Geppetto: Versatile Verifiable Computation

Craig Costello
craigco@microsoft.com
Microsoft Research

Cédric Fournet
fournet@microsoft.com
Microsoft Research

Jon Howell
howell@microsoft.com
Microsoft Research

Markulf Kohlweiss
markulf@microsoft.com
Microsoft Research

Benjamin Kreuter
brk7bx@virginia.edu
University of Virginia*

Michael Naehrig
mnaerig@microsoft.co
Microsoft Research

Bryan Parno
parno@microsoft.com
Microsoft Research

Samee Zahur
samee@virginia.edu
University of Virginia*

November 2014

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 results, but the overhead to *produce* proofs largely remains impractical.

Geppetto introduces complementary techniques for reducing prover overhead and increasing prover flexibility. With Multi-QAPs, 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 instantiation of cryptographic primitives, Geppetto also brings down the cost of verifying outsourced cryptographic computations (e.g., verifiably computing on signed data); together with Geppetto's notion of bounded proof bootstrapping, Geppetto improves on prior bootstrapped systems by five orders of magnitude, albeit at some cost in universality. Geppetto also supports qualitatively new properties like verifying 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 that consumes LLVM code generated from a variety of apps, as well as a large cryptographic library.

1 Introduction

The recent growth of mobile and cloud computing makes outsourcing computations from one party to another 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 [36, 57], relied on physical-security assumptions [45, 52, 54], assumed uncorrelated failures [21, 22, 40], or achieved good asymptotics [3, 32, 32, 34, 35, 37, 42, 48] but impractical concrete performance [51, 56].

Recently, several lines of work [10, 51, 55, 58] on verifiable computation [32] 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. Currently, the best performing, fully general-purpose verifiable computation protocols [51, 55] are based on Quadratic Arithmetic Programs (QAPs) [33]. To provide non-interactive, publicly verifiable computation, as well as zero-knowledge proofs (i.e., computations in which some or all of the worker's inputs are private) recent systems [4, 8, 10, 11, 18, 27, 43, 61] have converged on the Pinocchio protocol [51] as a cryptographic back end. Pinocchio, in turn, depends on QAPs.

While these protocols have made proof verification nearly practical, the cost to *generate* a proof remains a significant barrier to practicality. 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 introduce a series of interlocked techniques that support more flexible, and hence more efficient, provers. These techniques include MultiQAPs for sharing state between or within computations, efficient embeddings for verifying cryptographic computations, bounded bootstrapping for succinct proof aggregation, and energy-saving circuits to ensure that 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 many related proofs. For example, the

---

*Microsoft Research Intern

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

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 circuits that can verifiably and efficiently share state. Today, compiling code from C to a QAP typically requires unrolling all of the loops and inlining all of the 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 a hundred times, we can create a single circuit for the loop body and then use MultiQAPs to efficiently connect the state at the end of each iteration of the loop 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 [9, 14, 30, 33, 47], but implementations show that this approach increases the degree of the QAP by tens or hundreds per state element [10, 18, 61], whereas with MultiQAPs, the degree increases only by 1.

Second, we show how a careful choice of cryptographic primitives significantly improves the efficiency of generating proofs of *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 [53]. As another example, recent work [8, 27] shows how to anonymize Bitcoin transactions using Pinocchio [51] and would benefit from the ability to verify signatures within Bitcoin transactions. In existing QAP systems, computations take place over a small (e.g., 254-bit) field, so computing cryptographic operations requires an awkward and inefficient embedding of the cryptographic machinery via either a BigInteger library built out of field elements or via large extension fields [27]. With our techniques, all of these examples can be naturally and efficiently embedded into a proof of an outsourced computation.

With MultiQAPs, the prover generates multiple proofs about related data. This improves flexibility and performance for the prover, but it degrades an attractive feature of Pinocchio, in which the proof consisted of a (tiny) constant-sized proof, and the verifier's work scaled only with the IO.

As a third contribution, we combine our MultiQAPs and cryptographic embeddings to obtain MultiQAPs with constant-sized proofs via *bounded proof bootstrapping*. In theory, with proof bootstrapping [12, 59], the prover can combine any series of proofs into one constant-sized proof by verifiably computing the verification of all of those proofs. Very recent work elegantly achieves unbounded proof bootstrapping [10], but this generality comes at a cost (§5). Our 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 bootstrap, we can achieve more practical performance. Moreover, by considering (bounded or unbounded) bootstrapping in the context of our cryptographic embedding techniques, we show how to efficiently outsource computations where 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.

Lastly, just as MultiQAPs eliminate the prover 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. In the standard approach to compiling code to QAPs, the prover devotes considerable effort to proving the correctness of unused portions of the circuit. For example, in an if-else statement, the prover must perform computations for both the if and the else block. With energy-saving circuits, the prover only exerts cryptographic effort for the actual path taken (e.g., only the if block when the condition is true). While energy-saving circuits are generally useful, they are particularly beneficial when using bounded proof bootstrapping to condense many proofs from a MultiQAP. Such proof compaction requires the key generator to commit, in advance, to the 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 and plan to make the code available. Geppetto includes a compiler in F#, a cryptographic runtime in C++, and QAP-friendly libraries in C. 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. The compiler provides 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. It also provides higher-level C libraries for common programming patterns, such as MapReduce or loops.

2 Geppetto Overview

In this section, we give a formal but abstract description of Geppetto's main constructions, deferring cryptographic definitions to §3 and our protocol to §4.

We first explain how we share state between computations via MultiQAPs, beginning with an intuitive overview (§2.1). We then formalize related computations using a schedule of related proofs (§2.2) where the intermediate state can be represented compactly via a *commit-and-prove* (CP) scheme (§2.3). Next, we design an efficient MultiQAP-based CP system (§2.4). Finally, we turn to proofs for cryptographic operations and bootstrapping (§2.5), and to energy-saving circuits (§2.6).

2.1 MultiQAP Intuition

Prior verifiable computation systems like Pinocchio [51] provide proofs of properties, abstractly written $P(\mathbf{u})$, where $\mathbf{u}$ is
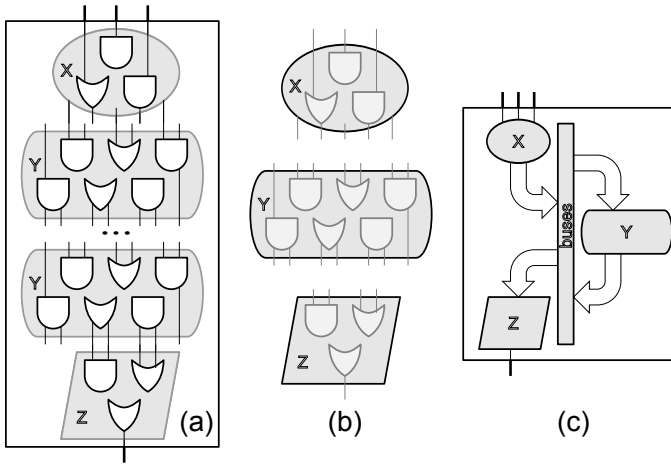
Figure 1: **MultiQAPs** *(a)* Most existing verifiable computation systems compile programs to a single large circuit representation, leading to internal redundancy. (b) By extracting common substructures, we can represent the program as a smaller collection of circuits, but the verifier must now laboriously check all of the internal IO between these circuits. (c) MultiQAPs connect circuits with a *bus* structure that supports a succinct and efficient commitment to the bus values.

Figure 2: Overview for § 2. Geppetto's compiler runs in two phases: (I) Key generation phase splits the code into properties (procedures) that are later expressed as Quadratic Arithmetic Programs (QAPs). A MultiQAP $Q^\star$ is then produced by overlapping the QAPs to create buses for sharing state. The compiler also produces a schedule σ specifying which bank, including locals, inputs and outputs, should be included at each step. Finally, key generation also produces keys for the prover and verifier as well as the verifier code. (II) The prover executes the program to create variable assignments for the MultiQAP $Q^\star$ and its buses according to the schedule. In each step, the prover generates a proof for the sub-QAP Q_i, where i is specified in the schedule, and commits to any outputs produced by Q_i. The verifier verifies the proofs as well as the commitments. *In addition, for bootstrapping compile the verifier's code and prove correct verification such that the top-level verifier is left to verify only the latter.

a tuple of parameters for P. Cryptographers refer to P as a language, and **u** an instance. Programmers may see $P(\mathbf{u})$ as a trace-property, e.g., interpreting **u** as a valid input-output sequence obtained by running a program whose specification is captured by P. As P grows to encompass larger and more complex functionality, the CPU and memory costs for the prover (as well as key size) increase linearly or worse. As §7.2 shows, this limits prior systems to modest application parameters.

To scale to larger problems, we can decompose the proof of P into a conjunction of proofs of m simpler properties $P_0,...,P_{m-1}$. Figure 1 illustrates this decomposition in the context of circuits. The decomposition may follow the structure of the source program; for instance, one P_i may prove the correct execution of a single function call, or a single loop iteration in the program. The whole proof then consists of n proof instances, since it may include multiple instances of each property P_i, with the various instances potentially sharing some of their parameters; e.g., the result of one loop iteration may be passed as the input for the next iteration. §2.2 defines the notations used in the rest of the paper for scheduling proofs and sharing parameters.

While proof decomposition aids prover scalability, used naively, it destroys the verifier's performance. In other words, if the prover simply uses Pinocchio to prove the correctness of each P_i instance, then he must send all of the computation's intermediate values back to the verifier, so that she could check, e.g., that the prover correctly transferred the output of one P_i to the input of the next. Handling so much intermediate state would make it difficult or impossible for the verifier to "win" from outsourcing.

To avoid placing this burden on the verifier, we will build a non-interactive commit-and-prove scheme [20, 28, 41], i.e., a scheme in which the prover can supply a short commitment to intermediate values and prove multiple statements about the
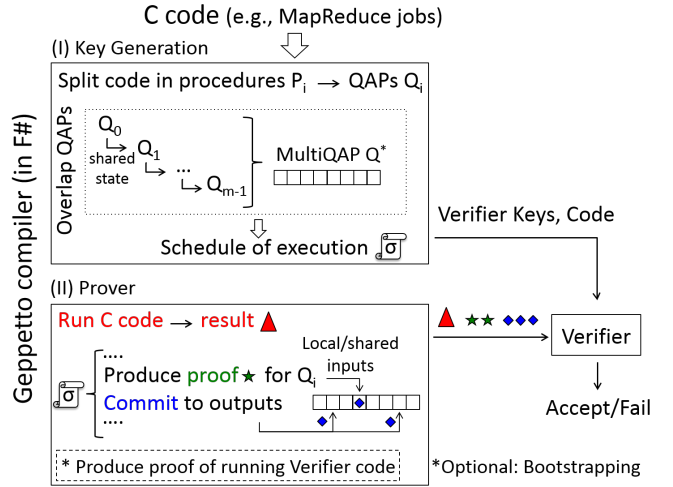
committed values. §2.3 gives a more formal definition. Our commitments optionally offer privacy, meaning that the committed values are hidden from the verifier.

Instead of building expensive commitments atop existing protocols as in prior work [18], in §2.4, we show how to build shared state *into* a generalization of the underlying QAP representation, which we call *MultiQAPs*, hence making commitments nearly free.

In more detail, we use Pinocchio's techniques to express each P_i as a Quadratic Arithmetic Program (QAP) Q_i, a format suitable for succinct cryptographic proofs. To share state between individual Q_i, a single MultiQAP $Q^\star$ combines the Q_i with one or more *buses*, each carrying some number of values. Each of its sub-QAPs can "tap into" a bus to write or read the bus value. In our cryptographic instantiation, writing to the bus corresponds to generating a commitment for the corresponding portion of the MultiQAP, while reading the bus corresponds to including the commitment in the proof for the sub-QAP. Connecting the QAPs via buses lets us use a *single* instance of our commit-and-prove scheme for $Q^\star$ for *all* proof schedules over $(Q_i)_{i\in[m]}$, with the capability to share compact, potentially private commitments between their proofs (rather than large collections of parameters in plaintext) without significantly increasing the prover's costs.

Figure 2 visualizes the flow of our compiler.

3

2.2 Scheduling Proofs With Shared State

Multi-Proof Systems To scale prover performance, we decompose instances of single, complex properties P into conjunctions of related instances of simpler properties $(P_i)_{i \in [m]}$ of the form

$$P(\mathbf{u_0}) \triangleq \exists \mathbf{u_1}. \bigwedge_{(i,\mathbf{t}) \in \sigma} P_i(\mathbf{u^t})$$

where σ is a 'schedule', that is, a set of pairs $(i, \mathbf{t})$ with i the index of the simpler property to use, and $\mathbf{t}$ a vector selecting variables from $\mathbf{u_0}, \mathbf{u_1}$. Crucially, the properties P_i may thus share intermediate worker variables.

For instance, we may decompose the Boolean function $P(u,r) \triangleq r = f(g(f(u)))$ into

$$P_0(u,r) \triangleq r = f(u) \qquad P_1(u,r) \triangleq u = g(r)$$

and the 3-proof schedule $\sigma = (0, (u_1, r_1)); (1, (u_2, r_1));$ $(0, (u_2, r_2))$, since we have $P(u_1, r_2) \Leftrightarrow \exists r_1 u_2. P_0(u_1, r_1) \land$ $P_1(u_2, r_1) \land P_0(u_2, r_2)$.

Banks In the following, we group the variables of a property into pairwise disjoint sets that we call *banks*. We let I indicate a bank, and let $\mathbf{I}$ range over tuples of banks. Hence, $P(\mathbf{I})$ specifies a property and a partition of its variables.

For instance, assuming P is compiled from the function `int f(int u[`ρ_u`])` we may use banks $\mathbf{I} \triangleq I_u, I_r$ where $I_r \triangleq \{r\}$ and $I_u \triangleq \{u_0, \ldots, u_{\rho_u - 1}\}$.

