Hash First, Argue Later

Adaptive Verifiable Computations on Outsourced Data

Dario Fiore
IMDEA Software Institute

Cédric Fournet
Microsoft Research

Esha Ghosh*
Brown University

Markulf Kohlweiss
Microsoft Research

Olga Ohrimenko
Microsoft Research

Bryan Parno
Microsoft Research

ABSTRACT

Proof systems for verifiable computation (VC) have the potential to make cloud outsourcing more trustworthy. Recent schemes enable a verifier with limited resources to delegate large computations and verify their outcome based on succinct arguments: verification complexity is linear in the size of the inputs and outputs (not the size of the computation). However, cloud computing also often involves large amounts of data, which may exceed the local storage and I/O capabilities of the verifier, and thus limit the use of VC.

In this paper, we investigate *multi-relation hash & prove schemes* for verifiable computations that operate on succinct data hashes. Hence, the verifier delegates both storage and computation to an untrusted worker. She uploads data and keeps hashes; exchanges hashes with other parties; verifies arguments that consume and produce hashes; and selectively downloads the actual data she needs to access.

Existing instantiations that fit our definition either target restricted classes of computations or employ relatively inefficient techniques. Instead, we propose efficient constructions that lift classes of existing arguments schemes for fixed relations to multi-relation hash & prove schemes. Our schemes (1) rely on hash algorithms that run linearly in the size of the input; (2) enable constant-time verification of arguments on hashed inputs; (3) incur minimal overhead for the prover. Their main benefit is to amortize the linear cost for the verifier across all relations with shared I/O. Concretely, compared to solutions that can be obtained from prior work, our new hash & prove constructions yield a 1,400x speed-up for provers. We also explain how to further reduce the linear verification costs by partially outsourcing the hash computation itself, obtaining a 480x speed-up when applied to existing VC schemes, even on single-relation executions.

*Work done at Microsoft Research.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978368

1. INTRODUCTION

Cryptographic proof systems let a verifier check that the computation executed by an untrusted prover was performed correctly [28]. These systems are appealing in a variety of scenarios, such as cloud computing, where a user outsources computations and wishes to verify their integrity given their inputs and outputs (I/O) [2, 36, 27, 25], or privacy-preserving applications, where a user owns sensitive data and wishes to release partial information with both confidentiality and integrity guarantees [42, 24]. Typically, these systems require the prover to perform considerable additional work to produce a proof that can be easily checked by the verifier.

Recent advances in verifiable computations have crossed an important practical threshold: verifying a proof given some I/O is faster than performing the computation locally [40, 7, 43, 45]. While these systems perform well when delegating computation-intensive algorithms, they do not help much with data-intensive applications, inasmuch as verification remains linear in the application's I/O.

Although some linear work is unavoidable when uploading data, ideally one would like to pay this price just once, rather than every time one verifies a computation that takes this data as input. This is particularly relevant for cloud computing on big data, where the verifier may not have enough local resources to encode and upload the whole database each time she delegates a query or, more generally, where many parties contribute data over a long period of time.

Approaches providing amortized verification do exist for limited classes of computations, such as data retrieval. For instance, the user may keep the root of a Merkle hash tree, and use it to verify the retrieved content. Unfortunately, as explained below, embeddings of this approach into generic proof systems incur large overheads for the prover.

Our goal is to enable practical verifiable computation for data-intensive applications. In particular, we wish to design schemes where verification time is independent of both the size of the delegated computations and the size of their I/O. Moreover, we wish to preserve the expressiveness of existing VC schemes (e.g., supporting NP relations) without adding to the prover's burden, which is already several orders of magnitude higher than the original computation.

Modelling Hash & Prove (HP) We first propose a model that captures the idea of hashing and uploading data once and then using the resulting hashes across multiple verifiable computations. In this model, the verifier needs only keep track of hashes, while the prover stores the corresponding data. The prover can use the data to perform computations and then (selectively) return results in plaintext to the verifier. As described below, hashes yield several benefits when delegating verifiable computations.

Flexible Reuse Hashes depend only on the data and are not tied to any particular computation. Hence, once a data hash is computed, it can be used to verify any computation that uses the corresponding data. It can also be used with different proof systems, as long as they rely on the same hash format.

Sharing Hashes are a compact representation of the data that can be easily shared and authenticated. Hence, verifiers can delegate computations on someone else's hashes, or chain multiple computations using intermediate hashes, without ever seeing or receiving the corresponding data.

Provenance A record of an input hash, an output hash, and a proof can serve as a succinct provenance token that can be easily and independently verified.

Confidentiality The verifier checks arguments on hashes of data that she may never see in plaintext; hence randomized hashes enable zero-knowledge arguments.

Updates If the hash mechanism also supports efficient updates, that is, given $\mathsf{hash}(x)$, one can compute $\mathsf{hash}(x')$ in time that depends only on the difference between x and x', then it also enables applications with dynamic data and streaming. For instance, a hashed database may be updated by uploading the new data and locally updating the hash.

Our hash & prove model extends non-interactive proof systems, with an intermediate hash algorithm between the input and the proof verification, and with the possibility of proving multiple relations. It is inspired by multi-function verifiable computation [41, 23], with relations instead of functions so that we can capture more general use cases, notably those where the prover provides its own (private) input to the computation.

Instantiating Hash & Prove Now equipped with a model for outsourcing multiple computations on authenticated data, we survey how existing work could be used to instantiate HP schemes. In particular, we observe that existing solutions have limitations either in efficiency or in generality.

Some prior work [11, 26, 5, 15] considers the idea of proving the correctness of a computation on data succinctly represented by a hash. This approach consists of encoding the verification of the hash as part of a relation for the underlying proof system. Namely, if $y = f(x)$ is the statement to be proved, then one actually proves an extended statement of the form $y = f(x) \wedge \sigma = \mathsf{hash}(x)$, essentially treating x as an additional witness. We henceforth refer to this method as an *inner encoding*. Inner encodings are simple and general, and can also be extended to more general data encodings such as Merkle trees or authenticated data structures (ADS) [44, 21]. On the other hand, inner encodings incur a significant overhead for the prover—indeed, unless hash is carefully tuned to the proof system, its verifiable evaluation on large inputs may dominate the prover costs.

Other works address reusability and succinct data representation by using different data encoding approaches that we will call *outer encodings*. The basic idea of outer encodings is that proofs are produced for the original statement, e.g., $y = f(x)$, and are linked to the encoded data x using some external mechanism. Works that can be explained under this approach are commit & prove schemes [33, 16, 19] and homomorphic authenticators [4, 17, 29]. While we discuss them in detail in §7, the main observation is that all these works fall short in generality; i.e., they limit the class of computations that can be executed on an hash value. While commit & prove schemes can achieve greater generality by using universal relations (as, e.g., in [5, 7, 9]), this typically entails a significant penalty in concrete efficiency.

New Hash & Prove Constructions Our main technical contributions are efficient, general HP constructions. Compared to general inner encoding solutions, ours incurs minimal overheads for the prover. Compared to prior outer encoding solutions, ours is fully general, in the sense that one can hash data first, without any restriction on the functions that may later be executed and verified on it.

We instantiate multi-relation hash & prove schemes both in the public and designated verifier settings. Our solutions are built in a semi-generic fashion by combining

(1) a verifiable computation (VC) or succinct non interactive argument (SNARG) scheme, and

(2) an HP scheme for simple, specific computations.

At a high level, our construction uses an *outer* data encoding, where general computation integrity is handled by (1), whereas data authentication and linking to the computation is handled by (2). As expected from an outer approach, this combination does not add any overhead in the use of (1), and the overhead introduced by (2) can be very low.

More specifically, for (1) we use any scheme where the input-processing part of the verification consists of a multi-exponentiation, that is, anything resembling a Pedersen commitment of the form $c_x = \prod_i F_i{}^{x_i}$, a property of virtually all modern, efficient SNARGs [40, 7, 19, 9, 31]. Our generic construction then outsources to the prover the original computation of (1) as well as the input-processing part of SNARG verification, $c_x = \prod_i F_i{}^{x_i}$. We then ask the prover to show the correctness of c_x using the auxiliary HP scheme (2). To this end, we only need a scheme that handles multi-exponentiation computations. We propose our own efficient constructions for such HP schemes. For the designated verifier setting, we adapt a multi-function VC scheme from prior work [23]. For public verifiability, we develop a new scheme, which requires new techniques to achieve adaptive security.

Our analysis in §6 shows that, in comparison to the inner encoding solution mentioned earlier, our HP scheme yields a $1,400\times$ speed-up for provers, as well as public (proving) keys that are shorter by the same factor.

Speeding up Hashing and Verification As mentioned above, VC schemes involve a verification effort linear in the size of the I/O. Concretely, this verification step is expensive because it relies on public-key operations (e.g., a few elliptic-curve multiplications for each word of I/O). With Hash & Prove, this linear work is first shifted to computing the hash, and then amortized across multiple computations, but the hash still has to be computed once.

When using inner encodings, one can choose standard, very efficient hash functions such as SHA2, which considerably reduces the effort of the verifier, at the expenses of the prover. Other trade-offs between verifier and prover costs are possible, e.g., by using algebraic hash constructions [1, 8, 15]. When using outer encodings, the choice of a hash function is more constrained. For instance, in Geppetto or in our HP scheme, the encoding still consists of a multi-exponentiation (i.e., n elliptic-curve multiplications where n is the size of the input).

As another contribution, we provide a technique to outsource such (relatively) expensive data encodings, at a moderate additional cost for the prover, while requiring only a trivial amount of linear work from the verifier: an arbitrary (fast) hash such as SHA2, and a few cheap field additions and multiplications, instead of elliptic curve operations. Concretely, this technique saves two orders of magnitude in verification time. It applies not only to our HP scheme, but also to existing VC systems [40, 19, 9].

Other Data Encodings In our presentation, we focus on plain hashes as a simple data encoding for all I/O, but many alternatives and variations are possible, depending on the needs of a given application. As a first example, the I/O can naturally be partitioned into several variables, each independently hashed and verified, to separate inputs from different parties, or with different live spans. (In a data-intensive application, for instance, one may use a hash for the whole database, and a separate hash for the query and its response.) More advanced examples include authenticated data structures, and more specific tools such as accumulators. To illustrate potential extensions of our work, we show that the HP model, and our generic HP construction, can be extended to work with such outer encodings. Concretely, we consider accumulators [37] and polynomial commitments [32], with set operations [38] and batch openings as restricted proof systems, respectively. By adapting our constructions, we obtain a new accumulate & prove system.

Contents The paper is organized as follows: §2 defines our notations, reviews assumptions we rely on, and recalls definitions of succinct non-interactive argument systems. §3 defines our hash & prove model, shows that some of the existing work satisfies it, and discusses their overhead for the prover. §4 presents our efficient HP construction and instantiates it for public and designated verifier settings. §5 presents the definition and construction of a hash & prove variant that supports hash outsourcing. §6 analyze the performance of our constructions. §7 discusses related work.

