A short report on:*

Pinocchio: Nearly Practical Verifiable Computation

Karim Baghery

Supervised by Prof. Dominique Unruh

University of Tartu, Estonia
karim.baghery@ut.ee

December 17, 2016

Abstract

In this report, we aim to present a short description on the *Pinocchio* which is an efficient solution for Verifiable Computations (VC) and presented by Parno, Howell, Gentry, and Raykova in 2013 IEEE Symposium on Security and Privacy [PHGR13]. In fact, *Pinocchio* is a practical implementation of SNARKs (Succinct non-interactive argument of knowledge) which are systems that allow a client to ask a server or cloud to compute $F(x)$ for a given function F and an input x and then verify the correctness of the returned result in considerably less time than it would take to compute F from scratch; this property is also known as VC. In addition, Pinocchio supports zero-knowledge property, in which the server convinces the client that it knows an input with a particular property, without revealing any information about the input.
Pinocchio takes a high-level C program and compiles it to a low-level logic circuit and then encodes the logic circuit to a quadratic program; And after that, it compiles the quadratic program to a cryptographic verification protocol. In this report, we mainly focus on encoding a logic circuit to a quadratic program and compiling it to a cryptographic verification protocol, which consists in systems-level improvements and brings time down 5-7 order of magnitude, which will be discussed in 7 practical applications.

1 Introduction

Verifiable computation (VC) is the process of enabling a client to outsource the computation of some function, to other untrusted servers (workers), while maintaining verifiable results.

*This report has submitted in partial fulfillment of the requirements for the course *Research Seminar in Cryptography* (MTAT.07.022) in Fall 2016 and has presented as a seminar at Institute of Computer Science, University of Tartu.

The workers evaluate and compute the function and send a proof to the user/individual in order to show the computation of the function was carried out correctly. The main aim of this process is to verify the results faster than the client computing the results itself. In other words, the outsourcing and verification procedures must be more efficient than performing the computation itself. It is also known as verified computation or verified computing [BDD+16].

From another point of view, the main motivation of VC is due to the growing desire to outsource computational tasks from computationally weak devices to a more powerful computation services. There has been considerable attention devoted towards verifying the computation of functions performed by untrusted workers which also includes the use of secure coprocessors, interactive proofs, and efficient arguments [IKO07, GGP10, SMBW12, SVP+12, PHGR13]. One of the new-found and efficient solutions is *Pinocchio* which has proposed by Parno, Howell, Gentry, and Raykova in 2013 [PHGR13]. In the rest of this report we will get more acquainted with this system and will discuss about its performance.

1.1 Pinocchio: an efficient solution for VC

Pinocchio is based on some cryptographic assumptions and supports public verifiable computation which allows an untrusted worker to produce a proof (signatures) of computation and anyone can verify the correctness of the computed result by the worker. In the other class of verification, designated-verifier, verification can be done just by the target verifier who has access to the secret key [GGPR13].

In summary, Pinocchio follows the pipeline of Fig. 1. According to figure, it can be seen that for all kind of computations, one first needs to write the C code of the computation and then uses the corresponding compiler (C code to arithmetic circuit compiler) to get an arithmetic circuit which can perform basic operations such as adding, multiplying and module in p. Beside standard basic operation gates, it uses some customized gates for special computations such as equality check or split-value to binary bits which allows several bitwise operations as well. In the next step, Pinocchio encodes the *arithmetic circuit* (AC) to the *Quadratic Arithmetic Program* (QAP) which provides efficient encoding of the original computation; which means instead of computing original circuit, one can compute obtained QAP efficiently. Finally, Pinocchio converts the obtained QAP to a cryptographic protocol and follows the procedure of the protocol. More precisely, in the verification protocol, initially the client chooses a function and generates a public evaluation key and a (small) public verification key. Given the evaluation key, a worker can choose an input (or verifiably use one provided by the client), compute the function, and produce a proof to accompany the result. Finally, anyone (not just the client) can then use the verification key to check the correctness of the workers result for the specific input used.

In addition, Pinocchio provides zero-knowledge verifiable computation; which means, in some applications, the worker convinces the client that it knows an input with a particular property, without revealing any information about the input. In this report, we discuss about the AC, QAP, and the cryptographic protocol blocks of the pipeline with more details.

