2015 IEEE Symposium on Security and Privacy

Geppetto: Versatile Verifiable Computation

Craig Costello Cédric Fournet Jon Howell Markulf Kohlweiss
Benjamin Kreuter[†] Michael Naehrig Bryan Parno Samee Zahur[†]
Microsoft Research [†] University of Virginia

Abstract

Cloud computing sparked interest in Verifiable Computation protocols, which allow a weak client to securely outsource computations to remote parties. Recent work has dramatically reduced the client's cost to verify the correctness of their results, but the overhead to *produce* proofs remains largely impractical.

Geppetto introduces complementary techniques for reducing prover overhead and increasing prover flexibility. With MultiQAPs, Geppetto reduces the cost of sharing state between computations (e.g., for MapReduce) or within a single computation by up to two orders of magnitude. Via a careful choice of cryptographic primitives, Geppetto's instantiation of bounded proof bootstrapping improves on prior bootstrapped systems by up to five orders of magnitude, albeit at some cost in universality. Geppetto also efficiently verifies the correct execution of proprietary (i.e., secret) algorithms. Finally, Geppetto's use of energy-saving circuits brings the prover's costs more in line with the program's actual (rather than worst-case) execution time.

Geppetto is implemented in a full-fledged, scalable compiler and runtime that consume LLVM code generated from a variety of source C programs and cryptographic libraries.

1 Introduction

The recent growth of mobile and cloud computing makes outsourcing computations from a weak client to a computationally powerful worker increasingly attractive economically. Verifying the correctness of such outsourced computations, however, remains challenging, as does maintaining the privacy of sensitive data used in such computations, or even the privacy of the computation itself. Prior work on verifying computation focused on narrow classes of computation [32, 51], relied on physical-security assumptions [41, 47], assumed uncorrelated failures [19, 20], or achieved good asymptotics [2, 28, 30, 31, 33, 38, 44] but impractical concrete performance [46, 50].

Recently, several lines of work [9, 46, 49, 52] on verifiable computation [28] have combined theoretical and engineering innovations to build systems that can verify the results of general-purpose outsourced computations while making at most cryptographic assumptions. Two of the best performing, general-purpose protocols for verifiable computation [46, 49] are based on Quadratic Arithmetic Programs (QAPs) [29]. To provide non-interactive, publicly verifiable computation, as well as zero-knowledge proofs (i.e., proofs of computations in which some or all of the worker's inputs are private), many recent systems [3, 7, 9, 10, 16, 25, 39, 54] have converged on the Pinocchio protocol [46] as a cryptographic back end. Pinocchio, in turn, depends on QAPs.

While these protocols have made verification nearly practical for clients, the cost to *generate* a proof remains a significant barrier to practicality for workers. Indeed, most applications are constrained to small instances, since proof generation costs 3–6 *orders of magnitude* more than the original computation.

With Geppetto[1], we combine a series of interlocked techniques that support more flexible, and hence more efficient, provers. These techniques include the new notion of MultiQAPs for sharing state between or within computations, bounded bootstrapping for succinct proof aggregation, a QAP-friendly C library for verifying cryptographic computations, and a new technique for energy-saving circuits, which ensures the prover's costs grow with actual execution time, rather than worst-case execution time.

In more detail, we first generalize QAPs to create *MultiQAPs*, which allow the verifier (or prover) to commit to data once and then use that data in multiple related proofs. For example, the prover can commit to a data set and then use it in many different MapReduce jobs. At a finer granularity, we show how to use MultiQAPs to break an arithmetic circuit up into many smaller, simpler verifiable circuits that efficiently share state. Today, compiling code from C to a QAP typically requires unrolling all loops and inlining all functions, leading to a huge circuit full of replicated subcircuit structures. Since key size, and key and proof generation time all depend linearly (or quasilinearly) on the circuit size, this blowup severely degrades performance. With MultiQAPs, instead of unrolling a loop, we can create a single circuit for the loop body, use a proof for each iteration of the loop, and connect the state at the end of each iteration to the input of the next iteration. This allows us to shrink key size and key generation time, and, more importantly, to save the prover time and memory. Prior work suggested achieving similar properties via Merkle hash trees [8, 12, 27, 29, 43], but implementations show that this approach increases the degree of the QAP by tens or hundreds per state element [9, 16, 54], whereas with MultiQAPs, the degree increases only by 1.

With MultiQAPs, the prover generates multiple proofs about related data. This improves flexibility and performance for the prover, but it degrades attractive features of Pinocchio, namely that the proof consists of a (tiny) constant-sized proof, and the verifier's work scales only with the IO.

[1] A skilled craftsman who can create and coordinate many Pinocchios.

© 2015, Craig Costello. Under license to IEEE.
DOI 10.1109/SP.2015.23

IEEE computer society

As a second contribution, we explore the use of *bounded proof bootstrapping* to obtain MultiQAPs with constant-sized proofs. In theory, with proof bootstrapping [11, 53], the prover can combine any series of proofs into one by verifiably computing the verification of all of those proofs. Very recent work elegantly instantiates unbounded proof bootstrapping [9], but this generality comes at a cost (§5,§7.3.1). Our instantiation and implementation of bounded proof bootstrapping shows that, as with semi-homomorphic vs. fully homomorphic encryption, if we pragmatically set a bound on the number of proofs we intend to combine, we can achieve more practical performance.

To support bounded proof bootstrapping, Geppetto includes a QAP-friendly C library for general-purpose *cryptographic* computations. Such computations arise in many outsourcing applications. For instance, a MapReduce job may need to compute over signed data, or a customer with a smart meter may wish to privately compute a bill over signed readings [48]. As another example, recent work [7, 25] shows how to anonymize Bitcoin transactions using Pinocchio [46] and would benefit from the ability to verify signatures within transactions. In existing QAP systems, computations take place over a relatively small (e.g., 254-bit) field, so computing cryptographic operations (e.g., a signature verification) requires an awkward embedding of the cryptographic machinery via either a BigInteger library built out of field elements or via large extension fields [25]. With our techniques, all of these examples can be naturally and efficiently embedded into a proof of an outsourced computation.

By considering (bounded or unbounded) bootstrapping in the context of our QAP-friendly crypto library, we show how to efficiently compile and outsource computations so that the computation itself is hidden from the verifier. For example, a patient might verify that a trusted authority (say the US FDA) signed the code for a medical-data analysis, and that the analysis was correctly applied to the patient's data, without the patient ever learning anything about the proprietary analysis algorithm. Previous systems could potentially support this scenario via universal circuits [46, 49] or circuits executing CPU instructions [9], but the extra level of interpretation potentially slows the computation down by orders of magnitude (see §7.3.1).

Lastly, just as MultiQAPs eliminate the redundancy that comes from code repetition (e.g., in the form of loops or function invocations), we introduce the notion of *energy-saving circuits* to eliminate the redundant work that arises from code branching. With energy saving, the prover only exerts cryptographic effort for the actual path taken (e.g., only the 'if' branch when the condition is true). While energy-saving circuits are generally useful, they are particularly beneficial when using bounded proof bootstrapping to combine many proofs from a MultiQAP. Such proof compaction requires the key generator to commit, in advance, to the maximal number of proofs to be combined. With energy saving circuits, the key generator can choose a large number, and if a particular computation requires fewer proofs, the prover only performs cryptographic operations proportional to the number of proofs used, rather than the maximum chosen by the key generator.

We have implemented Geppetto as a complete toolchain for verifying the execution of C programs. Geppetto's code is available at `https://vc.codeplex.com`. It includes a compiler in F#, a cryptographic runtime in C++, QAP-friendly libraries in C, and various programming examples. Our compiler takes as input LLVM code produced by clang, a mainstream state-of-the-art optimizing C compiler; this enables us to focus on QAP-specific compilation. Our libraries support explicit, low-level control for programming MultiQAPs, allowing the C programmer to dictate how state flows from one QAP to another and hence control the resulting cryptographic costs. Geppetto also provides higher-level C libraries for common programming patterns, such as MapReduce or loops.

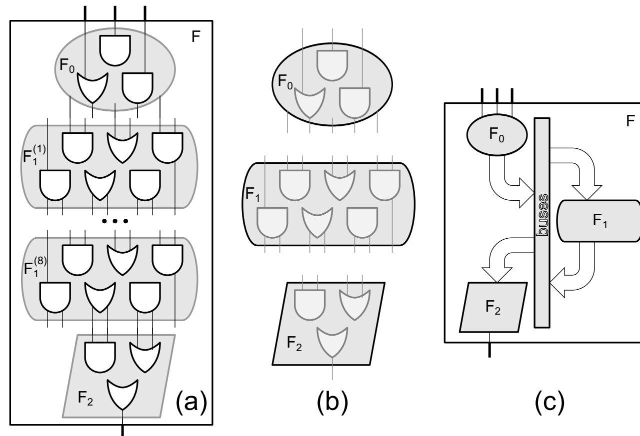

Figure 1: **MultiQAPs** *(a) Most existing verifiable computation systems compile programs to a single large circuit-like representation, leading to internal redundancy. (b) By extracting common substructures, we can represent a program as an assembly of smaller circuits, but the verifier must now also check all connections between circuits. (c) MultiQAPs connect circuits using* bus *structures that support succinct and efficient commitments to the bus values.*

2 Geppetto Overview

In this section, we give an overview of Geppetto's main constructions: MultiQAPs (§2.1), proofs for cryptographic operations and bootstrapping (§2.2), and energy-saving circuits (§2.3). We defer cryptographic definitions to §3 and our protocol to §4.

2.1 MultiQAPs

2.1.1 MultiQAP Intuition

At a high level, prior verifiable computation systems like Pinocchio [46] allow a prover to convince a skeptical verifier that $F(\mathbf{u}) = \mathbf{y}$, where $\mathbf{u}$ is a verifier-supplied vector of inputs. The prover accomplishes this with a constant-sized proof π, and the verifier's work scales linearly in $|\mathbf{u}| + |\mathbf{y}|$, regardless of the complexity of F. However, as F grows to encompass larger and more complex functionality (see Figure 1), the CPU and memory costs for the prover (as well as its key size) increase superlinearly. As §7.2 shows, this limits prior systems to modest application parameters.

To scale to larger problems, we can naturally decompose the proof of F into a conjunction of proofs of m simpler functions F_0, ..., F_{m-1}. For example, if $F(\mathbf{u}) = F_1(F_0(\mathbf{u}))$, then naïvely the prover could use Pinocchio twice to prove:

$$\mathbf{z} = F_0(\mathbf{u}) \quad (0)$$

$$\mathbf{y} = F_1(\mathbf{z}) \quad (1)$$

The verifier would check a proof for each equation separately *and* check that the output from F_0 was correctly used as input to F_1. Unfortunately, this means that the prover must send the intermediate state $\mathbf{z}$ to the verifier, and the verifier must perform work linear in $|\mathbf{z}|$. If $\mathbf{z}$ is large, then handling so much intermediate state would make it difficult or impossible for the verifier to benefit from outsourcing.

Instead, with Geppetto, we have the prover return a constant-sized *digest*, D_z, representing the intermediate state $\mathbf{z}$. The verifier uses this digest when checking the proof for Equation (0) *and* when checking the proof for Equation (1), ensuring that the prover consistently used the same intermediate state in both proofs, but without requiring the verifier to explicitly handle $\mathbf{z}$.

Prior work achieved a similar reduction in verifier effort by extending F_0 to hash its output and F_1 to hash its input, so the verifier need only handle the constant-sized hash value [8, 12, 29, 43]. However, those hash computations make both functions more expensive [9, 16]. In contrast, with Geppetto, we observe that Pinocchio already computes a digest-like structure and that, with a careful refinement of its encoding, we can have the prover compute digests almost for free.[2]

In more detail, we divide all of the variables used to compute F into disjoint sets we call *banks*. Each bank falls into one of three categories: a bank may represent F's (the overall computation's) input and output ($\mathbf{u}$ and $\mathbf{y}$ in our earlier example); it may represent a set of 'local' variables used within a single F_i; or it may be a *bus*, i.e., a set of variables shared between multiple F_i (e.g., $\mathbf{z}$).

Each bank is associated with its own cryptographic key material, used to compute a succinct digest of the values assigned to the bank's variables: the prover produces a digest for each local bank and for each bus, while the verifier produces a digest for the IO banks as part of the verification process. The latter ensures that the proof verification is with respect to the input the verifier supplied, and the alleged output the prover produced.

