VerSum: Verifiable Computations over Large Public Logs

Jelle van den Hooff
MIT CSAIL

M. Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

ABSTRACT

VERSUM allows lightweight clients to outsource expensive computations over large and frequently changing data structures, such as the Bitcoin or Namecoin blockchains, or a Certificate Transparency log. VERSUM clients ensure that the output is correct by comparing the outputs from multiple servers. VERSUM assumes that at least one server is honest, and crucially, when servers disagree, VERSUM uses an efficient conflict resolution protocol to determine which server(s) made a mistake and thus obtain the correct output.

VERSUM's contribution lies in achieving low server-side overhead for both incremental re-computation and conflict resolution, using three key ideas: (1) representing the computation as a functional program, which allows memoization of previous results; (2) recording the evaluation trace of the functional program in a carefully designed computation history to help clients determine which server made a mistake; and (3) introducing a new authenticated data structure for sequences, called SEQHASH, that makes it efficient for servers to construct summaries of computation histories in the presence of incremental re-computation. Experimental results with an implementation of VERSUM show that VERSUM can be used for a variety of computations, that it can support many clients, and that it can easily keep up with Bitcoin's rate of new blocks with transactions.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Verifiable Computation; Cloud Computing

1. INTRODUCTION

Systems such as Bitcoin [15] provide publicly available logs, whose validity is guaranteed. The logs are large (e.g., the Bitcoin blockchain was 14 GB as of January 2014) and many records are added daily (e.g., on average every 10 min an entry is added to the Bitcoin blockchain). To run computations over these logs requires a powerful computer, because a computation may require processing the entire log, which can take a long time (a few hours) and requires the computer to store the whole log.

VERSUM is a new system for securely outsourcing computations over frequently changing inputs, supporting many clients. With VERSUM, a lightweight client can outsource a computation over the entire Bitcoin blockchain and learn the correct result. VERSUM is a *refereed delegation of computation*, or RDoC, system [7]. A VERSUM client outsources a computation to a pool of independent servers, and can determine the correct result as long as it can reach one honest server. To handle large inputs, outputs, and intermediate state, VERSUM stores all data in authenticated data structures.

Refereed delegation of computation systems assume a trust model based on the practical assumptions that not *all* servers will be compromised. The challenge in designing a RDoC system is performing the *refereeing* on the client. It is not enough to merely look for a majority of agreeing servers: a RDoC system must provide the correct result even if all servers disagree. Using its *conflict resolution* protocol (§6), VERSUM can ask a server to prove that a conflicting server made a mistake. The conflict resolution protocol is based on Quin's [7]. When two servers disagree on the outcome of a computation, Quin splits the computation in many parts and finds the first point at which two servers disagreed. Then, it determines which server made a mistake, and continues with the honest server.

Unlike Quin, VERSUM supports incremental updates of inputs. Incremental updates happen, for example, when a new log entry is added to the Bitcoin blockchain, VERSUM can update its computation to include the latest transactions from the log. Supporting incremental updates is challenging because the server must prove that it performed the *entire* new computation correctly; handling updates one by one does not scale with many updates. VERSUM efficiently performs the new computation in its entirety by *reusing* parts of the old computation. To support reuse, VERSUM represents computations as *purely functional programs*. These programs have no global state, which can make programming more difficult; however, because they have no global state, computations can be reused. Quin cannot reuse computations as its x86 computations have global state, and thus change completely when the input is changed even slightly.

During conflict resolution, VERSUM uses *computation histories* (§4) that describe the evaluation of its functional programs. Computation histories can be reused to support incremental updates, and can be efficiently extended so that clients can perform conflict resolution. Conflict resolution requires a data structure for holding computation histories that supports fast lookups and comparison. Supporting incremental updates requires a data structure that supports efficient concatenation of computation histories. The SEQHASH (§5) is the first data structure to efficiently support all

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660327 .

of these operations in an adversarial setting, formally stated and proven in Appendix A.

We have implemented VERSUM and three applications on top of the Bitcoin blockchain in 2,800 lines of Go (§7). As an example, one of these applications computes the set of unspent transactions in Bitcoin, a common computation in practice. The VERSUM prototype can run this computation over the unmodified Bitcoin blockchain; it uses an existing Go library to read and store Bitcoin data structures. A VERSUM server running this computation creates around 365 GB of authenticated data structures for a recent snapshot of the blockchain, which contains 14 GB of Bitcoin blocks and a total of 30 million transactions. Our prototype automatically pages the authenticated data structures to and from disk, so that applications can effectively perform large computations.

Experiments with this prototype (§9) show that a server can support many clients (e.g., a server can serve 4,000 queries per second for the unspent Bitcoin transaction application), and that servers can incorporate new Bitcoin blocks in seconds, which is sufficient to keep up with the growth of the Bitcoin blockchain (which grows by one block approximately every ten minutes).

Although we demonstrated VERSUM's applicability using a Bitcoin log, we expect VERSUM to be applicable in increasingly more situations as more data is available in the form of authenticated logs. VERSUM's design could be used to determine if a certificate has been revoked in the Certificate Transparency system [12], to perform name lookups in Namecoin [2], etc. One can even view all git repositories as authenticated data structures that VERSUM could compute over.

2. BACKGROUND

VERSUM depends on two key previous ideas: authenticated data structures and the refereed delegation of computation model. This section provides the necessary background about these two ideas so that the reader can understand VERSUM's design.

2.1 Authenticated data structures

Authenticated data structures [20] (ADSs) allow a client to outsource storage of large data sets to a server without trusting that server [14]. The client stores a small authenticator (for example, a hash) that summarizes all data stored on the server. With that authenticator, the client can then verify operations, such as lookups in a dictionary, performed by the server.

Using Miller et al.'s approach [14], VERSUM transparently transforms any functional data structure into an ADS. Behind the scenes, such a data structure is recursively hashed (like a Merkle hash-tree), and the authenticator is the *root hash*, the hash at the top of the tree. The programmer writes only the normal lookup function, and all hash computation and verification is handled automatically.

For example, consider a client outsourcing the storage of a binary search tree to a server. To perform a lookup in the tree, the client asks the server to perform a lookup, and to transmit all the nodes that the lookup function accessed to the client. The client then checks that the server sent the correct nodes by computing the hashes, and performs the lookup itself to determine the result. If either the hashes are incorrect, or the server has not sent some required nodes, the computation fails. Otherwise, the client is guaranteed a correct result, as it has performed the computation itself on the correct input. For terminology, in the remainder of the paper, we will refer to the values accessed by the server that let a client verify a computation a *proof*.

While ADSs might seem to outsource computation as well as storage, ADSs do not speed up computation on the client, as the client must still perform the computation itself after the server has done so. For this reason, VERSUM does not use ADSs to outsource the entire computation, and instead uses ADSs to store data for the larger computation, so that the client can verify that small steps of the computation happened correctly.

2.2 Refereed delegation of computation

Refereed delegation of computation is a setting for outsourcing computation in a verifiable manner. A RDoC system allows a client to learn the correct result of a computation if it can talk to a pool of servers, at least one of which is honest. Even if all other dishonest servers are cooperating to try and deceive the client, the client will still learn the correct result. The client does not have to specify which server it thinks is honest; it merely needs to talk to a pool servers of which it believes at least one to be honest.

In effect, the client assumes that all server operators are honest (there is no point in talking to dishonest servers), but that a server might be compromised or coerced into deceiving a client. By outsourcing computation to a pool of independent servers, a client is protected against individual servers getting compromised.

Quin is a RDoC system for verifying the execution of Turing machines, applied to x86 binaries [7]. When two servers disagree on the outcome of a computation, Quin figures out which server made a mistake by finding the first step of the Turing machine the two servers disagreed on. To find this step, the client performs a binary search, with the goal of finding a state of the Turing machine that both servers agreed, with a consecutive state the servers disagree on:

Let n be the number of steps of the shortest computation. First, the client asks both servers to commit to their computation's state after $n/2$ steps. Then, if they agree, the client asks for the state at $3n/4$ steps, otherwise at $n/4$ steps, and so forth. Eventually, both servers will have committed to a state after m steps on which they agree, and two distinct states after $m+1$ steps. At this point, the honest server will be able to show that it advanced its state correctly, and the client can from then on ignore the dishonest server. Note that this point must exist, as the two servers agree on the initial state which defines the computation, and disagree, by definition, on the outcome of the computation.

