^ ord('n')]))
                validID = bytes(arcID[:-1] + bytes([arcID[-1] ^ ord('v')]))
                destStateCode = ks['_out_states'][destState]
                encNextState = encdec(nextID, destStateCode)
                encValidArc = encdec(validID, b'\0'*(KEY_LENGTH//8))
                encArcs.append((encNextState, encValidArc))
    random.shuffle(encArcs)
    return encArcs

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='PoC Garbler')
    parser.add_argument('--seed', default=None, help='Seed to use for deterministic garbling')
    parser.add_argument('--time_steps', default=10, type=int, help='Number of time steps to generate circuits for')
    parser.add_argument('circuit', help='JSON file to use as the circuit')
    args = parser.parse_args()
    if args.seed:
        random.seed(args.seed)
    with open(args.circuit) as fsmFile:
        circuit = json.load(fsmFile)
    in_states = None
    for t in range(args.time_steps):
        keys = gen_step_keys(circuit, in_states)
        save_keys(t, keys)
        garbled = garble(circuit, keys)
        save_gc(t, garbled)
        if t == 0:
            if not 'SInit' in keys['_in_states']:
                raise Exception('please create a state SInit and/or change ' 'this logic')
            with open('init.gc', 'w') as initFile:
                json.dump([keys['_in_states']['SInit'].hex()], initFile)
            in_states = keys['_out_states']

B.2 Verbose listing of Garbler code (with detailed comments)

The following version of the reference Python code for the Garbler includes detailed comments and some diagnostics. A much more concise listing without comments is given in subappendix B.1 above.
# Description:

This Python program is an example implementation of a Garbler for finite state machines. It takes an input a description of a finite state machine $F$, and produces as output an array of coded input keys $K(v_i^j,t)$ and set of encrypted arcs $E(t)$ making up the description of the garbled version $G$ of $F$, for each time step $t$ in the range from 0 to $L-1$, where $L$ is the number of supported time steps.

Language: Python 3 (version 3.6 or higher)

## Other dependencies:

- The present version assumes that the 'eth-hash' library has been installed; it is used to obtain the keccak() (Keccak-256) hash function, which is used because it is also available in Solidity (which the prototype Executor is written in). To install eth-hash in your environment:

  ```
  $ pip install eth-hash[pycryptodome]
  ```

## Usage:

```bash
./garbler.py [--seed <seed>] [--time_steps <L>] <FSM_file>
```

## Arguments:

- `--seed <string=None>`
  
  Optional argument. Provides a string to use as a random seed. If not provided, no consistent seed is used.

- `--time_steps <integer=10>`
  
  Number $L$ of time steps worth of garbled state-machine execution to synthesize. Defaults to 10 steps if not specified.

- `<FSM_file>`
  
  Required argument; the pathname of the .sm file describing the state machine. This is a JSON file consisting of an array of arcs, where each arc is an array with 3 elements:

  - `<origState> [string]` - Name of the arc's origin state.
  - `<varDict> [map]` - Condition set; maps an input variable name [string] to a value [string].
  - `<destState> [string]` - Name of the arc's destination state.
### Output files:

This program produces the following output files:

- `init.gc` [JSON]
  - An array with one element, which is the 64 hex digit code for the initial state [string].
- `<t>.gc` [JSON]
  - An array of encrypted arcs, each of which is an array of two strings, which are the 64 hex digit codes for the encrypted 'next state' and 'valid' entries, respectively.
- `<t>._in_states.keys` [JSON]
  - A map from state names to the 64 hex digit codes for those states for input to the garbled state machine on time step `<t>.
- `<t>._out_states.keys` [JSON]
  - A map from state names to the 64 hex digit codes for those states for output from the garbled state machine on time step `<t>.
  - Identical to `<t+1>._in_states.keys`
- `<t>._<V>.keys` [JSON]
  - For the input variable named `<V>`, a map from each name of a value of that variable to the 64 hex digit code (key) for that value for input to the machine on time step `<t>.

### Revision history:

```
# v0.1 (7/29/'18) - Original version by Kasimir Gabert
  <kasimir@gatech.edu>.
# v0.2 (7/18/'18) - Additional comments added by M.P. Frank
  <mpfrank@sandia.gov>.
# v0.3 (8/24/'18) - Further cleanup for inclusion in draft report (MPF).
# v0.4 (8/30/'18) - Continuing cleanup for draft report.
# v0.5 (10/8/'18) - Modifying to use keccak-256 hash func instead of 256-bit standard SHA-3. Also got rid of binascii library; using builtins instead. (MPF)
# v0.7 (3/19/'20) - Fix replacing input K(Vij) with hash(K(Vij) xor K(state)) by Ryan Kao <rkao@sandia.gov>.
```

An example of such a file is shown below:

```json
[
  "SInit", {"A": "1", "B": "1"}, "SInit"],
  "SInit", {"A": "0", "B": "0"}, "SInit"],
  "SInit", {"A": "0", "B": "1"}, "SInit"],
  "SInit", {"A": "1", "B": "0"}, "SInit"],
  "SFail", {"A": "0", "B": "0"}, "SFail"],
  "SFail", {"A": "1", "B": "1"}, "SFail"],
```

---
# Imports. [code section]
import argparse  # Parse command-line arguments.
import random    # Pseudo-random number generation.
import json      # JSON format file reading and writing.
import pprint    # Pretty-print Python data structures.

# Constant definitions. [code section]

# KEY_LENGTH                                [integer constant]  
# A "key" in the garbler means a random codeword representing a particular state in the state machine on a given cycle, or a particular value of an input variable to the state machine. For our purposes, we will use keys that are 256 bits long.

