Stateful Multi-Client Verifiable Computation

Christian Cachin[1], Esha Ghosh[2], Dimitrios Papadopoulos[3], and
Björn Tackmann[1]

[1] IBM Research – Zurich, Switzerland, `{cca,bta}@zurich.ibm.com`
[2] Brown University, USA, `esha_ghosh@brown.edu`
[3] Hong Kong University of Science and Technology, Hong Kong,
`dipapado@cse.ust.hk`

Abstract. This paper develops a cryptographic protocol for outsourcing arbitrary stateful computation among multiple clients to an untrusted server, while guaranteeing integrity of the data. The clients communicate only with the server and store only a short authenticator to ensure that the server does not cheat.

Our contribution is two-fold. First, we extend the recent hash&prove scheme of Fiore et al. (CCS 2016) to *stateful* computations that support arbitrary updates by the untrusted server, in a way that can be verified by the clients. We use this scheme to *generically* instantiate authenticated data types. Second, we describe a protocol for multi-client verifiable computation based on an authenticated data type, and prove that it achieves a computational version of *fork linearizability.* This is the strongest guarantee that can be achieved in the setting where clients do not communicate directly; it ensures correctness and consistency of outputs seen by the clients individually.

1 Introduction

Cloud services are nowadays widely used for outsourcing data and computation because of their competitive pricing and immediate availability on demand. They also allow for online collaboration by having multiple clients operate on the same data; such online services exist for, e.g., shared file storage, standard office applications, or software solutions for specific domains. For authenticity, confidentiality, and integrity of the data, however, the clients have to fully trust the cloud service providers, which can access and modify the raw data without the clients' consent or notice.

Cryptographic schemes and protocols have been developed for various specific tasks and security goals that arise in the context of cloud services. A (necessarily partial) list of examples includes storage auditing for outsourcing large files [3, 33], verifiable computation for outsourcing computational tasks [23, 24, 47, 50], private information retrieval (PIR) or oblivious RAM for accessing remote data without leaking access patterns [26, 17], and many more.

The scenario we are concerned with in this paper involves multiple clients that mutually trust each other and collaborate through an untrusted server. The protocol emulates multi-client access to an abstract data type F. On input an operation o, and current state s, the protocol computes $(s', r) \leftarrow F(s, o)$ to generate an updated state s' and an output r. The role of a client C_v is to invoke the operation o and obtain the response r; the purpose of the server is to store the state of F and to perform the computation. As an example, let F be defined for a set of elements where o can be adding or deleting an element to the set. The state of the functionality will consist of the entire set. We assume the availability of a public-key infrastructure, where each client registers its public key of a signature scheme. Clients communicate only with the server; no direct communication between the clients occurs. The goal of our protocol is to guarantee the integrity and freshness of responses, in the scenario where the server is untrusted and may be acting maliciously.

Related work. The described problem has received considerable attention from the viewpoint of distributed systems, starting with protocols for securing *untrusted storage* [38]. In the setting without communication among clients, the server may always perform a *forking attack* and omit the effects of operations by some clients in the communication with other clients. Clients cannot detect this attack unless they exchange information about the protocol progress or rely on synchronized clocks; the best achievable consistency guarantee has been called *fork linearizability* by Mazières and Shasha [38] and has been widely researched [14, 12, 35] and applied to actual systems [34, 14, 51, 13, 10]. Early works [34, 14] focused on simple read/write accesses to a storage service. More recent protocols such as BST [51] and COP [13] allow for emulating arbitrary data types and for exploiting the commutativity of certain operations under concurrent access. However, they require that the entire state be stored and the operations be computed on the client. ACOP [13] and VICOS [10] describe at a high level how to outsource both the state and the computation in a generic way, but neither work comes with an appropriate cryptographic security model nor are their protocols proven secure.

The purpose of an *authenticated data type* (ADT; often also referred to as authenticated data structure) is to allow a client to outsource data, and the computation on it, to a server, while guaranteeing the integrity of the data. In a nutshell, while the server stores the data, the client holds a small *authenticator* (sometimes called *digest*) that relates to it. Operations on the data are performed by the server, and for each operation the server computes a proof that, together with the authenticator, allows the client to check that the server performed the operation correctly. ADTs originated as a generalization of Merkle trees [39]; an excellent survey of early work is given by Tamassia [48]. Many instantiations of ADTs for specific data types have been described in the literature. There exist schemes for such diverse types as sets [46, 15], dictionaries [42, 2, 28], range trees [36], graphs [29], skip lists [27, 28], B-trees [41], or hash tables [45]. Recent work has targeted seamlessly integrating ADTs for general search problems into programming languages [40].

Non-interactive verifiable computation has been introduced as a concept to outsource computational tasks to untrusted workers [23], where it is crucial that the verification of the correctness is more efficient than solving the computational tasks. Verifiable computation schemes that can achieve this for arbitrary functionalities have been suggested [23, 24, 47, 18] and are closely related to work on SNARKs (e.g., [8]). These works have the disadvantage, however, that the client verifying the proof needs to know (and compute on) the complete input to the computation. This can always be avoided by having the client first hash its input and then outsource it storing only the hash locally. The subsequent verifiable computation protocol must then ensure not only the correctness of the computation but also that the input used matches the pre-image of the hash that the client is storing (which increases the concrete overhead), an approach that has been adopted in several works [11, 49, 18, 20]. In this work, we build on the latest in this line of works, the hash&prove scheme of Fiore et al. [20], by a mechanism that allows for stateful computation in which an *untrusted* party can update the state in a verifiable manner, and that can handle multiple clients. An alternative (restricted but in several cases more efficient) approach for verifiable computation focuses on specific computation tasks, such as polynomial evaluation [9, 5], database queries [52, 43], or matrix multiplication [21], aiming to gain efficiency by sacrificing generality.

All above works target a setting where a *single* client interacts with the server, i.e., they do not support *multiple* clients collaborating on outsourced data, as is the case here. The only existing works that capture multi-client verifiable computation are by Choi et al. [16] and Gordon et al. [30]; however, they only accommodate “one-off” stateless computations. More concretely, all clients send their inputs to the server once, the latter evaluates a function on the joint data and returns the output. In this work, we are interested in a scenario where the data is permanently outsourced to the server and updated upon request by the clients. Related recent work that targets multi-client authenticated access to computation results on data is on so-called multi-key homomorphic authenticators [22], which can support circuits of (bounded) polynomial depth. In contrast to our work, however, it does not (easily) support updating the state.

Contributions in this work. Our first contribution is a new and general security definition of a two-party ADT: The server manages the state of the computation, performs updates and queries; the client invokes operations and receives results. Our client only stores a short authenticator, and the server proves to the client that it performed the operation, i.e., compute the output and possibly a new authenticator, correctly. This significantly deviates from the standard three-party ADT definition (e.g. [48, 46]) where there is a separation between data owner and client(s). The former needs to store the entire data in order to perform updates and publish the new authenticator in a *trusted* manner, while the latter one(s) may issue read-only queries to the untrusted server. Our definition allows the untrusted server to perform updates such that the resulting authenticator can be verified for its correctness, eliminating the need to have a trusted party store the entire data. The definition generalizes existing ones for two-party ADTs [44, 25] that only support deterministic schemes.

We then provide a *general-purpose* instantiation of an ADT, based on verifiable computation, by extending the work of Fiore et al. [20]. Our instantiation can capture *arbitrary* stateful deterministic computation, and the client stores only a short authenticator which consists of two elements in a bilinear group. The subsequent parts of the paper are independent of the technical details discussed here, so this Section 4 can be skipped by readers more interested in the Byzantine emulation protocol.

We also devise *computational* security definitions that model the distributed-systems concepts of *linearizability* and *fork linearizability* [38] in the code-based game-playing framework of Bellare and Rogaway [7]. This allows us to prove the security of our protocol in a computational model by reducing from the security of digital signatures and ADTs—all previous work on fork linearizability used idealizations of the cryptographic schemes.

Finally, we describe a “lock-step” protocol to satisfy the computational fork linearizability notion, adapted from SUNDR [38] and Cachin et al. [14]. The protocol guarantees consistent—in the sense of fork-linearizability—multi-client access to a data type. The protocol is based on our definition of ADTs; if instantiated by the general-purpose ADT construction we provide, it is a protocol for outsourcing any stateful (deterministic) computation with shared access in a multi-client setting.

2 Preliminaries

We use the standard notation for the sets of integers $\mathbb{Z}$, positive integers $\mathbb{N}$, and integers $\mathbb{Z}_p$ modulo a number $p \in \mathbb{N}$. We let ϵ denote the empty string. If Z is a string then $|Z|$ denotes its length, and $\circ$ is an operation to concatenate two strings. We consider lists of items, where $[\,]$ denotes the empty list, $L[i]$ means accessing the i-th element of the list L, and $L \leftarrow L \circ x$ means storing a new element x in L by appending it to the end of the list. If $\mathcal{X}$ is a finite set, we let $x \leftarrow\!\!\$\; \mathcal{X}$ denote picking an element of $\mathcal{X}$ uniformly at random and assigning it to x. Algorithms may be randomized unless otherwise indicated. Running time is worst case. If A is an algorithm, we let $y \leftarrow A(x_1, \ldots; r)$ denote running A with random coins r on inputs $x_1, \ldots$ and assigning the output to y. We let $y \leftarrow\!\!\$\; A(x_1, \ldots)$ be the result of picking r at random and letting $y \leftarrow A(x_1, \ldots; r)$.

We use the code-based game-playing framework of Bellare and Rogaway [7]. (See Figure 1 for an example.) By $\Pr[\mathbf{G}]$ we denote the probability that the execution of game $\mathbf{G}$ results in the game returning TRUE. In games, integer variables, set and list variables, and boolean variables are assumed initialized, respectively, to 0, the empty set and list, and FALSE. Our security statements are *concrete* (as opposed to *asymptotic*), that is, we bound the adversary advantage by a concrete upper bound that is stated in terms of a reduction from an underlying problem, instead of merely showing the advantage to be *negligible*. (Our concrete security statements naturally imply asymptotic ones, such as about polynomial-time adversaries and negligible probabilities, should those be of interest.)

System model. The security definition for our protocol is based on well-established notions from the distributed-systems literature. In order to make *cryptographic* security statements and not resort to modeling all cryptography as ideal, we provide a computational definition that captures the same intuition. We recall the distributed-systems definitions, and try to give intuition, below.

Recall that our goal is to enable multiple clients $C_1, \ldots, C_u$, with $u \in \mathbb{N}$, to evaluate an abstract *data type* $F : (s, o) \mapsto (s', r)$, where $s, s' \in S$ describe the global state of F, $o \in O$ is an *input* of a client, and $r \in A$ is the corresponding *output* or *response*. The clients can provide inputs to F in an arbitrary order. Each execution defines a *history* σ, which is a sequence of input events (C_v, o) and output events (C_v, r) (for simplicity, we assume $O \cap A = \emptyset$). An operation directly corresponds to an input/output event pair and vice versa, and an operation is *complete* in a history σ if σ contains an output event matching the input event.

In a *sequential* history, the output event of each operation directly follows the corresponding input event. Moreover, an operation o *precedes* an operation o' in a history σ if the *output* event of o occurs before the *input* event of o' in σ. Another history σ' *preserves* the (real-time) order of σ if all operations of σ' occur in σ as well and their precedence relation in σ is also satisfied in σ'.

The goal of a protocol is to *emulate* F. The clients running the protocol only observe their own input and output events. The security of a protocol is defined in terms of how close the histories it produces are to histories that would have been produced with invocations of an ideal shared F.

Linearizability. A history σ is *linearizable with respect to a type* F [32] if and only if there exists a sequential permutation $\pi(\sigma)$ of σ such that

- $\pi(\sigma)$ preserves the (real-time) order of σ; and
- the operations of $\pi(\sigma)$ satisfy the sequential specification of F.

Satisfying the sequential specification of F means that if F starts in a specified initial state s_0, and all operations are performed sequentially as determined by $\pi(\sigma) = o_1, o_2, \ldots$, then with $(s_j, r_j) \leftarrow F(s_{j-1}, o_j)$, the output event corresponding to o_j contains output r_j.

Linearizability is a strong guarantee as it specifies that the history σ could have been observed by interacting with the ideal F, by only (possibly) exchanging the order of operations which were active concurrently. Unfortunately, as described in the introduction, linearizability cannot be achieved in the setting we are interested in.

Fork linearizability. A history σ is called *fork-linearizable with respect to a type* F if and only if, for each client C_v, there exists a subsequence σ_v of σ consisting only of complete operations and a sequential permutation $\pi_v(\sigma_v)$ of σ_v such that:

- All complete operations in σ occurring at client C_v are contained in σ_v, and
- $\pi_v(\sigma_v)$ preserves the real-time order of σ_v, and
- the operations of $\pi_v(\sigma_v)$ satisfy the sequential specification of F, and

– for every $o \in \pi_v(\sigma_v) \cap \pi_{v'}(\sigma_{v'})$, the sequence of events that precede o in $\pi_v(\sigma_v)$ is the same as the sequence of events that precede o in $\pi_{v'}(\sigma_{v'})$.