Quin's computational model is general enough to let Quin run standard x86 binaries, storing the state in a Merkle tree describing the process's memory. However, because Quin runs standard x86 binaries, it does not support incremental re-computation: when the input changes, the Turing machine must be re-run completely.

3. OVERVIEW

VERSUM is a system for outsourcing computations in the RDoC setting. VERSUM targets computations like processing large, growing logs: large computations with frequently updated inputs. An end-to-end diagram of VERSUM can be found in Figure 1.

VERSUM represents computations as purely functional programs. In these programs, individual function calls are self-contained and have no side effects, so they can be reused in other computations. This allows VERSUM to quickly update a computation when the inputs are updated. VERSUM can reuse computation histories because its computations, unlike Quin's Turing machines, are stateless.

To perform conflict resolution, VERSUM uses a conflict resolution protocol inspired by Quin, but modified to support VERSUM's functional model and incremental computation. Instead of working with Turing machine state, a VERSUM server stores computation histories that describe the evaluation of the functional program as a sequence of function call and return records. When the input to a VERSUM computation changes, the server can reuse (parts of) the previous computation history to quickly build a new computation

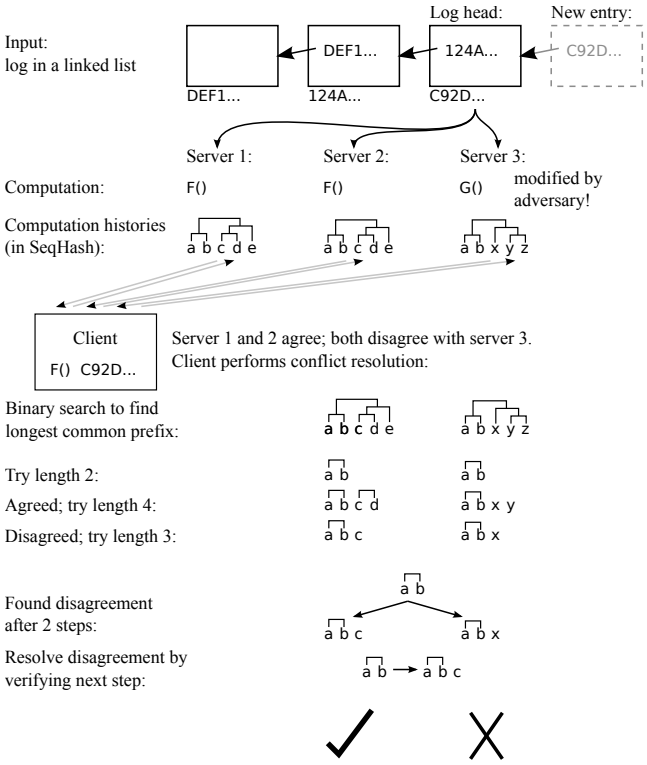

Figure 1: An overview of VERSUM. VERSUM computes over an input known to both client and server, the log at the top of the image. Three independent servers all compute `F()`. However, server 3 has been compromised, and instead computes `G()`. Each server stores a computation history describing the computation in a SEQHASH. The client, which knows both the input and `F()`, detects that server 3 has a different outcome than the other servers, and performs conflict resolution. After finding the longest common prefix (using binary search), the client discovers that server 3 made a mistake, and accepts the output from servers 1 and 2.

history. In Figure 1, all servers perform a computation `F` over an input log. Each server has a copy of the input, the computation, and the computation history. All should be identical on all servers. However, because server 3 has been compromised by an adversary, its computation and computation history are different.

VERSUM uses computation histories as the *state* of a computation. To advance this state, VERSUM uses a simple algorithm, called `DetermineNext`, that computes the next record from an unfinished computation history. `DetermineNext` reconstructs the evaluation state at the end of the unfinished computation history, and from that efficiently computes the next record. The algorithm is explained in detail in §4.

Using `DetermineNext`, VERSUM's conflict resolution protocol is similar to Quin's: when two servers disagree, a client performs a binary search to find the longest (unfinished) computation history that two servers agree on, and two computation histories one record longer that the servers disagree on. Then, using `DetermineNext`, the client can determine which server was honest. In Figure 1, VERSUM performs conflict resolution between servers 2 and 3, as their computation histories `abcde` and `abxyz` are different. The client finds the longest common prefix, `ab`, as well as two conflicting

```
F(x):
  return G(x) + H(x)
G(x):
  return H(x * 2) * 2
H(x):
  return x + 1

Step Action
1:   call F(5)
2:     call G(5)
3:       call H(10)
4:       return H(10) = 11, computed in 2 steps
-----------------------------------------------
5:     return G(5) = 22, computed in 4 steps
6:     call H(5)
7:     return H(5) = 6, computed in 2 steps
8:   return F(5) = 28, computed in 8 steps
```

Figure 2: An example program, together with a computation history of `F(5)`. The indentation and the step numbers (on the left) are for ease of reading only, and are not actually stored by VERSUM. The step counts (on the right) are stored by VERSUM. The prefix ending at step 4 is used as an example in §4.2.

prefixes, `abc` and `abx`. Using `DetermineNext`, the client can determine that `abc` is correct, and that server 3 must have performed an incorrect computation.

VERSUM needs to store computation histories in a data structure that supports a number of operations. First, a server must efficiently construct computation histories, both during a first run, and during future incremental computations. That means that two computation histories should be efficiently concatenable. Second, two computation histories must be efficiently comparable during conflict resolution, and must support `DetermineNext` invocations. Finally, the data structure must be efficient even if adversaries control its inputs, as VERSUM's external inputs might be contributed to by anyone.

The SEQHASH data structure supports all operations needed by VERSUM, and is the first data structure to do so. A SEQHASH is a deterministically shaped hash-tree for holding sequences, supporting efficient concatenation, comparison, and indexing. A SEQHASH's shape is determined by performing several merge rounds, during which adjacent (leaf-)nodes stochastically get merged.

VERSUM stores all function arguments, return values, and the computation history in authenticated data structures. When a server constructs a `DetermineNext` proof, it must include the inputs and outputs it computed over. Since internal state might be large, VERSUM uses ADSs to keep proofs small. `DetermineNext` also performs operations on a SEQHASH, which can also be used as an ADS because a SEQHASH is a functional data structure.

4. COMPUTATION HISTORIES

VERSUM's computation histories help clients decide which server performed a computation correctly. This section describes the structure of a computation history, how a computation history allows clients to verify computations, and how parts of a computation history can be reused among different computations.

4.1 Structure

VERSUM runs deterministic, side-effect-free functional programs. A *computation history* is a log of the evaluation of such a program: a computation history is a sequence of *steps* which are either function calls or returns. Each function call is annotated with the function name and arguments, and each function return contains the return

```
DetermineNext(prefix):
  index := length(prefix) - 1
  expected := []
  while prefix[index].type != call:
    // store already-known answer to nested
    // function call
    ret := prefix[index]
    call := prefix[index-ret.length+1]
    expected = call .. ret .. expected
    // move index to right before this
    // function call
    index -= ret.length
  // it must be that prefix[index].type == call
  innermost = prefix[index]
  run innermost until expected is finished
  return Concat(prefix, innermost's next step)
```

Figure 3: The algorithm to extend a history prefix by one step.

value as well as the number of steps inside the function. VERSUM runs functions with call-by-value evaluation. An example program containing three functions `F`, `G`, and `H`, and a computation history for `F(5)` can be found in Figure 2. The trace shows all calls, but does not show computations performed inside the functions: while the computation history shows that `F` called `G` and `H`, which returned 22 and 6 respectively, it does not show how `F` used those values.

This history has two important properties. First, a server can efficiently prove that it extended an unfinished computation history correctly, which is used by VERSUM's conflict resolution protocol. Second, (parts of) a computation history can be reused in a new computation history, used by VERSUM's computations over logs.

4.2 Extending computation histories

During conflict resolution VERSUM must extend computation histories. When two servers have different computation histories, a VERSUM client first finds the longest prefix common to the two histories. Then, the client asks both servers to prove that their version of the history correctly extended the prefix.

Such a proof can be constructed with the `DetermineNext` algorithm, which takes in a computation history, determines the next step, appends it to the history, and returns this new history. This proof is an ADS proof (as described in §2), and works by letting a client perform the computation locally. This proof should be small to keep conflict resolution fast, and so `DetermineNext` must not perform any unnecessary work.