We consider compound proofs that share multiple instances of each bank. Formally, we define these instances by variable renaming: given a bank $I \triangleq \{u_0, \ldots u_{\rho_I - 1}\}$, we define a sequence of pairwise-disjoint instances $I^t \triangleq \{u_0^t, \ldots, u_{\rho_I - 1}^t\}$ for $t \geq 1$. (We reserve the use of $t = 0$ for the instance that assigns the constant 0 to every variable u_k of I.)

We define proof schedules as follows.

Definition 1 (Multi-proof schedule) *A* schedule σ *is a series of* steps $(i, \mathbf{t})$ *where* $i \in [m]$ *indicates a property* P_i *and* $\mathbf{t}$ *is a tuple of integers, one for each bank of* P_i. *We define* $n \triangleq |\sigma|$ *as the length of the schedule.*

A proof *for* σ *consists of (1) a proof* π *for each step, and (2) for each bank, the values for the variables for every non-zero index* t *that appears in* σ.

Intuitively, in each step $(i, \mathbf{t})$ of a schedule, i indicates which P_i the prover must prove (or the verifier must verify), and each $t \in \mathbf{t}$ indicates which values the verifier should use for each bank $I \in \mathbf{I}_i$ that parameterize P_i, with $t = 0$ indicating that all its variables are set to zero.

2.3 Commit-and-Prove (CP) Systems

Rather than transmit intermediate state to the verifier, a commit-and-prove scheme [20, 28, 41] allows the prover to generate short commitments to the state and then prove multiple statements about the committed values.

We give formal cryptographic definitions and proofs in §3, but abstractly, a CP scheme introduces a (keyed) $\mathsf{Commit}_I(u)$ algorithm for committing to the values u in bank I, and a Verify algorithm that takes a list of commitments to variable values, and a proof, and outputs a Boolean indicating whether the values represented by the commitments are consistent with the proof. As we prove in §3, we can use a CP scheme to verify an execution following a proof schedule as defined in Definition 1.

To make this more concrete, we return to our earlier example (§2.2) of proving $P(u,r) \triangleq r = f(g(f(u)))$ via decomposition into schedule σ. The schedule breaks P into three steps:

$$P(u_1, r_2) = \exists r_1, u_2 \quad : \quad r_1 = f(u_1) \land u_2 = g(r_1) \land r_2 = f(u_2)$$

To generate a proof for this schedule the prover could compute commitments to the intermediate values r_1 and u_2:

$$C_{r,1} \leftarrow \mathsf{Commit}_r(r_1) \quad C_{u,2} \leftarrow \mathsf{Commit}_u(u_2)$$

along with proofs π_i proving that f and g are computed correctly, and the verifier would check:

$$\begin{aligned}
V_\sigma(u_1, r_2, &C_{r,1}, C_{u,2}, \pi_0, \pi_1, \pi_2) = \\
&\text{let } C_{u,1} = \mathsf{Commit}_u(u_1) \\
&\text{let } C_{r,2} = \mathsf{Commit}_r(r_2) \\
&\mathsf{Verify}_0(C_{u,1}, C_{r,1}, \pi_0) \land \\
&\mathsf{Verify}_1(C_{u,2}, C_{r,1}, \pi_1) \land \\
&\mathsf{Verify}_0(C_{u,2}, C_{r,2}, \pi_2)
\end{aligned}$$

where Verify_i is a specialization of the verification function that checks proofs relating to property P_i.

Pinocchio as a CP Scheme We note that Pinocchio can be viewed as a very restricted commit-and-prove system for a single QAP Q, in the sense that Pinocchio proves: $P(\mathbf{u}) \triangleq \exists \omega. Q(\mathbf{u}, \omega)$ where $\mathbf{u}$ are function inputs and outputs,[2] and ω are intermediate local variables for the prover. Instead of sending ω to the verifier, the Pinocchio prover computes and sends a single, digest C_ω of their values, and a proof π that $\mathbf{u}, \omega$ is a solution for Q. As evidence of $P(\mathbf{u})$, the verifier receives (C_ω, π) whose size is independent of Q. If $V(\mathbf{u}, C_\omega, \pi)$ is true, then the verifier has the (computational) guarantee that $\exists \omega : Q(\mathbf{u}, \omega)$, and hence $P(\mathbf{u})$. As a bonus feature, both C_ω and π are perfectly hiding, meaning that they do not convey any information about ω.

2.4 An Efficient CP System from MultiQAPs

To build an efficient, general-purpose commit-and-prove scheme, we generalize Pinocchio's QAPs into MultiQAPs.

Quadratic Programs Abstractly, Pinocchio compiles properties of interest to equations of the form

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

where $\mathbf{u}$ is a vector of variables that range over some large, fixed prime field $\mathbb{F}_p$, and where the vectors $\mathbf{v}_r, \mathbf{w}_r, \mathbf{y}_r$ each define

---

[2]Pinocchio requires the first input, u_0, always be set to 1 so that the QAP can use affine forms instead of linear ones.

linear combinations over the variables $\mathbf{u}$. We say that Q has size $\rho \triangleq |\mathbf{u}|$ and degree d. Equivalently, Q may be seen as three $d \times \rho$ matrices in $\mathbb{F}_p$, i.e., $\mathbf{V} \triangleq (\mathbf{v}_r)_{r \in [d]}$, $\mathbf{W} \triangleq (\mathbf{w}_r)_{r \in [d]}$, and $\mathbf{Y} \triangleq (\mathbf{y}_r)_{r \in [d]}$.

Geppetto One of our main contributions is an extension of Pinocchio that lets us prove any property P scheduled from $(P_i)_{i \in [m]}$ using a matching sequence of proofs for their quadratic specifications $(Q_i)_{i \in [m]}$.

To this end, we provide a new, generic reduction from schedules using m quadratic programs of degree d to schedules using a single quadratic program $Q^\star$ of degree $d + |\mathbf{s}|$, where $\mathbf{s}$ includes all intermediate variables shared between those specifications. By choosing a decomposition from P to $(P_i)_{i \in [m]}$ that exploits the structure of P, most intermediate variables are local to one P_i, so we typically achieve $|\mathbf{s}| \ll d$.

Overlapping Quadratic Programs To share state between individual Q_i, the MultiQAP $Q^\star$ supports one or more *buses*, each carrying some number of field values. Each sub-QAP can "tap into" a bus to write or read the bus value. In our cryptographic instantiation, writing to the bus corresponds to generating a commitment for the corresponding portion of the Multi-QAP, while reading the bus corresponds to including the commitment in the proof for the sub-QAPs as detailed in our adaptation of the Pinocchio protocol in §4.2.

Let $\mathbf{I}$ be a series of pairwise-disjoint banks, including some distinguished shared banks we call *buses* $\mathbf{S} \subseteq \mathbf{I}$.

Let $Q_i(\mathbf{I}_i)$ for $i = 0..m-1$ a series of quadratic programs, each of degree at most d, such that we have $\mathbf{I}_i \subseteq \mathbf{I}$ for all i and $\mathbf{I}_i \cap \mathbf{I}_j \subseteq \mathbf{S}$ for all $i \neq j$. (The first condition guarantees that $\mathbf{I}$ collects all the banks; the second that $\mathbf{S}$ collects the buses.)

Let $\mathbf{I}'_i$ be the tuple of banks obtained from $\mathbf{I}_i$ by replacing each $I \in \mathbf{S} \cap \mathbf{I}_i$ with $I'_i \triangleq \{c_i \mid c \in I\}$ (using variables c_i that do not occur in $\mathbf{I}$). Let $\mathbf{I}'$ be their concatenation $\mathbf{I}'_0, \ldots, \mathbf{I}'_{m-1}$, and let $\mathbf{I}^\star \triangleq \mathbf{I}', \mathbf{S}$, still a concatenation of pairwise-disjoint banks. Intuitively, $\mathbf{I}^\star$ contains one additional local copy I'_i of each shared bus $I \in \mathbf{S}$ for each Q_i that uses it.

Recall that each program is of the form

$$Q_i(\mathbf{I}_i) = \bigwedge_{r \in [d]} (\mathbf{v}_{i,r} \cdot \mathbf{I}_i)(\mathbf{w}_{i,r} \cdot \mathbf{I}_i) = (\mathbf{y}_{i,r} \cdot \mathbf{I}_i).$$

where $\rho_i \triangleq \sum_{I \in \mathbf{I}_i} |I|$ is the size of Q_i. For each r, we let $\mathbf{v}'_r$ be the vector concatenation $\mathbf{v}_{0,r}, \ldots, \mathbf{v}_{m-1,r}$, and similarly for $\mathbf{w}'_r$ and $\mathbf{y}'_r$.

We define one quadratic program $Q^\star$ that 'overlaps' the Q_is,

$$Q^\star(\mathbf{I}^\star) \triangleq \bigwedge_{r = 0..d-1} (\mathbf{v}'_r \cdot \mathbf{I}')(\mathbf{w}'_r \cdot \mathbf{I}') = (\mathbf{y}'_r \cdot \mathbf{I}') \wedge \bigwedge_{I \in \mathbf{S}, c \in I} \sum_{i \mid I \in \mathbf{I}_i} c_i = c$$

of degree $d + \sum_{I \in \mathbf{S}} |I|$ and size $\sum_{I \in \mathbf{I}} |I| + \sum_{I \in \mathbf{S}, i \mid I \in \mathbf{I}_i} |I|$.

Representing $Q^\star$ as matrices $\mathbf{V}, \mathbf{W}, \mathbf{Y}$, we can assemble them from those of the Q_is, diagonal matrices $\mathbf{1}_i^?$ with a 1 for every variable of Q_i shared in $\mathbf{S}$ and 0s elsewhere, and the identity matrix $\mathbf{1}$ of size $\sum_{I \in \mathbf{S}} |I|$.

$$\mathbf{V} \triangleq \begin{pmatrix} \mathbf{V}_0 & \cdots & \mathbf{V}_{m-1} & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \end{pmatrix} \qquad \mathbf{Y} \triangleq \begin{pmatrix} \mathbf{Y}_0 & \cdots & \mathbf{Y}_{m-1} & \mathbf{0} \\ \mathbf{1}_0^? & \cdots & \mathbf{1}_{m-1}^? & -\mathbf{1} \end{pmatrix}$$

with $d + \sum_{I \in \mathbf{S}} |I|$ rows and $\sum_{i \in [m]} \rho_i + \sum_{I \in \mathbf{S}} |I|$ columns. (We omit $\mathbf{W}$, built as $\mathbf{V}$.)

Pragmatically, our compiler represents $Q^\star$ implicitly from the Q_is. The degree of $Q^\star$ is not (much) higher that those of the Q_is.

Next, we translate proof schedules σ over $(Q_i)_{i=0..m-1}$ to proof schedules $\sigma^\star$ over $Q^\star$. We replace each instance $(i, \mathbf{t}) \in \sigma$ with an instance $(0, \mathbf{t}^\star)$ for $Q^\star$ where $\mathbf{t}^\star$ is defined as follows: for banks $I \in \mathbf{I}_i$, we use the corresponding index from $\mathbf{t}$; for local copies of buses in $\mathbf{I}'_i$, we use a fresh index (that is, the bank instance is used only once in the whole schedule); for all other banks, we use the special index 0, indicating a constant, all-zero instance.

Definition 2 (Valid schedule) *A schedule is* valid *if all repeated non-zero indexes belong to buses, i.e. all other indexes are fresh.*

Note that σ^* is valid by construction, but we will give theorems for arbitrary *valid* schedules.

Our next lemma states that every property proved by σ over $(Q_i)_{i=0..m-1}$ can also be proved by $\sigma^\star$ over $Q^\star$.

Lemma 1 (Overlapping Quadratic Programs) *For every schedule σ over quadratic programs $(Q_i(\mathbf{I}_i))_{i=0..m-1}$, we have*

$$\bigwedge_{(i,\mathbf{t}) \in \sigma} Q_i(\mathbf{I}_i^{\mathbf{t}}) \Leftrightarrow \bigwedge_{(i,\mathbf{t}^\star) \in \sigma^\star} \exists \mathbf{I}^{\star \mathbf{t}^\star} \setminus \mathbf{I}_i^{\mathbf{t}} . Q^\star(\mathbf{I}^{\star \mathbf{t}^\star})$$

where, on the right-hand-side, $\mathbf{t}^\star$ is the translation of $\mathbf{t}$ and $\mathbf{I}^{\star \mathbf{t}^\star} \setminus \mathbf{I}_i^{\mathbf{t}}$ provides a local copy of each shared bus used in Q_i.

Proof. By induction on the length of the schedule. For each $(i, \mathbf{t}) \in \sigma$, we verify that $Q_i(\mathbf{I}_i^{\mathbf{t}}) \Leftrightarrow \exists \mathbf{I}^{\star \mathbf{t}^\star} \setminus \mathbf{I}_i^{\mathbf{t}} . Q^\star(\mathbf{I}^{\star \mathbf{t}^\star})$. Following the definition of $Q^\star$,

- Each of the shared-variable equations is just $c_i = c$ since $c_j = 0$ for any $j \neq i$.

- Each of the inner products $(\mathbf{v}'_r, \mathbf{0}) \cdot \mathbf{I}^{\star \mathbf{t}^\star}$ is just $\mathbf{v}_{i,r} \cdot \mathbf{I}_i^{\mathbf{t}}$ since every coefficient for $j \neq i$ is multiplied by 0 and $c_i = c$ for every shared variable in I'_i. (And similarly for $\mathbf{v}'_r$ and $\mathbf{y}'_r$.) ∎

2.5 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 show how a careful choice of cryptographic primitives and parameters allows us to build a large class of crypto operations (e.g., signing, verification, encryption) using elliptic curves built "natively" on $\mathbb{F}_p$. This makes it cheap to, for example, verify computations on signed data, since the data and the signature both "live" in $\mathbb{F}_p$.