Fork linearizability is weaker than linearizability in that it requires consistency with F only with respect to permutations of sub-sequences of the history. This models the weaker guarantee that is achieved relative to a dishonest server that partitions the set of clients and creates independent *forks* of the computation in each partition. Intuitively, fork linearizability guarantees that the computation is still consistent within each partition individually, but does *not* guarantee that each client observes the operations of all other clients. Once two clients have been forked, however, they will remain forked forever—it is impossible for the server to make an operation of one client occurring after the fork visible to a client in the other fork. Fork linearizability is the strongest security guarantee that can be achieved in the setting where the clients cannot communicate among each other and the server may be dishonest [38].

Abortable services. When operations of F cannot be served immediately, a protocol may decide to either block or abort. Aborting and giving the client a chance to retry the operation at his own rate often has advantages compared to blocking, which might delay an application in unexpected ways. As in previous work that permitted aborts [1, 35, 13, 10], we allow operations to abort and augment F to an *abortable* type F' accordingly. F' is defined over the same set of states S and operations O as F, but returns a tuple defined over S and $A \cup \{\textsc{busy}\}$. F' may return the same output as F, but F' may also return $\perp$ and leave the state unchanged, denoting that a client is not able to execute F. Hence, F' is a non-deterministic relation and satisfies

$$F'(s, o) = \{(s, \textsc{busy}), F(s, o)\} \ . \tag{1}$$

Since F' is not deterministic, a sequence of operations no longer uniquely determines the resulting state and response value. Abortable types may be seen as obstruction-free objects [1, 31] and vice versa; such objects guarantee that every client operation completes assuming the client eventually runs in isolation.

Digital signatures. A *digital signature scheme* DS specifies the following. A probabilistic key-generation algorithm DS.KEYGEN that takes as input the security parameter and produces a pair $(ssk, spk) \leftarrow_{\$} \mathrm{DS.}\textsc{keygen}(\lambda)$ of (private) signature key *ssk* and (public) verification key *spk*. Second, a

```
Game $\mathbf{G}^{\mathrm{euf}}_{\mathrm{DS}}(\mathcal{A})$
    $(ssk, spk) \leftarrow\!\!\$\ \mathrm{DS.}\textsc{keygen}(\lambda)$
    $(m, \varphi) \leftarrow\!\!\$\ \mathcal{A}^{\textsc{sign}}(spk)$
    Return $\mathrm{DS.}\textsc{verify}(spk, m, \varphi)$ and $m \notin \mathcal{M}$

$\textsc{sign}(m)$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
    $\varphi \leftarrow\!\!\$\ \mathrm{DS.}\textsc{sign}(ssk, m)$
    Return $\varphi$
```

Fig. 1. The existential unforgeability security game for signatures.

(possibly probabilistic) signature algorithm DS.SIGN that takes as input a secret key *ssk* and a message m and outputs $\varphi \leftarrow\!\!\$\ \mathrm{DS.}\textsc{sign}(ssk, m)$, a signature. Third, a (deterministic) verification algorithm DS.VERIFY that takes as input public key *spk*, message m, and signature φ, and produces a Boolean $b \leftarrow \mathrm{DS.}\textsc{verify}(spk, m, \varphi)$. Correctness means that with probability 1, for $(ssk, spk) \leftarrow\!\!\$\ \mathrm{DS.}\textsc{keygen}$ and all messages m,

$$\mathrm{DS.}\textsc{verify}(spk, m, \mathrm{DS.}\textsc{sign}(ssk, m)) = \textsc{True}.$$

The security definition we use in this paper is existential unforgeability and is defined via the game specified in Figure 1. Formally, the EUF-advantage of an adversary $\mathcal{A}$ is defined as $\mathrm{Adv}^{\mathrm{EUF}}_{\mathrm{DS}}(\mathcal{A}) := \Pr\left[\mathbf{G}^{\mathrm{euf}}_{\mathrm{DS}}(\mathcal{A})\right]$.

Offline-online verifiable computation. For a relation $R \subseteq U \times W$, we are interested in proving statements of the type $\exists w \in W : R(u, w)$ for a given $u \in U$. We consider a setting where U splits into $U = X \times V$. For example, u may consist of the input x and output y of a function f with domain X, i.e., $y = f(x)$. The witness $w \in W$ can often speed up verification by providing a non-deterministic hint—verification may be more efficient that computation.

A *verifiable computation scheme* VC specifies the following. A key-generation algorithm VC.KEYGEN that takes as input security parameter λ and relation $R \subset U \times W$ and produces $(ek, vk) \leftarrow\!\!\$\ \mathrm{VC.}\textsc{keygen}(\lambda, R)$, a pair of evaluation key *ek* and verification key *vk*. An algorithm VC.PROVE that takes as input evaluation key *ek*, $u \in U$, and witness $w \in W$ such that $(u, w) \in R$, and returns a proof $\xi \leftarrow\!\!\$\ \mathrm{VC.}\textsc{prove}(ek, u, w)$. An algorithm VC.VERIFY that takes as input the verification key *vk*, input u, and proof ξ, and returns a Boolean $\textsc{True}/\textsc{False} \leftarrow \mathrm{VC.}\textsc{verify}(vk, u, \xi)$ that signifies whether ξ is valid.

The correctness error of VC is the probability that the verification of an honestly computed proof for a correct statement returns FALSE. The soundness advantage of an adversary is defined via game $\mathbf{G}^{\mathrm{vc}}_{\mathrm{VC},R}$ in

```
Game $\mathbf{G}^{\mathrm{vc}}_{\mathrm{VC},R}(\mathcal{A})$
$(\mathit{ek}, \mathit{vk}) \leftarrow\$ \mathrm{VC}.\textsc{keygen}(\lambda, R)$
$(u, \xi) \leftarrow\$ \mathcal{A}(\mathit{ek}, \mathit{vk}, R)$
Return $\mathrm{VC}.\textsc{verify}(\mathit{vk}, u, \xi)$
    and $\neg\exists w : (u, w) \in R$
```

```
Game $\mathbf{G}^{\mathrm{ext}}_{\mathrm{HP},\mathcal{X}}(\mathcal{A}, \mathcal{E})$
$\mathit{pp} \leftarrow\$ \mathrm{HP}.\textsc{setup}(\lambda)$
$\mathit{aux} \leftarrow\$ \mathcal{X}(\mathit{pp})$
$(h; x_e) \leftarrow\$ (\mathcal{A}; \mathcal{E})(\mathit{pp}, \mathit{aux})$
Return $\mathrm{HP}.\textsc{check}(\mathit{pp}, x)$
    and $\mathrm{HP}.\textsc{hash}(\mathit{pp}, x_e) \neq h$
```

Fig. 2. Left: The soundness game for verifiable computation schemes. **Right:** the hash-extractability game.

```
Game $\mathbf{G}^{\mathrm{hps}}_{\mathrm{HP},R}(\mathcal{A})$
$\mathit{pp} \leftarrow\$ \mathrm{HP}.\textsc{setup}(\lambda)$
$(\mathit{ek}, \mathit{vk}) \leftarrow\$ \mathrm{HP}.\textsc{keygen}(\mathit{pp}, R)$
$(x, v, \pi) \leftarrow\$ \mathcal{A}(\mathit{pp}, \mathit{ek}, \mathit{vk})$
$h_x \leftarrow \mathrm{HP}.\textsc{hash}(\mathit{pp}, x)$
Return $\mathrm{HP}.\textsc{verify}(\mathit{vk}, h_x, v, \pi)$
    and $\neg\exists w : ((x, v), w) \in R$
```

```
Game $\mathbf{G}^{\mathrm{hphs}}_{\mathrm{HP},R}(\mathcal{A}; \mathcal{E})$
$\mathit{pp} \leftarrow\$ \mathrm{HP}.\textsc{setup}(\lambda)$
$(\mathit{ek}, \mathit{vk}) \leftarrow\$ \mathrm{HP}.\textsc{keygen}(\mathit{pp}, R)$
$(h, v, \pi; x_e) \leftarrow\$ (\mathcal{A}; \mathcal{E})(\mathit{pp}, \mathit{ek}, \mathit{vk})$
Return $\mathrm{HP}.\textsc{verify}(\mathit{vk}, h, v, \pi)$
    and $\mathrm{HP}.\textsc{check}(\mathit{pp}, h)$
    and $\neg\exists w : ((x_e, v), w) \in R$
```

Fig. 3. Soundness and hash-soundness games for hash&prove schemes.

Figure 2, in which a malicious prover must produce a proof for a false statement. Both quantities must be small for a scheme to be useful.

The verifiable computation schemes we use in this work have a special property referred to as *offline-online verification*, and which is defined when the set U can be written as $U = X \times V$. In particular, for those schemes there exist algorithms VC.OFFLINE and VC.ONLINE such that

$$\mathrm{VC}.\textsc{verify}(\mathit{vk}, (x, v), \xi) = \mathrm{VC}.\textsc{online}(\mathit{vk}, \mathrm{VC}.\textsc{offline}(\mathit{vk}, x), v, \xi) .$$

Hash&prove schemes. We again consider the relation $R \subseteq U \times W$ as above. A hash&prove scheme HP then allows to prove statements of the type $\exists w \in W : R(u, w)$ for a given $u \in U$; one crucial property of hash&prove schemes is that one can produce a short proof of the statement (using the witness w), such that the verification does not require the element $u \in U$ but only a short representation of it.

In more detail, a multi-relation hash&prove scheme as defined by Fiore et al. [20] consists of five algorithms. Algorithm HP.SETUP takes as input security parameter λ and produces public parameters $\mathit{pp} \leftarrow\$ \mathrm{HP}.\textsc{setup}(\lambda)$. Algorithm HP.HASH takes as input public parameters pp and a value $x \in$

X and produces a hash $h_x \leftarrow \text{HP.}\textsc{hash}(pp, x)$. Algorithm HP.KEYGEN takes as input public parameters pp and a relation R and outputs a pair of evaluation key and verification key $(ek_R, vk_R) \leftarrow\!\!\$\ \text{HP.}\textsc{keygen}(pp, R)$. Algorithm HP.PROVE takes as input evaluation key ek_R, values $(x, v) \in X \times V$ and witness $w \in W$ such that $((u, v), w) \in R$, and produces a proof $\pi \leftarrow\!\!\$\ \text{HP.}\textsc{prove}(ek_R, (x, v), w)$. Finally, algorithm HP.VERIFY takes as input verification key vk_R, hash h_x, value v, and proof π and outputs a Boolean TRUE/FALSE $\leftarrow \text{HP.}\textsc{verify}(vk_R, h_x, v, \pi)$.

An *extractable* hash&prove scheme has an additional (deterministic) algorithm HP.CHECK that takes as input pp and a hash h and outputs TRUE/FALSE $\leftarrow \text{HP.}\textsc{check}(pp, h)$, a Boolean that signifies whether the hash is well-formed (i.e., there is a pre-image). For defining the hash-extraction property, we consider an extractor $\mathcal{E}$ for adversary $\mathcal{A}$, and in the game $\mathbf{G}^{\text{ext}}_{\text{HP},\mathcal{X}}$ in Figure 2 we mean by $(h; x_e) \leftarrow\!\!\$\ (\mathcal{A}; \mathcal{E})(pp)$ that both algorithms $\mathcal{A}$ and $\mathcal{E}$ are run on the same input and random tape, and that h is the output of $\mathcal{A}$ and x_e is the output of $\mathcal{E}$. For adversary $\mathcal{A}$ and extractor $\mathcal{E} = \mathcal{E}(\mathcal{A})$, the *hash-extraction advantage of* $\mathcal{A}; \mathcal{E}$, relative to benign distribution $\mathcal{X}$ (from which auxiliary input aux is drawn), is defined as $\text{Adv}^{\text{EXT}}_{\text{HP},\mathcal{X}}(\mathcal{A}, \mathcal{E}) := \Pr[\mathbf{G}^{\text{ext}}_{\text{HP},\mathcal{X}}(\mathcal{A}, \mathcal{E})]$.

Correctness of HP is defined in the natural way; namely by requiring that the evaluation of the above algorithms honestly leads to HP.VERIFY outputting TRUE. We define *soundness advantage* and the *hash-soundness advantage* of an adversary $\mathcal{A}$ as

$$\text{Adv}^{\text{HPS}}_{\text{HP},R}(\mathcal{A}) := \Pr[\mathbf{G}^{\text{hps}}_{R,\text{HP}}(\mathcal{A})]\ ;\ \text{Adv}^{\text{HPHS}}_{\text{HP},R}(\mathcal{A}; \mathcal{E}) := \Pr[\mathbf{G}^{\text{hphs}}_{R,\text{HP}}(\mathcal{A}; \mathcal{E})]\ .$$

Both games are described in Figure 3; in contrast to the original definitions [20], we describe non-adaptive versions for a single relation, since this is simpler and sufficient for our setting.

At a high level, both soundness games formalize as a goal for an adversary to produce a proof for a false statement that will be accepted by HP.VERIFY. Adversary $\mathcal{A}$ is given public parameters pp, evaluation key ek, and verification key vk. In the soundness game, $\mathcal{A}$ has to produce a proof for a statement (x, v) that is wrong according to the fixed relation R, but the proof is accepted by HP.VERIFY when $h_x \leftarrow \text{HP.}\textsc{hash}(pp, x)$ is computed honestly.