To verify a proof that a given F_i was computed correctly, the verification algorithm will need a digest for F_i's local bank, and digests for any buses or IO banks that F_i reads or writes. Continuing our earlier example, the verifier computes IO digests D_u and D_y. The prover computes and returns digests D_{F_0} and D_{F_1} summarizing the intermediate variables used by F_0 and F_1 respectively, and a *single* digest D_z representing the values on the bus between them. He also returns proofs π_0 and π_1 to demonstrate that F_0 and F_1 were computed correctly. The verifier runs the verification algorithm twice:

$$\mathsf{Verify}((D_u, D_{F_0}, D_z), \pi_0) \quad (2)$$

$$\mathsf{Verify}((D_z, D_{F_1}, D_y), \pi_1) \quad (3)$$

[2] We use 'digest' rather than 'commitment', since only some of the digests need to be binding—see §3.1.

$F; F_0, \ldots, F_{m-1}$	Function F is decomposed into m functions F_i
$\boldsymbol{\chi}$	Formal variables used when computing F
$\mathbf{B}, \ell$	A partition of $\boldsymbol{\chi}$ into banks $B_b \in \mathbf{B}$ with $\ell \triangleq \lvert\mathbf{B}\rvert$
$B_b^{(t_b)}$	An instance t_b of bank B_b;
χ_b	Commit-and-prove message for bank B_b (Defn. 2)
σ, n	A proof schedule (Defn. 1) with length $n \triangleq \lvert\sigma\rvert$
$Q^\star$, Q_i	The MultiQAP $Q^\star$, combining sub-QAPs Q_i
ρ, d	A QAP has size ρ and degree d

Figure 2: **Notation summary for §2.**

and accepts y as $F(u)$ if both checks succeed.[3] Note that D_z occurs in both verification checks. Formally, a system that allows a prover to commit to state in this fashion and use the resulting commitments in multiple proofs is known as a commit-and-prove (CP) scheme (see §3.1).

As shown in Figure 1, proofs of complex functions F may involve multiple instances of a simpler function F_i. For example, F_i may represent the execution of a single function call, or a single loop iteration in F. Each instance of F_i requires the prover to generate (and the verifier to check) a fresh proof, along with digests for the banks involved. In §2.1.2, we formalize these relationships with a *proof schedule* (Defn 1); each step in the schedule indicates which F_i is "active", which banks it depends on, and which set of bank values this particular instance of F_i depends on.

To efficiently build a commit-and-prove system supporting such schedules, we use Pinocchio's techniques to express each function F_i as a Quadratic Arithmetic Program (QAP) Q_i, a format suitable for succinct cryptographic proofs. To share state between individual Q_i, we combine them into a single *Multi-QAP* $Q^\star$ that also efficiently incorporates the buses connecting them. Using a MultiQAP also simplifies our definitions, constructions, and security proofs. In particular, we can repeatedly use a commit-and-prove scheme for a *single* relation for *all* proof schedules composed of different Q_i steps, with the ability to share compact, private digests between the proof steps. MultiQAPs support this functionality without significantly increasing the prover's costs beyond what is required to handle each sub-QAP of the schedule individually.

2.1.2 Scheduling Proofs With Shared State

As described in §2.1.1, we decompose the proof of a complex function F into a conjunction of proofs of m simpler functions $F_0,\ldots,F_{m-1}$.[4] Let $\boldsymbol{\chi}$ represent all of the formal variables used when computing F; this includes F's input and output variables, variables "local" to the computation of each F_i, and the variables shared across the F_i. Based on these different roles, we partition $\boldsymbol{\chi}$ into banks $B_b \in \mathbf{B}$.

A given execution of F may involve several instances of the same bank (e.g., if F_i represents a loop body, then the banks

[3] This approach generalize's Pinocchio's, which calls (D_{F_0}, π_0) the proof for F_0 and has the verifier compute D_u and D_z inside the verification algorithm.

[4] Cryptographers think of F as a language, and F's IO as a language instance. Programmers may see this proof as a trace-property, e.g., interpreting $\mathbf{u}, \mathbf{y}$ as a valid input-output sequence obtained by running a program whose specification is captured by F.

corresponding to its IO and local variables may take on different concrete values on each loop iteration). We refer to these distinct instances of bank B_b as $B_b^{(t_b)}$ for $t_b = 1,2,\ldots$ reserving $t_b = 0$ for the instance that assigns the constant 0 to every variable in B_b. With these notations (summarized in Figure 2), we can define proof schedules.

Definition 1 (Multi-proof schedule) *A* schedule σ *is a sequence of* steps *of the form* $(i,\mathbf{t})$ *where* $i \in [m]$ *and* $\mathbf{t}$ *is a vector with an index* $t_b \geq 0$ *for each bank* $B_b \in \mathbf{B}$. *We define* $n \triangleq |\sigma|$ *to be the* length *of the schedule, and* $\ell \triangleq |\mathbf{B}|$ *the number of banks.*

Each step $(i,\mathbf{t})$ *of the schedule selects a function* F_i *and the instances* $B_b^{(t_b)}$ *of the banks it uses, with* $t_b = 0$ *whenever* F_i *does not use* B_b. *We require that* F_i *use only its local bank* B_{F_i}, *that is,* $t_{F_j} = 0$ *whenever* $i \neq j$.

A proof *for* σ *consists of (1) a proof* π_i *for each of its steps, and (2) a digest* $D_b^{(t)}$ *for each of its bank instances* $B_b^{(t)}$.

Intuitively, the schedule indicates a sequence of calls to F_is for which the prover must generate (or the verifier must check) a proof, and the indexes $\mathbf{t}$ of the banks digests that the prover (or the verifier) should use with that proof. The variables in any banks not used in a given step are implicitly set to 0 and hence can be represented with a trivial digest.

Returning to our example from §2.1.1, we have $\mathbf{B} = (B_u, B_y, B_{F_0}, B_{F_1}, B_z)$ and the schedule for Equations (0) and (1) would be $\sigma = [(0,(1,0,1,0,1)),(1,(0,1,0,1,1))]$.

2.1.3 An Efficient CP System from MultiQAPs

To understand Geppetto's MultiQAPs, it helps to review how Pinocchio encodes computations as QAPs. This encoding enables Pinocchio's efficient cryptographic protocol.

Quadratic Arithmetic Programs (QAPs) [29, 46] Abstractly, Pinocchio compiles a function F into a conjunction of d equations of the form

$$Q(\boldsymbol{\chi}) \triangleq \bigwedge_{r\in[d]} (\mathbf{v}_r \cdot \boldsymbol{\chi})(\mathbf{w}_r \cdot \boldsymbol{\chi}) = (\mathbf{y}_r \cdot \boldsymbol{\chi}) \tag{4}$$

where $\boldsymbol{\chi}$ is the vector of F's variables, which range over some large, fixed prime field $\mathbb{F}_p$, and the vectors $\mathbf{v}_r$, $\mathbf{w}_r$, $\mathbf{y}_r$ each define linear combinations over the variables $\boldsymbol{\chi}$. Each equation (indexed by r) can be thought of as encoding a two-input multiplication gate in the arithmetic circuit computing F, with $\mathbf{v}_r$ indicating each variable's contribution (if any) to the gate's left input, $\mathbf{w}_r$ indicating each variable's contribution to the gate's right input, and $\mathbf{y_r}$ indicating the variable's relation to the gate's output. We say that Q has size $\rho \triangleq |\boldsymbol{\chi}|$ and degree d.

Crucially, Pinocchio's evaluation key (used by the prover to create his proof) contains cryptographic key material for each variable $\chi \in \boldsymbol{\chi}$, and the structure of that key material depends directly on which (and how) χ participates in each of the d equations in Equation (4), i.e., on the value of χ's entry in each of the vectors $\mathbf{v}_r$, $\mathbf{w}_r$, $\mathbf{y}_r$.

From QAPs to MultiQAPs If we decompose F into simpler functions F_i, then we can create a corresponding QAP Q_i for each F_i. Suppose we wish to connect Q_0, which has some variables $\mathbf{z_0}$ representing F_0's output, with Q_1, which has some variables $\mathbf{z_1}$ representing F_1's input, with $|\mathbf{z_0}| = |\mathbf{z_1}|$. Since F_0 and F_1 are different functions, $\mathbf{z_0}$ and $\mathbf{z_1}$ undoubtedly participate in different equations in Q_0 and Q_1, and hence, as explained above, will have different key material representing $\mathbf{z_0}$ and $\mathbf{z_1}$. As a result, a digest for $\mathbf{z_0}$ will be completely different from a digest for $\mathbf{z_1}$, *even if* $\mathbf{z_0} = \mathbf{z_1}$! We could fix this by combining Q_0 and Q_1 into a single QAP and adding equations requiring that $\mathbf{z_0} = \mathbf{z_1}$, but then we lose the benefits we hoped to gain from decomposing F.

Instead, we combine all of the $(Q_i)_{i\in[m]}$ into a single MultiQAP $Q^\star$. $Q^\star$ has the same equations and variables $\boldsymbol{\chi}$ used in the Q_i. In addition, for each variable s that we wish to share between some subset $\hat{\mathbf{Q}}$ of the Q_i, we add a new variable $\hat{s}$ to a new bus bank associated with $\hat{\mathbf{Q}}$, and we add an equation relating $\hat{s}$ to the local copy of s in each of the Q_i in $\hat{\mathbf{Q}}$. Continuing our earlier example, we will introduce a new bus for variables $\hat{\mathbf{z}}$ with $|\hat{\mathbf{z}}| = |\mathbf{z_0}| = |\mathbf{z_1}|$, and for each $\hat{z}$ in $\hat{\mathbf{z}}$, we will add an equation:

$$z_0 + z_1 = \hat{z} \tag{5}$$

relating it to the corresponding variables in Q_0 and Q_1. By adding the $\hat{\mathbf{z}}$ bus as a layer of indirection, it no longer matters if z_0 is used differently in Q_0 than z_1 is in Q_1; the prover can create a single digest $D_{\hat{\mathbf{z}}}$ representing the values on the bus, and the verifier can use this digest when checking the correct execution of Q_0, as well as that of Q_1, just as in the example in §2.1.1, when computing Equations (2) and (3). Because the verifier only accepts proof schedules with trivial digests for all other local banks (Definition 1), when she verifies a proof of Q_0, all of the variables in Q_1 are set to 0, and hence Equation (5) says that $z_0 = \hat{z}$, whereas when she verifies a proof of Q_1, all of the variables in Q_0 are set to 0, and hence (5) says that $z_1 = \hat{z}$.

If we follow these steps to combine m sub-QAPs $(Q_i)_{i\in[m]}$, each of size ρ_i and degree at most d, along with the buses connecting them, into a single MultiQAP $Q^\star$, then $Q^\star$ has size $\rho^\star = |\mathbf{s}| + \sum_{i\in[m]} \rho_i$ and degree $d^\star = d + |\mathbf{s}|$, where $\mathbf{s}$ includes all intermediate variables shared between the Q_i. By choosing a decomposition from F to $(F_i)_{i\in[m]}$ that exploits the structure of F, Geppetto's compiler can ensure that most variables are local to one F_i, so we typically achieve $|\mathbf{s}| \ll d$. Since each step in a proof schedule considers only one Q_i at a time, the size and degree of the "active" QAP within $Q^\star$ is only slightly larger than the original Q_i. Thus, MultiQAPs enable state sharing across sub-QAPs without significantly increasing the prover's costs beyond what is required to handle the sub-QAPs of the schedule individually.

2.1.4 Other Techniques for Stateful Computations

Prior work explores other, largely complementary mechanisms for handling verifiable computations over state. As discussed in §2.1.1, a classic way to condense state is to commit to it via a hash [8, 12, 29, 43]. When specifying the IO to a function F, the verifier only gives the hash value $h = H(\mathbf{u})$. The prover supplies the full data values and, as part of the verifiable computation, hashes the data and proves that the hash matches the

one supplied by the verifier. A recent system, Pantry [16], implements such collision-resistant hashing on top of the existing QAP-based Pinocchio [46] and Zaatar protocols [49].

As shown in §7.2, using MultiQAPs is much cheaper than hashing when all (or most) of the state will be used in a given computation. Thus, MultiQAPs will typically be advantageous when passing state between computations, such as between mappers and reducers in a MapReduce job or within a decomposed program such as the one shown in Figure 1, since a good compiler will ensure that state is passed between computations only if both computations actually need it. MultiQAPs are also advantageous for IO when the verifier's inputs can be split in two pieces, a (mostly) static and a dynamic portion, that interact in each computation. For example, we might see this pattern if the computation takes in a large dataset and a small query, and the query needs to verifiably compute on most of the dataset.

