ALITHEIA: Towards Practical Verifiable Graph Processing

Yupeng Zhang
ECE Dept. & UMIACS
University of Maryland
zhangyp@umd.edu

Charalampos Papamanthou
ECE Dept. & UMIACS
University of Maryland
cpap@umd.edu

Jonathan Katz
Computer Science Dept. & UMIACS
University of Maryland
jkatz@cs.umd.edu

ABSTRACT

We consider a scenario in which a data owner outsources storage of a large graph to an untrusted server; the server performs computations on this graph in response to queries from a client (whether the data owner or others), and the goal is to ensure *verifiability* of the returned results. Existing work on verifiable computation (VC) would compile each graph computation to a circuit or a RAM program and then use generic techniques to produce a cryptographic proof of correctness for the result. Such an approach will incur large overhead, especially in the proof-computation time.

In this work we address the above by designing, building, and evaluating ALITHEIA, a nearly practical VC system tailored for graph queries such as computing shortest paths, longest paths, and maximum flow. The underlying principle of ALITHEIA is to minimize the use of generic VC systems by leveraging various algorithmic techniques specific for graphs. This leads to both theoretical and practical improvements. Asymptotically, it improves the complexity of proof computation by at least a logarithmic factor. On the practical side, we show that ALITHEIA achieves significant performance improvements over current state-of-the-art (up to a $10^8\times$ improvement in proof-computation time, and a 99.9% reduction in server storage), while scaling to 200,000-node graphs.

Categories and Subject Descriptors

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

Keywords

Verifiable Computation; Graph Processing; Cloud Computing

1. INTRODUCTION

Graph algorithms are everywhere. For instance, navigation systems run the Dijkstra or Floyd-Warshall algorithms to compute the shortest route between two locations, and various problems in transportation networks can be modeled as maximum-flow computations. In the era of cloud computing, however, the owner of the (data underlying the) graph may not be the same entity running computations over this graph. Specifically, a (trusted) data owner with small local memory might outsource storage of a large graph to a server, who will then answer queries about the graph made by various clients. The goal is to ensure *verifiability* of the returned results, thus protecting clients against bugs in the server's code, malicious behavior by the server, or server compromise. (The naive solution of having the data owner authenticate the graph, and then having the client download/verify the graph and compute the result on its own, imposes unacceptable bandwidth, storage, and computational costs on the client.)

Precisely this setting is addressed by early work on *authenticated data structures* [34, 28, 5], as well as more recent work on the broader problem of *verifiable computation* (VC) [18, 14, 9, 10, 19, 11, 33, 32, 30, 7, 12, 36]. Such schemes enable the server to provide a cryptographic *proof* of correctness of the returned result, which can be verified by the client posing the query.

There are several parameters of interest when it comes to designing VC protocols. Perhaps the main concerns are that the *size* of the proof should be small (ideally, proportional to the size of the result itself), and the *verification time* for the client should be low. (We stress, however, that in our setting we are mainly interested in outsourcing *storage* rather than *computation*.) These is particularly important when proof verification might be done by resource-constrained clients (e.g., smartphones). Other measures of interest include the *time for the server to compute the proof* (especially important for latency), the *storage* required by both the server and the clients, and the *preprocessing time* required by the data owner.

A VC protocol for graph algorithms can be derived in theory via black-box use of existing general-purpose VC schemes [18, 14, 24, 9, 10, 19, 11], or one of the systems that have been built to apply these techniques [33, 32, 30, 7, 12, 36]. Applying these general-purpose protocols to graph algorithms, however, does not necessarily yield practical protocols. (See Table 1.) This is partly just a consequence of their generality, but is more specifically due to the fact that most of these systems require the computation being performed to be represented as a (boolean or arithmetic) circuit. For graph computations, a circuit-based representation will not be optimal, and a RAM-based representation is preferable. Recent work on RAM-based VC schemes by Braun et al. [12] and Ben-Sasson et al. [7], however, are not quite practical yet—see Table 1.

Our contributions. To address the above problems we design, build, and evaluate ALITHEIA, a system for nearly practical verifiable graph processing. Currently, ALITHEIA handles shortest-path, longest-path, and maximum-flow queries over weighted directed and undirected graphs (as applicable). The specific contributions of ALITHEIA are as follows:

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660354 .

Table 1: Summary of our results, and comparison with existing approaches, for verifying shortest-path queries. For asymptotic results, we consider a planar graph with n nodes and m edges, and let $|p|$ denote the length of the shortest path. Experimental results refer to the (planar) road network of the city of Rome ($n = 3{,}353$ nodes, $m = 8{,}870$ edges, $|p| = 13$), taken from the 9th DIMACS implementation challenge for shortest paths [3]. Nodes in this graph correspond to intersections between roads, and edges correspond to road segments. Results marked with * are estimates (see Section 6), since we could only run [30, 12] on small graphs and the full implementation of [7] is not yet available.

	PINOCCHIO [30] (Circuit-Based VC)	PANTRY [12] (RAM-Based VC)	SNARKS FOR C [7] (RAM-Based VC)	ALITHEIA (general graphs)	ALITHEIA (planar graphs)
Preprocessing Time	$O(nm)$	$O(m \log m)$	$O(m \log m)$	$O(m)$	$O(m\sqrt{m} \log m)$
Prover Time	$O(nm \log m)$	$O(m \log^2 m)$	$O(m \log^2 m)$	$O(m \log m)$	$O(\sqrt{m} \log^2 m)$
Proof Size	$O(1)$	$O(1)$	$O(1)$	$O(\|p\|)$	$O(\log m + \|p\|)$
Verification Time	$O(\|p\|)$	$O(\|p\|)$	$O(\|p\|)$	$O(\|p\|)$	$O(\log m + \|p\|)$
Preprocessing Time	19,000 hours*	270 hours*	52 hours*	69 minutes	4.1 minutes
Prover Time	7,600 hours*	550 hours*	30 hours*	3.3 minutes	13 seconds
Proof Size	288 bytes	288 bytes	288 bytes	704 bytes	3,872 bytes
Verification Time	0.008 seconds*	0.008 seconds*	0.049 seconds*	0.19 seconds	0.592 seconds

1. On the theoretical side, ALITHEIA asymptotically reduces the preprocessing and proof-computation time relative to the best previous approaches. E.g., for shortest-path queries in a graph with m edges, it saves a factor of $O(\log m)$ (see Section 3). For *planar* graphs (ones that can be drawn in the plane without edge crossings), ALITHEIA reduces the number of cryptographic operations needed for proof computation even more, by a factor of $O(\sqrt{m})$ (Section 4).

2. On the practical side, ALITHEIA achieves significant performance improvements, bringing verifiable computation on graphs closer to practice (Section 5). Specifically, compared to existing state-of-the-art it improves the running time of the prover by over a factor of 10^8 (for graphs with 100,000 nodes), and the server storage by 99.9%. We validate such performance savings on real-world graphs as well (e.g., the numbers in Table 1 are from running ALITHEIA on the road network of the city of Rome [3]). Finally, ALITHEIA is the first VC system that can scale up to 200,000-node graphs.

3. ALITHEIA supports *dynamic* updates (where edges can be added to or removed from the original graph) in logarithmic time (Section 3.4). Some existing RAM-based VC schemes support such updates in theory (e.g., see [12]), but not in practice for graphs of reasonable size.

Table 1 provides a detailed theoretical and practical comparison of ALITHEIA with current state-of-the-art systems [30, 12, 7] for answering shortest-path queries on general graphs. Especially for planar graphs, ALITHEIA offers the best practical performance among all presented schemes. Preprocessing time and proof-computation time are much improved, at the expense of a reasonable increase in the proof size and verification time. We note also that although the overall asymptotic complexity for preprocessing planar graphs using ALITHEIA is larger than for general graphs, the number of *cryptographic operations* needed in the planar case is also $O(m)$, but with a much lower constant. Asymptotically, for general graphs, although the proof size in ALITHEIA increases from $O(1)$ to $O(|p|)$ (where p is the shortest path), the protocol's asymptotic bandwidth remains the same (i.e., $O(|p|)$) since the actual path p must be communicated to the client anyway.

Our techniques. We first briefly describe the approach used by ALITHEIA for handling shortest-path queries in a general, non-planar (un)directed graph $G = (V, E)$; see Section 3 for further details. Consider a request for the shortest path from some node s to another node t. At a high level, what we want to do is to encode correct computation of the result as an NP statement whose validity can then be verified using existing systems (e.g., [30, 12]). The naive way to do this would be to certify correct execution of, say, Dijkstra's shortest-path algorithm on the given inputs; this approach, however, would be prohibitively slow. Instead, we rely on a *certifying algorithm* for shortest paths [31]. This allows us to encode the correct result as a simple set of constraints (see Relation 1) on the shortest paths from s to all nodes in the graph, which can be computed by the server with no cryptographic work. The only cryptographic work required is for the verification of these constraints using existing systems. We use similar techniques to design verifiable protocols for longest-path and maximum-flow queries.

Although the above technique significantly reduces the practical overhead of existing solutions, it requires the use of a general-purpose system (in our case, PINOCCHIO) on a relation of size $O(m)$, where m is the number of edges of the graph. Unfortunately, as we show in our experiments, such an approach does not scale for graphs that exceed 10,000 nodes. We address this problem in Section 4 (and scale up to 200,000 nodes) by taking advantage of the special structure of *planar* graphs, i.e., those that can be embedded in the plane without any crossings. Planar graphs are interesting in our context since they generally provide a good model for vehicular and road networks used by navigation applications. We derive a more efficient protocol for shortest-path queries in planar graphs by leveraging a data structure based on the celebrated *planar separator theorem* [22]. This data structure answers shortest-path queries in $O(\sqrt{n} \log n)$ time, and its main operation relies on performing a `MIN` computation over the sum of two vectors of size $O(\sqrt{n})$. To verify the data structure's operation, we cannot use common authenticated data-structure techniques (e.g., [34, 28, 5]) since these would yield proofs of $\Omega(\sqrt{n})$ size, where n is the number of nodes in the graph. Instead, we achieve logarithmic-sized proofs by using a general-purpose system only on the `MIN` relation. This approach, combined with an additively-homomorphic vector commitment scheme [27], yields an improvement of 29× in the prover time for graphs with 10,000 nodes (compared to the approach described before), and allows us to produce shortest-path proofs on graphs with up to 200,000 nodes.

Other related work. We have already discussed generic VC protocols above, so here we only briefly mention the few prior VC protocols we are aware of that are specifically tailored to graph computations. Yiu et al. [37] presented a verifiable protocol for shortest-path queries. However, although their proof-computation

time is shorter than ours, their protocols have worst-case proof size *linear* in the number of the edges of the graph. Goodrich et al. [20] presented authenticated data structures for various graph queries such as graph connectivity/biconnectivity but their work does not cover advanced graph computations such as shortest-path queries.