The full version [22] also includes auxiliary definitions, detailed proofs, and an extension of our work from hashes to cryptographic accumulators.

2. PRELIMINARIES

Notation. Given two functions $f, g : \mathbb{N} \to [0, 1]$ we write $f(\lambda) \approx g(\lambda)$ when $|f(\lambda) - g(\lambda)| = \lambda^{-\omega(1)}$. In other words, for all k, there exists an integer n_0 such that for all $\lambda > n_0$, we have $|f(\lambda) - g(\lambda)| < \frac{1}{\lambda^k}$. We say that f is negligible when $f(\lambda) \approx 0$.

Algebraic Tools and Complexity Assumptions. All our constructions make use of 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 use fixed groups for every value of the security parameter; this lets us compose schemes that use them without requiring a joint setup algorithm. Even when pairings are not required, we define schemes for group $\mathbb{G}_1$ and generator g_1 to anticipate their usage in later constructions. Our constructions are proven secure under the following assumptions.

Assumption 1 (Strong External Diffie-Hellman [39])**.** *The Strong External Diffie-Hellman (SXDH) assumption holds if every p.p.t. adversary solves the Decisional Diffie-Hellman (DDH) problems in $\mathbb{G}_1$ and $\mathbb{G}_2$ only with a negligible advantage.*

We introduce the Flexible co-CDH assumption and prove that it is implied by the above SXDH assumption.

Assumption 2 (Flexible co-CDH)**.** *The Flexible co-CDH assumption holds if, given $(g_2, {g_2}^a)$ where $g_2 \xleftarrow{\$} \mathbb{G}_2, a \xleftarrow{\$} \mathbb{Z}_p$, every p.p.t. adversary outputs a tuple $(h, h^a) \in {\mathbb{G}_1}^2$ such that $h \neq 1$ only with negligible probability.*

Lemma 2.1. *Strong External Diffie-Hellman implies Flexible co-CDH.*

Proof. Given $\mathcal{A}$ that solves Flexible co-CDH with non-negligible advantage, we show how to build an adversary $\mathcal{A}'$ for DDH in $\mathbb{G}_2$. $\mathcal{A}'$ is given a DDH instance $(g, g^a, g^b, C) \in \mathbb{G}_2^4$ and has to decide if $C = g^{ab}$. $\mathcal{A}'$ runs $\mathcal{A}$ with input (g, g^a). Let $\mathcal{A}$ output (h, h^a). Then $\mathcal{A}'$ can check if $C = g^{ab}$ by checking if $e(h^a, g^b) \stackrel{?}{=} e(h, C)$ holds. Hence $\mathcal{A}'$ succeeds in solving the DDH instance with $\mathcal{A}$'s success probability. □

For extractability, we optionally require the following assumption parameterized by hash size n:

Assumption 3 (Bilinear n-Knowledge of Exponent)**.** *The Bilinear n-Knowledge of Exponent assumption holds if, for every p.p.t. adversary $\mathcal{A}$, there exists a p.p.t. extractor $\mathcal{E}$ such that for all large enough λ and 'benign' auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$

$$\Pr\left[\begin{array}{l}\mathsf{pp} = \mathcal{G}_\lambda; H_i \xleftarrow{\$} \mathbb{G}_1; \omega \xleftarrow{\$} \mathbb{Z}_p; \\ \quad (A, B\,; (x_1, \ldots, x_n)) \leftarrow (\mathcal{A}\|\mathcal{E})\,(\mathsf{pp}, \{H_i, H_i^\omega\}_{i=1}^n, \mathsf{aux}) : \\ \qquad\qquad A^\omega = B \wedge A \neq \prod_{i=1}^{n} H_i^{x_i}\end{array}\right] \approx 0$$

In the game above, $(u; w) \leftarrow (\mathcal{A}\|\mathcal{E})$ indicates running both algorithms on the same inputs and random tape, and assigning their results to u and to w, respectively. This assumption can been seen as an n-Knowledge of Exponent Assumption [11] but for the general group model. Indeed the authors of [11] use the argument by Groth [30] to conjecture that their assumption must hold independently of the bilinear structure. Auxiliary input is required to be drawn from a 'benign distribution' to avoid impossibility of certain knowledge assumptions [12, 10].

2.1 Online-Offline SNARKs

We recall the definition of succinct non-interactive arguments (SNARG) and arguments of knowledge (SNARK) as used by our constructions.

Let $\{\mathcal{R}_\lambda\}_\lambda$ be a sequence of families of efficiently decidable relations $R \in \mathcal{R}_\lambda$, with $R \subset U_R \times W_R$. For pairs $(u\,; w) \in R$, we call u the *instance* and w the *witness*; we are interested in producing and verifying arguments that $\exists w. R(u\,; w)$ holds. We require that all instances include some data in a fixed format. That is, for each $R \in \mathcal{R}_\lambda$, we have $U_R = X \times V_R$

and instances are of the form $u = (x, v)$. For example, u may consist of the input x and output y of a function with domain X, i.e., $y = f(x)$. More generally, u may consist of the inputs x, y and output z of functions whose domains include X, i.e., $z = f(x, y)$.

For any sequence of families of efficiently decidable relations $\{\mathcal{R}_\lambda\}_\lambda$ as defined above, SNARGs and SNARKs consist of 3 algorithms $\mathsf{VC} = (\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$, as follows.

$(\mathsf{EK}, \mathsf{VK}) \leftarrow \mathsf{KeyGen}(1^\lambda, R)$ takes the security parameter and a relation $R \in \mathcal{R}_\lambda$ and computes evaluation and verification keys.

$\Pi \leftarrow \mathsf{Prove}(\mathsf{EK}, u\,; w)$ takes an evaluation key for R, an instance u, and a witness w such that $R(u\,; w)$ holds, and returns a proof.

$b \leftarrow \mathsf{Verify}(\mathsf{VK}, u, \Pi)$ takes a verification key and an instance u, and either accepts ($b = 1$) or rejects ($b = 0$) the proof Π.

$(\mathsf{EK}, \mathsf{VK})$ are also referred to as the common reference string.

Definition 2.1 (Soundness)**.** *A* VC *scheme is sound if, for all sequences $\{R_\lambda\}_{\lambda\in\mathbb{N}}$ in $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ and for all p.p.t. adversaries $\mathcal{A}$, we have*

$$\Pr\left[\begin{array}{l}\mathsf{EK}, \mathsf{VK} \leftarrow \mathsf{KeyGen}(1^\lambda, R_\lambda);\\ u, \Pi \leftarrow \mathcal{A}(\mathsf{EK}, \mathsf{VK}, R_\lambda);\\ \mathsf{Verify}(\mathsf{VK}, u, \Pi) \wedge \neg\exists w. R_\lambda(u\,; w)\end{array}\right] \approx 0.$$

Online-Offline Verification [1] The verification algorithm of many SNARG constructions can be split into *offline* and *online* computations. Specifically, for many SNARGs, there exists algorithms $(\mathsf{Online}, \mathsf{Offline})$ such that:

$$\mathsf{Verify}(\mathsf{VK}, u, \Pi) = \mathsf{Online}(\mathsf{VK}, \mathsf{Offline}(\mathsf{VK}, x), v, \Pi).$$

The offline phase can be seen as the computation of one or more Pedersen-like commitments c_x (here, $c_x = \mathsf{Offline}(\mathsf{VK}, x)$), some of which may be computed by the prover, and possibly never opened by the verifier. On their own, such commitments are not perfectly binding, so this involves modelling adversaries that do not output (u, w) but still must 'know' the value they are committing to. For such cases, we require the existence of an algorithm $\mathcal{E}$ that can extract x and w from a verifying proof.

Definition 2.2 (Online Knowledge Soundness)**.** *A* VC *scheme is online knowledge sound if, for all sequences $\{R_\lambda\}_{\lambda\in\mathbb{N}}$ in $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ and all p.p.t. adversaries $\mathcal{A}$, there exists a p.p.t. extractor $\mathcal{E}$ such that*

$$\Pr\left[\begin{array}{l}\mathsf{EK}, \mathsf{VK} \leftarrow \mathsf{KeyGen}(1^\lambda, R_\lambda);\\ (c_x, v, \Pi; x, w) \leftarrow (\mathcal{A}||\mathcal{E})(\mathsf{EK}, \mathsf{VK}, R_\lambda);\\ \mathsf{Online}(\mathsf{VK}, c_x, v, \Pi) = 1 \wedge \neg\exists w. R_\lambda(x, v; w)\end{array}\right] \approx 0$$

Instantiations of Online-Offline SNARKs Many succinct verifiable-computation constructions [20, 7, 9, 31] can be presented in a style that make more apparent their reliance on commitments on their inputs, outputs, and internal witnesses. We may instantiate VC using, for example, the Geppetto construction [19], which explicitly separates (offline) commitments and (online) proofs and provides online knowledge soundness.

[1]The offline phase is not to be confused with input-independent precomputation steps of the verifier in [8, 9].

Instantiation of Offline Verification In our work, we consider schemes where the *offline* computations consist purely of multi-exponentiations in $\mathbb{G}_1$ over the instance u, followed by *online* computations that accept or reject the proof. As mentioned above, we consider the case when U_R splits into X, V_R. More specifically, we assume that $X = \mathbb{Z}_p^n$ and $\mathsf{Offline}(\mathsf{VK}, x) = \prod F_i^{x_i}$ from X to $\mathbb{G}_1$, where the group elements $(F_1, F_2, \ldots, F_n) \in \mathbb{G}_1^n$ are part of the keys. The VC schemes discussed above follow this format.

3. MULTI-RELATION HASH & PROVE SCHEMES (HP)

We define our schemes for efficiently decidable relations $R \in \mathcal{R}_\lambda$, with $R \subset U_R \times W_R$. Recall that we are interested in producing and verifying arguments that $\exists w. R(u\,; w)$ holds for pairs $(u\,; w) \in R$, where u is the instance and w the witness. The witness can often speed up verification by providing a non-deterministic hint, as verification is often more efficient than computation, notably in the case of relations for NP complete languages. We keep the witness implicit when they can be efficiently computed from the instance. As in §2.1, we consider relations where U_R splits into X, V_R.

A multi-relation hash & prove scheme consists of 5 algorithms $\mathsf{HP} = (\mathsf{Setup}, \mathsf{Hash}, \mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$, as follows.

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ takes the security parameter and generates the public parameter for the scheme;

$\sigma_x \leftarrow \mathsf{Hash}(\mathsf{pp}, x)$ produces a hash given some data $x \in X$;

$\mathsf{EK}_R, \mathsf{VK}_R \leftarrow \mathsf{KeyGen}(\mathsf{pp}, R)$ generates evaluation key EK_R and verification key VK_R given a relation $R \in \mathcal{R}_\lambda$;

$\Pi_R \leftarrow \mathsf{Prove}(\mathsf{EK}_R, x, v\,; w)$ produces a proof of $R(x, v\,; w)$ given an instance and a witness that satisfy the relation.

