DOI:10.1145/2856449

Pinocchio: Nearly Practical Verifiable Computation

By Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova

Abstract

To instill greater confidence in computations outsourced to the cloud, clients should be able to *verify* the correctness of the results returned. To this end, we introduce Pinocchio, a built system for efficiently verifying general computations while relying only on cryptographic assumptions. With Pinocchio, the client creates a public evaluation key to describe her computation; this setup is proportional to evaluating the computation once. The worker then evaluates the computation on a particular input and uses the evaluation key to produce a proof of correctness. The proof is only 288 bytes, regardless of the computation performed or the size of the IO. Anyone can check the proof using a public verification key.

Crucially, our evaluation on seven applications demonstrates that Pinocchio is efficient in practice too. Pinocchio's verification time is a fixed 10 ms plus 0.4–15 µs per IO element: 5–7 orders of magnitude less than previous work[23]; indeed Pinocchio is the first general-purpose system to demonstrate verification cheaper than native execution (for some apps). The worker's proof effort is still expensive, but Pinocchio reduces it by 19×–60× relative to prior work. As an additional feature, Pinocchio allows the worker to include private inputs in the computation and prove that she performed the computation correctly without revealing any information about the private inputs to the client. Finally, to aid development, Pinocchio provides an end-to-end toolchain that compiles a subset of C into programs that implement the verifiable computation protocol.

1. INTRODUCTION

Since computational power is often asymmetric (particularly for mobile devices), a relatively weak client may wish to outsource computation to one or more powerful workers. For example, a scientist might want to run a protein folding simulation in the cloud or make use of volunteer distributed computing. In such settings, the client should be able to *verify* the results returned, to guard against malicious or malfunctioning workers. Even from a legitimate worker's perspective, verifiable results are beneficial, since they are likely to command a higher price. They also allow the worker to shed liability: any undesired outputs are provably the result of data the client supplied.

Considerable systems and theory research has looked at the problem of verifying computation (Section 6). However, most of this work has either been function specific, relied on assumptions we prefer to avoid, or simply failed to pass basic practicality requirements. Function specific solutions[13, 24] are often efficient, but only for a narrow class of computations. More general solutions often rely on assumptions that may not apply. For example, systems based on replication[5] assume uncorrelated failures, while those based on Trusted Computing[19] or other secure hardware[16] assume that physical protections cannot be defeated. Finally, the theory community has produced a number of beautiful, general-purpose protocols[1, 9, 12, 15] that offer compelling asymptotics. In practice however, because many rely on complex Probabilistically Checkable Proofs (PCPs)[1] or fully homomorphic encryption (FHE),[11] the performance is currently unacceptable—verifying small instances would take millions of years (Section 5.1). Recent work[7, 22, 23] has improved these protocols considerably, but efficiency is still problematic, and the protocols lack features like public verification. Without public verification, anyone who can verify a proof can also produce a cheating proof.

In contrast, Pinocchio is a concrete system for efficiently verifying general computations while making only cryptographic assumptions. In particular, Pinocchio supports public verifiable computation (VC),[9,20] which allows an untrusted worker to produce *signatures of computation*. Initially, the client chooses a function and generates a public evaluation key and a (small) public verification key. Given the evaluation key, a worker verifiably computes the function on an input and produces a proof (or signature) to accompany the result. Anyone (not just the client) can then use the verification key to check the correctness of the worker's result for the specific input used.

As an additional feature, Pinocchio supports zero-knowledge VC, in which the worker convinces the client that it knows one or more private inputs with a particular property, without revealing any information about the input. For example, Pantry[4] uses Pinocchio to compute Map-Reduce jobs (e.g., image matching) over private data (e.g., DMV photos) held by a server. Recent work also employs Pinocchio to anonymize Bitcoin transactions by proving, in zero knowledge, that the transactions do not create or destroy money.[2, 8]

Pinocchio's asymptotics are excellent: cryptographic operations required for key setup and proof generation are linear in the size of the original computation, and verification requires time linear in the size of the inputs and outputs. Even more surprising, Pinocchio's proof is constant sized, *regardless* of the computation performed. Crucially, our

The original version of this paper was published in the *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 238–252.

evaluation (Section 5) demonstrates that these asymptotics come with small constants.

Compared with previous work,[23] Pinocchio improves verification time by 5–7 *orders of magnitude* and requires less than 10 ms for applications with reasonably sized IO, enabling Pinocchio to beat native C execution for some apps. We also improve the worker's proof efforts by 19×–60× relative to prior work. The resulting proof is tiny, 288 bytes (only slightly more than an RSA-2048 signature), regardless of the computation. Making a proof zero-knowledge is also cheap, adding negligible overhead (213 μs to key generation and 0.1% to proof generation).

While these improvements are promising, additional progress is needed before the worker's proof overhead reaches true practicality. However, even now, this overhead may be acceptable in scenarios that require high assurance, or that need the zero-knowledge properties Pinocchio supports.

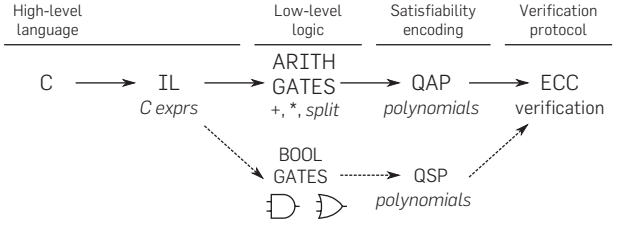
To achieve efficient VC, Pinocchio combines *quadratic programs*, a computational model introduced by Gennaro et al.,[10] with a series of theoretical refinements and systems engineering to produce an end-to-end toolchain for verifying computations. Specifically, via an improved protocol and proof technique relative to Gennaro et al., we slash the cost of key generation by 61%, and the cost of producing a proof by 64%. From a developer's perspective, Pinocchio provides a compiler that transforms C code into a circuit representation (we support both Boolean and arithmetic), converts the circuit into a quadratic program, and then generates programs to execute the cryptographic protocol (Figure 1).

Pinocchio's end-to-end toolchain allows us to implement real applications that benefit from verification. In particular, we implement two forms of matrix multiplication, multivariate polynomial evaluation, image matching, all-pairs shortest paths, a lattice-gas scientific simulator, and SHA-1. We find (Section 5) that the first three apps translate efficiently into arithmetic circuits, and hence Pinocchio can verify their results faster than native execution of the same program. The latter four apps translate less efficiently, due to their reliance on inequality comparisons and bitwise operations, and yet they may still be useful for zero-knowledge applications.

In summary, this paper contributes:

1. An end-to-end system for efficiently verifying computation performed by one or more untrusted workers. This includes a compiler that converts C code into a format suitable for verification, as well as a suite of tools for running the actual protocol.
2. Theoretical and systems-level improvements that bring performance down by 5–7 orders of magnitude relative to prior work,[23] and hence into the realm of plausibility.
3. An evaluation on seven real C apps, showing verification faster than 32-bit native integer execution for some apps.

Figure 1. Overview of Pinocchio's Toolchain. Pinocchio takes a high-level C program all the way through to a distributed set of executables that run the program in a verified fashion.



2. BACKGROUND

2.1. Verifiable computation (VC)

A public VC scheme allows a computationally limited client to outsource to a worker the evaluation of a function F on input u. The client can then verify the correctness of the returned result $F(u)$ while performing less work than required for the function evaluation.