In contrast, hashing is advantageous when the inputs are large, but the verifiable computation only accesses a small portion of the input at a time. For example, if the computation is over a large database but any given computation only selects a handful of records, then hashing makes sense. Hashing is also suitable for transferring state between verifiable computations performed with keys created by mutually distrusting parties.

As an orthogonal contribution, Pantry uses hashes to build a RAM abstraction based on Merkle trees [43], though subsequent work [10, 54] suggests that handling RAM via memory routing networks [8] performs better for most memory sizes. Regardless, these techniques are orthogonal to Geppetto in the sense that they focus on dynamic RAM access *within* a computation/QAP, rather than on transferring state between computations. Indeed, routing networks would likely be the most efficient way to allow a given Geppetto sub-QAP to incorporate a RAM abstraction. Recent work demonstrates [54] that such abstractions can be naturally integrated with Geppetto's compilation-based approach.

Finally, in concurrent work, Backes et al. modify the Pinocchio protocol to incorporate a linearly homomorphic MAC in order to optimize computing on authenticated data [3]. Using signed Geppetto commitments offers an alternate approach; we defer evaluating the tradeoffs to future work.

2.2 Verifiable Crypto and Bootstrapping Proofs

In theory, we should be able to verify cryptographic computations (e.g., a signature verification) just like any other computation. In practice, as discussed in §1, a naive embedding of cryptographic computations into the field $\mathbb{F}_p$ that our MultiQAPs operate over leads to significant overhead. In §5, we use a careful choice of cryptographic primitives and parameters to build a large class of crypto operations (e.g., signing, verification, encryption) using elliptic curves built "natively" on $\mathbb{F}_p$. For example, this makes it cheap to verify computations on signed data, since the data and the signature both "live" in $\mathbb{F}_p$. Prior work used such tailoring for unbounded bootstrapping [9] and hashing [9, 16].

Our most complex application of this technology is a form of proof bootstrapping [11, 53], which we use to address the main drawback of CP schemes. With CP schemes, including our MultiQAP-based scheme, the size of the cryptographic evidence—and the verifier costs—grow linearly with the number of digests and proofs. While often acceptable in practice, these costs can be reduced to a constant by using another instance of our CP scheme to outsource the verification of all of the cryptographic evidence according to a target proof schedule.

More formally, let $\mathsf{Verify}_{\sigma^\star}(\mathbf{D},\Pi)$ be the function checking that a scheduled CP proof cryptographically verifies, where $\mathbf{D}$ and Π are the collections of digests and proofs used in the schedule $\sigma^\star$. We recursively apply Geppetto to generate a quadratic program $Q_{\sigma^\star}$ for $\mathsf{Verify}_{\sigma^\star}$. This yields another, more efficient verifier $\mathsf{Verify}^\circ_{\sigma^\star}(D^\circ,\pi^\circ)$ with a single, constant-sized digest D° of $\mathbf{D}$, Π, and all intermediate variables used to verify them according to $\sigma^\star$, and with a single constant-sized proof π° to verify, now in constant time.

We further observe that $\mathsf{Verify}^\circ_{\sigma^\star}$ need not be limited to just verifying the execution of $\mathsf{Verify}_{\sigma^\star}$. For example, suppose an authority the client trusts (e.g., the US FDA) cryptographically signs the verification keys for $\mathsf{Verify}_{\sigma^\star}$, and we define $\mathsf{Verify}^\circ_{\sigma^\star}$ to first verify the signature on the keys before using them to run $\mathsf{Verify}_{\sigma^\star}$. If we use Geppetto's option to make digests and proofs perfectly hiding, then the verifier checks a constant-sized proof and learns that a trusted algorithm (for example, a medical diagnosis) ran correctly over her data, but she learns nothing about the algorithm. Thus, a client can efficiently and verifiably outsource computations with proprietary algorithms.

Although the general idea of bootstrapping is well-known [11, 53], its practicality relies on careful cryptographic choices to support an efficient embedding. Recent work [9] instantiated and implemented an embedding that supports bootstrapping an unbounded number of proofs but this generality comes at a cost (§5).

In §5, we explore a pragmatic alternative that supports only bounded-length schedules but can achieve better performance. Intuitively, the construction is based on the observation that the algorithm $\mathsf{Verify}^\circ_{\sigma^\star}$ described above, can itself be scheduled and bootstrapped. In other words, given an initial CP scheme $\mathcal{P}$, we define a second CP scheme $\mathcal{P}'$ that verifies a schedule for $\mathcal{P}$ of length at most L. If our application requires a schedule longer than L, we can define a third CP scheme $\mathcal{P}''$ that condenses digests and proofs from $\mathcal{P}'$. With enough levels, we can ensure that the verifier only receives a constant-sized digest and proof, and hence only performs work linear in the overall computation's IO, regardless of how the prover decomposes F into smaller functions. The overall protocol can be thought of as a tree of proof schedules, where the arity of each node is L, and as we move towards the root of the tree, each level condenses the digests and proofs from the nodes above it. Our full paper [23] formalizes this process, adapting the usual proof-of-a-proof bootstrapping techniques [11, 53].

Using multiple levels reduces both the key sizes and the prover's work. For example, suppose the application produces N proofs for $\mathcal{P}$. The naïve approach of using a single recursive level $\mathcal{P}'$ would require a key capable of consuming all N proofs. Instead, with multiple levels, we can design $\mathcal{P}'$ to consume $\sqrt{N}$ proofs from $\mathcal{P}$ and design $\mathcal{P}''$ to consume $\sqrt{N}$ proofs from $\mathcal{P}'$.

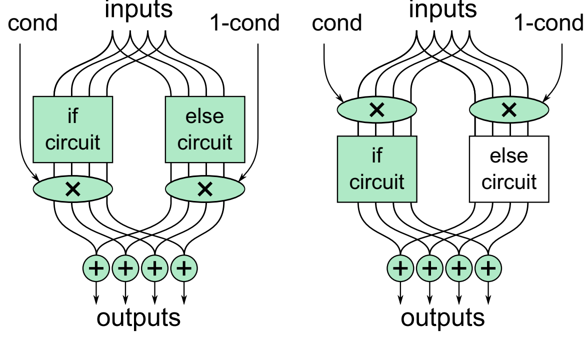

Figure 3: **Energy-Saving Circuits.** *Moving the multiplex step can nullify expensive crypto operations, since at runtime, in one of the two circuit blocks, every wire inside takes on the value zero.*

The resulting keys will be $O(\sqrt{N})$, instead of size N for a single recursive layer.

Our CP definitions and theorems (§3), as well as our compiler (§6.4), support multiple-levels of bootstrapping through such recursion. For example, our compiler (§6.4) rewrites source programs to replace outsourced function calls by proof verification and can be called on its own output.

2.3 Energy-Saving Circuits

Existing verifiable computation systems represent a computation as a quadratic program (informally, a circuit), which results in a program whose size reflects the worst-case computational resources necessary over all possible inputs. For instance, when branching on a runtime-value, Pinocchio's prover interprets and proves both branches and only then joins their results. Concretely, the command `if(b) {x = y} else {x=2*z}` is effectively compiled as `x = 2z + b*(y-2z)`, as shown generically in the left side of Figure 3. Similarly, if a loop has a static bound of N iterations, the prover must perform work for all N, even if the loop typically exits early.

Ideally, we would like to "turn off" parts of the circuit that are not needed for a given input, much the same way hardware circuits can power down parts not currently in use. Geppetto achieves this by observing that in our cryptographic protocol, there is no cryptographic cost for QAP variables that evaluate to zero (however these variables still increase the degree of the QAP, and hence the cost of the polynomial operations the prover performs). Thus, if at compile-time we ensure that *all* intermediate variables for the branch evaluate to 0 in branches that are not taken, then at run-time there is no need to evaluate those branches at all. The right side of Figure 3 shows an example of how we achieve this for branches by applying the condition variable to the *inputs* of each subcircuit, rather than to the outputs. Thus, in contrast with Pinocchio, the prover only does cryptographic work proportional to the path actually taken through the program.

Prior compilers [49] use a related technique that applies the condition variable to the *equations* in each branch, rather than to the inputs. This avoids the need to interpret untaken branches, but produces more constraints than Geppetto in the common case when the branch contains more equations than inputs.

§6.5 explains how our compiler produces energy-saving circuits, while §7.4 quantifies the significant savings we recoup via this technique.

3 Defining Proof Composition

We now give formal cryptographic definitions for the concepts introduced in §2, deferring our concrete protocol to §4.

3.1 Commit-and-Prove Schemes

As discussed in §2.1.1, Geppetto employs three types of digest, one for F's IO, one for the local variables for each F_i, and one for each bus. Each digest, D, may hide the values it represents via randomness o. Without hiding, we use a trivial opening $o = 0$ (and may omit it). We require that all digests of bus values be binding, as otherwise the prover could, say, use one set of values for the bus when proving that F_0 correctly wrote to the bus, while using a different set of values when proving that F_1 correctly read from the bus. In contrast, digests used only in a single proof, e.g., for intermediate local variables, need not be binding, since the verifier only needs to know that there exists an assignment of values to those variables corresponding to a single correct execution. Finally, digests of IO naturally need not be binding since the verifier computes them herself.

As a side note, while Geppetto uses commit-and-prove schemes to prove function executions, such schemes also enable interactive protocols where values are committed, used in proofs, and opened dynamically. For instance, they easily integrate with existing Σ-protocols as employed in anonymous credential systems [5, 17].

Since we are interested in *succinct* proofs, we modify earlier definitions of commit-and-prove schemes [18, 26, 37] to only consider computationally bounded adversaries. As a succinct digest implies that more than one plaintext maps to a given digest value, an unbounded adversary can always "escape" the digest's binding property.

Each MultiQAP $Q^\star$ in our construction defines a relation R from the family $\mathcal{R}$ of all MultiQAPs over a fixed field $\mathbb{F}$. As our security definition has a security parameter $\lambda \in \mathbb{N}$ (which intuitively determines the size of the field $\mathbb{F}$), we actually talk about a sequence of families of polynomial-time verifiable relations $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$.

Definition 2 (Succinct Commit-and-Prove) *Consider ℓ-ary polynomial-time verifiable relations $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ on tuples $\boldsymbol{\chi}$ of a fixed length ℓ.*

A succinct commit-and-prove scheme $\mathcal{P} = (\mathsf{KeyGen} = (\mathsf{KeyGen}_1, \mathsf{KeyGen}_2), \mathsf{Digest}, \mathsf{Prove}, \mathsf{Verify})$ for $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ consists of five polynomial-time algorithms as follows:

- *Key generation is split into two probabilistic algorithms: $\tau \leftarrow \mathsf{KeyGen}_1(1^\lambda)$ takes the security parameter λ as input and produces a trapdoor $\tau = (\tau_{\mathcal{S}}, \tau_{\mathcal{E}})$ (independent of R and consisting of a simulation and extraction component). $(EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R)$ takes the trapdoor and a relation $R \in \mathcal{R}_\lambda$ as input and produces a public evaluation key*

EK and a public verification key VK. To simplify notation, we assume that EK includes a copy of VK, and that EK and VK include digest keys EK_b and VK_b for $b \in [\ell]$.

- $D_b^{(t)} \leftarrow \mathsf{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)})$*: Given an evaluation key for b, message instance t for b ($\chi_b^{(t)}$), and corresponding randomness $o_b^{(t)}$, the deterministic digest algorithm produces a digest $D_b^{(t)}$ of $\chi_b^{(t)}$.*
- $\pi \leftarrow \mathsf{Prove}(EK, \boldsymbol{\chi}, \mathbf{o})$*: Given an evaluation key, messages $\boldsymbol{\chi} \in R$, and openings $\mathbf{o}$, the deterministic prove algorithm returns a succinct proof π; i.e., $|\pi|$ is* $\text{poly}(\lambda)$.
- $\{0,1\} \leftarrow \mathsf{Verify}(VK_b, D_b^{(t)})$*: Given a verification key for b, the deterministic digest-verification algorithm either rejects (0) or accepts (1) the digest $D_b^{(t)}$.*
- $\{0,1\} \leftarrow \mathsf{Verify}(VK, \mathbf{D}, \pi)$*: Given a verification key and ℓ digests $\mathbf{D}$, the deterministic verification algorithm either rejects (0) or accepts (1) the proof π.*

Proof-verification guarantees apply only when each digest $D_b^{(t)}$ in $\mathbf{D}$ either passes the digest-verification algorithm or was computed directly by the verifier.

We define two security requirements below. Standard definitions for correctness and zero-knowledge are in the full paper [23]. First, we require that digests shared across multiple proofs (i.e., those representing bus values) be *binding*, meaning the prover cannot claim the digest represents one set of values in the first proof and a different set of values in the second proof. We collect the indexes of their keys in what we call the *binding digest subset $S \subset [\ell]$*.