The purpose of hash soundness is to capture the scenario where HP can support arguments on untrusted, opaque hashes that are provided by the adversary. For this, the HP.HASH algorithm must be extractable. The hash-soundness game operates almost as the soundness game, but instead of x, the adversary provides a hash h. The adversary wins if the hash h cannot be opened in a consistent manner (by the extractor $\mathcal{E}$) to satisfy the relation; for further explanation, we point the readers to [20, Appendix A.1], but we stress that the explicit extraction in our definition is needed for our use of the scheme.

Finally, we define the collision advantage of adversary $\mathcal{A}$ as

$$\mathrm{Adv}_{\mathrm{HP}}^{\mathrm{CR}}(\mathcal{A}) := \Pr\left[\begin{array}{l} pp \leftarrow\!\!\$\ \mathrm{HP}.\textsc{setup};\, (x, y) \leftarrow\!\!\$\ \mathcal{A}(pp); \\ \mathrm{HP}.\textsc{hash}(pp, x) \stackrel{?}{=} \mathrm{HP}.\textsc{hash}(pp, y) \end{array}\right]$$

Hash&prove for multi-exponentiation. We recall the hash&prove scheme for multi-exponentiation introduced in [20], which uses asymmetric bilinear prime-order groups $\mathcal{G}_\lambda = (e, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$ with an admissible bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We keep the details light since we do not use properties other than those already used in [20], and we want to mostly focus on our new contributions.

The scheme, which we call MXP, works for relations $G \subseteq U \times \emptyset$ with $U = \mathbb{Z}_p^n \times \mathbb{G}_1$. Each such relation G is described by a vector $(G_1, \ldots, G_n) \in \mathbb{G}_1^n$. A pair $(x, c_x) \in \mathbb{Z}_p^n \times \mathbb{G}_1$ is in G if $\prod_{i=1}^n G_i^{x_i} = c_x$. The scheme, which is called $\mathsf{XP}_{\mathcal{E}}$ and described and proved in [20].

3 Authenticated data types

Authenticated data types, which can be thought of as an abstraction and generalization of Merkle trees [39], associate with a (potentially large) state of the data type a short *authenticator* (or *digest*) that is useful for verification of the integrity and authenticity of the data type. In more detail, an abstract data type is described by a state space S with a function $F : S \times O \to S \times A$ defined on it. F takes as input a state $s \in S$ of the data type and an operation $o \in O$ and returns a new state s' and the response $r \in A$. The data type also specifies the initial state $s_0 \in S$.

Here, we present a definition for what is known in the literature as a “two-party” *authenticated data type (ADT)* [44]. The interaction is between a *client*, i.e., party that owns a data type which it wants to outsource, and an untrusted *server* that undertakes storing the state of this outsourced data type and responding to subsequent queries issued. The client, having access only to a succinct *authenticator* and the secret key of the scheme, wishes to be able to efficiently test that requested operations have been performed honestly by the server (see [44] for a more detailed comparison of variants of ADT modes of operation). An authenticated data type ADT := ADT_F consists of the following algorithms.

$(sk, ad, a) \leftarrow_{\$} \mathrm{ADT}.\textsc{init}(\lambda)$: This algorithm sets up the secret key and the public key for the ADT scheme. It also outputs an initial amended state ad and a succinct authenticator a. (We implicitly assume from now on that pk is part of sk and ad, and that the actual initial state s_0 and authenticator a are part of ad.)

$\pi \leftarrow_{\$} \mathrm{ADT}.\textsc{exec}(ad, o)$: This algorithm takes an operation o, applies it on the current version of ad, and provides a correctness proof π (from which potential local output of the operation can be extracted).

$(\textsc{True}/\textsc{False}, r, a', t) \leftarrow_{\$} \mathrm{ADT}.\textsc{verify}(sk, a, o, \pi)$: The algorithm takes the current authenticator a, an operation o, and a proof π, verifies the proof with respect to the authenticator and the operation, outputting local output r, the updated authenticator a', and an additional authentication token t.

$ad' \leftarrow_{\$} \mathrm{ADT}.\textsc{refresh}(ad, o, t)$: This algorithm updates the amended state from ad to ad', using operation o and authentication token t provided by the client.

An ADT has to satisfy two conditions, correctness and soundness. Correctness formalizes that if the ADT is used faithfully, then the outputs received by the client are according to the abstract data type F.

Definition 1 (Correctness). *Let s_0 be the initial state of data type F and $o_1, \ldots, o_m$ be a sequence of operations. The ADT scheme* ADT *is* correct *if in the following computation, the assertions are always satisfied.*

```
$(sk, ad, a) \leftarrow_{\$} \mathrm{ADT}.\textsc{init}(\lambda)$ ; $s \leftarrow s_0$
For $j = 1, \ldots, m$ do
  $\pi \leftarrow_{\$} \mathrm{ADT}.\textsc{exec}(ad, o_j)$
  $(b, r, a', t) \leftarrow \mathrm{ADT}.\textsc{verify}(sk, a, o_j, \pi)$
  $(s', r') \leftarrow_{\$} F(s, o_j)$
  assert $b$ and $r = r'$
  $ad' \leftarrow_{\$} \mathrm{ADT}.\textsc{refresh}(ad, o_j, t)$
  $(ad, a, s) \leftarrow (ad', a', s')$
```

The second requirement for the ADT, soundness, states that a dishonest server cannot cheat. The game $\mathbf{G}^{\mathrm{sound}}_{\mathrm{ADT}}$ described in Figure 4 formalizes that it must be infeasible for the adversary (a misbehaving server) to produce a proof that makes a client accept a wrong response of an operation. The variable *forged* tracks whether the adversary has been successful. The list $L[\,]$ is used to store valid pairs of state and authenticator of the ADT, and is consequently initialized with (s_0, a) of a newly initialized ADT in position 0. The adversary $\mathcal{A}$ is initialized with (ad, a) and can repeatedly query the $\textsc{verify}$ oracle in the game by specifying an operation o, the

```
Game $\mathbf{G}^{\text{sound}}_{\text{ADT}}(\mathcal{A})$
$\mathit{forged} \leftarrow \textsc{False}$
$(\mathit{sk}, \mathit{ad}, a) \leftarrow\!\!\$\ \text{ADT.}\textsc{init}(\lambda)$
$L[0] \leftarrow (s_0, a)$
$\mathcal{A}^{\textsc{verify}}(\mathit{ad}, a)$
Return $\mathit{forged}$
```

```
$\textsc{verify}(o, \mathit{pos}, \pi)$
If $\mathit{pos} > |L|$ then return $\bot$
$(s, a) \leftarrow L[\mathit{pos}]$
$(b, r, a', t) \leftarrow\!\!\$\ \text{ADT.}\textsc{verify}(\mathit{sk}, a, o, \pi)$
If $b$ then
    $(s', r') \leftarrow F(s, o)$
    If $r' \neq r$ then $\mathit{forged} \leftarrow \textsc{True}$
    $L \leftarrow L \circ (s', a')$
    Return $(\textsc{True}, a', t, r)$
Else return $(\textsc{False}, \bot, \bot, \bot)$
```

Fig. 4. The security game formalizing soundness of an ADT.

index *pos* $\in \mathbb{N}$ of a state on which o shall be executed, and a proof π. The challenger then obtains state s and authenticator a of the i-th state from the list $L[\,]$. The challenger then (a) checks whether ADT.VERIFY accepts the proof π, and (b) computes the new state s' and the output r' using the "ideal" F and state s, and sets *forged* if the proof verified but the output r generated by ADT.VERIFY does not match the "ideal" output r'.

This formulation of the game ensures that the outputs provided to the clients are always correct according to F and the sequence of operations performed, but also allows the adversary to "fork" and compute different operations based on the same state.[4] This is necessary for proving the security of the protocol we describe in Section 6. Unlike for the output r, the game does not formalize an explicit correctness condition for ad' to properly represent the state s' of F as updated by o'; this is only modeled through the outputs generated during subsequent operations. Indeed, in the two-party model, the internal state of the (untrusted) server cannot be directly observed, and only the correctness of the responses provided to clients matters.

Definition 2 (Soundness). *Let F be an abstract data type and* ADT *an ADT for F. Let $\mathcal{A}$ be an adversary. The* soundness advantage of $\mathcal{A}$ against ADT *is defined as* $\mathrm{Adv}^{\text{SOUND}}_{\text{ADT}}(\mathcal{A}) := \Pr\left[\mathbf{G}^{\text{sound}}_{\text{ADT}}\right]$.

To exclude trivial schemes in which the server always sends the complete state to the clients, we explicitly require that the authenticator of

---

[4] A definition that only allows the adversary to compute a single sequence of operations is not generically equivalent for schemes where *sk* is non-trivial.

the clients must be *succinct*. More concretely, we require that the size of the authenticator is independent of the size of the state.

Definition 3 (Succinctness). *Let F be an abstract data type and* ADT *an ADT with security parameter λ for F. Then* ADT *is* succinct *if the bit-length of the authenticator a is always in $\mathcal{O}(\lambda)$.*

4 A general-purpose instantiation of ADT

This section contains one main technical contribution of this work, namely a general-purpose instantiation of the definition of ADT described in Section 3. Our scheme builds on the work of Fiore et al. [20], which defined hash&prove schemes in which a server proves the correctness of a computation (relative to a state) to a client that only knows a hash value of the state. The main aspect missing from [20] is the capability for an untrusted server to *update* the state and provide the client with a new hash value that authenticates the new state. Note that the hash of an updated state can be computed incrementally given the hash of the previous state, as described in [20, Section 4.4].

Before we start describing our scheme, we recall some details of the hash&prove scheme of Fiore et al. [20]. For a function $f : Z \to V$, they set $U = Z \times V$ and consider a relation $R_f \subseteq U \times W$ such that for a pair $(z, v) \in U$ there is a $w \in W$ with $((z, v), w) \in R_f$ if and only if $f(z) = v$. In other words, proving $\exists w : ((z, v), w) \in R_f$ implies that $f(z) = v$.

They then continue via an offline-online verifiable computation scheme VC and a hash-extractable hash&prove scheme for multi-exponentiations MXP. Recall that the offline-online property states that

$$\mathrm{VC}.\textsc{verify}(vk, (z, v), \xi) = \mathrm{VC}.\textsc{online}(vk, \mathrm{VC}.\textsc{offline}(vk, z), v, \xi)\ .$$

The server computes $c_z = \mathrm{VC}.\textsc{offline}(vk, z)$ and proves to the client via MXP that c_z contains the same value as the hash h_z known to the client. The client then concludes by verifying the proof via VC.ONLINE with input c_z.

Building the new hash&prove scheme. Our goal is to model stateful computations of the type $F(x, o) = (y, r)$. To that end, we use the set $U = (X \times Y) \times (O \times A)$ and consider a relation $R_F \subseteq U \times W$ such that for $((x, y), (o, r)) \in U$ there is a $w \in W$ with $(((x, y), (o, r)), w) \in R_F$ if and only if $F(x, o) = (y, r)$. This can—more programmatically—be realized as first computing $(y, \_) \leftarrow F(x, o)$ *without verification* and then *verifiably* computing $\tilde{F}((x, y), o) \mapsto (d, r)$ with $(y', r) \leftarrow F(x, o); d \leftarrow y' \overset{?}{=} y$.

The client then has to check the proof of the verifiable computation *and* that $d = \textsc{True}$. Putting the output y into the input of the verifiable computation of $\tilde{F}$ has the advantage that we already know how to handle hashes there: with the relation $R_{\tilde{F}}$ and (almost) the scheme of [20].

In [20], the output of $\mathrm{VC}.\textsc{offline}(vk, z)$ is a single value c_z that is then related to the hash h_z known to the client via MXP. As we have two individual hashes h_x and h_y for the input x and the output y, respectively, we modify the construction of [20]. For $z \in X \times Y$ with $X = Y = \mathbb{Z}_p^n$, we modify $\mathrm{VC}.\textsc{offline}(vk, z)$ to compute

$$c_x \leftarrow \prod_{i=1}^{n} G_i^{x_i} \quad ; \quad c_y \leftarrow \prod_{i=1}^{n} G_{n+i}^{y_i}$$

for elements $G_1, \ldots, G_{2n}$ that are specified in vk, and prove consistency of c_x with h_x and of c_y with h_y, again using MXP. (Note that this is $c_z = c_x c_y$.) As argued by [20], many existing VC/SNARK constructions can be written in this way.

Summarizing the above, the main modifications over [20] are (i) that we define the relation $R_{\tilde{F}}$ for *stateful* F, (ii) that $\mathrm{VC}.\textsc{online}$ obtains two elements c_x and c_y from $\mathrm{VC}.\textsc{offline}$, and (iii) that the output bit d has to be checked. Our stateful hash&prove system SHP for relation $R_{\tilde{F}}$ is specified formally in Figure 5.

Hash soundness. We show in Theorem 1 that SHP is hash sound, analogously to Corollary 4.1 of [20]. Hash soundness can intuitively be understood as that, for some given public parameter pp and relation R, it is infeasible for an adversary to output and a hash h and v along with a proof π such that $\mathrm{HP}.\textsc{verify}$ succeeds, but h was not produced by function $\mathrm{HP}.\textsc{hash}(pp, \cdot)$ on some input x, or there does not exist witness w such that $((x, v), w) \in R$.

Hash extraction enables us to use an HP scheme to verify arguments that include opaque hashes h provided by the adversary by first extracting their content then applying soundness. While we state and prove the following theorem for the non-adaptive case, we conjecture that an adaptive version can be done along the lines of [20].