More formally, we define public VC as follows, generalizing previous definitions.[9, 10, 20]

Definition 1 (Public Verifiable Computation). *A public verifiable computation scheme $\mathcal{VC}$ consists of a set of three polynomial-time algorithms* (KeyGen, Compute, Verify)*:*

- $(EK_F, VK_F) \leftarrow \mathsf{KeyGen}(F, 1^\lambda)$*: The randomized key generation algorithm takes the function F to be outsourced and security parameter λ; it outputs a public evaluation key EK_F, and a public verification key VK_F.*
- $(y, \pi_y) \leftarrow \mathsf{Compute}(EK_F, u)$*: The deterministic worker algorithm uses the public evaluation key EK_F and input u. It outputs $y \leftarrow F(u)$ and a proof π_y of y's correctness.*
- $\{0, 1\} \leftarrow \mathsf{Verify}(VK_F, u, y, \pi_y)$*: Given the verification key VK_F, the deterministic verification algorithm outputs 1 if $F(u) = y$, and 0 otherwise.*

Prior work gives formal definitions for correctness, security, and efficiency,[10] so we merely summarize:

- **Correctness**. For any function F, and any input u to F, if we generate keys for F, and run Compute with the resulting evaluation key EK_F, then Verify will always accept.
- **Security**. For any function F and any probabilistic polynomial-time adversary, the adversary cannot produce a proof for IO $\hat{u}, \hat{y}$ such that $F(\hat{u}) \neq \hat{y}$ but Verify accepts the proof.
- **Efficiency.** KeyGen is assumed to be a one-time operation whose cost is amortized over many calculations, but we require that Verify is cheaper than evaluating F.

Several previous VC schemes[9] were not public, but rather *designated verifier*, meaning that the verification key VK_F must be kept secret. Indeed, in these schemes, even revealing the output of the verification function (i.e., whether or not the worker had been caught cheating) could lead to attacks on the system. A public VC scheme avoids such issues.

Zero-Knowledge Verifiable Computation. We also consider

an extended setting where the outsourced computation is a function, $F(u, w)$, of two inputs: the client's input u and an auxiliary input w from the worker. A VC scheme is *zero-knowledge* if the client learns nothing about the worker's input beyond the output of the computation.

2.2. Quadratic programs

Gennaro, Gentry, Parno, and Raykova (GGPR) showed how to compactly encode computations as quadratic programs,[10] so as to obtain efficient VC and zero-knowledge VC schemes. Specifically, they show how to convert any arithmetic circuit into a comparably sized Quadratic Arithmetic Program (QAP).

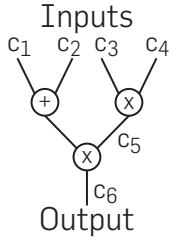
Standard results show that polynomially sized circuits are equivalent (up to a logarithmic factor) to Turing machines that run in polynomial time, though of course the actual efficiency of computing via circuits versus on native hardware depends heavily on the application; for example, an arithmetic circuit for matrix multiplication adds essentially no overhead, whereas a Boolean circuit for integer multiplication is far less efficient than executing a single 32-bit assembly instruction.

An arithmetic circuit consists of wires that carry values from a field $\mathbb{F}$ and connect to addition and multiplication gates—see Figure 2 for an example.

Before formally defining QAPs, we walk through the steps for encoding the circuit in Figure 2 into an equivalent QAP. First, we select two arbitrary values, r_5, $r_6 \in \mathbb{F}$ to represent the two multiplication gates (the addition gates will be compressed into their contributions to the multiplication gates). We define three sets of polynomials $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$ by letting the polynomials in $\mathcal{V}$ encode the left input into each multiplication gate, the $\mathcal{W}$ encode the right input into each gate, and the $\mathcal{Y}$ encode the outputs. Thus, for the circuit in Figure 2, we define six polynomials for each set $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$, four for the input wires, and two for the outputs from the multiplication gates. We define these polynomials based on each wire's contributions to the multiplication gates. Specifically all of the $v_k(r_5) = 0$, except $v_3(r_5) = 1$, since the third input wire contributes to the left input of c_5's multiplication gate. Similarly, $v_k(r_6) = 0$, except for $v_1(r_6) = v_2(r_6) = 1$, since the first two inputs both contribute to the left input of c_6's gate. For $\mathcal{W}$, we look at right inputs. Finally, $\mathcal{Y}$ represents outputs; none of the input wires is an output, so $y_k(r_5) = y_k(r_6) = 0$ for $k \in \{1, \ldots, 4\}$, and $y_5(r_5) = y_6(r_6) = 1$. As we explain below, we can use this encoding of the circuit to efficiently check that it was evaluated correctly.

More generally, we define a QAP, an encoding of an arithmetic circuit, as follows.

Figure 2. Arithmetic Circuit and Equivalent QAP. Each wire value comes from, and all operations are performed over, a field $\mathbb{F}$. The polynomials in the QAP are defined in terms of their evaluations at the two roots, r_5 and r_6. See text for details.



	(r_5, r_6)		(r_5, r_6)		(r_5, r_6)
$v_1(r_i)$	(0,1)	$w_1(r_i)$	(0,0)	$y_1(r_i)$	(0,0)
$v_2(r_i)$	(0,1)	$w_2(r_i)$	(0,0)	$y_2(r_i)$	(0,0)
$v_3(r_i)$	(1,0)	$w_3(r_i)$	(0,0)	$y_3(r_i)$	(0,0)
$v_4(r_i)$	(0,0)	$w_4(r_i)$	(1,0)	$y_4(r_i)$	(0,0)
$v_5(r_i)$	(0,0)	$w_5(r_i)$	(0,1)	$y_5(r_i)$	(1,0)
$v_6(r_i)$	(0,0)	$w_6(r_i)$	(0,0)	$y_6(r_i)$	(0,1)

$t(x) = (x - r_5)(x - r_6)$

Definition 2 (Quadratic Arithmetic Program (QAP)[10]). *A QAP Q over field $\mathbb{F}$ contains three sets of $m + 1$ polynomials $\mathcal{V} = \{v_k(x)\}, \mathcal{W} = \{w_k(x)\}, \mathcal{Y} = \{y_k(x)\}$, for $k \in \{0 \ldots m\}$, and a target polynomial $t(x)$. Suppose F is a function that takes as input n elements of $\mathbb{F}$ and outputs n' elements, for a total of $N = n + n'$ I/O elements. Then we say that Q computes F if: $(c_1, \ldots, c_N) \in \mathbb{F}^N$ is a valid assignment of F's inputs and outputs,* if and only if *there exist coefficients $(c_{N+1}, \ldots, c_m)$ such that $t(x)$ divides $p(x)$, where:*

$$p(x) = \left(v_0(x) + \sum_{k=1}^{m} c_k \cdot v_k(x)\right) \cdot \left(w_0(x) + \sum_{k=1}^{m} c_k \cdot w_k(x)\right) - \left(y_0(x) + \sum_{k=1}^{m} c_k \cdot y_k(x)\right). \quad (1)$$

In other words, there must exist some polynomial $h(x)$ such that $h(x) \cdot t(x) = p(x)$. The size of Q is m, and the degree is the degree of $t(x)$.