KEY_LENGTH = 256

# Function definitions. [code section]

def randBytes(length):
    return (bytes(random.getrandbits(8) for _ in range(length//8)))
    # (The above expression is using generator syntax.)
def hash256(seq):
    assert type(seq) == type(b'')  # Make sure argument is a bytes object.
    return keccak(seq)  # Keccak-256 hash function available in Solidity.

# |---------------------------------------------------------------------|
# | get_key_targets()                               [function] |
# | GET the names and the possible (plaintext) values |
# | of all of the lines in the machine. Here, 'line' |
# | means a variable, which may either be an input |
# | variable or a state variable. (Think of it like a |
# | hardware data bus.)                                |
# | Arguments:                                       |
# | ----------                                     |
# | mach - A state machine, represented as a sequence of   |
# | arcs, where an arc is a three-element              |
# | sequence [<orig>, <conds>, <dest>] of             |
# | origin state <orig>, set of transition            |
# | conditions <conds>, and destination state    |
# | <dest>.                                         |
# | in_states - If this is not None, then our input    |
# | states are already known (as outputs from         |
# | the prior iteration) and we do not need           |
# | to generate key targets for them.                |
# | Return value:                                   |
# | ----------                                     |
# | lineVals - A dictionary mapping each line name     |
# | (variable name) to its set of possible value      |
# | names [strings]. Two special reserved line       |
# | names '_in_states' and '_out_states' designate    |
# | the input state line to and output state line    |
# | from the machine on any given time step,         |
# | respectively.                                    |
# | states = set()                                  |

def get_key_targets(mach, in_states):
    # Initialize an empty dictionary 'lineVals', mapping the name   |
    # of each variable ('line') to its set of possible values. |
    lineVals = {}
    # The set of state names that we have seen so far ('states')     |
    # is initially the empty set.                                     |
    states = set()
# Iterate through each row in the transition table (an arrow or 'arc' of the state machine).

for arc in mach:
    # Decompose the row/arc data structure (sequence) into its 3 elements which are the origin state name, set of transition conditions, and destination state name, respectively.
    (origState, conditions, destState) = arc

    # The first (#0) element of the row is the initial state name. Make sure it's included in our state set.
    states.add(origState)

    # The third (#2) element of the row is the final state name. Make sure it's also included in our state set.
    states.add(destState)

    # The second (#1) element of the row is a map from input lines (input variables) to their values. Iterate through the keys of this map, which are the names of the input variables.
    for lineName in conditions:
        # This assert enforces that the input lines are distinct from the 'special' reserved lines (whose names start with '_') which encode state variables. If the user tries to start a variable name with an underscore, we flag this as an error.
        assert(not lineName.startswith('_'))

        # Get the name of the particular value that must be assigned to this particular input line in this particular arc's set of transition conditions.
        condVal = conditions[lineName]

        # This try/except clause adds the value name to the set of value names for this variable; or, if that set hasn't been created yet, we create it in the except clause.
        try:
            # Assuming that we already have a set of values for this variable in the dictionary, make sure this specific value is included in the set.
            lineVals[lineName].add(condVal)
        except KeyError:
            # Occurs if this key hasn't been assigned to yet.
            # Initialize the set of values for this variable, as a singleton set of the current value.
```python
    lineVals[lineName] = {condVal}
    
    #/ End try/except.
    #/ End for lineName in conditions.
    
    #/ End for arc in mach.
    #----------------------------------------------------------------------------------------------
    # Record the list of state names under the '_out_states'
    # attribute. This will exist for every time step.
    lineVals['_out_states'] = states
    
    #----------------------------------------------------------------------------------------------
    # On the first iteration only, when 'in_states' is None,
    # include the list of states in the '_in_states' attribute,
    # because we will need to generate keys for these states
    # (although really only the key for state SInit gets used).
    if not in_states:
        lineVals['_in_states'] = states
    return lineVals
    #/ End function get_key_targets().
```