$b \leftarrow \mathsf{Verify}(\mathsf{VK}_R, \sigma_x, v, \Pi_R)$ either accepts ($b = 1$) or rejects ($b = 0$) a proof of R given a hash of x and the rest of its instance v.

Note that hashes of inputs and the keys of a relation can be computed independently. In particular, σ_x can be computed 'offline', before generating keys, proving, or verifying instances of relations; and can be shared between all these operations.

3.1 Adaptive Soundness

We describe our intended security properties for an HP scheme, distinguishing two cases. We first define adaptive soundness with multiple relations and public verifiability, then describe a variant with a single relation.

Definition 3.1 (Adaptive Soundness)**.** *A multi-relation hash & prove scheme* HP *is adaptively sound if every p.p.t. adversary with access to oracle* KEYGEN *wins the game below with negligible probability.*

<u>*Adaptive Forgery Game*</u>
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$R, x, v, \Pi \leftarrow \mathcal{A}^{\mathsf{KEYGEN}}(1^\lambda, \mathsf{pp})$
$\mathcal{A}$ *wins if* $\mathsf{VERIFY}(R, x, v, \Pi) = 1$ *and* $\neg\exists w. R(x, v\,; w)$

<u>$\mathsf{KEYGEN}(R)$</u>
if $VK(R)$ exists, return $\perp$
$\mathsf{EK}, \mathsf{VK} \leftarrow \mathsf{KeyGen}(\mathsf{pp}, R)$
$VK(R) := \mathsf{VK}$;
return $(\mathsf{EK}, \mathsf{VK})$

<u>$\mathsf{VERIFY}(R, x, v, \Pi)$</u>
if $VK(R)$ undefined, return 0
$\sigma \leftarrow \mathsf{Hash}(\mathsf{pp}, x)$
return $\mathsf{Verify}(VK(R), \sigma, v, \Pi)$

The designated-verifier *variant of adaptive soundness is obtained by having* KEYGEN *return only* EK*, and giving* $\mathcal{A}$ *oracle access to* VERIFY*. The* single-relation *variant is obtained by requesting that the adversary calls* KEYGEN *once.*

Informally, adaptive soundness means that an adversary that interacts with a verifier on any number of chosen instances of relations supported by HP cannot forge any argument. Although the VERIFY procedure in the experiment always recomputes σ_x, this hash can of course be shared between verifications of multiple instances that use the same x.

Unfolding the definition, the single-relation, public verifiability game is defined by

Adaptive Forgery Game (single relation, public verifiability)

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$R, \mathsf{state} \leftarrow \mathcal{A}_0(1^\lambda, \mathsf{pp})$
$\mathsf{EK}, \mathsf{VK} \leftarrow \mathsf{KeyGen}(\mathsf{pp}, R)$
$x, v, \Pi \leftarrow \mathcal{A}_1(\mathsf{state}, \mathsf{EK}, \mathsf{VK})$
$\mathcal{A}$ wins if $\mathsf{Verify}(\mathsf{VK}, \mathsf{Hash}(\mathsf{pp}, x), v, \Pi) = 1$ and $\neg\exists w. R(x, v\,; w)$

This simpler, single-relation game is still adaptive, in the sense that the relation R can be chosen by $\mathcal{A}$ with knowledge of pp, and the instance x, v can depend on EK, VK. Using a standard hybrid argument, we confirm that adaptive single-relation soundness implies adaptive soundness.

Theorem 3.1 (Security of multi-relation HP)**.** *A* HP *scheme that is ϵ-secure as per Definition 3.1 for a single relation is $q\epsilon$-secure for multiple relations, where q bounds the number of calls to* KEYGEN *made by the adversary.*

3.2 Accepting Hashes from the Adversary

In the definition of adaptive soundness, all hash outputs need to be trusted: at some point, the verifier is given x and honestly computes its hash σ_x, or (equivalently) receives σ_x from a trusted party. However, there are cases where the verifier may be given σ_x but not x. As an example, a composite argument that there exists an intermediate $x \in X$ such that $f(z) = x$ and $g(x) = r$ may consist of $z, \sigma_x, r, \Pi_f, \Pi_g$ where Π_f and Π_g prove the two functional relations above. Passing an 'opaque' hash σ_x may be more efficient than passing x, and may enable the prover to keep x secret. Similarly, one may see σ_x as a binding commitment to some x, received from the adversary, then later used in arguments that disclose some of its contents. Definition 3.1 does not account for such arguments.

In order for HP to support arguments on hashes provided by the adversary, we further require that its Hash algorithm is an extractable collision-resistant hash function. The extractability property guarantees that σ_x was indeed produced by Hash on some input x. The collision-resistance property guarantees that it is hard to produce two inputs for which Hash produces the same output.

Definition 3.2 (Hash Extractability [11])**.** *A hash function* Hash *is extractable when, for any p.p.t. adversary $\mathcal{A}$, there exists a p.p.t. extractor $\mathcal{E}$ such that, for a large enough security parameter λ and 'benign' auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*, the adversary wins the game below with negligible probability.*

Hash Extraction Game

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\sigma; x_e) \leftarrow (\mathcal{A} \| \mathcal{E})\,(\mathsf{pp}, \mathsf{aux})$
$\mathcal{A}$ *wins if* $\exists x. \mathsf{Hash}(\mathsf{pp}, x) = \sigma \wedge \sigma \neq \mathsf{Hash}(\mathsf{pp}, x_e)$

and there is a p.p.t. algorithm $\mathsf{Check}(\mathsf{pp}, \sigma)$ *that returns 1 if* $\exists x. \mathsf{Hash}(\mathsf{pp}, x) = \sigma$ *and 0 otherwise.*

(In the game above, $(\mathcal{A} \| \mathcal{E})$ indicates running both algorithms on the same inputs and random tape, and assigning their results to σ and to x_e, respectively.) In contrast with the original definition of [11], we require the existence of Check so that our verifiers can check the well-formedness of hashes received from the adversary.

Adaptive soundness for HP schemes guarantees collision-resistance for Hash as long as, for all $x_0 \neq x_1$, there exists a relation $R \in \mathcal{R}_\lambda$ and $v \in V_R$ to separate them, that is, $\exists w. R(x_1, v; w) \wedge \neg\exists w. R(x_0, v; w)$. On the other hand, adaptive soundness does not guarantee that σ is unique, nor does it exclude adversaries able to forge σ that pass verification.

Complementarily, hash extraction enables us to verify arguments that include opaque hashes provided by the adversary by first extracting their content then applying adaptive soundness. To formalize this idea, we complete our definitions with a more generally useful notion of soundness, called *adaptive hash soundness*.

At a high level, an adaptively hash sound HP scheme allows us to verify a composite argument whose instances mix plaintext values $x \in X$ and opaque hashes $\sigma \in \Sigma$, where Σ is a finite set of hashes; importantly, the same σ can occur in multiple instances. To verify the argument, the verifier checks each proof using hashes that are either recomputed from $x \in X$ (once for each x, similar to Definition 3.1), or checked for well-formedness.

Our main result for this property is that any scheme HP that is both adaptively sound and hash extractable is also adaptively hash sound. This result relies on soundness of HP, provided that one has access to preimages of the hash values $\sigma \in \Sigma$; in turn, this requirement is guaranteed by the hash extractability property. See the full version for details.

Stronger Security Notions for HP Our security definitions for HP schemes model adaptive soundness and extractability of hash inputs, but not extractability of witnesses, i.e., an equivalent of knowledge soundness for HP schemes. While adaptive soundness is sufficient for applications such as verifiable computation in which the input data is supplied by the verifier, knowledge soundness can be useful when using HP schemes in larger cryptographic protocols and in applications where the prover also provides some input. Elaborating such a definition of knowledge soundness for HP schemes (and proving a construction using it) raises subtleties related to defining an extractor for an adversary that has *adaptive* access to the KEYGEN oracle. We believe this is an interesting direction, which we leave for future work. Another useful security notion that may be considered is zero-knowledge, which intuitively guarantees that proofs do not reveal any non-trivial information about the witnesses. A zero-knowledge definition for HP schemes is provided in the full version of this paper.

3.3 Hash & Prove Scheme via Inner Encoding

In the introduction, we distinguished between two ways of embedding data representation inside VC schemes: inner and outer encodings. Here we describe a construction proposed in [11, 26, 5, 15] which serves as an example of inner encoding. We call this scheme $\mathsf{HP}_{\mathsf{inn}}$. The construction is presented for completeness (to show that it formally adheres our new definitions), and to facilitate the comparison with our new constructions of §4.

The construction uses a keyed, collision-resistant hash scheme with domain X, consisting of two algorithms $k \leftarrow \mathsf{keygen}(1^\lambda)$ and $\sigma \leftarrow \mathsf{hash}_k(x)$, together with a succinct argument VC for a family of relations $\mathcal{R}'$, defined next.

Intuitively, we check the computation $\sigma = \mathsf{hash}_k(x)$ *within* the proof system: to argue on a relation R in $\mathsf{HP}_{\mathsf{inn}}$, our construction uses VC on a relation R':

$$R'(\sigma_x, v\,; x, w) = R(x, v\,; w) \wedge (\sigma_x = \mathsf{hash}_k(x)).$$

Compared with R, the relation R' uses σ_x instead of x in the instance, and takes x as an additional witness. (Presumably, σ_x is smaller than x and easier to process in proof verifications.) We define $\mathsf{HP}_{\mathsf{inn}}$ as follows:

$\mathsf{Setup}(1^\lambda)$ samples $k \leftarrow \mathsf{keygen}(1^\lambda)$ and returns k as pp;

$\mathsf{Hash}(\mathsf{pp}, x)$ computes $\sigma_x \leftarrow \mathsf{hash}_{\mathsf{pp}}(x)$;

$\mathsf{KeyGen}(\mathsf{pp}, R)$ generates $(\mathsf{EK}_R, \mathsf{VK}_R) \leftarrow \mathsf{VC.KeyGen}(1^\lambda, R')$;

$\mathsf{Prove}(\mathsf{EK}_R, x, v\,; w)$ returns $\Pi \leftarrow \mathsf{VC.Prove}(\mathsf{EK}_R, v, \sigma_x\,; x, w)$ for $\sigma_x = \mathsf{hash}_{\mathsf{pp}}(x)$;

$\mathsf{Verify}(\mathsf{VK}_R, \sigma_x, v, \Pi)$ returns $\mathsf{VC.Verify}(\mathsf{VK}_R, \sigma_x, v, \Pi)$.

