Verifiable Graph Processing

YUPENG ZHANG and CHARALAMPOS PAPAMANTHOU, ECE Department, University of Maryland, USA
JONATHAN KATZ, Department of Computer Science, University of Maryland, USA

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. Applying generic verifiable computation (VC) would involve compiling each graph computation to a circuit or a RAM program and would incur large overhead, especially in the proof-computation time.

In this work, we address the above by designing, building, and evaluating Alitheia, a VC system tailored for graph queries such as computing shortest paths, longest paths, and maximum flows. The underlying principle of Alitheia is to minimize the use of generic VC techniques by leveraging various algorithmic approaches 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, our system achieves significant performance improvements over current state-of-the-art VC systems (up to a 10-orders-of-magnitude improvement in proof-computation time, and a 99.9% reduction in server storage), while scaling to 200,000-node graphs.

CCS Concepts: • **Security and privacy** → **Cryptography**;

Additional Key Words and Phrases: Verifiable computation, graph processing, cloud computing

ACM Reference format:
Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. 2018. Verifiable Graph Processing. *ACM Trans. Priv. Secur.* 21, 4, Article 20 (September 2018), 23 pages.
https://doi.org/10.1145/3233181

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

This research was sponsored in part by the National Science Foundation under awards #1514261 and #1652259, and by the U.S. Army Research Laboratory and the U.K. Ministry of Defence 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.
Authors' addresses: Y. Zhang, C. Papamanthou, and J. Katz, ECE Department, 3400 A. V. Williams Building, University of Maryland, College Park, MD, 20742; emails: {zhangyp, cpap}@umd.edu, jkatz@cs.umd.edu.

2471-2566/2018/09-ART20 $15.00
https://doi.org/10.1145/3233181

computing, however, the owner of a graph may not be the same entity running computations over that 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, corruption of the data, or malicious behavior by the server.

Precisely this setting is addressed by early work on *authenticated data structures* [4, 36, 41], as well as more recent work on the broader problem of *verifiable computation* (VC) [3, 5–10, 12, 14, 16, 17, 19, 20, 23, 25, 26, 28, 31, 38–40, 43, 44, 46]. Such schemes would enable a server to provide cryptographic proof of correctness for a returned result, which can be verified by the client posing the query.

There are several parameters of interest when it comes to VC protocols. Perhaps the main concerns are that the *size* of the proof should be small (ideally, proportional to the size of the result rather than the size of the graph), and the *verification time* should be low. This is particularly important when proof verification might be done by resource-constrained clients such as smartphones. (We stress, however, that in our setting solutions are interesting even if the verification time is longer than the time to run the computation locally, since we are interested in outsourcing *storage* rather than *computation*.) Other measures of interest include the *time for the server to compute the proof*, the *storage* required by both the server and the clients, and the *setup (i.e., preprocessing) time* required by the data owner.

A trivial solution is to have the data owner authenticate the graph, and then have the client download/verify the graph and compute the result on its own; this solution, however, imposes high bandwidth, storage, and computational costs on the client. Better VC protocols for graph algorithms can be derived in principle using general-purpose VC schemes [9, 10, 12, 17, 25, 26, 34] or one of the systems that have been built to apply these techniques [6–8, 14, 19, 38–40, 43, 44, 46]. Applying these general-purpose results to graph algorithms, however, does not yield practical protocols. This is 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, however, a circuit-based representation will not be optimal. A RAM-based representation would be preferable, but existing RAM-based VC schemes [6–8, 14, 44, 47] are not yet practical (see below).

Our contributions. We design, build, and evaluate Alitheia, a system for nearly practical verifiable computation on graphs. 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:

(1) On the theoretical side, Alitheia asymptotically reduces the preprocessing and proof-computation times relative to the best previous approaches. For example, for shortest-path queries in a graph with m edges, it saves a factor of $O(\log m)$ (see Section 3). (Note that vRAM [47] was published subsequent to the proceedings version of this work.) For *planar* graphs (i.e., graphs that can be drawn in the plane without edge crossings), we show two schemes for verifying shortest paths. The first scheme reduces the number of cryptographic operations needed for proof computation by a factor of $O(\sqrt{m})$; the second scheme only reduces the asymptotic complexity by a logarithmic factor but is more efficient than the first scheme in practice.
(2) On the practical side, Alitheia achieves significant performance improvements, bringing VC for graphs closer to reality (see Section 5). Specifically, for shortest-path queries on a planar graph with 100,000 nodes, the proof-computation time for our first scheme is 50s and for our second scheme is only 0.1s. Compared to existing state-of-the-art approaches, our scheme improves the running time of the server by a factor of 10^7–10^{10} and the server's

Table 1. Summary of Our Results and Comparison with Existing Approaches for Verifying Shortest-path Queries

	Libsnark [7] [44] (circuit-based)	vRAM [47] (RAM-based)	Buffet (RAM-based)	Alitheia general graphs	Alitheia 1 planar graphs	Alitheia 2 planar graphs
Setup	$O(nm)$	$O(m)$	$O(m \log m)$	$O(m)$	$O(m\sqrt{m})$	$O(m\sqrt{m})$
Prover	$O(nm \log m)$	$O(m)$	$O(m \log^2 m)$	$O(m \log m)$	$O(\sqrt{m} \log m)$	$O(m)$
Proof Size	$O(1)$	$O(\text{polylog} m)$	$O(1)$	$O(\lvert p\rvert)$	$O(\log m + \lvert p\rvert)$	$O(\log m + \lvert p\rvert)$
Verification	$O(\lvert p\rvert)$	$O(\text{polylog} m + \lvert p\rvert)$	$O(\lvert p\rvert)$	$O(\lvert p\rvert)$	$O(\log m + \lvert p\rvert)$	$O(\log m + \lvert p\rvert)$
Setup	12,000 hours*	19 hours*	10 hours*	69 minutes	4.1 minutes	2.8 seconds
Prover	1,000 hours*	9 hours*	5 hours*	3.3 minutes	13 seconds	0.0036 seconds
Proof Size	127 bytes	300 kilobytes*	288 bytes	704 bytes	3,872 bytes	1,940 bytes
Verification	0.0014 seconds*	0.11 seconds*	0.008 seconds*	0.19 seconds	0.592 seconds	0.03 seconds

For asymptotic results, we consider a planar graph with n nodes and m edges and let $|p|$ denote the length of the shortest path; reported complexities are for cryptographic operations. Experimental results refer to the (planar) road network of the city of Rome ($n = 3{,}353$, $m = 8{,}870$, $|p| = 13$), taken from the 9th DIMACS implementation challenge for shortest paths [21]. Nodes in this graph correspond to intersections between roads, and edges correspond to road segments. Results marked with * are estimates due to memory limitations (see Section 6).

storage by 99.9%. Finally, Alitheia is the first VC system that can scale to 200,000-node graphs.

In Table 1, we provide a detailed comparison of Alitheia with current state-of-the-art VC systems; specifically, we compare our schemes with Libsnark [7], a circuit-based VC scheme, as well as Buffet[1] [44] and vRAM [47], two RAM-based VC schemes.

Especially for planar graphs, Alitheia offers the best practical performance among all schemes. Preprocessing time and proof-computation time are much improved, at the expense of a reasonable increase in the proof size and verification time. 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. See Section 6 for further discussion.

Our techniques. We first briefly describe the approach used by Alitheia for handling shortest-path queries in a general, (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, we want to encode correct computation of the result as an NP statement whose validity can then be verified using existing systems (e.g., References [14, 38]). 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 [33]. This allows us to encode the correct result as a simple set of constraints (cf. 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 VC system on a relation of size $O(m)$, where m is the number of edges in the graph. Unfortunately, as we show in our experiments, such an approach does not scale for large graphs. We address this in Section 4 (and scale to graphs with up to 200,000

[1]The performance of other RAM-based VCs [6–8, 14] is worse than Buffet; see [44].

nodes) by taking advantage of the special structure of *planar* graphs. Such 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* [32]. This data structure answers shortest-path queries in $O(\sqrt{m})$ time, and its main operation involves performing a MIN computation over the sum of two vectors of length $O(\sqrt{m})$. To verify the data structure's operation, we cannot use common authenticated data-structure techniques (e.g., References [4, 36, 41]), since these would yield proofs of size $\Omega(\sqrt{n})$, where n is the number of nodes in the graph. Instead, we achieve logarithmic-sized proofs by using a general-purpose VC system only on the MIN relation. This approach, combined with an additively homomorphic vector-commitment scheme [35], yields an improvement (compared to the approach described above) of 29× in the prover time for graphs with 10,000 nodes, and allows us to produce shortest-path proofs on graphs with up to 200,000 nodes.

To further eliminate the use of generic VC systems and improve efficiency, we propose a special-purpose vector-commitment scheme that supports verifying a MIN operation on the sum of two vectors. The scheme is inspired by a recent construction of a set accumulator by Zhang et al. [48] and is new to this version of our work. This vector-commitment scheme increases the asymptotic complexity of answering shortest-path queries to $O(m)$, but as shown in Section 6 it improves the prover time in practice by 4000× compared to using a generic VC scheme.

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. [45] 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. [27] 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.

Efficient and simple certifying algorithms have been used for verifying set queries [37] and data-structure queries [42], but to the best of our knowledge they have not been used previously for graph queries.