The idea behind the algorithm is that the function that will perform the next step (the innermost incomplete, meaning not yet returned, function call) can efficiently be run on the client up to the point where it performs the next step, because any nested function calls made by the innermost incomplete function have their results already available in the computation history. For example, consider the prefix of the example history ending at step 4 in Figure 2. The innermost incomplete function call is `G(5)` at step 2. `DetermineNext` can efficiently compute the result of `G(5)`, as it can use the result of `H(10)` at step 4 in the history.

Pseudocode for the `DetermineNext` algorithm can be found in Figure 3. The `DetermineNext` algorithm determines the innermost incomplete function call and the results of finished nested calls by jumping from function return to call in the computation history using the function lengths stored at each return step. `DetermineNext` starts at the last step of the log. If this last step is a function call, `DetermineNext` has found the innermost incomplete function call and can continue to the next phase. Otherwise it has found the return record of a completed nested call. It records the result of the nested call and jumps to the location right before the corresponding call, and repeats until it finds a call record.

In the example from Figure 2, `DetermineNext` starts by looking at step 4, which is a return record. It stores the result of `H(10) = 11`. Then, using the number of steps stored in the return record, `DetermineNext` jumps to the record right before the start of `H(10)`, which is `call G(5)`. This is the innermost incomplete function call for this prefix.

Once `DetermineNext` has found the innermost incomplete call and the results of already-completed nested calls, `DetermineNext` can run the innermost incomplete function without recomputing the already-known nested functions. Once the innermost incomplete function executes past all of the already-known nested function calls, it will either return or make a new call not yet in the history. In both cases, `DetermineNext` obtains the next step of the innermost incomplete function without computing any other functions. `DetermineNext` proofs remain short, even for highly recursive calls over long linked lists.

4.3 Reusing computation histories

To support incremental updates, computation histories can be reused. If a new computation calls a function with the same arguments as in a previous computation, VERSUM can reuse that invocation's computation history from the previous computation. For example, if a server had previously computed `G(5)` before computing `F(5)`, then it could copy the 4 steps describing `G(5)` from the previous computation history, and concatenate it to `F(5)`'s computation history, without performing the computation of `G(5)` again.

For VERSUM, reusing computation histories is especially useful when computing over growing logs. For example, consider a log structured as a linked list. First, the server processes the entire log with a recursive function. Then, the log gains a new entry, pointing to the previous latest entry. Now the server can process this new log efficiently by reusing the computation history for processing the old log. To do so, the server starts by performing the new computation as normal: it creates a new computation history, calls the recursive processing function on the head of the new log, and adds a corresponding call record to the computation history. Then, the processing function recursively calls itself on the previous log head to process the log up to the new entry. Now the server can reuse its previous computation, as that computation already processed the previous log; instead of performing the computation again, VERSUM can look up the cached result of the function. However, the result by itself is not enough; the server must also construct a valid computation history. To do so, it concatenates the entire cached computation history of the previous invocation, including all internal call and return records. Afterwards, the processing function can continue processing the new entry, and finish the computation. In the end, the server has constructed a complete computation history for the entire new log, though it only had to actually perform computation to process the new log entry.

Computation history reuse relies on VERSUM's programming model of side-effect-free functional programs: because a function cannot access global state, its computation history for fixed arguments must always be the same, and can safely be copied into a new computation history.

4.4 Discussion

Not all function calls need to be part of the computation history, and it is up to the developer to determine which calls to include in the log. In practice, a short function like `H` should not be in the log, as it will not significantly decrease `DetermineNext` proof sizes.

The size of a `DetermineNext` proof is determined by the number of calls made by the innermost unfinished function: for each of those, the proof must include the result of the function. It is up to the developer to keep the number of calls made by each function reasonably small. Note that if a function becomes too long, it can always be split in two parts, with the first part passing local variables to the second part as arguments.

VERSUM efficiently supports incremental updates to the input data structure if it can reuse previous computation histories. Functions processing unchanged data should not have their arguments changed. That is, computations should be structured in a memoization-friendly way. For example, to compute a sum over a list, the running total should not be passed as an argument, but should instead be kept and later added by the calling function.

5. SEQHASH

VERSUM stores computation histories in SEQHASH, a novel hash-tree structure for storing sequences. SEQHASH supports fast positional indexing, fast concatenation, and is efficiently comparable thanks to its deterministic structure. This section describes and motivates SEQHASH.

5.1 Goals

To hold computation histories for VERSUM, SEQHASH needs several properties:

Efficient lookup and concatenation. To keep `DetermineNext` proofs small, SEQHASH must support fast lookups. When VERSUM reuses (parts of) a computation history, the corresponding SEQHASH must be efficiently concatenable to the new computation history.

Note that SEQHASH must support general concatenation (that is, a SEQHASH will be constructed in different orders): at the end of `DetermineNext`, SEQHASH must support concatenation with a single step. However, when reusing computations, SEQHASH must support concatenation with a large number of steps.

Efficient comparison. During conflict resolution, VERSUM must determine if two SEQHASH's are equal. Computation histories are too large to transmit in their entirety, so a SEQHASH is stored as an ADS. To efficiently test for equality, two SEQHASH's representing identical computation histories must have the same root hashes. This means that a SEQHASH's internal structure must be deterministic for given a computation history; it cannot depend on the order in which the SEQHASH was constructed.

Resistance to adversarial inputs. A deterministic structure comes with a risk: there might be a set of inputs that leads to an unbalanced internal structure, so that operations on the SEQHASH become slow (see §10.2). If an adversary could cause operations on a SEQHASH to take linear time by modifying the input, they could easily cause a VERSUM server to become unavailable. Constructing such an adversarial input must be hard.

5.2 Structure

A SEQHASH is a forest of balanced binary trees that is constructed over several merge rounds. At the start of construction, all of the elements making up the sequence that the SEQHASH holds are placed as leaves at the bottom of the SEQHASH. Then, as long as at least one node remains, SEQHASH performs a merge round. The input and output of each round is the current sequence of nodes representing the roots of SEQHASH's trees. Each round merges several adjacent pairs of input nodes, forming new trees as input for the next round. Each merged node contains hashes of the merged children, much like a Merkle tree [13]. SEQHASH keeps running until no more nodes remain. Figure 4 contains an example SEQHASH, where the leaf nodes *a*-*k* are the SEQHASH's elements.

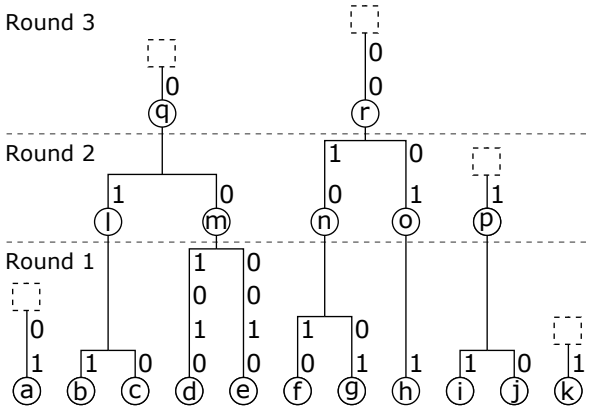

Figure 4: A SEQHASH constructed over the sequence of leaf nodes *a*-*k*. The digits on top of each node represent the output bits of the hash of each node. Dashed squares indicate roots of trees that might be merged when another SEQHASH is concatenated to this SEQHASH.

Since SEQHASH produces a forest containing a variable number of trees, we summarize the entire forest at the end by hashing together the roots of all trees, from left to right, to produce a single final hash value. In VERSUM, all these hashes are computed automatically by the ADS code.

5.3 Merge round

The idea behind SEQHASH's merge round is to use a cryptographic hash function to determine which nodes to merge.

The input to SEQHASH's round function is a sequence of nodes. The first round starts with all sequence elements as individual nodes as input; in the case of a computation history, these nodes store individual steps in the history.

Using the hash function, each input node is hashed to construct an infinite sequence of bits. The round then proceeds one bit at a time, considering the output of all nodes at the same time. SEQHASH merges each adjacent pair of unmerged nodes when the left node generated a 1 and the right node generated a 0 (with the exception of nodes on the side of the sequence). The round function continues until no two unmerged adjacent nodes exist. The tree has a deterministic shape because the bit sequence is deterministically generated by each node.

Consider the first round in the example SEQHASH from Figure 4, ignoring nodes *a* and *k*. While processing the first bit, nodes *b* and *c* are merged, as well as *i* and *j*. Nodes *d* and *e* are not merged because *d*'s first bit is not a 1, nor is *e*'s bit a 0. After several more bits, all nodes except *h* are merged. Because *h* is between two merged nodes, it will never merge during this round no matter how many bits are considered, and so the round ends.