Theorem 3.2. *If* VC *is knowledge-sound and* hash *is collision-resistant, then* $\mathsf{HP}_{\mathsf{inn}}$ *is adaptively sound (Definition 3.1 for multiple relations).*

Hash Extractability. The above construction naturally extends to extractable hashes, by applying VC to the relation that checks the hash computation, defined by

$$R_k(\sigma\,; x) = (\sigma = \mathsf{hash}_k(x)).$$

We write $\mathsf{HP}_{\mathcal{E}}$ for the resulting scheme, obtained from $\mathsf{HP}_{\mathsf{inn}}$ above by extending the Setup and Hash algorithms and adding a Check algorithm:

$\mathsf{Setup}_{\mathcal{E}}(1^\lambda)$ samples $k \leftarrow \mathsf{keygen}(1^\lambda)$; generates $\mathsf{EK}_{\mathsf{pp}}, \mathsf{VK}_{\mathsf{pp}} \leftarrow \mathsf{VC.KeyGen}(\mathsf{pp}, R_k)$; and returns $\mathsf{pp} = (k, \mathsf{EK}_{\mathsf{pp}}, \mathsf{VK}_{\mathsf{pp}})$;

$\mathsf{Hash}_{\mathcal{E}}(\mathsf{pp}, x)$ computes $\sigma_x \leftarrow \mathsf{hash}_k(x)$; $\Pi \leftarrow \mathsf{VC.Prove}(\mathsf{EK}_{\mathsf{pp}}, \sigma_x\,; x)$ and returns $\sigma = (\sigma_x, \Pi)$.

$\mathsf{Check}(\mathsf{pp}, \sigma)$ parses σ as (σ_x, Π) and returns $\mathsf{VC.Verify}(\mathsf{VK}_{\mathsf{pp}}, \sigma_x, v, \Pi)$.

Theorem 3.3. *If* VC *is knowledge-sound, then* $\mathsf{HP}_{\mathcal{E}}$ *is hash extractable (Definition 3.2).*

The proof of Theorem 3.3 follows from the existence of the VC extractor.

By using a separate VC scheme on a new relation R_k, rather than re-using a VC scheme on one of the relations R', we can use knowledge soundness in a completely standard manner, taking only the key k as 'benign' auxiliary input.

Discussion. The $\mathsf{HP}_{\mathsf{inn}}$ construction is simple, and can be extended to Merkle trees [15] to provide logarithmic random access in data structures. Its main practical drawback is that the relation to be verified now includes a hash computation, which adds tens of thousands of cryptographic operations to the prover's workload for each block of input when using standard algorithms such as SHA2 (§6). To lower this considerable cost for the prover, one pragmatically chooses custom, algebraic hash functions, which in turn increases the cost for the verifier that computes the digest. In the following sections we present constructions that are efficient for both the prover and the verifier.

4. HASH & PROVE CONSTRUCTIONS

In this section we present our main technical contribution: two efficient *multi-relation hash & prove* schemes for families of relations $\mathcal{R}_\lambda$. We let $R(x, v\,; w)$ range over these relations.

Our two schemes are obtained via a generic hash & prove construction that relies on two main building blocks: (i) any SNARK scheme that has offline/online verification algorithms (cf. §2.1) and where the offline verification consists of a multi-exponentiation in a group $\mathbb{G}_1$; (ii) any HP scheme that allows to prove the correctness of such multi-exponentiations.

Before presenting our generic construction in full detail, we provide some intuition. We start from the observation that in offline/online SNARKs the verifier already computes an element $c_x = \prod_i F_i{}^{x_i}$. Although c_x can be seen as a hash of the input x, such hash is relation-specific because the elements F_i depend on the relation R that was used in the SNARK's KeyGen. Our main idea is to outsource the computation of c_x to the prover in order to obtain an HP scheme where x can be hashed in a *relation-independent* manner. Then, we ask the prover to show the correctness of c_x using an HP scheme (where hashes are indeed relation-independent) that supports relations of the form $(x, c_x) : c_x = \prod_i F_i{}^{x_i}$.

Building an HP scheme from another HP scheme may look silly at first, however the key point is that we require an HP that supports a specific class of relations: only multi-exponentiations. Conversely, our method can be seen as a way to bootstrap, via SNARKs, an HP scheme that supports one specific class of computations into another one that can support arbitrary computations.

Following the generic HP construction from a hash & prove scheme for multi-exponentiation, we propose new constructions to instantiate the latter. The first, called XP_1, is publicly verifiable, whereas the second one, called XP_2, is in the designated verifier model but enjoys better efficiency. The two new schemes are significantly more efficient than what could be obtained using known techniques (e.g., the construction based on inner encoding in §3.3).

As a result, the instantiation of our generic construction with state-of-the-art SNARKs and our new HP for multi-exponentiation yields an HP system that, compared to the solution in §3.3, is at least $1,400\times$ times faster for the prover and the key generator (cf. §6).

The rest of the section is organized as follows. In §4.1 we describe the generic construction; in §4.2 we give our publicly verifiable HP scheme for multi-exponentiation, and in §4.3 we give the designated verifier one. Finally, in §4.4 we outline additional properties of our constructions, including data updates and extension of HP to accumulators.

4.1 Generic Hash & Prove Scheme ($\mathsf{HP}_{\mathsf{gen}}$)

Let $\mathsf{VC} = (\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$ be a SNARG scheme that supports a sequence of relations $\{\mathcal{R}_\lambda\}_\lambda$ and that has offline/online verification, as described in §2.1: we assume that every verification key VK of VC includes group elements $F_1, \ldots, F_n \in \mathbb{G}_1$ and that $\mathsf{Offline}(\mathsf{VK}, x) = \prod_{i=1}^n F_i{}^{x_i}$ computes a commitment c_x.

Let $\mathsf{XP} = (\mathsf{Setup}, \mathsf{Hash}, \mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$ be an HP scheme that supports relations $\mathcal{F} \subset U \times \emptyset$ where u is $\mathbb{Z}_p^n \times \mathbb{G}_1$, every $F \in \mathcal{F}$ is defined by a vector $F = (F_1, \ldots, F_n) \in \mathbb{G}_1^n$, and a pair $(x, c_x) \in \mathbb{Z}_p^n \times \mathbb{G}_1$ is in F iff $\prod_{i=1}^n F_i{}^{x_i} = c_x$.

We use XP and VC to construct a scheme $\mathsf{HP}_{\mathsf{gen}}$ that supports any combination of relations $R(x, v\,; w)$ supported by VC. The only requirement is that both schemes have compatible (or identical) public parameters. Namely, they share the same bilinear group setting, and the number of inputs in x, n, should be the same as in XP.

$\mathsf{HP}_{\mathsf{gen}}$ is defined as follows:

$\mathsf{Setup}(1^\lambda)$ runs $\mathsf{XP.Setup}(1^\lambda)$ and returns its public parameters pp.

$\mathsf{Hash}(\mathsf{pp}, x)$ returns $\sigma_x := \mathsf{XP.Hash}(\mathsf{pp}, x)$.

$\mathsf{KeyGen}(\mathsf{pp}, R)$ takes a relation R and runs

$\mathsf{EK}, \mathsf{VK} \leftarrow \mathsf{VC.KeyGen}(1^\lambda, R)$;
Let $F := (F_1, F_2, \ldots, F_n)$ *be the 'offline' elements in* VK;
$\mathsf{EK}_F, \mathsf{VK}_F \leftarrow \mathsf{XP.KeyGen}(\mathsf{pp}, F)$;
return $\mathsf{EK}_R := (\mathsf{EK}, \mathsf{VK}, \mathsf{EK}_F), \mathsf{VK}_R := (\mathsf{VK}, \mathsf{VK}_F)$.

$\mathsf{Prove}(\mathsf{EK}_R, x, v\,; w)$ parses EK_R as $(\mathsf{EK}, \mathsf{VK}, \mathsf{EK}_F)$ then runs

$c_x \leftarrow \mathsf{VC.Offline}(\mathsf{VK}, x)$;
$\Pi \leftarrow \mathsf{VC.Prove}(\mathsf{EK}, (x, v)\,; w)$;
$\Phi_x \leftarrow \mathsf{XP.Prove}(\mathsf{EK}_F, x, c_x)$;
return $\Pi_R := (c_x, \Pi, \Phi_x)$.

$\mathsf{Verify}(\mathsf{VK}_R, \sigma_x, v, \Pi_R)$ parses VK_R as $(\mathsf{VK}, \mathsf{VK}_F)$ and Π_R as (c_x, Π, Φ_x), and returns

$$\mathsf{VC.Online}(\mathsf{VK}, c_x, v, \Pi) \wedge \mathsf{XP.Verify}(\mathsf{VK}_F, \sigma_x, c_x, \Phi_x).$$

Hence, proofs Π_R in $\mathsf{HP}_{\mathsf{gen}}$ carry three representations of x: its portable hash σ_x; its offline relation-specific commitment c_x; and a multi-exponentiation proof Φ_x that binds the two. Compared with VC proofs, and using our instantiations of XP described later in this section, the communication overhead for $\mathsf{HP}_{\mathsf{gen}}$ proofs is two group elements (or three if we want hash extractability).

The following theorem states the security of $\mathsf{HP}_{\mathsf{gen}}$.

Theorem 4.1. *If* XP *is adaptively sound in the publicly verifiable (resp. designated verifier) setting, and* VC *is sound, then the* $\mathsf{HP}_{\mathsf{gen}}$ *construction in §4.1 is adaptively sound in the publicly verifiable (resp. designated verifier) setting.*

The idea is rather simple: any adversary which breaks $\mathsf{HP}_{\mathsf{gen}}$ has to either break the security of the underlying VC scheme, or cheat on the value of c_x, thus breaking the security of XP. Our proof shows a reduction for each case.

We also give a corollary that essentially says that, by instantiating our generic construction with a hash extractable XP scheme, we can handle arguments with untrusted hashes. It follows by construction of $\mathsf{HP}_{\mathsf{gen}}$, observing that this scheme uses the hashing algorithm of XP.

Corollary 4.1. *If* XP *is hash extractable, then the* $\mathsf{HP}_{\mathsf{gen}}$ *construction in §4.1 is also hash extractable.*

4.2 Our Publicly Verifiable HP Scheme for Multi-Exponentiation (XP_1)

We present our second key technical contribution: a hash & prove scheme, called XP_1, for the class of multi-exponentiation relations $\mathcal{F}$ described above.

For clarity, we write Φ instead of Π for restricted proofs.

$\mathsf{Setup}(1^\lambda)$ samples $H_i \xleftarrow{\$} \mathbb{G}_1$ for $i \in [1, n]$ and returns $\mathsf{pp} = (\mathcal{G}_\lambda, H)$ where $H = (H_1, \ldots, H_n)$.

$\mathsf{Hash}(\mathsf{pp}, (x_1, \ldots, x_n))$ returns $\sigma_x \leftarrow \prod_{i \in [1,n]} {H_i}^{x_i}$.

