Verifiable state machines: Proofs that untrusted services operate correctly

Srinath Setty
Microsoft Research

Sebastian Angel
University of Pennsylvania

Jonathan Lee
Microsoft Research

Abstract

This article describes recent progress in realizing *verifiable state machines*, a primitive that enables untrusted services to provide cryptographic proofs that they operate correctly. Applications of this primitive range from proving the correct operation of distributed and concurrent cloud services to reducing blockchain transaction costs by leveraging inexpensive off-chain computation without trust.

1 Introduction

When a database-backed service is hosted on untrusted infrastructure (e.g., the cloud), how can the service prove to its clients that it is operating according to a pre-defined specification? An example service is a payment system that enables financial institutions, such as banks, to transact with one another. But, more generally, it could be any cloud service where incorrectness can have serious security or financial consequences, for example, a service that hosts online video games such as Poker, a public key directory service for end-to-end encryption, an auction service, etc.

Providing proofs of correct operation eliminates the need for clients to trust the service or any of the infrastructure on which the service is hosted. This includes the hardware and software stacks of a cloud provider, and the implementation of the service itself—including the distributed software infrastructure used to achieve high availability, fault-tolerance, consistency, low latency, etc. Note that this problem is different from, but complementary to, a large body of work on formal methods and program verification, which aims to prove equivalence between a program and its formal specification. While program verification can categorically rule out a large class of bugs, it does not eliminate issues that stem from an incorrect *execution* of a program. Such issues could arise from hardware failures (e.g., bit flips), misconfigurations, unverified code, interactions among verified and unverified components, a compromised service, or a malicious service provider that actively tampers with execution (perhaps to hand your opponent that royal flush!).

In the status quo, solutions to this problem include trusting the service and its infrastructure, or employing trusted execution environments (TEEs) such as Intel SGX enclaves. While the former offers no protection against the issues listed above, the latter is a bit too optimistic. Even on their best day, TEEs are plagued by bugs and a never-ending laundry list of side channels and microarchitectural vulnerabilities. A sophisticated adversary (or really just about anyone who knows how to undervolt a processor [29]) can exploit these vulnerabilities to violate the integrity guarantees of TEEs, or extract confidential key material from TEEs and forge incorrect responses.

In this article, we focus instead on a recent line of work designed specifically to answer our animating question—even when the service is distributed and concurrent, and even when it operates on secret state (e.g., the seed of the random number generator should be kept secret from players of a game as otherwise they could cheat). The provided abstraction is that of a general state machine that can prove the correctness of its state transitions using cryptography. We call such machine a *verifiable state machine* (VSM) [10, 25, 35]. Specifically, in a VSM, an untrusted service, in addition to executing requests, runs a program called a *prover*. The role of this prover program is to produce short *proofs* that another program called a *verifier*, which typically runs on clients' machines, can check to ensure that the service executed its requests in accordance with a pre-defined specification. A key property here is that the verifier can check these proofs without having to reexecute any requests. In fact, the verifier does not even need to know the service's internal state or the actual content of any request in order to be able to verify these proofs.

Because of these properties, the potential applications for VSMs extend far beyond the cloud. For example, the Libra blockchain, modeled as a service implemented by a distributed system, can make itself verifiable by producing proofs, which obviates the need to publish a public ledger of raw transactions. Similarly, a user can run the prover on their mobile device to produce proofs about their personal information (e.g., prove that the user's age is over 18 using a digital credential such as Estonia's digital identity card) without revealing anything else about the underlying data.

Algorithms to produce and verify such proofs exist since the early 1990s [3–5, 21, 28]. But, it is only in the last decade that researchers began to build systems that actually run on real hardware and that can prove their correct executions [14, 36, 37, 39, 40]. In this article, we will focus on the latest frameworks for implementing VSMs, namely Spice [35] and Piperine [25]. Our goal is to introduce concepts, explain how developers can use these frameworks to build VSMs, and describe open problems in the area.