2 PRELIMINARIES

Define $[z] = \{1, 2, \ldots, z\}$. We let k denote the security parameter and let PPT stand for "probabilistic polynomial time." We use $(\mathbf{a}; \mathbf{b}) \leftarrow (\mathsf{A}||\mathsf{B})$ to denote joint execution of interactive algorithms A and B, where A outputs **a** and B outputs **b**. We use neg to denote a negligible function.

Bilinear pairings. We denote by $(p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^k)$ generation of bilinear-map parameters, where $\mathbb{G}$, $\mathbb{G}_T$ are groups of prime order p with g a generator of $\mathbb{G}$, and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ is an efficiently computable bilinear map, i.e., for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$ it holds that $e(P^a, Q^b) = e(P, Q)^{ab}$.

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 for graphs $\mathcal{V}$ consists of three PPT algorithms: **(1)** genkey takes as input the security parameter and a graph G and outputs an evaluation key ek_G and a verification key vk_G; **(2)** compute takes as input an evaluation key ek_G and a graph query q, and outputs an answer α along with a proof π_q; and **(3)** verify takes as input a verification key vk_G, a query q, and answer α, and a proof π_q, and outputs a bit.

The above algorithms are used as follows. First, the data owner executes genkey and sends the evaluation key ek_G to the server. (We may assume without loss of generality that ek_G includes G itself.) 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 in the graph. The server computes the answer $\alpha = q(G)$ (e.g., the shortest path p) along with a proof π_q using compute. The client then verifies the validity of the response α by executing verify. We define correctness and security as follows.

Definition 2. We say that a VC scheme for graphs $\mathcal{V}$ supporting a set of queries Q is **correct** if for all $k \in \mathbb{N}$, for all graphs G, for all $(\mathsf{ek}_G, \mathsf{vk}_G)$ output by $\mathsf{genkey}(1^k, G)$, for all queries $q \in Q$, and for all α, π_q output by $\mathsf{compute}(\mathsf{ek}_G, q)$, it holds that $\alpha = q(G)$ and $\mathsf{verify}(\mathsf{vk}_G, q, \alpha, \pi_q) = 1$. $\mathcal{V}$ is **secure** if, for any PPT adversary Adv, the following is negligible in k:

$$\Pr\left[\begin{array}{c} G \leftarrow \mathsf{Adv}(1^k); \\ (\mathsf{ek}_G, \mathsf{vk}_G) \leftarrow \mathsf{genkey}(1^k, G); \\ (q, \alpha, \pi_q) \leftarrow \mathsf{Adv}(\mathsf{ek}_G, \mathsf{vk}_G); \\ \end{array} \quad : \quad \mathsf{verify}(\mathsf{vk}_G, q, \alpha, \pi_q) = 1 \bigwedge \alpha \neq q(G) \right].$$

SNARKs. A succinct non-interactive argument of knowledge (SNARK) [26, 38] enables an untrusted prover to prove that some statement x is in some NP language L; specifically, to prove that there exists a w such that $C(x, w) = 1$ for C some Boolean circuit. The proof is required to be succinct, i.e., with size independent of the witness w.

Definition 3 (SNARK). A SNARK $\mathcal{G}$ consists of three PPT algorithms: **(1)** genkey takes as input the security parameter and a circuit C and outputs an evaluation key ek_C and a verification key vk_C; **(2)** compute takes as input an evaluation key ek_C and values x, w, and outputs a proof π_x; **(3)** verify takes as input a verification key vk_C, a value x, and a proof π_x and outputs a bit.

Definitions of correctness and security for SNARKs follow.

Definition 4. SNARK $\mathcal{G}$ is **correct** if, for all $k \in \mathbb{N}$, all circuits C, all $\mathsf{ek}_C, \mathsf{vk}_C$ output by $\mathsf{genkey}(1^k, C)$, all x, w for which $C(x, w) = 1$, and all π_x output by $\mathsf{compute}(\mathsf{ek}_C, x, w)$, it holds that $\mathsf{verify}(\mathsf{vk}_C, x, \pi_x) = 1$.

$\mathcal{G}$ is **secure** if, for all $k \in \mathbb{N}$, all circuits C, and any PPT adversary Adv, the following is negligible in k:

$$\Pr\left[\begin{array}{c} (\mathsf{ek}_C, \mathsf{vk}_C) \leftarrow \mathsf{genkey}(1^k, C); \\ (x, \pi_x) \leftarrow \mathsf{Adv}(1^k, \mathsf{ek}_C, \mathsf{vk}_C) \end{array} \quad : \quad \mathsf{verify}(\mathsf{vk}_C, x, \pi_x) = 1 \bigwedge \nexists w : C(x, w) = 1 \right]$$

and, moreover, if for any polynomial-sized prover Prv there is an extractor Ext such that for any x, the following is negligible in k:

$$\Pr\left[\begin{array}{c} (\mathsf{ek}_C, \mathsf{vk}_C) \leftarrow \mathsf{genkey}(1^k, C); \\ \pi_x \leftarrow \mathsf{Prv}(\mathsf{ek}_C, \mathsf{vk}_C, x; \omega); \\ w \leftarrow \mathsf{Ext}(\mathsf{ek}_C, \mathsf{vk}_C, x, \omega) \end{array} \quad : \quad \mathsf{verify}(\mathsf{vk}_C, x, \pi_x) = 1 \bigwedge C(x, w) \neq 1 \right].$$

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 $|E| = m$; we assume $m = \Omega(n)$. 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 for queries on general (un)directed graphs. Sections 3.1, 3.2, and 3.3 deal with shortest-path queries, while Section 3.4 discusses longest-path and maximum-flow queries.

3.1 Strawman Solution

One trivial approach to constructing a VC scheme for shortest paths in a graph G is for the data owner to simply pre-compute all shortest paths and sign them. That is, the genkey algorithm computes the shortest path p_{uv} for all pairs of vertices $(u, v) \in V \times V$ and signs the resulting tuples (u, v, p_{uv}). The evaluation key includes all these signatures, and the verification key is just the verification key for the signature scheme. The compute algorithm simply returns the appropriate path along with its associated signature; verification is done in the natural way.

Although this solution has reasonable proof size and verification time, it requires $O(n^3)$ setup time and produces an evaluation key of size $O(n^2)$.

3.2 Using General-Purpose Systems

A second idea is to use a SNARK or a generic VC scheme to verifiably execute an algorithm for computing shortest paths. We sketch the idea based on a SNARK (genkey, compute, verify).

Let G be a graph, and let C_G be a circuit that takes as input two vertices u, v along with a path p, and outputs 1 iff p is a shortest path between u and v in G. We can construct a VC scheme for shortest-path queries by having the data owner who holds G run $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{genkey}(1^k, C_G)$ and then output these as the evaluation key and verification key, respectively. Given a pair of vertices u, v, the server computes a shortest path p and then uses the SNARK to compute $\pi_x \leftarrow \mathsf{compute}(\mathsf{ek}, x, w)$, where $x = (u, v, p)$ and w is set to the empty string. The client can then verify correctness of p using verify.

The main drawback of this approach is that a circuit C_G for computing/verifying shortest paths is quite large due to the lack of random-access memory and the need to unroll loops. One could avoid expressing the computation as a circuit by using recently proposed methods [6, 7, 14, 44] for verifying RAM programs; unfortunately, the constant terms involved are quite significant. As we show in Section 5, this severely restricts the practicality of such approaches.

3.3 Our Method: Using Certifying Algorithms

We show here how to avoid using generic VC schemes applied to an entire shortest-path computation. Our main observation is that it instead suffices to apply VC to a small set of constraints.

Fix a graph $G = (V, E)$, and let $s, t \in V$ be the source/destination pair of a shortest-path query. Let $\mathsf{S}[v]$ denote the distance from node s to node v for all $v \in V$, where we view S as a vector of length n. To verify that a shortest s-t path in G has length $|p|$, it suffices to verify that S satisfies

$$\begin{aligned} &1.\ \mathsf{S}[s] = 0 \text{ and } \mathsf{S}[t] = |p|. \\ &2.\ \text{All entries of } \mathsf{S} \text{ are positive.} \\ &3.\ \forall (u, v) \in E: \ \mathsf{S}[v] \le \mathsf{S}[u] + c_{uv}\,. \end{aligned} \tag{1}$$

A straightforward proof can be found in Reference [33].

Let C_G be a circuit that takes s, t, and a path length $|p|$ as inputs, along with a witness S, and outputs 1 iff the above conditions hold. Note that as the graph G is fixed and "hardcoded" into the circuit, conditions 1 and 2 can be checked by an $O(n)$-sized circuit, and condition 3 can be verified by an $O(m)$-sized circuit. Thus, C_G can be expressed as a circuit of size $O(m)$ where the constant term is small, and in particular the resulting circuit will be smaller than a circuit for computing/verifying the shortest path directly as considered in the previous section.

We can use this to build a VC scheme for shortest paths as follows. The genkey algorithm generates parameters for a SNARK for the circuit C_G, and also signs all the edges (u, v, c_{uv}) of the graph G. The evaluation key includes all these signatures, and the verification key contains the SNARK parameters and the verification key for the signature scheme. Given query (s, t), the compute algorithm returns the edges and weights comprising a shortest path p, the corresponding signatures on the edges and weights, and a proof π that there exists $\mathbf{S}$ with $C_G(s, t, |p|; \mathbf{S}) = 1$. The verify algorithm works in the obvious way. Security of the scheme follows directly from the security of the SNARK and the signature scheme. Using the SNARK by Parno et al. [38], we thus obtain:

Theorem 3.1. *Let n, m denote the number of nodes and edges, respectively, in the input graph, and let $|p|$ be the length of the shortest path returned. The VC scheme for shortest paths described above has $O(m)$ preprocessing time, $O(m \log m)$ prover time, and $O(|p|)$ proof length and verification time.*

3.4 Longest Paths and Maximum Flows

It is straightforward to verify longest paths in directed acyclic graphs by slightly changing Relation 1. Specifically, for longest paths, one has to check that for all $(u, v) \in E$ it holds that $\mathbf{S}[v] \geq \mathbf{S}[u] + c_{uv}$ (rather than $\mathbf{S}[v] \leq \mathbf{S}[u] + c_{uv}$).