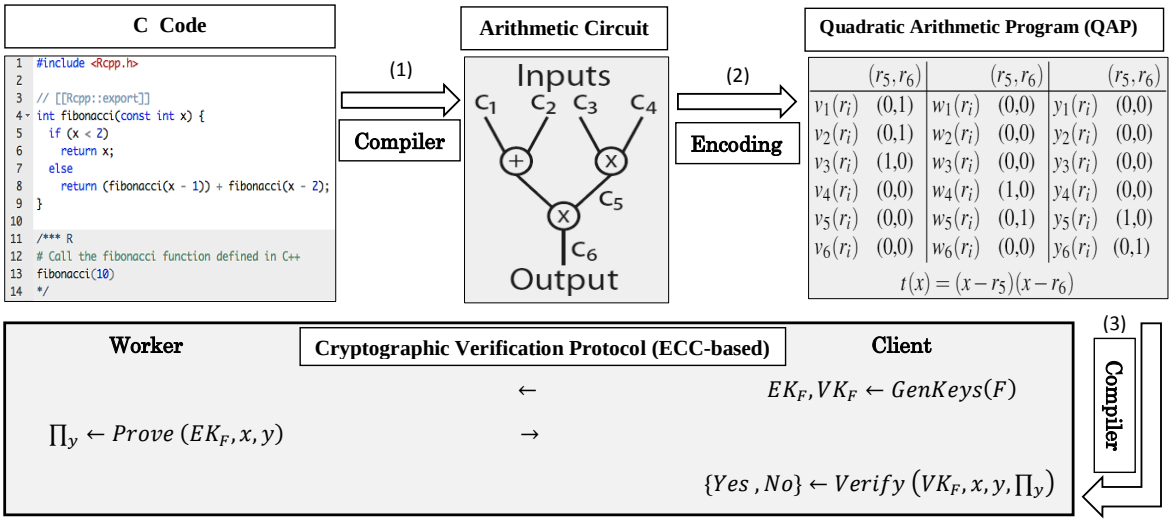


Figure 1: Pinocchio's verification pipeline

1.2 Overview on Results

In Pinocchio, key generation and proof generation by the worker require cryptographic effort linear in the size of the original computation, and verification requires time linear in the size of the inputs and outputs. An interesting fact about Pinocchio is that its proof is constant sized, regardless of the computation performed. Several evaluation of different application shows that these performances can be achieved with small constants, making Pinocchio close to practical for a variety of applications. In comparison with previous work, Pinocchio improves verification time by 5-7 orders of magnitude and requires less than $10ms$ in most applications, which make it possible to get closer to local C code execution for some applications. In Pinocchio, the workers proof efforts has improved by 19-60× relative to prior work. And as mentioned already the proof is tiny, 288 bytes (slightly more than an RSA-2048 signature), regardless of the computation. According to the result of the paper, making a proof zero-knowledge adds $213\mu s$ to key generation and 0.1 % to proof generation.

In summary, mainly contribution of the Pinocchio are as, 1: An end-to-end system for efficiency verifying computation performed by one or more untrusted servers. This include a compiler that converts 'C' code into a format suitable for verification, as well as a suit of tools for running the actual protocol; 2: Theoretical and systems-level improvements that bring time down 5-7 order 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 input of output. 3: An evaluation on several real C applications, showing verification faster than 32-bit native integer execution for some applications.

Note that Pinocchio offers public verifiability, but no input-output privacy. Due to its preprocessing phase that runs in time proportional to a one time execution of function f it only achieves amortized efficiency.

2 Preliminaries

We will give an overview of some basic background on verifiable computations, quadratic arithmetic programs which are essential concepts in Pinocchio and are helpful to follow the rest of report.

2.1 Verifiable Computation

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. According to the paper, the public VC is defined as follows [PHGR13],

Definition 1 (Public Verifiable Computation) A public verifiable computation scheme VC consists of a set of three polynomial-time algorithms $(\mathsf{KeyGen}; \mathsf{Compute}; \mathsf{Verify})$ defined as follows.

- $(EK_F; VK_F) \leftarrow \mathsf{KeyGen}(F, 1^\lambda)$: 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 iff $F(u) = y$, and 0 otherwise.