---

```python
# gen_step_keys()
# [function] #
# This function generates a data structure 'keys'
# which contains all the mappings from state lines
# and input variables to the random keys for their
# possible values, for the present iteration of the
# state machine 'mach'. If the argument 'in_states'
# is provided and not 'None', it's assumed to be a
# map of already-known state keys (output from the
# last iteration) to be used as the input state keys
# for the current iteration.
#
# Arguments:
#
# mach - A state machine, represented as a sequence of
#        arcs, where an arc is a three-element
#        sequence [<orig>, <conds>, <dest>] of
#        origin state <orig>, set of transition
#        conditions <conds>, and destination state
#        <dest>.
#
# in_states - Map from state names to keys to use
#              for the input state line to this step of
#              the machine, or None if not yet known.
#              When a non-None value is provided, it is
#              the same as the keys for the output
#              states from the previous time step.
#
# Returns:
#
# keys [map] - A map from line names to maps from
#              value names to 32-byte random codes.
#```
def gen_step_keys(mach, in_states=None):
    #-----------------------------------------------
    # Obtain a data structure 'lineVals' which gives the names of
    # all the states, input variables, and input variable values for
    # this state machine. The input state and output state can also
    # be considered variables. All the variables (including input
    # variables and state variables) are called 'key targets',
    # because we will need to assign a key to each of their possible
    # values.
    lineVals = get_key_targets(mach, in_states)

    pp.pprint(lineVals)  # Diagnostic.
    #-----------------------------------------------
    # Generate keys for each key target. The 'keys' data structure
    # will map each variable name to a map from its possible value
    # names to their corresponding random keys. Initially empty.
    keys = {}

    #-----------------------------------------------
    # Iterate through the (input and state) variable names. These
    # are the keys of the lineVals dictionary. (Sorting just
    # ensures deterministic behavior regardless of how the
    # dictionary happens to be ordered internally.)
    for target in sorted(lineVals):
        #-----------------------------------------------
        # Initialize an empty dictionary for this target's keys.
        keys[target] = {}

        #-----------------------------------------------
        # Iterate through the possible values of that target variable.
        for val in sorted(lineVals[target]):
            #-----------------------------------------------
            # That key is a newly-created random number of the given length.
            keys[target][val] = randBytes(KEY_LENGTH)

        # End for val in target values.
    # End for target in list of target variable names.
    #-----------------------------------------------
    # On every iteration after the first, the '_in_states' key
    # assignments, provided in the in_states argument, will just be
    # a copy of the key assignments for the output states from the
    # previous iteration.
    if in_states:
        keys['_in_states'] = in_states

    pp.pprint(keys)  # Diagnostic
    return keys

#-----------------------------------------------
# save_keys()                                      [function]  |
def save_keys(t, keys):
    for key in keys:
        with open('%s-%s.keys' % (t, key), 'w') as keyFile:
            json.dump(json_keys, keyFile)
            for k, v in keys[key].items()
        # End with open output file.
    #__/ End function save_keys().

def save_gc(t, gc):
    # Saves the garbled circuit for the current iteration to a file named "<t>.gc", where <t> is the time step number, in the range from 0 to L-1.
    Arguments:
    ---------
    t [integer] - Current time step number, in the range from 0 through L-1.
    gc [sequence] - Garbled circuit for a given time step, consisting of a sequence E of encrypted arcs, each a pair (Enext,Evalid) of encrypted entries.
    Output file:
    ---------
    <t>.gc [JSON] - A JSON-format representation of the gc object as an array of 2-element arrays of 64 hex digit strings giving the encrypted 'next'
def save_gc(t, gc):
    with open('%s.gc' % t, 'w') as gcFile:
        json.dump(
            list(map(
                lambda earc: (earc[0].hex(), earc[1].hex(), gc),
                gcFile),
            gcFile)
        )
    #__/ End with open output file.
    #__/ End function save_gc().

def encdec(k, v):
    # Make sure the 'value' we're encoding has our standard key length.
    assert(len(v) == KEY_LENGTH//8)
    return bytes(map(lambda h, x: h^x, hash256(k), v))
    #__/ End function encdec().

def garble():
    # Produces the garbled representation of a given 'circuit' (state machine transition table) circ
def garble(circ, ks):
    # Initialize the list of encrypted arcs to output; initially
    # this is just the empty list.
    encArcs = []
    # Each arc in the machine needs to be garbled (encrypted).
    for arc in circ:
        # Decompose the row/arc data structure (sequence) into its 3
        # elements which are the origin state name, set of transition
        # conditions, and destination state name, respectively.
        (origState, conditions, destState) = arc

        # In the following, we are constructing an 'arc identifier',
        # which is a code used to identify and encrypt/decrypt this
        # specific arc of the state machine.
        arcID = ks['_in_states'][origState]

        # For each input line (variable) name 'lineName', which consists
        # of the keys of the 'ks' dictionary, except for the special
        # state variable names starting with underscore...
        for lineName in filter(lambda varName: not varName.startswith('_'), ks.keys()):
            #
If that input line is included among the arc's conditions, then we'll XOR its code into the arc ID.

```python
if lineName in conditions:
    Pij = hash256(bytes(map(lambda b1,b2: b1^b2,
                                ks[lineName][conditions[lineName]],
                                ks["_in_states"]origState)))
    arcID = bytes(map(lambda b1,b2: b1^b2, arcID, Pij))
```

To preclude retrospection of alternate paths, we require a state-dependent input provision key hash(Kij XOR Ks).

```python
if lineName in conditions:
    arcID = bytes(map(lambda b1,b2: b1^b2, arcID, Pij))
```

Get the 'entry identifiers' nextID and validID for this arc's 'next state' and 'valid' data entries, respectively. We derive these from the 32-byte arcID by just tweaking its last byte.

```python
nextID = bytes(arcID[:-1] + bytes([arcID[-1] ^ ord('n')])))
validID = bytes(arcID[:-1] + bytes([arcID[-1] ^ ord('v')])))
```

Get the 32-byte random code for this arc's destination state. This is the main data to encrypt in this arc's representation.

```python
destStateCode = ks["_out_states"]destState
```

We use the initial state and the keys for all the input line values XOR'ed together with the ASCII byte 'n' (for 'next state') as the information that will be hashed together to generate the one-time pad for encrypting the destination state code.

```python
eencNextState = encdec(nextID, destStateCode)
```

We also encrypt an extra output whose only purpose is to let us know which of the outputs is the correct one (since their order will be randomized). Note that this will generate a false positive once every 2^256 times—i.e., basically never.

```python
eValidArc = encdec(validID, b\"0\"*(KEY_LENGTH//8))
```

Add the encrypted next-state key and valid identifier to the list of encrypted arcs.

```python
eArcs.append((encNextState, eValidArc))
```

End for arc in FSM.

Randomize the order of the encrypted arcs and return them.