2 Verifiable state machines (VSMs)

To employ verifiable state machines, one must express a service as a state machine $\mathcal{M} = (\mathcal{R}, \mathcal{S}_0)$, where $\mathcal{R}$ is a request-handler program that specifies state transitions and $\mathcal{S}_0$ is the initial state of the machine.

In the payment service example, the state machine's internal state is set of accounts maintained in a key-value store where keys are identities of participants and values are balances of the corresponding participants. A state transition here is an operation that transfers balances from a sender's account to a receiver's account. For authentication, a simple scheme is where identities in the key-value store are public keys in a digital signature scheme and transfer operations are signed by the private key of the sender account's owner.

2.1 Setup and properties

The service publishes its state machine's program $\mathcal{R}$ and a digest of its initial state $\mathcal{H}(\mathcal{S}_0)$, where $\mathcal{H}$ is a collision-resistant hash function. In Spice [35], $\mathcal{R}$ is expressed in a broad subset of C, which includes functions, structs, typedefs, preprocessor macros, if-else statements, statically bounded loops, explicit type conversions, and standard integer and bitwise operations. For internal state, Spice offers: (1) a block store with `GetBlock`/`PutBlock` APIs; and (2) a key-value store with a standard `get`/`put` interface. To support concurrent executions where multiple threads execute a batch of requests in parallel, $\mathcal{R}$ can use synchronization primitives (e.g., `lock`/`unlock`) and simple transactions.

For each batch of requests executed by the service, the service runs a *prover* $\mathcal{P}$ to produce a proof. The proof, in addition to proving the correct authentication of operations, proves sequential consistency [23] for single-object operations (where an object is a key-value pair) and serializability [8, 31] for multi-object transactions. Anyone that wishes to verify the correct operation of the service, for example an *auditor*, runs a *verifier* $\mathcal{V}$, which either accepts or rejects the proof produced by the service. Informally, VSMs ensure the following guarantees under a set of cryptographic assumptions.

- **Completeness.** If the service correctly executes the requests according to its specification in $\mathcal{R}$ while respecting the memory consistency and transaction isolation semantics, then $\mathcal{P}$ can produce a proof such that $\mathcal{V}$ accepts.
- **Soundness.** If the service misbehaves during its execution, then $\Pr\{\mathcal{V}\text{accepts any proof}\} \leq \epsilon$ (e.g., $\epsilon \leq 1/2^{128}$).
- **Zero-knowledge.** $\mathcal{V}$ learns nothing about the internal state of the machine, requests, or the responses.
- **Succinctness.** The size of each proof should be small (e.g., a few hundred bytes).

2.2 Mechanics of verifiable state machines

Spice [35] proceeds in two steps. First, it transforms the task of proving the correct execution of state machine transitions to the task of proving the satisfiability of a system of equations (these equations are derived from the program's specification and are known to both the prover and the verifier). Second, Spice employs a cryptographic machinery, called an *argument protocol*, where the prover produces a proof that establishes that it knows a solution to the system of equations without having to actually send that solution to the verifier; the verifier can efficiently check the proof. If the check passes, the verifier determines that the prover indeed knows a solution, and must have therefore executed the state transitions correctly. We now provide details of each of these steps.

(1) Compiling state machine executions to equations. This step is performed by a compiler. The specific system of equations used by Spice is called rank-1 constraint satisfiability (R1CS), an NP-complete problem analogous to circuit satisfiability. At a high level, R1CS is a system of equations where variables take values from a finite field, for example $\mathbb{F}_p = \{0, 1, \ldots, p-1\}$ for a large prime p, and each equation can have at most one multiplication operation between variables (many multiplications by constants are fine).