In a VC system, there are essential concepts which are as correctness, security and efficiency which can be summarized as follows [PHGR13],
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 A, $Pr[(\hat{u}, \hat{y}, \hat{\pi}_y) \leftarrow A(EK_F, VK_F) : F(\hat{u}) \neq \hat{y}$ and $1 = \mathsf{Verify}(VK_F, \hat{u}, \hat{y}, \hat{\pi}_y)] \leq \mathsf{negl}(\lambda)$
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.

Beside defined properties, Zero-Knowledge is a relevant concept in applications which the server's input is private. In Pinocchio, this concept defined as follows,
Zero-Knowledge: In the setting that the outsourced computation is a function, $F(u; w)$, of two inputs: the clients input u and an auxiliary input w from the worker. A VC scheme is zero-knowledge if the client learns nothing about the workers input beyond the output of the computation.

2.2 Arithmetic Circuits and Quadratic Arithmetic Program

As we already observed in previous section, one of the main blocks in Pinocchio is encoding AC to QAP. In this subsection, we will introduce QAP circuits and summarize necessary concepts which will has used in Pinocchio and will be used in the rest of report.

Recently in [GGPR13], Gennaro, Gentry, Parno, and Raykova (GGPR) described that how to compactly encode computations as quadratic programs to obtain efficient VC and zero-knowledge scheme. More precisely, they illustrated how to convert any AC into a comparably sized QAP, and any Boolean circuit into a Quadratic Span Program (QSP). Basically an QAP is encoding of an AC which has consisted of some wires that carry values from a field $\mathbb{F}$ and connect to addition and multiplication gates and output a computed value. An QAP can be defined formally as follows;

Definition 2 (Quadratic Arithmetic Program (QAP) [GGPR13]) A QAP Q over field $\mathbb{F}$ contains three sets of $m+1$ polynomials $V = \{v_k(x)\}$, $W = \{w_k(x)\}$, $Y = \{y_k(x)\}$ for $k \in \{0, 1, ..., m\}$, and a target polynominal $t(x)$. 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 $(c_1, ..., c_N) \in \mathbb{F}^N$ is a valid assignment of F's inputs and outputs, if and only if there exists coefficients $(C_{N+1}, ..., c_m)$ such that $t(x)$ divides $p(x)$, where

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

Actually 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 $t(x)$. Note that in the case that each set of polynomials have different coefficients (i.e. a_k, b_k and c_k) the QAP called strong-QAP which presented VC scheme in [GGPR13] is base on the strong-QAP, which in their case, the presented definition in Eq. (1) for $p(x)$ will be as,

$$p(x) = \left(v_0(x) + \sum_{k=1}^{m} a_k.v_k(x)\right) \times \left(w_0(x) + \sum_{k=1}^{m} b_k.w_k(x)\right) - \left(y_0(x) + \sum_{k=1}^{m} c_k.y_k(x)\right). \quad (2)$$

As already mentioned, Pinocchio uses regular QAP in its encoding which will be explained with more details in the following subsection.

3 How *Pinocchio* Encodes an AC to a QAP

In the Pinocchio, encoding an arithmetic Circuit to a QAP is as follows; First we pick an arbitrary root $r_g \in \mathbb{F}$ for each multiplication gate g in the circuit and define the target polynomial as $t(x) = \prod_g (x - r_g)$. We associate an index $k \in \{1, 2, ..., m\}$ to each input of the circuit and to each output from a multiplication gate. Note that the addition gates will be modeled with their contributions to the multiplication gates. Then, we define the polynomials in V , W, and Y by letting the polynomials in V encode the left input into

each multiplication gate, the W encode the right input into each gate, and the Y encode the outputs. In order to encode, for each multiplication gate g, we set $v_k(r_g) = 1$ if the kth wire is a left input to the gate, and similarly $w_k(r_g) = 1$ if the kth wire is a right input to the gate, and $y_k(r_g) = 1$ if the kth wire is the output of the gate g, and in each of the other case the respective coefficient is 0; it means $v_k(r_g) = 0, w_k(r_g) = 0, y_k(r_g) = 0$. With this encoding it can be seen that Eq. (1) can be rewritten as,

$$\left(\sum_{k=1}^{m} c_k.v_k(r_g)\right)\left(\sum_{k=1}^{m} c_k.w_k(r_g)\right) = \left(\sum_{k\in I_{left}} c_k\right)\left(\sum_{k\in I_{right}} c_k\right) = c_g y_k(r_g) = c_g,$$

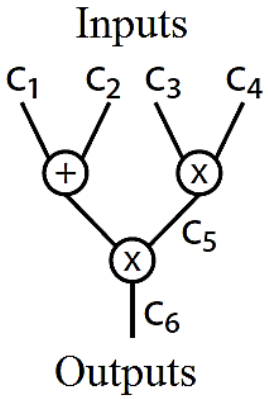
which says that the output of the gate g is equal to product of its inputs. Note that it can be seen that the divisibility check that $t(x)$ divides $p(x)$ decomposes into $\mathsf{deg}(t(x))$ separate checks, one for each gate g and root r_g of $t(x)$, which checks that for each gate g with root r_g that $p(r_g) = 0$.