Building a QAP Q for a general arithmetic circuit C is fairly straightforward. We pick an arbitrary root $r_g \in \mathbb{F}$ for each multiplication gate g in C and define the target polynomial to be $t(x) = \Pi_g(x - r_g)$. We associate an index $k \in [m] = \{1 \ldots m\}$ to each input of the circuit and to each output from a multiplication gate. Finally, we define the polynomials in $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$ by letting the polynomials in $\mathcal{V}$ encode the left input into each gate, the $\mathcal{W}$ encode the right input into each gate, and the $\mathcal{Y}$ encode the outputs. For example, $v_k(r_g) = 1$ if the kth wire is a left input to gate g, and $v_k(r_g) = 0$ otherwise. Similarly, $y_k(r_g) = 1$ if the kth wire is the output of gate g, and $y_k(r_g) = 0$ otherwise. Thus, if we consider a particular gate g and its root r_g, Equation (1) simplifies to: $\left(\sum_{k=1}^{m} c_k \cdot v_k(r_g)\right) \cdot \left(\sum_{k=1}^{m} c_k \cdot w_k(r_g)\right) = \left(\sum_{k \in I_{left}} c_k\right) \cdot \left(\sum_{k \in I_{right}} c_k\right) = c_g y_k(r_g) = c_g$, which just says that the output value of the gate is equal to the product of its inputs, the very definition of a multiplication gate. For example, in the QAP for the circuit in Figure 2, if we evaluate $p(x)$ at r_5, we get $(c_3) \cdot (c_4) = c_5$, which directly encodes the first multiplication gate, and similarly, at r_6, $p(x)$ simplifies to $(c_1 + c_2) \cdot (c_5) = c_6$, that is, an encoding of the second multiplication gate.

In short, the divisibility check that $t(x)$ divides $p(x)$ decomposes into $\deg(t(x))$ separate checks, one for each gate g and root r_g of $t(x)$, that $p(r_g) = 0$.

The actual construction[10] is a bit more complex, as it handles addition and multiplication by constants. Nonetheless, GGPR show that for any arithmetic circuit with d multiplication gates and N I/O elements, one can construct an equivalent QAP with degree (the number of roots r_g) d and size (number of polynomials in each set) $d + N$. Note that addition gates and multiplication-by-constant gates do not contribute to the size or degree of the QAP. Thus, these gates are essentially "free" in QAP-based VC schemes.

Strong QAPs. In their QAP-based VC scheme, described below, GGPR unfortunately require a strong property

from the QAP. Note that Definition 2 only considers the case where the same set of coefficients c_i are applied to all three sets of polynomials. GGPR additionally require the if-and-only-if condition in Definition 2 to hold even when different coefficients a_i, b_i, c_i are applied—that is, when $p(x) = \left(\sum_{k=1}^{m} c_k \cdot v_k(x)\right) \cdot \left(\sum_{k=1}^{m} b_k \cdot w_k(x)\right) - \left(\sum_{k=1}^{m} a_k \cdot y_k(x)\right)$. They show how to convert any QAP into a *strong QAP* that satisfies this stronger condition. Unfortunately, this strengthening step increases the QAP's degree to $3d + 2N$, more than *tripling* it. This in turn, more than triples the cost of key generation, the size of the evaluation key, and the worker's effort to produce a proof.

2.3. Building VC from quadratic programs

To construct a VC protocol from a quadratic program, we map each polynomial—for example, $v_k(x)$—of the quadratic program to an element $g^{v_k(s)}$ in an elliptic curve group $\mathbb{G}$, where s is a secret value selected by the client, and g is a generator of $\mathbb{G}$. These group elements are given to the worker. For a given input, the worker evaluates the circuit directly to obtain the output and the values of the internal circuit wires. These values correspond to the coefficients c_i of the quadratic program. Thus, the VC worker can evaluate $v(s) = \sum_{k \in [m]} c_k \cdot v_k(s)$ "in the exponent" to get $g^{v(s)}$; it computes $w(s)$ and $y(s)$, in the exponent, similarly.

To allow the worker to prove that Equation (1) holds, we also, as part of the evaluation key, give the worker $g^{(s^i)}$ terms. The worker computes $h(x) = p(x)/t(x) = \sum_{i=0}^{d} h_i \cdot x^i$, and then uses the h_i, along with $g^{(s^i)}$ terms, to compute $g^{h(s)}$. To oversimplify, the proof consists of $(g^{v(s)}, g^{w(s)}, g^{y(s)}, g^{h(s)})$. To check that $p(s) = h(s)t(s)$, the verifier uses a *bilinear map* that allows him to take two elliptic curve elements and "multiply" their exponents together to create an element in a new group. The actual protocol[10] is a bit more complex, because additional machinery is needed to ensure that the worker incorporates the client's input u correctly, and that the worker indeed generates (say) $v(s)$ in the exponent as some linear function of the $v_k(s)$ values.

Regarding efficiency, GGPR[10] show that the one-time setup of KeyGen runs in time linear in the original circuit size, $O(|C|)$. The worker performs $O(|C|)$ cryptographic work, but he must also perform $O(|C|\log^2|C|)$ non-cryptographic work to calculate $h(x)$. To achieve this performance, the worker exploits the fact that the evaluation vectors $(v_k(r_1), \ldots, v_k(r_d))$ are all very sparse (also for the w and y polynomials). The proof itself is constant size, with only 9 group elements for QAPs, though the verifier's work is still linear, $O(N)$, in the size of the inputs and outputs of the function.

In terms of security, GGPR[10] show this VC scheme is sound under the d-PKE and q-PDH assumptions, which are weak versions of assumptions in prior work.

Zero Knowledge. Making the VC scheme zero-knowledge is remarkably simple. One simply includes the target polynomial $t(x)$ itself in the polynomial sets $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$. This allows the worker to "randomize" its proof by adding $\delta_v t(s)$ in the exponent to $v_{mid}(s)$, $\delta_w t(s)$ to $w(s)$, and $\delta_y t(s)$ to $y(s)$ for random δ_v, δ_w, δ_y, and modifying the other elements of the proof accordingly. The modified value of $p(x)$ remains divisible by $t(x)$, but the randomization makes the scheme statistically zero-knowledge.[10]

3. THEORETICAL REFINEMENTS

In this section, we improve GGPR's protocol[10] to significantly reduce key generation time, evaluation key size, and worker effort. We analyze our improvements empirically in Section 5.

Our main optimization is that we construct a VC scheme that uses a *regular* QAP (as in Definition 2), rather than a *strong* QAP. Recall that GGPR show how to transform a regular QAP into a strong QAP, but the transformation more than *triples* the degree of the QAP. Consequently, when they plug their strong QAP into their VC construction, the strengthening step more than triples the key generation time, evaluation key size, and worker computation. We take a different approach that uses a regular QAP, and hence we do not need a strengthening step at all. Instead, we embed additional structure into our new VC proof that ensures that the worker uses the same linear combination to construct the v, w, and y terms of its proof.[a] Surprisingly, this additional structure comes at no cost, and our VC scheme is actually *less* complicated than GGPR's! Finally, we expand the expressivity and efficiency of the functions QAPs can compute by designing a number of custom circuit gates for specialized functions.

3.1. Our new VC protocol

Next we describe our more efficient VC scheme, with some remarks afterwards on some its properties.

Protocol 1 (Verifiable Computation from regular QAPs).