We can also handle maximum-flow queries in directed graphs by relying on the maxflow-mincut theorem [18], which states that in a directed graph G with source s, sink t, and capacities c_{uv} on the edges $(u, v) \in E$, a maximum flow f always equals the weight of a minimum cut. Thus, given a candidate flow assignment $\mathbf{F}$ on every edge of the graph, and a disjoint partition (S, T) of the nodes (where S and T are binary vectors of length n indicating whether a given vertex is in S, respectively, T), it suffices to check

$$\begin{aligned}
&1.\ S \text{ and } T \text{ form a partition, with } s \in S \text{ and } t \in T.\\
&2.\ \textstyle\sum_{e \in out(s)} \mathbf{F}_e = f.\\
&3.\ \forall e \in E:\ \mathbf{F}_e \leq c_e.\\
&4.\ \forall u \notin \{s, t\}:\ \textstyle\sum_{e \in in(u)} \mathbf{F}_e = \sum_{e \in out(u)} \mathbf{F}_e.\\
&5.\ \textstyle\sum_{e \in S \times T} c_e = f.
\end{aligned} \tag{2}$$

Condition 1 ensures that the source and the sink are on two different sides of the cut. Condition 2 ensures that the outgoing flow of the source node is f. Condition 3 ensures that the flow assignment on every edge is less than or equal to the capacity of that edge. Condition 4 ensures that the incoming flow of every node is equal to the outgoing flow, except for the source and the sink node. Condition 5 ensures that the total capacity from nodes in set S to set T is exactly equal to f, as implied by the maxflow-mincut theorem.

Let C_G be a circuit that takes s, t, f as inputs, along with witness $\mathbf{F}, S, T$, and outputs 1 iff the above conditions hold. We argue that C_G can be implemented as an $O(m)$-size circuit. For condition 1 we check that S and T are complementary and that $S[s] = 1$ and $T[t] = 1$, which can be done by a circuit of size $O(n)$. As the graph is fixed, the indices in the summations of conditions 2 and 4 are predetermined and independent of the inputs and the witness; thus, there is a circuit of size $O(m)$ that can check those conditions. For condition 3, we hardcode the capacity c_e of each edge in the circuit and both $\mathbf{F}_e$ and c_e follow the same ordering of the edges. Therefore, there is a circuit of size $O(m)$ for checking condition 3. As for condition 5, for every edge e with source node u and destination node v we multiply c_e by $S[u]$ and $T[v]$. The result is c_e if e crosses the cut, and 0 otherwise. We then sum up the result and compare to f. This can also be done using an $O(m)$-size circuit. Therefore, verifying the above maximum-flow relation can be done in $O(m)$ time.

4 HANDLING PLANAR GRAPHS

A planar graph is a graph that can be drawn in the plane without any crossings. We remark that in planar graphs it holds that $m = O(n)$. Due to their special structure, algorithms for planar graphs

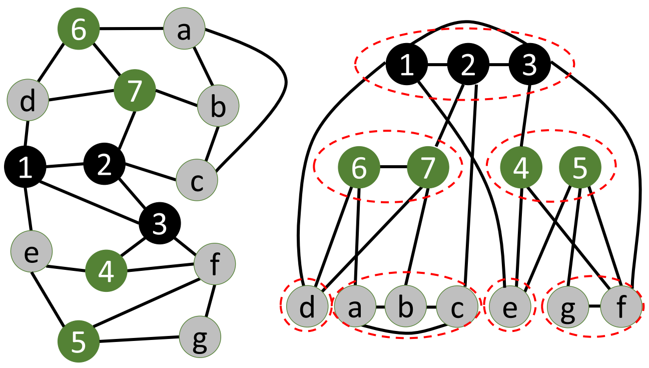

Fig. 1. A planar graph (left) along with its planar separator tree (right). The dotted red circles denote the nodes of the planar separator tree. Nodes 1, 2, 3 comprise the main separator of the graph and nodes 4, 5 and 6, 7 comprise the separators at the second level.

can be more efficient and Alitheia can take advantage of this. Specifically, in this section we construct a VC scheme for verifying shortest-path queries in (un)directed planar graphs. As opposed to the case of general graphs, we show that for planar graphs we can construct a prover that runs in time $O(\sqrt{n}\log n) = O(\sqrt{m}\log m)$. We also propose a scheme that has asymptotically worse prover time ($O(n)$), but runs faster in practice. As we will see in the experimental section, this enables us to scale verifiable computation to 200,000-node graph.

4.1 The Planar Separator Data Structure

Our approach is based on a novel data structure that makes use of the planar separator theorem [32]. The planar separator theorem states that the vertices of every planar graph $G = (V, E)$ on n nodes can be partitioned into three sets G_l, G_m, G_r such that $|G_l| \leq \frac{2n}{3}$, $|G_r| \leq \frac{n}{3}$, and $|G_m| = O(\sqrt{n})$, and such that all the paths from G_l to G_r pass through G_m. Many data structures (see Reference [22]) use various versions of this theorem to answer shortest-path queries in sublinear time—instead of the quasilinear time that Dijkstra's algorithm requires. We describe one simple such technique (and the one we use in Alitheia) in what follows.

Data structure setup. Let $G = (V, E)$ be an undirected planar graph of n nodes where each edge $(u, v) \in E$ has positive weight c_{uv}. We first decompose G into the partition (G_l, G_m, G_r) using the planar separator theorem. Then we recursively apply the planar separator theorem on G_l and G_r until we are left with vertex sets containing $O(\sqrt{n})$ nodes. After the recursion terminates, the initial graph G will be represented by a binary tree $\mathcal{T}$ (called a separator tree) of depth $O(\log n)$ and with $O(\sqrt{n})$ internal nodes where each internal node $\mathbf{t}$ of this tree can be viewed as containing the vertices of the separator of the graph induced by the vertices contained in $\mathbf{t}$'s subtree. See Figure 1.

Every internal node in $\mathcal{T}$ represents a set containing $O(\sqrt{n})$ vertices of the original graph. If u is a node of the original graph, then we define $path(u)$ to be a path in $\mathcal{T}$ from the tree node $\mathbf{u}$ containing u to the root node $\mathbf{r}$ of $\mathcal{T}$. For a tree node $\mathbf{p}$ on $path(u)$, we let $\mathbf{S}_{up}$ be a vector containing the shortest path s_{uv} from u to v for all vertices v contained in (the set corresponding to) ($\mathbf{p}$). Our separator tree data structure contains, for all vertices $u \in V$, the precomputed distances $\{\mathbf{S}_{up} : \mathbf{p} \text{ on } path(u)\}$. The total space required to store the data structure and the time to construct the data structure are both $O(n^{3/2})$.[2] This is a significant improvement over the naive data structure that precomputes the shortest paths between all pairs of nodes that requires $O(n^2)$ space.

[2] A detailed analysis is in Appendix A.

Answering queries. Suppose now a client asks for 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 two cases:

(1) If tree node $\mathbf{u}$ is an ancestor of tree node $\mathbf{v}$ (or vice-versa), then simply return s_{uv}; note this was precomputed during setup and is an element of the vector $\mathrm{S}_{u\mathbf{v}}$.
(2) Otherwise, find tree nodes $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k$ that are on both $path(u)$ and $path(v)$. Then return

$$s_{uv} = \min_{i=1,\ldots,k} \{\min(\mathrm{S}_{u\mathbf{t}_i} + \mathrm{S}_{v\mathbf{t}_i})\}, \tag{3}$$

where + above denotes vector addition.

By using the separator tree data structure, we have reduced the problem of computing shortest paths on planar graphs to the problem of performing vector additions and minimum computations. The output of Equation (3) can be computed in $O(\sqrt{n})$ time.

4.2 Vector Commitment Schemes

To create a verifiable scheme based on the data structure described above, it remains to validate the correctness of the minimum of the sum of two vectors. This can be supported by a *vector commitment scheme* (VCS) [15]. Here we present definitions and propose two constructions.

4.2.1 Definitions. A vector commitment scheme enables a client to generate short digests of vectors and then outsource storage of the vectors to a server. The client can later ask the server to compute various functions on some of those vectors, and should be able to verify the returned results. The particular functions of interest here are membership, and the minimum element of the sum of two vectors. Definitions of a VCS and its security requirements are given below; we stress that for our applications we do not need hiding.

Definition 5. A vector commitment scheme $\mathcal{V}$ consists of the following PPT algorithms:

(1) genkey takes as input the security parameter, a maximum length n, and a bound max, and outputs a public key pk.
(2) digest takes as input pk and a vector $\mathrm{S} \in N_{\max}^{\leq n}$, where $N_{\max} = \{0, \ldots, \max - 1\}$. It outputs a digest d.
(3) Membership takes as input pk, a vector S, and an index i. It outputs the i-th element $S[i]$ along with a proof π.
(4) verify_Membership takes as input pk, a digest d, an index i, a value α, and a proof π, and outputs a bit.
(5) MinSum takes as input pk and two vectors $\mathrm{S}_1, \mathrm{S}_2$ of the same length. It returns the minimum value in $\mathrm{S}_1 + \mathrm{S}_2$, along with a proof π.
(6) verify_MinSum takes as input pk, two digests d_1, d_2, a result min, and a proof π, and outputs a bit.