$\mathsf{KeyGen}(\mathsf{pp}, F)$ samples $u, v, w \xleftarrow{\$} \mathbb{Z}_p^*$ and computes $U \leftarrow {g_2}^u, V \leftarrow {g_2}^v, W \leftarrow {g_2}^w$; samples $R_i \xleftarrow{\$} \mathbb{G}_1$ and computes $T_i \leftarrow {H_i}^u {R_i}^v {F_i}^w$ for $i \in [1, n]$; and returns $\mathsf{EK}_F = (F, T, R)$ and $\mathsf{VK}_F = (U, V, W)$ where $R = (R_1, \ldots, R_n)$ and $T = (T_1, \ldots, T_n)$.

$\mathsf{Prove}(\mathsf{EK}_F, (x_1, \ldots, x_n), c_x)$ computes $T_x \leftarrow \prod_{i \in [1,n]} {T_i}^{x_i}$ and $R_x \leftarrow \prod_{i \in [1,n]} {R_i}^{x_i}$; and returns $\Phi_x = (T_x, R_x)$.

(Implicitly we require that $c_x = \prod_{i \in [1,n]} {F_i}^{x_i}$, though the c_x part of the instance is not used in the computation of the proof.)

$\mathsf{Verify}(\mathsf{VK}_F, \sigma_x, c_x, \Phi_x)$ parses $\Phi_x = (T_x, R_x)$ and returns

$$e(T_x, g_2) \stackrel{?}{=} e(\sigma_x, U)\, e(R_x, V)\, e(c_x, W).$$

The following theorem states that XP_1 scheme is secure. Correctness follows by inspection.

Theorem 4.2 (Adaptive Soundness of XP_1)**.** *If the Strong External DDH Assumption holds, then the* XP_1 *scheme above is adaptively sound (Definition 3.1 for multiple relations).*

Proof Outline. The proof works by considering the case of a single relation as the extension to multiple relations is obtained by applying Theorem 3.1.

Below we provide the outline of the security proof via a sequence of game hops.

Game 0: this is the adaptive soundness game of Definition 3.1 restricted to a single relation.

Game 1: this is a modification of Game 0 as follows. When answering the (single) $\underline{\mathsf{KEYGEN}(F)}$ oracle query, the challenger sets $w = \gamma v + \delta$ for random $\gamma, \delta \xleftarrow{\$} \mathbb{Z}_p$ (instead of sampling $w \xleftarrow{\$} \mathbb{Z}_p$). Next, when the adversary returns the proof (x^*, c^*, Φ^*), with $\Phi^* = (T^*, R^*)$, the challenger computes $\hat{T} \leftarrow \prod_{i \in [1,n]} {T_i}^{x_i^*}$ and $\hat{c} \leftarrow \prod_{i \in [1,n]} {F_i}^{x_i^*}$. Then, if $(T^*/\hat{T})(\hat{c}/c^*)^\delta = 1$ the outcome of the game is changed so that the adversary does *not* win.

We claim that Game 0 and Game 1 are statistically indistinguishable. The intuition is that δ is information theoretically hidden from the adversary, which implies that the only event which changes the game's outcome happens with negligible probability.

Game 2: this is a modification of Game 1 as follows. When answering the (single) $\underline{\mathsf{KEYGEN}(F)}$ oracle query, the challenger sets $u = \alpha v + \beta$ for random $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_p$ (instead of sampling $u \xleftarrow{\$} \mathbb{Z}_p$). Second, the challenger computes $R_i \leftarrow H_i^{-\alpha} F_i^{-\gamma}$ and $T_i \leftarrow H_i^{\beta} F_i^{\delta}$.

This game is essentially changing the distribution of the evaluation keys returned to the adversary. The distribution in this game however is computationally indistinguishable from the one in Game 1 under the Strong External DDH (SXDH) assumption. Finally, once accounted for this game difference it is possible to show that any p.p.t. adversary has negligible probability of winning in Game 2, under the Flexible co-CDH assumption (which in turn reduces to SXDH).

Detailed proofs for the indistinguishability of the three games as well as a reduction from winning in Game 2 to breaking Flexible co-CDH are in the full version. $\square$

We can make the XP_1 construction hash extractable by adding a knowledge component. The resulting scheme, $\mathsf{XP}_{\mathcal{E}}$, consists of algorithms KeyGen and Prove from XP_1 together with the following additional algorithms.

$\mathsf{Setup}_{\mathcal{E}}(1^\lambda, \mathcal{F})$ samples $H_i \xleftarrow{\$} \mathbb{G}_1$ for $i \in [1, n]$ and $\omega \xleftarrow{\$} \mathbb{Z}_p$ and returns $\mathsf{pp} = (\mathcal{G}_\lambda, g_2^\omega, \{H_i, H_i^\omega\}_{i\in[1,n]})$.

$\mathsf{Hash}_{\mathcal{E}}(\mathsf{pp}, (x_1, \dots x_n))$ computes $A_x \leftarrow \prod_{i\in[1,n]} H_i{}^{x_i}$ and $B_x \leftarrow \prod_{i\in[1,n]}(H_i{}^\omega)^{x_i}$. Returns $\sigma_x = (A_x, B_x)$.

$\mathsf{Check}(\mathsf{pp}, \sigma_x)$ takes g_2 and g_2^ω from pp; parses σ_x as (A_x, B_x); and checks that $e(A_x, g_2^\omega) \stackrel{?}{=} e(B_x, g_2)$.

$\mathsf{Verify}_{\mathcal{E}}(\mathsf{VK}_F, \sigma_x, c_x, \Phi_x)$ returns

$$e(T_x, g_2) \stackrel{?}{=} e(A_x, U)\, e(R_x, V)\, e(c_x, W) \wedge \mathsf{Check}(\mathsf{pp}, \sigma_x) \stackrel{?}{=} 1.$$

Lemma 4.1 (Hash Extractability of $\mathsf{XP}_{\mathcal{E}}$)**.** *If the Bilinear n-Knowledge of Exponent Assumption holds, then the $\mathsf{XP}_{\mathcal{E}}$ scheme above is hash extractable.*

Proof. The existence of an extractor for the Bilinear n-Knowledge of Exponent Assumption implies the existence of an extractor for the $\mathsf{XP}_{\mathcal{E}}$ construction. $\square$

4.3 Our Designated Verifier HP Scheme for Multi-Exponentiation (XP_2)

We present another hash & prove scheme for multi-exponentiation, called XP_2, which works in the designated verifier setting. XP_2 works similarly to XP_1 but has the advantage of requiring one less element in the proof and one less multi-exponentiation for the prover.

The XP_2 scheme works for the same class of relations $\mathcal{F}$ supported by XP_1, and the construction is obtained by adapting a multi-function verifiable computation scheme by Fiore and Gennaro [23], which works for a similar restricted class of functions, $(f_1, \dots, f_n) \in \mathbb{Z}_p^n$. In the full version we define XP_2 more generically based on homomorphic weak pseudo-random functions [23]. For simplicity, we describe below the instantiation of the scheme based on the SXDH assumption.

The scheme XP_2 works as follows:

$\mathsf{Setup}(1^\lambda)$ samples $H_i \xleftarrow{\$} \mathbb{G}_1$ for $i \in [1, n]$ and returns $\mathsf{pp} = (\mathcal{G}_\lambda, H)$ where $H = (H_1, \dots, H_n)$.

$\mathsf{Hash}(\mathsf{pp}, (x_1, \dots, x_n))$ returns $\sigma_x \leftarrow \prod_{i\in[1,n]} H_i{}^{x_i}$.

$\mathsf{KeyGen}(\mathsf{pp}, F)$ generates $\delta, k \xleftarrow{\$} \mathbb{Z}_p^*$; computes $T_i \leftarrow F_i{}^\delta H_i{}^k$ for $i \in [1, n]$; and returns $\mathsf{EK}_F = (F, T), \mathsf{VK}_F = (\delta, k)$ where $T = (T_1, \dots, T_n)$.

$\mathsf{Prove}(\mathsf{EK}_F, (x_1, \dots, x_n), c_x)$ computes $\Phi_x \leftarrow \prod_{i\in[1,n]} T_i{}^{x_i}$; and returns Φ_x. (Implicitly we require that $c_x = \prod_{i\in[1,n]} F_i{}^{x_i}$, though the c_x part of the instance is not used in the computation of the proof.)

$\mathsf{Verify}(\mathsf{VK}_F, \sigma_x, c_x, \Phi_x)$ returns $\Phi_x \stackrel{?}{=} c_x{}^\delta \cdot \sigma_x{}^k$.

Theorem 4.3 (Adaptive Soundness of XP_2)**.** *If the SXDH assumption holds in $\mathbb{G}_1$, then the XP_2 construction above is adaptively sound (Definition 3.1 for multiple relations and a designated verifier).*

We outline the intuition behind the proof of the Theorem. The values $(H_i^k)_i$ are pseudorandom (by SXDH), and thus so are $(T_i)_i$. After making a hybrid step where their distribution is changed to random, the value of δ becomes information-theoretically hidden from the adversary, making its probability of cheating negligible.

A publicly verifiable variant in the generic group. Interestingly, the above scheme can be modified to become publicly verifiable as follows: we publish (g_2^δ, g_2^k) as part of VK_F, and use these elements with a pairing in the verification algorithm. The resulting scheme has the advantage of being more efficient than XP_1. As a drawback, we can only argue its security in the generic group model, and leave this analysis for the full version of this work.

Hash Extractability. We note that we can make the construction XP_2 hash extractable by incorporating a knowledge component, in the same way as we show for XP_1.

4.4 Additional Properties of Our Instantiation

By plugging XP_1 (or XP_2) into the generic $\mathsf{HP}_{\mathsf{gen}}$ construction of §4.1 we obtain an efficient HP scheme that can handle any relation supported by the underlying SNARK system.

A useful property of the hash function of both our constructions XP_1 and XP_2 is its (additive) homomorphism, i.e., $\mathsf{Hash}(x_1) \cdot \mathsf{Hash}(x_2) = \mathsf{Hash}(x_1 + x_2)$. This property turns out to have several applications, which we summarize below.

Incremental hashing for data streaming applications. The hash of our construction can be computed incrementally as $\sigma_i \leftarrow \sigma_{i-1} \cdot H_i^{x_i}$ (with $\sigma_0 = 1$). This is particularly useful in applications where a resource-constrained device outsources a data stream $x_1, x_2, \dots$ to a remote server while keeping locally only a small digest σ_i computed as above. Later, at any point, the client will be able to verify a computation on the stream $x_1, \dots, x_i$ by only using σ_i. Furthermore, when hash extractability is not needed, the XP_1 construction can be modified by letting $H_i = \mathsf{RO}(i)$ where RO is a hash function that in the security proof is modeled as a random oracle (we omit a proof for this case which is straightforward: simply simulate $\mathsf{RO}(i)$ as the H_i in the current proof). This simple trick allows for constant-size public parameters and, more interestingly, to work with a potentially unbounded input size n—a feature particularly useful in streaming scenarios.