```python
random.shuffle(eArcs)
return eArcs
```

End function garble().

# |=========================================================================|
# |      Main program.        [code section]    |
# |vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv|
#|=========================================================================|
if __name__ == '__main__':  # Skip this section if loading file as a module.

    # Parse command-line arguments.
    parser = argparse.ArgumentParser(description='PoC Garbler')
    parser.add_argument('--seed', default=None, help='Seed to use for deterministic garbling')
    parser.add_argument('--time_steps', default=10, type=int, help='Number of time steps to generate circuits for')
    parser.add_argument('circuit', help='JSON file to use as the circuit')
    args = parser.parse_args()

    if args.seed:
        random.seed(args.seed)

    with open(args.circuit) as fsmFile:
        circuit = json.load(fsmFile)

    in_states = None

    pp = pprint.PrettyPrinter(indent=4)

    for t in range(args.time_steps):
        keys = gen_step_keys(circuit, in_states)
        save_keys(t, keys)

        garbled = garble(circuit, keys)
        save_gc(t, garbled)

    # For the first time step only, save the key of the initial state
    # (which must have the special name 'SInit') in the file "init.gc".
if t == 0:
    if not 'SInit' in keys['_in_states']:
        raise Exception('please create a state SInit and/or change '
                         'this logic')
    with open('init.gc', 'w') as initFile:
        json.dump([keys['_in_states']['SInit'].hex()],
                   initFile)

# __/ End if t=0.

# -------------------------------
# Pass forward the output-state keys to also use as the input-state keys on the next iteration.

in_states = keys['_out_states']

# __/ End for t in range 0 through L-1.

# __/ End if loading this file as the main program...

# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^|
# END OF FILE:    garbler.py                                             |
# ===========================================================================|
APPENDIX C  REFERENCE SOLIDITY CODE FOR EXECUTOR

Below is an initial reference implementation in Solidity of the Executor Exec[G] for the garbled version G of the state machine F illustrated in Figure 5-3. Please note that this reference implementation is very simple; it uses a basic gather-all-inputs input model, and it does not authenticate input providers.

C.1 Concise version of reference Solidity implementation of Executor

This version has comments stripped out for conciseness and elides most of the repetitive data section. Section C.2 has the full code, with comments.

```solidity
pragma solidity >=0.4.24;

contract ExecutableMachine {

struct EncryptedArc {
    uint256 encNext;
    uint256 encValid;
}

uint16 constant maxSteps = 10;
uint8 constant nInputs = 2;
uint16 constant nArcs = 8;
uint256 sInit;
EncryptedArc[nArcs][maxSteps] arcs;
uint16 nextStep = 0;
uint256 public curState;
uint8 nInputsRecd = 0;
bool[2] gotInput = [false, false];
uint256 combinedInputs = 0;

constructor () public {
    sInit = 0x7128b98ce5a2a8cce5f8db6fc52cbf6ce17b20b6122e8168650b607d1d194fb6;
    curState = sInit;
    arcs[0][0] = EncryptedArc(0x8b40c8d1f770cfcadfd6d8b0b17003fbef22597928ac9afdf95eeeb1d1417ce8,
    0x2e735e54eb61b5a9ea17893dccf533cc472c4e6b84668b36e6f60330acc85e);
    arcs[0][1] = EncryptedArc(0x7042af3d7aa8de40a73ad98365e793ad1485bce2e8cc26f4fe95e5179ff4c50,
    0xe48e3b789ba6efa75f9ab00de822fc554972510758a9b97c41f9793ca3c1);
    arcs[0][2] = EncryptedArc(0xa2879d9562dfad7ace4d5a5e286d93e8e60720ec14702b1f7665f863b6255131,
    0xe8a6e76644a2f2549972510758a9b97c41f9793ca3c1);
    arcs[0][3] = EncryptedArc(0x8e151a6e7711e4c9e19467eff78a79d94e67e92575fd359296ac624b268797c7e7566a8f0b0de9d);
    arcs[0][4] = EncryptedArc(0x2f67ef7e67171acec64d6d8ab0d0b73e711851fb647a371fe3d0dcd95da65b54,
    0x798ca1b5f579a96447f7811ce895f5e966b291c15f7253558ca1b0bebe7a95f38);
    arcs[0][5] = EncryptedArc(0x11dc9df283fde79645f4e4531d5a1f6c7d4a24f20e52bf6911f8e4416c3,
    0xdb3be5d959e3a9a9a9b29b52147495141c04ebe5c9faea544fa117009a7a661ea8d);
    arcs[0][6] = EncryptedArc(0x1f1d6783650a502a69311e6a0977f12b9af32783f6fe9c13c29f000d442ce4ae,
    0xe31b413fbac4e467e92575fd359296ac624b268797c7e7566a8f0b0de9d);
    arcs[0][7] = EncryptedArc(0x69f5598e13acfffe355defefe2545b353742c66cedee3010e66e8b10f14094,
    0x0c6ccfa45c5c16b7a617f8f212b52b597f2913a892d2f42ac3536a2a60);
    arcs[0][8] = EncryptedArc(0x69f5598e13acfffe355defefe2545b353742c66cedee3010e66e8b10f14094,
    0x0c6ccfa45c5c16b7a617f8f212b52b597f2913a892d2f42ac3536a2a60);
}

// ... and likewise for the other 9 time steps ...