The compiler operates line-by-line over $\mathcal{R}$: loops are unrolled and then each program statement is compiled to one or more equations using a pre-designed set of recipes. The key invariant maintained by the compiler is that the satisfiability of the system of equations is tantamount to correct program execution. To illustrate this, consider a toy computation and its equivalent system of equations (uppercase letters denote variables and lowercase letters denote concrete values):

```
int incr(int x) {
   int y = x + 1;
   return y;
}
```

$$\mathcal{C} = \left\{ \begin{array}{rcl} X - x & = & 0 \\ Y - (X+1) & = & 0 \\ Y - y & = & 0 \end{array} \right\}$$

For the above equations, if $y = x + 1$, $\{X \leftarrow x, Y \leftarrow y\}$ is a solution. If $y \neq x + 1$, then there is no solution and the system of equations is not satisfiable.

We now describe a more complex example.

```
int inc_and_neq(int x1, int x2) {
    int z = x1 + 1;
    return y = (z != x2);
}
```

$$\mathcal{C} = \left\{ \begin{array}{rcl} Z_1 - (x_1 + 1) & = & 0 \\ Z_2 \cdot (Z_1 - x_2) - Y & = & 0 \\ (1 - Y) \cdot (Z_1 - x_2) & = & 0 \\ Y - y & = & 0 \end{array} \right\}$$

To show that these equations are equivalent to the program, we note a few things.

- The first equation is satisfied only when $Z_1 = x_1 + 1$.
- Y (which must equal the value of y from the last equation) can only have a value of 0 or 1. This is because the second and third equations cannot simultaneously be satisfied when Y is not a 0 or 1. This shows how even though the domain of the variables is a finite field (integers modulo p), the valid values of Y can be restricted to 0 or 1. There are two cases here.
    1. When $Z_1 = x_2$, then Y must be 0. Otherwise, the second equation cannot be satisfied since $Z_2 \cdot (0) - Y = 0$ has only one solution: $Y = 0$.
    2. When $Z_1 \neq x_2$, then Y must be 1. Otherwise, the third equation cannot be satisfied since $(1 - Y) \cdot M = 0$ (for $M \neq 0$) only has one solution: $Y \equiv 1 \mod p$.

Prior work provides details of how different program constructs (e.g., conditionals, comparisons, bitwise operations, etc.) can be translated to rank-1 constraints [32, 38, 39]. The next subsection discusses how to encode operations on state using constraints.

(2) Proving the satisfiability of constraints succinctly and in zero-knowledge. The prover identifies a solution to the equations using a given input x. The prover cannot send its solution to the verifier because it would violate zero-knowledge. It also violates succinctness since the size of the solution is proportional to the running time of the state machine transition. Instead, Spice employs an argument protocol, referred to as a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) [9, 19], to encode the prover's solution as a small proof π.

zkSNARKs rely on probabilistic proof protocols (such as probabilistically checkable proofs, interactive proofs, etc.) in conjunction with cryptographic primitives. Unlike mathematical proofs, probabilistic proofs rely on randomization. For example, with interactive proofs, the prover responds to a sequence of random challenges from the verifier such that at the end of the interaction, if the system of equations is not satisfiable, the probability that a cheating prover convinces a verifier is negligible. Such interactive proofs are turned into non-interactive proofs (i.e., a one-shot message from the prover to a verifier) through a standard transformation [17] where the verifier's challenges are simulated using a hash function over the prover's messages. Similarly, there are standard transformations to add zero-knowledge [7, 12, 16, 18, 43, 45].

To add a bit more detail, a recent high-speed zkSNARK, called Spartan [34], transforms a system of equations such as the one above into a polynomial G such that $\sum_{x \in \{0,1\}^\ell} G(x) = 0$ if and only if the system of equations is satisfiable, where ℓ depends on the size of the equation set (number of equations, number of variables, etc.). Spartan's prover employs an interactive proof protocol called the sum-check protocol [26] in conjunction with a cryptographic primitive called polynomial commitments [20, 42] to prove that the above sum of polynomial evaluations is indeed zero, which implies the satisfiability of equations, which in turn implies correct program execution.

