Pinocchio: Nearly Practical Verifiable Computation

Bryan Pano, Jon Howell, Craig Gentry, Mariana Raykova

Presentated by Hui Zhang
Phd@ECE,UMD wayne.huizhang@gmail.com

Introduction

- Outsourcing complex computation to powerful servers is becoming popular.
- However, the workers that help the client to do the job is not always reliable: malicious or malfunctioning workers.
- Previous work has many shortages: impractical time consumption, function specific, not public verification, not zero-knowledge, etc.

Related works

- Function specific solutions [2–6] 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 [1, 7, 8] assume uncorrelated failures, while those based on Trusted Computing [9–11] or other secure hardware [12–15] assume that physical protections cannot be defeated. Finally, the theory community has produced a number of beautiful, general purpose protocols [16–23] that offer compelling asymptotics. In practice however, because they rely on complex Probabilistically Checkable Proofs (PCPs) [17] or fully-homomorphic encryption (FHE) [24], the performance is unacceptable –verifying small instances would take hundreds to trillions of years. Very recent work [25–28] has improved these protocols considerably, but efficiency is still problematic, and the protocols lack features like public verification.

Contributions of the paper

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.

- We’ll see how to use the tool later

Contributions

2. Theoretical and systems-level improvements that bring time down by ***5-7*** orders of magnitude, and hence into the realm of plausibility. The proof is only ***288*** bytes, regardless of the computation performed or the size of the inputs and outputs.

3. An evaluation on seven real C applications, showing verification faster than 32-bit native integer execution for **some** apps.

Background Knowledge

- Verifiable Computation (VC)

A public verifiable computation (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 native execution.

Public Verifiable Computation

- $(EK_F, VK_F) \leftarrow \mathsf{KeyGen}(F, 1^{\lambda})$:
- $(y, \pi_y) \leftarrow \mathsf{Compute}(EK_F, u)$:
- $\{0,1\} \leftarrow \mathsf{Verify}(VK_F, u, y, \pi_y)$:

Correctness, Security, Efficiency

- **Correctness** For any function F, and any input u to F, if we run $(EK_F, VK_F) \leftarrow \mathsf{KeyGen}(F, 1^\lambda)$ and $(y, \pi_y) \leftarrow \mathsf{Compute}(EK_F, u)$, then we always get $1 = \mathsf{Verify}(VK_F, u, y, \pi_y)$.
- **Security** For any function F and any probabilistic polynomial-time adversary $\mathcal{A}$, $\Pr[(\hat{u}, \hat{y}, \hat{\pi}_y) \leftarrow \mathcal{A}(EK_F, VK_F) : F(\hat{u}) \neq \hat{y}$ and $1 = \mathsf{Verify}(VK_F, \hat{u}, \hat{y}, \hat{\pi}_y)] \leq \mathtt{negl}(\lambda)$.
- **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.

Zero-knowledge Verifiable Computation

- 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

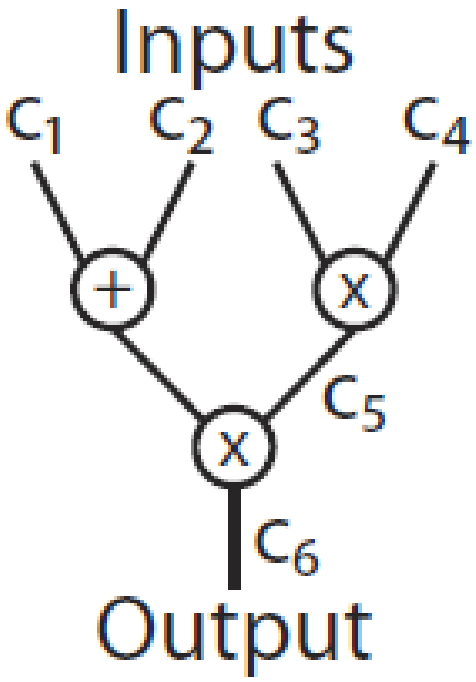
Quadratic Programs

- GGPR[2] has shown how to compactly encode computations as quadratic programs, so as to obtain efficient VC and zero-knowledge VC scheme.