For instance, assume that we are going to encode the AC (left figure) in Fig. 2 to a QAP (right figure). In this case, since we have two multiplication gates in the circuit, we choose two roots r_5 and r_6 from $\mathbb{F}$ to represent them. As a result the degree of QAP is equal to 2. Then we define six polynomials for each set V, W, and Y, four for the input wires (C_1, C_2, C_3, C_4), and two (C_5, C_6) for the outputs from the multiplication gates. Thus, the QAPs size is 6. We define these polynomials based on each wires contributions to the multiplication gates. In this concrete example, for gate C_5 the left input is third input (C_3), as a result $v_3(r_5) = 1$ and its right input is from C_4 which means $W_4(r_5) = 1$, and its output is C_5 which results that $y_5(r_5) = 1$; All the rest of polynomials for this gate are equal to zero which means for left input polynomials $v_1(r_5) = v_2(r_5) = v_4(r_5) = v_5(r_5) = v_6(r_5) = 0$, and for right input polynomials $w_1(r_5) = w_2(r_5) = w_3(r_5) = w_5(r_5) = w_6(r_5) = 0$ and for output polynomials $y_1(r_5) = y_2(r_5) = y_3(r_5) = y_4(r_5) = y_6(r_5) = 0$. Similarly for multiplication gate C_6, it can be seen that its left input is equal to sum of two first inputs $(C_1 + C_2)$ which says that $v_1(r_6) = v_2(r_6) = 1$, and its right input is from C_5 which means $w_5(r_6) = 1$ and its output is C_6 which results that $y_6(r_6) = 1$ and similar to first gate, all other polynomials will be zero (i.e. $v_3(r_6) = v_4(r_6) = v_5(r_6) = v_6(r_6) = 0$).

In [GGPR13], GGPR show that for any AC with d multiplication gates and N I/O elements, one can construct an equivalent QAP with degree d (the number of roots r_g which is equal to multiplication gates) and size $d + N$ (number of polynomials in each set $\in \{V, W, Y)$.

4 How *Pinocchio* builds a VC protocol from a QAP

The main idea of Pinocchio to construct a VC protocol from a quadratic program, is that each one of the polynomials $v_k(x), w_k(x), y_k(x) \in \mathbb{F})$ of the quadratic program (QAP or QSP) is mapped to elements $g^{v_k(s)}$, $g^{w_k(s)}$ and $g^{y_k(s)}$ in a bilinear group, where s is a secret value selected by the client, g is a generator of the group, and F is the field of discrete logarithms of g. These group elements are given to the worker. For a given input, the worker evaluates the



	(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)

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

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.

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 $v(s) = \sum_{k\in[m]} c_k v_k(s)$, $w(s) = \sum_{k\in[m]} c_k w_k(s)$, and $y(s) = \sum_{k\in[m]} c_k y_k(s)$, "in the exponent" to get $g^{v(s)}$, $g^{w(s)}$ and $g^{y(s)}$; Finally, 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 in the evaluation key, to compute $g^{h(s)}$. Note that in order to make the verification simple, the proof consists of $(g^{v(s)}, g^{w(s)}, g^{y(s)}, g^{h(s)})$.

4.1 Refinements on the GGPR VC Protocol

Pinocchio is obtained by modifying the GGPR VC protocol which is based on strong-QAPs and has proposed in [GGPR13]. Practical implementation of Pinocchio show that using regular QAP and new modifications significantly reduce key generation time, evaluation key size, and worker effort. More details will be discussed in the next section of report. The main optimization in Pinocchio is designing new VC scheme which uses regular QAP and some new embedding structures which decrease proof elements from 9 in the GGPR protocol to 8. Pinocchio also uses some custom circuits gates for specialized functions such as *Split Gate* (for converting a value to a binary string), *Equality-Assertion Gate* (to check equality of values of two wires) and *Zero-Equality Gate* (to check whether a value is equal to zero).