Theorem 1. *Let* SHP *the scheme from Section 4 and $\mathcal{A}$ be an adversary for hash soundness. Then there is an extractor $\mathcal{E}_\mathcal{A}$ and adversaries $\mathcal{B}$, $\mathcal{C}_1$, and $\mathcal{C}_2$, explicitly described in the proof, such that, with extractors $\mathcal{E}_1$ and $\mathcal{E}_2$, guaranteed for $\mathcal{C}_1$ and $\mathcal{C}_2$ by the hash-extraction property of* MXP*,*

$$\mathrm{Adv}^{\mathrm{HPHS}}_{\mathrm{SHP},R}(\mathcal{A}; \mathcal{E}_\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{VC}}_{\mathrm{VC},R}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{EXT}}_{\mathrm{MXP},\mathcal{X}}(\mathcal{C}_1, \mathcal{E}_1) + \mathrm{Adv}^{\mathrm{EXT}}_{\mathrm{MXP},\mathcal{X}}(\mathcal{C}_2, \mathcal{E}_2),$$

```
$\underline{\mathrm{SHP}.\textsc{setup}(\lambda)}$
$pp \xleftarrow{\$} \mathrm{MXP}.\textsc{setup}(\lambda)$
Return $pp$

$\underline{\mathrm{SHP}.\textsc{hash}(pp, (x, y))}$
$h_x \leftarrow \mathrm{MXP}.\textsc{hash}(pp, x)$ ; $h_y \leftarrow \mathrm{MXP}.\textsc{hash}(pp, y)$
Return $(h_x, h_y)$

$\underline{\mathrm{SHP}.\textsc{keygen}(pp, R)}$
$(ek, vk) \xleftarrow{\$} \mathrm{VC}.\textsc{keygen}(\lambda, R)$
Let $G_1, \ldots, G_{2n}$ be the "offline" elements in $vk$, see discussion in text.
$(ek_{\mathsf{i}}, vk_{\mathsf{i}}) \xleftarrow{\$} \mathrm{MXP}.\textsc{keygen}(pp, (G_1, \ldots, G_n))$
$(ek_{\mathsf{o}}, vk_{\mathsf{o}}) \xleftarrow{\$} \mathrm{MXP}.\textsc{keygen}(pp, (G_{n+1}, \ldots, G_{2n}))$
Return $(ek_R, vk_R) = ((ek, vk, ek_{\mathsf{i}}, ek_{\mathsf{o}}), (vk, vk_{\mathsf{i}}, vk_{\mathsf{o}}))$

$\underline{\mathrm{SHP}.\textsc{prove}(ek_R, (x, y), v, w)}$
$(c_x, c_y) \leftarrow \mathrm{VC}.\textsc{offline}(vk, (x, y))$
$\xi \xleftarrow{\$} \mathrm{VC}.\textsc{prove}(ek, ((x, y), v), w)$
$\pi_x \xleftarrow{\$} \mathrm{MXP}.\textsc{prove}(ek_{\mathsf{i}}, x, c_x)$ ; $\pi_y \xleftarrow{\$} \mathrm{MXP}.\textsc{prove}(ek_{\mathsf{o}}, y, c_y)$
Return $\pi_R = (c_x, c_y, \xi, \pi_x, \pi_y)$

$\underline{\mathrm{SHP}.\textsc{check}(pp, (h_x, h_y))}$
Return $\mathrm{MXP}.\textsc{check}(pp, h_x) \wedge \mathrm{MXP}.\textsc{check}(pp, h_y)$

$\underline{\mathrm{SHP}.\textsc{verify}(vk_R, (h_x, h_y), v, \pi_R)}$
Return $\mathrm{VC}.\textsc{online}(vk, (c_x, c_y), v, \xi) \wedge \mathrm{SHP}.\textsc{check}(pp, (h_x, h_y))$
        $\wedge \mathrm{MXP}.\textsc{verify}(vk_{\mathsf{i}}, h_x, c_x, \pi_x) \wedge \mathrm{MXP}.\textsc{verify}(vk_{\mathsf{o}}, h_y, c_y, \pi_y)$
```

Fig. 5. The hash&prove scheme SHP for updates by untrusted servers.

with $\mathcal{X}$ *instantiated as* $\mathrm{SHP}.\textsc{keygen}(\cdot, R)$.

Proof. This proof will proceed as follows. First, we will show that soundness and hash extractability of SHP imply hash soundness. (This part of the proof is analogous to Theorem A.1 in [20] and we repeat it for completeness.) Then, we will show that our SHP scheme satisfies hash soundness by proving its satisfies these two properties.

We prove the first part as follows. Let $\mathcal{A}$ be an adversary against hash soundness of SHP. Then, we build from it an adversary $\mathcal{C}'$ in the hash extraction game and an adversary $\mathcal{B}'$ in the soundness game. In particular, for each adversary $\mathcal{A}$, let $\mathcal{C}'$ be the adversary that receives pp and aux for SHP as input and runs $\mathcal{A}$ internally, emulating the interaction during the hash soundness game. More concretely, $\mathcal{C}'$ parses its auxiliary input as $(ek, vk) \leftarrow aux$. It then runs $\mathcal{A}(pp, ek, vk)$, but only outputs the hash $h = (h_x, h_y)$ from the output of $\mathcal{A}$. Let $\mathcal{E}'$ be the extractor associated with $\mathcal{C}'$

from the hash extraction property. Then, let the extractor $\mathcal{E}_{\mathcal{A}}$ associated with $\mathcal{A}$ be exactly the same as $\mathcal{E}'$, except that the input is formatted differently. While $\mathcal{E}_{\mathcal{A}}$'s input is formatted as (pp, ek, vk), the input to the extractor $\mathcal{E}'$ is $(pp, aux = (ek, vk))$. Finally, let $\mathcal{B}'$ be an adversary that receives as input (pp, ek, vk) and runs $\mathcal{A}$ on input (pp, ek, vk) to obtain (h, v, ξ). Then, $\mathcal{B}'$ runs $\mathcal{E}_{\mathcal{A}}$ on input (pp, ek, vk) and with the same randomness tape as $\mathcal{A}$ to obtain x with $h = \mathrm{SHP.}\textsc{hash}(pp, x)$ and outputs (x, v, ξ).

We can now define the following sequence of games.

Game 0: is the hash soundness game with $\mathcal{A}$ and $\mathcal{E}_{\mathcal{A}}$.
Game 1: is the same as Game 0, except that Game 1 aborts if

$$\mathrm{SHP.}\textsc{check}(pp, h) = 1 \wedge h \neq \mathrm{SHP.}\textsc{hash}(pp, x) \; .$$

Game 2: is the same as Game 1, except that Game 2 aborts if

$$\mathrm{SHP.}\textsc{verify}(vk, h, v, \xi) \wedge \neg((x, v), w) \in R \; .$$

Intuitively, assuming Game 1 does not abort, it is indistinguishable from Game 0 and assuming Game 2 does not abort, it is indistinguishable from Game 1. More concretely, if $\mathcal{A}$, with extractor $\mathcal{E}_{\mathcal{A}}$, wins the hash soundness game, this implies that $\mathcal{A}$ outputs (h, v, ξ) and $\mathcal{E}_{\mathcal{A}}$ outputs x with $\mathrm{SHP.}\textsc{verify}(vk, h, v, \xi) = 1$, $\mathrm{SHP.}\textsc{check}(pp, h) = 1$, and $\neg((x, v), w) \in R$. Then, either (a) Game 1 did not abort, i.e, $\mathcal{E}'$ successfully extracted x such that $\mathrm{SHP.}\textsc{hash}(pp, x) = h \wedge \mathrm{SHP.}\textsc{check}(pp, h) = 1$. In that case $\mathcal{B}'$ can win the soundness game by outputting (x, v, ξ). Or (b) Game 1 aborted in which case $\mathcal{C}'$ with extractor $\mathcal{E}'$ can win the hash extractability game by outputting h. By a simple union bound, this shows $\mathrm{Adv}^{\mathrm{HPHS}}_{\mathrm{SHP},R}(\mathcal{A}; \mathcal{E}_{\mathcal{A}}) \leq \mathrm{Adv}^{\mathrm{HPS}}_{\mathrm{SHP},R}(\mathcal{B}') + \mathrm{Adv}^{\mathrm{EXT}}_{\mathrm{SHP},\mathcal{X}}(\mathcal{C}', \mathcal{E}')$. This concludes the first part of the proof.

Now, to show that our SHP scheme satisfies hash soundness, we show that it satisfies both hash extractability and soundness. In particular, we have to show that if VC is sound and MXP is hash-extractable, then SHP satisfies both hash extractability and soundness. To prove this claim, let us assume that $\mathcal{C}'$ is an adversary that breaks the hash extractibility of SHP and $\mathcal{B}'$ breaks the soundness of the SHP. Using $\mathcal{C}'$ and $\mathcal{B}'$, we can either build an adversary $\mathcal{C}$ that breaks the security of the MXP scheme or we can build an adversary $\mathcal{B}$ that breaks the soundness of the VC as follows.

- From $\mathcal{C}'$, we build two algorithms $\mathcal{C}_1$ and $\mathcal{C}_2$ that obtain (pp, aux) and run $\mathcal{C}'$ on the same input. From the output $h = (h_x, h_y)$ of $\mathcal{C}'$, algorithms $\mathcal{C}_1$ and $\mathcal{C}_2$ output h_x and h_y, respectively. By the hash extractibility of MXP, we have extractors $\mathcal{E}_1 = \mathcal{E}_1(\mathcal{C}_1)$ and $\mathcal{E}_2 = \mathcal{E}_2(\mathcal{C}_2)$, from which we then build an extractor $\mathcal{E}'$ for $\mathcal{C}'$ that runs both $\mathcal{E}_1$ and $\mathcal{E}_2$ on its inputs, obtains x and y, and succeeds if both $\mathcal{E}_1$ and $\mathcal{E}_2$ are successful.

- From $\mathcal{B}'$, we build $\mathcal{B}$ by generating $pp \leftarrow_\$ \text{MXP.}\textsc{setup}(\lambda)$, and using (ek, vk) obtained in the game to run $\mathcal{B}'$ on input (pp, ek, vk), which has the correct distribution. A winning output (x, v, ξ) of $\mathcal{B}'$ is a winning output for $\mathcal{B}$.

Therefore, we have

$$\begin{aligned}&\mathrm{Adv}^{\mathrm{HPHS}}_{\mathrm{SHP},R}(\mathcal{A};\mathcal{E}_{\mathcal{A}})\\ &\qquad\le \mathrm{Adv}^{\mathrm{VC}}_{\mathrm{VC},R}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{EXT}}_{\mathrm{MXP},\mathcal{X}}(\mathcal{C}_1,\mathcal{E}_1) + \mathrm{Adv}^{\mathrm{EXT}}_{\mathrm{MXP},\mathcal{X}}(\mathcal{C}_2,\mathcal{E}_2)\ .\end{aligned}$$

□

Building a general-purpose ADT using our HP. The scheme SHP constructed above lends itself well to building a general-purpose ADT. Note that verifiable computation schemes explicitly construct the witness w required for the correctness proof; in fact, the computation of F can also be used to produce a witness w for the correctness according to $\tilde{F}$, which is immediate for VC schemes that actually model F as a circuit [24, 47].

The general-purpose ADT GA, which is more formally described in Figure 6 and proved below, works as follows. Algorithm GA.INIT generates public parameters pp and a key pair (ek, vk) for SHP, and then computes the authenticator $(a, \_) \leftarrow \text{SHP.}\textsc{hash}(pp, (s_0, \epsilon))$ for the initial state s_0 of F.[5] Algorithm GA.EXEC computes the new state s' via F and authenticator $(a', \_) \leftarrow \text{SHP.}\textsc{hash}(pp, (s', \epsilon))$, and generates a correctness proof ξ for the computation of $\tilde{F}$ via SHP.PROVE. Algorithm GA.VERIFY checks the proof ξ via SHP.VERIFY and also checks the bit d output by $\tilde{F}$ to ensure that the authenticator a' is correct. Algorithm GA.REFRESH simply updates the server state—recomputing s' and a' can be spared by caching the values from GA.EXEC.

Instantiating GA with the schemes of [20] leads to a succinct ADT.