Using certifying algorithms for fast cryptographic verification has been proposed for NP-complete problems on graphs [7], where it is clear that, assuming P≠NP, verifying is cheaper than computing. This is known to be the case for only a few problems in P. Efficient and simple certifying algorithms have been used for verifying set queries [29] and data-structure queries [35], but to the best of our knowledge they have not been used for graph queries.

2. PRELIMINARIES

We now present definitions for the cryptographic primitives that we use: (1) *verifiable computation for graphs*, (2) *vector commitments* [13], and (3) *succinct non-interactive argument of knowledge* (SNARKs) [19].

Verifiable computation for graphs. In our setting, there are three parties: A trusted *data owner*, an untrusted *server*, and a *client* (who may also correspond to the data owner). The data owner outsources storage of a graph G to the untrusted server, who answers queries posed by the client.

DEFINITION 1 (VC FOR GRAPHS). *A VC scheme $\mathcal{V}$ consists of 3 PPT algorithms:* ***(1)*** $\{\mathsf{ek}_G, \mathsf{vk}_G\} \leftarrow \mathsf{genkey}(1^k, G)$*: Given a graph G, it outputs evaluation key ek_G and verification key vk_G;* ***(2)*** $\{\pi_q, \alpha\} \leftarrow \mathsf{compute}(q, \mathsf{ek}_G)$*: On input a graph query q, it outputs a proof π_q and an answer α;* ***(3)*** $\{0, 1\} \leftarrow \mathsf{verify}(\pi_q, q, \alpha, \mathsf{vk}_G)$*: On input π_q, q, and α, it outputs* 0 *or* 1.

The above algorithms are used as follows. First, the data owner executes genkey and sends the evaluation key ek_G and the graph G to the server. The verification key vk_G can be published (in the setting of public verifiability) or held privately by the data owner (in the setting of private verifiability). The client, who is assumed to know vk_G, can then send a query q to the server (e.g., q might ask for the shortest path from s to t). The server computes the answer α (e.g., the shortest path p) and proof π_q using compute. The client then verifies the validity of the response α by executing verify. Definitions of correctness and security for VC for graphs can be found in the Appendix (see Definition 3).

Vector commitments. A vector commitment scheme (VCS) enables a prover to commit to a vector $\mathbf{S}$ via a small digest $d(\mathbf{S})$ (usually of constant size). The prover can later open specific entries of $\mathbf{S}$. A concrete example is given by a Merkle hash tree [23], if the entries of the vector are placed at the leaves of the tree. Definitions of a VCS and it security requirements are given in the Appendix (see Definition 4); we stress that for our applications we care about binding but not privacy.

In our protocol for planar graphs (see Section 4), we use a VCS that is *additively-homomorphic*, i.e., having the property that for any two vectors $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$, it holds that $d(\boldsymbol{S}_1) + d(\boldsymbol{S}_1) = d(\boldsymbol{S}_1 + \boldsymbol{S}_2)$. We implement an additively-homomorphic VCS (see Figure 10 in the Appendix) using the streaming authenticated data structure of Papamanthou et al. [27], which is based on a cryptographic assumption related to lattices.

SNARKs. A SNARK enables an untrusted prover to prove that some statement x is indeed in some NP language L. Specifically, what is proved is that there exists a valid NP witness w for x. The proof for such a statement is succinct (i.e., constant size), even if the witness w is large. In the following definition, we let a two-input circuit C define the language L where $x \in L$ iff there exists a w such that $C(x, w) = 1$.

DEFINITION 2 (SNARK [30]). *A SNARK scheme $\mathcal{G}$ consists of three PPT algorithms:* ***(1)*** $\{\mathsf{ek}_L, \mathsf{vk}_L\} \leftarrow \mathsf{genkey}(1^k, C)$*: On input the security parameter and a circuit C, output evaluation key ek_L and verification key vk_L;* ***(2)*** $\pi_x \leftarrow \mathsf{compute}(x, \mathsf{ek}_L)$*: On input $x \in L$, output proof π_x;* ***(3)*** $\{0, 1\} \leftarrow \mathsf{verify}(\pi_x, x, \mathsf{vk}_L)$*: On input π_x and x, output either* 0 *or* 1.

Definitions of correctness and security for SNARKs can be found in the Appendix (see Definition 5). As in the case of VC, a SNARK can be publicly or privately verifiable. As with VC, a SNARK is secure if no polynomially-bounded adversary can compute a proof π and a statement $x \notin L$ such that $1 \leftarrow \mathsf{verify}(\pi, x, \mathsf{vk}_L)$, except with negligible probability. One important property of SNARKs, not shared with VC, is that they support *extractability*; namely, there is an extractor that can use a valid proof π_x for x to extract the corresponding NP witness w for x.

As we discuss in Section 5, we use the recent SNARK implementation by Parno et al. [30] in our schemes. In their SNARK, the complexity of genkey is $O(|C|)$, the complexity of compute is $O(|C| \log |C|)$, and the complexity of verify is $O(|x|)$ (i.e., the length of the first input to C). Also, the size of the proof π_x is $O(1)$ (288 bytes), the size of the evaluation key is $O(|C|)$, and the size of the verification key is $O(|x|)$. We note here that the extra multiplicative logarithmic overhead of a SNARK is removed by the bootstrapping approach of [10], implemented in [8]. However, this does not change the bottom line of our work since both the generic and tailored solution benefit.

3. VC FOR GENERAL GRAPHS

Let $G = (V, E)$ be an (un)directed graph with positive weights c_{uv} on its edges. Set $|V| = n$ and $|E| = m$. Let $p = v_1 v_2 \dots v_k$ denote a path in G of length $|p| = \sum_{i=1}^{k-1} c_{v_i v_{i+1}}$. In this section we show how to construct VC schemes and protocols for general (un)directed graphs. Sections 3.1, 3.2, 3.3, 3.4 refer to shortest paths while Section 3.5 refers to longest paths and maximum flow.

3.1 Strawman Solution

We can construct a VC scheme for shortest paths in a graph G as follows. In the genkey algorithm of the VC scheme, we compute and then cryptographically sign the shortest paths (u, v, p_{uv}) for all $(u, v) \in V \times V$. Then we output *all* the signatures as the evaluation key and the public key of the signature as the verification key. Depending on the user query, the compute algorithm just returns the respective signature along with the path itself. The verify algorithm verifies the signature and decides whether the correct path was returned. This solution however is very expensive: It requires $O(n^3)$ cost for setup and produces an evaluation key of $O(n^2)$ size (all the signatures).

3.2 Using General-Purpose Systems

To reduce the asymptotic complexity of the strawman approach above, one can use a general-purpose system for VC/SNARKs [30] to verifiably execute BFS—which we did in Section 5 (assume unit-weight graphs for the purpose of this section). However, most existing systems implementing VC require the computation (in our case, the BFS algorithm) to be expressed as a circuit. This leads to a blow-up in complexity since the size of a circuit implementing BFS is $O(mn)$ (see Figure 3), which is not linear. As we show in the evaluation section, this significantly affects practical performance—apart from the quadratic complexity itself, large constants are also involved.

To avoid expressing the computation as a circuit, one can use recently proposed methods [12, 7] for verifying RAM programs. Roughly speaking, for a RAM program executing in time $T(n)$ using memory of size n, such an approach requires using a VC scheme on a circuit of size $O(T(n)\log n)$. Also, there is the extra multiplicative logarithmic overhead of a SNARK, thus the prover runs in $O(T(n)\log T(n)\log n)$ time. E.g., to verifiably execute BFS, the prover cost is $O(m\log^2 m)$. Unfortunately, as we show in the evaluation section, such approaches do not scale in practice.

3.3 Our Method: Using Certifying Algorithms

Let s be the source and t be the destination of our shortest path query, and let $\mathbf{S}[v]$ denote the distance to node v from node s for $v \in V$ (we view $\mathbf{S}$ as a vector of n entries).

Our approach is based on the observation that in order to verify that path p is the shortest path from s to t in G, there is no need to verify every step of Dijksrta's algorithm that computes p. Instead one can verify a few constraints that need to hold on the claimed distance vector $\mathbf{S}$ and on the path p. (The distance vector $\mathbf{S}$ can be computed *independently* with any algorithm we wish.) Specifically, consider the following NP language

$$\mathcal{F}_G = \left\{ \begin{array}{l} (s,t,p) : \exists\, \mathbf{S} \textit{ such that} : \\ \quad \textbf{(1)}\ \mathbf{S}[s] = 0 \ \wedge\ \mathbf{S}[t] = |p|\,; \\ \quad \textbf{(2)}\ \forall (u,v) \in E :\ \mathbf{S}[v] \le \mathbf{S}[u] + c_{uv}\,. \end{array} \right\}. \quad (1)$$

One can easily see that if $(s,t,p) \in \mathcal{F}_G$ then $|p| \le d_{st}$, where d_{st} is the actual shortest path length from s to t in G (a straightforward proof for that claim can be found in [31]). Therefore we can write

$$p \textit{ is a shortest path from } s \textit{ to } t \textit{ in } G \Leftrightarrow$$
$$(s,t,p) \in \mathcal{F}_G \wedge p \textit{ is an } s\textit{-to-}t \textit{ path in } G\,, \quad (2)$$

since, if p is an s-to-t path in G, there is no way it can be shorter than the shortest path from s to t and therefore it has to be the case that $|p| = d_{st}$. As we see in the following, our VC scheme is based on exactly verifying the above relation.

VC construction. We build the VC scheme for shortest paths as follows. First, in the genkey algorithm, we instantiate a SNARK for $\mathcal{F}_G$ and also sign all the edges (u,v,c_{uv}) of the graph G.

Given query (s,t), the compute algorithm just returns the signatures on the edges (and weights) comprising the shortest path p and a proof π that $(s,t,p) \in \mathcal{F}_G$. Computing the SNARK proof takes time $O(m\log m)$ [30]. Moreover, it scales in practice because $\mathcal{F}_G$ has a very efficient circuit representation (see below).

The verify algorithm outputs `1` only if it all the signatures on the edges of p are valid and the SNARK proof π is correct. The security of the scheme follows directly from the security of the used SNARK for $\mathcal{F}_G$ and the security of the signature scheme.