Efficient hash updates. Another application of the homomorphic property is efficient hash updates. Given a hash $\sigma_x = \prod_i H_i{}^{x_i}$ on a vector $x = (x_1, \dots, x_n)$, one can easily update the i-th location from x_i to x_i'. Instead of recomputing the hash from scratch (which would require work linear in n), one simply does a constant-time computation $\sigma_{x'} = \sigma_x \cdot H_i{}^{x_i' - x_i}$. This trick also generalizes to updating multiple locations in time linear only in the number of locations that require an update.

Multiple data sources. The homomorphic property also implies that the hash can be computed in a distributed manner. For instance, one user computes $\sigma_{x,k} = \prod_{i\in[1,k]} H_i^{x_i}$, a second user computes $\sigma_{x,\ell} = \prod_{i\in[k+1,\ell]} H_i^{x_i}$, and then a verifier who receives $\sigma_{x,k}$ and $\sigma_{x,\ell}$ can reconstruct the full digest on $(x_1, \dots, x_\ell)$ with a single multiplication. This feature is useful in those applications where the data is provided by multiple trusted sources, in which case only small digests have to be communicated. (For example, consider training a machine learning model using different datasets.)

Randomizing hash values. If one of the x_i inputs of the hash is uniformly random in $\mathbb{Z}_p$, then the output of $\mathsf{Hash}(\mathsf{pp}, x)$ is a uniformly random element in $\mathbb{G}_1$. Showing that SNARK systems randomized in this fashion do not leak anything about their hashed data is less trivial as the same randomness is reused by σ_x and the c_x values of different relations. This is akin to randomness reuse in ElGamal encryption, which is permissible. However, in most SNARK systems the group elements used for commitment randomization have structure, precluding a straightforward reduction to DDH. A detailed analysis of a multi-relation zero-knowledge property for specific VC schemes is thus an interesting open problem.

From hashes to accumulators. Accumulators are often used as succinct representations of sets that enable fast, limited, verifiable processing. For example, one can efficiently prove and verify arguments on set operations by exploiting the structure of accumulators [38], with better performance than by relying on a general-purpose VC scheme. To this end, we offer schemes that allow one to transition between proof systems that operate on hashes and accumulators. In particular, we introduce Accumulate & Prove scheme which is a variant of HP that operates on accumulators and builds on HP and XP (to verify that the hash and the accumulator were computed from same data). See the full version.

5. OUTSOURCING HASH COMPUTATIONS

In our efficient HP constructions of §4, the Hash algorithm computes a succinct digest σ_x using one exponentiation for every element of x. Hence, when using instantiations with XP_1 or XP_2, an $\mathsf{HP}_{\mathsf{gen}}$ verifier that wishes to relate computations verified using σ_x to their actual inputs x must still perform $|x|$ exponentiations, or trust some data provider that associates σ_x to x. Though the same σ_x could be used to verify many computations that involve x, thereby amortizing the cost of hash computation, we are looking to further optimize this cost.

In this section, complementarily, we describe a technique to outsource hash computations to an untrusted party such that the verifier (or its trusted data provider) only needs to perform $|x|$ field multiplications and one efficient cryptographic hash on x, say SHA2, typically saving two orders of magnitude.

We present our construction, called HP^*, as a generic extension of any HP system to which it adds support for verifiable outsourcing of hash computations. The main benefit of this extension is that the verifier does not need to run the Hash algorithm: instead, it can upload x to the untrusted prover; obtain its hash σ_x together with a proof of hashing Π_h, verify them; and finally keep σ_x. Intuitively, the verifier can then use σ_x to refer to x as if it had computed it itself.

5.1 Definition

We define HP^* as an extension of a given hash & prove scheme HP. In particular, the functionality of the trusted Hash algorithm is supplemented with a pair of new algorithms, HashProve and HashVerify, run respectively by the untrusted prover and by the verifier. HashProve computes a hash of data x and augments it with a proof that the hash is computed correctly (that is, it is computed according to Hash algorithm). HashVerify then accepts σ_x as the hash if the proof verifies correctly.

Formally, HP^* is a multi-relation hash & prove scheme that supports hash outsourcing and consists of 7 algorithms $\mathsf{HP}^* = (\mathsf{Setup}, \mathsf{Hash}, \mathsf{HashProve}, \mathsf{HashVerify}, \mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$. We omit a description of Setup, Hash, KeyGen, Prove and Verify, as they are defined identically to those in HP (§3).

$\Pi_h \leftarrow \mathsf{HashProve}(\mathsf{pp}, x, \sigma_x)$ produces a proof of $R_{\mathsf{Hash}}(x, \sigma_x) = (\sigma_x \stackrel{?}{=} \mathsf{Hash}(\mathsf{pp}, x))$ given some data $x \in X$ and hash σ_x;

$b_h \leftarrow \mathsf{HashVerify}(\mathsf{vp}, x, \sigma_x, \Pi_h)$ either accepts ($b_h = 1$) or rejects ($b_h = 0$) a proof that σ_x is a hash of data x.

In addition to being a Hash & Prove scheme (i.e., satisfying adaptive soundness or adaptive hash soundness), HP^* must be secure with regards to outsourcing, as defined below.

Definition 5.1 (Sound Hash Outsourcing)**.** *Outsourcing of* HP^* *hash computation is secure if every p.p.t. adversaries wins the game below only with negligible probability.*

<u>Outsourced Hash Game</u>
$\mathsf{pp}, \mathsf{vp} \leftarrow \mathsf{Setup}(1^\lambda)$
$x, \sigma_x^*, \Pi_h \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{vp})$
$\mathcal{A}$ *wins if* $\mathsf{HashVerify}(\mathsf{vp}, x, \sigma_x^*, \Pi_h) = 1$ *and* $\sigma_x^* \neq \mathsf{Hash}(\mathsf{pp}, x)$

This game is similar to the Hash Extraction game, but it does not involve extraction, as the verifier is given both x and σ_x^*. (The designated-verifiability variant is obtained by keeping vp private and, instead, giving the adversary oracle access to HashVerify.)

Hash outsourcing ensures that, when verifying composite arguments as in adaptive hash sound schemes (cf. §3.2), one can safely replace calls to Hash with calls to HashVerify. In particular, with HP^*, an argument can be passed to a relation either as data x, as a hash σ or as (x, σ^*). Our definition can be trivially satisfied by ignoring Π_h and setting $\mathsf{HashVerify}(\mathsf{pp}, x, \sigma, \Pi_h) = (\sigma \stackrel{?}{=} \mathsf{Hash}(\mathsf{pp}, x))$ but of course we are looking for more efficient constructions.

5.2 Efficient Construction (HP^*)

We build HP^* out of any hash & prove scheme HP, and two additional tools: an almost universal hash function h (recalled below) and a regular hash function H (that will be modeled as a random oracle).

Almost Universal Hash Functions. An ϵ-almost universal hash function h is such that, for all $x \neq x'$ chosen before h is sampled, we have $\Pr_h[h(x) = h(x')] \leq \epsilon$ [13]. We will use such functions from $\mathbb{Z}_p^n$ to $\mathbb{Z}_p$, instantiated by $h_\alpha(x) = \sum_{i=1}^n x_i \alpha^{i-1}$ and keyed with a random $\alpha \in \mathbb{Z}_p$. These functions can be computed as $h_\alpha(x) = x_1 + \alpha(x_2 + \ldots \alpha(x_{n-1} + \alpha x_n)))$ using n additions and $n-1$ multiplications by α, which is particularly efficient in verifiable-computation schemes for arithmetic circuits.

Lemma 5.1. *h_α is $(n-1)/p$-almost universal.*

Proof. Expanding the collision equality, we get $\sum_{i=1}^n x_i \alpha^{i-1} = \sum_{i=1}^n x'_i \alpha^{i-1}$, that is, $\sum_{i=1}^n (x_i - x'_i)\alpha^{i-1} = 0$. If $x \neq x'$, we have a non-zero polynomial in α of degree at most $n-1$, with at most $n-1$ roots, so this equality holds with probability at most $(n-1)/p$. ☐

Before delving into the details of the construction, let us describe its main ideas. The first idea is to build HP^* by extending any HP with algorithms $\mathsf{HashProve}$ and $\mathsf{HashVerify}$ that allow to prove and verify the correctness of $\sigma_x \stackrel{?}{=} \mathsf{Hash}(x)$. Notably, $\mathsf{HashVerify}$ must be significantly faster than recomputing $\mathsf{Hash}(x)$. To this end, our second idea is to let $\mathsf{HashProve}$ compute a (freshly sampled) universal hash function $h_\alpha(x)$ and generate a proof Π_h that links $h_\alpha(x)$ to the correct σ_x. Then our $\mathsf{HashVerify}$ simply checks Π_h (in constant time) and recomputes the universal hash $h_\alpha(x)$, which is much faster than the multi-exponentiation Hash. The security of universal hash functions relies on their input being chosen before h_α is sampled. To this end, we require that h_α depend on the input x by setting $\alpha = H(x, \sigma_x)$ where H is a hash function.

We are now ready to give our HP^* construction. Let R_h be the relation defined by $R_h(x, \alpha, \mu) = (\mu \stackrel{?}{=} h_\alpha(x))$, and let H be a hash function. We build HP^* using any HP that supports relation R_h and is hash-extractable.

$\mathsf{Setup}(1^\lambda)$ runs setup and generates keys for outsourcing h:

$$\mathsf{pp}' \leftarrow \mathsf{HP.Setup}(1^\lambda);$$
$$\mathsf{EK}_h, \mathsf{VK}_h \leftarrow \mathsf{HP.KeyGen}(\mathsf{pp}', R_h);$$
$$\textit{return } \mathsf{pp} = (\mathsf{pp}', \mathsf{EK}_h) \textit{ and } \mathsf{vp} = \mathsf{VK}_h;$$

$\mathsf{HashProve}(\mathsf{pp}, x, \sigma_x)$ computes $\alpha = H(x, \sigma_x)$; $\mu = h_\alpha(x)$; $\Pi_h \leftarrow \mathsf{HP.Prove}(\mathsf{EK}_h, x, (\alpha, \mu))$ and returns Π_h;

$\mathsf{HashVerify}(\mathsf{vp}, x, \sigma_x, \Pi_h)$ computes $\alpha = H(x, \sigma_x)$; $\mu = h_\alpha(x)$ and checks $\mathsf{HP.Verify}(\mathsf{VK}_h, \sigma_x, (\alpha, \mu), \Pi_h)$.

We omit Hash, KeyGen, Prove and Verify algorithms as they are simply calls to their counterparts in the HP scheme (for example, $\mathsf{HP}^*.\mathsf{KeyGen}$ calls $\mathsf{HP.KeyGen}(\mathsf{pp}', R)$).

We stress that, even if asymptotically our new construction is not better than the original one (the verifier performs $\Theta(n)$ operations), in practice, the operations performed by the verifier in $\mathsf{HP}^*.\mathsf{HashVerify}$ are orders of magnitude faster than those in $\mathsf{HP.Hash}$.