---

[5] The function SHP.HASH produces pairs of hashes from pairs of states. As in GA we only need to hash one state, we ignore the second component. We could—and would in an implementation—alternatively re-define SHP.HASH to only process one state at a time, but this would contradict the formal definition of a hash&prove scheme.

```
$\underline{\mathrm{GA}.\textsc{init}_F(\lambda)}$
$pp \leftarrow\!\!\$\; \mathrm{SHP}.\textsc{setup}(\lambda)$
$(ek, vk) \leftarrow\!\!\$\; \mathrm{SHP}.\textsc{keygen}(pp, R_{\tilde{F}})$
$(a, \_) \leftarrow \mathrm{SHP}.\textsc{hash}(pp, (s_0, \epsilon))$
Return $(vk, (s_0, a, ek, vk), a)$

$\underline{\mathrm{GA}.\textsc{exec}_F(ad, o)}$
$(s, a, ek, vk) \leftarrow ad$
$(s', r) \leftarrow F(s, o)$        $\triangleright$ Get witness $w$
$\xi \leftarrow\!\!\$\; \mathrm{SHP}.\textsc{prove}(ek, (s, s'), (o, r), w)$
$(a', \_) \leftarrow \mathrm{SHP}.\textsc{hash}(pp, (s', \epsilon))$
Return $\pi = (\xi, a', r)$

$\underline{\mathrm{GA}.\textsc{verify}(sk, a, o, \pi)}$
$(\xi, a', r') \leftarrow \pi$ ; $(d, r) \leftarrow r'$
$b \leftarrow d \wedge \mathrm{SHP}.\textsc{verify}(sk, (a, a'), (o, r'), \xi)$
Return $(b, r, a', \epsilon)$

$\underline{\mathrm{GA}.\textsc{refresh}_F(ad, o, t)}$
$(s, a, ek, vk) \leftarrow ad$
$(s', r) \leftarrow F(s, o)$
$(a', \_) \leftarrow \mathrm{SHP}.\textsc{hash}(pp, (s', \epsilon))$
Return $(s', a', ek, vk)$
```

Fig. 6. The general-purpose ADT scheme GA that can be instantiated for any data type F. Algorithm GA.REFRESH does not use the value t; this value only appears because it is included in the general definition of ADT and could be useful in other schemes.

Theorem 2 (ADT Soundness). *Let* GA *be the scheme as described above and* $\mathcal{A}$ *be an adversary in the* $\mathbf{G}_{\mathrm{GA}}^{\mathrm{sound}}$ *game. Then there is an adversary* $\mathcal{B}$*, described explicitly in the proof, such that with the extractor* $\mathcal{E}_{\mathcal{B}}$ *guaranteed for* $\mathcal{B}$*,*

$$\mathrm{Adv}_{\mathrm{GA}}^{\mathrm{SOUND}}(\mathcal{A}) \leq \mathrm{Adv}_{\mathrm{SHP},R_{\tilde{F}}}^{\mathrm{HPHS}}(\mathcal{B}, \mathcal{E}_{\mathcal{B}}) + \mathrm{Adv}_{\mathrm{SHP}}^{\mathrm{CR}}(\mathcal{C}) \ . \tag{2}$$

If $\mathcal{A}$ *makes* q VERIFY *calls, then* $\mathcal{B}$ *makes* q VERIFY *calls in its game.*

Proof. Let $\mathcal{A}$ be an adversary in the $\mathbf{G}_{\mathrm{GA}}^{\mathrm{sound}}$ game. We show that, given $\mathcal{A}$, we can either build an adversary $\mathcal{B}$ that breaks the hash soundness of the SHP or and adversary $\mathcal{C}$ that breaks the collision resistance of the hash function. The proof proceeds as follows:

We describe the adversary $\mathcal{B} := \mathcal{B}(\mathcal{A})$ that plays game $\mathbf{G}_{\mathrm{SHP},R_{\tilde{F}}}^{\mathrm{hphs}}$ and simulates to $\mathcal{A}$ the game $\mathbf{G}_{\mathrm{GA}}^{\mathrm{sound}}$. Adversary $\mathcal{B}$ initially obtains the parameters pp of SHP and keys (ek, vk). It computes $a \leftarrow \mathrm{SHP}.\textsc{hash}(pp, s_0)$ and calls $\mathcal{A}$ with input $((s_0, a, ek, vk), a)$. Adversary $\mathcal{B}$ internally keeps variables i and $L[\,]$ analogously to $\mathbf{G}_{\mathrm{GA}}^{\mathrm{sound}}$.

For (valid) oracle calls $\textsc{verify}(o, i, \pi)$, with $\pi = (\xi, a', (d, r))$ with $d = \textsc{True}$, issued by $\mathcal{A}$, adversary $\mathcal{B}$ obtains $(s, a) \leftarrow L[i]$ and computes and returns $\mathrm{SHP}.\textsc{verify}(vk, (a, a'), (d, r), \xi)$. Adversary $\mathcal{B}$ also computes $(s', r') \leftarrow F(s, o)$ and stores (s', a') in $L[\,]$. If $r \neq r'$ but SHP.VERIFY returned True, then $\mathcal{B}$ outputs $((a, a'), (o, r), \xi)$ as a solution to the game $\mathbf{G}^{\text{hphs}}_{\text{SHP},R_{\tilde{F}}}$.

Adversary $\mathcal{B}$ emulates the game to $\mathcal{A}$ perfectly, and whenever $\mathcal{A}$ wins the emulated game, there are three possibilities.

1. Let the corresponding state as looked up from $L[.]$ be s and $\mathcal{E}_{\mathcal{B}}$ extracts a state $s' = s$. In this case $\mathcal{B}$ wins the hash-soundness game.
2. If $s' \neq s$ and $\text{SHP.}\textsc{hash}(pp, s) \neq \text{SHP.}\textsc{hash}(pp, s')$, then also, $\mathcal{B}$ wins the hash-soundness game.
3. If $s' \neq s$ and $\text{SHP.}\textsc{hash}(pp, s) = \text{SHP.}\textsc{hash}(pp, s')$, then, an adversary $\mathcal{C}$ that runs both $\mathcal{B}$ and $\mathcal{E}_{\mathcal{B}}$ breaks collision resistance of the hash function.

Thus, whenever $\mathcal{A}$ wins the emulated game $\mathbf{G}^{\text{sound}}_{\text{GA}}$, either $\mathcal{B}$ wins the hash-soundness game or $\mathcal{C}$ wins the collision resistance game. Therefore, we have

$$\text{Adv}^{\text{SOUND}}_{\text{GA}}(\mathcal{A}) \leq \text{Adv}^{\text{HPHS}}_{\text{SHP},R_{\tilde{F}}}(\mathcal{B}, \mathcal{E}_{\mathcal{B}}) + \text{Adv}^{\text{CR}}_{\text{SHP}}(\mathcal{C}) \ . \tag{3}$$

□

5 Computational fork-linearizable Byzantine emulation

The application we target in this paper is verifiable multiple-client computation of an ADT F with an untrusted server for coordinating the joint computation. As the clients may not be online simultaneously, we do not assume any direct communication among the clients. The goal of the protocol is to emulate an abstract data type $F : (s, o) \mapsto (s', r)$. As the server may be malicious, this setting is referred to as *Byzantine emulation* in the literature [14].

A Byzantine emulation protocol BEP specifies the following: A setup algorithm BEP.setup takes as parameter the number $u \in \mathbb{N}$ of clients and outputs, for each client $v \in \mathbb{N}$, key information clk_v, server key information svk, and public key information pks. (The variable pks models information that is considered public, such as the clients' public keys.) A client algorithm BEP.invoke takes as input an operation $o \in \{0,1\}^*$, secret information $clk \in \{0,1\}^*$, public keys $pks \in \{0,1\}^*$ and state $S \in \{0,1\}^*$, and outputs a message $m \in \{0,1\}^*$ and a new state $S' \in \{0,1\}^*$. A client algorithm BEP.receive takes as input a message $m \in \{0,1\}^*$, and clk, pks, and S as above, and outputs a value $r \in \{0,1\}^* \cup \{\textsc{abort}, \textsc{busy}\}$, a message $m' \in \{0,1\}^* \cup \{\perp\}$, and a new state $S' \in \{0,1\}^*$. The return value ABORT means that the operation has been aborted because of an error or inconsistency of the system, whereas BUSY means that the server is busy executing a different operation and the client shall repeat the invocation later. A server algorithm BEP.PROCESS takes as input a message $m \in \{0,1\}^*$, purported sender $v \in \mathbb{N}$, secret information $svk \in \{0,1\}^*$, public keys $pks \in \{0,1\}^*$and state $S_\mathsf{s} \in \{0,1\}^*$, and outputs a message $m' \in \{0,1\}^*$, intended receiver $v' \in \mathbb{N}$, and updated state $S'_\mathsf{s} \in \{0,1\}^*$.

We then define the (parametrized) security game $\mathbf{G}^{\mathrm{emu}}_{\mathrm{BEP},u,\mathrm{P}}$ described in Figure 7, which is roughly inspired by the key-establishment game of Bellare and Rogaway [6]. Initially, the game calls BEP.SETUP to generate the necessary keys; the setup phase modeled here allows the clients to generate and distribute keys among them. This allows for modeling, for instance, a public-key infrastructure, or just a MAC key that is shared among all clients. (Note that we consider all clients as honest.) The adversary $\mathcal{A}$, which models the network as well as the malicious server, is executed with input *pks*—the public keys of the scheme—and has access to four oracles. Oracle INVOKE(v, o) models the invocation of operation o at client C_v, updates the state S_v, and appends the input event (C_v, o) to the history σ. The oracle returns a message m directed at the server. Oracle RECEIVE(v, m) delivers the message m to C_v, updates the state S_v, and outputs a response r and a message m'. If $r \neq \bot$, the most recently invoked operation of C_v completes and the output event (C_v, r) is appended to σ. If $m' \neq \bot$, then m' is a further message directed at the server. Oracle CORRUPT returns the server state S_s, and oracle PROCESS(v, m) corresponds to delivering message m to the server as being sent by C_v. This updates the server state S_s, and may return a message m' to be given to C_v. The game returns the result of predicate P on the history σ.

We define two classes of adversaries: *full* and *benign*, that we use in the security definition.

Full adversaries: A *full* adversary $\mathcal{A}_{\mathrm{FULL}}$ invokes the oracles in any arbitrary order. The only restriction is the following: for each $v \in [1, u]$, after $\mathcal{A}_{\mathrm{FULL}}$ has invoked an operation of C_v (with INVOKE$(v, \cdot)$), then $\mathcal{A}_{\mathrm{FULL}}$ must not invoke another operation of C_v until after the operation completes (when RECEIVE$(v, \cdot)$ returns $r \neq \bot$). This condition is often called *well-formedness* and means that a single client does not run concurrent operations.

Benign adversaries: A *benign* adversary $\mathcal{A}_{\mathrm{BEN}}$ is restricted like $\mathcal{A}_{\mathrm{FULL}}$. Additionally, it makes no query to the CORRUPT oracle and obeys the following conditions:

```
Game $\mathbf{G}^{\mathrm{emu}}_{\mathrm{BEP},u,\mathrm{P}}(\mathcal{A})$
    $(clk_1, \ldots, clk_u, svk, pks)$
        $\leftarrow\!\$\ \mathrm{BEP.SETUP}(u)$
    $S_1, \ldots, S_u, S_{\mathsf{s}}, \sigma \leftarrow \epsilon$
    $\mathcal{A}^{\mathrm{INVOKE,RECEIVE,PROCESS,CORRUPT}}(pks)$
    Return $\neg\mathrm{P}(\sigma)$

INVOKE$(v, o)$
    $(m, S_v) \leftarrow\!\$\ \mathrm{BEP.INVOKE}(o, clk_v, pks, S_v)$
    $\sigma \leftarrow \sigma \circ (C_v, o)$
    Return $m$

RECEIVE$(v, m)$
    $(r, m', S_v) \leftarrow\!\$\ \mathrm{BEP.RECEIVE}(m, clk_v, pks, S_v)$
    $\sigma \leftarrow \sigma \circ (C_v, r)$
    Return $(r, m')$

CORRUPT
    Return $S_{\mathsf{s}}$

PROCESS$(v, m)$
    $(m', v', S_{\mathsf{s}})$
        $\leftarrow\!\$\ \mathrm{BEP.PROCESS}(m, v, svk, pks, S_{\mathsf{s}})$
    Return $(v', m')$
```

Fig. 7. The emulation game parametrized by a predicate P.

- For each output m obtained from INVOKE$(v, \cdot)$ or $(\_, m)$ obtained from RECEIVE$(v, \cdot)$, such that $m \neq \bot$, query the oracle PROCESS *exactly once* on (v, m). (Every protocol message is delivered to the server.)
- For each output (v, m) of PROCESS, query the oracle RECEIVE *exactly once* on (v, m). (Every server message to the client is delivered.)
- Never provide any inputs to those two types of oracles beyond those described above. (The inputs to INVOKE are not further restricted compared to the case of $\mathcal{A}_{\mathrm{FULL}}$.)
- Client operations are executed sequentially, that is, after each operation o is INVOKEd, $\mathcal{A}_{\mathrm{BEN}}$ only uses oracles RECEIVE and PROCESS until RECEIVE outputs some $r \neq \bot$; it may only afterwards submit a next operation with INVOKE.

The definition of security consists of two conditions, which are made formal in Definition 4. The first condition models the security of the protocol against malicious servers, and uses the concept of fork linearizability as defined in Section 2. In more detail, we use a predicate $\mathrm{fork}_{F'}$ on histories that determines whether the history σ is fork linearizable with respect to the abortable type F', and the advantage of a full adversary $\mathcal{A}_{\mathrm{FULL}}$ is defined as the probability of producing a history that is not fork-linearizable. The second condition formalizes linearizability with respect to benign adversaries $\mathcal{A}_{\mathrm{BEN}}$ and is defined with respect to a predicate $\mathrm{lin}_{F'} \wedge \mathrm{live}_{F'}$ that formalizes both linearizability and liveness.