2.3 Supporting state in the constraints formalism

The constraints formalism is essentially stateless, for example, there is no notion of RAM or a persistent storage abstraction. To support these constructs, Spice and other works employ cryptography to encode checks rather than encoding state operations themselves.

To illustrate this concept, consider the following program that takes as input a *digest* (e.g., a SHA-256 hash) and accesses a block in a content-addressable block store using `GetBlock`/`PutBlock` APIs [10]:

```
Digest increment(Digest d) {
  // produces equations that check d == Hash(b)
  Block b = GetBlock(d);
  Block b_new = b + 1;
  // produces equations that check d_new == Hash(b_new)
  Digest d_new = PutBlock(b_new);
  return d_new;
}
```

Spice's compiler translates each `GetBlock` call to a set of equations that check if the hash of the returned block equals the argument digest. This requires representing a hash function, such as SHA-256, as a set of constraints, which is possible because computing a hash function is pure computation. `PutBlock` translates to a similar set of constraints. Because of these checks, unless the prover (or the untrusted service) identifies a hash collision (which is computationally infeasible), the prover (and the untrusted service) is compelled to supply the correct block to each `GetBlock` and supply the correct digest as the response to each `PutBlock`.

Pantry [10] shows how to build more expressive abstractions, such as RAM and a key-value store, using `GetBlock`/`PutBlock`. The core idea is to represent state using a Merkle tree [27], and then use `GetBlock`/`PutBlock` APIs to implement read and write operations on state. The verifier maintains the root of the Merkle tree. Each state transition proves the correct execution of a program whose input is the current root of the Merkle tree and outputs an updated root of the Merkle tree, which is used in the next state transition. The example below illustrates this idea.

```
// Suppose d is a digest representing the root
// of a searchable Merkle tree with n nodes
// (a searchable merkle tree is an AVL tree where
// internal nodes store the hash of their children,
// along with a (key, value).
// This root succinctly represents the database.
// Example code to implement get(key)
Value get(Digest d, Key key) {
  Value y = null;
  for (i = 0, i < log(n); i++) {
    // prover supplies a block that contains
    // two hashes of children and a (key,val) pair
    Block block = GetBlock(d);

    if (key == block.key)
      y = block.val
    else if (key < block.key)
      d = block.left
    else
      d = block.right
  }
  return y;
}
```

Unfortunately, the root of a Merkle tree acts as a point of contention, which requires a serial execution of state transitions. We now discuss a key innovation in Spice [35], where by using a set data structure, the service can execute a batch of state transitions concurrently.

A set-based key-value store. Spice supports a key-value store using a particular type of hash function $\mathcal{H}(\cdot)$ that operates on sets and is incremental [2, 6, 13]: given a *set-digest* d_S for a set S, and a set W, one can efficiently compute a set-digest for $S \cup W$. Specifically, there is a constant time operation $\oplus$ where: $\mathcal{H}(S \cup W) = \mathcal{H}(S) \oplus \mathcal{H}(W) = d_S \oplus \mathcal{H}(W)$.

Spice encodes a key-value store into two sets: a read-set *RS* and a write-set *WS*. These sets contain (*key*, *value*, *timestamp*) tuples for every operation on the store. Neither the prover nor the verifier materializes these sets; they only operate on them using the corresponding digest. Specifically, the verifier's digest of the key-value store is:

```
struct KVDigest {
  SetDigest rs; // a set-digest of RS
  SetDigest ws; // a set-digest of WS
}
```

Example. If the key-value store is empty, $rs = ws = \mathcal{H}(\{\})$. Suppose the prover executes a program $\mathcal{R}$ that invokes `insert`(k, v), it is forced to return an updated `KVDigest` such that the following holds (this is done by translating `insert` into appropriate constraints, as in the `GetBlock` example): $rs = \mathcal{H}(\{\}), ws = \mathcal{H}(\{(k, v, 1)\})$.