- $(EK_F, VK_F) \leftarrow \mathsf{KeyGen}(F, 1^\lambda)$: *Let F be a function with N input/output values from $\mathbb{F}$. Convert F into an arithmetic circuit C; then build the corresponding QAP $Q = (t(x), \mathcal{V}, \mathcal{W}, \mathcal{Y})$ of size m and degree d. Let $I_{mid} = \{N+1, \ldots, m\}$, that is, the non-IO-related indices.*
  Let e be a non-trivial bilinear map $e: \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, and let g be a generator of G.
  Choose r_v, r_w, s, α_v, α_w, α_y, β, γ at random from $\mathbb{F}$ and set $r_y = r_v \cdot r_w$, $g_v = g^{r_v}$, $g_w = g^{r_w}$ and $g_y = g^{r_y}$.
  Construct the public evaluation key EK_F as:

$$\Big(\{g_v^{v_k(S)}\}k \in I_{mid}, \quad \{g_w^{w_k(S)}\}k \in I_{mid}, \quad \{g_y^{y_k(S)}\}k \in I_{mid}, \quad \{g_v^{\alpha_v v_k(S)}\}k \in I_{mid}, \{g_w^{\alpha_w w_k(S)}\}k \in I_{mid}, \{g_y^{\alpha_y y_k(S)}\}k \in I_{mid}, \quad \{g^{s^i}\}i \in [d], \quad \{g_v^{\beta v_k(S)} g_w^{\beta w_k(S)} g_y^{\beta y_k(S))}\}k \in I_{mid} \Big),$$

  and the public verification key as: $VK_F = (g^1, g^{\alpha_v}, g^{\alpha_w}, g^{\alpha_y}, g^{\gamma}, g^{\beta\gamma}, g_y^{t(s)}, \{g_v^{v_k(s)}, g_w^{w_k(s)}, g_y^{y_k(s)}\}k \in \{0\} \cup [N])$.
- $(y, \pi_y) \leftarrow \mathsf{Compute}(EK_F, u)$: *On input u, the worker evaluates the circuit for F to obtain $y \leftarrow F(u)$; he also learns the values*

[a] Our proof contains a term that enforces this linear constraint without increasing the degree. GGPR's generic strengthening step checked the consistency of the linear combinations via additional multiplication gates, which increased the degree of the QAP.

$\{c_i\}_{i\in[m]}$ *of the circuit's wires.*
He solves for $h(x)$ *(the polynomial such that* $p(x) = h(x)\cdot t(x)$*), and computes the proof* π_y *as:*

$$\Big(g_v^{v_{mid}(S)},\quad g_w^{w_{mid}(S)},\quad g_y^{y_{mid}(S)},\quad g^{h(S)},\quad g_v^{\alpha_v v_{mid}(S)},\quad g_w^{\alpha_w w_{mid}(S)},\quad g_y^{\alpha_y y_{mid}(S)},\quad g_v^{\beta_v v_{mid}(S)}\ g_w^{\beta w_{mid}(S)}\ g_y^{\beta y_{mid}(S)} \Big),$$

where $v_{mid}(x) = \Sigma_{k\in I_{mid}} c_k \cdot v_k(x)$*, and similarly for* $w_{mid}(s)$ *and* $y_{mid}(s)$*. Since these are linear equations, he can compute them "in the exponent" using the material in the evaluation key, for example,* $g^{v_m id(s)} = \prod_{k\in I_{mid}} (g^{v_k(s)})^{c_k}$.

- $\{0,1\} \leftarrow \mathsf{Verify}(VK_F, u, y, \pi_y)$*: The verification of an alleged proof with elements* $g^{V_{mid}}, g^{W_{mid}}, g^{Y_{mid}}, g^H, g^{V'_{mid}}, g^{W'_{mid}}, g^{Y'_{mid}}$*, and* g^z *uses the public verification key* VK_F *and the pairing function* e *for the following checks.*
  - *Divisibility check for the QAP: using elements from* VK_F*, compute a term representing the I/O,* u *and* y*, by representing them as coefficients* $c_1, \ldots, c_N \in \mathbb{F}$ *and computing:* $g_v^{v_{io}(s)} = \prod_{k\in[N]} \left(g_v^{v_k(s)}\right)^{c_k}$ *(and similarly for* $g_w^{w_{io}(s)}$ *and* $g_y^{y_{io}(s)}$*). Check:*

$$e\left(g_v^{v_0(s)} g_v^{v_{io}(s)} g_v^{V_{mid}}, g_w^{w_0(s)} g_w^{w_{io}(s)} g_w^{W_{mid}}\right) = \quad (2)$$

$$e\left(g_y^{t(s)}, g^H\right) e\left(g_y^{y_0(s)} g_y^{y_{io}(s)} g_y^{Y_{mid}}, g\right). \quad (3)$$

  - *Check that the linear combinations computed over* $\mathcal{V}$*,* $\mathcal{W}$*, and* $\mathcal{Y}$ *are in their appropriate spans:*

$$e\left(g_v^{V'_{mid}}, g\right) = e\left(g_v^{V_{mid}}, g^{\alpha_v}\right), \quad e\left(g_w^{W'_{mid}}, g\right) = e\left(g_W^{W_{mid}}, g^{\alpha_W}\right),$$
$$e\left(g_y^{Y'_{mid}}, g\right) = e\left(g_y^{Y_{mid}}, g^{\alpha_y}\right).$$

  - *Check that the same coefficients were used in each of the linear combinations over* $\mathcal{V}$*,* $\mathcal{W}$*, and* $\mathcal{Y}$*:*

$$e\left(g^Z, g^\gamma\right) = e\left(g_v^{V_{mid}} g_w^{W_{mid}} g_y^{Y_{mid}}, g^{\beta\gamma}\right).$$

In a designated verifier setting (where the verifier knows s*,* α*, etc.), pairings are only needed for divisibility check, and the I/O term can be computed directly over* $\mathbb{F}$*, rather than "in the exponent."*

The correctness of the VC scheme follows from the properties of the QAP. Regarding security, we have the following:

THEOREM 1. *Let* d *be an upper bound on the degree of the QAP used in the VC scheme, and let* $q = 4d + 4$*. The VC scheme is sound under the d-PKE, q-PDH, and 2q-SDH assumptions.*

The proof of Theorem 1 is in the full version of the paper.

Security Intuition. As intuition for why the VC scheme is sound, note that it seems hard for an adversary who does not know α to construct *any* pair of group elements h, h^α *except* in the obvious way: by taking pairs $(g_1, g_1^\alpha), (g_2, g_2^\alpha), \ldots$ that he is given, and applying the same linear combination (in the exponent) to the left and right elements of the pairs. This hardness is formalized in the d-PKE assumption, a sort of "knowledge-of-exponent" assumption, that says that the adversary must "know" such a linear combination, in the sense that this linear combination can be extracted from him. Roughly, this means that, in the security proof, we can extract polynomials $V_{mid}(x), W_{mid}(x), Y_{mid}(x)$ such that V_{mid} (from the proof) equals $V_{mid}(s)$, $W_{mid} = W_{mid}(s)$ and $Y_{mid} = Y_{mid}(s)$, and that moreover these polynomials are in the linear spans of the $v_k(x)$'s, $w_k(x)$'s, and $y_k(x)$'s, respectively. If the adversary manages to provide a proof of a false statement that verifies, then these polynomials must not actually correspond to a QAP solution. So, either $p(x)$ is not actually divisible by $t(x)$ (in this case we break $2q$-SDH) or $V(x) = v_{io}(x) + V_{mid}(x)$, $W(x)$ and $Y(x)$ do not use the same linear combination (in this case we break q-PDH because in the proof we choose β in a clever way).

Zero Knowledge. We can apply GGPR's rerandomization technique[10] (Section 2.3) to provide statistical zero-knowledge for our new VC construction. The worker chooses $\delta_v, \delta_w, \delta_y \xleftarrow{R} \mathbb{F}$ and in his proof, instead of the polynomials $v_{mid}(x)$, $v(x)$, $w(x)$, and $y(x)$, he uses the following randomized versions $v_{mid}(x) + \delta_v t(x)$, $v(x) + \delta_v t(x)$, $w(x) + \delta_w t(x)$, and $y(x) + \delta_y t(x)$.