Definition 4. *Let* BEP *be a protocol and* F *an abstract data type. The FLBE-advantages of* $\mathcal{A}_{\mathrm{FULL}}$ *w.r.t.* BEP *and* F *is defined as follows. Let* $\mathrm{fork}_{F'}$ *denote the predicate on histories that formalizes fork linearizability* *with respect to F'. Then*

$$\mathrm{Adv}^{\mathrm{FL}}_{\mathrm{BEP},u}\,(\mathcal{A}_{\mathrm{FULL}}) := \Pr\left[\mathbf{G}^{\mathrm{emu}}_{\mathrm{BEP},u,\mathrm{fork}_{F'}}(\mathcal{A}_{\mathrm{FULL}}) = 1\right] \; . \tag{4}$$

The linearizability advantage of $\mathcal{A}_{\mathrm{BEN}}$ *is defined as follows, using the predicate* lin_F *that formalizes linearizability with respect to F, and* live_F *that formalizes that no operations abort:*

$$\mathrm{Adv}^{\mathrm{LIN}}_{\mathrm{BEP},u}\,(\mathcal{A}_{\mathrm{BEN}}) := \Pr\left[\mathbf{G}^{\mathrm{emu}}_{\mathrm{BEP},u,\mathrm{lin}_F \wedge \mathrm{live}_F}(\mathcal{A}_{\mathrm{BEN}}) = 1\right] \; . \tag{5}$$

The predicates $\mathrm{fork}_{F'}$ and lin_F are easily made formal following the descriptions in Section 2. The predicate live_F simply formalizes that for every operation $o \in \sigma$ there is a corresponding output event.

6 A lock-step protocol for emulating shared data types

We describe a *lock-step* protocol that uses an ADT to give multiple clients access to a data type F, and achieves fork linearizability through the use of vector clocks [19, 37, 38, 14] in a setting where the server may be malicious. By *lock-step* we mean that while the server processes the request of one client, all other clients will be blocked. We prove the security of the scheme based on the unforgeability of the underlying signature scheme and the soundness of the underlying ADT. We remark that the VICOS protocol [10], which is also based on ADTs, achieves better efficiency by exploiting commutative properties of operations to prevent blocking. That protocol has, however, not (yet) been formally proven; providing a similar proof for that protocol is planned as future work.

The lock-step protocol LS, which is specified in detail in Figure 8, has a setup phase in which the keys of the ADT and one signature key pair for each client are generated and distributed. The processing then works as follows. A client C_i initiates an operation o by calling LS.INVOKE, which generates an SUBMIT message with o for the server. When this message is delivered to the server and the server is not busy processing another request, then it generates a REPLY message for the client. The client performs local computation, generates a COMMIT message for the server, finally completes the operation by returning the output r.

Authenticated data types ensure the validity of each individual operation invoked by a client. After the client submits the operation o, the server executes o via ADT.EXEC and returns the proof π together with the previous authenticator to the client in REPLY. The client then verifies the

```
$\text{LS.}\textsc{setup}(n, \lambda)$
$(sk, ad, a) \leftarrow\!\!\$\ \text{ADT.}\textsc{init}(\lambda)$
For $v = 1$ to $u$ do $(ssk_v, spk_v) \leftarrow\!\!\$\ \text{DS.}\textsc{keygen}(\lambda)$
Return $((ssk_1, sk, 1), \ldots, (ssk_u, sk, u), ad, (spk_1, \ldots, spk_u, a))$

$\text{LS.}\textsc{invoke}(o_v, clk_v, pks, T)$
If $s = \epsilon$ then $T \leftarrow (0, \ldots, 0)$          ▷ Obtain number of users from $pks$
Return $(\langle \textsc{submit}, o_v \rangle, T)$

$\text{LS.}\textsc{receive}(m, (ssk_v, sk, v), (spk_1, \ldots, spk_u, a_0), T)$
If $m = \langle \textsc{busy} \rangle$ then return $(\textsc{busy}, \bot, T)$
$\langle \textsc{reply}, V, \ell, a, \varphi', \xi \rangle \leftarrow m$ (or abort in not possible)
$(b, r, a', t) \leftarrow \text{ADT.}\textsc{verify}(sk, a, o_v, \xi)$
$b \leftarrow b \wedge ((V = (0, \ldots, 0) \wedge a = a_0) \vee \text{DS.}\textsc{verify}(spk_\ell, \varphi', \textsc{commit} \circ a \circ V))$
If $\neg ((T \leq V) \wedge (T[v] = V[v]) \wedge b)$ then return $(\textsc{abort}, \bot, T)$
$T \leftarrow V + 1_v$
$\varphi \leftarrow \text{DS.}\textsc{sign}(ssk_v, \textsc{commit} \circ a' \circ T)$
Return $(r, \langle \textsc{commit}, T, a', \varphi, t \rangle, T)$

$\text{LS.}\textsc{process}(m, v, ad_0, pks, s)$
If $s = \epsilon$ then $s \leftarrow (ad, a, 0, \epsilon, (0, \ldots, 0), 0)$          ▷ Initialize server state
$(ad, a, \ell, \omega, V, i) \leftarrow s$
If $i = 0$ and $m = \langle \textsc{submit}, o \rangle$ then          ▷ Expect a submit message
  $\pi \leftarrow \text{ADT.}\textsc{exec}(ad, o)$
  Return $(v, \langle \textsc{reply}, V, \ell, a, \omega, \pi \rangle, (ad, a, \ell, \omega, V, v))$
Else if $i = v$ and $m = \langle \textsc{commit}, T, a', \varphi, t \rangle$ then          ▷ Expected commit
  $ad' \leftarrow \text{ADT.}\textsc{refresh}(ad, a, o, t)$
  Return $(0, \bot, (ad', a', i, \varphi, T, 0))$
Else return $(v, \langle \textsc{busy} \rangle, s)$
```

Fig. 8. The lock-step protocol LS.

server's computation against the previous authenticator, computes the output and the new authenticator via ADT.VERIFY, and sends them to the server in COMMIT. Finally, the new authenticator and the authentication token of the ADT are sent to the server, which computes the new state via ADT.REFRESH.

Digital signatures are used in the protocol to authenticate the information that synchronizes the protocol state among the clients. After computing a new authenticator a' via ADT.VERIFY, a client signs a' and sends it back to the server in COMMIT. When the next client initiates an operation o, the REPLY message from the server contains the authenticator a' together with the signature. Checking the validity of this signature ensures that all operations are performed on a valid (though possibly outdated) state.

Vector clocks are a mechanism often used in distributed systems to synchronize events between different systems in a network [4]. For clients $C_1, \ldots, C_u$, a logical clock is described by a vector $V \in \mathbb{N}^u$, where the v-th component $V[v]$ contains the logical time of C_v. In the protocol LS, the clients increase their local logical with each operation they perform; the vector clock therefore ensures a partial order on the events. The clients in the protocol use the vector clock to ensure that all events they observe are totally ordered by updating their vector clock accordingly, and signing and communicating it together with the updated authenticator. Together with the above mechanism, this ensures that the only attack that is feasible for a server is partitioning the client set and *forking* the execution, leading to a disjoint but internally consistent execution for each branch.

We prove in the next theorem and that the protocol achieves fork linearizability if the signature scheme and the ADT satisfy the security notions described in the previous sections.

Theorem 3. *The protocol described above emulates the abortable type F' on a Byzantine server with fork linearizability. Furthermore, if the server is correct, then all histories of the protocol are linearizable w.r.t. F.*

More formally, let $\mathcal{A}$ be an adversary in the game $\mathbf{G}^{\mathrm{emu}}_{\mathrm{LS},u,\mathrm{fork}_{F'}}$. There exist adversaries $\mathcal{B}$ and $\mathcal{C}$, described explicitly in the proof, such that

$$\mathrm{Adv}^{\mathrm{FL}}_{\mathrm{LS},u}(\mathcal{A}) \leq u \cdot \mathrm{Adv}^{\mathrm{EUF}}_{\mathrm{DS}}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{SOUND}}_{\mathrm{ADT}}(\mathcal{C}) \ . \tag{6}$$

On a high level, the proof proceeds as follows. We first perform game hops in which we idealize the guarantees of the signature scheme DS and the ADT scheme ADT used by protocol LS. We then show that the history σ produced in the game with idealized cryptography is fork-linearizable. We start by idealizing the guarantees of the signature scheme.

Lemma 1. *For u users and every adversary $\mathcal{A}$ there exists an adversary $\mathcal{B}$, explicitly described in the proof, such that*

$$\mathrm{Adv}^{\mathrm{FL}}_{\mathrm{LS},u}(\mathcal{A}) \leq \Pr[\mathbf{G}_1] + u\mathrm{Adv}^{\mathrm{EUF}}_{\mathrm{DS}}(\mathcal{B}) \ . \tag{7}$$

If $\mathcal{A}$ makes q calls to RECEIVE, *then $\mathcal{B}$ makes at most q calls to* SIGN.

6.1 Detailed descriptions of games

The game $\mathbf{G}_1$ referenced in Lemma 1 is specified in Figure 9 and is almost the same as $\mathbf{G}^{\mathrm{emu}}_{\mathrm{LS},u,\mathrm{fork}_{F'}}$, but it performs an *idealized* check on the signature scheme, that is, clients only accept signatures that have been produced by other honest clients.

```
Game $\mathbf{G}_0$ $\boxed{\mathbf{G}_1}$
$(clk_1, \ldots, clk_u, svk, pks) \leftarrow\!\!\$\ \mathrm{LS}.\textsc{setup}(u)$
$T_1, \ldots, T_u, S_\mathsf{s}, \sigma \leftarrow \epsilon$
$\mathcal{A}^{\textsc{invoke},\textsc{receive},\textsc{process},\textsc{corrupt}}(pks)$
Return $\neg\mathrm{fork}_{F'}$

$\underline{\textsc{invoke}(v, o_v)}$
$(m, T') \leftarrow\!\!\$\ \mathrm{LS}.\textsc{invoke}(o_v, clk_v, pks, T_v)$
$\sigma \leftarrow \sigma \circ (C_v, o_v)$
Return $m$

$\underline{\textsc{receive}(v, m)}$
If $m = \langle\textsc{busy}\rangle$ then return $(\textsc{busy}, \bot)$
Parse $m$ as $\langle\textsc{reply}, V, \ell, a, \varphi', \xi\rangle$
$(b, r, a', t) \leftarrow \mathrm{ADT}.\textsc{verify}(sk, a, o_v, \xi)$
$b \leftarrow b \wedge ((V = (0, \ldots, 0) \wedge a = a_0) \vee \mathrm{DS}.\textsc{verify}(spk_\ell, \varphi', \textsc{commit} \circ a \circ V))$
If $\mathrm{DS}.\textsc{verify}(spk_\ell, \varphi', \textsc{commit} \circ a \circ V) \wedge (\ell, \textsc{commit} \circ a \circ V) \notin \mathcal{S}$ then
    $\textsc{bad} \leftarrow \textsc{True}$ ; $\boxed{b \leftarrow \textsc{False}}$
If $\neg\,((T_v \leq V) \wedge (T_v[v] = V[v]) \wedge b)$ then return $(\textsc{abort}, \bot)$
$T_v \leftarrow V + 1_v$
$\varphi \leftarrow \mathrm{DS}.\textsc{sign}(ssk_v, \textsc{commit} \circ a' \circ T)$ ; $\mathcal{S} \leftarrow \mathcal{S} \cup \{(v, \textsc{commit} \circ a' \circ T)\}$
$\sigma \leftarrow \sigma \circ (C_v, r)$
Return $(r, \langle\textsc{commit}, T, a', \varphi, t\rangle)$

$\underline{\textsc{corrupt}}$
Return $S_\mathsf{s}$

$\underline{\textsc{process}(v, m)}$
$(m', v', S_\mathsf{s}) \leftarrow\!\!\$\ \mathrm{LS}.\textsc{process}(m, v, svk, pks, S_\mathsf{s})$
Return $(v', m')$
```

Fig. 9. Idealization of the authenticity guarantee provided by the signature scheme.

Proof. In comparison with game $\mathbf{G}^{\mathrm{emu}}_{\mathrm{LS},u,\mathrm{fork}_{F'}}(\mathcal{A})$, game $\mathbf{G}_0$ described in Figure 9 is modified in two ways. First, whenever a client signature is created on a message $\textsc{commit} \circ a' \circ Y$, this event message also recorded in the set $\mathcal{S}$ together with the identifier ℓ of the client that signed the message. Second, the game is modified where the signatures are checked. If a signature verifies but no corresponding entry exists in the set $\mathcal{S}$, the flag BAD is set. This does not change the adversary's advantage in winning the game.

We then define the next game $\mathbf{G}_1$; the only difference between $\mathbf{G}_0$ and $\mathbf{G}_1$ is after BAD $\leftarrow$ TRUE, where $\mathbf{G}_1$ is defined to not accept forged signatures even if they verify. By the Fundamental Lemma of Game Playing [7], the difference in advantage between those two games can be bounded by the advantage of the adversary in provoking BAD to be set. Then we construct adversary $\mathcal{B}$ for the existential unforgeability game of the signature scheme as follows. Adversary $\mathcal{B}$ initially chooses $v \in \{1, \ldots, u\}$ uniformly at random, and then emulates game $\mathbf{G}_0$, using the signature verification key from the game $\mathbf{G}_{\mathrm{DS}}^{\mathrm{euf}}$ for user v and simulated key pairs for all other users. Queries to oracle RECEIVE of client v are performed using the SIGN oracle in game $\mathbf{G}_{\mathrm{DS}}^{\mathrm{euf}}$. For the flag BAD to be set, adversary $\mathcal{A}$ has to produce a signature φ' such that $\mathrm{DS.VERIFY}(spk_\ell, \textsc{commit} \circ a \circ V, \varphi')$ but $(\ell, \textsc{commit} \circ a \circ V) \notin \mathcal{S}$, i.e., the message $\textsc{commit} \circ a \circ V$ has not been signed through a call to RECEIVE, and therefore has not been queried to SIGN. Consequently, φ' can be used by $\mathcal{B}$ to win the signature game.