An efficient circuit for $\mathcal{F}_G$. The constraints in Relation 1 can be represented with an efficient circuit that takes as input (s,t,p) and some distance vector $\mathbf{S}$ and outputs `1` if and only of the constraints are satisfied. The size of this circuit is $O(m)$ since only two random accesses are required: One for accessing $\mathbf{S}[s]$ and one for accessing $\mathbf{S}[t]$ (this is because s and t can change depending on user input). These two accesses can be "unrolled" into an $O(n)$-sized circuit and therefore the circuit's asymptotic size is not affected. The second constraint is easily "hardcoded" into the circuit since the graph G is fixed. The resulting circuit is dramatically simpler than the circuit representing BFS or Dijkstra's algorithm.

THEOREM 1. *Let G be a graph with n nodes and m edges. Our VC scheme for shortest paths in G has (i) $O(m)$ preprocessing time; (ii) $O(m\log m)$ prover time (iii) $O(|p|)$ proof size and (iv) $O(|p|)$ verification time, where p is the output shortest path.*

3.4 Support for Dynamic Graphs

As we showed before, the circuit for $\mathcal{F}_G$ hardcodes the constraints on the edges of the graph. Therefore whenever G changes, one must re-execute the $\mathsf{genkey}(1^k, \mathcal{F}_G)$ algorithm (see Definition 2) to output the new evaluation and verification keys of the SNARK. This results in an $O(m)$ cost for updates. We now describe a VC scheme without this problem, with efficiently updatable verification and evaluation keys. For clarity of presentation, we consider the case where the number of the edges of the graph remains the same (equal to m) and the updates we are supporting are replacement of an edge e with another edge e'.

First, instead of signing every edge (u,v) of G (as we did in the previous construction), we represent G with a matrix $\mathbf{E}$ of n^2 entries such that $(i,j) \in E$ if and only if $\mathbf{E}[i,j] = 1$—otherwise $\mathbf{E}[i,j] = 0$ (sometimes we abuse notation and write $(i,j) \in \mathbf{E}$). Then we use a vector commitment scheme $\mathcal{V}_E$ to produce a graph digest $d_E = \mathcal{V}_E.\mathsf{digest}(\mathbf{E})$ (we view the matrix as a vector in the obvious way). Digest d_E comprises part of the verification key which can be efficiently updated using algorithm update from our additively-homomorphic VCS in Figure 10 in the Appendix.

Now, instead of hardcoding the edge constraints in $\mathcal{F}_G$'s circuit, we can write

$$\mathcal{F}_G = \left\{ \begin{array}{l} (d_E,s,t,p) : \exists\, \mathbf{E} \textit{ and } \mathbf{S} \textit{ such that} : \\ \quad \textbf{(1)}\ \mathcal{V}_E.\mathsf{digest}(\mathbf{E}) = d_E\,; \\ \quad \textbf{(2)}\ \mathbf{S}[s] = 0 \ \wedge\ \mathbf{S}[t] = |p|\,; \\ \quad \textbf{(3)}\ \forall (u,v) \in \mathbf{E} :\ \mathbf{S}[v] \le \mathbf{S}[u] + 1\,. \end{array} \right\}. \quad (3)$$

Although the above representation allows for efficient edge updates (one can just update d_E), it has $\Omega(n^2)$ size since it must iterate through all the entries of $\mathbf{E}$ in constraints (**1**) and (**3**).

To avoid this limitation we can rewrite this language by extending its inputs such that: (i) the edges (i,j) such that $\mathbf{E}[i,j] = 1$ are given as explicit input; (ii) the distances d_i and d_j corresponding to the endpoints of these edges (i,j) are also given as input. All these are captured by the version of $\mathcal{F}_G$ in Figure 9 in the Appendix. It is easy to see that the size of the circuit implementing $\mathcal{F}_G$ in Figure 9 is proportional to $O(m \cdot \mathsf{poly}(\log n))$, where the polylogarithmic factor depends on the type of VCS that we use.

3.5 Longest Paths, Maximum Flows

Finding longest paths in directed acyclic graphs (DAGs) from a source s to a sink t can be used in scheduling (e.g., finding the critical path) or in graph drawing (e.g., computing a layered drawing of a graph). It is straightforward to verify longest paths on DAGs by slightly changing Relation 1. Specifically, for longest paths, one has to check that $\forall (u,v) \in E$ it is $\mathbf{S}[v] \ge \mathbf{S}[u] + c_{uv}$ instead of $\mathbf{S}[v] \le \mathbf{S}[u] + c_{uv}$. Note that the edge set E is now directed.

Additionally, ALITHEIA can handle maximum-flow queries. We use the maxflow-mincut theorem [15] stating that given a directed graph G with source s, sink t, and capacities c_{uv} on the edges $(u,v) \in E$, a maximum flow $\mathbf{F}$ always equals the minimum cut. To verify a maximum flow based on this theorem we build a SNARK that takes as input the source s and destination t, the flow assignement on every edge f, a disjoint partition of the node set (S,T), and the maximum flow $\mathbf{F}$. Besides, the capacity of every edge is hardcoded in the SNARK. The following relation is checked:

$$\mathcal{M}_G = \left\{ \begin{array}{l} (s,t,\mathbf{F})\text{: } \exists\ f \text{ and disjoint } S \text{ and } T \textit{ such that} : \\ \quad \textbf{(1)}\ s \in S \wedge t \in T\,; \\ \quad \textbf{(2)}\ \sum_{e \in out(s)} f_e = \mathbf{F}\,; \\ \quad \textbf{(3)}\ \forall e \in E :\ f_e \le c_e\,; \\ \quad \textbf{(4)}\ \forall u \notin \{s,t\} :\ \sum_{e \in in(u)} f_e = \sum_{e \in out(u)} f_e\,; \\ \quad \textbf{(5)}\ \sum_{e \in S \times T} c_e = \mathbf{F}\,. \end{array} \right\}.$$

We note here that verifying the above maximum flow relation takes $O(m)$ time, while there is no existing algorithm with linear asymptotic complexity to solve maximum flow problem.

4. VC FOR PLANAR GRAPHS

A planar graph is a graph that can be drawn in the plane without any crossings [6]. Planar graphs have various applications, e.g., they can model vehicular and road networks. Due to their special structure, more efficient algorithms are known for planar graphs and ALITHEIA takes advantage of such structure.

Specifically, in this section we construct a VC scheme for verifying shortest path queries in (un)directed planar graphs. Contrary to the case of general graphs, we show that for planar graphs we can construct a prover that runs in time $O(\sqrt{n}\log n)$ (which is equivalent to $O(\sqrt{m}\log m)$ since in planar connected graphs it always is $m = \Theta(n)$). As we will see in the experimental section, this translates into significant practical savings as well, enabling us to scale verifiable computation on 200,000-node graphs for the first time.

4.1 The Planar Separator Data Structure

Our approach is based on a novel data structure that makes use of the planar separator theorem [22]. The planar separator theorem states the following: For every planar graph $G = (V, E)$ of n nodes, one can always partition the vertices V of G in three sets G_1, G_0, G_2 such that $|G_1| \leq \frac{2n}{3}$, $|G_2| \leq \frac{2n}{3}$ and $|G_0| = O(\sqrt{n})$ and such that all the paths from G_1 to G_2 go through G_0. Many data structures have appeared in the literature (e.g., see [16]) that use various versions of the above theorem to answer shortest path queries in sublinear time—instead of quasilinear time that Dijkstra's algorithm would require. We describe one simple such technique (and the one we are using in our approach) in the following.

Data structure setup. Let $G = (V, E)$ be a planar undirected graph of n nodes that has positive weights c_{uv}, where $(u, v) \in E$. We first decompose G into the partition (G_1, G_0, G_2) using the planar separator theorem. Then we recursively apply the planar separator theorem on G_1 and G_2 until we are left with partitions of $O(\sqrt{n})$ nodes. After the recursion terminates, the initial graph G will be represented with a binary tree $\mathcal{T}$ (called separator tree) of $O(\sqrt{n})$ nodes and $O(\log n)$ depth such that an internal tree node $\mathbf{t}$ contains the nodes of the separator of the graph which is induced by the nodes contained in $\mathbf{t}$'s subtree. See Figure 1.

Every internal separator tree node contains $O(\sqrt{n})$ graph nodes. Let now u be a node of the original graph G. We define $path(u)$ to be a tree path from tree node $\mathbf{u}$ that contains u to the root $\mathbf{r}$ of $\mathcal{T}$. The separator tree data structure will contain, for all graph nodes $u \in G$, the following precomputed distances (shortest paths):

$$\{\mathbf{S}_{u\mathbf{p}} : \mathbf{p} \in path(u)\},$$

where $\mathbf{S}_{u\mathbf{p}}$ is a vector of size $|\mathbf{p}|$ storing all the shortest paths s_{uv} from u to all $v \in \mathbf{p}$. Since tree $\mathcal{T}$ has $O(\sqrt{n})$ nodes and each node requires $O(\sqrt{n}\sqrt{n}\log n)$ space, the total space of the data structure is $O(n^{3/2}\log n)$. Note this is a significant improvement over the naive data structure that precomputes the shortest paths and requires $O(n^2)$ space.

Data structure querying. Suppose now we want to query the shortest path from u to v. First locate separator tree nodes $\mathbf{u}$ and $\mathbf{v}$ that contain u and v respectively. We now distinguish the cases:

1. If tree node $\mathbf{u}$ is an ancestor of the tree node $\mathbf{v}$ or vice-versa, then simply return s_{uv} (which was precomputed in setup and is an element of the vector $\mathbf{S}_{uv}$);

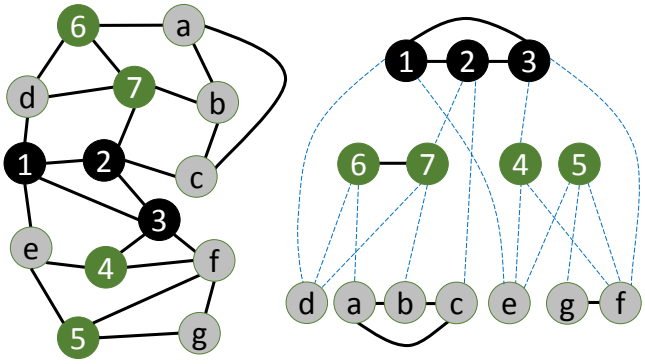

Figure 1: A planar graph (left) along with its planar separator tree (right). Nodes 1, 2, 3 comprise the main separator of the graph and nodes 4, 5 and 6, 7 comprise the separators at the second level.

2. Otherwise, find tree nodes $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k$ that are common in $path(u)$ and $path(v)$. Then return

$$s_{uv} = \min_{i=1,\ldots,k} \{\mathbf{S}_{u\mathbf{t}_i} + \mathbf{S}_{v\mathbf{t}_i}\}, \quad (4)$$

where $+$ above denotes vector addition.