- Specifically, they show how to convert any arithmetic circuit into a comparably sized QAP, and any Boolean circuit into a comparably sized QSP

Arithmetic Circuits and QAPs

An arithmetic circuit consists of wires that carry values from a field F and connect to addition and multiplication gates



Definition (QAP)

- QAP: an encoding of an Arithmetic Circuit

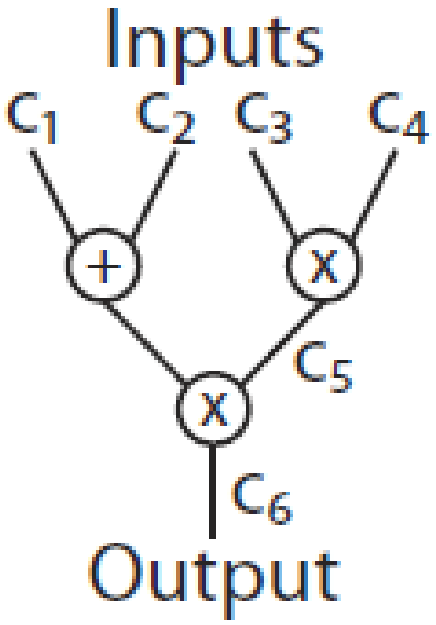
A QAP Q over field $\mathbb{F}$ contains three sets of $m+1$ polynomials $\mathcal{V} = \{v_k(x)\}$, $\mathcal{W} = \{w_k(x)\}, \mathcal{Y} = \{y_k(x)\}$, for $k \in \{0 \ldots m\}$, and a target polynomial $t(x)$. Suppose F is a function that takes as input n elements of $\mathbb{F}$ and outputs n' 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:*

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

Building a QAP

1. Pick an arbitrary root $r_g \in F$ for each multiplication gate g in C and define the target polynomial to be $t(x) = \prod g\left(x - r_g\right)$
2. Associate an index k={1,2…,m} to each input of the circuit and to each output from a multiplication gate
3. Define three sets V, W, Y: V encode left input to each gate, W encode right input, Y encode the output

Building a QAP



	(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)

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

Boolean Circuits and QSPs

- Boolean circuits operate over bits, with bitwise gates for AND, OR, XOR, etc. GGPR propose Quadratic Span Programs (QSPs) as a custom encoding for Boolean circuits

- QSPs are superficially similar to QAPs, but because they only support Boolean wire values, they use only two sets of polynomials V and W.