Our most complex application of this technology is proof bootstrapping [12, 59], which we use to address the main drawback of CP schemes. With CP schemes, including our MultiQAP-based scheme, the size of the cryptographic
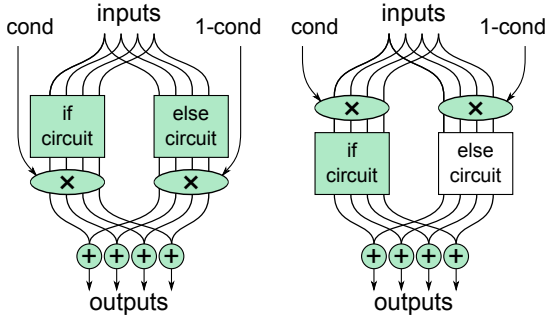
Figure 3: **Energy-Saving Circuits.** *Moving the multiplex step can nullify expensive crypto operations, since at runtime, in one of the two circuit blocks, every wire inside takes on the value zero.*

evidence—and the verifier costs—grow linearly with the number of commitments 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 $V_{\sigma^\star}(\mathbf{u}_0, \mathbf{C}, \Pi)$ be the property that a scheduled CP proof cryptographically verifies, where $\mathbf{C}$ and Π are the vectors of commitments and proofs used in the schedule $\sigma^\star$. We recursively apply Geppetto to generate a quadratic program $Q_{\sigma^\star}$ for $V_{\sigma^\star}$. This yields another, more efficient verifier $V_{\sigma^\star}^\circ(\mathbf{u}_0, C^\circ, \pi^\circ)$ with a single, constant-sized commitment C° to $\mathbf{C}, \Pi$, 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 $V_{\sigma^\star}^\circ$ need not be limited to just verifying the execution of $V_{\sigma^\star}$, as in traditional bootstrapped protocols [12, 59]. For example, suppose an authority the client trusts (e.g., the US FDA) cryptographically signs the verification keys for $V_{\sigma^\star}$, and we define $V_{\sigma^\star}^\circ$ to first verify the signature on the keys before using them to run $V_{\sigma^\star}$. If we use Geppetto's option to make commitments 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, we can verifiably outsource computations with proprietary algorithms.

Although the general idea of bootstrapping is well-known in principle [12, 59], its practicality relies on careful cryptographic choices to support an efficient embedding. Recent work suggested an embedding that supports bootstrapping an unbounded number of proofs but this generality comes at a significant cost (§5). In §5, we propose a more pragmatic embedding that supports only a bounded number of proofs but achieves significantly better performance.

2.6 Energy-Saving Circuits

A limitation of existing verifiable computation systems is that representing a computation as a quadratic program (informally, a circuit) 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, the prover needs to interpret and prove both branches then join their results. Concretely, the command if(b) {x = y} else {x=2*z} is ef-

fectively 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. With many branches, the resulting verifiable-computation tree may be much larger than any path in the tree, representing a verified program execution.

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. 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 prior system, the prover only does cryptographic work proportional to the path actually taken through the program. §6.4 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

Since we are interested in *succinct* proofs, we modify earlier definitions of commit-and-prove schemes [20, 28, 41] to only consider computationally bounded adversaries. As a succinct commitment implies that more than one plaintext maps to a given commitment value, an unbounded adversary can always "escape" the commitment's binding property.

In the following, we refer to representations of variable values as *digests*. Each digest C_j, may hide the values it represents via randomness o_j. Without hiding, we use a trivial opening $o_j = 0$ (and may omit it). We require that all digests of bus values be binding, and hence specifically refer to them as commitments. In contrast, digests used only in a single proof need not be binding.

As a side note, while Geppetto uses commit-and-prove schemes to prove properties, such schemes also enable interactive protocols where values are committed before being opened and/or used to prove statements about them. For instance, they easily integrate with existing Σ-protocols as employed by protocols such as anonymous credential systems [6, 19].

Definition 3 (Succinct Commit-and-Prove)
Consider a family of polynomial-time verifiable relations $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$ *on tuples* $\mathbf{u}$ *of a fixed length* ℓ.
A succinct commit-and-prove scheme $\mathcal{P} = (\mathsf{KeyGen} = (\mathsf{KeyGen}_1, \mathsf{KeyGen}_2), \mathsf{Commit}, \mathsf{Prove}, \mathsf{Verify})$ *for* $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$ *consists of five polynomial-time algorithms as follows:*

- *Key generation is split into two probabilistic algorithms:*

  $\tau \leftarrow \mathsf{KeyGen}_0(1^\lambda)$ *takes the security parameter λ as input and produces a trapdoor $\tau = (\tau_S, \tau_E)$ (independent of R and consisting of a simulation and extraction component).*

  $(EK, VK) \leftarrow \mathsf{KeyGen}_1(\tau, R)$ *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_j and VK_j for $j \in [\ell]$.*

- $C_j \leftarrow \mathsf{Commit}(EK_j, u_j, o_j)$*: Given an evaluation key for j, message u_j, and randomness o_j, the deterministic commitment algorithm produces a digest C_j to u_j.*

- $\pi \leftarrow \mathsf{Prove}(EK, \mathbf{u}, \mathbf{o})$*: Given an evaluation key, messages $\mathbf{u} \in R$, and openings $\mathbf{o}$, the deterministic prove algorithm returns a succinct proof π; i.e., $|\pi|$ is $\mathrm{poly}(\lambda)$.*

- $\{0,1\} \leftarrow \mathsf{Verify}(VK_j, C_j)$*: Given a verification key for j, the deterministic digest-verification algorithm either rejects (0) or accepts (1) the digest C_j.*

- $\{0,1\} \leftarrow \mathsf{Verify}(VK, \mathbf{C}, \pi)$*: Given a verification key and ℓ digests, the deterministic verification algorithm either rejects (0) or accepts (1) the proof π.*

Proof-verification guarantees apply only when all digests have been either honestly generated or verified.

Besides correctness, we define the security requirements of CP schemes.

Definition 4 (Correctness) *The commit-and-prove scheme $\mathcal{P}$ is perfectly correct for the ℓ-ary relation family $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$ if for all $R \in \mathcal{R}_\lambda$, all $\mathbf{u} \in R$ and all $\mathbf{o}$ from the randomness space:*

$$\Pr[\quad (EK, VK) \leftarrow \mathsf{KeyGen}(1^\lambda, R) :$$
$$\mathsf{Verify}(VK_j, \mathsf{Commit}(EK_j, u_j, o_j)) \qquad] = 1.$$

$$\Pr[\quad (EK, VK) \leftarrow \mathsf{KeyGen}(1^\lambda, R);$$
$$C_j \leftarrow \mathsf{Commit}(EK_j, u_j, o_j) \text{ for each } j \in [\ell] :$$
$$\mathsf{Verify}(VK, \mathbf{C}, \mathsf{Prove}(EK, \mathbf{u}, \mathbf{o})) \qquad] = 1.$$

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

$$\Pr[\quad \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); \tau = (\tau_S, \tau_E);$$
$$(EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);$$
$$(j, u, o, u', o') \leftarrow \mathcal{A}(EK, R, \tau_E) :$$
$$u \neq u' \wedge j \in S \wedge$$
$$\mathsf{Commit}(EK_j, u, o) = \mathsf{Commit}(EK_j, u', o') \quad] = \mathtt{negl}(\lambda).$$

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

$$\Pr[\quad \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); \tau = (\tau_S, \tau_E);$$
$$(EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);$$
$$(\mathbf{C}, \pi; \mathbf{u}, \mathbf{o}) \leftarrow (\mathcal{A}(EK, R) \parallel \mathcal{E}(EK, R, \tau_E)) :$$
$$(\exists j \in [\ell]. \, \mathsf{Verify}(VK_j, C_j) \wedge C_j \neq \mathsf{Commit}(EK_j, u_j, o_j)) \vee$$
$$(\mathsf{Verify}(VK, \mathbf{C}, \pi) \wedge \mathbf{u} \notin R) \quad] = \mathtt{negl}(\lambda).$$

For the case where we do not have commitments, i.e. $\ell = 2$, and $u = u_0$ is trivial and $\omega = u_1$ is non-binding, we also define ordinary soundness.

Definition 7 (Classical Soundness) *The proof scheme $\mathcal{P}$ is sound for the ℓ-ary relation family $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$, if for all efficient $\mathcal{A}$ and any $R \in \mathcal{R}_\lambda$,*

$$\Pr[\quad (EK, VK) \leftarrow \mathsf{KeyGen}(1^\lambda, R);$$
$$u, \pi \leftarrow (\mathcal{A}(EK, R) :$$
$$\mathsf{Verify}(VK, u, \pi) \wedge \neg(\exists \omega : (u, \omega) \in R) \quad] = \mathtt{negl}(\lambda).$$

A commit-and-proof scheme is zero-knowledge if it does not leak any information besides the truth of the statement.

Definition 8 (Perfect zero-knowledge) *A commit-and-prove scheme $\mathcal{P}$ is perfect zero-knowledge if there exists an efficient simulator $\mathcal{S}$ such that for all $\lambda \in \mathbb{N}, R \in \mathcal{R}_\lambda, \mathbf{u} \in R$, all $\{o_j\}_{j \in S}$ and random $\{o_j\}_{j \notin S}$ from the randomness space, and all adversaries $\mathcal{A}$, we have*

$$\Pr\Big[(\tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);$$
$$C_j \leftarrow \mathsf{Commit}(EK_j, u_j, o_j) \text{ for each } j \in [\ell];$$
$$\pi \leftarrow \mathsf{Prove}(\sigma, \mathbf{u}, \mathbf{o}, w) : \mathcal{A}(EK, \tau, \mathbf{C}, \pi) = 1\Big]$$

$$= \Pr\Big[(\tau_S, \tau_E) = \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);$$
$$C_j \leftarrow \mathsf{Commit}(EK_j, u_j, o_j) \text{ for each } j \in S :$$
$$\pi, \{C_j\}_{j \notin S} \leftarrow \mathcal{S}(\tau_S, \{C_j\}_{j \in S}) : \mathcal{A}(EK, \tau, \mathbf{C}, \pi) = 1\Big].$$

We use $j \notin S$ as a shorthand for $j \in [\ell] \setminus S$. To make the definition compositional, we let shared commitments have arbitrary openings $\{o_j\}_{j \in S}$. Digests on the other hand have random openings and can be replaced by digests produced by the simulator.

3.2 Composition by Scheduling

As discussed in §2, intuitively, we can prove a complex statement by proving simple statements about related language instances using shared commitments. We now formalize this intuition by extending knowledge soundness to multiple related proofs that share digests according to a valid schedule. This guarantees that the index of shared digests is in S.

Definition 9 (Scheduled Knowledge Soundness) *The commit-and-prove scheme $\mathcal{P}$ is scheduled knowledge sound for the ℓ-relation family $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$ 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 $R \in \mathcal{R}_\lambda$,*

$\Pr[\ \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda);$
$\quad (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R);$
$\quad (\sigma, \mathbf{C}, \Pi; \mathbf{u}, \mathbf{o}) \leftarrow (\mathcal{A}(EK, R) \parallel \mathcal{E}(EK, R, \tau)):$
$\quad \forall C_{jt} \in \mathbf{C}.\ (\mathsf{Verify}(VK_j, C_{jt}) \Rightarrow C_{jt} = \mathsf{Commit}(EK_j, u_{jt}, o_{jt})) \wedge$
$\quad \forall i \in 0..|\sigma| - 1.\ \sigma \text{ is valid} \wedge (\mathsf{Verify}(VK, \mathbf{C}^{\sigma(i)}, \pi_i)) \Rightarrow \mathbf{u}^{\sigma(i)} \in R$
$\quad] = 1 - \mathtt{negl}(\lambda).$

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

Proof of Theorem 1 (Scheduled soundness): Consider an adversary $\mathcal{A}$ against the scheduled knowledge soundness of the proof system. $\mathcal{A}$ takes EK as input. We define n adversaries $\mathcal{A}_1, \ldots, \mathcal{A}_n$ such that $\mathcal{A}_i$ takes EK as input and behaves like $\mathcal{A}$ except that it discards all sub-proofs except π_i. We construct our proof as a sequence of games.

Game 1 is the scheduled knowledge soundness game.

Game 2 is the same as Game 1, except that we run all of the extractors $E_1, \ldots, E_n$ for $\mathcal{A}_1, \ldots, \mathcal{A}_n$, whose existence is guaranteed by knowledge soundness, in parallel with $\mathcal{A}$ and on the same input and random tape. Game 2 aborts without $\mathcal{A}$ winning if for some π_i, $\mathsf{Verify}(VK, \mathbf{C}^{\sigma(i)}, \pi_i)$ and all $\mathsf{Verify}(VK_j, C_{jt})$ accept, but the E_i output witnesses $\mathbf{u}$ and randomness $\mathbf{o}$ such that either $\mathbf{u}^{\sigma(i)} \notin R$ or $C_{jt} \neq \mathsf{Commit}(EK_j, u_{jt}, o_{jt})$ for some $C_{jt} \in \mathbf{C}$.

Lemma The difference in the success probabilities of $\mathcal{A}$ between Game 1 and Game 2 is negligible, based on the knowledge soundness of $\mathcal{P}$.

Game 3 is the same as Game 2, except that it aborts without $\mathcal{A}$ winning if for some i, i', and $j \in S$, we have $u_{jt} \neq u'_{jt}$. (Recall that all digests that appear twice in σ have to be commitments and thus in S according to scheduled knowledge soundness.)

Lemma The difference in the success probabilities of $\mathcal{A}$ between Game 2 and Game 3 is negligible by the binding property of the commitment scheme.

The reduction runs the extractors and returns the collision to break the binding property. It relies on the adversary, and thus the reduction, receiving τ_E as input in the binding game.

In Game 3, $\mathcal{A}$'s probability of success is 0, since for every proof $\mathcal{A}$ returns, we can recover a witness such that $\mathbf{u}^{\sigma(i)} \in R$. ∎

Instead of applying different schedules to a given CP scheme $\mathcal{P}$, we may fix a schedule σ to produce another CP scheme $\mathcal{P}_\sigma$ with fixed, compound proofs. Let σ be a schedule of length n. Each proof schedule defines a relation $\mathbf{u} \in R_\sigma$ defined by the conjunction of $\mathbf{u}^{\sigma(i)} \in R$, $i \in [n]$. We construct the knowledge-sound commit-and-proof scheme for relation R_σ based on a scheduled knowledge-sound commit-and-prove scheme for R, as follows, with digest indexes now ranging over j, t instead of j, and with essentially the same keys.