Therefore, by using the separator tree data structure, we have reduced the problem of computing shortest paths on planar graphs to the problem of performing one minimum computation.

The above approach works because all the paths from one node of the planar separator tree to another node of the planar separator tree go through nodes that are common ancestors in the tree. Since the separator tree has $O(\log n)$ levels and each node of it has $O(\sqrt{n})$ nodes, it follows that the output of Relation 4 can be computed in $O(\sqrt{n}\log n)$ time.

4.2 VC Construction

We now present the VC construction for verifying shortest paths in planar graphs using the above data structure.

Setup. At setup, we initialize an additively-homomorphic vector commitment scheme $\mathcal{V}$ (as in Section 2). We also initialize a SNARK scheme $\mathcal{G}$ for the following language

$$L = \left\{ \begin{array}{l} (\mathsf{dig}, min) : \exists\, \mathbf{S} \textit{ and } \mathtt{ind} \textit{ such that}: \\ \quad \textbf{(1)}\ \mathcal{V}.\mathsf{digest}(\mathbf{S}) = \mathsf{dig}\,; \\ \quad \textbf{(2)}\ \mathbf{S}[\mathtt{ind}] \leq \mathbf{S}[i]\ \forall\, i = 0, \ldots, M-1\,; \\ \quad \textbf{(3)}\ \mathbf{S}[\mathtt{ind}] = min\,. \end{array} \right\}. \quad (5)$$

Namely language L contains pairs consisting of a vector digest dig and an element min such that element min is the minimum among the elements contained in the vector represented with dig. The SNARK is used for verifying a relation similar to Relation 4.

Then, working on graph G, we build the planar separator tree $\mathcal{T}$ and compute the shortest path vectors $\mathbf{S}_{u\mathbf{p}}$ for all $u \in G$ and $\mathbf{p} \in path(u)$. Thereafter, we *commit to the shortest path vectors* by computing the digests (using the vector commitment scheme $\mathcal{V}$)

$$d_{u\mathbf{p}} \leftarrow \mathcal{V}.\mathsf{digest}(\mathbf{S}_{u\mathbf{p}}, \mathcal{V}.\mathsf{pk}) \text{ for all } u \in G \text{ and } \mathbf{p} \in path(u).$$

For clarity of presentation, we are going to assume that the verification key of the final VC scheme contains (i) The digests $d_{u\mathbf{p}}$ for all $u \in G$ and $\mathbf{p} \in path(u)$; (ii) the structure $path(u)$ for all $u \in G$; (iii) the graph G itself, along with the weights c_{uv} on the edges (u, v). Although storing all this information requires at least linear space, it is easy to outsource it by computing a digital signature of each object above. Actually, this is how our implementation works (with the difference that an HMAC instead of a signature is used).

Proof computation and verification. In the proof computation phase, a proof must be constructed showing that s_{uv} is the shortest path from u to v. Let now $\mathbf{v}$ be the separator tree node containing graph node v and let $\mathbf{u}$ be the separator tree node containing graph node u. Then we need to distinguish two cases, depending on the location of the nodes u and v on the separator tree. We have the following cases:

Case 1. If $\mathbf{v}$ belongs on the separator tree path from $\mathbf{u}$ to the separator tree root $\mathbf{r}$ (or vice versa), then the shortest path s_{uv} has been precomputed and is an element of the vector $\mathbf{S}_{u\mathbf{v}}$. Therefore it suffices for the prover to return a proof for $\mathbf{S}_{u\mathbf{v}}$'s element that corresponds to the shortest path s_{uv}. Then the verifier can verify this proof using the digest $d_{u\mathbf{v}}$.

Case 2. Otherwise, the prover takes the following steps:

1. It computes the common ancestors $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k$ of $\mathbf{u}$ and $\mathbf{v}$ in the separator tree $\mathcal{T}$. Recall, that, due to the planar separator structure, all the shortest paths from u to v *must* pass through one of these nodes.

2. Let now min_i (for $i = 1, \ldots, k$) be the minimum element of the vector $\mathbf{S}_{u\mathbf{t}_i} + \mathbf{S}_{v\mathbf{t}_i}$, occurring at node $w_i \in \mathbf{t}_i$, i.e., $min_i = s_{uw_i} + s_{vw_i}$. For $i = 1, \ldots, k$, the prover outputs a SNARK proof π_i for $y_i = (d_{u\mathbf{t}_i} + d_{v\mathbf{t}_i}, min_i) \in L$ by calling $\mathcal{G}.\mathsf{compute}(y_i, \mathsf{ek}_L)$. This proof is used to prove that min_i is the minimum of vector $\mathbf{S}_{u\mathbf{t}_i} + \mathbf{S}_{v\mathbf{t}_i}$ (and therefore a potential length for the shortest path from u to v). After all SNARK proofs for min_i ($i = 1, \ldots, k$) are verified by the verifier, then he can verify the length of the shortest path as $\min\{min_1, min_2, \ldots, min_k\}$.

The detailed description of our VC scheme is shown in Figure 2.

Asymptotic complexity and security. Let $n = \Theta(m)$ be the number of nodes in a planar graph. First, the most costly operation of genkey is the computation of the planar separator data structure. By using standard results from the literature [16], this cost is $O(m^{3/2} \log m)$. Note that this is an one-time cost.

As far as algorithm compute is concerned, the cost is dominated by computing one proof using the vector commitment scheme, which takes $O(\log m)$ time—see [27]) and $O(\log m)$ SNARK proofs. Since computing a SNARK proof takes $O(\sqrt{m} \log m)$ time (since the description of the language we are encoding has size $O(\sqrt{m})$), the total worst-case cost of computing the proof is $O(\sqrt{m} \log^2 m)$.

Finally, the size of the proof is $O(\log m + |p|)$ since $O(\log m)$ SNARK proofs must be returned as well as signatures on the edges of the path, and the verification time is $O(\log m + |p|)$. The security of the final VC scheme follows directly from the security of VCS scheme and the security of the SNARK scheme. We summarize the above in the following theorem:

THEOREM 2. *Let G be a planar graph with $m = \Theta(n)$ edges. Our VC scheme for shortest paths in G has (i) $O(m\sqrt{m} \log m)$ preprocessing time; (ii) $O(\sqrt{m} \log^2 m)$ prover time (iii) $O(\log m + |p|)$ proof size and (iv) $O(\log m + |p|)$ verification time, where p is the output shortest path.*

5. IMPLEMENTATION

In this section we present the implementation of ALITHEIA. Recall that the main building blocks used by ALITHEIA are an additively homomorphic vector commitment scheme (VCS) and a SNARK. We give details about those in what follows.

Implementing VCS. A vector commitment scheme (VCS) can be implemented with a Merkle hash tree using a collision-resistant hash function such as SHA-2, e.g., see [23]. However, such a simple scheme is not additively-homomorphic and our planar separator scheme requires to use such a scheme. Luckily enough, additive homomorphism is satisfied by various existing schemes, e.g., see the vector commitment scheme of Catalano and Fiore [13] or the verifiable data streaming scheme of Papamanthou et al. [27].

For practical efficiency reasons (e.g., lack of modular exponentiations), our planar separator implementation uses the vector commitment scheme of Papamanthou et al. [27], the security of which is based on the difficulty of the small integer solution (SIS) problem in lattices (see Assumption 1 in [27]). Also its query and verification complexity is $O(\log^2 M)$, where M is the size of the input vector. The exact algorithms we implemented are in Figure 10 in the Appendix.

Roughly speaking, this scheme is a Merkle hash tree [23] that, instead of SHA-2, uses the lattice-based hash function $h(\mathbf{x}, \mathbf{y}) = \mathbf{Lx} + \mathbf{Ry} \mod q$, where $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k \times m}$ and $\mathbf{x}, \mathbf{y} \in [N]^m$ [4]. Due to additive properties of the lattice-based hash function, the digest

Algorithm $\{\mathsf{ek}_G, \mathsf{vk}_G\} \leftarrow \mathsf{genkey}(1^k, G)$

- $\mathcal{V}.\mathsf{pk} \leftarrow \mathcal{V}.\mathsf{genkey}(1^k, M)$.
- $\{\mathcal{G}.\mathsf{ek}_L, \mathcal{G}.\mathsf{vk}_L\} \leftarrow \mathcal{G}.\mathsf{genkey}(1^k, L)$, where L is from (5).
- Compute planar separator tree $\mathcal{T}$.
- Compute vectors $\mathbf{S}_{u\mathbf{p}}$ $\forall u \in G$ and $\forall \mathbf{p} \in path(u)$.
- Set $d_{u\mathbf{p}} \leftarrow \mathcal{V}.\mathsf{digest}(\mathbf{S}_{u\mathbf{p}}, \mathcal{V}.\mathsf{pk})$ $\forall u \in G$ and $\forall \mathbf{p} \in path(u)$.

Evaluation key ek_G contains keys $\mathcal{V}.\mathsf{pk}$, $\mathcal{G}.\mathsf{ek}_L$, the separator tree $\mathcal{T}$ and the vectors $\mathbf{S}_{u\mathbf{p}}$ for all $u \in G$ and $\mathbf{p} \in path(u)$. Verification key vk_G contains $\mathcal{V}.\mathsf{pk}$, $\mathcal{G}.\mathsf{vk}_L$, the digests $d_{u\mathbf{p}}$ for all $u \in G$ and $\mathbf{p} \in path(u)$ and the information $path(u)$ for all $u \in G$.

Algorithm $\{\pi, p\} \leftarrow \mathsf{compute}((u, v), \mathsf{ek}_G)$

- If $path(u) \subseteq path(v)$ (or vice-versa), output
  $$(s_{uv}, \pi_{uv}) \leftarrow \mathcal{V}.\mathsf{query}(v, \mathbf{S}_{u\mathbf{v}}, \mathcal{V}.\mathsf{pk}), \text{ where } v \in \mathbf{v}.$$
- Otherwise, let $\{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\} = path(u) \bigcap path(v)$.
  For $i = 1, \ldots, k$, let min_i be the minimum element of the vector $\mathbf{S}_{u\mathbf{t}_i} + \mathbf{S}_{v\mathbf{t}_i}$ occurring at graph node $w_i \in \mathbf{t}_i$.
  For $i = 1, \ldots, k$, output $\pi_i \leftarrow \mathcal{G}.\mathsf{compute}(y_i, \mathsf{ek}_L)$, where $y_i = (d_{u\mathbf{t}_i} + d_{v\mathbf{t}_i}, min_i) \in L$.