$\mathcal{V}$ is **correct** if, for all k, n, max, all pk output by genkey, all equal-length $\mathrm{S}_1, \mathrm{S}_2 \in N_{\max}^{\leq n}$, and all d_1 (respectively, d_2) output by $\mathsf{digest}(\mathsf{pk}, \mathrm{S}_1)$ (respectively, $\mathsf{digest}(\mathsf{pk}, \mathrm{S}_2)$):

(1) For all i, if α, π is output by $\mathsf{Membership}(\mathsf{pk}, \mathrm{S}_1, i)$ then $\mathsf{verify\_Membership}(\mathsf{pk}, d_1, i, \alpha, \pi) = 1$.
(2) If α, π is output by $\mathsf{MinSum}(\mathsf{pk}, \mathrm{S}_1, \mathrm{S}_2)$, then $\mathsf{verify\_MinSum}(\mathsf{pk}, d_1, d_2, \alpha, \pi) = 1$.

$\mathcal{V}$ is **secure** if for all k, n, max and any PPT adversary Adv the following are negligible in k:

$$\Pr\left[\begin{array}{c} \mathsf{pk} \leftarrow \mathsf{genkey}(1^k, n, \max); \\ \mathsf{S} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); \\ d \leftarrow \mathsf{digest}(\mathsf{pk}, \mathsf{S}); \\ (i, \alpha, \pi) \leftarrow \mathsf{Adv}(\mathsf{pk}, \mathsf{S}) \end{array} : \begin{array}{c} \mathsf{verify\_Membership}(\mathsf{pk}, d, i, \alpha, \pi) = 1 \\ \bigwedge \mathsf{S}[i] \neq \alpha \end{array}\right].$$

and

$$\Pr\left[\begin{array}{c} \mathsf{pk} \leftarrow \mathsf{genkey}(1^k, n, \max); \\ \mathsf{S}_1, \mathsf{S}_2 \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); \\ d_i \leftarrow \mathsf{digest}(\mathsf{S}_i, \mathsf{pk}) \text{ for } i = 1, 2; \\ (\alpha, \pi) \leftarrow \mathsf{Adv}(\mathsf{pk}, \mathsf{S}_1, \mathsf{S}_2) \end{array} : \begin{array}{c} \mathsf{verify\_MinSum}(\mathsf{pk}, d_1, d_2, \alpha, \pi) \\ \bigwedge \min(\mathsf{S}_1 + \mathsf{S}_2) \neq \alpha. \end{array}\right].$$

4.2.2 A Vector Commitment Scheme From SNARKs. In this section, we present our first construction of a VCS using hashing and general-purpose SNARKs. The idea is very straightforward. In the setup phase, the client builds a Merkle tree from the elements in a vector S and keeps the root $d(\mathsf{S})$; it also initializes a SNARK $\mathcal{G}$ for a circuit taking $\mathsf{dig}_1, \mathsf{dig}_2, min$ as inputs, $\mathsf{S}_1, \mathsf{S}_2$, `ind` as witness, and verifying the following conditions:

$$\begin{aligned} &1. d(\mathsf{S}_1) = \mathsf{dig}_1, d(\mathsf{S}_2) = \mathsf{dig}_2\,; \\ &2. \mathsf{S} = \mathsf{S}_1 + \mathsf{S}_2\,; \\ &3. \mathsf{S}[\mathtt{ind}] \leq \mathsf{S}[i] \text{ for } i = 0, \ldots, M-1\,; \\ &4. \mathsf{S}[\mathtt{ind}] = min. \end{aligned} \tag{4}$$

Namely, the circuit checks that element *min* is the minimum among the elements contained in the sum of two vectors with Merkle roots $\mathsf{dig}_1, \mathsf{dig}_2$. The SNARK is used for verifying a relation similar to Equation (3). Then MinSum simply runs $\mathcal{G}$.compute, and verify_MinSum runs $\mathcal{G}$.verify.

We can use a collision-resistant hash function such as SHA-2 to build the Merkle hash tree. However, such a scheme cannot be efficiently combined with a SNARK. Therefore, we use a SNARK-friendly hash function, the security of which is based on the difficulty of finding small integer solutions in lattices (i.e., the SIS problem). A description of the hash function and an analysis of its security can be found in Reference [49].

For vectors of maximum length n, the setup algorithm runs in $O(n)$ time, Membership and verify_Membership run in $O(\log n)$ time, MinSum runs in $O(n \log n)$ time, and verify_MinSum runs in $O(1)$ time.

4.2.3 A Customized Vector-Commitment Scheme Supporting Addition and Minimum. Though asymptotically close to optimal, the VCS based on SNARKs described above is efficient in practice because of the expensive cryptographic operations involved. Therefore, we propose a second vector commitment scheme that does not use SNARKs at all. As we will show in Section 6, using this customized VCS is 1000× faster (for the prover) than using SNARKs. This second construction is based on recent construction of an expressive set accumulator supporting intersection and minimum queries [48]. We explain the high-level idea of the construction below.

Given two vectors $\mathsf{S}_1, \mathsf{S}_2 \in N^n_{\max}$, we define two bivariate polynomials $f_1(x,y) = \sum_{i=1}^n x^i y^{\mathsf{S}_1[i]}$ and $f_2(x,y) = \sum_{i=1}^n x^{n-i} y^{\mathsf{S}_2[i]}$. Then we have

$$f_1(x,y) \cdot f_2(x,y) = x^n \sum_{i=1}^{n} y^{\mathsf{S}_1[i]+\mathsf{S}_2[i]} + q(x,y),$$

where $q(x,y)$ is a bivariate polynomial without an x^n term, and the element-wise sum of the two vectors for each index i is in the exponent of y with a x^n term in front. To delegate this computation

$\underline{\mathsf{pk} \leftarrow \mathsf{genkey}(1^k, n, \max)}$

Choose uniform $s, r, \alpha \in \mathbb{Z}_p^*$. Run $pub = (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^k)$, and output $\mathsf{pk} = (pub, g^{\alpha}, g^{s^n}, \{g^{s^i}, g^{r^i}, g^{\alpha r^i}\}_{i \in [q]}, \{g^{s^i r^j}\}_{(i,j) \in ([2n] \setminus \{n\}) \times [2\max]})$.

$\underline{d(\mathbf{S}) \leftarrow \mathsf{digest}(\mathbf{S}, \mathsf{pk})}$

Set $d_1(\mathbf{S}) = g^{\sum_{i=1}^n s^i r^{\mathbf{S}[i]}}$ and $d_2(\mathbf{S}) = g^{\sum_{i=1}^n s^{n-i} r^{\mathbf{S}[i]}}$. Output $d(\mathbf{S}) = (d_1(\mathbf{S}), d_2(\mathbf{S}))$.

$\underline{(a, \pi) \leftarrow \mathsf{Membership}(i, \mathbf{S}, \mathsf{pk})}$

Define polynomials $f_1(x, y) = \sum_{j=1}^n x^j y^{\mathbf{S}[j]}$ and $f_2(x, y) = x^{n-i}$. Compute

$$f_1(x, y) \cdot f_2(x, y) = x^n y^{\mathbf{S}[i]} + q(x, y),$$

where $q(x, y)$ is a bivariate polynomial with no x^n term. Set $\pi = g^{q(s,r)}$. Compute α in the natural way.

$\underline{\{\mathtt{1}, \mathtt{0}\} \leftarrow \mathsf{verify\_Membership}(i, a, \pi, d(\mathbf{S}), \mathsf{pk})}$

Output 1 if and only if $e(d_1(\mathbf{S}), g^{s^i}) = e(g^{s^n}, g^{r^a}) \cdot e(g, \pi)$.

$\underline{(min, \pi) \leftarrow \mathsf{MinSum}(\mathbf{S}_1, \mathbf{S}_2, \mathsf{pk})}$

Define polynomials $f_1(x, y) = \sum_{i=1}^n x^i y^{\mathbf{S}_1[i]}$ and $f_2(x, y) = \sum_{i=1}^n x^{n-i} y^{\mathbf{S}_2[i]}$. Compute

$$f_1(x, y) \cdot f_2(x, y) = \sum_{i=1}^n x^n y^{\mathbf{S}_1[i] + \mathbf{S}_2[i]} + q(x, y),$$

where $q(x, y)$ is a bivariate polynomial without the x^n term. Set $\pi_1 = g^{\sum_{i=1}^n r^{\mathbf{S}_1[i] + \mathbf{S}_2[i]}}$, $\pi_2 = g^{\alpha \sum_{i=1}^n r^{\mathbf{S}_1[i] + \mathbf{S}_2[i]}}$, $\pi_3 = g^{q(s,r)}$, $\pi_4 = g^{(\sum_{i=1}^n r^{\mathbf{S}_1[i] + \mathbf{S}_2[i]} - r^{min})/r^{min+1}}$, where min is the minimum element of vector $\mathbf{S}_1 + \mathbf{S}_2$. Set $\pi = (\pi_1, \pi_2, \pi_3, \pi_4)$.