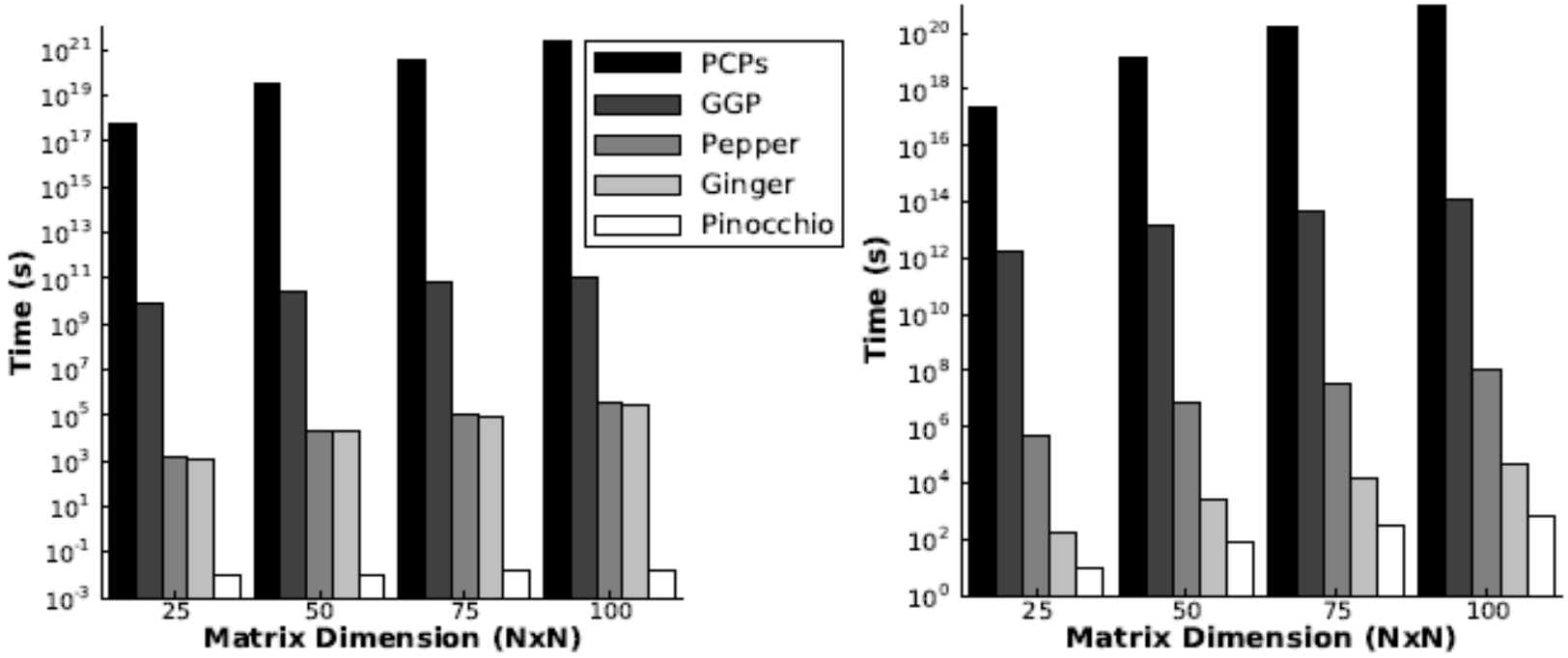
Theoretical Refinements

- Originally, we build VC from strong QAPs
- Its main optimization is that we construct a VC scheme that uses a regular QAP, rather than a strong QAP.
- They also remove the need for the worker to compute gah(s), and hence the gasii2[d] terms from EK. Finally, we expand the expressivity and efficiency of the functions QAPs can compute by designing a number of custom circuit gates for specialized functions.

Performance

- Proof size: 288-byte, regardless of the size of the computation
- Proof verification: 5~7 orders of magnitude performance improvement over prior work.
- Proof generation: 19-60 x faster.
- Cutting the cost of both key and proof generation by more than 60%.