function provideInput(uint8 varIndex, uint256 value) public {
    if (nInputsRecd == nInputs) return;
    if (varIndex >= nInputs) return;
    if (gotInput[varIndex]) return;
    nInputsRecd++;
    gotInput[varIndex] = true;
    combinedInputs ^= value;
    if (nInputsRecd == nInputs) {
        executeStep();
    }
}
```

120
C.2 Verbose version of reference Solidity code for Executor (with comments)

This version is documented with verbose comments. Section C.1 has bare code, with no comments.
(which is NOT the standard SHA function from sha256() (2018 v0.2 (2018 v0.1 (2018 v0.0 (2018 All revisions are by MPF (M.P. Frank, mpfrank@sandia.gov),

Revision history:

All revisions are by MPF (M.P. Frank, mpfrank@sandia.gov), w. contributions from CNC (C.N. Cordi, cnordi@sandia.gov).

v0.0 (2018-08-16) - First checkin; not yet compiled.

v0.1 (2018-08-22) - Compiles in Truffle.

v0.2 (2018-09-01) - Last version before debugging.

v0.3 (2018-10-08) - A few bug fixes; switched hash function from sha256() (256-bit SHA-2) to keccak256()

(which is NOT the standard SHA-3 function).

// Need to figure out the right version specifier still

pragma solidity >=0.4.24; // Need to figure out the right version specifier still
/// ExecutableMachine [contract]
/// The ExecutableMachine contract is the main (and at this point, only) smart contract making up the implementation of an executable garbled state machine G.

Public interface:

---

provideInput(varIndex, value) - Provide a (coded) value for a specified input variable for this time step.
curState() - This getter function returns the coded current state of the garbled state machine.

Internal types:  (Not intended to be externally used.)

---

struct EncryptedArc - Stores encrypted data for one arc of the garbled state machine.

Internal constants:

---

uint16 maxSteps - Maximum # of supported time-steps.
uint8 nInputs - Number of input lines into the machine.
uint16 nArcs - How many state-transition arcs machine has.
uint256 sInit - Coded rep. of initial state of machine.
EncryptedArc[][] arcs - Garbled array of arcs by time step.

Internal variables:

---

uint16 nextStep - Sequential index of current time step.
uint256 curState - Coded current state of machine; externally readable via the .curState() getter function.
uint8 nInputsRecd - Number of input variables whose values have been received so far on this time step.
bool[] gotInput - Whether a value has been received yet for each input variable on this time step.
uint256 combinedInputs - Represents the full set of all input values received so far on this time step.

Internal functions:

---

executeStep() - Execute a step of the garbled state machine, given the input values provided so far.
endecrypt(entryID, data) - Encrypt the given data block for storage under entry identifier entryID, or decrypt it if already encrypted.
/// hash(value) – Return a cryptographic hash of the given data value.

/// These are lexically defined locally within the contract, so I think they're effectively private. (Is that correct?)

/// @title Executable garbled state machine.
contract ExecutableMachine {

    /// This struct stores encrypted data for a given arc in the machine.
    struct EncryptedArc {
        uint256 encNext; // The arc's encrypted 'next state' entry.
        uint256 encValid; // The arc's encrypted 'valid' entry.
    }

    // Maximum number of supported time steps, L (1 <= L <= 65,535).
    uint16 constant maxSteps = 10; // In this simple example, we only support 10 steps.

    // Number of input lines (per step) to the machine, n (0<=n<=255).
    uint8 constant nInputs = 2; // There are only 2 inputs to this machine, named A and B.

    // Number of arcs (per step) in the machine, q (0 <= q <= 65,535).
    uint16 constant nArcs = 8; // This simple machine has only 8 arcs (2 states * 4 arcs)

    // Coded initial-state representation, 256 bits (32 bytes).
    uint256 sInit; // Initialized in constructor (with rest of machine).

    // Encrypted representations of all FSM arcs for all time steps.
    EncryptedArc[nArcs][maxSteps] arcs; // Initialized in constructor.

    // Next time step number to be executed, in range 0 to maxSteps.
    uint16 nextStep = 0; // The first time step to be executed is step #0.

    // Coded representation of current state, 256 bits (32 bytes).
    uint256 public curState; // Initialized to initial state in constructor.

    // Note: We mark 'curState' as 'public' because its getter is the only 'sanctioned' means at present of obtaining output from the machine.
Number of inputs received so far for the current time step.
uint8 nInputsRecd = 0; // Zero initially at the start of each step.

// Did we receive the given input yet on the current time step?
bool[2] gotInput = [false, false]; // False for all at the start of each step.

// All of the input values received so far, combined associatively.
uint256 combinedInputs = 0; // Zero at the start of each step.

---

// Constructor. [special public function]

constructor() public {
  // Coded representation of initial state.
  sInit = 0x712b98ce5a2a8ce5f8db66c52b6c1e7b2b06122e8168650b067did19ef6b;
  curState = sInit;

  // The actual "logic" of the garbled state machine is here. For each time step, for each arc index, we have an encrypted arc representation. The order of the arcs at each time step is random, and the encrypted data looks fully random to anyone not holding the input keys. Thus, this machine's function is fully obfuscated, to the level of only knowing its size parameters (number of time steps, input variables, and arcs).