Output π_{uv} or π_i ($i = 1, \ldots, k$) as π and p as the shortest path.

Algorithm $\{1, 0\} \leftarrow \mathsf{verify}(\pi, (u, v), p, \mathsf{vk}_G)$

- If $path(u) \subseteq path(v)$ (or vice-versa), check that
  $$1 \leftarrow \mathcal{V}.\mathsf{verify}(v, s_{uv}, \pi_{uv}, d_{u\mathbf{v}}, \mathsf{pk}).$$
- Otherwise, let $\{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\} = path(u) \bigcap path(v)$.
  For $i = 1, \ldots, k$ check that
  $$1 \leftarrow \mathcal{G}.\mathsf{verify}(\pi_i, (d_{u\mathbf{t}_i} + d_{v\mathbf{t}_i}, min_i), \mathsf{vk}_L).$$
- Check that path p in G has length $\min\{min_1, \ldots, min_k\}$.
- If all checks succeed output 1, else output 0.

Figure 2: A VC scheme for shortest paths in a planar graph G.

$d(\mathbf{S})$ of the produced Merkle hash tree can be expressed as

$$d(\mathbf{S}) = \sum_i \mathbf{S}[i] f(i) \mod q\,, \tag{6}$$

where $f(i) \in \{0,1\}^m$ is a function of the index i (called a "partial label" in [27]) and $\mathbf{S}[i]$ is the vector element at position i which is a scalar in $\mathbb{Z}_q$. The above digest is clearly additively-homomorphic.

The parameters k, q, m, N used above depend on the size M of the vector and on the maximum vector entry max and are computed by algorithm $\mathsf{genkey}()$ in Figure 10 in the Appendix. Specifically, algorithm $\mathsf{genkey}()$ calls $\mathsf{parameters}()$ which in turn imposes two constraints on the parameters k, q, m, N. The first constraint guarantees that the security reduction from a hard lattices problem to breaking the security of the vector commitment scheme will be successful [25]. The second constraint guarantees a level of security that is at least 100 bits [26]. An example set of parameters that satisfy the above constraints and that we have used in our experiments are $N = 2 \times 10^6$, $k = 850$, $q \approx 10^{8.862}$ and $m = 25500$. These parameters are good for any vector of size M and maximum entry max such that $M \times \mathsf{max} = N = 2 \times 10^6$.

Another reason (besides the additive homomorphism) that we use the lattice-based scheme [27] for implementing VCS is its efficient algebraic nature (i.e., lack of exponentiations), which allows for very efficient implementations in PINOCCHIO [30]. For example, we verified that while PINOCCHIO takes roughly 300 seconds to compute the SHA-1 Merkle hash of a vector of length 100, it only requires 3.6 seconds using the lattice-based hash. This was among the most crucial factors for the efficiency of planar separator implementation.

Existing SNARKs implementations. Parts of our implementation use PINOCCHIO [30], a SNARK implementation which is open-source and implements an optimized version of the SNARK construction presented in [19]. It can support any NP language L. To use it, one needs to write a C program that takes as input the NP statement and the witness and verifies the validity of the witness. Then this program is compiled into a boolean/arithmetic circuit which is used to produce the evaluation and verification keys. Our implementation uses a recently developed version of PINOCCHIO by Kosba et al. [21] that uses open-source libraries for the bilinear pairing function.

We emphasize here that PINOCCHIO is operating in $\mathbb{Z}_p$, where p is a 254-bit prime number, while some computation of our schemes takes place modulo q. For example, to verify the digest in PINOCCHIO computed in Relation 6, the following check is performed:

$$\sum_i \mathbf{S}[i] f(i) \stackrel{?}{=} d_{input} \mod q\,,$$

where d_{input} is the trusted digest input from the client, $\mathbf{S}$ is the untrusted vector input from the server and the whole equation should be evaluated in $\mathbb{Z}_q$, as required by the lattice-based hash function. Since PINOCCHIO does not support modulo-q operations, our implementation lets the server also input the quotient and uses PINOCCHIO to check the following equivalent condition

$$\sum_i \mathbf{S}[i] f(i) \stackrel{?}{=} d_{input} + quot_{input} \times q\,.$$

However, the server can potentially find a fake pair $\mathbf{S}'$ and $quot'_{input}$ which still satisfies the condition in $\mathbb{Z}_p$.[1]

[1] E.g., for any vector $\mathbf{S}'$ the server can set

$$quot_{input} = q^{-1}\left(\sum_i \mathbf{S}'[i] f(i) - d_{input}\right) \mod p\,.$$

```
PINOCCHIO_BFS(G,s)
1   head = first position in Q;
2   tail = first position in Q;
3   Q[tail] = s;
4   tail++;
5   for each node x ∈ V
6       if x = s
7           x.parent = −1;
8   for round = 1 to |V|
9     for each node x ∈ V
10        if x = Q[head]
11          for each node v ∈ Adj[x]
12              if v.visited = 0
13                  v.visited = 1;
14                  v.parent = Q[head];
15                  Q[tail] = v;
16                  tail++;
17    head++;
```

Figure 3: The BFS pseudocode we implemented in PINOCCHIO. Array Q has $|V|$ positions and simulates the queue in BFS.

To solve this problem, we instead let the client input $quot_{input}$. Specifically, $quot_{input}$ can be computed, signed and outsourced together with the digest. Meanwhile, extra conditions are added to bound every element in $\mathbf{S}$ such that it is impossible to overflow p—i.e., we set $0 \leq \mathbf{S}[i] \leq \mathsf{max}$. Note that such conditions are "for free," since they are required to guarantee collision resistance anyways, as mentioned in the beginning of Section 5.

5.1 Implementation of our VC protocols

We now present the implementations of four different VC schemes for shortest paths, one VC scheme for dynamic graphs and one VC scheme for maximum flow, that were described in Section 3 and Section 4. The parts that did not require PINOCCHIO (e.g., building the planar seperator tree) were implemented in C++.

Our implementation replaces expensive digital signatures (e.g., for signing the graph edges) with HMACs from **openssl** [1]. Therefore the VC schemes we are implementing are in the secret-key setting. To make our VC schemes publicly-verifiable without affecting performance, we could use standard techniques that combine Merkle hash trees with one signature of the roothash. To do a fair comparison, in our experiments we "turn off" public verifiability and zero knowledge which are both offered by PINOCCHIO.

Strawman scheme. As our baseline, we implemented the strawman algorithm as described in Section 3.1. Since we will be running experiments on unit-weight graphs (see next section), we use n rounds of BFS instead of the Floyd-Warshall algorithm to precompute and all shortest paths, which has reduced $O(nm)$ complexity (instead of $O(n^3)$). Subsequently we compute an HMAC of each shortest path.

PINOCCHIO BFS scheme. Our second attempt was to execute the BFS code directly in PINOCCHIO. However, due to various limitations analyzed in Section 3.2, we were not able to code up the linear-time algorithm from [15]. Figure 3 shows the BFS pseudocode that we eventually wrote. One of the most important limitations inherent to the circuit representation is that non-constant array index operations must be implemented by iterating through all elements in the array, which is why the complexity increases from $O(m+n)$ to $O(mn)$—see the portions of the pseudocode in Figure 3 (Lines 5-7 and 9-11).

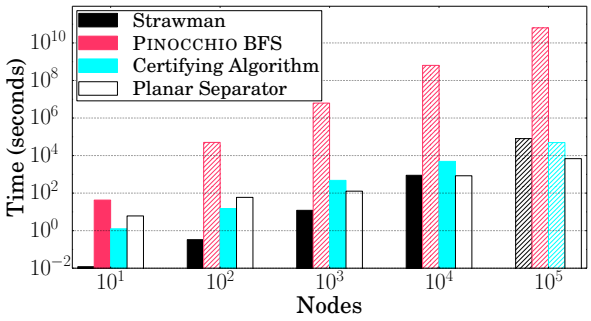

Figure 4: Preprocessing time. We were only able to execute BFS for graphs of up to 50 nodes. All other points (shaded bars) are estimated.

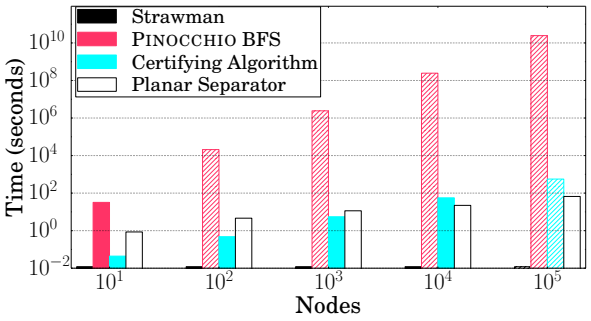

Figure 5: Proof-computation time. We were only able to execute the certifying algorithm for up to 10,000 nodes.

Also, we simulated the BFS queue with an array Q of fixed size since we cannot implement dynamic data structures in a circuit. In Figure 3, the index pointer $head$ records the starting point of the queue and the index pointer $tail$ records the end of the queue. The size of Q is equal to the number of the nodes of the graph, since every node is enqueued and dequeued exactly once.

However, we note that the same technique cannot be easily generalized to Dijkstra's algorithm on a weighted graph, since instead of a plain queue, a more complicated priority queue is required.

Due to the above awkward implementation, the BFS performance was not good and scaled up to only 50 nodes (see more details in the evaluation section).

Certifying algorithm scheme. We implemented the certifying algorithm with an efficient circuit of $O(m)$ size, as explained in Section 3.3. We compute the shortest path vector $\mathbf{S}$ that stores the distances from the source s to all the other nodes of the graph (which the server provides as input to Relation 1) using the BFS implementation in the LEDA library (version 6.4) [2]. We note here that, contrary to BFS, the certifying approach can be naturally applied for weighted graphs—see Relation 1.

During the implementation of Relation 1 in PINOCCHIO, we observed that comparison operations ($\leq, \geq$) are much more expensive than addition and multiplication. Therefore, in the case of a unit-weight graph, we replaced the second constraint of Relation 1 by an equivalent equality constraint with an additional input a_{uv}, i.e.,

$$\mathbf{S}[v] - \mathbf{S}[u] - a_{vu} = 0, \text{where } a_{vu} \in \{-1, 0, 1\}\,.$$

To check the domain of a_{vu}, we write $a_{vu} = a_1 + a_2 - 1$, where $a_1, a_2 \in \{0, 1\}$. To check that $a_i \in \{0, 1\}$ we leverage the equivalence $a_i \times a_i = a_i \Leftrightarrow a_i \in \{0, 1\}$ which can be implemented with one multiplication gate. Similarly, the equality constraint above can be checked by connecting the evaluation of $b_{vu} = \mathbf{S}[v] - \mathbf{S}[u] - a_{vu}$ to the two inputs and one output of an addition gate. In this way, b_{vu} must be 0 to pass the circuit ($b_{vu} + b_{vu} = b_{vu} \Leftrightarrow b_{vu} = 0$).