The nodes on the side represent a challenge because we do not yet know their neighbor nodes; for example, node *a* might end up adjacent to another node that generates 1 for the second bit of round 1, but it might also end up to a node that is merged during the first bit of round 1. We say that in this example SEQHASH the fate of node *a* is *unknown*, and so we keep it as its own tree in the final SEQHASH.

During the merge round function, any nodes on the side whose fate cannot yet be decided are marked as unknown. This happens

to the leftmost node if it generates a 0, and to the rightmost node if it generates a 1. Multiple nodes can be marked as unknown on the same side during a single round. For example, nodes q and r are both marked as unknown in round 3, as q might merge with a node to its left based on the first bit, or it might not, and so r might merge with q, or it might not.

5.4 SeqHash characteristics

During each round, the number of unmerged nodes is limited by the fact that each round continues until no two adjacent nodes are both unmerged. Consider all nodes continuing to the next round (and thus ignoring unknown nodes). Each unmerged node must have at least two merged nodes to its left, so the fraction of unmerged nodes can be at most $1/3$ of the total nodes. Because of that, the total number of rounds is bounded by $O(\log_{3/2}(n))$.

The length of each round is limited, as we will prove formally in Appendix A. Intuitively, two adjacent nodes will merge after 4 bits in expectation, because the bits used during merging are indistinguishable from random, and 1 out of 4 possible bit combinations results in merging. For adversarially constructed input sequences, the number of bits consumed in a round is bounded by a constant κ (which is the security parameter, roughly the log of the amount of attacker's computational resources), because the difficulty of coming up with random bit sequences that do not merge for at least κ bits is exponential in κ. The number of unknown nodes is bounded by 2κ, as after κ bits any remaining nodes not yet marked as unknown will have either merged or have two merged neighbors.

One detail is the possibility of identical nodes with the same hash value that will never merge. To solve this problem, SEQHASH merges all identical adjacent nodes at the beginning of each round.

5.5 Achieving SeqHash's goals

Fast lookup. By storing the total number of elements under each node in the SEQHASH, an element can be quickly found by its index, by first identifying the tree holding the element (by enumerating all trees), in time $O(\kappa \log n)$, and then performing a simple lookup in the identified tree, in time $O(\log n)$.

Fast concatenation. Two SEQHASH's can be concatenated by resolving the fate of the unknown nodes that now have known neighbors. Concatenating two SEQHASH's over n and m elements takes time linear in the number of unknown nodes, or $O(\kappa(\log n + \log m))$. Although an unknown node might result in another node that must be processed, each such merge can be accounted to one of the disappearing nodes to get the given runtime bound.

Efficient comparison. SEQHASH's deterministic structure, independent of merge order, allows VERSUM to efficiently test the equality of two SEQHASH's by comparing their root hashes.

Adversary resilience. SEQHASH provides two critical security properties. We provide the intuition behind these properties here, and defer the complete definitions and proofs to Appendix A. First, SEQHASH is collision-resistant; that is, an adversary cannot construct two sequences whose SEQHASH values (i.e., the root hashes of the trees in the forest) are identical. This is defined more precisely in Definition A.1 and proved in Theorem A.2. Second, an adversary is unable to construct a sequence that makes SEQHASH inefficient; that is, a sequence for which SEQHASH produces a forest with more than $O(\kappa \log n)$ trees. This is defined more precisely in Definition A.3 and proved in Theorem A.6.

5.6 Pseudocode

Pseudocode for SEQHASH's round function can be found in Figure 5 and pseudocode for SEQHASH's concatenation can be found in Figure 6. The round function implementation keeps track of the index of the leftmost and rightmost nodes in `l` and `r`. If their respective sides could have neighboring nodes that might merge, indicated using `volatileL` and `volatileR`, then the round function will mark them as unknown. A side can be non-volatile if it borders to an already-merged set of nodes in an existing SEQHASH.

The concatenation function repeatedly uses the round function to determine which nodes to merge, taking care to reuse unknown nodes from the existing SEQHASH's if it exists, which means that the side is not volatile. A special case involves the unknown nodes of the final round; they are stored in the list of unknown nodes on both sides.

```
DoRound(A, volatileL, volatileR):
  n := len(A)
  nextUnknownL := 0
  nextUnknownR := n-1
  i := 0

  Initialize output arrays unknownL, center, and
  unknownR.

  if volatileL:
    while A[nextUnknownL] == A[0]:
      Add A[nextUnknownL++] to unknownL.
  if volatileR:
    while A[nextUnknownR] == A[n-1]:
      Add A[nextUnknownR--] to unknownR.

  Merge all groups of adjacent identical nodes and
  add them to center.

  while not all nodes are in an output:
    for each adjacent pair of nodes (a,b) in A
        that are not in any output:
      if bit(a,i) == 1 && bit(b,i) == 0:
        Add merge(a, b) to center.
    for each node not in any output
        with two merged neighbors:
      Add the node to center.
    if volatileL && bit(A[nextUnknownL],i) == 0:
      Add A[nextUnknownL++] to unknownL.
    if volatileR && bit(A[nextUnknownR],i) == 1:
      Add A[nextUnknownR--] to unknownR.
    i++
  return unknownL, center, unknownR
```

Figure 5: SEQHASH round pseudocode.

6. VERSUM

The previous sections provide the building blocks to completely specify VERSUM's conflict resolution. This section describes VERSUM's conflict resolution protocol and the server API required for conflict resolution, and states a correctness theorem for VERSUM.

6.1 VerSum's conflict resolution protocol

If two servers claim two different outcomes for a computation, the VERSUM client uses the conflict resolution protocol to determine which server has performed its computation incorrectly, specified in the `DetermineWrong` algorithm in Figure 7.

At its core, VERSUM's conflict resolution protocol is similar to Quin's conflict resolution algorithm [7]. When two servers disagree, their computation histories must diverge at some point. The client finds this point by performing a binary search over prefix lengths, asking the servers to return the root hash of the SEQHASH representing the prefix of that length and seeing if they are equal. At the end of the binary search, the client will have found the longest

```
Concat(l, r):
  center := []
  m := SeqHash{
    h: 0,
    unknownL: empty array of node arrays,
    unknownR: empty array of node arrays,
    top: empty node array,
  }

  while true:
    if m.h < l.h:
      center = cat(l.unknownR[m.h], center)
    else if m.h == l.h:
      center = cat(l.top, center)
    if m.h < r.h:
      center = cat(center, r.unknownL[m.h])
    else if m.h == r.h:
      center = cat(center, r.top)
    if m.h >= l.h && m.h >= r.h && center == []:
      break

    volatileL := (m.h >= l.h)
    volatileR := (m.h >= r.h)
    unknownL, center, unknownR =
      DoRound(center, volatileL, volatileR)

    if volatileL:
      m.unknownL[m.h] = unknownL
    else: // otherwise, unknownL is empty
      m.unknownL[m.h] = l.unknownL[m.h]
    if volatileR:
      m.unknownR[m.h] = unknownR
    else:
      m.unknownR[m.h] = r.unknownR[m.h]
    m.h += 1

  m.top = cat(m.unknownL[m.h-1], m.unknownR[m.h-1])
  m.h -= 1
  return m
```

Figure 6: SEQHASH concatenation pseudocode.

prefix length for which the two servers agree and have obtained two contradicting claims for the next prefix. The client can then ask either server to prove the correct next prefix using `DetermineNext` and determine which server committed to an incorrect prefix.

If two servers claim to have computations of different lengths, the client performs a binary search up to the length of the smaller computation plus one, as the servers must disagree before that point. If the two servers agree up to the point of the shorter prefix, the client invokes `DetermineNext` to determine if the shorter was wrongly truncated or if the longer prefix was wrongly extended.

To support more than 2 servers, VERSUM can use any of the suggested schemes by Canetti et al. [7]. The number of interactions for m servers of a computation of length n becomes $O(m \log n)$, running in time $O(\log m \log n)$ by arranging the servers in a tournament tree.