Performance. Our main improvement is that our VC scheme only requires a regular QAP, rather than a strong QAP, which improves performance by more than a factor of 3. Moreover, the scheme itself is simpler, leading to fewer group elements in the keys and proof, fewer bilinear maps for Verify, etc.

3.2. Expressive circuit constructions

The QAP that we use in our VC scheme is defined over $\mathbb{F}_p$, where p is a large prime. We can, as explained previously, derive a QAP over $\mathbb{F}_p$ that efficiently computes any function F that can be expressed in terms of addition and multiplication modulo p. This provides no obvious way to express some operations, such as $a \geq b$ using mod-p arithmetic. On the other hand, given a and b as bits, comparison is easy. Hence, one might infer that Boolean circuits are more general.

However, we design an arithmetic *split gate* to translate an arithmetic wire $a \in \mathbb{F}_p$, known to be in $[0, 2^k - 1]$, into k binary output wires. Given such binary values, we can compute Boolean functions using arithmetic gates: NAND$(a, b) = 1 - ab$, AND$(a, b) = ab$, OR$(a, b) = 1 - (1 - a)(1 - b)$. Each embedded Boolean gate costs only one multiply.

Surprisingly, this arithmetic embedding gives a fairly efficient VC scheme. Embedding introduces an expensive initial gate that constrains each input to $\{0, 1\}$, but henceforth, each embedded gate preserves the $\{0, 1\}$ invariant, adding only 1 to the degree and size of the QAP. Furthermore, the expression $\sum_{i=1}^{k} 2^{i-1} a_i$ combines a bitwise representation of a back into a single wire. Because the sum consists of additions and multiplications by constants, recombination is free; it doesn't increase the size of the QAP.

In our full paper, we also design a gate that enforces equality between two wires and a gate that checks whether a wire is equal to zero. These can be composed (Thm 11 in Ref.[10]) with other gates.

4. IMPLEMENTATION

We implemented a compiler that takes a subset of C to an equivalent arithmetic circuit (Section 4.1). Our VC suite

then compiles the circuit representation to the equivalent QAP, and generates code to run the VC protocol, including key generation, proof computation, and proof verification (Section 4.2). The toolchain compiles a large collection of applications and runs them with verification (Section 4.3). Source code for the toolchain is available.[b]

4.1. Compiler toolchain

The applications described below (Section 4.3) and evaluated in Section 5 are each compiled using *qcc*, our C-to-arithmetic-expression compiler, a 3525-line Python program. They are also compiled with `gcc` to produce the Native timings in Figures 5 and 6.

The compiler understands a substantial subset of C, including global, function, and block-scoped variables; arrays, structs, and pointers; function calls, conditionals, loops; and static initializers (Figure 3). It also understands arithmetic and bitwise Boolean operators and preprocessor syntax.

Since the "target machine" (arithmetic circuits) supports only expressions, not mutable state and iteration, we restrict the C program's semantics accordingly. For example, pointers and array dereferences must be compile-time constants; otherwise, each dynamic reference would produce conditional expressions of size proportional to the addressable memory. Function calls are inlined, while preserving C variable scope and pointer semantics.

Imperative conditionals compile to conditional expressions that encode the imperative side effects. Static conditions are collapsed at compile time. Similarly, loops with statically computable termination conditions are automatically unrolled completely.

The only scalar type presently supported is **int**; a compiler flag selects the integer size. The compiler inserts masking expressions to ensure that a *k*-bit int behaves exactly as the corresponding C type, including overflow.

The compiler's intermediate language is a set of expressions of C-like operators, such as `+`, `*`, `<=`, `?:`, `&`, and `^`.

The compiler back-end expands each expression into the arithmetic gate language of mul, add, const-mul, wire-split, etc., eliminating common subexpressions. It carefully bounds the bit-width of each wire value:

- inputs have the compiler-specified **int** width;
- each constant has a known width (e.g., $13 = 1101_2$ has bit width 4);
- a bitwise op produces the *max* of its arguments' widths;
- add can produce *max* + 1 bits (for a carry); and
- mul can produce $2 \cdot max$ bits.

When the width nears the available bits in the field (254), the compiler generates a split gate to truncate the value back to the specified **int** width. Tracking bit width minimizes the cost of split gates.

[b] https://vc.codeplex.com.

Figure 3. Fixed-Matrix Multiplication. The `qcc` compiler unrolls the loops and decodes the struct and array references to generate an arithmetic expression for `Out` in terms of `In`.

```
int mat[SIZE*SIZE] = { 0x12, ... };
struct In { int vector[SIZE]; };
struct Out { int result[SIZE]; };

void compute(struct In *in, struct Out *out){
  int i, j, k, t;
  for (i=0; i<SIZE; i+=1) {
    int t=0;
    for (k=0; k<SIZE; k+=1) {
      t = t + mat->[i*SIZE+k] * in->vector[k];
    }
    out->result[i] = t;
  }
}
```

4.2. Quadratic programs and cryptographic protocol

The next pipeline stage accepts a Boolean or arithmetic circuit and builds a QSP or QAP (Section 2). Then, per Section 3.1, it compiles the quadratic program into a set of cryptographic routines for the client (key generation and verification) and the worker (computation and proof generation). For comparison, we also implement the original GGPR[10]; GGPR protocol Section 5 shows that Pinocchio's enhancements reduce overhead by 18–64%.

The key-generation routine runs at the client, with selectable public verification and zero-knowledge features (Section 5.2). The code transmits the evaluation key over the network to the worker; to save bandwidth, the program transmits as C and the worker compiles it locally.

The computation routine runs at the server, collecting input from the client, using the evaluation key to produce the proof, and transmitting the proof back to the client (or, if desired, a different verifier). The verification routine uses the verification key and proof to determine if the worker cheated.

Our cryptographic code is single-threaded, but each stage is embarrassingly parallel. Prior work[23] shows that standard techniques can parallelize work across cores, machines, or GPUs. For the cryptographic code, we use a high-speed elliptic curve library[18] with a 256-bit BN-curve that provides 128 bits of security. The quadratic-program-construction and protocol-execution code is 10,832 lines of C and C++.

Faster Exponentiation. Generating the evaluation key *EK* requires exponentiating the same base *g* to many different powers. We optimize this operation by adapting Pippenger's multi-exponential algorithm for use with a single base. Essentially this means that we build a table of intermediate powers of *g*, allowing us to compute any particular exponent with only a few multiplications.

In a similar vein, the worker's largest source of overhead is applying the coefficients from the circuit "in the exponent" to compute $g^{v(s)}$, etc. We optimize this operation via a sliding-window technique to build a small table of powers for each pair of bases. In practice, these tables can improve performance by a factor of three to four, even counting the time to build the tables in the first place.

Polynomial Asymptotics. To generate a proof, the worker

must compute the polynomial $h(x)$ such that $t(x) \cdot h(x) = P(x)$ (Section 1). Since we store $P(x)$ in terms of its evaluations at the roots of the quadratic program (recall Figure 2), the worker must first interpolate to find $P(x)$ and then perform a polynomial division to arrive at $h(x)$.

Note that all of these computations take place in a normal field, whereas all of the worker's other steps involve cryptographic operations, which are about three orders of magnitude more expensive.