$\underline{\{\mathtt{1}, \mathtt{0}\} \leftarrow \mathsf{verify\_Minsum}(d(\mathbf{S}_1), d(\mathbf{S}_2), min, \pi, \mathsf{pk})}$

Parse π as $\pi_1, \pi_2, \pi_3, \pi_4$. Output 1 if and only if

(1) $e(g^{\alpha}, \pi_1) = e(g, \pi_2)$;
(2) $e(d_1(\mathbf{S}_1), d_2(\mathbf{S}_2)) = e(g^{s^n}, \pi_1) \cdot e(g, \pi_3)$;
(3) $e(g, \pi_1) = e(g, g^{r^{min}}) \cdot e(g^{r^{min+1}}, \pi_4)$;
(4) $0 < min < \max$.

Fig. 2. Our vector commitment scheme supporting computation of the minimum of the sum of two vectors.

to the prover, we pick two random values $s, r \in \mathbb{Z}_p^*$ to replace the variables x, y, and compute the digests $d(\mathrm{S}_1) = g^{f_1(s,r)}$ and $d(\mathrm{S}_2) = g^{f_2(s,r)}$ during the setup phase, where g is the generator of a bilinear group. At query time, given appropriate public keys, the server computes and sends back $g^{\sum_{i=1}^n r^{\mathrm{S}_1[i] + \mathrm{S}_2[i]}}$ and $g^{q(s,r)}$ as the proof, and the verifier validates the relationship in the equation above using a bilinear pairing, which guarantees that $g^{\sum_{i=1}^n r^{\mathrm{S}_1[i] + \mathrm{S}_2[i]}}$ is indeed computed correctly. Finally, observe that the minimum element min in $\mathrm{S}_1 + \mathrm{S}_2$ is the lowest degree of r in $g^{\sum_{i=1}^n r^{\mathrm{S}_1[i] + \mathrm{S}_2[i]}}$, and we have a protocol to compute this verifiably.

Membership queries utilize a similar idea by setting the second polynomial above to $f_2(x, y) = x^{n-i}$ for a query at index i. The full scheme is presented in Figure 2. For simplicity, we write verify_Membership and verify_MinSum as taking all of pk as input, even though they only access a constant number of terms in pk. In practice, a Merkle tree or signatures can be used to reduce the verifier's storage.

Security of the scheme relies on the following assumptions [48]:

Assumption 1 (q-strong bilinear Diffie-Hellman (q-SBDH)). *For all polynomial q and all PPT algorithms A,*

$$\Pr\left[pub \leftarrow \mathsf{BilGen}(1^k); s, r, \alpha \leftarrow \mathbb{Z}_p^*; \sigma = pk; (c, h) \leftarrow A(\sigma) : \ h = e(g, g)^{1/(c+r)}\right] \leq \mathsf{neg}(k).$$

Assumption 2 (q-Power Knowledge of Exponent). *For every PPT adversary A there is a PPT extractor* Ext *such that for any auxiliary information z that comes from a benign distribution*[3]

$$\Pr\left[\begin{array}{c} pub \leftarrow \mathsf{BilGen}(1^k); \\ \alpha, r \leftarrow \mathbb{Z}_p^*; \\ \sigma = (pub, \{g^{r^i}, g^{\alpha r^i}\}_{i\in[q]}); \\ (c, \hat{c}) \leftarrow A(\sigma, z; \omega) \\ \{a_0, \ldots, a_q\} \leftarrow \mathsf{Ext}(\sigma, z, \omega) \end{array} : \hat{c} = c^\alpha \wedge c \neq \prod_{i=0}^{q} g^{a_i r^i}\right] \leq \mathsf{neg}(k).$$

Assumption 3. *For every PPT adversary A*

$$\Pr\left[pub \leftarrow \mathsf{BilGen}(1^k); s, r, \alpha \leftarrow \mathbb{Z}_p^*; (G, h) \leftarrow A(pk) : \ h = g^{s^n G(r)}\right] \leq \mathsf{neg}(k),$$

where pk is defined in Figure 2, and $G(\cdot)$ is a nonconstant polynomial of degree at most $2q$.

A justification of Assumption 3 in the generic group model can be found in Reference [48, Appendix B].

Theorem 4.1. *The vector commitment scheme in Figure 2 is correct and secure under Assumptions 1, 2, and 3.*

Proof. Correctness is immediate, and so we focus on security.
Membership query. Suppose Adv returns $a^* \neq \mathsf{S}[i]$ and passes verification. Then

$$\begin{aligned} e(d_1(\mathsf{S}), g^{s^{n-i}}) &= e(g^{s^n}, g^{r^{a^*}}) \cdot e(g, \pi) \\ \Leftrightarrow e(g,g)^{s^n r^{\mathsf{S}[i]} + q(s,r)} &= e(g,g)^{s^n r^{a^*}} \cdot e(g, \pi) \\ \Leftrightarrow g^{s^n (r^{\mathsf{S}[i]} - r^{a^*})} &= \pi \cdot g^{-q(s,r)}, \end{aligned}$$

violating Assumption 3.
Minsum query. Suppose Adv returns claimed minimum v and proof π_1^* that pass verification. Condition (1), along with Assumption 2, implies that there is an extractor Ext that can can compute $a_0, \ldots, a_{2q}$ such that $\pi_1^* = g^{\sum_{i=0}^{2q} a_i r^i}$. By condition (2),

$$\begin{aligned} e(d_1(\mathsf{S}_1), d_2(\mathsf{S}_2)) &= e(g^{s^n}, g^{\sum_{i=1}^{2q} a_i r^i}) \cdot e(g, \pi_3) \\ \Leftrightarrow e(g,g)^{\sum_{i=1}^{n} s^n r^{\mathsf{S}_1[i]+\mathsf{S}_2[i]} + q(s,r)} &= e(g,g)^{s^n (\sum_{i=1}^{2q} a_i r^i)} \cdot e(g, \pi_3) \\ \Leftrightarrow g^{s^n (\sum_{i=1}^{n} r^{\mathsf{S}_1[i]+\mathsf{S}_2[i]} - \sum_{i=1}^{2q} a_i r^i)} &= \pi_3 / g^{q(s,r)}. \end{aligned}$$

If $\sum_{i=1}^{n} r^{\mathsf{S}_1[i]+\mathsf{S}_2[i]} - \sum_{i=1}^{2q} a_i r^i$ is a nonconstant polynomial, then this violates Assumption 3. Therefore, $\pi_1^* = g^{\sum_{i=1}^{n} r^{\mathsf{S}_1[i]+\mathsf{S}_2[i]} + a_0}$.

Now if $a_0 \neq 0$, then condition (3) implies

$$\begin{aligned} e(g^{\sum_{i=1}^{n} r^{\mathsf{S}_1[i]+\mathsf{S}_2[i]} + a_0}, g) &= e(g^{r^v}, g) \cdot e(g^{r^{v+1}}, \pi_4) \\ \Leftrightarrow e(g,g)^{\frac{a_0}{r}} &= e(g,g)^{r^{v-1}} \cdot e(g, \pi_4)^{r^v} / e(g,g)^{\sum_{i=1}^{n} r^{\mathsf{S}_1[i]+\mathsf{S}_2[i]-1}} \end{aligned}$$

[3]This is defined as in [19, 23, 28]), to avoid the negative results of [11, 13].

$$\Leftrightarrow e(g,g)^{\frac{1}{r}} = (e(g,g)^{r^{v-1}} \cdot e(g,\pi_4)^{r^v} / e(g,g)^{\sum_{i=1}^n r^{S_1[i]+S_2[i]-1}})^{a_0^{-1}}.$$

As $v > 0$ by condition (4), this violates Assumption 1. Therefore, $a_0 = 0$ and $\pi_1 = g^{\sum_{i=1}^n r^{S_1[i]+S_2[i]}}$.

Now suppose $v < min$, then by condition (3),

$$e(g^{\sum_{i=1}^n r^{S_1[i]+S_2[i]}}, g) = e(g^{r^v}, g) \cdot e(g^{r^{v+1}}, \pi_4)$$
$$\Leftrightarrow e(g,g)^{r^v} = e(g^{\sum_{i=1}^n r^{S_1[i]+S_2[i]}}, g) / e(g^{r^{v+1}}, \pi_4)$$
$$\Leftrightarrow e(g,g)^{\frac{1}{r}} = e(g^{\sum_{i=1}^n r^{S_1[i]+S_2[i]-(v+1)}}, g) / e(g,\pi_4),$$

which breaks Assumption 1.

Suppose $v > min$, i.e., $v \geq min + 1$. If j is the index of the correct minimum value, then by condition (3) we have

$$e(g^{\sum_{i=1}^n r^{S_1[i]+S_2[i]}}, g) = e(g^{r^v}, g) \cdot e(g^{r^{v+1}}, \pi_4)$$
$$\Leftrightarrow e(g,g)^{r^{min} + \sum_{i=1, i\neq j}^n r^{S_1[i]+S_2[i]}} = e(g,g)^{r^v} \cdot e(g,\pi_4)^{r^{v+1}}$$
$$\Leftrightarrow e(g,g)^{r^{min}} = e(g,g)^{r^v} \cdot e(g,\pi_4)^{r^{v+1}} \cdot e(g,g)^{-\sum_{i=1, i\neq j}^n r^{S_1[i]+S_2[i]}}$$
$$\Leftrightarrow e(g,g)^{\frac{1}{r}} = e(g,g)^{r^{v-(min+1)}} \cdot e(g,\pi_4)^{r^{v-min}} \cdot e(g,g)^{-\sum_{i=1, i\neq j}^n r^{S_1[i]+S_2[i]-(min+1)}},$$

which breaks Assumption 1. Therefore, $v = min$. □

Complexity analysis. For vectors with maximum length n and maximum value max, algorithm genkey runs in $O(M \cdot \text{max})$ time, setup runs in $O(M)$ time, Membership runs in $O(M)$ time, verify_Membership runs in $O(1)$ time, MinSum runs in $O(M^2)$ time, and verify_MinSum runs in $O(1)$ time. In addition, the size of the public key is $O(M \cdot \text{max})$. Though the asymptotic complexity of MinSum is worse than the VCS based on SNARKs, it only involves modular multiplications and its concrete efficiency is much better for reasonable M and max [48]. In particular, in our scheme for planar graphs, $M = O(\sqrt{n})$ and $\text{max} = O(n)$, where n is the size of the graph.