  arcs[0][0] = EncryptedArc(0x8b40c8d1f7700cfcadf6dbbb91703fbeb2597928ac9af895eb1d1f417ce8,
                          0x2e735e54eb61b5a9e17893dcf5e3cc472ec4e6b8466b6c6e6f603305acc85e);
  arcs[0][1] = EncryptedArc(0x7042af3d7aa8de40a73ad98365ef79393488bce4d78c26f49f5e5796f6bc7,
                          0xed48b37b796a6f97540bd0d0d22d54997825107d5a8b794c74f7973ac2d1);
  arcs[0][2] = EncryptedArc(0xe2a879d562dfad7ace4d5a2862d9c9e8e06720e1c14702bf7665f6f861b2655131,
                          0xe8a764da7859f58024111bc93775f59cb11eefa58268949c6c5f5d76f1f487);
  arcs[0][3] = EncryptedArc(0xf1d6783650a502a69311e6a0977f12b9af327836f7e91c329fe0ed424e4ae,
                          0x2d9620ecaf2d71deb91a53e8dfaf05f383a3e554f2edf44993588e67e52f43e,}
0xa0f2e9b8d5a040bd6a0b3b23e079a7e67f3be6c8200e3354989c8f4a80e0a3a,
0x30980e9b3d7eb0360f6b782f1e84e9374a44b239397be5d9d40565d04d5,
0xe7f2e6c97611fbaa05c51c7e1931692646827684f8e49b2d4,
0x730bf8b42f2f2bfef68b4f841449f95143aad7cfb3dbbe6095a7350213d907b,
0x10480c94b2d601062a0e88702d464c05dbd1e4b214688a8e535553d322,
0x49b070d2a0b5e3dadd0c8d6573a68a82426152f23f29ef6a76c973930e6ed6,
0xa7f90c0476ef90e8f205014a7f9105ac51535b7c7d703da0731d393318,
0x63428ef31377034a52503c32a7a6712f86246fcfcbaf4e4ea3dab82a2287775,
0x87663f9e3f21c2da27575f3b3cf15951b8511d28e082e0c19e78bb7ebef3cb4a1431,

//------------------------
// Arcs for time step 5.

0x3f0a9f322b2aa786a970c0b136c1a2c984101ea252a6ab2e8d2f7c49f10d449a8d,
0x668f8e1e260a4bd87575a5d7459c41544f53105e23b03a0a80e604792da2a,
0x6b95314d4874da772da704941393e39327ed628b0ceee8a485e47d8e0e1413,
0x5557f495ee7d7c8e8a9d06a0b159281f02b23b17d607287a2eae2,
0xc6624250776e4643a639237b2e5ddb9a543793b905e8d13076995365,
0x4a0cfd7b996aef55f931af5e050e36b29f89c9f16b0f6db1d6131711984e976097,
0x4c28d4ed0dbb02e0a2e0bb26790d13d36ed36f03c4e1b48997011d2dbcb91ac9b,
0x82cceddd8b6006d4a16b67204a5731223e8390f6531a3ee869f663f311294d0d40,
0x2820c1acbbca7d23f3f8f52109a892e420d35f942d7d0e1b24172a9e4345a,
0x82cceddd8b6006d4a16b67204a5731223e8390f6531a3ee869f663f311294d0d40,
0x80b80e06d64e16b67204a5731223e8390f6531a3ee869f663f311294d0d40,
0x2820c1acbbca7d23f3f8f52109a892e420d35f942d7d0e1b24172a9e4345a,
0x2bb18f2775cb46c421918684d4606dc077496615934dc0721a4c6c8d6262122,
0xecd218fc767e23612bdfcc19ca4d3729540627a1a9f806417a308a33a531,

//------------------------
// Arcs for time step 6.

0x1502b9ad440b0512f6f3ebeb2da85796452979c071e7b93a127a9a31c9f9,
0x257f6f8a0f86d06a3f1a57690d4c8eb8bfbf30c907d7271461a76a3feceee42c,
0x216549f5d0c8bdfbb0fd49d3c9b7d0dece0a074ad7e46cf3f7db04d281506daced,
0x35321d1a2ef7703c3ac7e6e012d5c5719c3a4e725b3a525999290c18690898,
0x5590b558f3491e6c82f62105dd80459f8031c066573a71330a0000,
0x8222010338e0f8c350e36b56b9a3e575767d95ba5ef3e61b414b99272,
0x8222010338e0f8c350e36b56b9a3e575767d95ba5ef3e61b414b99272,
0x9814ec66b53b4af54e97d57522918a7d45945e4c702c0323a4c06746996a3638,
0xc6624250776e4643a639237b2e5ddb9a543793b905e8d13076995365,
0x5866c64a0e3004571e06267bd7c1741f79a5a4e669596705547e2a88f414c,
0xda19f4075aa9c95503709ad9422a2e0b901545239896726142f166,
0x4a5224a42e67240d31398c282fd98575ff9ec6b9ca6849bba4734ad83,
0x2bb18f2775cb46c421918684d4606dc077496615934dc0721a4c6c8d6262122,
0x5866c64a0e3004571e06267bd7c1741f79a5a4e669596705547e2a88f414c,
0x9814ec66b53b4af54e97d57522918a7d45945e4c702c0323a4c06746996a3638,
0xd93f575e60c96c092a9e53650799ad9422a2e0b901545239896726142f166,
0x4a5224a42e67240d31398c282fd98575ff9ec6b9ca6849bba4734ad83,
0x2bb18f2775cb46c421918684d4606dc077496615934dc0721a4c6c8d6262122,
0x5866c64a0e3004571e06267bd7c1741f79a5a4e669596705547e2a88f414c,
0xda19f4075aa9c95503709ad9422a2e0b901545239896726142f166,
Arcs for time step #8.

------------------------
// uint256 value - The 256-bit encrypted value of the provided variable for the current time-step.

// Error handling:

// If varIndex is out of range, or indicates an variable for which we have already received a value on this time step, the function has no effect.

// If invalid values are received for any input variables, after values have been received for all input variables, all provided values will be forgotten, and input collection for this time step starts over.

/** @dev Provide a value for an input variable for this time step. *
 * @param varIndex The index, 0 to nInputs-1, of the variable being provided.
 * @param value The 256-bit coded representation of the value being provided. */

function provideInput(uint8 varIndex, uint256 value) public {

    // Check for various error conditions.

    if (nInputsRecd == nInputs) return; // No more inputs needed. (SHOULD NEVER HAPPEN)
    if (varIndex >= nInputs) return; // Variable index out of range.
    if (gotInput[varIndex]) return; // Already got this input.

    // This code actually does the work of receiving the input.
    nInputsRecd++;
    gotInput[varIndex] = true;
    combinedInputs ^= value;

    // Here, we check whether all the inputs needed for this time step have now been provided. If they have, then we can go ahead and actually execute the time step, and then reinitialize our input-collection variables to prepare for the next time step.
    if (nInputsRecd == nInputs) {
        executeStep(); // Private internal function.
        // Reset the input-collection variables.
        nInputsRecd = 0;
        for (uint i=0; i<nInputs; i++)
            gotInput[i] = false;
        combinedInputs = 0;
    }
}

// Private/internal functions. [contract section]

---
/** @dev Attempt to carry out the next execution step of the machine. */

function executeStep() private {
    // Check for various error conditions.
    if (nInputsRecd != nInputs) return;  // Wrong # inputs received.  (SHOULD NEVER HAPPEN!)