Now, suppose the prover executes $\mathcal{R}'$ that invokes `get`(k), the prover is expected to supply a value and update the timestamp associated with the tuple. To explain how `KVDigest` is updated, there are two cases to consider.

First, suppose the prover behaves correctly and returns the value that was previously stored by $\texttt{insert}(k, v)$, then: $rs = \mathcal{H}(\{(k, v, 1)\}), ws = \mathcal{H}(\{(k, v, 1), (k, v, 2)\})$. A key invariant here is that whenever the prover executes the key-value store operations correctly, the set bound to *rs* is a subset of the set bound to *ws*. To illustrate the invariant further, consider the second case where the prover returns $v' \neq v$ (for $\texttt{get}(k)$), then the set-digests returned by the prover will be: $rs = \mathcal{H}(\{(k, v', 1)\}), ws = \mathcal{H}(\{(k, v, 1), (k, v', 2)\})$.

Observe that the set bound to *rs* is not a subset of the set bound to *ws*. However, the verifier cannot detect this: set-digests, like other kinds of hashes (e.g., SHA256), are just a string of bits (or a corresponding large integer) with no structure to check the subset property. Instead, the verifier requires the prover to produce a special proof π_{audit} periodically (e.g., for a sequence of inputs $x_1, \ldots x_n$) that proves the set bound by *rs* is a subset of the set bound by *ws*. To do so, the prover's π_{audit} proves the following (the check below is represented as a system of equations and then proved using a zkSNARK):

$$\exists\{(k_i, v_i, ts_i)\} : ws \ominus rs = \mathcal{H}(\{(k_i, v_i, ts_i)\}) \wedge \forall i, k_i < k_{i+1},$$

where $\ominus$ is the inverse of $\oplus$ (i.e., it results in removal of elements from a set bound to a digest). The set of key-value-timestamp tuples whose digest is $ws \ominus rs$ is, when the prover is honest, exactly the key-value-timestamp tuples in its current state. To produce π_{audit}, the prover incurs costs linear in the number of key-value tuples, but the linear cost is amortized over all the n inputs $x_1, \ldots x_n$ (which motivates producing π_{audit} periodically rather than for every operation).

3 Applications of verifiable state machines

We now discuss three applications that employ VSMs.

3.1 Privately auditing a cloud-hosted payment service

The first application is our running example of a payment service hosted on the cloud. Specifically, consider a service that maintains balances of assets of different clients (e.g., banks). Clients submit `transfer`, `issue`, and `retire`. `transfer` moves an assert from one client to another, whereas `issue` and `retire` move external assets in and out of the system (`issue` and `retire` could be privileged operations that only certain authorities such as the Central Bank can submit). To naively verify the correct operation of such a service, an auditor needs access to sensitive details of clients' requests (e.g., the amount of money) and the service's state. VSMs offers zero-knowledge auditing, where an auditor learns nothing beyond the correct operation of the service. The balances maintained by the service is the VSM's state and the request types discussed above are state transitions.

3.2 Verifiable auction platform

Online auctions are widespread and are used to sell everything from luxury items such as art and watches, to every-day goods such as shirts and even food. Auctions are also at the core of online advertising platforms such as Google's Doubleclick or Facebook's Ad Exchange. Existing auction platforms proceed on trust, with bidders and sellers trusting that, for each auction, the auctioneer chooses the correct winner, charges the correct price, and takes the correct commission. However, auctioneers might have financial incentives to deviate from the prescribed protocol. For example, an auctioneer may accept a side payment that is larger than its expected commission by declaring a colluding low bidder as the winner of the auction (so both the auctioneer and the bidder win at the expense of the seller).