Discussion. Applying HP^* to our efficient constructions of §4 (either public or designated verifier), our proofs now carry a *fourth* representation $\mu = h_\alpha(x)$ of x in addition to its hash σ_x, its commitment c_x, and a proof Φ_x. Note that we rely on extraction only for the witnesses x of the fixed relation R_h.

To avoid random oracles, we can use an interactive, designated verifier variant of HP^*, whereby (1) the prover commits to x and σ_x; (2) the verifier sends a fresh random α; (3) the prover produces a proof of R_h; (4) the verifier checks the proof against x and σ_x, as above.

Security. We finally state the security of hash outsourcing:

Theorem 5.1. *In the random oracle model for H, if h_α is an ϵ-almost universal hash function,* HP *is adaptively sound and hash extractable in publicly verifiable (resp. designated verifier) setting, then* HP^* *is sound for outsourcing of hash computations as per Definition 5.1 in publicly verifiable (resp. designated verifier) setting.*

We note that all HP constructions in §4 can be made hash extractable (meeting requirements of Theorem 5.1) and can be used for secure hash outsourcing.

6. EVALUATION

In this section, we analyze and measure the performance of our new HP constructions compared to previous solutions.

Our evaluation is twofold. First we analyze the efficiency of our scheme $\mathsf{HP}_{\mathsf{gen}}$ from §4 (instantiated with Geppetto [19] and XP_1) and we compare it against the inner encoding construction $\mathsf{HP}_{\mathsf{inn}}$ of §3.3 (also instantiated with Geppetto and various choices of the hash function). Second, we report on the impact of our hash outsourcing technique of §5 in speeding up hashing and verification time.

6.1 Microbenchmarks

We performed a series of microbenchmarks on a single core of a 2.4 GHz Intel Xeon E5-2620 with 32 GB of RAM. The table below gives the time for individual operations on the fields and elliptic curves used by Geppetto. The cost of multi-exponentiation and for SHA-256 is reported for each 254-bit word of input.

operation	time
field addition	45.2 ns
field multiplication	316.7 ns
multi-exponentiation	231.2 μs
pairing	0.7 ms
SHA-256	193.6 ns

6.2 Inner vs. Outer Encodings

We compare the asymptotic performance of inner and outer encodings and summarize the results in Figure 1.

In our evaluation, we make a distinction between different types of verifier effort, depending on whether the verifier's input to the computation is passed by *value* or by *reference* via a hash (referred to as an opaque hash for HP schemes in §3.2). In the figure, they are denoted as "Verify IO" and "Verify Intermediate Commitments", respectively.

When the verifier's input is passed by value, she (or someone she trusts) must directly handle each IO value, so the cost depends on the size, n, of the IO. Note that for any particular verifier, such computation is required only once for a given IO value, as the computed commitment (or hash) can be reused in subsequent computations.

When a verifier uses IO values passed by reference, she verifies a proof using a commitment or hash of the IO values without handling them directly. Since the commitment/hash values are constant size, the verification effort is also constant. A verifier may use IO values passed by reference when the corresponding hash comes from a trusted source (e.g., the verifier herself), or when it represents intermediate values in a computation (e.g., between mappers and reducers in a MapReduce computation) where the verifier merely needs to check the consistency of the IO, rather than the values themselves.

$\mathsf{HP}_{\mathsf{inn}}$. We consider the construction $\mathsf{HP}_{\mathsf{inn}}$ given in §3.3 instantiated with Geppetto and either SHA-1, SHA-256, or Ajtai's [1] hash function. On the positive side, $\mathsf{HP}_{\mathsf{inn}}$ has the same number of elements in the proof as Geppetto; its online verification cost is the same as in Geppetto, while offline verification consists of one hash computation plus a multi-exponentiation on a fixed size word. On the negative side, to support a relation R, $\mathsf{HP}_{\mathsf{inn}}$ forces Geppetto to work with a relation R' which (on top of encoding R) encodes hash computations. The latter adds significantly to the evaluation key size and the prover's work, which scale linearly

	Generality	Verify proof	Verify IO	Verify Interm. Commit	Prover Effort	Proof Size (group elts.)
$\mathsf{HP}_{\mathsf{inn}}$ (Ajtai)	Yes	12 pairings	$Ajtai(n) + 1$ MultiExp	1 MultiExp	$O(D \log D)$	8
Geppetto [19]	No	12 pairings	$3n$ MultiExp	5 pairings	$O(d \log d)$	8
$\mathsf{HP}_{\mathsf{gen}}$	Yes	12 pairings	n MultiExp	4 pairings	$O(d \log d) + 2n$ MultiExp	10
$\mathsf{HP}_{\mathsf{gen}}$ (extract)	Yes	12 pairings	n MultiExp	6 pairings	$O(d \log d) + 3n$ MultiExp	11
HP^*	Yes	12 pairings	$SHA(n) + n$ MulAdd + 16 pairings + 6 MultiExp	4 pairings	$O(d \log d) + 2n$ MultiExp + $UHash(n)$	20

Figure 1: Asymptotic Performance. ***Comparison of our schemes and prior work. For our schemes, we assume the use of our publicly verifiable*** XP_1 ***scheme, and*** HP^* ***is instantiated with*** $\mathsf{HP}_{\mathsf{gen}}$***. We use*** n ***for the size of the inputs/outputs (IO),*** $d \gg n$ ***for the degree of the QAP used for the outsourced computation, and*** $D = d + 350n$***. MultiExp is the cost of a multi-exponentiation, and MulAdd is the cost of a simple field multiplication and addition.*** $Ajtai(x)$ ***and*** $SHA(x)$ ***is the time needed to compute an Ajtai (resp. SHA-256) hash on*** x ***words of input, and*** $UHash(x)$ ***is*** $O(x \log x)$***, i.e., the time necessary to compute and prove correct a universal hash.***

and quasilineary respectively in the number of quadratic equations needed to represent the computation. Concretely, Geppetto includes libraries for verifiably computing SHA-1 and SHA-256 hashes. For each 254-bit I/O element, these libraries require approximately 22,400 equations for SHA-1 or 35,000 for SHA-256. Similar libraries for Ajtai require only 300–400 equations per word of input, but they increase the cost for the verifier and may not suffice for privacy applications that require stronger randomness properties from the hash function [15].

Geppetto. Geppetto is an example of an outer encoding scheme which avoids the expenses incurred by inner encodings. For example, compared with the hundreds or thousands of equations used for inner encodings, Geppetto only adds one equation per word of input, and hence they report improving prover performance by two orders of magnitude for processing I/Os [19]. However, Geppetto's approach requires the verifier to compute commitments using a multi-exponentiation (versus a hash in $\mathsf{HP}_{\mathsf{inn}}$) that is linear in the I/O size. Furthermore, Geppetto must specify which computations will be supported at setup time, before data is selected for said computations.

Our $\mathsf{HP}_{\mathsf{gen}}$ Scheme. Unlike Geppetto, which fixes at setup which computations will be supported for committed data, our $\mathsf{HP}_{\mathsf{gen}}$ scheme offers full generality; i.e., data can be hashed completely independently of the computations to be performed, and indeed, new and fully general computations can be verified over previously hashed data.

$\mathsf{HP}_{\mathsf{gen}}$'s new generality comes at a modest computational cost relative to Geppetto. In terms of communication, $\mathsf{HP}_{\mathsf{gen}}$ proofs include two more elements (three with hash extractability); the evaluation key and the verification key of every relation contain, respectively, $2n$ and 3 extra elements. In terms of computation, our prover has to perform two additional n-way multi-exponentiations. The verifier's online cost is the same as in Geppetto, whereas offline verification requires one hash computation (i.e., one n-multi-exponentiation) plus four pairings. If we wish to support hash extractability, then this adds an additional group element to the proof, an additional multi-exponentiation for the prover, and an additional pairing for the verifier. Overall, the additional burden (linear in the I/O size n) that $\mathsf{HP}_{\mathsf{gen}}$ adds relative to Geppetto is quite small, since both the size of the evaluation key and the prover's effort are typically dominated by the complexity of the outsourced computation, which, in most applications, is much larger than n.

Compared with inner encodings like $\mathsf{HP}_{\mathsf{inn}}$, however, $\mathsf{HP}_{\mathsf{gen}}$ saves the prover significant effort. Concretely, if we instantiate $\mathsf{HP}_{\mathsf{inn}}$ with Ajtai's hash, then $\mathsf{HP}_{\mathsf{gen}}$ is $1,400\times$ faster per I/O word (e.g., for $n = 1,000$, $\mathsf{HP}_{\mathsf{inn}}$ takes 10 minutes while $\mathsf{HP}_{\mathsf{gen}}$ takes half a second), while for SHA-256, the difference is closer to $140,000\times$ (e.g., $\mathsf{HP}_{\mathsf{inn}}$ takes 18 hours).

Our HP^* Scheme: Outsourcing Hash Computations. Compared with $\mathsf{HP}_{\mathsf{gen}}$, HP^* drastically improves the verifier's I/O processing time. For the verifier, whereas $\mathsf{HP}_{\mathsf{gen}}$ required a multi-exponentiation linear in the I/O, with HP^*, the linear costs consist of (1) a symmetric, fast SHA-256 hash computation to compute the key α; and (2) for each word, n additions and $n-1$ multiplications over $\mathbb{Z}_p$. A conservative comparison based on the results from §6.1 shows that (2) is $654\times$ cheaper per I/O word than a multi-exponentiation, and that (1) using SHA-256 is even cheaper than (2). Overall, compared with its current I/O processing, HP^* thus reduces the linear costs of the Geppetto verifier by two orders of magnitude. As a concrete example, with $n = 1,000,000$, $\mathsf{HP}_{\mathsf{gen}}$ takes 4 minutes to process the I/O, while HP^* needs half a second. Compared with Pantry, (2) takes one multiplication per word, which is also significantly cheaper than computing Ajtai's algebraic hash function on each word. An additional benefit of HP^* is that the verifier's key becomes constant size (a few group elements for encoding α and μ) rather than linear in n.

These benefits come at a low cost: HP^* increases the size of the proof from 11 to 20 elements. For the prover, the proof cost increases by just $2n$ field operations and a SHA-256 hash computation, plus the cost of generating Π_h, which only depends on n and is independent of the overall relation to be proven.

6.3 Application Performance

To evaluate the impact of our schemes at the application level, we evaluated them on two applications.

Statistics has a data generator commit to n 64-bit words. Later, clients can outsource various statistical calculations on that data; for example, we experiment with computing K-bucket histograms.

DNA matching creates a commitment to a string of n nucleotides, against which a client can then outsource queries, such as looking for a match for a length K substring.