- $\mathsf{KeyGen}_\sigma(1^\lambda, R_\sigma)$: the schedule in R_σ must correspond to the fixed schedule σ; then simply call $\mathsf{KeyGen}(1^\lambda, R)$ and return EK_j and VK_j for all EK_{jt} and VK_{jt}.

- $\mathsf{Commit}_\sigma(EK_{jt}, u_{jt}, o_{jt})$: return $C_{jt} \leftarrow \mathsf{Commit}(EK_j, u_{jt}, o_{jt})$, and similarly for $\mathsf{Verify}_\sigma(VK_{jt}, C_{jt})$.

- $\mathsf{Prove}_\sigma(EK, \mathbf{u}, \mathbf{o})$: For each $\mathbf{t} \in \sigma$, compute $\pi \leftarrow \mathsf{Prove}(EK, \mathbf{u}^{\mathbf{t}}, \mathbf{o}^{\mathbf{t}})$. Let $(\pi_i)_{i \in [n]}$ collect the resulting proofs. Return $(\pi_i)_{i \in [n]}$.

- $\mathsf{Verify}_\sigma(VK, \mathbf{C}, (\pi_i)_{i \in [n]})$ where $n = |\sigma|$. For each $i \in [n]$, assert $\mathsf{Verify}(VK, \mathbf{C}^{\sigma(i)}, \pi_i)$. Return 1 if all assertions succeed, 0 otherwise.

Theorem 2 (Scheduled CPs) *If $\mathcal{P}$ is a scheduled knowledge-sound commit-and-prove scheme for $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$, then $\mathcal{P}_\sigma$ is a knowledge-sound proof system for $\{\{R_\sigma\}_{R \in \mathcal{R}_\lambda}\}_{\lambda \in \mathbb{N}}$.*

The reduction from breaking soundness of $\mathcal{P}_\sigma$ to breaking scheduled soundness of $\mathcal{P}$ simply appends the fixed schedule to the output of the adversary.

Moreover, if $\mathcal{P}$ is binding for some commitments, then $\mathcal{P}_\sigma$ is also binding for them, hence scheduled knowledge-sound. Finally, if $\mathcal{P}$ is perfectly zero-knowledge then $\mathcal{P}_\sigma$ is perfectly zero-knowledge.

3.3 Combining Digests and Bootstrapping

As discussed in §2.5, the goal of proof bootstrapping [12, 59] is to compute a succinct proof whose length does not depend on the schedule length.

As a scheduled CPs may have many more commitments than proofs it is important for bootstrapping to also reduce the size of commitments. We achieve this by partitioning commitments into a smaller number of virtual banks. This is again a commit-and-proof scheme in which commitments are represented by tuples of commitments. In the bootstrapped CP each of these partitions will be represented by a single compact commitment.

For instance we may consider two virtual banks and write u_0 for the subset of messages tagged as public, and u_1 for the others, minimizing the number of commitments in the final proof. (Intuitively, u_0 includes messages passed in the clear and used to recompute commitments.)

We give the details of such a CP adapting the usual proof-of-a-proof bootstrapping construction to CPs $\mathcal{P}$ and $\mathcal{P}'$:

Protocol 1 (CP bootstrapping)

- KeyGen$^\circ(1^\lambda, R)$:
  $EK, VK \leftarrow \mathsf{KeyGen}(1^\lambda, R)$; $EK', VK' \leftarrow \mathsf{KeyGen}(1^\lambda, R_{EK})$ *where*

  $$(\mathbf{C}, \pi) \in R_{EK} \Leftrightarrow \bigwedge_{j=0}^{\ell-1} \mathsf{Verify}(VK_j, C_j) \wedge \mathsf{Verify}(VK, \mathbf{C}, \pi).$$

  Let $EK_j^\circ = (EK'_j, EK_j)$ *and* $VK_j^\circ = VK'_j$ *and output keys* $EK^\circ = (EK, EK')$ *and* $VK^\circ = VK'$.

- Commit$^\circ(EK_j^\circ, u_j, o_j)$:
  Return $C'_j \leftarrow \mathsf{Commit}(EK'_j, \mathsf{Commit}(EK_j, u_j, o_j), 0)$.

- Prove$^\circ((EK, EK'), \mathbf{u}, \mathbf{o})$: *Let* $C_j \leftarrow \mathsf{Commit}(EK_j, u_j, o_j)$, *and* $C_\pi \leftarrow \mathsf{Commit}(EK'_\ell, \pi, 0)$.

  Let $\pi \leftarrow \mathsf{Prove}(EK, \mathbf{u}, \mathbf{o})$; $\pi' \leftarrow \mathsf{Prove}(EK', (\mathbf{C}, \pi), (\mathbf{0}, 0))$.

  Return $\pi^\circ = (\pi', C_\pi)$.

- Verify$^\circ(VK'_j, C')$: $\mathsf{Verify}'(VK'_j, C')$.

- Verify$^\circ(VK', \mathbf{C}', (\pi', C_\pi))$:
  Return $\mathsf{Verify}(VK'_\ell, C_\pi) \wedge \mathsf{Verify}(VK', (\mathbf{C}', C_\pi), \pi')$.

Theorem 3 (Bootstrapped CPs) *If the CP schemes $\mathcal{P}$ and $\mathcal{P}'$ are knowledge-sound for $\{R_\lambda\}_{\lambda \in \mathbb{N}}$ and $\{R'_\lambda\}_{\lambda \in \mathbb{N}}$, respectively, then $\mathcal{P}^\circ$ as described in Protocol 1 is knowledge sound for $\{R_\lambda\}_{\lambda \in \mathbb{N}}$.*

Moreover, if $\mathcal{P}$ and $\mathcal{P}'$ are binding for some commitments, then $\mathcal{P}^\circ$ is also binding for them, hence scheduled knowledge-sound. Finally, if $\mathcal{P}$ and $\mathcal{P}'$ are perfectly zero-knowledge then $\mathcal{P}^\circ$ is perfectly zero-knowledge.

This describes a single level of bootstrapping, but as the result is again a commit-and-prove scheme, unbounded bootstrapping is covered as well.

Proof of Theorem 3 (CP bootstrapping soundness): Consider an adversary $\mathcal{A}^\circ$ against the knowledge soundness of the constructed proof system. $\mathcal{A}^\circ$ takes EK° as input. We define an adversary $\mathcal{A}'$ that takes EK' and R_{EK} as its input. $\mathcal{A}'$ obtains EK from R_{EK} and behaves like $\mathcal{A}^\circ$ except that it moves C_π from the proof to the commitments. We construct our proof as a sequence of games.

Game 1 is the original knowledge soundness game.

Game 2 is the same as Game 1, except that we run the extractors E' for $\mathcal{A}'$, whose existence is guaranteed by knowledge soundness, in parallel with $\mathcal{A}^\circ$ and on the same input and random tape as A'. Game 2 aborts without $\mathcal{A}^\circ$ winning if $\mathsf{Verify}(VK', (\mathbf{C}', C_\pi), \pi')$ accepts, but E' does not output a witness $\mathbf{C}, \pi$ such that $(\mathbf{C}, \pi) \notin R_{EK}$.

Lemma. The difference in the success probabilities of $\mathcal{A}^\circ$ between Game 1 and Game 2 is negligible, based on the knowledge soundness of $\mathcal{P}'$.

Let $\mathcal{E}^\circ$ be the extractor built from both $\mathcal{E}'$ and $\mathcal{E}$ for $\mathcal{P}$ and $\mathcal{P}'$ respectively. In Game 2 the success probability of $\mathcal{A}^\circ$ is negligible, as R_{EK} guarantees that $\mathsf{Verify}(VK, \mathbf{C}, \pi)$ accepts, and because $\mathcal{P}$ is a knowledge sound proof system for the relation R_λ. ∎

In the bootstrapping theorem we avoid controversial auxiliary input [13], contrary to [23]. The missing information for running $\mathcal{A}'$ is be obtained from the relation R_{EK} being proven.

4 Geppetto's CP Protocol

We now construct an efficient commit-and-prove protocol for a relation R for the MultiQAP $Q^\star$ derived from multiple QAPs Q_i, as described in §2.4.

4.1 MultiQAPs as Polynomials

We use Pinocchio's technique (which originated with Gennaro et al. [33]) to lift quadratic programs to polynomials.

For a $d \times \rho$ matrix V, we define polynomials $v_k(x)$ by polynomial interpolation over d roots, such that for $k \in [\rho]$: $v_k(r) = V_{rk}$ for all $r \in [d]$, and similarly for polynomials $w_k(x)$ and $y_k(x)$. Finally, we define the polynomial $d(x)$ which has all $r \in [d]$ as roots. We say that the polynomial MultiQAP is satisfied by $\mathbf{u}$ if $d(x)$ divides $p(x)$, where:

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

We use MultiQAPs to prove statements about shared state. To achieve this, the polynomials corresponding to shared values (i.e., bus values in S) need to fulfill an additional condition.

Definition 10 (Commitment Compatible MultiQAP)
Consider a polynomial MultiQAP Q and a partitioning $\mathcal{I} = \{I_0, \ldots, I_{\ell-1}\}$ of $[\rho]$. Q is commitment compatible for binding subset S if the polynomials in each set $\{y_k(x)\}_{k \in I_j}$, $j \in S$, are linearly independent, meaning that no linear combination of them cancels all coefficients, and all polynomials in the set $\{v_k(x), w_k(x)\}_{j \in S, k \in I_j}$ are 0.

By inspection, our MultiQAP construction in §2.4 is commitment compatible.

4.2 Commit-and-Prove Scheme for MultiQAPs

Geppetto's protocol inherits techniques from Pinocchio [51], the key differences are starting with MultiQAPs instead of QAPs, and splitting the prover's efforts into separate commit-and-prove steps.

We present our protocol in terms of a generic quadratic encoding E [33]. In our implementation, we use an encoding based on bilinear groups. Specifically, let e be a non-trivial bilinear map [16] $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ 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 $*$.

Protocol 2 (Geppetto)

- $\tau \leftarrow \mathsf{KeyGen}_1(1^\lambda)$: *Choose* $s, \{\alpha_{v,j}, \alpha_{w,j}, \alpha_{y,j}\}_{j\in[\ell]}, r_v,$ $r_w \overset{R}{\leftarrow} \mathbb{F}$. *Construct* τ *as* $(\tau_S, \tau_E) = (s, \{\alpha_{v,j}, \alpha_{w,j}, \alpha_{y,j}\}_{j\in[\ell]}, r_v, r_w), (r_v, r_w)$.

- $(EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R_Q)$: *Choose* $\{\gamma_j, \beta_j\}_{j\in[\ell]} \overset{R}{\leftarrow} \mathbb{F}$. *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 commitment-compatible MultiQAP* $Q = (\rho, d, \ell, \mathcal{J}, \mathcal{V}, \mathcal{W}, \mathcal{Y}, d(x))$ *with binding subset S corresponding to* R_Q*, construct the public evaluation key EK as:*

$$(EK_j)_{j\in[\ell]}, \quad (E(s^i))_{i\in[d]},$$

where the commitment keys EK_j *are*

$$\begin{pmatrix} E_v(v_k(s)), & E_w(w_k(s)), & E_y(y_k(s)) \\ E_v(\alpha_{v,j}v_k(s)), & E_w(\alpha_{w,j}w_k(s)), & E_y(\alpha_{y,j}y_k(s)), \\ & E(\beta_j(r_v v_k(s) + r_w w_k(s) + r_y y_k(s))) & \\ E_v(d(s)), & E_w(d(s)) & E_y(d(s)), \\ E_v(\alpha_{v,j}d(s)), & E_w(\alpha_{w,j}d(s)) & E_y(\alpha_{y,j}d(s)), \\ E_v(\beta_j d(s)), & E_w(\beta_j d(s)), & E_y(\beta_j d(s)) \end{pmatrix}_{k\in I_j}$$

Construct the public verification key VK as:

$$(VK_j)_{i\in[\ell]}, \quad E(1), \quad E_y(d(s)),$$

where

$$VK_j = E(\alpha_{v,j}), E(\alpha_{w,j}), E(\alpha_{y,j}), E(\gamma_j), E(\beta_j\gamma_j).$$

Additionally VK includes commitment keys EK_j *for commitments that are recomputed at the verifier. Since EK and VK are public, the split into prover and verifier keys is primarily designed to minimize the verifier's storage overhead.*

- $C \leftarrow \mathsf{Commit}(EK_j, u_j, o_j)$:

The values in EK_j *allow us to define an extractable and perfectly hiding but not in all cases binding digest.*

Let $(c_k)_{k\in I_j} = u_j$.

For buses $j \in S$, *compute a commitment as:*

$$E_y(y^{(j)}(s)), E_v(\alpha_{y,j}y^{(j)}(s)), E(\beta_j(r_y y^{(j)}(s)));$$

and for $j \notin S$, *compute a non-binding digest as:*

$$\begin{array}{lll} E_v(v^{(j)}(s)), & E_w(w^{(j)}(s)), & E_y(y^{(j)}(s)), \\ E_v(\alpha_{v,j}v^{(j)}(s)), & E_w(\alpha_{w,j}w^{(j)}(s)), & E_y(\alpha_{y,j}y^{(j)}(s)), \\ E(\beta_j(r_v v^{(j)}(s) + r_w w^{(j)}(s) + r_y y^{(j)}(s))) & & \end{array}$$

where $o_j = (o_v, o_w, o_y)$, *and* $v^{(j)}(s) = \sum_{k\in I_j} c_k v_k(s) + o_v d(s)$ *(and similarly for* $w^{(j)}(s)$ *and* $y^{(j)}(s)$*). For commitments* $v^{(j)}(s), v^{(j)}(s), o_v, o_w$ *are all 0. Note that all of these terms can be computed using the values in* EK_j*, thanks to the linear homomorphism of the encoding E.*

- $\pi \leftarrow \mathsf{Prove}(EK, \mathbf{u}, \mathbf{o})$: *Let* $\mathbf{c} = u_0, \ldots, u_{\ell-1}$. *Parse* o_j *as* $(o_{j,v}, o_{j,w}, o_{j,y})$ *and use the coefficients* $\mathbf{c}$ *to calculate:* $v(x) = \sum_{k\in[\rho]} c_k v_k(x) + \sum_{j\in[\ell]} o_{j,v} d(x)$, *and similarly for* $w(x)$, *and* $y(x)$.