This optimized version of the certifying algorithm improved the prover performance by 55×. However, this method cannot be applied to graphs with general weights since the domain of the additional input is much larger and checking their domain might even slow down the performance.

We also implemented the **Dynamic Graph Scheme** from Section 3.4 in the same way as the certifying algorithm scheme. **Maximum Flow Scheme** is implemented following the maxflow-mincut theorem described in Section 3.5.

Planar-separator scheme. We implemented the VC scheme for planar graphs described in Figure 2. To build the planar separator tree, we first triangulate the input planar graph using the LEDA library [2] and the triangulated graph is input recursively into the recent planar separator implementation by Fox-Epstein et al. [17]. The digests of the shortest path vectors are computed using our implementation of the additively-homomorphic lattice-based VCS described before. These digests, along with precomputed distances and paths, are HMACed using **openssl** and then outsourced. For every vector of different length storing precomputed distances, a SNARK for Relation 5 using PINOCCHIO is constructed.

6. EVALUATION

We now evaluate the shortest path SNARKs for (i) the strawman scheme; (ii) the PINOCCHIO BFS scheme; (iii) the certifying algorithm scheme; and (iv) the planar separator scheme.

We also present experiments for the dynamic graph scheme as well as for the maximum flow scheme. We do not present results for longest paths since we use the same certifying algorithm as the shortest-path one (with a change in the direction of the inequality).

Experiment setup. We executed our experiments on an Amazon EC2 machine with 15GB of RAM running a Linux kernel.

Plots for *preprocessing time*, *proof-computation time*, *verification time* and *server storage* are presented (in log scale). All schemes were run on the same randomly-generated planar undirected graph (we use the LEDA function `random_planar_graph` for that) with unit weights. We collected 10 runs for each data point, and we report the average in Figures 4, 5, 6, and 7.

Also, in all these figures, estimated data points (due to increased memory/time requirements) are marked as lightly shaded bars. We experiment on planar graphs with $n = 10, 10^2, \ldots, 10^5$, where the number of edges is at most $3(n-2)$ (due to planarity). Our planar scheme was the only one to execute on a graph of 200,000 nodes.

Preprocessing time. Figure 4 shows the comparison of preprocessing time among the schemes. The preprocessing time of both BFS and certifying algorithm schemes is defined as the time to compile the corresponding PINOCCHIO codes into circuits plus the time to generate the keys. The results show that the certifying algorithm outperforms BFS by orders of magnitude. Specifically, the optimized certifying algorithm runs approximately 10,000 times faster than BFS on a graph with 10,000 nodes.

In our experiments, the PINOCCHIO BFS implementation is so inefficient that it takes too long to get a result even on a graph with only 100 nodes. Thus the statistics on graphs with more than 100 nodes for BFS are estimated based on data points on small graphs. All estimated data points are marked as lighted bars in the figures. We use minimum mean square error for the estimation.

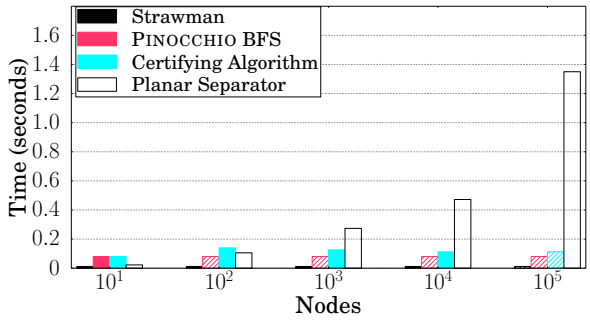

Figure 6: Verification time.

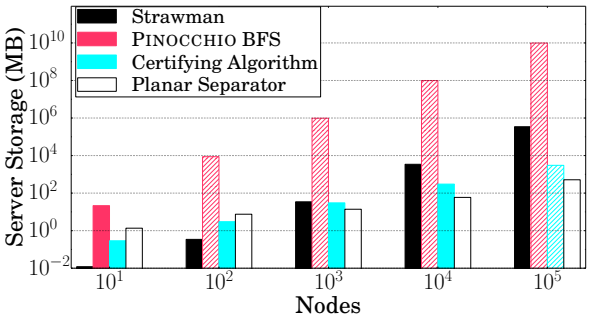

Figure 7: Server storage.

On the contrary, the certifying algorithm schemes can run on larger graphs. The only limitation in this case is that the memory consumption is proportional to the input size (note the input in this case is $O(m)$ instead of $O(1)$ as in the BFS) and our machine runs out of memory when compiling the certifying algorithm code on a graph with more than 10,000 nodes.

Surprisingly, although the complexity of preprocessing time in the planar separator scheme is $O(n^{3/2} \log n)$, it executes faster than the certifying algorithm (that has linear preprocessing time), because most of the work is non-cryptographic and can be implemented efficiently in regular C++ code. In particular, the planar separator scheme outperforms the certifying algorithm scheme in the preprocessing time by 29× on a graph with 10,000 nodes and it can scale to a graph with up to 200,000 nodes. Also, as shown in Table 3, for small graphs, compiling PINOCCHIO codes to circuits and generating keys contribute to the biggest part of the preprocessing, while in large graphs, the portion of the separator tree construction as well as computing and HMACing the precomputed distances and paths dominates most of the time.

Finally, since the strawman scheme does no cryptographic work other than HMACs, it runs extremely fast on small graphs. However, its execution time becomes equal to that of the planar separator scheme on a graph with 10,000 nodes, and is 12× worse on a graph with 100,000 nodes. We had to estimate the preprocessing time of the strawman approach on a graph of 100,000 nodes since storing an all pair shortest path matrix of size 100,000x100,000 requires too much memory (we ran BFS 100,000 times and compute $100{,}000^2$ HMACs to estimate the cost).

Proof-computation time. Figure 5 shows a comparison among the implemented schemes in terms of proof-computation time. The results clearly indicate that the certifying algorithm approach outperforms the BFS approach and that the planar separator approach outperforms both BFS and the certifying algorithm dramatically—which is expected since the planar separator proof-computation time grows sublinearly with the number of nodes. In particular, proof-computation time of the planar separator scheme has a speedup of more than $1.4\text{x}10^5\times$ on a 1,000-node graph, and a speedup of more than $2.3\text{x}10^8\times$ on a 100,000-node graph (compared to BFS).

We note here, that in the case of the planar separator scheme, we report *worst case* results in Figure 5. Worst case proof-computation time is derived when the source s and the destination t of our query are sibling leaves in the planar separator tree, in which case we need to perform the maximum number of `MIN` computations (approximately $O(\log n)$). This is because the proof computation algorithm always examines all common parents of the two planar separator tree nodes containing s and t. On the contrary, if there is only one common parent, namely the root, of the two tree nodes, the proof computation only does one `MIN` computation and is defined as the *best case*. Table 4 shows the comparison between the worst and the best case. It can be observed that the time of the worst case is roughly $\log n$ times the time of the best case. Verification times also have similar relationships.

The planar separator scheme reduces the proof-computation time dramatically and this is one of the main contributions of this work since this metric (proof-computation time) is the most expensive in existing work [30, 12]. For example, our work shows that graph processing can scale up to 200,000 nodes since it takes only tens of seconds as shown in Table 4 (and in the best case, only 7 seconds!) to produce a proof for such large graphs. The exact numbers of our proof-computation time can be found in Table 7 in the Appendix.

Verification time. Figure 6 shows statistics about verification time. In accordance with the asymptotics (see Table 1), the verification for BFS and the certifying algorithm is faster than the planar separator scheme. Still, as shown in Figure 6, the verification time of the planar separator scheme does not grow that much. It requires less than 2 seconds on a graph with 100,000 nodes. Therefore, considering the significant improvements on proof-computation time, the small increase in the verification time is a good trade-off. Finally, similarly to proof-computation time, the verification of the strawman approach requires a small amount of time since it only requires verifying HMACs of the shortest path edges, which is negligible.

Server storage. Here we compare the total amount of storage required on the server side. In the strawman scheme, the server stores the all-pairs-shortest-path matrix and the corresponding HMACs. In the BFS scheme, the server stores the PINOCCHIO circuit and the PINOCCHIO evaluation key. Note that in both these schemes, it is not necessary to store the graph G itself (in the strawman scheme the shortest paths are precomputed; in the BFS scheme the graph is embedded in the circuit) and therefore we do not count the graph size. On the contrary, the certifying algorithm scheme requires the server to store the graph (to compute the shortest paths), the PINOCCHIO circuit, the respective evaluation key and the HMACs of edges—all these are included in the server storage of this scheme. Finally, in the planar separator scheme, the separator tree is part of the server storage in addition to that of the certifying algorithm.

Figure 7 shows the comparison of the server storage. Since the PINOCCHIO evaluation key size is proportional to the size of the circuit and the certifying algorithm has a more efficient circuit implementation than the one of BFS in Figure 3, the server storage for the certifying algorithm is much smaller. In particular, the server storage is reduced by 32340× on a graph with 1,000 nodes and the gap grows on larger graphs.

The planar separator scheme has the least storage requirements. As shown in Figure 7, although the server storage is 4.7× larger on a small graph with 10 nodes, it is reduced by 7× for larger graphs and in particular by 1.9X10^6×compared to BFS on a graph with 100,000 nodes. Finally, the server storage is a major drawback of the strawman approach, as the server needs to store n^2 HMACs (each being 256 bits) for the shortest paths. As such, our planar separator tree scheme outperforms the strawman scheme by 3× on an 1000-node graph and by 669× on an 100,000-node graph.

Proof size. One drawback of our approaches compared to the BFS approach is the proof size. The proof size of the PINOCCHIO BFS scheme is always a constant (288 bytes)—see [30]. However, in our approaches (certifying algorithm and planar separator), the proof contains a 256-bit (32 bytes) HMAC for each edge contained in the shortest path, therefore being proportional to the size $|p|$ of the shortest path p. Specifically, for the case of the certifying algorithm scheme the proof size is $|p| \times 32 + 288$ bytes while for the planar separator scheme the proof size is $|p| \times 32 + 288 \times num$ bytes, where num is the number of PINOCCHIO circuits involved in the proof computation of the planar separator scheme (num is bounded above by the levels of the planar separator tree). As a reference, the proof size of the strawman scheme is $(|p|+1) \times 32$ since the length of the path as well as every path edge needs to be HMACed.