6.2 Reliance on SeqHash uniqueness

A malicious server could store a correct computation history in a badly constructed SEQHASH (by performing merge rounds incorrectly). It is crucial that VERSUM treats such a badly constructed SEQHASH as an incorrect computation: to make progress, VERSUM must be able to find at least one server that made a mistake at the end of the binary search. If an invalidly shaped SEQHASH holding the correct computation would be accepted by the client, then both servers would be correct, and the binary search would have been pointless.

```
DetermineWrong(serverA, serverB):
  n := min(serverA.getLength(),
           serverB.getLength()) + 1
  agreed := prefix of length 1 starting
            with the desired computation
  claimA := serverA.getPrefix(n)
  claimB := serverB.getPrefix(n)

  // perform a binary search
  while len(agreed) + 1 < n:
    mid := (len(agreed) + n)) / 2
    a := serverA.getPrefix(mid)
    b := serverB.getPrefix(mid)
    if a == b:
      agreed = a
    else:
      n = mid
      claimA = a
      claimB = b

  next :=
    serverA.proofDetermineNext(agreed)
  if next == claimA:
    return B
  if next == claimB:
    return A
  return both
```

Figure 7: The algorithm to determine which of two servers has an incorrect computation history. If at any point during the conversation one server stops responding, or returns an invalid response (e.g. to `DetermineNext`), that server is considered wrong.

6.3 Handling uncooperative servers

A malicious server could try to break the conflict resolution protocol in several ways: a server could try to make the binary search take too long, a server could stop responding, or a server could provide invalid answers to queries. The client handles all these problems by checking that all answers are correctly formatted with reasonable size limits. If a server ever gives a badly formatted answer, or an invalid proof, or fails to reply, the client assumes that server is wrong.

6.4 Server API

To support VERSUM's conflict resolution, a VERSUM server supports several functions:

- `getLength()`: Return the length of the computation history.
- `getPrefix(length)`: Return a prefix of the computation history of the requested length, represented as a SEQHASH.
- `proofDetermineNext(prefix)`: Given a prefix of a history as SEQHASH, run the `DetermineNext` algorithm on the prefix and return the next prefix. Since `prefix` is an ADS, the server returns a proof of a valid `DetermineNext` computation.
- `proofGetResult(history)`: Given a complete history as SEQHASH, return the output of the complete computation (as returned by the last step of the history). Since `history` is an ADS, the server returns a proof that this is indeed the last element in `history`.

All operations besides `getPrefix` have a straightforward implementation using the underlying SEQHASH. The `proofGetResult` and `proofDetermineNext` calls depend on SEQHASH's ability to perform lookups in $O(\kappa \log n)$.

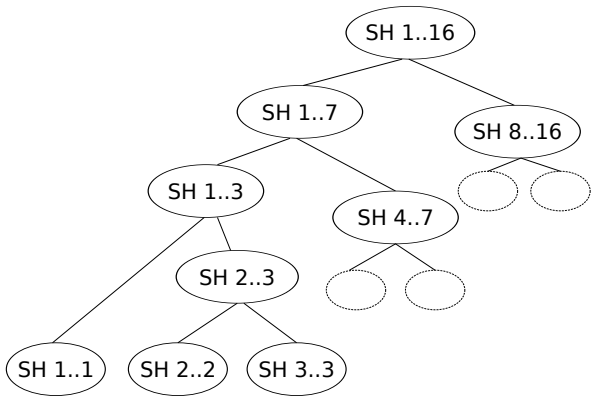

Figure 8: The partial SEQHASH's stored in a balanced binary tree on a VERSUM server to quickly answer `getPrefix` queries. Each node stores the concatenation of the SEQHASH's of its children so that a server only needs to concatenate $O(\log n)$ SEQHASH's to respond to any query.

For `getPrefix` the server must construct a SEQHASH for any requested length. To do this efficiently, the server stores partially constructed SEQHASH's for many different subsequences of the computation history in a balanced binary tree over the computation history, as shown in Figure 8. The server constructs this binary tree when it performs the computation. Each node in the tree stores the concatenation of the SEQHASH's of its two children, so that a SEQHASH covering any prefix can be constructed in $O(\log n)$ SEQHASH concatenations. For performance, `getPrefix` depends on SEQHASH's ability to concatenate two arbitrary SEQHASH's in time $O(\kappa(\log n + \log m))$. For a computation history of length n, a `getPrefix` call takes $O(\kappa \log^2 n)$ time for $O(\log n)$ concatenations.

6.5 Security

To formally state VERSUM's security properties, we have two theorems guaranteeing VERSUM's correctness. First, an honest server will always be able to quickly respond to queries, even in the case of a computation that contains adversarially constructed inputs influencing the underlying SEQHASH. This follows from SEQHASH being adversary-proof. Second, if a VERSUM client can reach at least one honest server, it will learn the correct result of a computation with high probability. This follows from the correctness of Miller et al.'s approach [14].

First, to ensure that a VERSUM server can quickly respond to all queries, we must make sure the computation is suitable for VERSUM:

DEFINITION 6.1 (SUITABLE COMPUTATION). *A computation for* VERSUM *is suitable if it is a purely functional, side-effect-free computation with short functions. That is, there exists a small constant c such that, already knowing the result of nested functions, each function can be computed in c steps.*

A suitable computation has small `DetermineNext` proofs, as each function itself will call no more than c functions and access no more than c ADS nodes,

Then, because SEQHASH is efficient, an efficient server exists, as described in §6.4:

THEOREM 6.2. *After performing an initial suitable computation of n steps, an honest* VERSUM *server can respond to any query in $O(\kappa \log n(\log n + c))$.*

PROOF. By SEQHASH's security proof, all SEQHASH operations are efficient, even under adversarial input. `getLength`, `getPrefix`, and `proofGetResult` all have efficient implementations using SEQHASH, running in time $O(\kappa \log^2 n)$ or better.

As the computation is suitable, all function calls in the computation history have a bound number of calls, and so `determineNext` runs in time $O(c\kappa \log n)$ to perform c SEQHASH lookups, and then takes c steps to run the function itself.

Combining both runtimes gives the desired bound. □

Building on an efficient and honest server, we can prove VERSUM correct:

THEOREM 6.3. *A single instance of* `DetermineWrong` *will declare an honest server wrong with negligible probability.*

PROOF. A proof by contradiction. Suppose a VERSUM client has incorrectly kicked out an honest server.

The honest server cannot have been labeled as an uncooperative server: it is honest, reachable, and has an efficient implementation for all API calls.

Instead, the other server must have constructed a `DetermineNext` proof of a next computation history different from the honest server's. However, the honest server's claim is, by definition, the result of an invocation of `DetermineNext`. This means that somehow, the other server constructed an invalid `DetermineNext` proof. This is a violation of the underlying ADS security property, and can happen only with negligible probability. □

Our final theorem states that VERSUM is both correct and efficient for a pool of servers:

THEOREM 6.4. *A* VERSUM *client that can reach at least one honest server out of m servers, will, with high probability, learn the correct result of a computation in $O(m \log n)$ interactions with $O(mc\kappa \log(n))$ local work.*

PROOF. By repeatedly refereeing disagreeing servers, a client will eventually learn the computation's result with high probability.

Each such interaction requires two `getLength` calls, $O(\log n)$ `getPrefix` calls, and a single `proofDetermineNext` call, for a total of $O(m \log n)$ interactions.

Verifying the m `determineNext` proofs will take time $O(mc\kappa \log n)$, to verify the $O(mc)$ lookups in the computation history, and run m functions of no more than c steps each. □

This correctness statement provides the same security, but not the same runtime, as Canetti et al.'s RDoC [7], because RDoC does not assume precomputation. A single invocation of the conflict resolution protocol in RDoC can take as long as the entire computation, while VERSUM's conflict resolution protocol takes time (poly-)logarithmic in the length of the computation.

7. BITCOIN

Our main use-case for VERSUM is processing the Bitcoin blockchain. This section describes how and why we used VERSUM to process the Bitcoin blockchain.

7.1 Why use VerSum for Bitcoin?

VERSUM lets lightweight clients securely outsource complex computations. The main use case for VERSUM in this paper is to support secure, lightweight Bitcoin clients. Users may need to perform many kinds of computations over the Bitcoin blockchain, such as:

- Determining the current balance held by a single Bitcoin account.
- Determining all (recent) incoming transactions to an account.
- Determining all (recent) outgoing transactions from an account.
- Determining all (recent) outgoing transactions from an account over a threshold amount.
- Various statistics, such as daily transaction volume, transaction size, etc.

One solution is to perform these computations locally on the user's device. Unfortunately, the Bitcoin blockchain is too large to reasonably store on mobile phones, and most PC users are unwilling to spend several hours downloading and processing the blockchain.

Instead of downloading the entire log, Bitcoin users currently outsource their computations to various parties on the Internet. These service providers tell Bitcoin users their balance, list various transactions, provide statistics, and so forth. Unfortunately, this approach has a significant security problem: users blindly trust the results sent to them from the server, and a malicious server can manipulate this financial information.