Definition 3 (Binding) *The commit-and-prove scheme $\mathcal{P}$ is binding for ℓ-ary relations $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ and binding digest subset $S \subset [\ell]$, if for all efficient $\mathcal{A}$ and any $R \in \mathcal{R}_\lambda$,*

$$\begin{aligned}\Pr[\quad & \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); \tau = (\tau_{\mathcal{S}}, \tau_{\mathcal{E}});\\ & (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);\\ & (b, \chi, o, \chi', o') \leftarrow \mathcal{A}(EK, R, \tau_{\mathcal{E}}):\\ & \chi \neq \chi' \wedge b \in S \wedge\\ & \mathsf{Digest}(EK_b, \chi, o) = \mathsf{Digest}(EK_b, \chi', o') \quad] = \mathtt{negl}(\lambda).\end{aligned}$$

Second, we require that if an adversary creates a set of digests and a proof that Verify accepts, then the adversary must "know" a valid witness, in the sense that this witness can be successfully extracted by "watching" the adversary's execution. Note that the trapdoor the extractor receives from KeyGen_1 is generated independently of relation R and hence cannot make it easier for the extractor to produce its own witnesses.

Definition 4 (Knowledge Soundness) *The commit-and-prove scheme $\mathcal{P}$ is knowledge sound for ℓ-ary relations $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$, if for all efficient $\mathcal{A}$ there is an efficient extractor $\mathcal{E}$ taking the random tape of $\mathcal{A}$ such that, for any $R \in \mathcal{R}_\lambda$,*

$$\begin{aligned}\Pr[\quad & \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); \tau = (\tau_{\mathcal{S}}, \tau_{\mathcal{E}});\\ & (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);\\ & (\mathbf{D}, \pi; \boldsymbol{\chi}, \mathbf{o}) \leftarrow (\mathcal{A}(EK, R) \parallel \mathcal{E}(EK, R, \tau_{\mathcal{E}})):\\ & \left(\exists b \in [\ell].\ \mathsf{Verify}(VK_b, D_b^{(t)}) \wedge D_b^{(t)} \neq \mathsf{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)})\right) \vee\\ & \left(\forall b \in [\ell].\ \mathsf{Verify}(VK_b, D_b^{(t)}) \wedge \mathsf{Verify}(VK, \mathbf{D}, \pi) \wedge \boldsymbol{\chi} \notin R\right)\\ ] & = \mathtt{negl}(\lambda).\end{aligned}$$

3.2 Composition by Scheduling

As discussed in §2, intuitively, we can verify the correct execution of a complex F by verifying simpler functions and using digests to share state between them. We now formalize this intuition by extending knowledge soundness to multiple related proofs that share digests according to a proof schedule.

Definition 5 (Scheduled Knowledge Soundness) *The commit and-prove scheme $\mathcal{P}$ is scheduled knowledge sound for ℓ-ary relations $\{\mathcal{R}_\lambda\}_{\lambda\in\mathbb{N}}$ and binding digest subset $S \subset [\ell]$, if for all efficient $\mathcal{A}$ there is an efficient extractor $\mathcal{E}$ taking the random tape of $\mathcal{A}$ such that, for any $R \in \mathcal{R}_\lambda$,*

$$\begin{aligned}\Pr[\ & \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda);\\ & (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);\\ & (\sigma, \mathbf{D}, \Pi; \boldsymbol{\chi}, \mathbf{o}) \leftarrow (\mathcal{A}(EK, R) \parallel \mathcal{E}(EK, R, \tau)):\\ & \forall D_b^{(t)} \in \mathbf{D}.\ (\mathsf{Verify}(VK_b, D_b^{(t)}) \Rightarrow D_b^{(t)} = \mathsf{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)})) \wedge\\ & \big(\forall D_b^{(t)} \in \mathbf{D}.\ \mathsf{Verify}(VK_b, D_b^{(t)}) \wedge\\ & \ \forall (i, \mathbf{t}) \in \sigma.\ \mathsf{Verify}(VK, \mathbf{D}^{(\mathbf{t})}, \pi_i)\big)\\ & \Rightarrow \forall (i, \mathbf{t}) \in \sigma.\ \boldsymbol{\chi}^{(\mathbf{t})} \in R\\ ] & = 1 - \mathtt{negl}(\lambda),\end{aligned}$$

where $\mathbf{D}^{(\mathbf{t})}$ indicates a digest instance t for each bank b used in a given proof (and default digests of 0 values for any bank not used), and $\boldsymbol{\chi}^{(\mathbf{t})}$ represents the digested values.

Theorem 1 (Scheduled Knowledge Soundness) *If a CP $\mathcal{P}$ is knowledge sound and binding for ℓ-ary relations and binding digest subset, then it is scheduled knowledge sound for the same relations and subset.*

The proof of Theorem 1 can be found in the full paper [23]. Intuitively, it follows from extracting valid digest openings from all subproofs, and leveraging the binding property of the bus digests to guarantee consistency across subproofs.

4 Geppetto's CP Protocol

We now construct an efficient commit-and-prove protocol for ℓ-ary relations $\{\mathcal{R}_{Q^\star_\lambda}\}_{\lambda\in\mathbb{N}}$ (see §3.1) defined by a MultiQAP $Q^\star$ derived from multiple QAPs Q_i, as described in §2.1.3.

4.1 MultiQAPs as Polynomials

We use Pinocchio's technique (which originated with Gennaro et al. [29]) to lift quadratic programs to polynomials.

Given MultiQAP $Q^\star$, of size $\rho^\star$ and degree $d^\star$, we first define a set $\mathcal{D}$ of $d^\star$ "root values" of the form $r \in \{2^i\}_{i=1}^{d^\star}$,[5] and we define the polynomial $\delta(x)$ as the polynomial with all $r \in \mathcal{D}$ as roots. Recalling §2.1.3, we then define a set $\mathcal{V}$ of $\rho^\star$ polynomials $v_k(x)$ by interpolation over the roots in $\mathcal{D}$ such that for $k \in [\rho^\star], r \in \mathcal{D}$: $v_k(r) = \mathbf{v}_{r,k}$. Each of the k polynomials essentially summarizes the effect one of $\boldsymbol{\chi}$'s variables has on the computation. We define similar sets $\mathcal{W}$ and $\mathcal{Y}$ using the vectors $\mathbf{w}_r$ and $\mathbf{y}_r$.

We say that the polynomial MultiQAP is satisfied by χ if $\delta(x)$ divides $p(x)$, where:

$$p(x) = \left(\sum_{k=0}^{\rho} \chi_k \cdot v_k(x)\right) \cdot \left(\sum_{k=0}^{\rho} \chi_k \cdot w_k(x)\right) - \left(\sum_{k=0}^{\rho} \chi_k \cdot y_k(x)\right) .$$

We use MultiQAPs to prove statements about shared state. To achieve this, the polynomials corresponding to bus values need to fulfill an additional condition. We say that a bus bank B_b is *commitment compatible* if (i) the polynomials in each set $\{y_k(x)\}_{k \in B_b}$ are linearly independent, meaning that no linear combination of them cancels all coefficients, and (ii) all polynomials in the set $\{v_k(x), w_k(x)\}_{k \in B_b}$ are 0. The first property is crucial for commitments to be binding, while the second improves performance and facilitates zero-knowledge when using externally generated commitments.

By inspection of Equation (5), the buses in our MultiQAP construction in §2.1.3 are commitment compatible. Concretely, continuing our example from that section, Equation (5) will be encoded as the QAP equation:

$$(0 + \cdots + 0)(0 + \cdots + 0) = (1 \cdot z_0 + 1 \cdot z_1 + (-1) \cdot \hat{z}).$$

4.2 Commit-and-Prove Scheme for MultiQAPs

Geppetto's protocol inherits techniques from Pinocchio [46]; the key differences are starting with MultiQAPs instead of QAPs, and splitting the prover's efforts into separate digest and proof computations.

We present our protocol in terms of a generic quadratic encoding E [29]. In our implementation, we use an encoding based on bilinear groups. Specifically, let e be a non-trivial bilinear map [13] $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and let g_1, g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. To simplify notation, we define the encoding $E(x)$ to be either g_1^x or g_2^x depending on whether it appears on the left or the right side of a product $*$.

Below, each $B_b \in \mathbf{B}$ represents a subset of $[\rho^\star]$, and we use the commit-and-prove message $\chi_b^{(t)}$ to represent the values of bank instance $B_b^{(t)}$.

Protocol 1 (Geppetto)

- $\tau \leftarrow \mathsf{KeyGen}_1(1^\lambda)$:
  Choose $s, \{\alpha_{v,b}, \alpha_{w,b}, \alpha_{y,b}\}_{b \in [\ell]}, r_v, r_w \xleftarrow{R} \mathbb{F}$. *Construct* τ *as* $(\tau_{\mathcal{S}}, \tau_{\mathcal{E}}) = (s, \{\alpha_{v,b}, \alpha_{w,b}, \alpha_{y,b}\}_{b \in [\ell]}, r_v, r_w), (r_v, r_w)$.
- $(EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R_{Q^\star})$:
  Choose $\{\gamma_b, \beta_b\}_{b \in [\ell]} \xleftarrow{R} \mathbb{F}$. *Set* $r_y = r_v \cdot r_w$. *To simplify notation, define* $E_v(x) = E(r_v x)$ *(and similarly for* E_w *and* E_y*).*
  For the MultiQAP $Q^\star = (\rho^\star, d^\star, \mathbf{B}, \mathcal{V}, \mathcal{W}, \mathcal{Y}, \delta(x))$, *construct the public evaluation key EK as:*

$$(EK_b)_{b \in [\ell]}, \quad (E(s^i))_{i \in [d]}, \quad E_v(\delta(s)), E_w(\delta(s)), E_y(\delta(s))$$

  where each bank's digest key EK_b *is defined as:*

$$\Big(E_v(v_k(s)), \quad E_w(w_k(s)), \quad E_y(y_k(s)) \\ E_v(\alpha_{v,b} v_k(s)), \quad E_w(\alpha_{w,b} w_k(s)), \quad E_y(\alpha_{y,b} y_k(s)), \\ E(\beta_b(r_v v_k(s) + r_w w_k(s) + r_y y_k(s))), \Big)_{k \in B_b} \\ E_v(\alpha_{v,b}\delta(s)), \quad E_w(\alpha_{w,b}\delta(s)) \quad E_y(\alpha_{y,b}\delta(s)), \\ E_v(\beta_b\delta(s)), \quad E_w(\beta_b\delta(s)), \quad E_y(\beta_b\delta(s)).$$

  Construct the public verification key VK as:

$$(VK_b)_{b \in [\ell]}, \quad E(1), \quad E_y(\delta(s)) ,$$

  where each bank's digest verification key VK_b *is:*

$$VK_b = E(\alpha_{v,b}), E(\alpha_{w,b}), E(\alpha_{y,b}), E(\gamma_b), E(\beta_b\gamma_b) .$$

  Additionally VK includes digest keys EK_b *for digests that the verifier computes (e.g., for IO banks). Since EK and VK are public, the split into prover and verifier keys is primarily designed to reduce the verifier's storage overhead.*
- $D_b^{(t)} \leftarrow \mathsf{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)})$:
  Parse $o_b^{(t)}$ *as* (o_v, o_w, o_y).
  If B_b *is an IO bank, simply return:*

$$E_v(v^{(b)}(s)), \quad E_w(w^{(b)}(s)), \quad E_y(y^{(b)}(s)),$$

  where $v^{(b)}(s) = \sum_{k \in B_b} \chi_k v_k(s) + o_v \delta(s)$ *(and similarly for* $w^{(b)}(s)$ *and* $y^{(b)}(s)$*). Since the verifier typically computes these digests,* o_v *is typically 0. Note that all of these terms can be computed using the values in* VK_b*, thanks to the linear homomorphism of the encoding E.*
  For any other bank, compute:

$$E_v(v^{(b)}(s)), \quad E_w(w^{(b)}(s)), \quad E_y(y^{(b)}(s)), \\ E_v(\alpha_{v,b} v^{(b)}(s)), \quad E_w(\alpha_{w,b} w^{(b)}(s)), \quad E_y(\alpha_{y,b} y^{(b)}(s)), \\ E(\beta_b(r_v v^{(b)}(s) + r_w w^{(b)}(s) + r_y y^{(b)}(s))) .$$

  Note that all of these terms can be computed using the values in EK_b*. The values above constitute an extractable digest of the* $\chi_b^{(t)}$ *values, perfectly hidden via* $o_b^{(t)}$*. For commitment-compatible buses, this digest is also binding. Furthermore, for all commitment-compatible buses,* $v^{(b)}(s), w^{(b)}(s), o_v, o_w$ *are all 0, so the digest above simplifies to:*

$$E_y(y^{(b)}(s)), E_y(\alpha_{y,b} y^{(b)}(s)), E(\beta_b(r_y y^{(b)}(s))) ;$$

- $\pi \leftarrow \mathsf{Prove}(EK, \chi, \mathbf{o})$: *Parse each* $o_b \in \mathbf{o}$ *as* $(o_{b,v}, o_{b,w}, o_{b,y})$ *and use the coefficients* χ *to calculate:*

$$v(x) = \sum_{k \in [\rho^\star]} \chi_k v_k(x) + \sum_{b \in [\ell]} o_{b,v} \delta(x),$$

  and similarly for $w(x)$, *and* $y(x)$.
  Just as in a standard QAP proof [29], calculate $h(x)$ *such that* $h(x)\delta(x) = v(x)w(x) - y(x)$, *that is, the polynomial that proves that* $\delta(x)$ *divides* $v(x)w(x) - y(x)$. *Compute the proof as* $\pi \leftarrow E(h(s))$ *using the* $E(s^i)$ *terms in EK.*

[5] Choosing roots of this form enables our C++ library to implement an efficient $d^\star \log d^\star$ algorithm [15] for the prover's polynomial division.