Table 2 compares the proof size of all four schemes for $|p| = 10$. Note that although the proof size of our approach slightly increases, the bandwidth of all approaches is the same and proportional to $|p|$, since the answer p is always required to be returned to the client.

Table 2: Proof size in KB for $|p| = 10$.

n	Strawman	PINOCCHIO BFS	Certifying Algorithm	Planar Separator
100	0.344	0.281	2.781	3.906
1,000	0.344	0.281	2.781	5.312
10,000	0.344	0.281	2.781	6.719

Evaluation of the dynamic graph scheme. We compared the proof-computation time for the dynamic graph scheme (Section 3.4) with the certifying algorithm and the planar separator scheme. As shown in Figure 8, the proof-computation time of the dynamic graph scheme does not scale that well, when compared to the other schemes (which is expected, given its $O(n^2)$ complexity). However, it is still feasible to execute it on small graphs.

For better performance, we also tried to implement the RAM-based, improved version of the dynamic graph scheme described in Figure 9. However, only verifying one digest in step **(3)** takes more than 10 seconds, on a small graph with 16 edges, and there are $2m$ such verifications to be performed, thus it is not practical at this time. Therefore, although the complexity of the algorithm is $O(m \log^2 n)$, the constants involved are really large due to the verification scheme used. We hope to develop more efficient algorithms for verifying RAM-based tasks in the future.

Table 3: Breakdown for planar separator preprocesssing cost.

n	Planar Separator Scheme			
	total time (s)	tree construction	digest and HMAC	PINOCCHIO work
100	59.273	0.20%	0.47%	99.32%
1,000	126.88	2.18%	4.16%	93.19%
10,000	838.924	8.70%	26.44%	57.52%
100,000	6844.024	44.47%	37.50%	18.03%

Evaluation of the maximum flow scheme. Table 5 shows the proof time comparison between implementing the maximum flow algorithm directly on PINOCCHIO and using the certifying algorithm for maximum flow. Actually, maximum flow algorithms are much more complicated than BFS and we could not implement them directly on PINOCCHIO easily. However, we observed that the Edmonds-Karp algorithm [15] computes maximum flows by calling BFS as a subroutine. Thus we use the proof time of BFS on the same graph as a lower bound. Even so, Table 5 shows that the certifying algorithm for maximum flow outperforms the lower bound by orders of magnitude. In particular, the certifying algorithm speeds up by 2405× on a graph with 1,000 nodes.

Table 4: Worst and best case for planar proof-computation time.

n	Planar Separator Scheme			
	proof computation		verification	
	worst case (s)	best case (s)	worst case (s)	best case (s)
1,000	11.425	1.572	0.547	0.063
10,000	22.223	4.099	0.944	0.085
100,000	66.374	6.147	1.350	0.127
200,000	93.853	7.472	1.729	0.194

Table 5: Maximum flow scheme proof-computation time.

n	PINOCCHIO Implementation (s)	Maximum Flow Scheme (s)
10	>32.072	0.701
100	>20,782	31.1509
1,000	>2,413,013	1,003

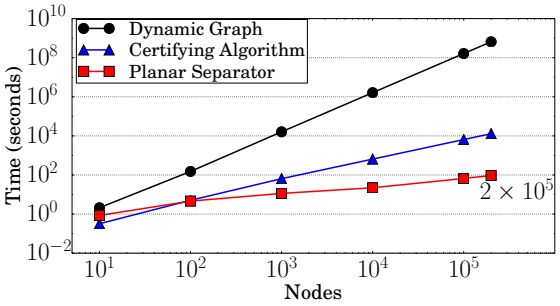

Figure 8: Comparison of the dynamic graph scheme, the certifying algorithm scheme and the planar separator scheme in terms of proof-computation time.

Table 6: PANTRY [12] benchmarks (in seconds).

n	Verifiable PUT	Verifiable GET	estimated PANTRY BFS
100	3.3	2.52	6,200
1,000	30.92	31.22	650,000

6.1 Using RAM-based VC

One of the main sources of inefficiency in our PINOCCHIO implementations is the circuit representation. To overcome this limitation, Braun et al. [12] presented PANTRY, a system for verifying RAM computations. PANTRY solves the non-constant index problem by providing a digest of the memory as an input to the SNARK, along with a Merkle hash tree proof of the accessed index.

We estimated the time that BFS would require to execute on PANTRY by multiplying the time overhead of each RAM PUT/GET operation with the number of such operations the BFS algorithm takes—$3m + 2n + 1$ PUT operations and $3m + n + 2$ GET operations. Table 6 shows the estimated times for running BFS on PANTRY for graphs with 100 and 1,000 nodes.

We also estimated the performance of the system of Ben-Sasson et al. [7]. First, we wrote BFS in TinyRAM. Then we computed

the three parameters that affect the performance of the system: the number of instructions ($L = 127$), the number of cycles ($T = 100m + 27$) and the size of the input ($N = m + 2$). Based on Figure 9 of [8] (a follow-up work of [7]), we used the time for a TinyRAM program with parameters (L, T, N) that are closest to ours. Numbers in column 3 of Table 1 are derived in this way.

Acknowledgments

This research was sponsored in part by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence, or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

We thank Roberto Tamassia for useful discussions.

7. REFERENCES

[1] https://www.openssl.org/docs/crypto/hmac.html.

[2] http://www.algorithmic-solutions.com/leda/index.htm.

[3] http://www.dis.uniroma1.it/challenge9/.

[4] M. Ajtai. Generating Hard Instances of Lattice Problems (extended abstract). In *STOC*, pp. 99–108, 1996.

[5] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *ISC*, pp. 379–393. 2001.

[6] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998

[7] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying Program Executions Succin -ctly and in Zero Knowledge. In *CRYPTO*, pp. 90–108, 2013.

[8] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security*, 2014.

[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 *ITCS*, pp. 326–349, 2012.

[10] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive Composition and Bootstrapping for SNARKS and Proof-carrying Data. In *STOC*, pp. 111–120, 2013.

[11] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct Non-interactive Arguments via Linear Interactive Proofs. In *TCC*, pp. 315–333, 2013.

[12] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying Computations with State. In *SOSP*, pp. 341–357, 2013.

[13] D. Catalano and D. Fiore. Vector Commitments and Their Applications. In *PKC*, pp. 55–72, 2013.

[14] K.-M. Chung, Y. T. Kalai, and S. P. Vadhan. Improved Delegation of Computation Using Fully Homomorphic Encryption. In *CRYPTO*, pp. 483–501, 2010.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Vol. 3.)*. MIT Press, 2009.

[16] J. Fakcharoenphol. Planar Graphs, Negative Weight Edges, Shortest Paths, and Near Linear Time. In *Journal of Computer and System Sciences*, 72(5):868-889, 2006.

[17] E. Fox-Epstein, S. Mozes, P. M. Phothilimthana, and C. Sommer. Short and Simple Cycle Separators in Planar Graphs. In *ALENEX*, pp. 26–40, 2013.

[18] R. Gennaro, C. Gentry, and B. Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *CRYPTO*, pp. 465–482, 2010.

[19] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT*, pp. 626–645, 2013.

[20] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient Authenticated Data Structures for Graph Connectivity and Geometric Search Problems. *Algorithmica*, 60(3):505–552, 2011.

[21] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TrueSet: Faster Verifiable Set Computations. In *USENIX Security*, 2014.

[22] R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[23] R. C. Merkle. A Certified Digital Signature. In *CRYPTO*, pp. 218–238, 1990.

[24] S. Micali. Computationally Sound Proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[25] D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. In *CRYPTO*, pp. 21–39, 2013.

[26] D. Micciancio and O. Regev. Lattice-based Cryptography. In *Post-quantum cryptography*, pp. 147–191, 2009.

[27] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming Authenticated Data Structures. In *EUROCRYPT*, pp. 353–370, 2013.

[28] C. Papamanthou and R. Tamassia. Time and Space Efficient Algorithms for Two-party Authenticated Data Structures. In *ICICS*, pp. 1–15, 2007.

[29] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal Verification of Operations on Dynamic Sets. In *CRYPTO*, pp. 91–110, 2011.

[30] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *SSP*, pp. 238–252, 2013.

[31] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying Algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[32] S. T. V. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the Conflict Between Generality and Plausibility in Verified Computation. In *EUROSYS*, pp. 71–84, 2013.

[33] S. T. V. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making Argument Systems for Outsourced Computation Practical (sometimes). In *NDSS*, 2012.

[34] R. Tamassia. Authenticated Data Structures. In *ESA*, pp. 2–5, 2003.

[35] R. Tamassia and N. Triandopoulos. Certification and Authentication of Data Structures. In *AMW*, 2010.

[36] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A Hybrid Architecture for Interactive Verifiable Computation. In *SSP*, pp. 223–237, 2013.

[37] M. L. Yiu, Y. Lin, and K. Mouratidis. Efficient Verification of Shortest Path Search via Authenticated Hints. In *ICDE*, pp. 237–248, 2010.

$$\mathcal{F}_G = \left\{ (d_E, s, t, p) : \begin{array}{l} \exists \{[e_i = (u_i, v_i), \pi_{e_i}], [d_{u_i}, \pi_{u_i}], [d_{v_i}, \pi_{v_i}]\}_{i=1}^{m} \textit{ and } \mathbf{S} \textit{ and } \mathsf{dig} \textit{ such that :} \\ \textbf{(1)}\ 1 \leftarrow \mathcal{V}_E.\mathsf{verify}(u_i n + v_i, 1, \pi_{e_i}, d_E, \mathsf{pk}) \textit{ for } i = 1, \ldots, m\,; \\ \textbf{(2)}\ \mathcal{V}.\mathsf{digest}(\mathbf{S}, \mathsf{pk}) = \mathsf{dig}\,; \\ \textbf{(3)}\ 1 \leftarrow \mathcal{V}.\mathsf{verify}(u_i, d_{u_i}, \pi_{u_i}, \mathsf{dig}, \mathsf{pk}) \textit{ and } 1 \leftarrow \mathcal{V}.\mathsf{verify}(v_i, d_{v_i}, \pi_{v_i}, \mathsf{dig}, \mathsf{pk}) \textit{ for } i = 1, \ldots, m\,; \\ \textbf{(4)}\ \mathbf{S}[s] = 0 \ \wedge\ \mathbf{S}[t] = |p|\,; \\ \textbf{(5)}\ \mathbf{S}[v_i] \leq \mathbf{S}[u_i] + 1 \textit{ for } i = 1, \ldots, m\,. \end{array} \right\}$$

Figure 9: The language for the dynamic updates. To support dynamic updates, we need to build a SNARK for this language instead of 1.