VERSUM enables the best of both worlds: it allows clients to verify the results of arbitrary computations over the blockchain, without requiring them to either blindly trust some server, or to download the blockchain and perform the computation themselves.

Various lightweight Bitcoin clients currently exist. They trust a single server to provide them with correct information on the state of the Blockchain. The lightweight Electrum client [1] now requires balance information to come from an authenticated data structure (ADS). However, the identity of this ADS itself is not verified, and so an attacker can simply tell the client to trust an incorrect ADS.

An alternative to VERSUM might be to modify the Bitcoin protocol. For example, the Bitcoin community has proposed modifying the Bitcoin blockchain to store the set of unspent transactions outputs as an ADS, similar to the one used by Electrum. Under this proposal, lightweight clients need not process the entire blockchain, and can instead ask an untrusted third-party to perform a lookup in the ADS. If adopted, this will help a lightweight client determine if a transaction has been spent or not. However, this ADS is specific to *one* particular computation. Every other computation would require changing the Bitcoin protocol again, which not only increases the size of the blockchain but also requires buy-in to *each* such computation from the entire Bitcoin community.

VERSUM needs no modifications of the underlying log to verify *any* outsourced computation. If lightweight clients were to switch to VERSUM, they would be secure immediately without Bitcoin protocol changes, and with more flexibility.

7.2 Calculating unspent transactions

To illustrate the power of VERSUM's verifiable computations, consider the Bitcoin blockchain. Bitcoin tracks money as *unspent transaction outputs*. Each transaction spends several previously unspent transaction outputs, and makes several new transaction outputs available. Transactions are grouped into blocks, and the entire Bitcoin blockchain consists of a singly linked list of these blocks. The Bitcoin protocol ensures that all participants agree on the same blockchain [15].

A developer can use VERSUM to summarize the Bitcoin log into a set of unspent transaction outputs, as shown in Figure 9. The developer summarizes the Bitcoin log using the recursive `Summarize` function. Since the blockchain is a linked list, `Summarize` first recursively calls itself to summarize the entire blockchain before the current block. It then processes all transactions in the current block using `ProcessTxn`, marking their inputs as spent in `SpendOutput` and making their outputs available.

```
Summarize(block):
  if block is nil:
    return empty
  balances := Summarize(block.previous)
  for txn in block.transactions:
    balances = ProcessTxn(balances, txn)
  return balances

ProcessTxn(balances, txn):
  for output in txn.inputs:
    balances = SpendOutput(balances, output)
  balances = balances.makeAvailable(txn)
  return balances

SpendOutput(balances, previousOutput):
  balances = balances.spend(previousOutput)
  return balances
```

Figure 9: A program to summarize the Bitcoin blockchain into a set of balances for each account.

This example demonstrates two important properties of VERSUM computations. First, all computations must be *purely functional* so that VERSUM can track the input and output of each function. In this example, the state of the blockchain is stored in the immutable dictionary `balances`, which in our prototype implementation is an authenticated binary Patricia tree. The input, the blockchain itself, is also an ADS; clients can learn the current head of the blockchain by participating as a lightweight node in the Bitcoin network. Because the computation is purely functional, previous calls to `Summarize` can be reused in new computations when a new block is added to the Blockchain.

Second, all functions should make relatively few calls, to keep `DetermineNext` proofs small, while the total runtime of a function (including recursive calls) can be very long. In our example, a `Summarize` can take a long time to compute on the server as it recursively calls itself, but a client will never have to perform that nested call as the result will already be known.

7.3 Intermediate output

Once the client has determined the set of unspent transaction outputs in an ADS, it can then ask any untrusted server to prove, in that given set, whether a transaction is spent or not. For our evaluation of the unspent transaction output computation, we implemented a function `proofTxnSpent` which checks if a transaction is spent.

From the point of view of a single client, having to first obtain the root hash of the output ADS and to then query it to obtain the ultimate result seems less efficient than asking the server to compute the desired result directly. However, this intermediate output ADS allows VERSUM to share the same computation among many clients, and allows for compact proofs of arbitrary queries on this intermediate ADS.

8. IMPLEMENTATION

We built a prototype of VERSUM in Go, as well as several applications that compute over the Bitcoin blockchain, in a style similar to the code from Figure 9. The line count for each VERSUM component is shown in Figure 10. The implementation reuses the existing "btcwire" (github.com/conformal/btcwire) Go library to parse and store Bitcoin's internal datastructures, with a small wrapper to integrate with VERSUM's authenticated data structure support.

To handle large authenticated data structures, our prototype transparently pages ADS nodes to and from disk.

Component	Line count
authenticated data structures core	713
SEQHASH	215
computation proofs, `DetermineNext`	524
authenticated map	441
Bitcoin wrapper	580

Figure 10: Major VERSUM components.

Computation	Line count
Unspent Bitcoin transaction outputs	123
Incoming Bitcoin transactions	90
Name registration and transfer	119

Figure 11: Different VERSUM computations.

9. EVALUATION

To evaluate VERSUM we wanted to measure its practicality by answering the following questions:

1. Can VERSUM support a variety of computations? (§9.1)
2. Can VERSUM support many clients? (§9.3)
3. How much bandwidth does a VERSUM client need? (§9.4)
4. Can VERSUM quickly update its computation when a log grows? (§9.5)
5. Can VERSUM be used to summarize a large log? (§9.6)

9.1 Computations

To evaluate the utility and versatility of VERSUM we implemented several computations in VERSUM. Figure 11 lists these computations along with how many lines of code each implementation consists of. Using VERSUM, implementing new verifiable computations is easy, and can be done in about a hundred lines of code.

Besides the unspent Bitcoin transaction outputs example from §7, we implemented two extra computations, as follows.

The first, a list of incoming Bitcoin transactions for each Bitcoin account, stores all incoming transactions for each account in a linked list held in a map of accounts.

The second computation, a name registration and transfer scheme, uses Bitcoin in an unconventional way. The Bitcoin blockchain allows transactions to include 40 bytes of arbitrary data, which the name registration program uses to store two types of commands: registering a name to a public key, or transferring a name from one public key to another key. The 40 bytes hold a 32-byte hash of a name, along with an 8 byte-tag to indicate name registration transactions. The computation tracks registrations, and ensures that no name gets registered twice and that transfers include a signature from the current owner. This computation implements a subset of Namecoin on top of the Bitcoin blockchain, showing that VERSUM can implement interesting computations and build on existing public logs to support new features.

9.2 Experimental setup

To evaluate the performance of VERSUM, we tested our implementation of the unspent Bitcoin transaction outputs calculation, and ran experiments using the following setup:

Blockchain statistics. We used a snapshot from January 2014 of the Bitcoin blockchain to perform our experiments. This snapshot, used to bootstrap new Bitcoin nodes, contains 14 GB of Bitcoin blocks, storing 279,000 blocks holding approximately 30 million transactions.

Request	Warm cache	Cold cache
`proofTxnSpent`	4000 op/s	76 op/s
`getPrefix`	87 op/s	9 op/s
`proofDetermineNext`	55 op/s	6 op/s

Figure 12: Throughput in operations per second for client requests with warm and a cold disk cache.

Server. The VERSUM server ran as single-threaded process on an Intel E7-8870 2.4 GHz processor, with a single 1TB HDD as permanent storage and 256 GB of RAM. Although the machine has a large amount of memory, the computation was configured to use no more than 4 GB of RAM, to decrease pressure on Go's garbage collector. The remaining RAM was used as a buffer cache for the slow HDD.

9.3 Server performance

To understand whether VERSUM can support many clients, we measured the throughput that a VERSUM server can achieve. In particular, we measured the throughput of three different types of requests that the VERSUM server supports: queries on the output ADS of the Bitcoin computation (`proofTxnSpent`), prefixes requested during the conflict resolution protocol (`getPrefix`), and `DetermineNext` proofs requested at the end of the conflict resolution protocol (`proofDetermineNext`). Because a large part of the server time is spent paging in ADS nodes from disk, we measured the performance of the queries with both warm and cold OS disk caches. The results are shown in Figure 12.

A single core supports thousands of ADS queries per second on the output of the Bitcoin computation, with a warm disk cache. This suggests that VERSUM should be able to support a large number of clients querying the outputs of their computations.

`getPrefix` and `proofDetermineNext` are significantly slower because both of them must construct the SEQHASH for the requested prefix. This is also reflected in the cold cache performance numbers, as all the partial SEQHASH's must be loaded from disk.