4.3 VC Scheme for Planar Graphs

We now present our construction for verifying shortest paths in planar graphs using the planar separator tree data structure and our VCS.

Setup. At setup, given a graph G we run $\mathcal{V}$.genkey for a VCS $\mathcal{V}$, build the planar separator tree $\mathcal{T}$ for G, and compute the shortest-path vectors $S_{u\mathbf{p}}$ for all $u \in G$ and $\mathbf{p} \in path(u)$. Then, we commit to the shortest-path vectors by computing their digests using the vector commitment scheme $\mathcal{V}$, i.e., we compute

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

For clarity of presentation, we assume 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) $path(u)$ for all $u \in G$; and (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 MAC (for private verification) or digital signature (for public verification) of each object above. (Indeed, our implementation employs this strategy using HMAC.)

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 vertex v and let $\mathbf{u}$ be the separator-tree node containing vertex u. We distinguish two cases, depending on the location of u and v in the separator tree:

```
Algorithm $(\mathsf{ek}_G, \mathsf{vk}_G) \leftarrow \mathsf{genkey}(1^k, G)$
    • Compute planar separator tree $\mathcal{T}$.
    • $\mathcal{V}.\mathsf{pk} \leftarrow \mathcal{V}.\mathsf{genkey}(1^k, M)$, where $M = O(\sqrt{n})$ is the maximum number of vertices in planar
      separators.
    • 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 $\mathsf{ek}_G$ contains keys $\mathcal{V}.\mathsf{pk}$, 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 $\mathsf{vk}_G$ contains $\mathcal{V}.\mathsf{pk}$, 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
          $(s_{uv}, \pi_{uv}) \leftarrow \mathcal{V}.\mathsf{Membership}(v, \mathbf{S}_{u\mathbf{v}}, \mathcal{V}.\mathsf{pk})$, 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 $\pi_i \leftarrow \mathcal{V}.\mathsf{MinSum}(\mathbf{S}_{u\mathbf{t}_i}, \mathbf{S}_{v\mathbf{t}_i}, \mathcal{V}.\mathsf{pk})$.
Output $\pi_{uv}$ or $\pi_i$ $(i = 1, \ldots, k)$ as $\pi$ 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
          $1 \leftarrow \mathcal{V}.\mathsf{verify\_Membership}(v, s_{uv}, \pi_{uv}, d_{u\mathbf{v}}, \mathsf{pk})$ .
    • Otherwise, let $\{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\} = path(u) \bigcap path(v)$.
      For $i = 1, \ldots, k$ check that
          $1 \leftarrow \mathcal{V}.\mathsf{verify\_MinSum}(d_{u\mathbf{t}_i}, d_{v\mathbf{t}_i}, min_i, \pi_i, \mathcal{V}.\mathsf{pk})$.
    • Check that path $p$ in $G$ has length $\min\{min_1, \ldots, min_k\}$.
    • If all checks succeed output 1, else output 0.
```

Fig. 3. A VC scheme for shortest paths in a planar graph G.

Case 1. If $\mathbf{v}$ is 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 $\mathrm{S}_{u\mathbf{v}}$. Therefore it suffices for the prover to return a proof for the element of $\mathrm{S}_{u\mathbf{v}}$ that corresponds to the shortest path s_{uv}. 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 $\mathrm{S}_{u\mathbf{t}_i} + \mathrm{S}_{v\mathbf{t}_i}$, 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 VCS proof π_i by calling $\mathcal{V}.\mathsf{MinSum}(\mathrm{S}_{u\mathbf{t}_i}, \mathrm{S}_{v\mathbf{t}_i}, \mathcal{V}.\mathsf{pk})$. This proof is used to prove that min_i is the minimum of vector $\mathrm{S}_{u\mathbf{t}_i} + \mathrm{S}_{v\mathbf{t}_i}$ (and therefore a potential length for the shortest path from u to v). After all VCS 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 3.

Asymptotic complexity and security. Let $m = O(n)$ be the number of edges 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 [22], this cost is $O(m^{3/2})$. Note that this is a one-time cost.

As far as algorithm compute is concerned, the cost is dominated by computing one proof using the vector commitment scheme. When initiated using SNARKs, each SNARK proof takes $O(\sqrt{m} \log m)$ time (since the description of the language we are encoding has size $O(\sqrt{m})$) and the total worst-case cost of computing the proof is $O(\sqrt{m} \log m)$. When initiated using our customized VCS, the total worst-case cost of computing the proof is $O(m)$.

Finally, the size of the proof is $O(\log m + |p|)$, since $O(\log m)$ VCS 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 the VCS. We summarize the above in the following theorem:

Theorem 4.2. *Let G be a planar graph with $m = O(n)$ edges. Our VC scheme for shortest paths in G has (i) $O(m\sqrt{m})$ preprocessing time; (ii) $O(\sqrt{m} \log m)$ prover time using the first VCS and $O(m)$ prover time using the second VCS; (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 describe the implementations of six different VC schemes for shortest paths and one VC scheme for maximum flow. Parts of our implementation (Libsnark BFS) use Libsnark [7], which 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 that is used to produce the evaluation and verification keys. The parts that did not require Libsnark (e.g., building the planar separator tree) were implemented in C++.

Our implementation uses MACs (specifically, HMAC from **openssl** [1]) in place of digital signatures, and so all our schemes achieve only private verifiability. To make our schemes publicly verifiable without significantly affecting performance, we could use standard techniques combining Merkle hash trees with a signature on the root. To ensure a fair comparison with Libsnark, in our experiments we "turn off" public verifiability and zero knowledge in the latter.

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 all shortest paths, which has $O(nm)$ complexity (instead of $O(n^3)$). We then compute an HMAC of each shortest path.

Libsnark BFS scheme. Our second attempt was to execute the BFS code directly in Libsnark.[4] We implemented the algorithm shown in Figure 4 and compiled it into a circuit using the compiler provided by Pinocchio [38]. The two most important limitations, inherent to using a circuit representation, are that (1) array accesses on nonconstant addresses must be implemented by a circuit of linear size; (2) the bound of a loop must be predefined.

Figure 4 shows Boolean-circuit pseudocode for an implementation of breadth-first-search (BFS) to find the shortest path in a unit-weight graph. Note that Lines 9–11 of the pseudocode replace random accesses, and this increases the complexity from $O(m + n)$ to $O(mn)$. To make the algorithm more circuit friendly, we simulated the BFS queue with an array Q of fixed size (we cannot

[4]A recent optimization by Reference [28] is enabled.

```
LIBSNARK_BFS($G$,$s$)
    $head$ = first position in $Q$;
    $tail$ = first position in $Q$;
    $Q[tail] = s$;
    $tail$++;
    for each node $x \in V$
        if $x = s$
            $x.parent = -1$;
    for $round$ = 1 to $|V|$
        for each node $x \in V$
            if $x = Q[head]$
                for each node $v \in Adj[x]$
                    if $v.visited = 0$
                        $v.visited = 1$;
                        $v.parent = Q[head]$;
                        $Q[tail] = v$;
                        $tail$++;
        $head$++;
```

Fig. 4. The BFS pseudocode we implemented in Libsnark. Array Q has $|V|$ positions and simulates the queue in BFS.

implement dynamic data structures in a circuit). 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 vertices in 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 that algorithm uses a more complicated priority queue.

Buffet **BFS scheme.** We implemented the BFS algorithm in a subset of C and compiled it to a RAM-based VC instance using the frontend compiler of Buffet. We then executed the backend[5] of Buffet on that instance.

Certifying algorithm scheme. We implemented the certifying algorithm with an efficient circuit of $O(m)$ size, as explained in Sections 3.3 and 3.4. We compute the shortest-path vector **S** (which the server provides as input to Relation 1) using the BFS implementation in the LEDA library (version 6.4) [30]. We note here that, contrary to the BFS algorithm, the certifying approach can be naturally applied for weighted graphs—see Relation 1. As in Reference [49], our implementation of certifying algorithms and the first planar-separator scheme uses Pinocchio [29, 38], an old implementation of SNARKs. Integrating optimized SNARKs [7, 28] could lead to an improvement of up to an order of magnitude but is left as future work.

While implementing the condition of Equation (1) in Pinocchio, we observed that comparison operations ($\leq, \geq$) are much more expensive than addition and multiplication. Therefore, for unit-weight graphs we replaced the second condition of Equation (1) by an equivalent equality constraint with an additional input a_{uv}, i.e.,

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

[5]The backend of Buffet can also be re-factored to use an optimized version [28], which would improve the setup time, prover time, and server storage by approximately 30%, the proof size by approximately 50%, and the verification time by about 67%. This optimization is not included in the current implementation of Buffet.

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 checking the domain of the additional input may slow down the performance.