Since the emulation of game $\mathbf{G}_0$ is perfect, if $\mathcal{A}$ forges such that BAD is set, then the probability that the forgery was for user v is $1/u$. Using the fundamental lemma then concludes the proof. □

The next step is to idealize the guarantee provided by the ADT. We prove the game hop by reduction from the soundness of the ADT.

Lemma 2. *For each adversary $\mathcal{A}$ there exists an adversary $\mathcal{C}$, explicitly described in the proof, such that*

$$\Pr[\mathbf{G}_2] \leq \Pr[\mathbf{G}_3] + \mathrm{Adv}_{\mathrm{ADT}}^{\mathrm{SOUND}}(\mathcal{C}) \ . \tag{8}$$

If $\mathcal{A}$ makes q calls to its RECEIVE *oracle, then $\mathcal{C}$ makes at most q calls to its* VERIFY *oracle.*

Proof. Game $\mathbf{G}_2$, formally described in Figure 10, differs from $\mathbf{G}_1$ as follows. In oracle RECEIVE, the (ideal) state of the computation is tracked using map $L[\ ]$, which maps a version vector V to an *ideal* state of F associated to that version, and by *ideally* computing the functionality F whenever the verification in ADT.VERIFY succeeds. Using this ideal representation, the game checks whether the output r given to the client is correct, analogously to $\mathbf{G}_{\mathrm{ADT}}^{\mathrm{sound}}$, and sets BAD to TRUE the client accepts although the output is wrong (i.e., ADT has been broken).

Game $\mathbf{G}_3$ differs from $\mathbf{G}_2$ in that the client will not accept in the above described case; we simply set b to FALSE, which leads to the client rejecting the output. As this modifies the behavior of the game only after BAD is set, we can use the Fundamental Lemma to bound the difference in adversary advantage by the probability of provoking BAD to be set.

We now describe an adversary $\mathcal{C}$ that plays game $\mathbf{G}_{\mathrm{ADT}}^{\mathrm{sound}}$, emulates game $\mathbf{G}_2$ to $\mathcal{A}$, and wins $\mathbf{G}_{\mathrm{ADT}}^{\mathrm{sound}}$ in case $\mathcal{A}$ provokes BAD to be set. This

```
Game $\mathbf{G}_2$ $\boxed{\mathbf{G}_3}$
$(clk_1, \ldots, clk_u, svk, pks) \leftarrow\!\$\ \mathrm{LS.}\textsc{setup}(u)$
$T_1, \ldots, T_u, S_{\mathsf{s}}, \sigma \leftarrow \epsilon\ ;\ L \leftarrow [\ ]$
$\mathcal{A}^{\textsc{invoke},\textsc{receive},\textsc{process},\textsc{corrupt}}(pks)$
Return $\neg\mathrm{fork}_{F'}(\sigma)$

$\textsc{invoke}(v, o_v)$
$(m, T') \leftarrow\!\$\ \mathrm{LS.}\textsc{invoke}(o_v, clk_v, pks, T_v)$
$\sigma \leftarrow \sigma \circ (C_v, o_v)$
Return $m$

$\textsc{receive}(v, m)$
If $m = \langle \textsc{busy} \rangle$ then return $(\textsc{busy}, \bot)$
Parse $m$ as $\langle \textsc{reply}, V, \ell, a, \varphi', \xi \rangle$
$(b, r, a', t) \leftarrow \mathrm{ADT.}\textsc{verify}(sk, a, o_v, \xi)$
$b \leftarrow b \wedge \Big( (V = (0, \ldots, 0) \wedge a = a_0)$
        $\vee \big(\mathrm{DS.}\textsc{verify}(spk_\ell, \varphi', \textsc{commit} \circ a \circ V) \wedge (\ell, \textsc{commit} \circ a \circ V) \in \mathcal{S}\big)\Big)$
If $b$ then $s \leftarrow L[V]\ ;\ (s', \tilde{r}) \leftarrow F(s, o_v)\ ;\ L[V + 1_v] \leftarrow s'$
If $b$ and $\tilde{r} \neq r$ then $\textsc{bad} \leftarrow \textsc{True}$ ; $\boxed{b \leftarrow \textsc{False}}$
If $\neg\left((T_v \leq V) \wedge (T_v[v] = V[v]) \wedge b\right)$ then return $(\textsc{abort}, \bot)$
$T_i \leftarrow V + 1_v$
$\varphi \leftarrow \mathrm{DS.}\textsc{sign}(ssk_v, \textsc{commit} \circ a' \circ T)\ ;\ \mathcal{S} \leftarrow \mathcal{S} \cup \{(v, \textsc{commit} \circ a' \circ T)\}$
$\sigma \leftarrow \sigma \circ (C_v, r)$
Return $(r, \langle \textsc{commit}, T, a', \varphi, t \rangle)$

$\textsc{corrupt}$
Return $S_{\mathsf{s}}$

$\textsc{process}(v, m)$
$(m', v', S_{\mathsf{s}}) \leftarrow\!\$\ \mathrm{LS.}\textsc{process}(m, v, svk, pks, S_{\mathsf{s}})$
Return $(v', m')$
```

Fig. 10. Idealization of the authenticity guarantee provided by the ADT.

adversary $\mathcal{C}$ obtains (ad, a) from $\mathbf{G}^{\text{sound}}_{\text{ADT}}$ and emulates $\mathbf{G}_2$ to $\mathcal{A}$. For an oracle query $\textsc{receive}(v, m)$ with $m = \langle \textsc{reply}, V, \ell, a, \varphi', \xi \rangle$, instead of calling $\mathrm{ADT.}\textsc{verify}(sk, a, o_v, \xi)$, $\mathcal{C}$ queries (o_v, j, ξ) to its VERIFY oracle, where j is the number of the VERIFY-query that corresponds to version V (which is the query in which the digest a was generated by the ideal guarantee of the signature scheme).

Adversary $\mathcal{C}$ emulates $\mathbf{G}_2$ perfectly, and that setting the flag BAD in $\mathbf{G}_2$ corresponds to winning the game $\mathbf{G}^{\text{sound}}_{\text{ADT}}$. ☐

The previous lemmas allow us to now work in game $\mathbf{G}_3$, where the cryptographic schemes are idealized, meaning that all messages accepted by the signature verification have actually been signed before, and all outputs accepted by the ADT verification are correct. Our goal is to now prove that the history σ that is generated in $\mathbf{G}_3$ is indeed fork linearizable. As in previous work [14], we assume that all initiated operations in σ have completed. To show fork linearizability, we define subsequences σ_v of σ as follows.

1. All operations $o \in \sigma$ executed at client C_v are also contained in σ_v,
2. for each $o \in \sigma_v$, include also all $o' \in \sigma$ with associated version number less than or equal to that of o.

Then, we obtain $\pi_v(\sigma_v)$ by sorting all the operations in σ_v according to:

1. by the ascending order of their associated version vectors,
2. by their real-time order,
3. by the real-time order of their completion event.

Before moving to the proof of Theorem 3, we prove another lemma which states that π_v preserves the real-time order of the history σ_v.

We also introduce further conventions and notation. First, we assume that each operation $o \in \sigma$ is unique; this simplifies the notation and is easy to achieve by including the client identifier and local timestamp as part of o. For an operation $o \in \sigma$, we then write $\mathsf{ver}(o)$ to denote the version vector that the client assigns to operation o in the COMMIT message. The following lemma shows that π_v preserves the real-time order, and is an extension of the corresponding argument in [14].

Lemma 3. *The permutation π_v preserves the real-time order of the history σ_v. This means that for two operations $o, o' \in \sigma_v$, if $o \prec_{\sigma_v} o'$, then also $o \prec_{\pi_v(\sigma_v)} o'$.*

Proof. The proof starts by showing several helper statements. First, by the fact that client $C_{v'}$ is honest, we show that there is at most one operation $o \in \sigma_v$ in which a certain increase in position v' of the version vector occurs. More formally, for each $j \in \mathbb{N}$, there is at most one operation $o \in \sigma_v$ invoked by client $C_{v'}$ with $\mathsf{ver}(o)[v'] = j$.

Claim. Let $v', j \in \mathbb{N}$ and $C_{v'}$ be a client. In history σ_v of some C_v, there is at most one operation $o \in \sigma$ of client $C_{v'}$ with $\mathsf{ver}(o)[v'] = j$.

Proof. This is since client $C_{v'}$ increases the version vector at position v' during each invocation, and the version vector it starts from is greater than or equal to the one computed in the previous invocation. $\square$

We then show that each operation has a "parent" whose version vector differs only in one position.

Claim. Let $o \in \sigma_v$ be a (complete) operation performed by some client $C_{v'}$. If $|\mathsf{ver}\,(o)| > 1$, then there is exactly one (complete) operation $o' = \mathsf{par}_\sigma\,(o) \in \sigma$ that immediately precedes it in the sense that the signature φ' received by the client as part of the REPLY-message was generated by some client $C_{v''}$ during the RECEIVE query related to operation o'. Then $o' \in \sigma_v$ and $\mathsf{ver}\,(o') < \mathsf{ver}\,(o)$, with the two vectors differing exactly by 1 in the v'-th component and $o \not\prec_{\sigma_v} o'$.

Proof. First, since $o \in \sigma_v$ completes and $|\mathsf{ver}\,(o)| > 1$, the verification of signature φ' in LS.RECEIVE has succeeded. As the client that generated signature φ' generates signatures only for the purpose of committing operations, there is a tuple $(\ell, \text{COMMIT} \circ x \circ V) \in \mathcal{S}$ that was inserted into $\mathcal{S}$ when some operation $o' \in \sigma$ was processed.[6] The statement about the versions follows via the definition of the protocol (in particular the line $T \leftarrow V + 1_{v'}$). The construction of σ_v then also implies that $o' \in \sigma_v$. As o uses the signature φ' generated during o', we obviously have $o \not\prec_{\sigma_v} o'$. □

The same argument can be applied iteratively, to build a sequence of "ancestors" of the operation o, by including grandparents and great-grandparents and so on. We omit the obvious proof.

Claim. Define $\mathsf{anc}\,(o,\sigma) \coloneqq (o_1,\ldots,o_m)$ with $o_m \coloneqq o$, $o_j = \mathsf{par}_\sigma\,(o_{j+1})$, and $|\mathsf{ver}\,(o_1)| = 1$. This is a sequence of operations in σ_v such that the version numbers of each o_j and o_{j+1} differ only in one position, and by 1. Furthermore, $o_j \not\prec_{\sigma_v} o_{j'}$ for all $j \geq j'$.

Finally, we leverage the above statement to conclude that if there are operations o, o' such that the version vector of o' is smaller than that of o, then o' must indeed appear in the ancestry of o.

Claim. Let $o, o' \in \sigma$ with $\mathsf{ver}\,(o) \geq \mathsf{ver}\,(o')$. Then $o' \in \mathsf{anc}\,(o,\sigma)$.

Proof. We know that $\mathsf{ver}\,(o) \geq \mathsf{ver}\,(o')$. Let C_v be the client that invoked o', then by the second claim, this implies that $\mathsf{ver}\,(\mathsf{par}_\sigma\,(o'))\,[v] < \mathsf{ver}\,(o')\,[v] \leq \mathsf{ver}\,(o)\,[v]$. The third claim then guarantees that in $\mathsf{anc}\,(o,\sigma)$ there is an operation $\hat{o}$ where the same increase in version number of client C_v occurs, and the first claim implies that $\hat{o} = o'$. Therefore, $o' \in \mathsf{anc}\,(o,\sigma)$. □

---

[6] o' is uniquely determined since the party's component in the time stamp V increases during each RECEIVE, so there can only be one operation with this value V.

Now we go on to show that $\pi_v(\sigma_v)$ indeed preserves the real-time order of σ_v. Since $o \prec_{\sigma_v} o'$, the first two rules of the description of $\pi_v(\sigma_v)$ ensure that o precedes o' in $\pi_v(\sigma_v)$ unless $\mathsf{ver}(o) > \mathsf{ver}(o')$. Assume that $\mathsf{ver}(o) > \mathsf{ver}(o')$. Then, by the final claim above, we know that $o' \in \mathsf{anc}(o, \sigma)$ and by the third claim this also means $o \not\prec_{\sigma_v} o'$. This contradicts the precondition, therefore $\mathsf{ver}(o) \not> \mathsf{ver}(o')$ and therefore $o \prec_{\pi_v(\sigma_v)} o'$. This proves the lemma. □

We now proceed to the proof of Theorem 3.