- $\{0,1\} \leftarrow \mathsf{Verify}(VK_b, D_b^{(t)})$: *Verify digest $D_b^{(t)}$ by checking*

$$E_v(v^{(b)}(s)) * E(\alpha_{v,b}) = E_v(\alpha_{v,b} v^{(b)}(s)) * E(1) \quad (6)$$
$$E_w(w^{(b)}(s)) * E(\alpha_{w,b}) = E_w(\alpha_{w,b} w^{(b)}(s)) * E(1) \quad (7)$$
$$E_y(y^{(b)}(s)) * E(\alpha_{y,b}) = E_y(\alpha_{y,b} y^{(b)}(s)) * E(1) \quad (8)$$

and the β check:

$$E\left(\beta_b(r_v v^{(b)}(s) + r_w w^{(b)}(s) + r_y y^{(b)}(s))\right) * E(\gamma_b) = \quad (9)$$
$$\left(E_v(v^{(b)}(s)) + E_y(y^{(b)}(s))\right) * E(\beta_b \gamma_b) + E(\beta_b \gamma_b) * E_w(w^{(b)}(s)).$$

(For buses, we do not require the checks in Equations (6) *and* (7)*, and we can simplify the β check (Eqn* (9)*).)*

- $\{0,1\} \leftarrow \mathsf{Verify}(VK, D_0, \ldots, D_{\ell-1}, \pi)$: *Combine the digests and perform the divisibility check on the proof term $E(h(s))$ in π:*

$$\left(\sum_{b \in [\ell]} E_v(v^{(b)}(s))\right) * \left(\sum_{b \in [\ell]} E_w(w^{(b)}(s))\right) \quad (10)$$
$$- \left(\sum_{b \in [\ell]} E_y(y^{(b)}(s))\right) * E(1) = E(h(s)) * E_y(\delta(s)) .$$

As described, the protocol supports non-interactive zero-knowledge proofs, in addition to verifiable computation. For applications that only desire the latter, the multiples of $\delta(s)$ in the EK and the use of digest randomizations o may be omitted.

Theorem 2 *Protocol 1 has binding digests, as defined by Definition 3 under the d-SDH assumption.*

Theorem 3 *Protocol 1 is a knowledge-sound commit-and-prove scheme, as defined by Definition 4.*

Theorem 4 *Protocol 1 is a perfectly zero-knowledge commit-and-prove scheme.*

We refer to the full paper [23] for the proofs of these theorems and the definition of their assumptions. Like the protocol, the proofs inherit their techniques from Pinocchio.

5 Verifiable Crypto Computations

Background Pinocchio, along with the systems built atop it, instantiates its cryptographic protocol using pairing-friendly elliptic curves. Such curves ensure good performance and compact keys and proofs. An elliptic curve E defines a group of prime order p' where each element in the group is an (x,y) point, with x and y drawn from a second field $\mathbb{F}_p$ of large prime characteristic p. When Pinocchio is instantiated with such a curve, the QAPs (and hence all verifiable computations) are defined over $\mathbb{F}_{p'}$, and hence code that compiles naturally to operations on $\mathbb{F}_{p'}$ is cheap.

Approach At a high-level, we choose the curve E we use to instantiate Geppetto such that the group order "naturally supports" operations on a second curve $\tilde{E}$, which we can use for any cryptographic scheme built on $\tilde{E}$, e.g., anything from signing with ECDSA to the latest attribute-based encryption scheme.

In more detail, suppose we want to verify ECDSA signatures over an elliptic curve $\tilde{E}$ built from points chosen from $\mathbb{F}_q$. If we instantiate Geppetto using a pairing-friendly elliptic curve E with a group of prime order $p' = q$, then operations on points from $\tilde{E}$ embed naturally into our QAPs, meaning that basic operations like adding two points cost only a handful of cryptographic operations, rather than hundreds or thousands required if p' did not align with q.

Bootstrapping As described in §2.2, proof bootstrapping is a particularly compelling example of verifying cryptographic operations, since it allows us to condense a long series of proofs and digests into a single proof and digest.

Remarkably, Karabina and Teske [35] show that it is possible to generate two MNT curves [45] E and $\tilde{E}$ that are pairing friendly and, more importantly, $\tilde{E}$ can be embedded in E, *and* E can be embedded in $\tilde{E}$.

Ben-Sasson et al. [9] recently instantiated and implemented such curves to bootstrap the verification of individual CPU instructions. Geppetto can use a similar approach to achieve unbounded bootstrapping of entire QAPs. Specifically, we could instantiate two versions of Geppetto, one built on E that condenses proofs consisting of points from $\tilde{E}$ and another built on $\tilde{E}$ that condenses proofs consisting of points from E.

Unfortunately, there are drawbacks to using the curves Ben-Sasson et al. found. First, they were only able to find a pair of curves that provide 80 bits of security. Finding cycles of performant curves for the more standard 128-bit setting appears non-trivial, since just finding 80-bit curves required over 610,000 core-hours of computation. Second, the MNT curve family is not the most efficient family at higher security levels, and achieving a cycle requires larger-than-usual fields, creating additional inefficiency [9].

To estimate the costs of using MNT curves at the 128-bit security level used by Pinocchio, we coded up all of the relevant curve operations in Magma [14] and counted the group operations required. We made very optimistic assumptions about the optimal implementation of the curves, e.g., by assuming that the operations employ all available EC tricks within the pairing computation, even though the actual curves may not allow for them. Even under these assumptions, our measurements indicate that key and proof generation, as well as IO verification, for Geppetto's first batch of proofs would be 34-77× slower than a standard Pinocchio-style proof, while the constant pairing-based portion of proof verification would be 11× slower; subsequent batches would cost more, due to technical challenges in the way the curves fit together [9].

As a pragmatic alternative, we use a sequence of nested curves (an option suggested previously [9, Footnote 10]) to instantiate and implement *bounded bootstrapping*, Specifically, we instantiate one version of Geppetto with the same highly efficient BN curve [6] employed by Pinocchio. We use the BN curve to generate a collection of digests and proofs for our MultiQAP-based CP scheme. We then construct a second curve capable of efficiently embedding the BN curve operations. When instantiated with the second curve, Geppetto can efficiently verify crypto operations on the BN curve. Thus,

a verifier can, for example, check signatures on the verification key built on the BN curve and then use that key to verify the BN digests and proofs. To gain greater scalability, this process can be repeated with a bounded number of additional carefully constructed curves, each used to verify the digests and proofs from the previous curve. Unfortunately, none of the curves can efficiently embed later curves, and hence when generating keys, the client must ultimately commit to the maximum number of BN proofs that will be verified. Fortunately, our use of energy-saving circuits saves the prover effort if it ends up using fewer proofs.

Details We construct bilinear systems, $\mathcal{G}_{IN}$ and $\mathcal{G}_{OUT}$. To achieve this at the 128-bit security level, we instantiate $\mathcal{G}_{IN}$ using a Barreto-Naehrig (BN) elliptic curve [6], and then construct $\mathcal{G}_{OUT}$ accordingly with the Cocks-Pinch method [21]. Roughly, the latter constructs a pairing-friendly curve by outputting a finite field corresponding to a given, prescribed group order. We fix the prime p from the BN parameterization as the group order, so that the output of the Cocks-Pinch algorithm is the prime $\tilde{p}$ (as well as the other parameters required in the description of $\mathcal{G}_{OUT}$). The following lemma makes this explicit in a special case that is of most interest in the current work.

Lemma 1 *Let $x \in \mathbb{Z}$ be such that $p = 36x^4 + 36x^3 + 24x^2 + 6x + 1$ and $p' = 36x^4 + 36x^3 + 18x^2 + 6x + 1$ are prime. If*

$$\tilde{p} = 5184x^8 + 10368x^7 + 12204x^6 + 8856x^5 + 4536x^4 + 1548x^3 + 363x^2 + 48x + 4 \quad (11)$$

is also prime, then there exists both an elliptic curve $E/\mathbb{F}_p$ of order $\#E(\mathbb{F}_p) = p'$ with embedding degree $k = 12$ (with respect to p'), and an elliptic curve $\tilde{E}/\mathbb{F}_{\tilde{p}}$, such that its order $\#\tilde{E}(\mathbb{F}_{\tilde{p}})$ is a multiple of p and $\tilde{E}$ has embedding degree $\tilde{k} = 6$ (w.r.t. p).

Our full paper contains proofs and construction details [23].

To construct additional nesting curves, given a group order, we once again apply the Cocks-Pinch approach to produce a sequence of curves $E^{(i)}$, defined over prime fields $\mathbb{F}_{p_i}$, respectively, such that p_i divides $\#E^{(i+1)}(\mathbb{F}_{p_{i+1}})$. Each hop creates a larger curve, and hence will eventually produce curves equal to or larger than the MNT curves that support unbounded bootstrapping. For example, for the first Cocks-Pinch curve, $\tilde{p}$ is 509 bits (with embedding degree 6), and the next two levels are 1023 bits and 2055 bits with embedding degrees 3 and 1.

Even when we reach these larger sizes, the inner layers (especially the BN curve where most of the "real" computation happens) are still more efficient than the MNT curves, and even at comparable sizes, exponentiations on the Cocks-Pinch curves are faster due to a CM endomorphism (not available for MNT curves) and a $\mathbb{G}_2$ cubic twist. Of course, for sufficiently large problems, the unbounded approach eventually offers better performance.

6 Implementation

The Geppetto system includes a library for guiding the compilation of banks and buses, a cryptographic compiler that operates on C programs via LLVM, and libraries that support common programming patterns and bootstrapped computation.

Although it has been applied to over 10,000 lines of C and supports many LLVM instructions, Geppetto imposes semantic restrictions on source programs, thereby reflecting limitations of compilation to QAP encodings. For instance, it offers almost no support for computations on pointers. Recent work shows how to remove many of Geppetto's restrictions [54].

We first explain our programming model by example, then describe the design and selected features of our compiler, and finally discuss C libraries and programming patterns.

6.1 Programming Model

A Geppetto programmer defines the structure of outsourced computations, their compound proofs, and the shared buses that connect them, thereby explicitly controlling cost and amortization of proof and digest generation. This structure is embedded in source C programs via library invocations. (The design of higher-level syntactic sugar and programming abstractions is left as future work.)

From the verifier's viewpoint, Geppetto's C programming model is reminiscent of remote procedure calls (RPCs). The programmer marks some function calls as *outsourced*, indicating that the verifier should remote the calls to an untrusted machine, then verify their results using the accompanying cryptographic evidence. This approach provides a clear operational specification of the verified computation, even for complex proof schedules: when the main program of the verifier completes, its outputs and return values must be the same as those that would be obtained by executing the entire program on a single trusted machine.

We illustrate the definition of outsourced functions on the Geppetto program `sample.c`, outlined in Figure 4. The program defines some application code (elided), notably `compute()` that operates on a matrix and a vector of integers.

The programmer intends to fix the matrix across instances of `compute`, and vary the input vector. To this end, `sample.c` declares three banks for verifiable outsourced computation. For instance, relying on the Geppetto header file, `BANK(QUERY, vector)` defines a `QUERY` bank datatype that carries values of type `vector`, and functions like `save_QUERY` and `load_QUERY`, analogous to RPC marshalling and unmarshalling functions. By convention, each bank instance can be assigned only once, and must be assigned before being loaded.