Planar-separator scheme. We implemented the VC scheme for planar graphs as described in Figure 3. To build the planar separator tree, we first triangulate the input graph using the LEDA library [30]; the triangulated graph is then input recursively into the recent planar separator implementation by Fox-Epstein et al. [24]. The digests of the shortest-path vectors are computed using our implementations of the vector commitment schemes described earlier. We compute an HMAC of these digests, along with precomputed distances and paths, and then outsource them. For our first VCS, we use SNARKs based on Pinocchio. For our second VCS, we use the ate-paring library [2] on a 254-bit curve for the bilinear group.

Verifying max-flow. Verification of maximum flow is implemented as described in Section 3.4.

6 EVALUATION

We now evaluate the shortest-path verification using (i) the strawman scheme, (ii) the Libsnark BFS scheme, (iii) the Buffet BFS scheme, (iv) the certifying-algorithm scheme, and (v) the planar separator scheme with two different vector commitment schemes. We also present experiments for maximum-flow verification. We do not present results for longest-path verification, since this can be done using essentially the same certifying algorithm as for shortest paths (with a change in direction of the inequality).

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

We present results for the *preprocessing time*, *proof-computation time*, *verification time*, and *server storage*. All schemes were run on the same randomly generated planar, undirected graph (we use the LEDA function `random_planar_graph` for this) with unit weights. We collected 10 runs for each data point, and we report the average in Figures 5, 6, 7, and 8. In all these figures, estimated data points (due to exceeding memory/time) are marked as lightly shaded bars. We experiment on planar graphs with $n = 10, 10^2, \ldots, 10^5$ vertices, where the number of edges is at most $3(n-2)$ (due to planarity). Our planar scheme was the only one to successfully execute on a graph of 200,000 nodes.

Preprocessing time. Figure 5 compares preprocessing time across the schemes. The results show that the certifying algorithm outperforms Libsnark and Buffet BFS by orders of magnitude. Specifically, the optimized certifying algorithm runs approximately 6,300 times faster than Libsnark, and 140 times faster than Buffet, on a graph with 10,000 nodes.

In our experiments, the Libsnark and Buffet BFS implementations are so inefficient that it takes too long to get a result even for graphs with more than 100 nodes. Thus, the results for larger graphs are estimated by extrapolating from the results on small graphs.

Surprisingly, although the complexity of preprocessing time in the planar separator schemes is $O(n^{3/2})$, that scheme 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 first planar separator scheme outperforms the certifying algorithm scheme in the preprocessing time by 29× on a graph with 10,000 nodes and the second planar separator scheme is faster by 453×. Both schemes can scale to a graph with up to 200,000 nodes.

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 second planar separator scheme on a graph with 100 nodes and is 16× worse on a graph with 1,000 nodes. We had to estimate the preprocessing time of the strawman approach on a graph of 100,000 nodes,

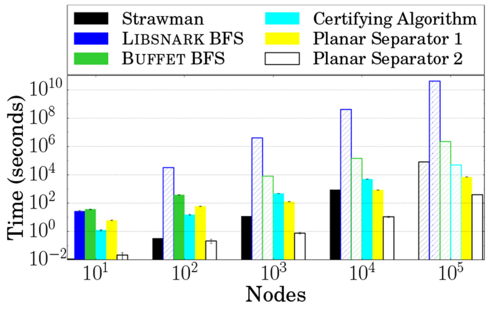

Fig. 5. Preprocessing time. We were only able to execute BFS for graphs of up to 50 nodes. All other points (shaded bars) are estimated.

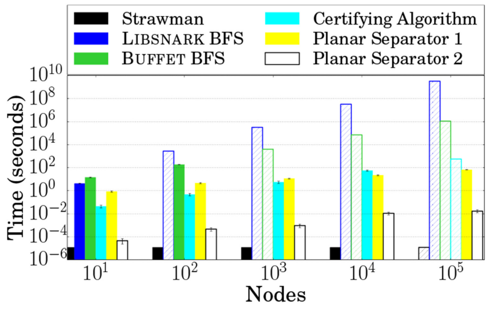

Fig. 6. Proof-computation time. We were only able to execute the certifying algorithm for up to 10,000 nodes.

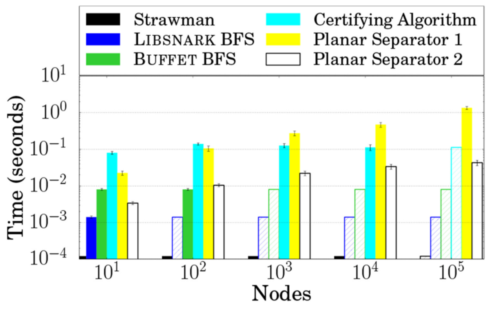

Fig. 7. Verification time.

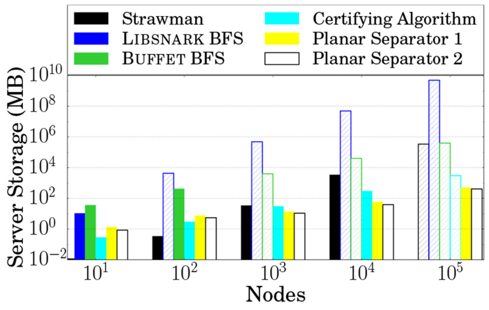

Fig. 8. Server storage.

since storing an all-pairs-shortest-path matrix of size 100,000×100,000 requires too much memory (we ran BFS 100,000 times and compute $100{,}000^2$ HMACs to estimate the cost).

Proof-computation time. Figure 6 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 approaches and that the planar separator approach outperforms both BFS and the certifying algorithm dramatically. In particular, proof-computation time of the first planar separator scheme has a speedup of more than $1.4 \cdot 10^6\times$ compared to Libsnark and 180× compared to Buffet, on a 10,000-node graph. Our second planar separator scheme, though asymptotically worse than the first one, further improves the performances by more than 4000× on the first scheme.

We note here that in the case of the planar separator schemes, we report *worst case* results in Figure 6. 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 2 shows the comparison between the worst and the best case in our first planar separator scheme. 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 schemes reduce 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

Table 2. Worst and Best Cases for Planar Proof-computation Time

n	Planar Separator Scheme 1			
	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 3. Proof-computation Time (seconds)

	Strawman	Libsnark BFS	Buffet BFS	Certifying Algorithm	Planar Separator 1	Planar Separator 2
10	0.00001	4.24	14.41	0.044	0.85	0.000045
100	0.00001	2,700*	186.56	0.48	4.62	0.00047
1,000	0.00001	320,000*	4,000*	5.50	11.43	0.00094
10,000	0.00001	32,000,000*	70,000*	56.02	22.22	0.011
100,000	0.00001	3,300,000,000*	1,100,000*	560*	66.37	0.017
200,000	0.00001	13,000,000,000*	2,500,000*	1120*	93.85	0.022

expensive in existing work. For example, our work shows that graph processing can scale up to 200,000 nodes, since it takes only tens of milliseconds as shown in Figure 6 to produce a proof for such large graphs. The exact numbers of our proof-computation time can be found in Table 3.

Verification time. Figure 7 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 schemes. Still, as shown in Figure 7, the verification time of the planar separator schemes does not grow that much. It requires less than 2s on a graph with 100,000 nodes for the first scheme, and only around 0.04s for the second scheme. 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 insignificant.

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 schemes, the server stores the Libsnark and Buffet circuits and evaluation keys. 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 schemes the graph is embedded in the circuits) 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 Libsnark 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 schemes, the separator tree is part of the server storage in addition to that of the certifying algorithm.

Figure 8 shows the comparison of the server storage. Since the Libsnark and Buffet evaluation key sizes are proportional to the size of the circuits and the certifying algorithm has a more efficient circuit implementation, the server storage for the certifying algorithm is much smaller. In particular, the server storage is reduced by 145× compared to Libsnark and 133× compared to Buffet, on a graph with 100 nodes and the gap grows on larger graphs.

Table 4. Proof Size in KB for $|p| = 10$

n	Strawman	LIBSNARK BFS	BUFFET BFS	Certifying Algorithm	Planar Separator 1	Planar Separator 2
100	0.352	0.127	0.288	0.608	1.76	0.955
1,000	0.352	0.127	0.288	0.608	3.2	1.59
10,000	0.352	0.127	0.288	0.608	4.64	2.25

Table 5. Maximum-flow Scheme Proof-computation Time

n	LIBSNARK Implementation (s)	BUFFET Implementation (s)	Maximum-flow Scheme (s)
10	>4.24	>14.414	0.701
100	>2,700	>186.56	31.15
1,000	>320,000	>4,000	1,003

The planar separator schemes have the least storage requirements. As shown in Figure 8, although the server storage is around 4× larger on a small graph with 10 nodes compared to the certifying algorithm, it is reduced by 3× or more for larger graphs with more than 1,000 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 schemes outperform the strawman scheme by 3× on an 1000-node graph and by 700× on an 100,000-node graph. Here the storage of our second planar separator scheme is only slightly better than the first one, because the storage is dominated by the planar separator tree structure, which is the same for both schemes.

Proof size. One drawback of our approaches compared to the BFS approach is the proof size. The proof size of the LIBSNARK BFS and the BUFFET BFS scheme is always a constant (127 and 288 bytes)—see Reference [38]. 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 $|p| \times 32 + c \times \log n$ bytes, where c is the size of each VCS proof (288 bytes for the first scheme and 127 bytes for the second scheme). 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 4 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.