  Just as in a standard QAP proof, calculate $h(x)$ *such that* $h(x)d(x) = v(x)w(x) - y(x)$, *i.e., the polynomial that proves that* $d(x)$ *divides* $v(x)w(x) - y(x)$*. Calculate the proof as* $\pi \leftarrow E(h(s))$ *using the* $E(s^i)$ *terms in EK.*

- $\{0,1\} \leftarrow \mathsf{Verify}(VK_j, C_j)$: *Verify the digest* C_j *by checking*

$$E_v(v^{(j)}(s)) * E(\alpha_{v,j}) = E_v(\alpha_{v,j}v^{(j)}(s)) * E(1)$$
$$E_w(w^{(j)}(s)) * E(\alpha_{w,j}) = E_w(\alpha_{w,j}w^{(j)}(s)) * E(1)$$
$$E_y(y^{(j)}(s)) * E(\alpha_{y,j}) = E_y(\alpha_{y,j}y^{(j)}(s)) * E(1)$$

and

$$E\left(\beta_j(r_v v^{(j)}(s) + r_w w^{(j)}(s) + r_y y^{(j)}(s))\right) * E(\gamma_j) =$$
$$\left(E_v(v^{(j)}(s)) + E_y(y^{(j)}(s))\right) * E(\beta_j\gamma_j) + E(\beta_j\gamma_j) * E_w(w^{(j)}(s)).$$

  (Note that we do not require $\alpha_{v,j}$ *and* $\alpha_{w,j}$ *checks for commitments and can simplify the* β_j *checks.)*

- $\{0,1\} \leftarrow \mathsf{Verify}(VK, C_0, \ldots, C_{\ell-1}, \pi)$: *The verifier then combines the terms from the commitments to perform the divisibility check on the proof term* $E(h(s))$ *in* π:

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

As described, the protocol supports non-interactive zero-knowledge proofs, in addition to verifiable computation. For applications that only desire VC, the multiples of $d(s)$ in the *EK* may be omitted, as can the use of commitment randomizations o in the digest steps.

Theorem 4 *Protocol 2 has binding commitments, as defined by Definition 5 under the d-SDH assumption.*

Theorem 5 *Protocol 2 is a knowledge-sound commit-and-prove scheme, as defined by Definition 6.*

Theorem 6 *Protocol 2 is a perfectly zero-knowledge commit-and-prove scheme, as defined by Definition 8.*

Like the protocol, the proofs inherit techniques from Pinocchio's proof. Before presenting the proofs, we recall its cryptographic assumptions.

4.3 Asymptotic Cryptographic Assumptions

To prove the security of Geppetto's protocol (§4), we make the same hardness assumptions as Pinocchio [51], summarized below.

Let λ be a security parameter, $q = \mathrm{poly}(\lambda)$, and let $\mathcal{A}$ be a non-uniform probabilistic polynomial time adversary. For each

λ, let $\mathcal{G}(1^\lambda)$ output a description $\mathcal{G}_\lambda = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathbb{F}_{\tilde{p}})$ of a cyclic bilinear group [16] of increasing order p defined over a base field of characteristic $\tilde{p}$, where $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is the usual pairing (bilinear map) function. For $(g_1, g_2) = g \in \mathbb{G}_1 \times \mathbb{G}_2$ we write g^x to denote (g_1^x, g_2^x). We write $\mathbb{G}^*$ for $\mathbb{G} \backslash \{1\}$.

Assumption 1 (q-PKE [37]) *The q-power knowledge of exponent assumption holds for $\mathcal{G}$ if for all $\mathcal{A}$ there exists a probabilistic polynomial time extractor $\chi_\mathcal{A}$ such that for all benign state generators $\mathcal{S}$[3]*

$$Pr[\quad \mathcal{G}_\lambda \leftarrow \mathcal{G}(1^\lambda) \, ; \, g \leftarrow \mathbb{G}_1^* \times \mathbb{G}_2^* \, ; \, \alpha, s \leftarrow \mathbb{Z}_p^* \, ;$$
$$\sigma \leftarrow (\mathcal{G}_\lambda, g, g^s, \ldots, g^{s^q}, g^\alpha, g^{\alpha s}, \ldots, g^{\alpha s^q})$$
$$z \leftarrow \mathcal{S}(\mathcal{G}_\lambda, g, g^s, \ldots, g^{s^q}) \, ;$$
$$(c, \hat{c} \, ; \, a_0, \ldots, a_q) \leftarrow (\mathcal{A} \parallel \chi_\mathcal{A})(\sigma, z) :$$
$$\hat{c} = c^\alpha \wedge c \neq \prod_{i=0}^q g_1^{a_i s^i}] = \mathtt{negl}(\lambda)$$

Note that $(y; z) \leftarrow (\mathcal{A} \parallel \chi_\mathcal{A})(x)$ signifies that on input x, $\mathcal{A}$ outputs y, and that $\chi_\mathcal{A}$, given the same input x and $\mathcal{A}$'s randomness, produces z.

Assumption 2 (q-PDH [37]) *The q-power Diffie-Hellman (q-PDH) assumption holds for $\mathcal{G}$ if for all $\mathcal{A}$ we have*

$$Pr[\quad \mathcal{G}_\lambda \leftarrow \mathcal{G}(1^\lambda) \, ; \, g \leftarrow \mathbb{G}_1^* \times \mathbb{G}_2^* \, ; \, s \leftarrow \mathbb{Z}_p^* \, ;$$
$$\sigma \leftarrow (\mathcal{G}_\lambda, g, g^s, \ldots, g^{s^q}, g^{s^{q+2}}, \ldots, g^{s^{2q}}) \, ;$$
$$y \leftarrow \mathcal{A}(\sigma) : y = g_1^{s^{q+1}}] = \mathtt{negl}(\lambda).$$

Assumption 3 (q-SDH [15, 31]) *The q-strong Diffie-Hellman (q-SDH) assumption holds for $\mathcal{G}$ if for all $\mathcal{A}$:*

$$Pr[\, (\mathcal{G}_\lambda \leftarrow \mathcal{G}(1^\lambda) \, ; \, g \leftarrow \mathbb{G}_1^* \times \mathbb{G}_2^* \, ; \, s \leftarrow \mathbb{Z}_p^* \, ; \sigma \leftarrow (\mathcal{G}_\lambda, g, g^s, \ldots, g^{s^q}) \, ;$$
$$y \leftarrow \mathcal{A}(\sigma) : y = e(g_1, g_2)^{\frac{1}{s+c}}, c \in \mathbb{Z}_p^*] = \mathtt{negl}(\lambda).$$

4.4 Proofs of Security

Proof of Theorem 4 (Binding): We use the $E(s^i)$ values given as input to the d-SDH assumption to generate EK and VK. An adversary that breaks the binding property produces $u = (c_k)_{k \in I_j}, o, u' = (c'_k)_{k \in I_j}, o', u \neq u'$, such that $\varphi(s) = \sum_{k \in I_j} c_k y_k(s) + od(s) - \sum_{k \in I_j} c'_k y_k(s) - o'd(s) = 0$, i.e. s is a root of $\varphi(s)$, and by factoring $\varphi(x)$, a reduction can easily find s and thus break d-SDH. ∎

Proof of Theorem 5 (Knowledge Soundness): Consider an efficient adversary $\mathcal{A}$ who succeeds in the knowledge-soundness game in Definition 6. Given $\mathcal{A}$, we need to show that there exists an algorithm $\mathcal{E}$ that, given the same randomness, input, and auxiliary information as $\mathcal{A}$, and the additional trapdoor $\tau_\mathcal{E}$, produces the witnesses and openings $\mathcal{A}$ uses in its commitments and proof.

We will show that $\mathcal{E}$ exists if extractors $\chi_{v,j}, \chi_{w,j}, \chi_{y,j}, j \in [\ell]$ exist. We do this by constructing $\mathcal{E}$ from these constituent extractors. They in turn exist under the d-PKE assumption, whenever $\mathcal{A}$ exists, as we can construct the corresponding adversaries $\mathcal{A}_{v,j}, \mathcal{A}_{w,j}$, and $\mathcal{A}_{y,j}$ from $\mathcal{A}$.

---

[3]The use of a state generator was suggested by Jens Groth.

To give more detail, we show how to construct $\mathcal{A}_{v,j}, j \in [\ell]$ from $\mathcal{A}$. Suppose $\mathcal{A}$ outputs commitments $C_0, \ldots, C_{\ell-1}$ and proof $\pi = H$, where each $C_j = (V_j, V'_j, W_j, W'_j, Y_j, Y'_j, Z_j), j \in [\ell]$. Each adversary $\mathcal{A}_{v,j}$ takes $\{E(s^i)\}_{i \in [0..d]}, \{E(\alpha_{v,j} s^i)\}_{i \in [0..d]}$ as its main input and $(EK \backslash (\{E_v(v_k(s)) \cup E_v(\alpha_{v,j} v_k(s))\}_{k \in I_j}), r_v, VK \backslash (E(\alpha_{v,j})))$ as its z input. Note that for the assumption $\alpha = \alpha_{v,j}$, and that z is independent of α and thus computable by the state generator $\mathcal{S}$. $\mathcal{A}_{v,j}$ runs $\mathcal{A}$ after extending z to the full EK and VK by using its main input and r_v to recomputing $(\{E_v(v_k(s)) \cup E_v(\alpha_{v,j} v_k(s))\}_{k \in I_j})$, but outputs only (V_j, V'_j). We construct adversaries $\mathcal{A}_{w,j}$ and $\mathcal{A}_{y,j}$ in a similar fashion.

Since verification succeeds, we know that $V_j * E_v(\alpha_{v,j}) = V'_j * E_v(1)$. Under the d-PKE assumption, we thus know that for every $\mathcal{A}_{v,j}$ there exists an extractor $\chi_{v,j}$ (taking the same inputs) that produces the coefficients of a polynomial $V_j(x)$, such that $E_v(V_j(s)) = V_j$. And similarly with respect to $W_j(x)$ for $\mathcal{A}_{w,j}$ and $Y_j(x)$ for $\mathcal{A}_{y,j}$.

Given the constituent extractors $\chi_{v,j}, \chi_{w,j}, \chi_{y,j}, \mathcal{E}$ computes $\{E(\alpha_{v,j} s^i)\}_{i \in [0..d]}, \{E(\alpha_{w,j} s^i)\}_{i \in [0..d]}$, and $\{E(\alpha_{y,j} s^i)\}_{i \in [0..d]}$ using r_v and r_w and splits EK into σ and z to call them on their correct inputs.

We now consider several games to bound the success probability of the main extractor $\mathcal{E}$.

- *Game 1* is the same as the original knowledge soundness game, except that we abort without $\mathcal{A}$ winning if for verifying commitments $\chi_{v,j}, \chi_{w,j}, \chi_{y,j}, j \in [\ell]$ do not succeed in producing polynomials $V_j(x)$ such that $V_j \neq E_v(V_j(s))$ (and similarly for $W_j(x)$ and $Y_j(x)$).

  Lemma 2 *The difference in the success probability of $\mathcal{A}$ between the original knowledge soundness game and Game 1 is bounded (via a union bound) by the sum of the failure probabilities of $\chi_{v,j}, \chi_{w,j}, \chi_{y,j}, j \in [\ell]$.*

- *Game 2* is like Game 1, except that it sets $H(x) = (V(x)W(x) - Y(x))/d(x)$, where $V(x) = \sum_{j \in [\ell]} V_j(x)$, where $V_j(x)$ is the polynomial extracted by $\chi_{v,j}$ (and similarly for $W(x)$ and $Y(x)$). Game 2 aborts without $\mathcal{A}$ winning if $H(x)$ has a non-trivial denominator.

  Lemma 3 *The difference in the success probability of $\mathcal{A}$ between Game 1 and Game 2 is bounded by the success probability of an attacker $\mathcal{B}_1$ breaking the $2q$-SDH assumption, and is thus negligible.*

- *Game 3* is like Game 2, except that it aborts without $\mathcal{A}$ winning, if one of the commitment polynomial triples $R_j(x) = (V'_j(x), W'_j(x), Y'_j(x))$, where $V'_j(x) = V_j(x)$ mod $d(x)$ (and similarly for W'_j and Y'_j), is not in the linear subspace S_j, generated by the polynomial triples $\{(v_k(x), w_k(x), y_k(x))\}_{k \in I_j}$, where the linear operations are done element-wise.

  Lemma 4 *The difference in the success probability of $\mathcal{A}$ between Game 2 and Game 3 is bounded by the success probability of an attacker $\mathcal{B}_2$ breaking the q-PDH assumption, and is thus negligible.*

In Game 3, $\mathcal{A}$ has zero success probability.

For this, we show that the aborts in Game 2 and 3 are the only two cases in which the proof verifies but the extracted witness is invalid. We assume that neither case 1 nor case 2 holds; i.e., $H(x)$ has no non-trivial denominator, and each of the $R_j(x)$ is in the linear subspace S_j. We will show that $V(x)' = V(x) \bmod d(x)$, $W'(x) = W(x) \bmod d(x)$, and $Y'(x) = Y(x) \bmod d(x)$ are a MultiQAP solution; i.e., they can be written as linear combinations of $\{v_k(x)_{k \in [0..\rho]}\}$, $\{w_k(x)_{k \in [0..\rho]}\}$ and $\{y_k(x)_{k \in [0..\rho]}\}$ using the same coefficients $\mathbf{c}$ as required by the commit-and-prove relation $R^\star$, and $V'(x)W'(x) - Y'(x)$ is divisible by $d(x)$.