Thus, one might naïvely conclude, as we did, that simple polynomial algorithms, such as Lagrangian interpolation and "high-school" polynomial multiplication, suffice. However, we quickly discovered that the $O(n^2)$ behavior of these algorithms, at the scale required for verifiable computing, dwarfed the linear number of cryptographic operations (Section 5). Hence we implemented an FFT-based $O(n\log n)$ polynomial multiplication library and used a polynomial interpolation algorithm that builds a binary tree of polynomials, giving total time $O(n\log^2 n)$. Even so optimized, solving for $h(x)$ is the second largest source of worker overhead.

Preparing for the Future; Learning from the Past. In our implementation and evaluation, we assume a worst case scenario in which the client decides, without any warning, to outsource a new function, and similarly that the worker only ever computes a single instance for a given client. In practice, neither scenario is plausible. When the client first installs Pinocchio, the program, could build the single base exponent table discussed earlier. Further, it can choose a random s and begins computing powers of s in the background, since these are entirely independent of the computation. The worker can optimize similarly, given the client's key.

4.3. Applications

Pinocchio runs several applications; each can be instantiated with some static *parameters*, and then each instance can be executed with dynamic *inputs*. While it may be possible to use custom verification checks for some of these applications (e.g., matrix multiplication), we include them to illustrate their performance within a general-purpose system like Pinocchio.

Fixed Matrix multiplies an $n \times n$ matrix parameter M by an n-length input vector A, and outputs the resulting n-length vector $M \cdot A$. We choose five parameter settings that range from $|M| = 200 \times 200$ to $|M| = 1000 \times 1000$.

Two Matrices has parameter n, takes as input two $n \times n$ matrices M_1 and M_2, and outputs the $n \times n$ matrix $M_1 \cdot M_2$. Matrix operations are widely used, for example, in collaborative filtering ($|M| = 30 \times 30$ to $|M| = 110 \times 110$).

MultiVar Poly evaluates a k-variable, m-degree multivariate polynomial. The $(m+1)^k$ coefficients are parameters, the k variables $x_1, \ldots, x_k$ are the inputs, and the polynomial's scalar value is the output ($k = 5$, $m = 6$, 16,807 coeff. to $k = 5$, $m = 10$; 644,170 coeff.).

Image Matching is parameterized by an $i_w \times i_h$ rectangular image and parameters k_w, k_h. It takes as input a $k_w \times k_h$ image kernel, and outputs the minimum difference and the point (x, y) in the image where it occurs ($i_w \times i_h = 25$, $k_w \times k_h = 9$ to $i_w \times i_h = 2025$, $k_w \times k_h = 9$).

Shortest Paths implements the Floyd-Warshall $O(n^3)$ graph algorithm, useful for network routing and matrix inversion. Its parameter n specifies the number of vertices, its input is an $n \times n$ edge matrix, and its output is an $n \times n$ matrix of all-pairs shortest paths ($n = 8$, $e = 64$ to $n = 24$, $e = 576$).

LGCA is a Lattice-Gas Cellular Automata implementation that converges to Navier-Stokes. It has parameter n, the fluid lattice size, and k, the iteration count. It inputs one n-cell lattice and outputs another reflecting k steps ($n = 294$, $k = 5$ to $n = 294$, $k = 40$).

SHA-1 has no parameters. Its input is a 13-word (416-bit) input string, and it outputs its 5-word (160-bit) SHA-1 hash.

5. EVALUATION

We experiment on a Lenovo X201 ThinkPad. We run on a single core of a 2.67 GHz Intel Core i7 with 8 GB of RAM.

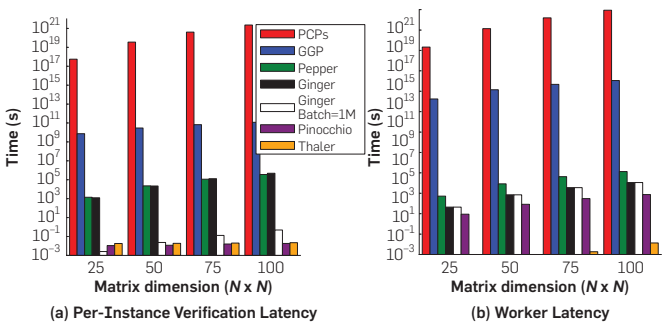
Below, we focus on comparisons with previous work and app-level performance. In the full paper, we present microbenchmarks to quantify the basic cost units of our protocol. Our results show that the optimizations described in Section 4.2.1 reduce costs by 2–3 orders of magnitude for polynomial operations, and factors of 3–10 for exponentiations. At the macro level, relative to the original GGPR protocol, KeyGen and Compute are more than twice as fast, and even verification is 24% faster. Pinocchio also drastically reduces the size of the evaluation key and even manages to reduce the size of GGPR's already svelte 9 element proof to 8 elements.

5.1. Comparison with related work

Figure 4 plots Pinocchio's performance against that of related systems. We use the multiplication of two matrices as our test application since it has appeared in several prior papers, though simpler, non-cryptographic verification procedures exist. Since all of these prior schemes are designated verifier, we measure against Pinocchio's designated verifier mode.

We compare against (1) a naïve version of a PCP-based scheme[22]; (2) GGP,[9] an early scheme that defined VC, but which relies on FHE; (3) Pepper,[22] an optimized refinement of (1); (4) Ginger,[23] a further refinement of Pepper; (5) Ginger with a batch of one million simultaneous

Figure 4. Performance Relative to Related Schemes. Pinocchio reduces costs by orders of magnitude (note the log scale on the y-axis). We graph the time necessary to (a) verify and (b) produce a proof result for multiplying two $N \times N$ matrices.



instances (see below); and (6) a subsequent system by Thaler,[25] tailored specifically for matrix multiplication and extending work based on interactive protocols.[7, 12] See Section 6 for more details on these schemes and the tradeoffs between them. Since most of these schemes are ridiculously impractical, we model, rather than measure, their performance. For GGP, we built a model of its performance based on recent performance results for FHE; for Thaler, we extrapolated from reported results[25]; while for the others, we used previously published models.[22, 23] For Pinocchio, however, we use real numbers from our implementation.

Figure 4 shows that Pinocchio continues the recent trend of reducing costs by orders of magnitude. A naive PCP-based scheme requires trillions of years to produce or verify a single proof. The FHE-based GGP protocol improves this performance significantly but remains impractical. Pepper and Ginger have made huge improvements over prior work, but, as we discuss in more detail in Section 6, they do not offer public verification or zero knowledge.

In addition to offering new properties, Pinocchio significantly improves performance and security. Except for Thaler's work, the systems shown in Figure 4 amortize setup work across many work instances,[c] but the characteristics of the amortization differ. To reach a break-even point, where the client does less work verifying than performing the work locally, Pepper and Ginger must batch work instances, whereas GGP and Pinocchio must perform enough instances to amortize key setup costs. These approaches have very different effects on latency. A client cannot benefit from Pepper or Ginger until it has accumulated an entire batch of instances. In Pinocchio, key setup can be precomputed, and henceforth every instance (including the first one) enjoys a better-than-break-even latency. Figure 4 shows the minimum latency achievable by each system. Compared with Ginger for a single instance, Pinocchio's verifier is ~120,000×–17,000,000× faster, and the worker is 19×–60× faster. To improve performance, Ginger's parameters are chosen such that the probability that the adversary can successfully cheat can be as high as $\frac{1}{2^{20}}$,(Figure 2 in Ref.[23]) while in Pinocchio, the probability is roughly $\frac{1}{2^{128}}$.

Finally, Pinocchio's verification is more efficient than Thaler's custom protocol, but Thaler's protocol is the only one to achieve practicality for the worker, showing the price the other systems pay for generality.