    if (nextStep >= maxSteps) return;  // No more execution steps are supported.

    // Construct the arc identifier, by combining the (garbled)
    // current state ID with the combined input keys.
    uint256 arcID = curState ^ combinedInputs;

    // Construct the entry identifiers for the 'next-state' and
    // 'valid' entries from the arc identifier by combining it
    // with some arbitrary constants.
    uint256 nextID = arcID ^ (uint256(bytes32('n')) >> 248);
    uint256 validID = arcID ^ (uint256(bytes32('v')) >> 248);
    // NOTE: The '>>248' above is necessary to move the nonzero byte
    // representing 'n' or 'v' from the MSB to the LSB position, for
    // compatibility with our garbler.py code.

    // Search for the arc whose encrypted 'valid' entry is 0.
    bool foundIt = false;
    uint16 arcIndex;
    for (arcIndex = 0; arcIndex < nArcs; arcIndex++) {
        uint256 valid = endecrypt(validID, arcs[nextStep][arcIndex].encValid);
        if (valid == 0) {  // All 0's is our code meaning "this is the right arc"
            foundIt = true;
            break;
        }
    }

    // If we didn't find it, then return (don't update the state).
    if (!foundIt) return;

    // If we found it, then unencrypt the next state, and update our state.
    curState = endecrypt(nextID, arcs[nextStep][arcIndex].encNext);
    nextStep++;
} // End function ExecutableMachine.executeStep().
//| endecrypt()               [private pure function]   |
//| This private function, called internally       |
//| by executeStep(), uses a 256-bit key to         |
//| encrypt (or decrypt, if already encrypted)     |
//| a 256-bit data entry. It works similarly       |
//| to a one-time pad, by XOR'ing the data          |
//| with a 'random' pad that is computed as         |
//| the hash of the key. Unless the full key       |
//| is known, the result will appear                |
//| completely random. This encryption method       |
//| is unbreakable given standard assumptions       |
//| about the security properties of               |
//| cryptographic hash functions.                   |
//| Arguments:                                       |
//| uint256 entryID - This 256-bit value            |
//| identifies a specific data entry to be          |
//| encrypted or decrypted; this value is           |
//| used as the encryption/decryption key           |
//| for the data entry. "If you can name           |
//| it, you can access it" is the idea.             |
//| uint256 data - This 256-bit value is the        |
//| (plaintext or encrypted) entry data to          |
//| be encrypted or decrypted, respectively.        |
//| Return value:                                    |
//| uint256 res - The 256-bit result of the         |
//| encryption or decryption of the data.           |

/** @dev Encrypt/decrypt a data value keyed by a given entry identifier.  
 * @param entryID The entry identifier, which is a 256-bit derived key.   
 * @param data The data to be encrypted or (if already encrypted) decrypted.  
 * @param res Result of the encryption/decryption. 
 */
function endecrypt(uint256 entryID, uint256 data) private pure returns (uint256 res) {
    res = hash(entryID) ^ data;  // This is like a one-time pad of the data.
}  // End function ExecutableMachine.endecrypt().

/** @dev Hash function to be used in this program. 
 * @param value A 256-bit data value to be hashed. 
 */
function hash() private pure returns (uint256) {
    // This private function is called internally from within endecrypt() to compute a 256-bit cryptographic hash of a 256-bit data value.
    // Any suitable hash function could be used, but we use sha256() for the time being.
    // Argument:
    // uint256 value - A 256-bit data value to be hashed.
    // Return value:
    // uint256 h - The 256-bit hash of the data.
}
* @param value 256-bit value to be hashed.
* @return h The 256-bit hash of that value.

function hash(uint256 value) private pure returns (uint256 h) {
    h = uint256(keccak256(abi.encodePacked(value)));
    // ^^^^^^^^^ sha256 (256-bit SHA-2) is another available option.
}

} // End function ExecutableMachine.hash().

} // End contract ExecutableMachine.

//|^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^|
//|                     END OF FILE:   ExecutableMachine.sol                   |
//|============================================================================|
APPENDIX D DEVELOPMENT TESTING

Below are some steps for exercising the above example contract **ExecutableMachine** of an executable smart contract **Exec[G]** for a garbled state machine **G** in a development environment.

For this, we used the Truffle development environment (https://truffleframework.com/), which makes development testing very easy. Once Truffle has been installed on a development machine, the following steps will suffice to exercise the contract.

1. Create a working directory and cd into it:
   
   ```
   $ mkdir truffle_project
   $ cd truffle_project
   ```

2. Initialize the directory to contain a new Truffle project:

   ```
   $ truffle init
   ```

3. Place the contract file **ExecutableMachine.sol** into the **contracts** subdirectory.

4. In the **migrations** subdirectory, create the file **2_deploy_contracts.js** as follows:

   ```javascript
   var ExecutableMachine = artifacts.require("ExecutableMachine");
   module.exports = function(deployer) {
       deployer.deploy(ExecutableMachine);
   }
   ```

5. Compile the contract:

   ```
   $ truffle compile
   ```

6. Start a development blockchain and enter the Truffle console:

   ```
   $ truffle develop