4.2 Pinocchio: VC Protocol from regular QAP

As explained in section 1.1 and showed in Fig. 1, after encoding the compiled AC (obtained from the C program) to a QAP , Pinocchio compiles the QAP to a cryptographic verification protocol which will be explained in this section. By applying some modifications in the GGPR protocol [GGPR13] and using regular QAP in the new VC protocol, the Pinocchio VC protocol has obtained which can be expressed in three algorithms KeyGen, Compute, and

Verify as follows,

- $(EK_F, VK_F) \leftarrow \mathsf{KeyGen}(F; 1^\lambda)$: Let F be a function with N input/output values from F. Convert F into an arithmetic circuit C; then build the corresponding QAP $Q = (t(x), V, W, Y)$ of size m and degree d. Let $I_{mid} = N+1, \ldots, m$ i.e., the non-IO-related indices.

  Let e be a non-trivial bilinear map $e : G \times G \to G_T$, and let g be a generator of G.

  Choose $r_v, r_w, s, \alpha_v, \alpha_w, \alpha_y, \beta, \gamma \xleftarrow{R} \mathbb{F}$ 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:

$$\{g_v^{v_k(s)}\}_{k \in I_{mid}} \ , \{g_w^{w_k(s)}\}_{k \in I_{mid}} \ , \{g_y^{y_k(s)}\}_{k \in I_{mid}}$$

$$\{g_v^{\alpha_v v_k(s)}\}_{k \in I_{mid}} \ , \{g_w^{\alpha_w w_k(s)}\}_{k \in I_{mid}} \ , \{g_y^{\alpha_y y_k(s)}\}_{k \in I_{mid}}$$

$$\{g^{s^i}\}_{i \in d} \ , \{g_v^{\beta v_k(s)} g_w^{\beta w_k(s)} g_y^{\beta y_k(s)}\}_{k \in I_{mid}} \ .$$

  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 $\{c_i\}_{i \in [m]}$ of the circuits wires. He solves for h(x) (the polynomial such that $p(x) = h(x) \cdot t(x)$), and computes the proof pi_y as:

$$g_v^{v_{mid}(s)} \ , g_w^{w_{mid}(s)} \ , g_y^{y_{mid}(s)} \ , g^{h(s)}$$

$$g_v^{\alpha_v v_{mid}(s)} \ , g_w^{\alpha_w w_{mid}(s)} \ , g_y^{\alpha_y y_{mid}(s)}$$

$$g_v^{\beta_v v_{mid}(s)} \ , g_w^{\beta_w w_{mid}(s)} \ , g_y^{\beta_y y_{mid}(s)}$$

  where $v_{mid}(x) = \sum_{k \in I_{mid}} c_k \cdot v_k(x)$, and similarly for $w_{mid}(s)$ and $y_{mid}(s)$.

- $\{0, 1\} \leftarrow \mathsf{Verify}(VK_F, u, y, \pi_y)$: 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 $g^{v_{io}(s)} = \prod_{k \in [N]} (g^{v_k(s)})^{c_k}$ (and similarly for $g^{w_{io}(s)}$ and $g^{y_{io}(s)}$), and check:

$$e(g_v^{v_o(s)} \ , g_v^{v_{io}(s)} \ , g_v^{V_{mid}} \ , g_w^{w_o(s)} \ , g_w^{w_{io}(s)} \ , g_w^{W_{mid}}) = \tag{3}$$

$$e(g_y^{t(s)}, g^H) e(g^{y_o(s)}, g_y^{y_{io}(s)}, g_y^{Y_{mid}(s)}, g) \tag{4}$$

- Check that the linear combinations computed over V, W and Y are in their appropriate spans:

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

- Check that the same coefficients were used in each of the linear combinations over V, W and Y:

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

The correctness of the Pinocchio VC protocol follows from the properties of the QAP and can be shown by directly extending verification equations. But main intuition behind the security of the protocol is this fact that it seems hard for an adversary who does not know a to construct any pair of group elements h, h^a *except* in the obvious way: by taking pairs $(g_1, g_1^a), (g_2, g_2^a), \ldots$ 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 [Dam91], 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, one 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 do not necessarily correspond to a QAP solution. So, either $p(x)$ is not actually divisible by $t(x)$ (in this case one 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 one break q-PDH because in the proof we choose b in a clever way). Complete proof can be found in appendix of [PHGR13].