Proof (of Theorem 3). We first use Lemmas 1 and 2 to observe

$$\mathrm{Adv}^{\mathrm{FL}}_{\mathrm{LS},u}(\mathcal{A}) \leq \Pr[\mathbf{G}_3] + \mathrm{Adv}^{\mathrm{SOUND}}_{\mathrm{ADT}}(\mathcal{C}) + u\mathrm{Adv}^{\mathrm{EUF}}_{\mathrm{DS}}(\mathcal{B}) \ ,$$

and then we prove that $\Pr[\mathbf{G}_3] = 0$. In particular, we prove fork linearizability of the by showing all required conditions for the constructed σ_v and $\pi_v(\sigma_v)$. First, all complete operations $o \in \sigma$ occurring at client C_v are contained in σ_v and $\pi_v(\sigma_v)$ by construction. Second, $\pi_v(\sigma_v)$ preserves the real-time order of σ_v by Lemma 3. The third condition is that the operations of $\pi_v(\sigma_v)$ satisfy the sequential specification of F.

Claim. The operations of $\pi_v(\sigma_v)$ satisfy the sequential specification of F.

Proof. Let $o \in \pi_v(\sigma_v)$. Every $o' \in \mathsf{anc}(o, \sigma)$ is also $o' \in \pi_v(\sigma_v)$—this follows from the third claim in the proof of Lemma 3. Let o be the last operation in $\pi_v(\sigma_v)$ executed by C_v, then for every $o' \in \pi_v(\sigma_v)$ it is $\mathsf{ver}(o') \leq \mathsf{ver}(o)$ by construction of $\pi_v(\sigma_v)$ and—by the final claim of Lemma 3—also $o' \in \mathsf{anc}(o, \sigma)$. Then $\pi_v(\sigma_v)$ and $\mathsf{anc}(o, \sigma)$ contain the same operations, and by construction they also have the same order.

We then continue to show the statement by induction for all elements in $\pi_v(\sigma_v)$. If $\pi_v(\sigma_v)$ is empty then we are done, otherwise there is a first operation $o_1 \in \pi_v(\sigma_v)$. In particular, the associated version vector contains only a single entry that is non-zero (and in particular it is 1), because otherwise—by the second claim of Lemma 3—there would exist another operation $o' \in \pi_v(\sigma_v)$ with a smaller non-zero version vector. In particular, the client invoking o_1 checks that $a = a_0$ and verifies that the server has computed $F(s_0, o_1) = (s_1, r_1)$ correctly. Therefore, the first operation is computed according to the specification of F.

For any subsequent operation $o \in \pi_v(\sigma_v)$, let $o' = \mathsf{par}_{\sigma}(o) \in \pi_v(\sigma_v)$. By the construction of $\pi_v(\sigma_v)$ it holds that $o' \prec_{\pi_v(\sigma_v)} o$, and by the fact that $\pi_v(\sigma_v) = \mathsf{anc}(o, \sigma)$ it also holds that there is no $\tilde{o} \in \sigma_v$ with $o' \prec_{\pi_v(\sigma_v)} \tilde{o} \prec_{\pi_v(\sigma_v)} o$. Let $C_{v'}$ be the client that executes o. The fact that operations with the same version vectors $\leq \mathsf{ver}(o)$ are unique in σ—by the final claim in Lemma 3—means that the signature φ' verified during o was generated in o', and that o is evaluated on the state s_{j-1} computed by F during o'. This means that $C_{v'}$ verifies that $(s_j, r_j) = F(s_{j-1}, o)$ is computed correctly by the server. This concludes the proof that $\pi_v(\sigma_v)$ satisfies the sequential specification of F. ☐

To complete the proof, we have to show that for every $o \in \pi_v(\sigma_v) \cap \pi_{v'}(\sigma_{v'})$, the sequence of events that precede o in $\pi_v(\sigma_v)$ is the same as the sequence of events that precede o in $\pi_{v'}(\sigma_{v'})$. This, however, holds by a similar argument as in the beginning of the proof of the above claim. Indeed, for any operation $o \in \pi_v(\sigma_v) \cap \pi_{v'}(\sigma_{v'})$, it holds that $\pi_v(\sigma_v)|_o = \mathsf{anc}(o, \sigma) = \pi_{v'}(\sigma_{v'})|_o$, which concludes the proof of fork linearizability.

What remains to be shown is the statement that the history is indeed linearizable if the server is correct, meaning for benign adversaries $\mathcal{A}_{\text{BEN}}$. We show this via a permutation $\pi(\sigma)$ of σ by sorting the events as follows:

1. by the ascending order of their associated version vectors,
2. by their real-time order,
3. by the real-time order of their completion event.

As $\mathcal{A}_{\text{BEN}}$ delivers all messages faithfully, correctness of DS and ADT is sufficient to work with a history σ in which we assume the cryptographic schemes to be perfect (as in $\mathbf{G}_3$ before).

The permutation $\pi(\sigma)$ preserves the real-time order of σ; this follows by exactly the same arguments as in Lemma 3. Permutation $\pi(\sigma)$ satisfies the sequential specification of F; this follows as the above claim by observing that, with an honest server, the history never “forks” and $\pi(\sigma)$ is strictly ordered by the version vectors of the operations. ☐

7 Conclusion

Our work combines and extends three different lines of work. The first one is work on SNARKs and verifiable computation, where we build on the scheme of Fiore et al. [20] and extend it by a mechanism that allows for stateful computation in which an *untrusted* party can update the state in a verifiable manner in a multi-client setting. The second one is the work on authenticated data types, where many schemes have been proposed for different scenarios and for various *specific* data types. We develop a scheme for the so-called two-party setting which is *generic* in that it works for *all* data types where the operations can be efficiently computed, and adds only little overhead over the methods of [20]. Third, we extend the work on Byzantine-emulation protocols from the distributed-systems literature by providing a model for *computational* security that allows us to prove the protocol with the actual cryptography (instead of resorting to a model with idealized cryptography such as, e.g., [14, 13]). We describe a protocol based on ADTs (similarly to [10]) in which the computation is performed by an untrusted server and the clients are only required to store a short authenticator, but can still verify the correctness of the server's operations. We prove that our protocol achieves fork linearizability.

Acknowledgments

This work has been supported in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreements 644371 WITDOM and 644579 ESCUDO-CLOUD and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts 15.0098 and 15.0087. The work by Esha Ghosh was supported in part by NSF grant CNS-1525044.

References

1. Aguilera, M.K., Frölund, S., Hadzilacos, V., Horn, S.L., Toueg, S.: Abortable and query-abortable objects and their efficient implementation. In: ACM PODC. pp. 23–32 (2007)
2. Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: ISC 2001. pp. 379–393 (2001)
3. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: ACM CCS. ACM (2007)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. Wiley, second edn. (2004)
5. Backes, M., Fiore, D., Reischuk, R.M.: Verifiable delegation of computation on outsourced data. In: ACM CCS. pp. 863–874 (2013)
6. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO. LNCS, vol. 773, pp. 232–249. Springer (1993)
7. Bellare, M., Rogaway, P.: Code-based game-playing proofs and the security of triple encryption. In: Vaudenay, S. (ed.) EUROCRYPT. LNCS, vol. 4004. Springer (2006)
8. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for c: Verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J. (eds.) Advances in Cryptology — CRYPTO 2013. LNCS, vol. 8043 (2013)
9. Benabbas, S., Gennaro, R., Vahlis, Y.: Verifiable delegation of computation over large datasets. In: Advances in Cryptology - CRYPTO 2011. pp. 111–131 (2011)
10. Brandenburger, M., Cachin, C., Knežević, N.: Don't trust the cloud, verify: Integrity and consistency for cloud object stores. In: SYSTOR '15. ACM (2015)
11. Braun, B., Feldman, A.J., Ren, Z., Setty, S.T.V., Blumberg, A.J., Walfish, M.: Verifying computations with state. pp. 341–357 (2013)

12. Cachin, C., Keidar, I., Shraer, A.: Fork sequential consistency is blocking. Information Processing Letters 109(7), 360–364 (2009)
13. Cachin, C., Ohrimenko, O.: Verifying the consistency of remote untrusted services with commutative operations. In: OPODIS. LNCS, vol. 8878. Springer (2014)
14. Cachin, C., Shelat, A., Shraer, A.: Efficient fork-linearizable access to untrusted shared memory. In: ACM PODC. pp. 129–138. ACM (2007)
15. Canetti, R., Paneth, O., Papadopoulos, D., Triandopoulos, N.: Verifiable set operations over outsourced databases. In: PKC 2014. pp. 113–130 (2014)
16. Choi, S.G., Katz, J., Kumaresan, R., Cid, C.: Multi-client non-interactive verifiable computation. In: Theory of Cryptography. pp. 499–518 (2013)
17. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM 45(6), 965–981 (1998)
18. Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: Versatile verifiable computation. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE (2015)
19. Fidge, C.J.: Timestamps in message-passing systems that preserve the partial ordering. In: Raymond, K. (ed.) ACSC. pp. 56–66 (1988)
20. Fiore, D., Fournet, C., Ghosh, E., Kohlweiss, M., Ohrimenko, O., Parno, B.: Hash first, argue later: Adaptive verifiable computations on outsourced data. In: ACM CCS. pp. 1304–1316. ACM (2016)
21. Fiore, D., Gennaro, R.: Publicly verifiable delegation of large polynomials and matrix computations, with applications. In: ACM CCS. pp. 501–512 (2012)
22. Fiore, D., Mitrokotsa, A., Nizzardo, L., Pagnin, E.: Multi-key homomorphic authenticators. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology — ASIACRYPT. LNCS, vol. 10032, pp. 499–530. Springer (2016)
23. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: CRYPTO. vol. 6223 (2010)
24. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: EUROCRYPT. vol. 7881, pp. 626–645 (2013)
25. Ghosh, E., Goodrich, M.T., Ohrimenko, O., Tamassia, R.: Verifiable zero-knowledge order queries and updates for fully dynamic lists and trees. In: SCN. LNCS, vol. 9841, pp. 216–236 (2016)
26. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Aho, A.V. (ed.) ACM STOC. pp. 182–194. ACM (1987)
27. Goodrich, M.T., Papamanthou, C., Tamassia, R.: On the cost of persistence and authentication in skip lists. In: WEA. pp. 94–107 (2007)
28. Goodrich, M.T., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01. Proceedings (2001)
29. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Efficient authenticated data structures for graph connectivity and geometric search problems. Algorithmica 60(3), 505–552 (2011), `http://dx.doi.org/10.1007/s00453-009-9355-7`
30. Gordon, S.D., Katz, J., Liu, F., Shi, E., Zhou, H.: Multi-client verifiable computation with stronger security guarantees. In: TCC. pp. 144–168 (2015)
31. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: DISC. p. 522 (2003)
32. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Progr. Lang. Syst. 12(3), 463–492 (Jul 1990)
33. Juels, A., Kaliski, B.S.: PORs: proofs of retrievability for large files. In: ACM CCS. pp. 584–597. ACM (2007)

34. Li, J., Krohn, M., Mazières, D., Shasha, D.: Secure untrusted data repository (SUNDR). In: USENIX. p. 9. USENIX Association (2004)
35. Majuntke, M., Dobre, D., Serafini, M., Suri, N.: Abortable fork-linearizable storage. In: ACM PODC. pp. 255–269 (2009)
36. Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39, 21–41 (2004)
37. Mattern, F.: Virtual time and global states of distributed systems. In: Cosnard, M. (ed.) Proc. Workshop on Parallel and Distributed Algorithms. pp. 215–226 (1988)
38. Mazières, D., Shasha, D.: Building secure file systems out of byzantine storage. In: ACM PODC. pp. 108–117. ACM (2002)
39. Merkle, R.C.: A certified digital signature. In: CRYPTO. pp. 218–238 (1989)
40. Miller, A., Hicks, M., Katz, J., Shi, E.: Authenticated data structures, generically. In: ACM POPL. pp. 411–424 (2014)
41. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. TOS 2(2), 107–138 (2006)
42. Naor, M., Nissim, K.: Certificate revocation and certificate update. IEEE Journal on Selected Areas in Communications 18(4), 561–570 (2000)
43. Papadopoulos, D., Papadopoulos, S., Triandopoulos, N.: Taking authenticated range queries to arbitrary dimensions. In: ACM CCS. pp. 819–830 (2014)
44. Papamanthou, C.: Cryptography for Efficiency: New Directions in Authenticated Data Structures. Ph.D. thesis, Brown University (2011)
45. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: ACM CCS. pp. 437–448 (2008), `http://doi.acm.org/10.1145/1455770.1455826`
46. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal verification of operations on dynamic sets. In: CRYPTO. pp. 91–110 (2011)
47. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy (SP) (2013)
48. Tamassia, R.: Authenticated data structures. In: Algorithms - ESA 2003. pp. 2–5 (2003), `http://dx.doi.org/10.1007/978-3-540-39658-1_2`
49. Wahby, R.S., Setty, S.T.V., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: NDSS (2015)
50. Walfish, M., Blumberg, A.J.: Verifying computations without reexecuting them. Commun. ACM 58(2) (Feb 2015)
51. Williams, P., Sion, R., Shasha, D.: The blind stone tablet: Outsourcing durability to untrusted parties. In: NDSS (2009)
52. Zhang, Y., Katz, J., Papamanthou, C.: Integridb: Verifiable SQL for outsourced databases. In: ACM CCS. pp. 1480–1491 (2015)