Since for each j, $(V_j'(x), W_j'(x), Y_j'(x))$, is in the linear subspace, generated by the tuples $\{(v_k(x), w_k(x), y_k(x))\}_{k \in I_j}$, we can write $V_j'(x) = \sum_{k \in I_j} c_k v_k(x)$, $W_j'(x) = \sum_{k \in I_j} c_k w_k(x)$, $Y_j'(x) = \sum_{k \in I_j} c_k y_k(x)$. We thus have that $V'(x)$, $W'(x)$, and $Y'(x)$ indeed can be written as the same linear combination $\{c_k\}_{k \in [0..\rho]}$ of their polynomial sets, as required in a MultiQAP.

As $V_j'(x) = V_j(x) \bmod d(x)$, we also have that $V_j(x) = V_j'(x) + o_{j,v}d(x)$ for some $o_{j,v}$, and similarly for $W_j(x)$ and $Y_j(x)$. Therefore, $V(x) = \sum_{j \in [\ell], k \in I_j} c_k v_k(x) + \sum_j o_{j,v}d(x)$, $W(x) = \sum_{j \in [\ell], k \in I_j} c_k w_k(x) + \sum_j o_{j,w}d(x)$, and $Y(x) = \sum_{j \in [\ell], k \in I_j} c_k y_k(x) + \sum_j o_{j,y}d(x)$. Since $H(x)$ has no non-trivial denominator, it follows that $d(x)$ evenly divides $V(x)W(x) - Y(x)$ and thus also $V'(x)W'(x) - Y'(x)$.

The commitment opening values are defined as $o_j = (o_{v,j}, o_{w,j}, o_{y,j})$, for $j \in [\ell]$. Hence $V'(x)$, $W'(x)$, and $Y'(x)$ constitute a MultiQAP solution $\mathbf{c}$ for the commit-and-prove relation R, and hence valid witness and openings $u_0, \ldots, u_{\ell-1}, o_0, \ldots, o_{\ell-1}$. ∎

Lemma 3 follows directly from the definitions of extractors $\chi_{\cdot, j}$. The proofs of Lemmas 3 and 4 follow the security proof of PGHR [51] and are omitted for lack of space. The main difference is that we range over multiple uniform banks, instead of the special purpose banks of Pinocchio.

Proof of Lemmas 3 and 4: Given $\mathcal{A}$ and $\mathcal{E}$ such that Game 2 or Game 3 aborts for a verifying proof, we construct two algorithms $\mathcal{B}_1$ and $\mathcal{B}_2$ that break the 2q-SDH or q-PDH assumption respectively where $q = 4d + 4$. The proof is adapted from PGHR [51]. Like them, we describe $\mathcal{B}_1$ and $\mathcal{B}_2$ interleaved, sharing large parts of the exposition. $\mathcal{B}_2$ is given q-PDH challenge instance $E(1), E(s), \ldots, E(s^q), E(s^{q+2}), \ldots, E(s^{2q})$. $\mathcal{B}_1$ is given the same challenge, except for an extra $E(s^{q+1})$ which will remain unused.

Both $\mathcal{B}_1$ and $\mathcal{B}_2$ generate a MultiQAP evaluation key and a verification key for the relation R and provide them to $\mathcal{A}$, using the same structure as in Protocol 2. The only difference is that they choose r_v' and r_w' at random and implicitly set $r_v = r_v's^{d+1}$, $r_w = r_w's^{2(d+1)}$, letting $r_y' = r_v'r_w'$.

Regarding the β_j terms, write the final term in the commitment for index j as:

$$E(\beta_j(r_v v^{(j)}(s) + r_w w^{(j)}(s) + r_y y^{(j)}(s)))$$
$$= E(\beta_j(r_v's^{d+1}v^{(j)}(s) + r_w's^{2(d+1)}w^{(j)}(s) + r_y's^{3(d+1)}y^{(j)}(s))) \quad (1)$$

That is, inside the encoding, β_j is multiplied with a certain polynomial that is evaluated at s. $\mathcal{B}_1$ and $\mathcal{B}_2$ also generate β_j as a polynomial evaluated at s. In particular, they sets $\beta_j = s^{q-(4d+3)}\beta_j^{poly}(s)$, where $\beta_j^{poly}(x)$ is a polynomial of degree at most $3d + 3$ sampled uniformly at random such that $\beta_j^{poly}(x) \cdot (r_v v_k(x) + r_w x^{d+1}w_k(x) + r_y x^{2(d+1)}y_k(x))$ has a zero coefficient in front of x^{3d+3} for all k. We know that such a polynomial $\beta_j^{poly}(x)$ exists by Lemma 10 of GGPR [33]. By inspection, when we now write out β_j in terms of s, we see that the exponent in Equation 1 has a zero in front of s^{q+1}, and also that the powers of s only go up to degree $q + 3d + 3 \leq 2q$. Therefore, $\mathcal{B}_1$ and $\mathcal{B}_2$ can efficiently generate the terms in the evaluation key that contain β_j using the elements given in their respective challenges. $\mathcal{B}_1$ and $\mathcal{B}_2$ can clearly perform the same steps for all $j \in [\ell]$ and hence compute all of the β_j terms.

Regarding γ_j, $\mathcal{B}_1$ and $\mathcal{B}_2$ generate $\bar{\gamma}_j$ uniformly at random from $\mathbb{F}$ and sets $\gamma_j = \bar{\gamma}_j s^{q+2}$. $\mathcal{B}_1$ and $\mathcal{B}_2$ can generate $E(\gamma_j)$ efficiently from their challenges, since $E(s^{q+2})$ is given. Also, $\beta_j\gamma_j = s^{q-(4d+3)} \cdot \beta_j^{poly}(s)\bar{\gamma}_j s^{q+2}$ does not have a term s^{q+1} and has degree at most $q - (4d+3) + (3d+3) + (q+2) \leq 2q$, and so $\mathcal{B}_1$ and $\mathcal{B}_2$ can generate $E(\beta_j\gamma_j)$ from the elements in its challenge.

Similarly none of the other elements in the keys contain a term s^{q+1} in the exponent, since all of the polynomials $v_k(x)$, $w_k(x)$ and $y_k(x)$ are of degree d and $q \geq 4d + 4$. Hence, $\mathcal{B}_1$ and $\mathcal{B}_2$ can generate them using the terms from their challenges.

Thus, the evaluation and verification keys generated by $\mathcal{B}_1$ and $\mathcal{B}_2$ have a distribution statistically identical to the real scheme.

Given *EK* and *VK*, $\mathcal{A}$ can run the Commit, Compute, and Verify algorithms on its own.

Since the proof verifies correctly, we have that $V_j' * E_v(1) = V_j * E_v(\alpha_{v,j})$. As a result, for each commitment, $\mathcal{B}_1$ and $\mathcal{B}_2$ can run the d-PKE extractors $\chi_{v,j}, \chi_{w,j}, \chi_{y,j}$ used by $\mathcal{E}$ to recover a polynomial $V_j(x)$ of degree at most d such that $V_j = E_v(V_j(s))$.

Similarly, they recover $W_j(x)$ and $Y_j(x)$ such that $W_j = E_w(W_j(s))$ and $Y_j = E_y(Y_j(s))$. Then, they set $H(x) = V(x)W(x) - Y(x)/d(x)$, where $V(x) = \sum_{j \in [\ell]} V_j(x)$ (and similarly for $W(x)$ and $Y(x)$).

Since the proof verifies, but either Game 2 or Game 3 aborts there are two possible cases depending on which game aborted: **(1)** $H(x)$ has a non-trivial denominator, or **(2)** One of the commitment polynomials triples $(V_j'(x), W_j'(x), Y_j'(x))$, is not in the linear subspace, generated by $\{(v_k(x), w_k(x), y_k(x))\}_{k \in I_j}$.

In Case 1, $d(x)$ does not divide $p(x) := V(x)W(x) - Y(x)$. Let $(x - r)$ be a polynomial that divides $d(x)$ but not $p(x)$, and let $T(x) = d(x)/(x - r)$. Let $\bar{d}(x) = \gcd(d(x), p(x))$. Since $d(x)$ and $p(x)$ have degrees at most d and $2d$ respectively, $\mathcal{B}_1$ can use the extended Euclidean algorithm for polynomials to find polynomials $a(x)$ and $b(x)$ of degrees $2d - 1$ and $d - 1$ respectively such that $a(x)d(x) + b(x)p(x) = \bar{d}(x)$. Set $A(x) = a(x) \cdot (T(x)/\bar{d}(x))$ and $B(x) = b(x) \cdot (T(x)/\bar{d}(x))$; these polynomials have no denominator since $\bar{d}(x)$ divides $T(x)$. Then $A(x)d(x) + B(x)p(x) = T(x)$. Dividing by $d(x)$, we have $A(x) + B(x)H(x) = 1/(x - r)$. Since $A(x)$ and $B(x)$ have degree at most $2d - 1 \leq q$, $\mathcal{B}_1$ can use the terms in its challenge to com-

pute $E(A(s)) + E(B(s)) \cdot E(H) = E(1/(s-r))$, and thus solve $2q$-SDH.

In Case 2, for at least one of the digests $j \in [\ell]$, there does not exist $\{c_k\}_{k \in I_j}$ such that $V'_j(x) = \sum_{k \in I_j} c_k v_k(x)$, $W'_j(x) = \sum_{k \in I_j} c_k w_k(x)$, and $Y'_j(x) = \sum_{k \in I_j} c_k y_k(x)$.

Since $V'_j(x)$, $W'_j(x)$ and $Y'_j(x)$ have degree d, and since the linear subspaces $\{r'_v x^{d+1+i} | i \in [0,d]\}$, $\{r'_w x^{2(d+1)+i} | i \in [0,d]\}$, and $\{r'_w x^{3(d+1)+i} | i \in [0,d]\}$ are disjoint (except at the origin), this also means that $R_j(x) = r'_v x^{d+1} V'_j(x) + r'_w x^{2(d+1)} W'_j(x) + r'_y x^{3(d+1)} Y'_j(x)$ is not in the linear subspace, S_j, generated by the polynomials $\{r_k(x) = r'_v x^{d+1} v_k(x) + r'_w x^{2(d+1)} w_k(x) + r'_y x^{3(d+1)} y_k(x)\}_{k \in I_j}$.

By Lemma 10 [33], we have that with high probability $x^{q-(4d+3)} \cdot \beta_j^{poly}(x) \cdot (r'_v x^{d+1} V'_j(x) + r'_w x^{2(d+1)} W'_j(x) + r'_y x^{3(d+1)} Y'_j(x))$ has a non-zero coefficient for the term x^{q+1}. Thus, $\mathcal{B}_2$ can subtract off all elements of the form $E(s^i)$ where $i \neq q+1$ from $Z_j = E(s^{q-(4d+3)} \beta_j^{poly}(s)(r'_v s^{d+1} V_j(s) + r'_w s^{2(d+1)} W_j(s) + r'_y s^{3(d+1)} Y_j(s)))$ to obtain $E(s^{q+1})$. This breaks the q-PDH assumption. ∎

Lemma 5 (Restating of Lemma 10 [33]) *Let $F[x]^d$ denote polynomials over field F of degree at most d. Let $F[x]^{\neg d}$ denote polynomials over F that have a zero coefficient for x^d. For some q, let $U \subset F[x]^q$, and let $span(U)$ denote the polynomials that can be generated as linear combinations of the polynomials in U. Let $a(x) \in F[x]^{q+1}$ be generated uniformly at random subject to the constraint that $\{a(x) \cdot u(x) : u(x) \in U\} \subset F[x]^{\neg q+1}$. Let $s \in F \setminus \{0\}$. Then, for all algorithms $\mathcal{A}$,*

$$Pr[u^*(x) \leftarrow \mathcal{A}(U, s, a(s)) : u^*(x) \notin span(U)$$
$$\wedge a(x) \cdot u^*(x) \in F[x]^{\neg q+1}] \leq 1/|F| \,.$$

Proof of Theorem 6 (Zero Knowledge): We first define the simulator $(\{C_j\}_{j \notin S}, \pi) \leftarrow \mathcal{S}(\tau_S, \{C_j\}_{j \in S})$: We have that each $C_j = (0, 0, 0, 0, Y_j, Y'_j, Z_j)$, $j \in S$. Sample $\delta_{j,v}, \delta_{j,w}, \delta_{j,y}$ for $j \notin S$.
Compute $C_j = (E_v(\delta_{j,v}), E_v(\alpha_{v,j} \delta_{j,v}), E_w(\delta_{j,w}), E_w(\alpha_{w,j} \delta_{j,w}), E_y(\delta_{j,y}), E_y(\alpha_{y,j} \delta_{j,y}), E(\beta_j(r_v \delta_{v,j} + r_w \delta_{w,j} + r_y \delta_{y,j})))$, for $j \notin S$.
Let

$$h = \frac{(\sum_{j \notin S} \delta_{v,j}) * (\sum_{j \notin S} \delta_{w,j}) - \sum_{j \notin S} \delta_{y,j}}{d(s)} \,.$$

Return $(\{C_j\}_{j \notin S}, \pi = E(h) - d(s)^{-1} \sum_{j \in S} Y_j)$.

Perfect ZK follows from both the real and the simulated commitments $\{C_j\}_{j \notin S}$ being uniformly at random subject to the constraints of commitment verification. Once they are fixed, the verification equation of the proof uniquely determines π. ∎

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 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 crypto 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 §2.5, proof bootstrapping is a particularly compelling example of verifying cryptographic operations, since it allows us to condense a long series of proofs and commitments into a single proof and commitment.

Remarkably, Ben-Sasson et al. [10] recently discovered a pair of MNT curves [49] E and $\tilde{E}$ that are pairing friendly and, more importantly, not only can $\tilde{E}$ be embedded in E, but E can be embedded in $\tilde{E}$. While Ben-Sasson et al. use these curves to bootstrap the verification of individual CPU instructions, Geppetto can use them 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 several 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 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 [10].