The program then defines two functions: `job`, the outsourced function, and `main`, that repeatedly calls `job` and processes its arguments. Note that the call to `job` is marked as `OUTSOURCE`, and that the digest `db` to the largest input `M` is computed just once, outside the loop.

- When compiling `sample.c` natively outside Geppetto, `geppetto.h` provides trivial definitions that implement `DATA`, `QUERY`, and `RESULT` as in-memory buffers and `OUTSOURCE` as a local call: `OUTSOURCE(job, db, q)` is replaced with `job(db, q)`.
- During *compilation*, Geppetto interprets the outsourced function of `sample.c`, using symbolic values for the pay-

```
#include "geppetto.h"  // Geppetto banks and proofs

// application code
typedef struct { int M[SIZE][SIZE]; ...} bigdata;
typedef struct { int x[SIZE];       ...} vector;
void compute(bigdata *db, vector *in, vector *out);

BANK(DATA, bigdata)  // we define 3 IO banks
BANK(QUERY, vector)
BANK(RESULT, vector)

RESULT job(DATA db, QUERY in) {
  bigdata M;
  vector query, result;
  load_DATA(db,&M);
  load_QUERY(in,&query);
  compute(&M, &query, &result);
  return (save_RESULT(&result));
}
int main() {
  bigdata M;
  vector query[N], result[N];
  ... // prepare the data & queries
  DATA db = save_DATA(&M); // digest M once into db
  for (i=0; i<N; i++) {
    QUERY q = save_QUERY(&query[i]);
    RESULT r = OUTSOURCE(job, db, q);
    load_RESULT(r,&result[i]);
  }
  ... // do something with the results
}
```

Figure 4: **Example Geppetto Program** (`sample.c`).

load of its input banks, and generates a public key pair (EK, VK).

- In *prove mode*, using EK, Geppetto interprets `sample.c` with concrete values to produce cryptographic digests (**D**) for each bank; it intercepts `OUTSOURCE` to accumulate intermediate values during the execution of `job` and to produce a proof (π) for each outsourced call.
- In *verify mode*, using VK, Geppetto produces a version of the program that replaces bank loads and outsource calls with cryptographic verifications; this version can then be natively compiled with `clang -DVERIFY sample.c`.

In both modes, the execution flow of `main` determines the schedule (σ) of calls to outsourced functions.

In more details, in verify mode, verification keys are initially loaded from files, banks are supplemented with cryptographic functions for verifying digests, and `OUTSOURCE(job, db, q)` is replaced with the function call `verify_job(db, q)`. In Figure 5, we show the implementation of `verify_job`, generated by Geppetto during the Geppetto compilation of `sample.c` and included during its native compilation with the `-DVERIFY` flag. Just like `job`, `verify_job` takes two banks and returns a bank. The input banks propagate previously computed (or verified) digests from the caller; in particular, the `bigdata` digest is shared across all calls. The function loads and digests the prover's proposed value for the output bank, verifies the local bank's digest

```
RESULT verify_job(DATA b0, QUERY b1) {
  digest D[4];
  D[0] = b0->d; // use digest produced by save_DATA
  D[1] = b1->d; // use digest produced by save_QUERY
  RESULT b2 = load_redigest_RESULT();
  D[2] = b2->d;
  load_verify_digest(&STATE.vk, &D[3], LOCALS);
  proof pi;
  load_proof("job", &pi);
  verify_proof(&STATE.vk, &pi, 4, D);
  return b2;
}
```

Figure 5: **Simplified Verification Example.** *Geppetto replaces the original outsourced function* `job` *with a version that loads the function result and cryptographic evidence and then verifies that the function was computed correctly.*

evidence, and verifies the computation's proof. If any verification fails, the program exits with an error. Otherwise, the resulting bank `b2` carries the correct response to the outsourced computation.

6.2 MultiQAP Programming Patterns

Geppetto provides additional support for common commit-and-prove patterns, coded as generic C libraries.

Sequential Loops Many large computations consist of a main loop with a code *body* that updates *loop variables* at every iteration, and also reads (but does not modify) *outer variables*.

Geppetto provides a generic template for outsourcing each loop iteration (or, more generally, for outsourcing fixed numbers of iterations that fit within a single QAP), with a bank for the outer variables; hence the cost to digest and verify the outer bank is amortized across all loop iterations.

What about the loop variables? Recall that our commit-and-prove scheme requires that each bank be assigned *at most once* in every proof. Thus, we use *two* buses for the loop variables, alternating between odd and even iterations of the loop, and we compile the loop body *twice*, once reading the even loop variables and writing the odd loop variables, and once the other way round. Hence, our generic template defines three banks, two outsourced functions, and a refined loop that alternates calls between the two. The verifier then checks two digests and one proof for each iteration, except for the first iteration (where it computes a digest of the initial values of the loop variables) and the last (where it recomputes a digest of the final values returned by the prover).

MapReduce Geppetto also provides a few generic templates for parallel loops (like `sample.c` above) and MapReduce computations. As with sequential loops, for MapReduce computations, we use a series of buses to succinctly share potentially many variables between mappers and reducers. Specifically, we adopt Pantry's model [16] in which M mappers feed R reducers. Geppetto compiles a MapReduce job into a MultiQAP with two sub-QAPs (Q_m for the mapper computation and Q_r for the

reducer computation) with $\mathtt{max}(M,R)$ shared buses in between them. Each reducer reads from M buses and computes its output. Each mapper computation takes an ID as input, telling it which R buses to write its outputs to. For example, suppose $M = 10$ and $R = 2$, and hence we have 10 shared buses. The first mapper writes its output for reducer 1 to bus 1 and for reducer 2 to bus 2 (and implicitly writes zeros to the other buses). The second mapper writes its output for reducer 1 to bus 2 and for reducer 2 to bus 3. This continues until the tenth mapper writes its output for reducer 1 to bus 10 and its output for reducer 2 to bus 1. The prover sends the digests for all of the computations and buses, along with the proofs binding them together, back to the verifier, who ensures (via the digests fed into each Verify call) that the data was routed correctly between mappers and reducers. If desired, all of the proofs and digests can be made zero knowledge, and since the dataflow between mappers and reducers is data independent, the computation as a whole is zero knowledge as well.

Automated QAP Partitioning As explained above, Geppetto's libraries enable programmer-directed QAP partitioning. We also experimented with automated partitioning of large monolithic QAPs, expressed as finding hyper-graph cuts. We had some success efficiently finding approximate cuts in graphs of up to 200,000 equations with the METIS tool [36]. However, the programmer-directed approach is more flexible and better exploits regular structure such as loops.

6.3 Symbolic Interpretation via LLVM

Next, we provide details on the construction of the Geppetto compiler. We elide QAP techniques described elsewhere [46].

General-Purpose LLVM Front-End As a front-end compiler, we use clang [40], a fast full-fledged C compiler with rich syntax, standard semantics, and optimizations. Hence, Geppetto compilation to quadratic equations starts from a low-level, typed, integer-centric representation of the program, obtained by running (for instance) `clang -O2 -S -DQAP -emit-llvm sample.c -o sample.s`, where `-DQAP` declares but does not define Geppetto primitive types and functions.

Compiling to QAPs benefits from clang's aggressive inlining and partial evaluation. We disable other, unhelpful clang optimizations, such as its replacement of multiplication by a constant $x * 8$ (free in QAPs) with a bit shift $x << 3$ (which incurs bit splitting). Using clang should also facilitate extension to other LLVM-supported languages, but this may require adding support for more of LLVM's instruction set.

Interpreting LLVM Bitcode Instead of emitting an arithmetic circuit, Geppetto first compiles, then evaluates programs (in *prove* mode) by symbolic interpretation of LLVM code. Keeping the circuit implicit facilitates the generation of proofs for large computations, inasmuch as the unfolded circuits can be much larger than the LLVM code that generates them.

Our interpreter relies on a shallow embedding into F#, relying on the F# control stack and heap; i.e., function calls are implemented by calls to an F# `call` function, and mallocs are F# array creations.

Some values are known at compile time, and used to specialize the QAP equations (Eqn. (4)) we produce. Others are known only at run time; these values are treated symbolically, using an abstract domain for integers; their operations generally involve adding QAP equations.

Interpretation is cheap relative to cryptography, so, in prove mode, we simply re-interpret the LLVM code to produce concrete witnesses for all run-time intermediate variables (the 'wires' of the implicit circuit), and we accumulate them into digests and proofs. Thus, Geppetto uses *two* related interpreters (described below) that differ in their interpretation of integers.

Symbolic Interpretation (1): Compilation Geppetto separately interprets each outsourced function. As a side-effect of their operations, variables and equations are added to the function's QAP. For instance, multiplying two unknown integers adds a variable (for the result) and an equation. Global caches identify and eliminate common subexpressions.

For this interpretation, we represent unknown integers as a triple of (i) a linear combination of QAP variables; (ii) a source semantics: either some LLVM `int`*n* integer (e.g., int, short, char) or a field element (for embedded cryptography); and (iii) a range: an interval in $\mathbb{Z}$ that covers any value this integer may have at run time. Keeping track of ranges enables us to optimize precomputations for fast exponentiations, to minimize binary decompositions (which cost one equation per potentially active bit), to detect field overflows, and to defer integer truncation (which require binary decompositions) for almost all operations. For instance, our compiler may represent the (unknown) value of an LLVM local variable as 'an int32, obtained by adding the 5th and 6th QAP variables, with range 1..100'.

At this stage of the compilation, the MultiQAP consists of one QAP per function, plus 'linking' information on the shared buses. This suffices to generate keys, as the compiler traverses each function's QAP in turn, while keeping the buses virtual.

Symbolic Interpretation (2): Evaluation in 'prove' mode We use another, faster instance of our interpreter, and we now interpret the whole program, not just its outsourced code. Depending on the program's control flow, one outsourced function may be interpreted many times with different 'run-time' values. The evaluator still distinguishes between 'compile-time' and 'run-time' values, although it has values for both, because it needs to accumulate QAP witnesses for any operation on 'run-time' values, in strict correspondence with the QAP variables and equations produced by the compiler. Hence, values for all QAP variables introduced at compile time are stored as inputs for the cryptographic digests.

Because some compiler optimizations depend on data not available at run time (such as integer ranges and caches), the evaluation of some operations depends on auxiliary 'interpretation hints' passed along by the compiler. For example, before XORing a variable with a constant, a hint tells the evaluator whether a new binary decomposition is required (as we try to re-use existing decompositions) and, when required, how many

bits it uses (as the evaluator must record a witness for every bit of the decomposition provisioned by the compiler).

The evaluator also intercepts calls to load or save banks, as well as `OUTSOURCE` calls, and computes digests and proofs respectively. When evaluation completes, the prover will have produced exactly the evidence expected by the verifier.

Cryptography (FFLib) All cryptographic operations are implemented in a separate high-performance C++ library, with efficient support for many base fields and elliptic curves. FFLib is optimized for native x64 execution, unlike our QAP-friendly crypto libraries (§6.4). In addition to default curves that achieve 128-bit security, it also supports toy curves for testing and debugging. Since as much as 75% of the total run time (for key, digest, and proof generation) is spent multiplying and exponentiating elliptic curve points, we optimize these operations using standard pre-computation and batching techniques [46].

Primitive Libraries Whenever possible, we reflect (and even implement) primitive features of the interpreter using C types and functions. Pragmatically, this keeps our code base small, and lets us rely on standard (non-cryptographic) tools for testing and debugging purposes—for instance by comparing `printfs` between native clang runs and interpreted runs of the same code.

We provide a basic IO library. When loading from a file, a flag indicates whether this is a 'compile-time' or a 'run-time' file. Values from compile-time files are baked into the compiled QAP. For run-time files, the compile-time interpreter allocates fresh local QAP variables, and the evaluation-time interpreter loads the file's contents as run-time values. Thus, the file represents private, untrusted inputs provided by the prover.

As another example, for many programs, QAP size intricately depends on compile-time values; the interpreter provides a primitive function `int nRoot()` that returns the degree of the QAP being generated (or proved), thereby letting C programmers debug the cryptographic performance of their code and even control the partitioning of their code between several QAPs of comparable degrees—for instance by unrolling a loop until four million QAP equations have been generated.

6.4 Cryptographic Libraries and Bootstrapping

Geppetto has specific support for the compilation of programs that evaluate cryptographic operations, to enable bootstrapping and other flexible applications of nested evaluation.