Zero Knowledge. Pinocchio applies GGPRs rerandomization technique [GGPR13] to provide zero-knowledge. The worker chooses $\delta_v, \delta_w, \delta_y \xleftarrow{R} F$ 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 $y(x) + \delta_y t(x)$. In order to facilitate the randomization of the proof they have add the following terms to the evaluation key:

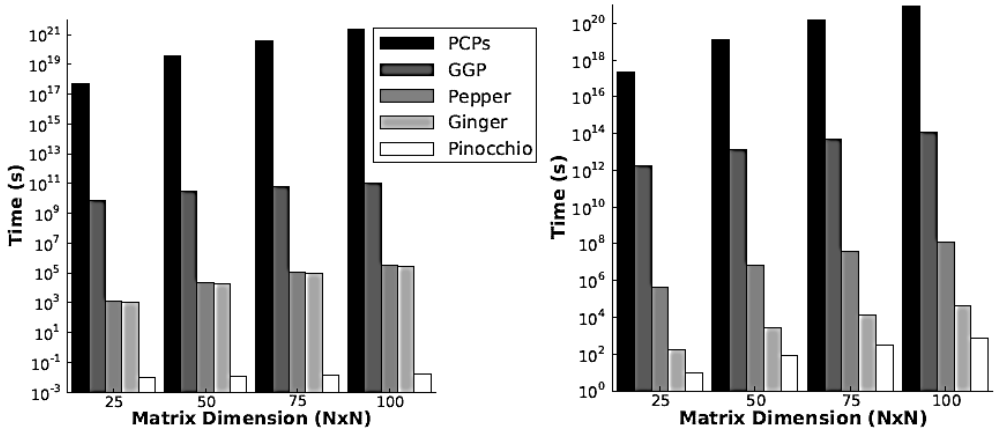
$$g_v^{\alpha_v t(s)} \ , g_w^{\alpha_w t(s)} \ , g_y^{\alpha_y t(s)} \ , g_v^{\beta_v t(s)} \ , g_w^{\beta_w t(s)} \ , g_y^{\beta_y t(s)} \ .$$

Performance. Pinocchio VC scheme requires a regular QAP, which improves performance significantly, and the scheme has simpler construction, which leads to fewer group elements in the keys and proof, fewer bilinear maps for *Verify*, etc. The scheme above assumes a symmetric bilinear map. But, in practice, in order to increase performance, they use an asymmetric bilinear map $e : G_1 \times G_2 \longmapsto G_T$ where G_1 is an elliptic curve group called the "base" curve, and G_2 is the "twist" curve. Operations over the base curve are about 3 times faster than over the twist curve. Due to some optimizations approaches, while the worker must compute the $g_w^{w(s)}$ term over the twist curve, all of the other proof terms can be over the base curve.

5 Implementation of Pinocchio

Pinocchio contains a compiler that takes a subset of C code to an equivalent AC, which then encodes the circuit to the equivalent QAP, and generates code to run the VC protocol, including key generation, proof computation, and proof verification. Author evaluated the performance of Pinocchio for following applications (with given parameters) and for the matrix multiplication they have compared it with previous literature.

- 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 a parameter n, takes as input two $n \times n$ matrices M_1 and M_2, and outputs the $n \times n$ matrix $M_1 \cdot M_2$. ($|M| = 30 \times 30$ 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 $x_1, ..., x_k$ 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.($i_w \times i_h = 25, k_h \times k_h = 9$ 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 $n = 24, e = 576$)



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

Figure 3: A comparison of pre-instance verification latency and worker latency in Pinocchio and previous VC schemes for multiplying two $N \times N$ matrices.