To estimate the costs of using MNT curves at the 128-bit level used by Pinocchio, we coded up all of the relevant curve operations in Magma [17] 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 34-77x slower than a standard Pinocchio-style proof, while the constant pairing-based portion of proof verification would be 11x slower; subsequent batches would cost even more, due to technical challenges in the way the curves fit together [10].

As a pragmatic alternative, we propose and implement *bounded bootstrapping*. Specifically, we instantiate one version of Geppetto with the same highly efficient BN curve [7] employed by Pinocchio. We use the BN curve to generate a collection of commitments 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, we can, for example, verify signatures on the verification key built on the BN curve and then use that key to verify the BN commitments and proofs. However, the BN curve cannot efficiently embed the second curve, and so when generating keys, the client must 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. Our measurements suggest that this pragmatic approach saves us several orders of magnitude in performance.

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 [7], and then construct $\mathcal{G}_{OUT}$ accordingly with the Cocks-Pinch method [24]. 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 6 *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*

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

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).

Appendix A contains a proof and more construction details.

6 Implementation

The Geppetto system includes a library that expresses compilation blocks and buses, a compiler that evaluates C programs via LLVM, and libraries that support bootstrapped computation.

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 controls the structure of compound proofs and shared buses that connect them, explicitly controlling cost and amortization. This structure is expressed via library invocations, not compiler extensions.

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 the results with the accompanying cryptographic evidence. This structure provides a clear operational specification of the verified computation, even for complex proof schedules: when the main program completes, its outputs and return values must be equivalent to executing the entire program on a single trusted machine. A Geppetto program `sample.c` defines its outsourced functions and buses as follows:

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

void compute(bigdata *db, vector *in, vector *out) ...

BUS(DATA, bigdata)  // we define 3 buses
BUS(QUERY, vector)
BUS(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];
  ...
  DATA db = save_DATA(&M); // commit DATA just once
  for (i=0; i<N; i++) {
    QUERY q = save_QUERY(&query[i]);
    RESULT r = OUTSOURCE(job, db, q);
    load_RESULT(r,&result[i]);
  }
}
```

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 instance of `compute()`, and vary the vector. To this end, `sample.c` declares three buses for verifiable outsourced computation. Here, `BUS(QUERY, vector)` defines the QUERY bus datatype and accessor functions like `load_QUERY` analogous to RPC marshaling functions. Each bus can be assigned only once, and must be assigned before being loaded.

Compiled natively outside Geppetto, `geppetto.h` provides trivial definitions that implement QUERY as an in-memory buffer and OUTSOURCE as a local call.

During *compilation*, Geppetto interprets `sample.c`, using symbolic values for unknown inputs, to generate a verification key. In *prove mode*, Geppetto interprets `sample.c` with concrete values to produce the bus value commitments and proofs of each outsourced call. In *verify mode*, Geppetto produces a version of the program that replaces bus loads and outsource calls with cryptographic verifications; this version can be natively compiled with `clang -DVERIFY sample.c`. In every mode, the execution flow of `main` determines the schedule of calls to outsourced functions.

In verify mode, verification keys are initially loaded from files, buses are supplemented with cryptographic functions for verifying commitments, and `OUTSOURCE(job, db, q)` is implemented with:

```
RESULT verify_job(DATA b0, QUERY b1) {
  commitment cs[4];
  cs[0] = b0->c; // reuse commitment produced by save_DATA
  cs[1] = b1->c; // reuse commitment produced by save_QUERY
  RESULT b2 = load_recommit_RESULT();
  cs[2] = b2->c;
  load_verify_commit(&STATE.vk, &cs[3], C_job_LOCALS);
  cProof pi;
  load_cProof("job", &pi, outsource_id, RUN_TIME);
  verify_proof(&STATE.vk, &pi, 4, cs);
  return b2;
}
```

Just like `job`, `verify_job` takes two buses and returns a bus. The input buses propagate trusted commitments from the caller; in particular, the `bigdata` commitment is shared across all calls. The function loads the remote's proposed value for the output bus, verifies the bus' commitment evidence, and verifies the computation proof. If any verification fails, the program exits with an error.

6.1.1 MultiQAP Patterns

Geppetto provides additional support for two common commit-and-prove patterns.

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 C template for outsourcing each iteration of such loops, with a bus for the outer variables, whose cost 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 has three banks, two outsourced functions, and a refined loop that alternates calls between the two.

MapReduce Geppetto also provides a few generic templates for parallel loops (as outlined in §6.1) and MapReduce computations. Similar to sequential loops, we use a series of buses to succinctly share potentially many variables between mappers and reducers.

Automated QAP Partitioning Geppetto's libraries enable programmer-directed QAP partitioning. We experimented with automatically partitioning monolithic QAPs, expressed finding hyper-graph cuts. We had some success efficiently finding approximate cuts in graphs up to 200,000 equations with the METIS tool . However, the programmer-directed approach is

more flexible and better exploits regular structure such as loop iterations.

6.2 Symbolic Interpretation via LLVM

This section provides details on the construction of the Geppetto compiler. It elides QAP techniques already described in [51].

General-Purpose LLVM Front-End As a front-end compiler, we use clang [44], a fast full-fledged C compiler with rich syntax, standard semantics, and optimizations. Geppetto compilation to quadratic programs starts with a low-level, typed, integer-centric program representation. We run `clang -O2 -S -DQAP -emit-llvm sample.c -o sample.s`, where `-DQAP` declares but not defines Geppetto primitive types and functions.

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

Interpreting LLVM Bitcode Instead of directly emitting a circuit, Geppetto compiles and evaluates programs by symbolic interpretation of LLVM code. Our interpreter relies on a shallow embedding into F#, relying on the F# control stack and heap. Function calls are implemented by calls to an F# `call`, function, and mallocs are integer-array creations. Compile-time values are known to the interpreter, and used to specialize the equations we produce. Run-time values are treated symbolically, represented by an abstract domain, and generally involve allocating QAP equations.

Interpretation is cheap relative to cryptography, so we re-interpret the program to generate concrete values and verification evidence in 'prove' mode. Thus there are *two* related interpreters that differ in their interpretation of integer values.

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 and an equation. Global caches identify and eliminate common subexpressions.

We represent unknown integers as a triple of (i) a linear combinations 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. Our compile-time abstract integers precisely tracks possible intervals as ranges in $\mathbb{Z}$. This precision detects field overflows, and enables us defer truncation (truncating a 35 bit value to its 32 bit C representation). It also minimizes binary decompositions; these cost one equation per potentially active bit, and are important for fast exponentiations during bootstrapping.

After compilation, the MultiQAP is one QAP per function, plus 'linking' information for the buses they share. Thus, compilation traverses the function QAPs while keeping the

buses virtual, enumerating their coefficients only to generate the crypto keys.

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 control flow, one outsourced functions 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. Values of variables or equations allocated at compile time are stored as witnesses that comprise cryptographic digests. Evaluating some QAP optimizations requires 'interpretation hints' passed from the compiler. For example, before XORing a variable with a constant, a hint tells the evaluator whether binary decomposition is needed and how many bits of witness are required.

The evaluator also intercepts outsource and bank functions, producing commitments and proofs accordingly. Upon completion, the prover has collected 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. 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 runtime is spent multiplying and exponentiating elliptic curve points, we optimize these operations using standard pre-computation and batching techniques [51]. Our C++ library also implements an efficient $d \log d$ algorithm for the prover's polynomial division.

Primitive Libraries Whenever possible, we reflect (and even implement) primitive features of the interpreter as C types and functions. Pragmatically, it keeps our code base small, and lets us rely on standard (non-cryptographic) tools for testing and debugging purposes—for instance by comparing `printf`s 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 compiler interpreter allocates fresh local QAP variables, and the evaluation 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 sizes intricately depend 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 4 million equations have been generated.

6.3   Cryptographic Libraries and Bootstrapping

Geppetto has specific support to enable compilation of pro-grams that evaluate verifications to enable bootstrapping and other flexible applications of nested evaluation.

Field arithmetic and cryptography To support bootstrapping, we provide libraries that implement primitive field operations including addition, multiplication, division, and binary decomposition. These enable fast embedding of cryptography that benefits from 254-bit words. The field type is also implemented natively so that it works with both clang and Geppetto.

Our file 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 flexiblity (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.

As in prior work [10], 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 two cases at each iteration of the loop, due to the special treatment of infinite points in addition. To avoid these expensive branches, we add an initial summand and remove it at the end.

Bounded Bootstrapping Continuing with our example in §6.1, assume we wish to compress the *N* proofs by bootstrapping, by writing a function that aggregates the result vectors. To this end, we include another, similar but distinct copy of our Geppetto library which lets us 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, commitment 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.

For 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.

For 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).

For verifying, we simply compile the source program with the `-DVERIFY` flag for level 2.

Although our compiler formally supports additional levels of bootstrapping, all runs of Geppetto at level 2 must use the 'outer' curve presented in §5, and we do not currently provide cryptographic support for higher levels.

6.4 Branching and Energy-Saving

Continuing from the discussion in §6.1.1, we explain how Geppetto implements 'energy-saving' QAP encodings.

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, and we multiply commitments by 1 (a key element exponentiated by 0). 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.

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.

Op	BN Base	BN Twist	CP Base	CP Twist
Fixed Base Exp	$21.3\mu s$	$86.4\mu s$	$160.0\mu s$	$160.0\mu s$
Multi Exp (254 bit)	$55.5\mu s$	$238.9\mu s$	$450.0\mu s$	$449.6\mu s$
Pairing	0.7ms		4.9ms	
Field Add	43.6ns		43.9ns	
Field Mul	289.2ns		290.4ns	

Figure 4: **Microbenchmarks.** *Breakdown of the main sources of performance overhead in the larger protocol. Each value is the average of 90 trials. Standard deviations are less than 4%.*

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

7 Evaluation

Below, we evaluate the effect of Geppetto's optimizations on prover performance. 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.

7.1 Microbenchmarks

To calibrate our results, we briefly summarize the cost of our cryptographic primitives in Figure 4. We generally use a Barreto-Naehrig (BN) curve for generating commitments and proofs, and we use the Cocks-Pinch (CP) curve to handle embedded cryptographic computations like bootstrapping. The BN curve is asymmetric, meaning one source group (base) is cheaper than the other (twist). Geppetto's protocol is designed to keep most of the work on the base group.

The CP curve is slower than the BN for two reasons. First, it was chosen to support bounded bootstrapping, so it uses 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 [17], the CP curve should be within 2-4x of 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 [18]. At a micro level, Pantry's results suggest that hashing an element of state increases the degree of the QAP by ~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 ~0.03/byte, a savings of 375×. Even if we want to operate on 32-bit values, instead of full field elements, Geppetto only costs 0.25/byte, a savings of 45x. Since the QAP degree directly impacts proof time, as well as the size of and time to generate the keys, such savings translate into concrete performance improvements. Experiments to quantify these improvements are ongoing work.

A naive 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; 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.

7.3 Verifying Crypto 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 into the realm of feasibility. For example, a pairing only requires a QAP of degree 14K, an improvement of $3100\times$ vs. the naïve approach, while an exponentiation, i.e., g^x, increases the degree by $\sim$60 per bit in x.

Furthermore, as discussed in §5, our curves improve on the performance of previous bootstrapping curves [10] by 34-77x, at the expense of unbounded bootstrapping.

7.3.1 Bootstrapping

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

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 (§2.4) show is infeasible. Nonetheless, bootstrapping does come at cost. While bootstrapping, the "outer" QAP's degree grows with each commitment or proof that it must verify. We summarize these costs below assuming that the verification keys are known at compile-time.

- For each recomputed commitment, we increase the degree by 2K for each 32-bit integer value committed.

- For each bus commitment verification, we pay 33.8K (including the pairings needed for the bus alpha and beta checks).

- For each full commitment verification, we pay 79.6K (including 3 full alpha checks and a full beta check).

- For each proof verification, we pay 28.2K.

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

We also observe that the prover's cryptographic cost for "outer" proofs and commitments 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
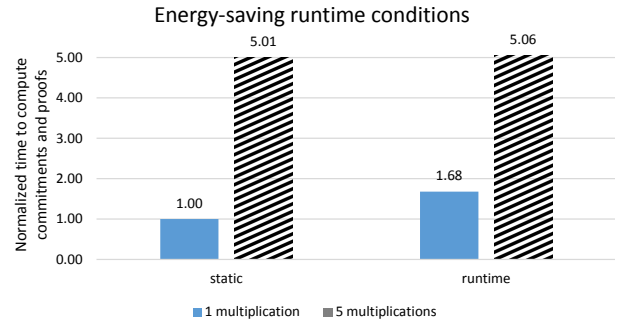

Figure 5: **Energy-Saving Circuits.** *The energy-saving multiplexer allows us to include an optional circuit that has low cost when unused.*

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 not yet practical, they are low enough that we can employ bootstrapping to scale the prover to previously unreachable 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 constant-size proof for a computation with a useful (i.e., not counting bootstrapping costs) QAP degree of over 50 M, an order of magnitude larger than previously reported verifications. This experiment also allows us to measure an effective "clock rate" for our prover. When evaluating the computation, the prover executes 24M LLVM instructions and generates a proof in 152 minutes, giving us a clock rate of 2.6 KHz. While low, this is five orders of magnitude faster than the unbounded bootstrapping in previous work [10], which reported a clock rate of 26 milliHz with a lower security level, though that work offers more generality than Geppetto.

7.4 Energy-Saving Circuits

As a targeted microbenchmark to evaluate the benefits of energy-saving circuits (§2.6), in Figure 5, 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 $5\times$), and costs a negligible 1% in the five-multiply case.

7.5 Compiler

Some previous verifiable computations systems do not include a compiler [25, 58], while those that do [11, 18, 51] have typically compiled small examples with less than 100 lines of C code. In contrast, Geppetto's compiler handles large cryptographic libraries, with the largest clocking in at 4,159 SLOC [62] of com-

plex 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 [10, 51, 55, 58]. 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). A classic way to condense state is to commit to it via a hash [9, 14, 33, 47]. When specifying the IO to a program P, 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 [18], implements collision-resistant hashing on top of the existing QAP-based Pinocchio [51] and Zaatar protocols [55] and uses it both for IO specification and for the intermediate state in MapReduce computations. With Geppetto, the intermediate state produced by the mappers and consumed by the reducers is represented simply as a MultiQAP bus. Hence, each value on the bus only increases the QAP's degree by 1, and crypto operations are only needed when the values are written.

As an orthogonal contribution, Pantry uses hashes to build a memory abstraction based on Merkle trees [47], though subsequent work [11, 61] suggests that memory routing networks [9] generally perform better. Regardless, these techniques for dynamic memory accesses are orthogonal to Geppetto, and as recent work demonstrates [61], can be integrated quite naturally with Geppetto for computations that need them.

As discussed in §5, Ben-Sasson et al. [10] propose to use suitably related 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 leads 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 would 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 [41]. Canetti et al. [20] define CP schemes in the UC model and realize such schemes in the $\mathcal{F}_{ZK}$-hybrid model. Escala and Groth [28] design CP schemes from Groth-Sahai proofs [38].

Zero Knowledge Several systems compile high-level functions to zero-knowledge (ZK) proofs [1, 5, 46]. Compilers from Almeida et al. [1] and Meiklejohn et al. [46] build on Σ-protocols [26], while the work of Backes et al. [5] uses Groth-Sahai ZK proofs [38]. 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, QAPable 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 verification of hidden computations. Geppetto's scalable compiler exposes this power and flexibility to developers, bringing verifiable computation one step closer to practicality.

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] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López. Faster explicit formulas for computing pairings over ordinary curves. In *EUROCRYPT*, 2011.

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

[4] M. Backes, D. Fiore, and R. M. Reischuk. Nearly practical and privacy-preserving proofs on authenticated data. IACR Cryptology ePrint Archive, Report 2004/155, 2014.

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120, 2013.

[13] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen. On the existence of extractable one-way functions. In *Symposium on Theory of Computing, STOC*, 2014.

[14] 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.

[15] D. Boneh and X. Boyen. Short signatures without random oracles. In *EUROCRYPT*, 2004.

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

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

[18] 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.

[19] J. Camenisch and A. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In *EUROCRYPT*, volume 2045, 2001.

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

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

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

[23] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn. Succinct malleable NIZKs and an application to compact shuffles. In *TCC*, pages 100–119, 2013.

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

[25] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.

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

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

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

[29] D. Freeman, M. Scott, and E. Teske. A taxonomy of pairing-friendly elliptic curves. *J. Cryptology*, 23(2):224–280, 2010.

[30] 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.

[31] R. Gennaro. Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks. In *CRYPTO*, 2004.

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

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

[34] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.

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

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

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

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

[39] K. Ireland and M. I. Rosen. *A classical introduction to modern number theory*, volume 1982. Springer, 1982.

[40] G. O. Karame, M. Strasser, and S. Capkun. Secure remote execution of sequential computations. In *Intl. Conf. on Information and Communications Security*, 2009.

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

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

[43] 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.

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

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

[46] 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.

[47] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

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

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

[50] M. Naehrig. *Constructive and computational aspects of cryptographic pairings*. PhD thesis, Eindhoven University of Technology, May 2009.

[51] 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.

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

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

[54] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, 2010.

[55] 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.

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

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

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

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

[60] F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, 2010.

[61] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. Cryptology ePrint 2014/674.

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

A Elliptic Curve details

We construct bilinear systems, $\mathcal{G}_{IN}$ and $\mathcal{G}_{OUT}$. Let $\mathcal{G}_{IN} = (p', \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathbb{F}_p)$: that is, $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are groups of prime order p', all defined over fields of large prime characteristic p, such that there exists an efficiently computable bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Analogously, define $\mathcal{G}_{OUT} = (p, \tilde{\mathbb{G}}_1, \tilde{\mathbb{G}}_2, \tilde{\mathbb{G}}_T, \tilde{e}, \mathbb{F}_{\tilde{p}})$. Bounded bootstrapping requires that the group order in the definition of $\mathcal{G}_{OUT}$ is equal to the prime characteristic in the definition of $\mathcal{G}_{IN}$.

Proof of Lemma 6: $E/\mathbb{F}_p$ is a BN curve [7], therefore $k = 12$. We construct $\tilde{E}$ by following the four steps in [29, Th. 4.1], with inputs the (parameterized) subgroup order p and the embedding degree $k = 6$. For Step 1, we always have $6 \mid p - 1$, and therefore $(-3/p) = 1$ as follows: when $p \equiv 1 \bmod 12$, $(-1/p) = 1$ from $p \equiv 1 \bmod 4$ and $(3/p) = 1$ from [39, §5, Theorem 2]; when $p \equiv 7 \bmod 12$, $(-1/p) = -1$ from $p \equiv 3 \bmod 4$ and $(3/p) = -1$ from [39, §5, Theorem 2]. For Step 2, a primitive 6th root of unity in $(\mathbb{Z}/p\mathbb{Z})^\times$ is parameterized as $\zeta_6(x) = -18x^3 - 18x^2 - 9x - 1$ [50, Lemma 2.10], so we can take $t'(x) = \zeta_6(x) + 1 = -18x^3 - 18x^2 - 9x$. For Step 3, $\sqrt{-3} \in (\mathbb{Z}/p\mathbb{Z})^\times$ is parameterized by $\sqrt{-3} = \pm(36x^3 + 36x^2 + 18x + 3)$, from which $y' = (t' - 2)/\sqrt{-3}$ depends on the sign in $\sqrt{-3}$; the choice of $\sqrt{-3} = 36x^3 + 36x^2 + 18x + 3$ gives $y' = 6x^3 + 6x^2 + 3x$. In Step 4, we are free to choose any $t \in \mathbb{Z}$ congruent to $t' \bmod p$, and likewise for $y \in \mathbb{Z}$ congruent to $y' \bmod p$, provided that $4 \mid (t^2 - 3y^2)$. Setting $t(x) = 2p(x) + t'(x)$ and $y(x) = 2p(x) + y'(x)$ into $\tilde{p}(x) = (t(x)^2 - 3y(x)^2)/4$ gives (2). Finally, to see the divisibility condition on $\#\tilde{E}$, we write the trace of Frobenius as $t(x) = 2p(x) + t'(x) = 72x^4 + 54x^3 + 30x^2 + 3x + 2$, from which it follows that $\#\tilde{E}(\mathbb{F}_{\tilde{p}(x)}) = \tilde{p}(x) + 1 - t(x) = 3 \cdot (48x^4 + 48x^3 + 33x^2 + 9x + 1) \cdot p(x)$. ∎

Taking $x = -(2^{62} + 2^{55} + 1)$ in Lemma 6 gives rise to p and p' as 254-bit primes, and $\tilde{p}$ as a 509-bit prime. The curve $E/\mathbb{F}_p\colon y^2 = x^3 + 2$ has prime order p' and the curve $\tilde{E}/\mathbb{F}_{\tilde{p}}\colon y^2 = x^3 + 21$ has composite order $h \cdot p$, where $h = 3 \cdot (48x^4 + 48x^3 + 33x^2 + 9x + 1)$. Both curves E and $\tilde{E}$ support a maximal degree-6 twist which offers several efficiency benefits: the twist of E is $E'/\mathbb{F}_{p^2}\colon y^2 = x^3 + (1 - u)$, $u^2 = -1$,

that of $\tilde{E}$ is $\tilde{E}'/\mathbb{F}_{\tilde{p}} : y^2 = x^3 + 63$. The groups in $\mathcal{G}_{IN}$ and $\mathcal{G}_{OUT}$ are defined as:

$$\mathbb{G}_1 = E(\mathbb{F}_p), \qquad \mathbb{G}_2 = E'(\mathbb{F}_{p^2})[p'], \quad \mathbb{G}_T = \mu_{p'} \subset \mathbb{F}_{p^{12}},$$

$$\tilde{\mathbb{G}}_1 = \tilde{E}(\mathbb{F}_{\tilde{p}})[p], \quad \tilde{\mathbb{G}}_2 = \tilde{E}'(\mathbb{F}_{\tilde{p}})[p], \qquad \tilde{\mathbb{G}}_T = \mu_p \subset \mathbb{F}_{\tilde{p}^6}. \quad (3)$$

where $E[\ell]$ denotes the ℓ-torsion elements on the curve E, and μ_ℓ denotes the ℓth roots of unity in a finite field. Note that the four elliptic curve groups, $\mathbb{G}_1$, $\mathbb{G}_2$, $\tilde{\mathbb{G}}_1$ and $\tilde{\mathbb{G}}_2$, have prime orders of 254 bits, and that $\mathbb{G}_T$ and $\tilde{\mathbb{G}}_T$ are contained in finite fields of over 3000 bits. Thus, all of the groups in (3) meet the respective requirements of the 128-bit security level [29, §1.1].

To complete the description of $\mathcal{G}_{IN}$ and $\mathcal{G}_{OUT}$, it remains to detail the respective pairings e and $\tilde{e}$. Since both E and $\tilde{E}$ are *ordinary* curves, both pairings are *asymmetric* and are instantiated using the optimal ate pairing [60]. The curve E is *the* BN curve used in several "speed record" papers (cf. [2]), so we refer there for a description of an optimized implementation of e. Lemma 7 below describes the pairing $\tilde{e}$.

The specific construction given here provides a way to construct a pair of curves with the desired properties depending on a single parameter from a parameterized family of pairs of curves. We note that, in general, the Cocks-Pinch approach can be applied to a specific desired group order, not necessarily in parameterized form. This means that starting from a given group order, one can use the Cocks-Pinch approach to construct a sequence of curves $E^{(i)}$, defined over prime fields $\mathbb{F}_{p_i}$, respectively, such that p_i divides $\#E^{(i+1)}(\mathbb{F}_{p_{i+1}})$.

The Cocks-Pinch algorithm implies that this method doubles the bit length of the base field prime for each layer, i.e., the bit length of p_{i+1} is roughly twice the bit length of p_i. Assuming an implementation with Karatsuba multiplication, one would expect the cost of base field arithmetic in $\mathbb{F}_{p_{i+1}}$ to be 3 times that in $\mathbb{F}_{p_i}$. Due to the large bit sizes, it is possible to reduce the embedding degree with each further step successively, possibly to the minimal value 1, and still maintain 128-bit security.

A pairing for $\mathcal{G}_{OUT}$ The following Lemma describes the pairing $\tilde{e}$ on the Cocks-Pinch curve $\tilde{E}$ in detail. As usual, let $f_{m,Q}$ be a function with divisor $(f_{m,Q}) = m(Q) - ([m]Q) - (m-1)(O)$, $\ell_{Q,R}$ be a function with divisor $(\ell_{Q,R}) = (Q) + (R) + (-(Q+R)) - 3(O)$, and v_Q be a function with divisor $(v_Q) = (v_Q) + (v_{-Q}) - 2(O)$.

Lemma 7 *Let $\tilde{E}/\mathbb{F}_{\tilde{p}}$ be a Cocks-Pinch curve parameterized as in Lemma 6. For $P \in \tilde{\mathbb{G}}_1$ and $Q \in \tilde{\mathbb{G}}_2$, the function*

$$\tilde{e}(Q,P) = \left(f_{6x^2+2x+1,Q}^{\tilde{p}}(P) \cdot f_{2x,Q}(P) \cdot \frac{\ell_{[(6x^2+2x+1)\tilde{p}]Q,[2x]Q}(P)}{v_{[(6x^2+2x+1)\tilde{p}+2x]Q}(P)} \right)^{(\tilde{p}^6-1)/p} \quad (4)$$

defines a bilinear pairing on $\tilde{\mathbb{G}}_2 \times \tilde{\mathbb{G}}_1$. Moreover, asides from the final exponentiation by $(\tilde{p}^6 - 1)/p$, the pairing can be computed using $\log_2(6x^2) + \varepsilon$ basic Miller iterations and (at most) an additional $2\log_2(3x)$ multiplications in $\mathbb{F}_{\tilde{p}^6}$.

Proof of Lemma 7: Recall that $\tilde{E}$ has embedding degree $\tilde{k} = 6$ with respect to the prime subgroup of order $p(x) = 36x^4 + 36x^3 + 24x^2 + 18x + 1$. Following [60, §3.3], we work with the 2-dimensional lattice $\mathcal{L}$, spanned by $\langle p(x), 0 \rangle$ and $\langle -\tilde{p}(x), 1 \rangle$. Equation (4) follows from applying [60, Eq. 5] to the short vector in $\langle 2x, 6x^2 + 2x + 1 \rangle$ in $\mathcal{L}$. To compute $\tilde{e}$, we note that the computation of $f_{6x^2+2x+1,Q}(P)$ can make use of $f_{2x,Q}(P)$, since (in terms of divisors) we have

$$(f_{6x^2,Q}) = (f_{2x,Q})^{3x} \cdot (f_{3x,[2x]Q}), \quad (5)$$

and

$$(f_{6x^2+2x+1,Q}) = \left(f_{6x^2,Q} \cdot f_{2x,Q} \cdot \frac{\ell_{[6x^2]Q,[2x]Q}}{v_{[6x^2+2x]Q}} \cdot \frac{\ell_{[6x^2+2x]Q,Q}}{v_{[6x^2+2x+1]Q}} \right). \quad (6)$$

For (5), it takes $\log_2(2x)$ basic Miller iterations to compute $f_{2x,Q}$ and another $\log_2(3x)$ iterations to subsequently compute $f_{3x,[2x]Q}$, giving $\log_2(6x^2)$ basic Miller iterations. The one additional iteration required for $f_{3,[2x^2]Q}$ is included in the ε term, along with the two "one-off" line functions in (6), and the other "one-off" multiplications throughout. Finally, the exponentiation by $3x$ in (5) incurs at most and additional $2\log_2(3x)$ multiplications. ∎