Field arithmetic and cryptography To support bootstrapping, we provide custom C libraries that implement primitive field operations including addition, multiplication, division, and binary decomposition. These enable fast, field-based embedding of cryptography, intuitively taking advantage of 254-bit words. The field type is also implemented natively. Thus, field operations can be compiled both with clang and for bootstrapping by Geppetto.

Accordingly, our IO library supports loading C structs that mix machine integers and field elements. As shown in the code of `verify_job`, we use it to load cryptographic evidence as 'run-time' data, and similarly for all other pieces of evidence. By choosing to load the verification keys at 'compile time' or 'run time', we select a different trade-off between performance and flexibility (see §7.3).

QAP-Friendly Elliptic Curves Cryptography We developed a plain, QAP-friendly C implementation of the elliptic-curve algorithms for §5, including optimal Ate pairings. We briefly discuss two specific optimizations.

As in prior work [9], we use affine coordinates (2 field elements) instead of projective ones (3 field elements). Native implementations use projective coordinates to avoid a field division when adding two points; since we verify the computation, however, a field division is just as fast as a field multiplication.

For fast multiplication, the native algorithm has four cases at each iteration of the loop, due to the special treatment of infinite points in addition. To prevent these conditional branches, which are costly when compiling to QAPs, we add an initial summand and remove it at the end.

Bounded Bootstrapping Our compiler implements multiple levels of bootstrapping, as described in §2.2. Continuing with our example in §6.1, assume we wish to compress the N proofs by writing a bootstrapped function that aggregates the values in the `result[N]` array. Geppetto's libraries will ensure that all N proofs (and corresponding digests) are verifiably verified, in addition to verifying the aggregation of the `result` array. To this end, we include another, similar but distinct copy of our Geppetto library that lets the C programmer define 'level 2' or 'outer' banks and outsourced functions. We can then program with two nested levels of verifiable computations, with the outer top-level calling 'level 2' outsourced functions, which in turn call inner 'level 1' outsourced functions according to their own schedules. Hence, we also support proof schedules, digest re-use, and MultiQAP programming at 'level 2'. As before, we obtain our verification specification by using a trivial implementation of banks as local buffers and ignoring `OUTSOURCE` annotations.

When compiling, we first run the Geppetto compiler with the trivial definition of 'level 2' banks and `OUTSOURCE`, and the primitive Geppetto definitions for 'level 1'. This generates keys and code for outsourcing all 'level 1' functions. We then run the Geppetto compiler with the primitive Geppetto definitions for 'level 2', and with the `-DVERIFY` flag for 'level 1', thereby including, e.g., the code of `verify_job` instead of `job`, as well as our supporting cryptographic libraries for all 'level 1' elliptic-curve verification steps.

When proving, we run the Geppetto prover first at level 1 (producing evidence for its outsourced calls) then at level 2 (loading that evidence from untrusted, 'run-time' files). When verifying, we simply compile the source program with the `-DVERIFY` flag for level 2.

The approach above applies for further bootstrapping levels.

6.5 Branching and Energy-Saving

When evaluating a program, there is no proof cost involved for QAP variables that evaluate to zero: formally, we add a polynomial contribution multiplied by 0 (§4.1), and we multiply digests by 1 (a key element exponentiated by 0). Thus, if at compile time we ensure that *all* intermediate variables for a branch evaluate to 0 when the branch is not taken, then at run time there is no need to evaluate that branch at all.

For example, consider the code fragment `if(b) t = f(x)`. At compile time, if `b` is known, we just interpret the test, and compile the call to `f` only if `b` is true. If `b` is unknown, we interpret this fragment as `t = f(b*x) + (1 - b)*t` and, crucially, we compile the call to `f` *conditionally on* the guard `b`, with the following invariant: if `b` is 0 and `f`'s inputs are all 0, then its result must be zero, and zero must be a correct assignment for all its intermediate variables. Additionally, any `store` in `f` is conditionally handled, using similar multiplications by `b`. Note that the addition of `(1 - b)*t` is generally required to ensure that, if the branch is not taken, then the value of `t` is unchanged.

More generally, we extend our 'compile-time' interpreter so that its main evaluation function takes an additional parameter: its *guard*, g, with range 0..1. The guard is initially 1, but it can also be unknown (typically one of the QAP variables). Except for branches, the guard is left unchanged by the interpreter. Whenever the interpreter accesses a register with a less restrictive guard, it multiplies it by g before using it. (We cache these multiplications.) When branching on an unknown boolean, say b, both branches are evaluated with guards $g * b$ and $g * (1 - b)$, respectively. When joining, we sum the results of the corresponding branches, as explained next.

The single-static-assignment discipline of LLVM and its explicit handling of joins help us implement this feature. In our example, the code actually passed from clang to Geppetto is

```
entry:
  %tobool = icmp eq i32 %b, 0
  br i1 %tobool, label %if.end, label %if.then
if.then:
  %result = ...
  br label %if.end
if.end:
  %t = phi i32 [ %result, %if.then ], [ %t, %entry ]
  ...
```

where the compile-time function `phi` selects which register to use for the resulting value of `t` after the join. At compile time, as we symbolically execute *all* branches, we simply interpret the `phi` function as a weighted sum instead of a selector.

At run time, our representation of `b` tells us whether it was known at compile time or not; we use that information to (implicitly) provide 0 values for any branch not actually taken.

7 Evaluation

Below, we evaluate the effect of Geppetto's optimizations on the performance of the prover. We run our experiments on an HP Z420 desktop, using a single core of a 3.6 GHz Intel Xeon E5-1620 with 16 GB of RAM.

	Barreto-Naehrig		Cocks-Pinch	
Op	Base	Twist	Level 1	Level 2
Fixed Base Exp.	21.2μs	87.2μs	161.3μs	1027.5μs
Multi Exp. (254 bit)	55.6μs	241.5μs	454.5μs	2008.2μs
Pairing	0.6ms		5.0ms	31.9ms
Field Addition	44.2ns		43.3ns	65.2ns
Field Multiplication	288.2ns		288.0ns	726.0ns

Figure 6: **Microbenchmarks for Cryptography.** *Breakdown of the main sources of performance overhead in Geppetto's larger protocol. Each value is the average of 100 trials. Standard deviations are all less than 4%.*

7.1 Microbenchmarks

To calibrate our results, we summarize the cost of our cryptographic primitives in Figure 6. We generally use a Barreto-Naehrig (BN) curve for generating digests and proofs, and we use the Cocks-Pinch (CP) curves to handle embedded cryptographic computations like bootstrapping. We show measurements from two CP curves to illustrate how the costs grow for each progressive level. The BN curve is asymmetric, meaning that one source group (base) is cheaper than the other (twist). Geppetto's protocol and compiler are designed to keep most of the work on the base group.

The CP curves are slower than the BN for two reasons. First, the CP curves are chosen to support bounded bootstrapping, so they use larger field elements than the BN curve (see §5). Second, the BN code has been extensively optimized, including hand-tuned assembly code, while the CP code is newly written C. Based on operation counts from Magma [14], the first CP curve should be within 2-4$\times$ of the BN curve, and indeed comparing the CP curve's performance with a similar C version of the BN curve confirms this.

7.2 MultiQAPs

We compare the use of MultiQAPs for shared state with the use of hashing in prior work such as Pantry [16]. At a micro level, Pantry's results suggest that hashing an element of state increases the degree of the QAP by $\sim$11.25/byte. In contrast, with MultiQAPs, a full field element only increases the degree by one, so with Geppetto, the degree only increases by $\sim$0.03/byte, a savings of 375$\times$. Even if we want to operate on 32-bit values, instead of full field elements, Geppetto only costs 0.25/byte, a savings of 45$\times$.

At a macro level, for MapReduce, Pantry and Geppetto share the same costs for proving that the core mapper and reducer computations were performed correctly; on top of that, to handle state transferred between mappers and reducers, Pantry proves the correctness of $2M \cdot R$ hashes (since both the mappers and the reducers must prove they hashed the state correctly), while Geppetto proves the correctness of $M \cdot R$ bus digests. As a result, Geppetto's keys end up being a bit smaller; Geppetto's

		QAP Degree		KeyGen		Prover		Verifier		Baseline
MR: Dot product	Geppetto	10K		5s		3s		10ms		0.5ms
($m = 10K$)	Pantry	1.1M	(103×)	643s	(114×)	3696s	(1169×)	58ms	(5.8×)	
MR: Dot product	Geppetto	161K		49s		53s		10ms		6.3ms
($m = 160K$)	Pantry	16.8M	(104×)	14130s†	(283×)	22187s†	(412×)	58ms†	(5.8×)	
MR: Nucleotide substring	Geppetto	1K		0.7s		1s		36ms		0.2ms
($m = 6K, d = 10$)	Pantry	98K	(84×)	39s	(55×)	126s	(72×)	58ms	(1.6×)	
MR: Nucleotide substring	Geppetto	26K		21s		43s		35ms		0.3ms
($m = 60K, d = 32$)	Pantry	327K	(13×)	155s	(7×)	909s	(21×)	58ms	(1.7×)	
Loop: Matrix exponentiation	Geppetto	8K		5s		51s		411ms		0.9ms
($n = 10, e = 40$)	Pantry	32K	(4.0×)	15s	(3.1×)	405s	(8×)	211ms	(0.5×)	
Loop: Matrix exponentiation	Geppetto	37K		15s		253s		421ms		1.1ms
($n = 20, e = 40$)	Pantry	131K	(3.5×)	54s	(3.5×)	1463s	(6×)	211ms	(0.5×)	

Figure 7: **Apps with Shared State.** *For MapReduce (MR) apps, we give per-mapper statistics. For Loop, we consider the entire computation. These apps do not use bootstrapping. Parenthetical values show Pantry's relative overhead. Entries with † indicate simulated Pantry values.*

keys save further relative to Pantry, as Pantry needs key material for R hashes for each mapper and M hashes for each reducer, while Geppetto only needs $\texttt{max}(M,R)$ shared buses (§6.2).

A naïve alternative to MultiQAPs and hashing is to build one gigantic Pinocchio QAP, so that the shared state becomes simply internal circuit wires. However, our experiments quickly showed the futility of this approach; even for the relatively modest applications shown in Figure 7 and assuming only 10 mappers, this approach would require a QAP with a degree of 10M+, while the Pinocchio prover keels over (i.e., begins swapping) before it can reach 3M on a 16 GB machine.

7.2.1 Applications

To measure the end-to-end effect of MultiQAPs, we evaluate Geppetto on the following applications. We compare Geppetto's results against Pantry's implementation running on the same hardware, except when Pantry runs out of memory, in which case we use Pantry's validated cost model [16]. We borrow the first two examples from Pantry [16] to give a direct comparison with their work. We adopt Pantry's ratio of 10 mappers to 1 reducer, and we use their extension of Pinocchio to ensure an apples-to-apples comparison.

MapReduce: Dot Product [16] The verifier specifies (in Pantry via hash, in Geppetto via a digest) two vectors of integers; each mapper receives m integers and computes a partial dot product, and the reducer sums the mapper outputs.

MapReduce: Nucleotide Substring Search [16] The verifier specifies a DNA string that is divided amongst the mappers, each receiving m nucleotides. The mapper then searches for dynamically supplied length-d substrings reporting a match (if any) to the reducer which combines the matches.

Loop: Matrix Exponentiation The verifier supplies a dynamically chosen $n \times n$ matrix M and an exponent e, and the prover returns M^e. Matrix exponentiation is useful for many applications, e.g., to compute the width of a graph represented as an adjacency matrix [52].

This example shows the benefits of intertwined MultiQAPs. With Pinocchio, the QAP would scale with e, limiting the size of the problem, whereas, with MultiQAPs, we only need to compile the loop body (after some loop unfolding), which can then be used for arbitrary values of e. With Pantry, the loop body needs to hash a matrix on the way in and again on the way out, whereas MultiQAPs incur a handful of crypto operations per intermediate state generated.

7.2.2 MultiQAP Results

Figure 7 summarizes the impact of using MultiQAPs for shared state. The results only show CPU costs and do not include network latency or bandwidth, though the latter is unlikely to be a problem for either Pantry or Geppetto, given that proofs and digests are only a few hundred bytes each.

For MapReduce, we see the largest discrepancy between Geppetto and Pantry on the dot-product app. For this app, the QAP for the computation itself is quite simple, so for Pantry, the cost of hashing dwarfs the cost of the computation. For the nucleotide app, the shared state is still a dominant portion of the calculation for Pantry (though not as dominant as in dot product), and hence Geppetto maintains a wide margin.

For the Loop application, the QAP for the computation itself is non-trivial and grows faster than the IO between loop iterations; thus, the cost of state sharing relative to the computation is lower than for dot product, and the ratio drops further for larger matrices. Since Geppetto and Pantry generate essentially the same QAP for the computation itself, Geppetto's relative advantage drops accordingly.