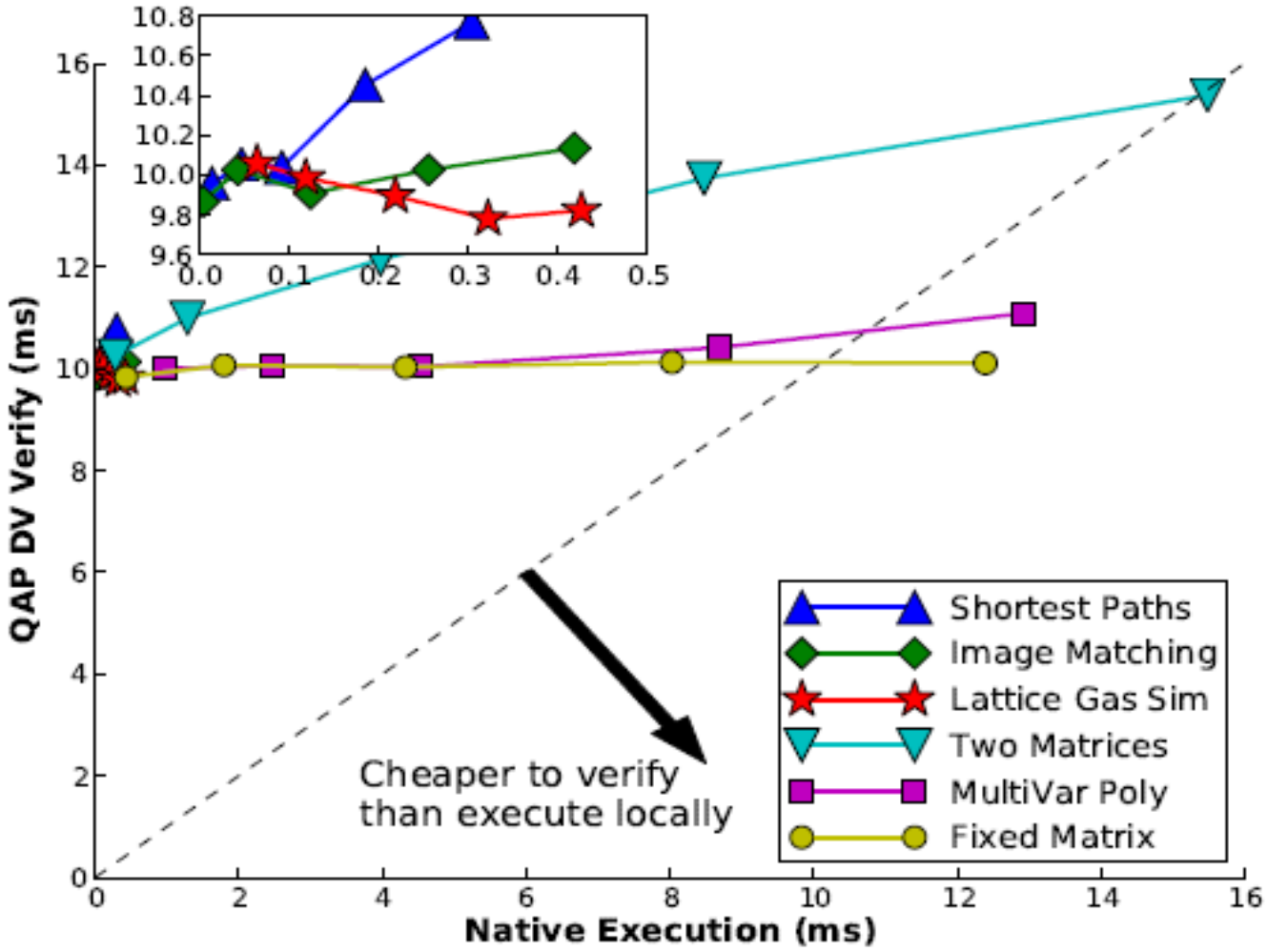
Performances



(a) Per-Instance Verification Latency (b) Worker Latency

Figure 6: **Performance Relative to Prior 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 NxN matrices.*

Performances



Tool Usage

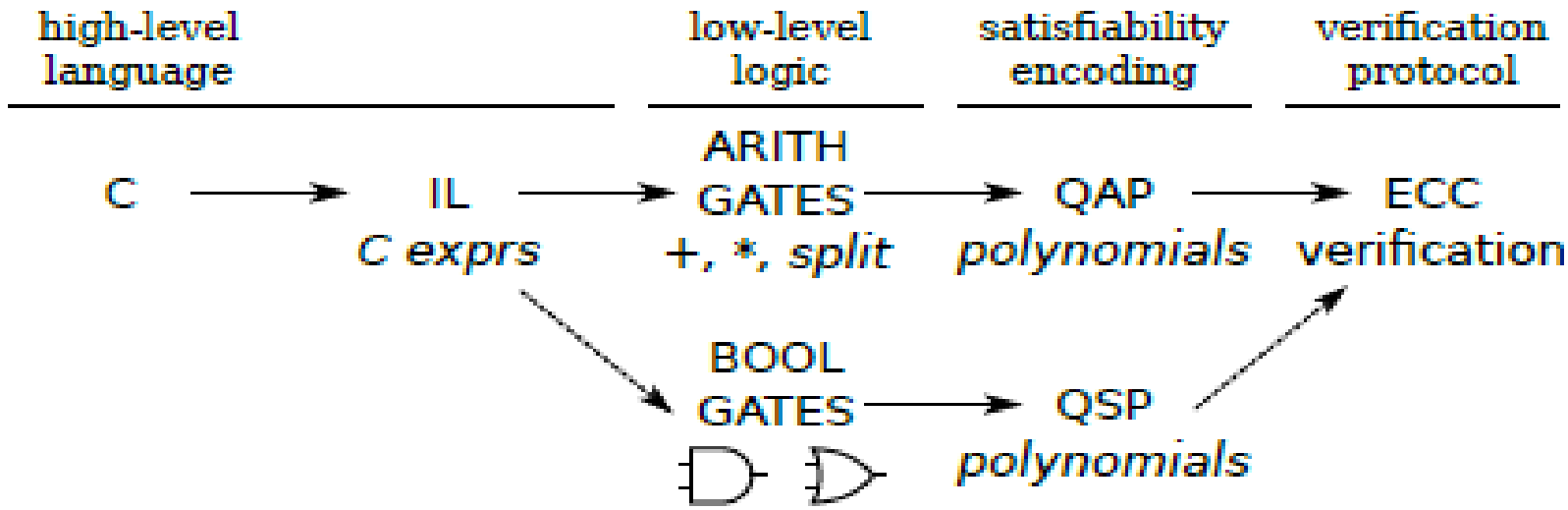


Figure 1: Overview of Pinocchio's Toolchain. *Pinocchio takes a high-level C program all the way through to a distributed set of executables that run the program in a verified fashion. It supports both arithmetic circuits, via Quadratic Arithmetic Programs (§2.2.1), and Boolean circuits via Quadratic Span Programs (§2.2.2).*

Steps:

- *First step: Generate a .arith file that describes the circuit you will be operating on.*
  - 1) go to common/ , change corresponding parameters and name of file you wan to test in App.py
  - 2) go to ccompiler/input, run ../src/build-test-matrix.py
  - 3) run make -f make.matrix
- *Second step: Run Pinocchio using the arith file obtained from the previous step*
  - pinocchio-v0.4.exe --qap --pv --bits 32 --mem 1 --file /path/to/.arith

Q&A

Thank you !

Reference

- [1] Parno, Bryan, et al. "Pinocchio: Nearly practical verifiable computation."*Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

- [2] Gennaro, Rosario, et al. "Quadratic span programs and succinct NIZKs without PCPs." *Advances in Cryptology–EUROCRYPT 2013*. Springer Berlin Heidelberg, 2013. 626-645.