Appendix

Algorithm $\mathsf{pk} \leftarrow \mathsf{genkey}(1^k, M)$
Set $N = M \cdot \mathsf{max}$, where max is the maximum entry that can appear in vector $\mathbf{S}$. Call $\{q, m\} \leftarrow \mathsf{parameters}(1^k, N)$. Set $\mathsf{pk} = \{\mathbf{L}, \mathbf{R}, q\}$, where $\mathbf{L}, \mathbf{R}$ are matrices picked uniformly at random from $\mathbb{Z}_q^{k \times m}$.

Algorithm $d(\mathbf{S}) \leftarrow \mathsf{digest}(\mathbf{S}, \mathsf{pk})$
Let T be a binary tree built on top of vector $\mathbf{S}$. For each node v of the tree T set $\lambda(v) = \sum_{i \in \mathsf{range}(v)} \mathbf{S}[i]\mathcal{L}_v(i)$, where $\mathcal{L}_v(i)$ is the partial label of node v with respect to i and $\mathsf{range}(v)$ is the range of node v, both defined in [27]. Finally set $d(\mathbf{S}) = \lambda(\epsilon)$, where ϵ is the root of T.

Algorithm $d(\mathbf{S}) \leftarrow \mathsf{update}(d(\mathbf{S}), i, \alpha, \mathsf{pk})$
Set $d(\mathbf{S}) = d(\mathbf{S}) + \mathcal{L}_\epsilon(i)(\alpha - \mathbf{S}[i])$, where $\mathcal{L}_\epsilon(i)$ is the partial label of the root ϵ with respect to i (defined in [27]).

Algorithm $\{\mathbf{S}[i], \Pi_i\} \leftarrow \mathsf{query}(i, \mathbf{S}, \mathsf{pk})$
Let $v_\ell, \ldots, v_1$ be the path in T from node i to the child v_1 of the root ϵ of T. Let also $w_\ell, \ldots, w_1$ be the *sibling nodes* of $v_\ell, \ldots, v_1$. Proof Π_i contains the ordered sequence of pairs $\{(\lambda(v_\ell), \lambda(w_\ell)), (\lambda(v_{\ell-1}), \lambda(w_{\ell-1})), \ldots, (\lambda(v_1), \lambda(w_1))\}$.

Algorithm $\{1, 0\} \leftarrow \mathsf{verify}(i, \mathbf{S}[i], \Pi_i, d(\mathbf{S}), \mathsf{pk})$
Parse proof Π_i as $\{(\lambda(v_\ell), \lambda(w_\ell)), \ldots, (\lambda(v_1), \lambda(w_1))\}$. If $\lambda(v_\ell) \neq \mathbf{S}[i]\mathbf{1}$ or $\lambda(v_\ell), \lambda(w_\ell) \neq [N]^m$, output 0. Otherwise compute the values $y_{\ell-1}, y_{\ell-2}, \ldots, y_0$ as $y_i = \mathbf{L} \cdot \lambda(v_{i+1}) + \mathbf{R} \cdot \lambda(w_{i+1})$ (if v_{i+1} is v_i's left child) or $y_i = \mathbf{R} \cdot \lambda(v_{i+1}) + \mathbf{L} \cdot \lambda(w_{i+1})$ (if v_{i+1} is v_i's right child). For $i = \ell - 1, \ldots, 1$, if $f(\lambda(v_i)) \neq y_i$ **or** $\lambda(v_i), \lambda(w_i) \notin [N]^m$ output 0. If $f(d(\mathbf{S})) \neq y_0$, output 0 (function $f(.)$ takes as input a radix-2 representation and returns the respective number, see [27]). Output 1.

$\{q, m\} \leftarrow \mathsf{parameters}(1^k, N)$.
Let q be a prime and $k \in \mathbb{N}$. Find smallest q and k such that $q/\sqrt{\lceil \log q \rceil} \geq \sqrt{2} \cdot N \cdot k^{0.50001}$ and $\sqrt{2k \log q} < \min\{q, 2^{\sqrt{0.0086 \cdot k \log q}}\}$. Set $m = k\lceil \log q \rceil$.

Figure 10: The additively-homomorphic verifiable vector scheme, adjusted from [27]. Note that the procedure parameters is called by algorithm genkey.

DEFINITION 3. *We say that a VC scheme for graphs $\mathcal{V}$ is correct if, for all graphs G, for all $k \in \mathbb{N}$, for all $\mathsf{ek}_G, \mathsf{vk}_G$ output by algorithm* genkey*, for all queries q on G and for all π_q, α output by algorithm* $\mathsf{compute}(q, \mathsf{ek}_G)$*, it is* $1 \leftarrow \mathsf{verify}(\pi_q, q, \alpha, \mathsf{vk}_G)$.

We say that a VC scheme $\mathcal{V}$ is secure if, for all graphs G, for all $k \in \mathbb{N}$, for all $\mathsf{ek}_G, \mathsf{vk}_G$ output by algorithm genkey *and for any PPT adversary* Adv *it is*

$$\Pr\left[\begin{array}{l} \{q, \pi_q, \alpha\} \leftarrow \mathsf{Adv}(\mathsf{ek}_G, \mathsf{vk}_G); \\ 1 \leftarrow \mathsf{verify}(\pi_q, q, \alpha, \mathsf{vk}_G); \\ \alpha \textit{ is incorrect.} \end{array}\right] \leq \mathsf{neg}(k)\,.$$

DEFINITION 4 (VECTOR COMMITMENT). *A vector commitment scheme $\mathcal{V}$ has five PPT algorithms: **(1)** $\mathsf{pk} \leftarrow \mathsf{genkey}(1^k, M)$: On input the security parameter k and the vector size M, it outputs the public key* pk*; **(2)** $d(\mathbf{S}) \leftarrow \mathsf{digest}(\mathbf{S}, \mathsf{pk})$: On input vector $\mathbf{S}$ and* pk*, it outputs the digest $d(\mathbf{S})$; **(3)** $d(\mathbf{S}) \leftarrow \mathsf{update}(d(\mathbf{S}), i, \alpha, \mathsf{pk})$: On input vector $\mathbf{S}$, (i, α) and* pk*, it sets $\mathbf{S}[i] = \alpha$ and outputs the new digest $d(\mathbf{S})$; **(4)** $(\mathbf{S}[i], \Pi_i) \leftarrow \mathsf{query}(i, \mathbf{S}, \mathsf{pk})$: On input an index i and* pk*, it returns the value $\mathbf{S}[i]$, along with a proof Π_i (run by prover); **(5)** $\{1, 0\} \leftarrow \mathsf{verify}(i, \mathbf{S}[i], \Pi_i, d(\mathbf{S}), \mathsf{pk})$: On input an index i, a value $\mathbf{S}[i]$, a proof Π_i, a digest $d(\mathbf{S})$ and* pk*, it outputs either* 1 *or* 0 *(run by verifier);*

We say that a VCS scheme $\mathcal{V}$ is correct *if, for all $k, M \in \mathbb{N}$, for all* pk *output by* genkey*, for all $\mathbf{S}$ and for all $d(\mathbf{S})$ output by algorithm* $\mathsf{digest}(\mathbf{S}, \mathsf{pk})$ *(or* $\mathsf{update}(d(\mathbf{S}), i, \alpha, \mathsf{pk})$*) and for all $\mathbf{S}[i], \Pi_i$ output by* $\mathsf{query}(i, \mathbf{S}, \mathsf{pk})$*, it is* $1 \leftarrow \mathsf{verify}(i, \mathbf{S}[i], \Pi_i, d(\mathbf{S}), \mathsf{pk})$.

We say that a VCS scheme is secure *if for all $k, M \in \mathbb{N}$, for all* pk *output by algorithm* genkey*, for all $\mathbf{S}$ and for all $d(\mathbf{S})$ output by algorithm* $\mathsf{digest}(\mathbf{S}, \mathsf{pk})$ *(or algorithm* $\mathsf{update}(d(\mathbf{S}), i, \alpha, \mathsf{pk})$*) and for any PPT adversary* Adv *it is*

$$\Pr\left[\begin{array}{l} \{i, \Pi, \alpha\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); \\ 1 \leftarrow \mathsf{verify}(i, \alpha, \Pi, d(\mathbf{S}), \mathsf{pk}); \\ S[i] \neq \alpha. \end{array}\right] \leq \mathsf{neg}(k)\,.$$

DEFINITION 5. *We say that a SNARK $\mathcal{G}$ is* correct *if, for all $k \in \mathbb{N}$, for all NP languages L, for all ek_L and vk_L output by* genkey*, for all $x \in L$, it is* $1 \leftarrow \mathsf{verify}(\mathsf{compute}(x, \mathsf{ek}_L), x, \mathsf{vk}_L)$.

We say that a SNARK $\mathcal{G}$ is secure *if, for all $k \in \mathbb{N}$, for all NP languages L, for all ek_L and vk_L output by algorithm* genkey *and for any PPT adversary* Adv *it is for all $x \in L$, it is*

$$\Pr\left[\begin{array}{l} \{\pi_x, x\} \leftarrow \mathsf{Adv}(1^k, \mathsf{ek}_L, \mathsf{vk}_L); \\ 1 \leftarrow \mathsf{verify}(\pi_x, x, \mathsf{vk}_L); \\ x \notin L. \end{array}\right] \leq \mathsf{neg}(k)\,.$$

A SNARK should also have an extractor: I.e., for any polynomial-sized prover Prv*, there exists an extractor* Ext *such that for any statement x, auxiliary information μ, the following holds:*

$$\Pr\left[\begin{array}{c} \{\mathsf{ek}_L, \mathsf{vk}_L\} \leftarrow \mathsf{genkey}(1^k, L) \\ \pi_x \leftarrow \mathsf{Prv}(\mathsf{ek}_L, x, \mu) \\ 1 \leftarrow \mathsf{verify}(\pi_x, x, \mathsf{vk}_L) \\ \wedge \\ w \leftarrow \mathsf{Ext}(\mathsf{ek}_L, \mathsf{vk}_L, x, \pi_x) \\ w \notin R_L(x) \end{array}\right] \leq \mathsf{negl}(k)\,.$$

Table 7: Proof-computation time (seconds).

	Strawman	PINOCCHIO BFS	Certifying Algorithm	Planar Separator
10	0.01	32.072	0.044	0.853
100	0.01	21,000*	0.477	4.616
1,000	0.01	2,400,000*	5.502	11.425
10,000	0.01	240,000,000*	56.02	22.223
100,000	0.01	25,000,000,000*	560*	66.374
200,000	0.01	98,000,000,000*	1120*	93.853