7.3 Verifying Cryptography and Bootstrapping

In §5, we claimed that embedding cryptographic operations without matching field sizes was exorbitantly expensive. To validate this claim, we combined data from a basic ECC pairing operation coded in Magma with cost models from Pinocchio

for various operations such as bit splitting. Our calculations estimate that the pairing alone would require a QAP with degree of 44 million.

Fortunately, our choice of matching curves in §5 brings this cost down significantly. For example, a pairing only requires a QAP of degree 14K, an improvement of 3100× vs. the naïve approach, while an exponentiation, i.e., g^x, increases the degree by ∼60 per bit in x.

Furthermore, as discussed in §5, for a comparable security level, our initial curves for bounded bootstrapping provide approximately 34-77× better performance than curves supporting unbounded bootstrapping [9]. As §7.1 shows, however, performance degrades with each level added, and hence will eventually reach a point where they fall short of the unbounded curves' performance.

7.3.1 Bootstrapping

From the verifier's perspective, one level of bootstrapping is attractive, since she only receives (and only verifies) one constant-sized, 512-bit proof, and one constant-sized, 448-byte digest.

Without bootstrapping, the only way for the prover to generate such concise proofs would be via one massive Pinocchio-style QAP, which our results above (§7.2) show is infeasible. Nonetheless, bootstrapping does come at a cost. While bootstrapping, the "outer" QAP's degree grows with each digest or proof that it must verify. We summarize these costs below assuming that the verification keys are known at compile-time.

- For each recomputed digest, we increase the degree by 2K for each 32-bit integer value committed.
- For each full digest verification, we pay 79.6K (including the pairings needed for the checks from Equations (6)-(9)).
- For each bus digest verification, we pay 33.8K (since, as noted in §4.2, buses require fewer checks).
- For each proof verification (Eqn (10)), we pay 28.2K.

With keys unknown at compile-time, we pay instead 89.8K and 30.6K for full digest and proof verification, respectively.

We also observe that the prover's cryptographic cost for "outer" proofs and digests is typically higher than for work on the "inner" instance, even for QAPs of the same size. One reason is that the outer CP curve is less efficient than the inner BN curve (§7.1). A second reason is that many of the values the prover commits to for the inner instance arise from the program being verified, and hence they are often 1, 32, or 64 bits. In contrast, the outer curve verifies elliptic curve operations and hence many values are full-fledged 254-bit values.

While these costs are substantial, they are low enough that we can employ bootstrapping to scale the prover to much larger computations. For example, with our existing implementation, we could bootstrap up to 14 "inner" proofs sharing 16 buses; applying this to, say, the matrix exponentiation example allows us to produce a single, constant-size proof for a computation with a useful (i.e., not counting bootstrapping costs) QAP degree of over 50 M. When evaluating the computation, the prover executes 24M LLVM instructions and generates a proof in 152 minutes. While slow, this is five orders of magnitude faster than the unbounded bootstrapping in previous work (BCTV) [9], which, with a reported clock rate of 26 milliHz (and a lower 80-bit security level), would take approximately 29 years.

No source code was available for BCTV, so analyzing the causes of this large performance gap requires some guesswork. First, we estimate that one order of magnitude comes from the different choices of curves.

Second, BCTV use a circuit that checks a general-purpose CPU transition function for each program instruction. Thus, for straight-line code like matrix multiplication, they use hundreds of equations for each operation, whereas Geppetto generally uses one. BCTV's interpreter, however, supports RAM access and data-dependent control flow, while Geppetto's compilation-based implementation currently does not, and thus, one might expect a smaller performance gap on applications making use of those features. However, recent work [54] indicates that the compilation-based approach can incorporate these features and still outperform interpretation by 2-4 orders of magnitude on straight line code, and 1-3 orders of magnitude on RAM and data-dependent benchmarks.

Finally, BCTV apply bootstrapping at a very fine granularity. At every step of their CPU, they produce a proof with one curve, and then they use their second curve to verify that proof and translate it back into a proof on the first curve. Thus, each CPU instruction requires two bootstrapped proof verifications, whereas in this application, each Geppetto proof verification covers 1.7M LLVM instructions.

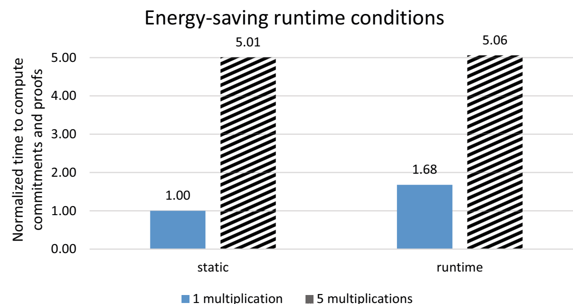

Figure 8: **Energy-Saving Circuits.** *The energy-saving multiplexer allows us to include an optional circuit that has low cost when unused.*

7.4 Energy-Saving Circuits

As a targeted microbenchmark to evaluate the benefits of energy-saving circuits (§2.3), in Figure 8, we compare a static compile-time condition to a runtime condition. The left group shows a static computation with a single matrix multiplication and a static computation containing five multiplications that takes proportionally longer. On the right, a single computation supports up to five multiplications, but is organized using energy-saving circuits to make the one-multiplication case inexpensive. Using this circuit to compute one matrix multiply costs 68% more than the static version (rather than 5×), and costs a negligible 1% in the five-multiply case.

7.5 Compiler

Some previous verifiable computations systems do not include a compiler [22, 52], while those that do [10, 16, 46] have typically compiled small examples with less than 100 lines of C code. In contrast, Geppetto's compiler handles non-trivial cryptographic libraries, with the largest clocking in at 4,159 SLOC [55] of complex cryptographic code supporting elliptic curve operations, including pairing.

8 Related Work

Verifiable Computation As discussed in §1, many previous systems for verifying outsourced computation make undesirable assumptions about the computation or the prover(s). Recently however, several lines of work have refined and implemented protocols for verifiable computation that make at most cryptographic assumptions [9, 46, 49, 52]. These systems offer different tradeoffs between generality, efficiency, interactivity, and zero-knowledge, but they share a common goal of achieving strong guarantees with practical efficiency.

However, these systems typically verify a single program at a time, leading to performance issues for state shared across computations (see §2.1.1). We compare and contrast alternate techniques for handling state in §2.1.4.

As discussed in §5 and §7.3.1, Ben-Sasson et al. [9] instantiate and implement suitable elliptic curves for unbounded bootstrapping. Geppetto can leverage unbounded bootstrapping, but it also supports bounded bootstrapping for better performance. Ben-Sasson et al. bootstrap the verification of individual CPU instructions using handwritten circuits, whereas Geppetto uses compiled cryptographic libraries to bootstrap high-level operations (e.g., procedure calls) following our belief that C should be compiled, not interpreted. Compilation plus bounded bootstrapping can provide up to five orders of magnitude faster performance, though both techniques sacrifice generality compared with unbounded interpretation.

Interpreting CPU instructions means that Ben-Sasson et al. natively avoid the redundancy of executing both branches of an if-else branch in the source program, but the interpretation circuit itself is repeated for every instruction and contains circuit elements that are not active for every instruction, and hence could benefit from Geppetto's energy-saving circuit's ability to power down unused portions of the CPU verifier. Similarly, programs interpreted in this framework can benefit from Geppetto's MultiQAP-based approach to state.

Commit-and-Prove To our knowledge, commit-and-prove (CP) schemes are first mentioned by Kilian [37]. Canetti et al. [18] define CP schemes in the UC model and realize such schemes in the $\mathcal{F}_{ZK}$-hybrid model. Escala and Groth [26] design CP schemes from Groth-Sahai proofs [34].

Zero Knowledge Several systems compile high-level functions to zero-knowledge (ZK) proofs [1, 4, 42]. Compilers from Almeida et al. [1] and Meiklejohn et al. [42] build on Σ-protocols [24], while the work of Backes et al. [4] uses Groth-Sahai ZK proofs [34]. For the subset of functionality these systems support, they are likely to outperform Geppetto at least for the prover, but none offer the degree of efficient generality and concise proofs that Geppetto provides.

9 Conclusions

Geppetto employs four independent but carefully intertwined techniques: MultiQAPs, QAP-friendly cryptography, bounded bootstrapping, and energy-saving circuits. We increase the efficiency of the prover by orders of magnitude, and we improve the versatility of its proofs, e.g., by enabling the efficient verification of hidden computations. Geppetto's scalable compiler exposes this power and flexibility to developers, bringing verifiable computation one step closer to practicality.

10 Acknowledgements

The authors gratefully thank Joppe Bos, Olga Ohrimenko, Srinath Setty, Michael Walfish, and the anonymous reviewers.

References

[1] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In *Proc. of ESORICS*, 2010.

[2] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.

[3] M. Backes, D. Fiore, and R. M. Reischuk. Nearly practical and privacy-preserving proofs on authenticated data. In *Proc. of IEEE Symposium on Security and Privacy*, 2015.

[4] M. Backes, M. Maffe, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. In *Proc. of ISOC NDSS*, 2012.

[5] F. Baldimtsi and A. Lysyanskaya. Anonymous credentials light. In *Proceedings of ACM CCS*, 2013.

[6] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography (SAC)*, 2006.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

[8] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Innovations in Theoretical Computer Science (ITCS)*, Jan. 2013.

[9] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proc. of CRYPTO*, 2014.

[10] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proc. of USENIX Security*, 2014.

[11] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, 2013.

[12] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[13] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *Proceedings of IACR CRYPTO*, 2001.

[14] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4), 1997.

[15] A. Bostan and E. Schost. Polynomial evaluation and interpolation on special sets of points. *Journal of Complexity*, 21(4), 2005.

[16] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proc. of the ACM SOSP*, 2013.

[17] J. Camenisch and A. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In *EUROCRYPT*, 2001.

[18] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of ACM STOC*, 2002.

[19] B. Carbunar and R. Sion. Uncheatable reputation for distributed computation markets. In *FC*, 2006.

[20] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Comp. Sys.*, 20(4), 2002.

[21] C. Cocks and R. Pinch. Identity-based cryptosystems based on the Weil pairing. *Manuscript*, 2001.

[22] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science*, 2012.

[23] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. Cryptology ePrint Archive, Report 2014/976, Nov. 2014.

[24] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Proc. of CRYPTO*, 1994.

[25] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio coin: Building Zerocoin from a succinct pairing-based proof system. In *ACM PETShop*, 2013.

[26] A. Escala and J. Groth. Fine-tuning Groth-Sahai proofs. Cryptology ePrint Archive, Report 2004/155, Oct. 2013.

[27] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Trans. on Comp. Sys.*, 20(1), Feb. 2002.

[28] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of IACR CRYPTO*, 2010.

[29] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proc. of EUROCRYPT*, 2013.

[30] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *Proc. of the Symposium on Theory of Computing (STOC)*, 2008.

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

[32] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proc. of CT-RSA*, 2001.

[33] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *IACR ASIACRYPT*, 2010.

[34] J. Groth and A. Sahai. Efficient non-interactive proof systems for bilinear groups. In *Proc. of EUROCRYPT*, 2008.

[35] K. Karabina and E. Teske. On prime-order elliptic curves with embedding degrees $k = 3$, 4, and 6. In *Proc. of the Conference on Algorithmic Number Theory (ANTS)*, 2008.

[36] G. Karypis. METIS 5.1.0: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical report, Department of Computer Science, University of Minnesota, Mar. 2013.

[37] J. Kilian. *Uses of Randomness in Algorithms and Protocols*. PhD thesis, MIT, Apr. 1989.

[38] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.

[39] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TrueSet: Nearly practical verifiable set computations. In *Proc. of USENIX Security*, 2014.

[40] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Symposium on Code Generation and Optimization*, Mar 2004.

[41] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA*, 2005.

[42] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *Proc. of USENIX*, 2010.

[43] R. C. Merkle. A certified digital signature. In *Proc. of CRYPTO*, 1989.

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

[45] A. Miyaji, M. Nakabayashi, and S. Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Trans. on Fundamentals*, 84(5), 2001.

[46] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2013.

[47] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[48] A. Rial and G. Danezis. Privacy-preserving smart metering. In *Proc. of the ACM WPES*, 2011.

[49] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, 2013.

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

[51] R. Sion. Query execution assurance for outsourced databases. In *Proc. of VLDB*, 2005.

[52] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proc. of IACR CRYPTO*, 2013.

[53] P. Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC*, 2008.

[54] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the ISOC NDSS*, Feb. 2015.

[55] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.