5.2. End-to-end application performance

We measure Pinocchio's performance for the applications and parameter settings described in Section 4.3. All applications are written in C and compile to both QAPs and to native executables. We measure performance using 32-bit input values, so we can compare against the native C version. This obviously makes things more challenging for Pinocchio, since Pinocchio operates over a 254-bit field using multi-precision integers, whereas the local execution uses the CPU's native 32-bit operations.

Figure 5. Cost of Verification versus Local. Verification must be cheaper than native execution for outsourcing to make sense, though for applications that want zero-knowledge, more expensive verification may be acceptable. All apps trend in the right direction, and three apps cross the plane where verification is cheaper than native. Error bars, often too small to see, represent 95% confidence intervals ($N = 50$, $\sigma \leq 2\%$).

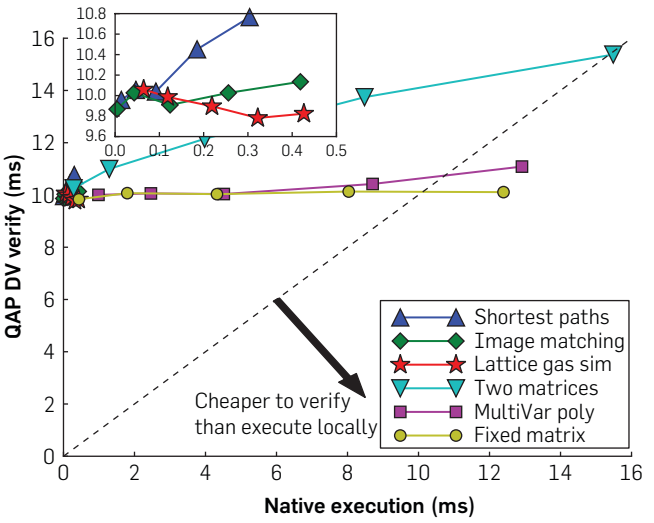


Figure 5 plots Pinocchio's verification time against the time to execute the same app natively; each line represents a parameterized app, and each point represents a particular parameter setting. Our key finding is that, for sufficiently large parameters, three apps cross the line where outsourcing makes sense; that is, verifying the results of an outsourced computation is cheaper than local native execution. Note that the slope of each app's line is dictated by the size of the app parameters we experimented with (e.g., we reached larger parameters for fixed matrix than for two matrices).

On the downside, the other three apps, while trending in the right direction, fail to cross the outsourcing threshold. The difference is that these three apps perform large numbers of inequality comparisons and/or bitwise operations. This makes our circuit-based representation less efficient relative to native, and hence on our current experimental platform, we cannot push the application parameter settings to the point where they would beat local execution. Nonetheless, these applications may still be useful in settings that require Pinocchio's zero-knowledge proofs.

Fortunately, additional experiments show that enabling zero-knowledge proofs adds a negligible, fixed cost to key generation (213 µs), and re-randomizing a proof to make it zero-knowledge requires little effort (e.g., 300 ms or 0.1% for the multivariate polynomial app).

Figure 6 provides more details of Pinocchio's performance. For KeyGen, our experiments conservatively assume that the client does no precomputation in

[c] In contrast, Pinocchio's public verifier (not shown) enables a client to benefit from a third party's key setup work.

Figure 6. Application Performance. Pinocchio's performance for a sampling of the parameter settings (Section 4.3). All programs are compiled directly from C. The first two columns indicate the number of application inputs and outputs, and the number of gates in the corresponding arithmetic circuit. KeyGen is a one-time setup cost per application; Compute is the time the worker spends proving it computed correctly; Verify is the time the client spends checking the proof. Verification values in bold indicate verification is cheaper than computing the circuit locally; those with stars (*) indicate verification is cheaper than native execution. Public verification, while more expensive, allows anyone to check the results; private verification is faster, but allows anyone who can verify a proof to potentially generate a cheating proof. The Circuit column reports the time to evaluate the application's circuit representation, while Native indicates the time to run the application as a local, native executable. The last three columns indicate the size of the keys necessary to produce and verify proofs, as well as the size of the proof itself.

	|IO|	Mult gates	KeyGen pub(s)	Compute (s)	Verify Pub	(ms) Priv	Circuit (ms)	Native (ms)	EvalKey (MB)	VerKey (KB)	Proof (B)
Fixed matrix, Medium	1201	600	0.7	0.4	**39.5**	**10.0**	123.7	4.3	0.3	37.9	288
Fixed matrix, Large	2001	1000	1.5	0.9	**58.9**	***10.1**	337.4	12.4	0.5	62.9	288
Two matrices, Medium	14,701	347,900	79.8	269.4	340.7	**12.1**	124.9	4.0	97.9	459.8	288
Two matrices, Large	36,301	1,343,100	299.3	1127.8	882.2	***15.4**	509.5	15.5	374.8	1134.8	288
MultiVar poly, Medium	7	203,428	41.9	246.1	**11.6**	**10.0**	93.1	4.5	55.9	0.6	288
MultiVar poly, Large	7	571,046	127.1	711.6	***12.7**	***11.1**	267.2	12.9	156.8	0.6	288
Image matching, Medium	13	86,345	26.4	41.1	11.1	9.9	5.5	0.1	23.6	0.8	288
Image matching, Large	13	277,745	67.0	144.4	**11.4**	**10.1**	18.0	0.4	75.8	0.8	288
Shortest paths, Medium	513	366,089	85.4	198.0	25.5	**10.0**	18.7	0.1	99.6	16.4	288
Shortest paths, Large	1153	1,400,493	317.5	850.2	**48.9**	**10.8**	69.5	0.3	381.4	36.4	288
Lattice gas sim, Medium	21	144,063	38.2	76.4	**10.9**	**9.9**	91.4	0.2	39.6	1.1	288
Lattice gas sim, Large	21	283,023	75.6	165.8	**10.9**	**9.8**	176.6	0.4	77.7	1.1	288
SHA-1	22	23,785	12.0	15.7	**11.1**	**9.9**	18.8	0.0	6.5	1.1	288

anticipation of outsourcing a function, and for Compute, we assume that the worker only does a single work instance before throwing away all of its state. As discussed in Section 4.2.1, in practice, we would take advantage of both precomputation and caching of previous work, which on average saves at least 43% of the effort for KeyGen and 16% of the effort for Compute.

In Figure 6, we see again that three apps (starred) beat native execution, including one in the public verifier setting (which requires more expensive operations per IO). The data also reinforces the point that using a circuit representation imposes a significant cost on image matching, shortest paths, and the lattice gas sim relative to native, suggesting a target for optimization. Relative to the circuit representation, Pinocchio's verification is cheap: both the public and the designated verifier "win" most of the time when compared to the circuit execution. Specifically, the designated verifier wins in 12 of 13 (92%) application settings. Public verification is more expensive, particularly for large IO, but still wins in 9 of 13 (69%) settings.

Since Pinocchio offers public verification, some clients will benefit from the KeyGen work of others, and hence only care about the verification costs. For example, a cellphone carrier might perform the one-time KeyGen so that its customers can verify computations done by arbitrary workers.

However, in other settings, for example, a company outsourcing work to the cloud, the key generator and verifier may be the same entity, and will wish to amortize the cost of key generation via the savings from verification. Figure 6 shows that most apps have a low "break even" point vs. circuit execution: the median for the designated verifier is 555 instances and for public verifier is 500 instances. Every instance afterwards is a net "win," even for the key generator.