Both of these functions are used in the conflict resolution protocol, which consists of two phases: first, a binary search over prefixes using `getPrefix`, and then verifying a prefix with `proofDetermineNext`. The binary search takes time logarithmic in `getLength`. The client enforces an upper bound on `getLength` so that the conflict resolution protocol never takes more than 60 binary search steps. This means that the end-to-end runtime of the conflict resolution protocol is less than a second for two servers with a warm cache.

We expect that conflict resolution protocol invocations should be relatively rare, as they should occur only after a server is determined to have given a wrong result. Once a client has a proof that a server misbehaved, the client can publish the proof to other servers and clients, so that the server is not used again. As a result, we expect that VERSUM's performance is sufficient to support a large number of clients, even if some of them do invoke conflict resolution.

9.4 Bandwidth usage

To support lightweight clients, values returned to clients cannot be too big; Figure 13 lists the sizes of return values for several operations. We measured the size of query proofs for (`proofTxnSpent`) by picking 1000 random unspent transactions and invoking `proofTxnSpent` on them. No proof was bigger than 4267 bytes. Such small proof sizes can be explained by noting that each query in

Operation	Size of return value
`proofTxnSpent`	< 4267 bytes
`getPrefix`	32 bytes
`proofDetermineNext`	10 KB – 600 KB

Figure 13: Sizes of return values

the authenticated map, holding the output of the computation over the blockchain, must access only a logarithmic number of nodes.

The size of the value returned by `getPrefix` is 32 bytes in size; a single hash representing the root of the partial SEQHASH holding the prefix.

The value of `proofDetermineNext` varied from less than 10 KB to a maximum of 600 KB. Because they varied more wildly, we sampled 10000 random prefix lengths, as well as 20 random series of 5000 consecutive prefixes. This roughly corresponds with the expected maximum proof size. A `Summarize` proof on a block with the maximum number of transactions, 6000, will contain the transaction hash, as well as a call and return record for each block, totaling around 70 bytes per block. With additional overhead from lookups in the SEQHASH to find these entries, 600 KB is around the theoretical maximum proof size.

9.5 Incremental computation updates

For VERSUM to be practical a server must be able to quickly incorporate changes to the underlying log. We measured the average time to include each of the last 2000 blocks in the Bitcoin blockchain (around 20 days worth of blocks). Adding a block took approximately 1.12 seconds per block, including paging data in and out of memory. Over the last 2000 blocks, the average block size was 0.17 MB, and the theoretical maximum block size is 1 MB, so even the largest possible blocks can be included in seconds. The Bitcoin blockchain grows one block approximately every 10 minutes, which VERSUM can incorporate in a fraction of that time.

9.6 Initializing VerSum

To bootstrap our VERSUM server we summarized all 14 GB of the Bitcoin blockchain. This entire computation took 25 hours, and the final computation history contains 195 million steps. The final computation history serialized to disk, including all arguments and return values, measures approximately 365 GB. The growth from 14 GB to 365 GB happens because VERSUM must store all intermediate results.

Processing the complete Bitcoin log is expensive even without VERSUM. For example, developers of a Bitcoin client in Go report that the time for processing 9.1 GB (250,000 blocks) of the Bitcoin log took 4.5 hours.[1] VERSUM processes a bigger log (14 GB) and does it in a verifiable way, which takes 25 hours in total. The VERSUM prototype is not optimized for the initial computation, but these numbers demonstrate VERSUM is practical.

10. RELATED WORK

10.1 Outsourcing computation

Various schemes have been proposed to outsource computation not in the RDoC model. AVM [11] provides auditable virtual machines to clients. However, these virtual machines do not guarantee correctness; instead, their mistakes can be proven to others. Pioneer [19] provides timing proofs of correctness, but these proofs cannot be reused for multiple clients.

[1] https://blog.conformal.com/deslugging-in-go-with-pprof-btcd/

A number of systems have required servers to produce an efficiently checkable cryptographic proof that the computation was performed correctly [3, 18]. Pantry [6] has shown that this approach can be coupled with authenticated data structures to verify computations with state. While not relying on any trusted servers, these systems suffer from high server-side computation overheads, making it impractical to run computations over a large input such as the Bitcoin blockchain.

In cases where the system needs to perform only a limited class of computations, specialized schemes have been designed, for example to support range queries over streaming data [17]. However, such systems cannot support arbitrary computations; in contrast, VERSUM can outsource and verify the results of any computation expressed as a functional program.

Finally, some systems rely on a piece of trusted server hardware, such as a TPM [21], to generate an attestation that the computation was performed correctly. Although efficient, this plan requires trusting the trusted hardware manufacturer. If any trusted hardware device or the root cryptographic key is compromised, it can produce incorrect attestations, and can trick the client into accepting an incorrect computation result.

10.2 Other data structures

SEQHASH used as an ADS is the first data structure for holding sequences supporting both fast comparison and fast concatenation. Although there are various other candidate schemes, they do not provide the same functionality as SEQHASH:

Merkle trees. Standard Merkle trees [13] have a rigid shape, and thus support fast comparison. However, because of that rigid shape, Merkle trees are not efficiently concatenable: for example, a Merkle tree of the sequence $\{1,2,\ldots,2n\}$ always combines $2i+1$ on the left with $2i+2$ on the right to form an intermediate node. When we prepend the value 0 to this sequence, all internal nodes change, as now $2i$ on the left is always paired with $2i+1$ on the right.

Balanced binary trees. Balanced binary trees, as implemented as an ADS by Miller et al. [14], are efficiently concatenable. However, a single sequence can be stored in many distinct balanced binary trees (by performing simple tree rotations). This means that two different trees are not efficiently comparable: deciding if two trees contain the same elements might require inspecting the entire tree, since the hash at the top of the tree depends on the internal structure of the tree. Thus, balanced binary trees are useless for testing sequence equality.

The shape of a balanced binary tree is determined by the order in which the sequence elements were inserted into or concatenated onto the tree. In the context of VERSUM, if balanced binary trees were used instead of SEQHASH, the tree representing a prefix might have a different shape depending on if it was formed during conflict resolution (where prefixes are extended one entry at a time) or during computation (where large sequences of entries get reused and concatenated all at ones).

Uniquely represented data structures. Since SEQHASH provides efficient equality comparison, it can be thought of as a *uniquely represented* data structure, or a *strongly history-independent* data structure [16]. Such data structures have been used to perform $O(1)$ comparisons, such as between dictionaries [8], hash tables [5], or B-Treaps [10]. Similar data structures have also been used to enforce append-only properties on logs [9, 22], including the Bitcoin blockchain itself [15]. Unlike SEQHASH, none of these data structures support $O(\log n)$ concatenation of sequences in the face of adversaries choosing the input data.

For example, in the case of a treap, an adversary could construct a sequence of steps with monotonically increasing priority, leading to an imbalanced tree.

11. CONCLUSION

This paper introduced VERSUM, a system that allows lightweight clients to outsource computations over large, frequently changing public logs to a collection of servers. As long as one of the servers is not compromised, VERSUM will return the correct result to the client. VERSUM achieves its efficiency using three key ideas: express the computations as functional programs, record the evaluation trace of programs in a computation history that helps clients determine which server is honest, and summarize computation histories using the SEQHASH authenticated data structure. Experiments with the Bitcoin log demonstrate that VERSUM is practical. We believe that VERSUM can also enable lightweight clients to perform verifiable name lookup in Namecoin and to validate certificates for Certificate Transparency. As more publicly available logs become available, we expect the number of use cases for VERSUM to grow.

The VERSUM prototype is publicly available at `https://github.com/jellevandenhooff/versum`.

Acknowledgments

We thank Sergio Benitez, Raluca Ada Popa, the anonymous reviewers, and our shepherd, Charalampos Papamanthou, for their help and feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF awards CNS-1053143 and CNS-1413920.

References

[1] Electrum Bitcoin wallet, 2014. URL `https://electrum.org`.

[2] Namecoin, 2014. URL `http://namecoin.info`.

[3] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2013.

[4] G. Bertoni, J. Maemen, M. Peeters, and G. van Assche. On the indifferentiability of the sponge construction. In *Proceedings of the 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 181–197, Istanbul, Turkey, Apr. 2008.

[5] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, Providence, RI, Oct. 2007.

[6] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[7] R. Canetti, B. Riva, and G. N. Rothblum. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 445–454, Chicago, IL, Oct. 2011.