Evaluation of the maximum-flow scheme. Table 5 compares the prover time between implementing the maximum-flow algorithm directly using LIBSNARK and using the certifying-algorithm approach. Actually, maximum flow algorithms are much more complicated than BFS and we could not implement them directly on LIBSNARK and BUFFET easily. However, we observed that the Edmonds-Karp algorithm [18] 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 at least 320× and 4×, respectively, on a graph with 1,000 nodes.

Discussion. Our experiments show that the certifying-algorithm approach can scale to 10,000-node graphs for verifying shortest paths and 1,000-node graphs for verifying maximum flow. In contrast, generic VC (namely, LIBSNARK) runs out of memory on larger graphs. To improve the scalability of these approaches, we need to rely on generic VC schemes with less memory usage and faster prover time, such as those in References [46, 47], and exploring the performance of these schemes for verifying graph queries is left as future work.

Our verifiable shortest-path schemes for planar graphs can scale to graphs with 200,000 nodes. The preprocessing in these schemes is slow and starts to run out of memory on larger graphs, as the complexity of the preprocessing phase is $O(m^{3/2})$. To scale to larger graphs, we need to find more efficient algorithms for finding planar separators and for storing the planar separator tree efficiently on disk.

APPENDIX

A COMPLEXITY ANALYSIS OF THE PLANAR SEPARATOR TREE

Setup. Suppose the time to construct the planar separator tree structure is $T(n)$. By the planar separator theorem, the planar separator of the first level contains $O(\sqrt{n})$ vertices. The complexity to compute the distances between all other vertices to every vertex in this planar separator is thus $O(n^{3/2})$. Therefore, $T(n) = T(\varepsilon n) + T((1-\varepsilon)n) + O(n^{3/2})$, where $\frac{1}{3} \leq \varepsilon \leq \frac{2}{3}$. Therefore, $T(n) = O(n^{3/2})$ and the derivation is as follows:

$$\begin{aligned}
T(n) &= T(\varepsilon n) + T((1-\varepsilon)n) + O(n^{3/2}) \\
&= T(\varepsilon^2 n) + 2T(\varepsilon(1-\varepsilon)n) + T((1-\varepsilon)^2 n) + (\varepsilon^{3/2} + (1-\varepsilon)^{3/2})O(n^{3/2}) + O(n^{3/2}) \\
&= T(\varepsilon^3 n) + 3T(\varepsilon^2(1-\varepsilon)n) + 3T(\varepsilon(1-\varepsilon)^2 n) + T((1-\varepsilon)^3 n) + (\varepsilon^{3/2} + (1-\varepsilon)^{3/2})^2 O(n^{3/2}) \\
&\quad + (\varepsilon^{3/2} + (1-\varepsilon)^{3/2})O(n^{3/2}) + O(n^{3/2}) \\
&= \ldots \\
&= O(n^{3/2}) + \rho O(n^{3/2}) + \rho^2 O(n^{3/2}) + \cdots \\
&= O(n^{3/2}),
\end{aligned}$$

where $\rho = \varepsilon^{3/2} + (1-\varepsilon)^{3/2} < 1$. The space required to store the data structure follows the same analysis and is also $O(n^{3/2})$.

Query. In the worst case, the two vertices u, v in the shortest path query are in the two siblings of the planar separator tree on the leaf level. To answer the query, one needs to perform one vector addition and one minimum for every planar separator node on **path**(u) (or **path**(v)). The total complexity is bounded by $O(\sqrt{n}) + O(\sqrt{\varepsilon n}) + O(\sqrt{\varepsilon^2 n}) + \cdots = O(\sqrt{n})$, where $\frac{1}{3} \leq \varepsilon \leq \frac{2}{3}$.

ACKNOWLEDGMENT

We thank Roberto Tamassia for useful discussions.

REFERENCES

[1] 2016. openSSL library. Retrieved from https://www.openssl.org/.
[2] 2017. Ate pairing. Retrievved from https://github.com/herumi/ate-pairing.
[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'17)*.
[4] Aris Anagnostopoulos, Michael Goodrich, and Roberto Tamassia. 2001. Persistent authenticated dictionaries and their applications. In *Proceedings of 4th International Conference on Information Security (ISC'01)*. 379–393.
[5] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. 2017. Computational integrity with a public

random string from quasi-linear PCPs. In *Proceedings of the Annual Conference on Advances in Cryptology (EUROCRYPT'17)*. 551–579.

[6] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the Annual Conference on Advances in Cryptology (CRYPTO'13)*. Springer, 90–108.

[7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the USENIX Security Symposium*. 781–796.

[8] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2017. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica* 79, 4 (2017), 1102–1160.

[9] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 326–349.

[10] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*. 111–120.

[11] Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. 2014. On the existence of extractable one-way functions. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'14)*. 505–514.

[12] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. 2013. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography*. Springer, 315–333.

[13] Elette Boyle and Rafael Pass. 2015. Limits of extractability assumptions with distributional auxiliary input. In *Proceedings of the Annual Conference on Advances in Cryptology (ASIACRYPT'15)*. 236–261.

[14] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. 2013. Verifying computations with state. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 341–357.

[15] Dario Catalano and Dario Fiore. 2013. Vector commitments and their applications. In *Public Key Cryptography*. 55–72.

[16] Alessandro Chiesa, Eran Tromer, and Madars Virza. 2015. Cluster computing in zero knowledge. In *Proceedings of the Annual Conference on Advances in Cryptology (EUROPCRYPT'15)*. 371–403.

[17] Kai-Min Chung, Yael Tauman Kalai, and Salil P. Vadhan. 2010. Improved delegation of computation using fully homomorphic encryption. In *Proceedings of the Annual Conference on Advances in Cryptology (CRYPTO'10)*. 483–501.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[19] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*. 253–270.

[20] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. 2014. Square span programs with applications to succinct NIZK arguments. In *Proceedings of the Annual Conference on Advances in Cryptology (ASIACRYPT'14)*. 532–550.

[21] DIMACS. 2006. 9th DIMACS Implementation Challenge—Shortest Paths. Retrieved from http://www.dis.uniroma1.it/challenge9/.

[22] Jittat Fakcharoenphol and Satish Rao. 2006. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.* 72, 5 (2006), 868–889.

[23] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. 2016. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*. 1304–1316.

[24] Eli Fox-Epstein, Shay Mozes, Phitchaya Mangpo Phothilimthana, and Christian Sommer. 2016. Short and simple cycle separators in planar graphs. *J. Exp. Algor.* 21, 2 (2016), 2–2.

[25] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the Annual Conference on Advances in Cryptology (CRYPTO'10)*, 465–482.

[26] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the Annual Conference on Advances in Cryptology (EUROCRYPT'13)*. 626–645.

[27] Michael T. Goodrich, Roberto Tamassia, and Nikos Triandopoulos. 2011. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica* 60, 3 (2011), 505–552.

[28] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Proceedings of the Annual Conference on Advances in Cryptology (EUROCRPYPT'16)*. 305–326.

[29] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. 2014. TRUESET: Faster verifiable set computations. In *Proceedings of the USENIX Security Symposium 2014*. 765–780.

[30] LEDA. 2017. LEDA library. Retrieved from http://www.algorithmic-solutions.com/leda/index.htm.

[31] Helger Lipmaa. 2013. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *Proceedings of the Annual Conference on Advances in Cryptology (ASIACRYPT'13)*. 41–60.

[32] Richard J. Lipton and Robert Endre Tarjan. 1979. A separator theorem for planar graphs. *SIAM J. Appl. Math.* 36, 2 (1979), 177–189.

[33] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. 2011. Certifying algorithms. *Comput. Sci. Rev.* 5, 2 (2011), 119–161.

[34] Silvio Micali. 2000. Computationally sound proofs. *SIAM J. Comput.* 30, 4 (2000), 1253–1298.

[35] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. 2013. Streaming authenticated data structures. In *Proceedings of the Annual Conference on Advances in Cryptology (EUROCRYPT'13)*. 353–370.

[36] Charalampos Papamanthou and Roberto Tamassia. 2007. Time and space efficient algorithms for two-party authenticated data structures. In *Proceedings of 9th International Conference Information and Communications Security (ICICS'07)*. 1–15.

[37] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2011. Optimal verification of operations on dynamic sets. In *Proceedings of the Annual Conference on Advances in Cryptology (CRYPTO'11)*. 91–110.

[38] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*. 238–252.

[39] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. 2013. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 71–84.

[40] Srinath T. V. Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. 2012. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the Network and Distributed System Security Symposium*, Vol. 1. 17.

[41] Roberto Tamassia. 2003. Authenticated data structures. In *Proceedings of European Symposium on Algorithms*, Vol. 2832. 2–5.

[42] Roberto Tamassia and Nikos Triandopoulos. 2010. Certification and authentication of data structures. In *Proceedings of the 4th Alberto Mendelzon International Workshop on Foundations of Data Management*.

[43] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. 2013. A hybrid architecture for interactive verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*. 223–237.

[44] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium*.

[45] Man Lung Yiu, Yimin Lin, and Kyriakos Mouratidis. 2010. Efficient verification of shortest path search via authenticated hints. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'10)*. 237–248.

[46] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'17)*. 863–880.

[47] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. 2018. vRAM: Faster verifiable RAM with program-independent preprocessing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'18)*. 203–220.

[48] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2017. An expressive (zero-knowledge) set accumulator. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P'17)*. 158–173.

[49] Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. 2014. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*. 856–867.

Received January 2018; accepted June 2018