Figure 6 holds more good news for Pinocchio: the keys it generates are reasonably sized, with the evaluation key (which describes the entire computation) typically requiring 10s or 100s of MB. The weak verifier's key (which grows linearly with the I/O) is typically only a few KB, and even at its largest, for two-matrix multiplication, it requires only slightly more than 1 MB. This suggests that the keys are quite portable and will not require excessive bandwidth to transmit.

Finally, from the client's perspective, if the worker's efforts are free, then the worker's additional overhead of generating a proof is irrelevant, as long as it doesn't hurt response latency. Our results, combined with prior work on parallelization,[23] suggest that latency can be brought down to reasonable levels. And indeed in high-assurance scenarios, scenarios where the client is incapable of performing the calculation itself (e.g., a power-limited device), or scenarios where the worker's resources are otherwise idle, the client may very well view the worker as "free."

However, in other scenarios, such as cloud computing, the worker's efforts are not free. Even here, however, Chen and Sion[6] estimate that the cost of cloud computing is about 60× cheaper than local computing for a small enterprise. This provides an approximate upper-bound for the amount of extra work we should be willing to add to the worker's overhead.

6. RELATED WORK

When implementing verified computation, prior efforts focused on either interactive proofs or PCPs. One effort[7, 25] builds on the interactive proofs of Goldwasser et al.[12] (GKR). They target a streaming setting where the client cannot store all of the data it wishes to compute over; the system currently requires the function computed to be highly parallelizable. On the plus side, it does not require cryptography, and it is secure against computationally unbounded adversaries.

Setty et al. produced a line of PCP-based systems called Pepper[22] and Ginger.[23] They build on a particular type of PCP called a linear PCP,[14] in which the proof can be represented as a linear function. This allows the worker to use a linearly homomorphic encryption scheme to create a commitment to its proof while relying only on standard cryptographic assumptions. Through a combination of theoretical and systems-level improvements, this work made tremendous progress in making PCP-based systems practical. Indeed, for applications that can tolerate large batch sizes, the amortized costs of verification can be quite low.

A few downsides remain, however. Because the work builds on the Hadamard PCP,[1] the setup time, network overhead, and the prover's work are quadratic in the size of the original computation, unless the protocol is hand-tailored. To achieve efficiency, the verifier cannot verify the results until a full batch returns. The scheme is designated verifier, meaning that third parties cannot verify the results of outsourced computations without sharing the client's secret key and risking fraud. The scheme also does not support zero-knowledge proofs.

Concurrent work[21] also builds on the quadratic programs of Gennaro et al.[10] They observe that QAPs can be viewed as linear PCPs and hence can fit into Ginger's cryptographic framework.[23] Their work shows worker computation improvements similar to those of Pinocchio. They retain PCPs and Ginger's cryptographic protocol, so they rely on simpler cryptographic assumptions than Pinocchio, but they must still batch computations to obtain an efficient verifier. They also remain designated verifier and do not support zero-knowledge proofs.

A subsequent line of work[3] expands application expressivity by combining Pinocchio's cryptographic protocol with an innovative encoding of RAM accesses. They also propose an elegant program encoding based on a general-purpose CPU, but this leads to overheads, for applications like matrix multiplication, of 5–7 orders of magnitude compared with Pinocchio.

Several systems provide compilers for zero-knowledge (ZK) proofs.[17] In general, these systems are likely to exhibit better performance than Pinocchio for their particular subset of functionality, but they do not possess the same level of efficient generality.

7. CONCLUSION

We have presented Pinocchio, a system for public verifiable computing. Pinocchio uses quadratic programs, a new method for encoding computation, combined with a highly efficient cryptographic protocol to achieve both asymptotic and concrete efficiency. Pinocchio produces 288-byte proofs, regardless of the size of the computation, and the proofs can be verified rapidly, typically in tens of milliseconds, beating native execution in several cases. This represents five to seven *orders of magnitude* performance improvement over prior work.[23] The worker also produces the proof 19×–60× faster. Pinocchio even slashes the cost of its underlying protocol, cutting the cost of both key and proof generation by more than 60%. The end result is a cryptographic protocol for efficiently signing computations. Combined with a compiler for real C programs, Pinocchio brings VC much closer to practicality. C

References

1. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M. Proof verification and the hardness of approximation problems. *J. ACM 45*, 3 (1998).
2. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy* (2014).
3. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of USENIX Security* (2014).
4. Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M. Verifying computations with state. In *Proceedings of the ACM SOSP* (2013).
5. Castro, M., Liskov, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comp. Syst. 20*, 4 (2002).
6. Chen, Y., Sion, R. To cloud or not to cloud? Musings on costs and viability. In *Proceedings of the ACM Symposium on Cloud Computing* (2011).
7. Cormode, G., Mitzenmacher, M., Thaler, J. Practical verified computation with streaming interactive proofs. In *ITCS* (2012).
8. Danezis, G., Fournet, C., Kohlweiss, M., Parno, B. Pinocchio coin: Building Zerocoin from a succinct pairing-based proof system. In *ACM Workshop on Language Support for Privacy Enhancing Technologies* (2013).
9. Gennaro, R., Gentry, C., Parno. B. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of IACR CRYPTO* (2010).
10. Gennaro, R., Gentry, C., Parno, B., Raykova, M. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT* (2013). Originally published as Cryptology ePrint Archive, Report 2012/215.
11. Gentry, C. A fully homomorphic encryption scheme. PhD thesis, Stanford University (2009).
12. Goldwasser, S., Kalai, Y.T., Rothblum, G.N. Delegating computation: Interactive proofs for muggles. In *STOC* (2008).
13. Golle, P., Mironov, I. Uncheatable distributed computations. In *Proceedings of CT-RSA* (2001).
14. Ishai, Y., Kushilevitz, E., Ostrovsky, R. Efficient arguments without short PCPs. In *IEEE Conference on Computational Complexity* (2007).
15. Kilian, J. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC* (1992).
16. Lee, R.B., Kwan, P., McGregor, J.P., Dwoskin, J., Wang, Z. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the International Symposium on Computer Architecture* (*ISCA*) (2005).
17. Meiklejohn, S., Erway, C.C., Küpçü, A., Hinkle, T., Lysyanskaya, A. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of USENIX Security* (2010).
18. Naehrig, M., Niederhagen, R., Schwabe, P. New software speed records for cryptographic pairings. In *Proceedings of LATINCRYPT* (2010).
19. Parno, B., McCune, J.M., Perrig, A. *Bootstrapping Trust in Modern Computers*. Springer, New York/Dordrecht/Heidelberg/London, 2011. DOI: 10.1007/978-1-4614-1460-5.
20. Parno, B., Raykova, M., Vaikuntanathan, V. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *IACR Theory of Cryptography Conference* (*TCC*) (2012).
21. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the ACM European Conference on Computer Systems* (*EuroSys*) (Apr. 2013).
22. Setty, S., McPherson, R., Blumberg, A.J., Walfish, M. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the ISOC NDSS* (2012).
23. Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of USENIX Security* (2012).
24. Sion, R. Query execution assurance for outsourced databases. In *The Very Large Databases Conference* (*VLDB*) (2005).
25. Thaler, J. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of CRYPTO* (2013).

Bryan Parno and Jon Howell ({parno, howell}@microsoft.com), Microsoft Research.

Craig Gentry (cbgentry@us.ibm.com), IBM Research.

Mariana Raykova (mariana@cs.columbia.edu), SRI International.

Copyright held by authors. Publication rights licensed to ACM $15.00.

Watch the authors discuss their work in this exclusive *Communications* video. http://cacm.acm.org/videos/pinocchio-nearly-practical-verifiable-computation