[8] S. A. Crosby. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Transactions on Information and System Security*, 14(2), Sept. 2011.

[9] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Usenix Security Symposium*, Montreal, Canada, Aug. 2009.

[10] D. Golovin. B-treaps: A uniquely represented alternative to B-Trees. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, Rhodes, Greece, July 2009.

[11] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[12] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. RFC 6962, Internet Engineering Task Force (IETF), June 2013.

[13] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, pages 369–378, Santa Barbara, CA, Aug. 1987.

[14] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages*, pages 411–423, San Diego, CA, Jan. 2014.

[15] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008. URL `https://bitcoin.org/bitcoin.pdf`.

[16] M. Naor and V. Teague. Anti-persistence: History independent data structures. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, Heraklion, Greece, July 2001.

[17] S. Papadopoulos, Y. Yang, and D. Papadias. CADS: Continuous authentication on data streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, Sept. 2007.

[18] B. Parno and C. Gentry. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013.

[19] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[20] R. Tamassia. Authenticated data structures. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA)*, pages 2–5, Budapest, Hungary, Sept. 2003.

[21] Trusted Computing Group. Trusted computing group – Developers – Trusted platform module, 2014. URL `https://www.trustedcomputinggroup.org/developers/trusted_platform_module`.

[22] A. A. Yavuz, P. Ning, and M. K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security*, Bonaire, Feb. 2012.

APPENDIX

A. PROOF OF SECURITY FOR SEQHASH

To precisely state and prove SEQHASH's security properties, we will use standard cryptographic notation: κ will be a security parameter, $\mathrm{negl}(\cdot)$ is a negligible function, $a \leftarrow B$ denotes sampling a randomly from a uniform distribution over B, and $a \leftarrow \mathrm{Alg}()$ denotes running a probabilistic algorithm Alg to produce result a. We use $\mathrm{Alg}(f(\cdot))$ to represent algorithm Alg running with query access to function $f(\cdot)$.

Formalizing SEQHASH's security properties requires addressing one technicality: because SEQHASH uses a public hash function, it is difficult to bound the amount of pre-computation done by an adversary in constructing some adversarial algorithm Adv. In practice, this is not a problem, because we assume our hash function is indifferentiable from a random oracle [4] and assume the adversary has been running from the time SEQHASH was designed. However, to capture the notion of efficiency in our definitions, we temporarily assume that SEQHASH uses a keyed PRF P instead of a pseudo-random oracle, and we give the adversary's algorithm query access to this keyed PRF.

First, we define and prove collision-resistance.

DEFINITION A.1 (COLLISION-RESISTANCE). *F is collision-resistant if, for every probabilistic polynomial-time adversary Adv, for every sufficiently large security parameter κ,*

$$\begin{aligned}\Pr[&k \leftarrow \{0,1\}^\kappa;\\ &(a,b) \leftarrow \mathsf{Adv}(P_k(\cdot));\\ &F(k,a) = F(k,b) \text{ and } a \neq b] \leq \mathrm{negl}(\kappa)\end{aligned}$$

THEOREM A.2. SEQHASH *is collision-resistant, as defined in Definition A.1.*

PROOF. Suppose that there exists an adversary Adv that can construct a pair of sequences a and b with identical SEQHASH values with non-negligible probability. We will use Adv to construct a reduction to break the security of the Merkle tree used for each tree in SEQHASH's forest.

This reduction works as follows. Invoke Adv to obtain a and b. Pick the first tree hash h in $\mathrm{SEQHASH}(a) = \mathrm{SEQHASH}(b)$ where the nodes corresponding to h in a and b are different. Such a tree hash h must exist because $a \neq b$. Output the corresponding nodes from a and b. The resulting Merkle tree nodes are different but lead to the same Merkle root hash.

This reduction works with non-negligible probability, but by assumption Merkle trees cannot be broken with non-negligible probability. Thus, our supposition was wrong, and such an adversary Adv does not exist. □

Second, we define and prove efficiency.

DEFINITION A.3. *A function F that maps a key and a sequence to a forest of trees is efficient if, for every probabilistic polynomial-time adversary Adv, for every sufficiently large security parameter κ,*

$$\begin{aligned}\Pr[&k \leftarrow \{0,1\}^\kappa;\\ &\mathsf{seq} \leftarrow \mathsf{Adv}(P_k(\cdot));\\ &\mathsf{NumTrees}(F(k,\mathsf{seq})) \geq 2\kappa \log n] \leq \mathrm{negl}(\kappa)\end{aligned}$$

where $\mathsf{NumTrees}(\cdot)$ counts the number of trees in a forest.

Before we can prove that SEQHASH is efficient, we first prove a lemma about the maximum number of bits that has to be processed in a SEQHASH round until n different nodes merge. Let P be the keyed PRF used by SEQHASH, which produces an infinite stream of bits. Let $m(k,a,b)$ be the number of bits until nodes a and b merge in a given round; that is, the number of bits until $P_k(a)$ has a 1 bit and $P_k(b)$ has a 0 bit.

LEMMA A.4 (MERGE PROBABILITY). *For every n, for every sufficiently large security parameter κ, for every p which is polynomial in n,*

$$\begin{aligned}\Pr[&k \leftarrow \{0,1\}^\kappa;\\ &\max(\{m(k,a,b) \mid a,b \in \{1,\ldots,p\} \text{ and}\\ &\qquad a \neq b\}) \geq \kappa] \leq \mathrm{negl}(\kappa).\end{aligned}$$

PROOF. For every distinct a and b, $\Pr[m(k,a,b) \geq \kappa] = (\frac{3}{4})^\kappa$, since $P_k(a)$ and $P_k(b)$ are indistinguishable from random, and every bit position causes a and b to merge with probability $\frac{1}{4}$. The probability that at least one of the $p(p-1)$ pairs (a,b) satisfies $m(k,a,b) \geq \kappa$ is at most $p^2 \cdot (\frac{3}{4})^\kappa$. This is $\mathrm{negl}(\kappa)$. □

Now we will prove a lemma about the maximum length of a SEQHASH round.

LEMMA A.5 (SEQHASH ROUND LENGTH). *For every probabilistic polynomial-time adversary Adv, for every sufficiently large security parameter κ,*

$$\begin{aligned}\Pr[&k \leftarrow \{0,1\}^\kappa;\\ &\mathsf{seq} \leftarrow \mathsf{Adv}(P_k(\cdot));\\ &\mathsf{MaxRound}(k,\mathsf{seq}) \geq \kappa] \leq \mathrm{negl}(\kappa)\end{aligned}$$

where $\mathsf{MaxRound}(k,\mathsf{seq})$ is the maximum number of bits consumed in any round while computing $\mathrm{SEQHASH}(k,\mathsf{seq})$.

PROOF. Assume there exists such an adversary Adv that succeeds with non-negligible probability. Since we assume that P is indifferentiable from a random oracle, let p be the number of times that Adv invokes $P_k(\cdot)$. Since Adv is polynomial-time, p is polynomial in 1^κ. Without loss of generality, include in p all invocations of $P_k(\cdot)$ needed to compute SEQHASH on the sequence produced by Adv, even if Adv did not make such queries. Because P is a PRF, the exact arguments on which $\mathsf{Adv}(k)$ queries P are irrelevant, so without loss of generality, assume they are $1,2,\ldots,p$.

Since Adv produces seq with $\mathsf{MaxRound}(k,\mathsf{seq}) \geq \kappa$, it must be that $\mathrm{SEQHASH}(k,\mathsf{seq})$ runs into some pair of distinct $a,b \in \{1,\ldots,p\}$ such that $m(k,a,b) \geq \kappa$. But according to Lemma A.4, the probability that this would be true for any such pair a,b is negligible. Thus, Adv cannot succeed with non-negligible probability. □

Now we can prove the theorem about efficiency.

THEOREM A.6. SEQHASH *is efficient, as defined in Definition A.3.*

PROOF. Suppose there exists an adversary Adv that, given argument k, constructs a sequence seq such that

$$\mathsf{NumTrees}(\mathrm{SEQHASH}(k,\mathsf{seq})) \geq 2\kappa \log n,$$

with non-negligible probability. By construction, SEQHASH runs for $O(\log n)$ rounds, so there must be some round that produces at least 2κ trees. The number of trees produced in a round is bounded by the number of unmerged nodes, which is bounded by $2\times$ the length of the round (number of bits consumed). But by Lemma A.5, no adversary can construct a sequence that causes a round to consume at least κ bits, with non-negligible probability. So such an adversary Adv cannot exist. □