VSMs can be used to endow an auction service (e.g., an ad exchange) with verifiability while maintaining all bids—which could be proprietary—secret from any auditor. The overall approach is similar to prior privacy-preserving verifiable auctions platforms [1], but VSMs are more general and can capture richer and more complex protocols. In a nutshell, the auctioneer stores bids in the VSM's state, perhaps using sealed bids as in VEX [1]. Once the auction closes (which could be in milliseconds in the case of ads, or weeks in the case of luxury goods), the auctioneer can output the auction's outcome, along with a proof establishing the correctness of the result. Clients can check this proof in zero-knowledge without learning the value of any bid.

3.3 Reducing the costs of large-scale replicated systems

Replication is a classic systems technique to achieve fault-tolerance, where a deterministic service, expressed as a state machine, is executed on a distributed set of replicas such that they collectively emulate a correct service—despite failures of a subset of the replicas and the network that connects the replicas [22, 24, 33]. A blockchain network is a large-scale replicated system in a Byzantine fault model, meaning that nodes can deviate arbitrarily from their prescribed behavior. Unfortunately, given a large number of replicas, the cost of replicated execution is extremely expensive.

A promising approach, demonstrated by Piperine [25] and several other works [11, 30, 44], is to employ VSMs where an untrusted service aggregates a batch of transactions, executes the state machine replicated by a blockchain, and produce proofs. The blockchain then only replicates the verifier to verify the proofs produced by the prover. Since verifying a proof is cheaper than executing the corresponding batch of state machine transitions (for a sufficiently large batch size), employing VSMs leads to lower costs than naively replicating execution. This, of course, assumes that there are enough replicas to amortize the cost of producing proofs, which we find to be the case in our blockchain instantiation of Piperine. Furthermore, the size of the succinct proofs produced by Piperine for a batch of transactions is smaller (by constant factors) than the size of a batch of raw transactions, so Piperine's proofs act as compressed information, which reduces network costs besides saving computation.

Note that the use of Piperine does not introduce new assumptions for safety or liveness of a replicated system. Specifically, a powerful property ensured by Piperine is that the service that produces proofs is untrusted for both safety and liveness: the proofs produced not only prove the correct execution of state machine transitions, but they also produce state changes that result from the execution, which nodes in the blockchain verify as part of proof verification and are persisted on the blockchain (note that in this application we do not preserve the zero-knowledge property of VSMs in favor of preserving liveness). In other words, Piperine offers a generic mechanism to transform any replicated state machine (RSM) $\mathcal{RSM}$ into another RSM $\mathcal{RSM}'$ such that the latter incurs lower execution costs—while retaining the safety and liveness properties of $\mathcal{RSM}$.

Details of application to Ethereum. Ethereum is a blockchain network that instantiates a large-scale replicated state machine. In Ethereum, state consists of a set of accounts, each of which possesses a balance in a currency (ether). Optionally, each account can possess bytecode written for the Ethereum Virtual Machine (EVM) and internal persistent storage. Such bytecode is called a smart contract and can be deployed to an account by a developer; this facility can be used to implement decentralized applications such as payment services, games, auctions, etc.. State transitions (also known as transactions) in Ethereum consist of transfers of balances between accounts, deploying new smart contracts, and calls to methods exposed by smart contracts (which in turn can make calls to other smart contracts).

A simple approach to leverage Piperine is by enhancing an individual Ethereum smart contract; this requires no modification to the underlying Ethereum platform and so Piperine can be deployed transparently. Under this approach, developers re-implement their smart contract as a state machine $(\Psi, \mathcal{S}_0)$ using Piperine's toolchain. Clients who wish to invoke the smart contract submit their transactions to one or more untrusted services that run the prover. These untrusted services aggregate transactions, execute them in batches, and produce proofs that are then sent to a verifier. Piperine implements the verifier as a smart contract that runs naively on Ethereum, verifies proofs produced by an untrusted service, and keeps track of the application state. In practice, Piperine reduces per-transaction costs by 5.4×, as the replicating the verifier computation is cheaper than replicating the execution of transactions. In addition, Piperine reduces network costs by 2.7× since a batch of transactions is compressed to a succinct specification of their net effect.