   ```

7. (Within the Truffle console) Migrate the contract onto the blockchain:

   ```
   truffle(develop)> migrate
   ```

8. At this point, the contract should be deployed on the development blockchain, and its constructor executed. Now we can type interactive JavaScript commands at the Truffle console to call functions of the contract, such as the following (but omitting the line breaks):

   ```
   truffle(develop)> ExecutableMachine.deployed().then(
       inst => inst.curState.call()
     ).then(
         val => val.toString(16)
     );
   ```

9. This calls the **curState** getter function and converts the result to a hexadecimal string. Note that the returned value (shown above in green) is the same constant sInit that the **curState** state variable was initialized to in the contract’s constructor function. This shows that the constructor was successfully executed, and the garbled machine is sitting in its initial state.

10. If the above works, the Machine is ready to be taken through a more complete test. We can write a test bench script to do this, using input keys produced by **garbler.py** (Appendix B) as follows. This goes in a file named **test1.sol** in the **test** subdirectory.
pragma solidity ^0.4.24;

import "truffle/Assert.sol"; // Allows asserting conditions to be tested.
import "truffle/DeployedAddresses.sol"; // For obtaining address of deployed contract.
import "../contracts/ExecutableMachine.sol"; // Allows us to call functions of the contract.

contract test1 {

    ExecutableMachine em = ExecutableMachine(DeployedAddresses.ExecutableMachine());

    function hash(uint256 value) private pure returns (uint256 h) {
        h = uint256(keccak256(abi.encodePacked(value)));
    }

    function testRun() public {
        // Take the sample machine through a 3-step run.
        // Coded keys of provided inputs for time step #0. State transition: SInit -> SReset.
        em.provideInput(0, hash(0xef45564b403fca79f6082cf6b4eaccf61ab7f55710433dfd8ec9b15f470257c"em[curState()]")); // A:=0
        em.provideInput(1, hash(0xa24009f7f00d5cda3441cf759a563b0f9a4d49373c2d326d7e09f8e3f47291d"em[curState()]")); // B:=1
        // Coded keys of provided inputs for time step #1. State transition: SReset -> SInit.
        em.provideInput(0, hash(0x7e5e55a46b6a9c9bf4566247e4e6ccd65c7df6asead2c4dfaf72e879dc3a1b96"em[curState()]")); // A:=1
        em.provideInput(1, hash(0x249e225232bf8b3ddd5b2a09acdf7d14a1d3e4a40c17a66aa4a26ef06d028a"em[curState()]")); // B:=0
        // Coded keys of provided inputs for time step #2. State transition: SInit -> SPass.
        em.provideInput(0, hash(0x7ff2e199244f261b1310311d103712cde4d354baa97b0b94f23a4ec853137b"em[curState()]")); // A:=1
        em.provideInput(1, hash(0x09805b1dd58a23797b847bf9dfe5f2362eaf207b230f50d1eeef45e2383969d"em[curState()]")); // B:=1
        // At this point, we should be in the halting state SPass at time t=3, which is
coded as 0x14eeba216d0cd87ef7ec29449e7fbd1e906b064d438d2f783cf2b7b7603e4a1.
        uint cs = em.curState();
        uint expectedState = 0x14eeba216d0cd87ef7ec29449e7fbd1e906b064d438d2f783cf2b7b7603e4a1;
        Assert.equal(cs, expectedState, "curState should be 0x14ee...4a1");
    }
}

11. At this point, we can run the test bench as follows:

$ truffle test test1.sol

12. If it works, we will see output something like the following:

Using network 'development'.
Compiling ./contracts/ExecutableMachine.sol...
Compiling ./test/test1.sol...
Compiling truffle/Assert.sol...
Compiling truffle/DeployedAddresses.sol...

test1
✓ testRun (821ms)

1 passing (2s)

The check mark and “1 passing” mean that this single test succeeded. The machine was taken by the inputs through the (coded) state sequence SInit -> SReset -> SInit -> SPass.
## DISTRIBUTION

### Email—Internal

<table>
<thead>
<tr>
<th>Name</th>
<th>Org.</th>
<th>Sandia Email Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nicholas D. Pattengale</td>
<td>05682</td>
<td><a href="mailto:ndpatte@sandia.gov">ndpatte@sandia.gov</a></td>
</tr>
<tr>
<td>Michael P. Frank</td>
<td>01421</td>
<td><a href="mailto:mpfrank@sandia.gov">mpfrank@sandia.gov</a></td>
</tr>
<tr>
<td>Christopher Cordi</td>
<td>05684</td>
<td><a href="mailto:cncordi@sandia.gov">cncordi@sandia.gov</a></td>
</tr>
<tr>
<td>Carollan Beret Helinski</td>
<td>05646</td>
<td><a href="mailto:cbehn@sandia.gov">cbehn@sandia.gov</a></td>
</tr>
<tr>
<td>Ryan Kao</td>
<td>05683</td>
<td><a href="mailto:rkao@sandia.gov">rkao@sandia.gov</a></td>
</tr>
<tr>
<td>Technical Library</td>
<td>01977</td>
<td><a href="mailto:sanddocs@sandia.gov">sanddocs@sandia.gov</a></td>
</tr>
</tbody>
</table>

### Email—External (encrypt for OUO)

<table>
<thead>
<tr>
<th>Name</th>
<th>Company Email Address</th>
<th>Company Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vladimir Kolesnikov</td>
<td><a href="mailto:kolesnikov@gatech.edu">kolesnikov@gatech.edu</a></td>
<td>Georgia Inst. of Tech.</td>
</tr>
<tr>
<td>Kasimir G. Gabert</td>
<td><a href="mailto:kasimir@gatech.edu">kasimir@gatech.edu</a></td>
<td>Georgia Inst. of Tech.</td>
</tr>
<tr>
<td>Abraham K. Ladha</td>
<td><a href="mailto:abrahamladha@gatech.edu">abrahamladha@gatech.edu</a></td>
<td>Georgia Inst. of Tech.</td>
</tr>
</tbody>
</table>

### Hardcopy—Internal

<table>
<thead>
<tr>
<th>Number of Copies</th>
<th>Name</th>
<th>Org.</th>
<th>Mailstop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Hardcopy—External

<table>
<thead>
<tr>
<th>Number of Copies</th>
<th>Name</th>
<th>Company Name and Company Mailing Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This page left blank
This page left blank