The performance results for both applications appear in Figure 2. As expected, IO verification in $\mathsf{HP}_{\mathsf{inn}}$ is more efficient compared to the outer encodings schemes. Among outer encodings, our HP^* outperforms others as the size of

	Verify proof	Verify IO	Prover Effort
Statistics ($n = 256, K = 8$)			
$\mathsf{HP}_{\mathsf{inn}}$ (Ajtai)	17ms	0.070ms	117s
Geppetto [19]	17ms	1380ms	113s
$\mathsf{HP}_{\mathsf{gen}}$	17ms	557ms	114s
HP^*	17ms	31ms	114s
Statistics ($n = 1024, K = 8$)			
$\mathsf{HP}_{\mathsf{inn}}$ (Ajtai)	17ms	0.3ms	2,100s
Geppetto [19]	17ms	6,267ms	2,084s
$\mathsf{HP}_{\mathsf{gen}}$	17ms	2,096ms	2,085s
HP^*	17ms	30ms	2,092s
DNA Search ($n = 600, K = 4$)			
$\mathsf{HP}_{\mathsf{inn}}$ (Ajtai)	17ms	0.079ms	13.64s
Geppetto [19]	17ms	1611ms	5.00s
$\mathsf{HP}_{\mathsf{gen}}$	17ms	574ms	5.01s
HP^*	17ms	31ms	6.07s
DNA Search ($n = 60,000, K = 4$)			
$\mathsf{HP}_{\mathsf{inn}}$ (Ajtai)	17ms	6.4ms	1,695s
Geppetto [19]	17ms	46,980ms	706s
$\mathsf{HP}_{\mathsf{gen}}$	17ms	15,636ms	710s
HP^*	17ms	104ms	931s

Figure 2: Application Performance. ***Comparison of our schemes and prior work for two example applications.***

the input grows and n multi-exponentiations start dominating the cost of verifying hash outsourcing in HP^*. On the other hand, the outer encodings schemes are more prover-friendly. In particular, the prover's total effort (IO plus computation) is 1.02-2.3x higher for $\mathsf{HP}_{\mathsf{inn}}$ than for HP^* (note that even though our schemes significantly reduce the prover's burden for IO, they do not affect the effort for the computation itself, and hence Amdahl's law limits the overall impact). Finally, the results for HP^* show that the additional computation the scheme imposes on the prover pays off: verification is 18-150x more efficient than for $\mathsf{HP}_{\mathsf{gen}}$ with at most a 30% increase in the prover's efforts.

7. RELATED WORK

Cryptographic proof systems come in a variety of shapes, with inherent trade-offs between the efficiency of their provers and verifiers and the expressiveness of the statements being proven. One particularly interesting point in the design space are computationally-sound non-interactive proof systems, also known as argument systems [14], that can be verified faster than by directly checking NP witnesses. Starting with the work of Micali [36], there has been much progress [11, 30, 6, 26, 20, 31] leading to succinct non-interactive argument systems often referred to as SNARKs or SNARGs, depending on whether they establish knowledge rather than just existence of the NP witness. Significant theoretical improvements have been complemented with nearly-practical general-purpose implementations [40, 7, 19, 9, 47].

As noted in §1 and §3.3, some prior work fits our hash & prove model with data verification embedded via inner and outer encodings. Here we review other solutions that follow the outer encoding approach.

In commit & prove schemes [33, 16], one can create a commitment to the data, and use it in multiple proofs. Costello *et al.* [19] and implicitly Lipmaa [35] use this idea for verifiable computation to efficiently share data between proofs. However, in this approach all computations have to be fixed *before* one creates commitments to data. In other words, one has to know a-priori which computations will be executed on the data, which may not be the case in applications like MapReduce. This issue can be mitigated by fixing a universal relation, i.e., a relation which contains all relations that can be executed within a fixed time bound. However, this generality comes at a performance cost.

Several works by Ben-Sasson *et al.* investigate how to efficiently build universal relations for predicates described as random-access machine algorithms [5, 7, 9]. For instance, they describe a SNARK scheme [9] supporting bounded-length executions on a universal von Neumann RISC machine with support for data dependent memory access, but this generality comes at a cost [19]. To achieve full generality, the bound on the execution length can be removed via proof bootstrapping [46]. Despite recent improvements and innovation [8], such bootstrapping is costly.

Memory delegation [18] also models a scenario where one outsources memory and only later chooses computations (including updates) to be executed on it in a verifiable way. In this model, after a preprocessing phase whose cost is linear in the memory size, the verifier's work in the online verification phase is sublinear in the memory size. In contrast, with HP schemes the verifier also needs to do linear work once to hash the input, but then the verification cost is constant with respect to the the input size.

Another possibility to address computation on previously outsourced data is to use homomorphic message authenticators [4] or signatures [17, 29]. With the former, data is flexibly authenticated when uploaded and then multiple functions can be executed and proved on it. Homomorphic authenticators share the limitation of commit & prove schemes: the class of computations has to be fixed before the data can be authenticated. Moreover, homomorphic authenticator constructions that offer more practical efficiency [4] work only for quite restricted classes of computations (low degree polynomials). The approach based on leveled homomorphic signatures [29] is more expressive but still very expensive in practice, as the size of the proof (i.e., evaluated signature) is polynomial in the depth of the computation's circuit.

AD-SNARKs [3] provide a functionality similar to homomorphic authenticators, working efficiently for arbitrary computations, but even in their case the set of computations has to be fixed a priori. As a further restriction, the model of both homomorphic authenticators and AD-SNARKs requires a secret key for data outsourcing, and it only supports append-only data uploading (i.e., it does not support changing the uploaded data). In contrast, the hash & prove model considered by this work supports delegating computation on public data, since hashes are publicly computable.

Finally, TRUESET [34] uses a Merkle hash tree over I/O commitments in a VC scheme to support computations on a subset of committed inputs (namely, a collection of sets). While this adds flexibility as to which inputs can be used in the computation, these inputs still have to be fixed a-priori.

Acknowledgments We thank the reviewers for their insightful comments and suggestions. The research of Dario Fiore is partially supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement 688722 (NEXTLEAP), the Spanish Ministry of Economy under project reference TIN2015-70713-R (DEDETIS) and a Juan de la Cierva fellowship, and by the Madrid Regional Government under project N-Greens (ref. S2013/ICE-2731). The research of Esha Ghosh is supported in part by the National Science Foundation under grant CNS–1525044.

8. REFERENCES

[1] Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: STOC (1996)

[2] Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: STOC (1991)

[3] Backes, M., Barbosa, M., Fiore, D., Reischuk, R.M.: ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In: IEEE Symposium on S&P (2015)

[4] Backes, M., Fiore, D., Reischuk, R.M.: Verifiable delegation of computation on outsourced data. In: CCS (2013)

[5] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In: ITCS (2013)

[6] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete efficiency of probabilistically-checkable proofs. In: STOC (2013)

[7] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: CRYPTO (2013)

[8] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: CRYPTO (2014)

[9] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: USENIX Security (2014)

[10] Bitansky, N., Canetti, R., Chiesa, A., Goldwasser, S., Lin, H., Rubinstein, A., Tromer, E.: The hunting of the SNARK. Cryptology ePrint Archive, Report 2014/580

[11] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: ITCS (2012)

[12] Bitansky, N., Canetti, R., Paneth, O., Rosen, A.: On the existence of extractable one-way functions. In: STOC (2014)

[13] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: Umac: Fast and secure message authentication. In: crypto. vol. 1666 (1999)

[14] Brassard, G., Chaum, D., Crépeau, C.: Minimum disclosure proofs of knowledge. J. Comput. Syst. Sci. 37(2) (1988)

[15] Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: Proc. of the ACM SOSP (2013)

[16] Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: STOC (2002)

[17] Catalano, D., Fiore, D., Warinschi, B.: Homomorphic signatures with efficient verification for polynomial functions. In: CRYPTO (2014)

[18] Chung, K.M., Kalai, Y.T., Liu, F.H., Raz, R.: Memory delegation. In: CRYPTO. pp. 151–165 (2011)

[19] Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: Versatile verifiable computation. In: IEEE Symposium on S&P (2015)

[20] Danezis, G., Fournet, C., Groth, J., Kohlweiss, M.: Square span programs with applications to succinct NIZK arguments. In: ASIACRYPT (2014)

[21] Devanbu, P.T., Gertz, M., Martel, C.U., Stubblebine, S.G.: Authentic third-party data publication. In: IFIP TC11/ WG11.3 (2001)

[22] Fiore, D., Fournet, C., Gosh, E., Kohlweiss, M., Ohrimenko, O., Parno, B.: Hash first, argue later: Adaptive verifiable computations on outsourced data. Cryptology ePrint Archive (2016)

[23] Fiore, D., Gennaro, R.: Publicly verifiable delegation of large polynomials and matrix computations, with applications. In: CCS (2012)

[24] Fournet, C., Kohlweiss, M., Danezis, G., Luo, Z.: ZQL: A compiler for privacy-preserving data processing. In: USENIX Security (2013)

[25] Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: CRYPTO (2010)

[26] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: EUROCRYPT (2013)

[27] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: STOC (2008)

[28] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. 18(1) (1989)

[29] Gorbunov, S., Vaikuntanathan, V., Wichs, D.: Leveled fully homomorphic signatures from standard lattices. In: STOC (2015)

[30] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: ASIACRYPT (2010)

[31] Groth, J.: On the size of pairing-based non-interactive arguments. In: EUROCRYPT (2016)

[32] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: ASIACRYPT (2010)

[33] Kilian, J.: Uses of randomness in algorithms and protocols. PhD thesis, Massachusetts Institute of Technology (1989)

[34] Kosba, A.E., Papadopoulos, D., Papamanthou, C., Sayed, M.F., Shi, E., Triandopoulos, N.: TRUESET: Faster verifiable set computations. In: USENIX Security (2014)

[35] Lipmaa, H.: Prover-efficient commit-and-prove zero-knowledge SNARKs. In: AFRICACRYPT (2016)

[36] Micali, S.: Computationally sound proofs. SIAM J. Comput. 30(4) (2000)

[37] Nguyen, L.: Accumulators from bilinear pairings and applications. In: IEEE Symposium on S&P (2005)

[38] Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal verification of operations on dynamic sets. In: CRYPTO (2011)

[39] Pape, S.: Authentication in Insecure Environments - Using Visual Cryptography and Non-Transferable Credentials in Practise. Springer (2014)

[40] Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: IEEE Symposium on S&P (2013)

[41] Parno, B., Raykova, M., Vaikuntanathan, V.: How to delegate and verify in public: Verifiable computation from attribute-based encryption. In: TCC (2012)

[42] Rial, A., Danezis, G.: Privacy-preserving smart metering. In: WPES (2011)

[43] Setty, S., McPherson, R., Blumberg, A.J., Walfish, M.: Making argument systems for outsourced computation practical (sometimes). In: Proc. ISOC NDSS (2012)

[44] Tamassia, R.: Authenticated data structures. In: ESA (2003)

[45] Thaler, J., Roberts, M., Mitzenmacher, M., Pfister, H.: Verifiable computation with massively parallel interactive proofs. In: USENIX HotCloud Workshop (2012)

[46] Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: TCC (2008)

[47] Wahby, R.S., Setty, S., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: Proc. of the ISOC NDSS (Feb 2015)