4 Discussion, open problems, and outlook

The current state-of-the-art systems for VSMs represent massive progress. For example, Spice [35] achieves a throughput that is 4–5 orders of magnitude higher than its predecessors [10, 15]. A principal issue with Spice and Piperine is that they require executing state transitions in batches to amortize the linear cost of producing π_{audit}. As a result, these systems incur higher latency than their predecessors that do not support concurrent executions. A key open problem is to reduce latency without reducing throughput. Another problem with Spice's set-based key-value store, unlike a Merkle-tree-based key-value store, is that it does not support efficient reads—even when one does not require zero-knowledge (as in our third application). Specifically, given a `KVDigest`, there is no efficient proof that a certain key has a certain value: either the prover or the verifier must incur a linear-time cost.

VSMs do not prevent an untrusted service from *equivocation*: the service can expose different sequences of proofs to different auditors (or verifiers). This does not arise in our third application as the blockchain that replicates the verifier prevents any equivocation by the untrusted service. For other settings, one can use a similar solution [41] to agree on a single sequence of proofs. Besides equivocation, an untrusted service can omit clients' requests. To address this, clients must check if their requests are included in the sequence of proofs agreed upon by the auditors.

More fundamentally, VSMs do not ensure the confidentiality of state from the untrusted service. This is a key difference with homomorphic encryption (HE) or even Intel SGX (without side channels) where the data can also be hidden from a service provider. A straw man approach would combine VSMs with a flavor of HE, but state-of-the-art HE schemes do not support storage abstractions nor concurrent executions. Furthermore, the overheads from employing HE will be far too high to be practical. Addressing this issue would open up a new domain of applications.

Acknowledgments. Sebastian Angel is supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] S. Angel and M. Walfish. Verifiable auctions for online ad exchanges. In *Proceedings of the ACM SIGCOMM Conference*, 2013.

[2] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45(3), May 1998.

[4] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM (JACM)*, 45(1):70–122, Jan. 1998.

[5] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1991.

[6] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1997.

[7] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali, and P. Rogaway. Everything provable is provable in zero-knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 37–56, 1988.

[8] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[9] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2012.

[10] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[11] V. Buterin. On-chain scaling to potentially 500 tx/sec through mass tx validation. `https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477`, Sept. 2018.

[12] A. Chiesa, M. A. Forbes, and N. Spooner. A zero knowledge sumcheck and its applications. *CoRR*, abs/1704.02086, 2017.

[13] D. Clarke, S. Devadas, M. V. Dijk, B. Gassend, G. Edward, and S. Mit. Incremental multiset hash functions and their application to memory integrity checking. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2003.

[14] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2012.

[15] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[16] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 424–441, 1998.

[17] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 186–194, 1986.

[18] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[19] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 99–108, 2011.

[20] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 177–194, 2010.

[21] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1992.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, July 1978.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[25] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[26] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1990.

[27] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the International Cryptology*

Conference (CRYPTO), 1988.

[28] S. Micali. CS proofs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1994.

[29] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[30] A. Ozdemir, R. S. Wahby, and D. Boneh. Scaling verifiable computation using efficient set accumulators. Cryptology ePrint Archive, Report 2019/1494, 2019.

[31] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4), Oct. 1979.

[32] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[33] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[34] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[35] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018.

[36] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.

[37] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.

[38] S. Setty, V. Vu, N. Panpalia, B. Braun, M. Ali, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality (extended version). ePrint Report 2012/598, 2012.

[39] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[40] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[41] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[42] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[43] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[44] B. WhiteHat, A. Gluchowski, HarryR, Y. Fu, and P. Castonguay. Roll_up / roll_back snark side chain ~17000 tps. `https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/3675`, Oct. 2018.

[45] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. ePrint Report 2019/317, 2019.