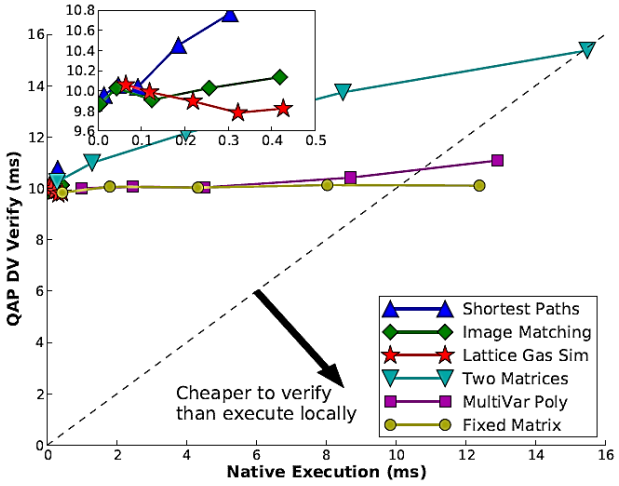


Figure 4: A comparison of the cost of verification by Pinocchio and by local execution. Three applications cross the scattered line where verification starts to be cheaper than local execution.

- *LGCA* is a Lattice-Gas Cellular Automata implementation that converges to Navier-Stokes [50]. 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.

For the cryptographic code, they have used a high speed elliptic curve library [NNS10] with a 256-bit BN-curve [BN05] that provides 128 bits of security. In the rest of section, we summarize the result of implementation of Pinocchio in comparison with other similar schemes. Figure 3 plots Pinocchios pre-instance verification latency and worker latency in comparison with some previous general-purpose systems. As already mentioned, this figure is plotted for multiplication of two matrices. They have compared with a PCP-based scheme [IKO07], the GGP which is based on fully-homomorphic encryption (FHE) [GGP10], the Pepper [SMBW12] which is an optimized version of [IKO07] and Ginger [GGP10] which is a modified version of of Pepper. More details about these schemes can be found in section 6 of original paper of Pinocchio [PHGR13]. According to the figure, it can be seen that Pinocchio continues the trend of reducing costs by orders of magnitude. We see that Pepper and Ginger have made efficient improvements over prior works, but they do not offer public verification or zero knowledge. The next evaluation is regard to end-to-end performance of Pinocchio in different implemented applications versus local execution of a C code of the mentioned programs. In other word, all the mentioned applications are written in C and compile to both QAPs and to native executables. Figure 4 plots Pinocchio's verification time against the time to execute the same applications without outsourcing; each line represents a parameterized application, and each point represents a particular parameter setting.

	|IO|	Mult Gates	KeyGen Pub (s)	Compute (s)	Verify (ms) Pub	Verify (ms) Priv	Circuit (ms)	Native (ms)	EvalKey (MB)	VerKey (KB)	Proof (B)
Fixed Matrix, Medium	1,201	600	0.7	0.4	**39.5**	**10.0**	123.7	4.3	0.3	37.9	288
Fixed Matrix, Large	2,001	1,000	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	1,153	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

Figure 5: Performance of studied applications with Pinocchio for the assumed parameters in implementations. Verification values in bold indicate verification is cheaper than computing the circuit locally; those with stars (*) indicate verification is cheaper than local execution.

Their key finding is that, for sufficiently large parameters, three applications (Fixed Matrix, MultivAR Poly and Two Matrix Multiplication) cross the line where outsourcing makes sense; i.e., verifying the results of an outsourced computation is cheaper than local execution of C code of the program. On the downside, the other three applications, while trending in the right direction, fail to cross the outsourcing threshold. The main reason for this fact is that the three mentioned applications perform a large number of inequality comparisons and/or bitwise operations which make Pinocchio's circuit-based representation less efficient relative to native, and there is not any setting for the applications which can beat local execution. Finally, Fig. 5 provides more details of Pinocchio's performance. In the case of KeyGen, it is assumed that the client does no precomputation in anticipation of outsourcing a function. According to the figure, again it can be seen that three application (starred) beat local execution of C codes, including one in the public verifier setting. 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. From Fig. 5, we see that the keys that Pinocchio generates are reasonably sized, with the evaluation key typically requiring $10s$ or $100s$ of MB. The weak verifier's key (which grows linearly with the I/O) is a few KB, and even at its largest, for two-matrix multiplication, it requires only slightly more than 1 MB.

In Fig. 6, compare the performance of GGPR's protocol [GGPR13] with Pinocchio for the same underlying cryptographic and polynomial libraries. It shows that Pinocchio's VC protocol improvements had a significant impact on KeyGen and Compute are more than twice as fast, and even verification is 24% faster. Mainly these improvements are because of using regular QAP in Pinocchio. For Compute, the multi-exponentiation required to compute the

	GGPR [30]	This Paper	Reduction
KeyGen	108.7s	41.9s	61%
Build table	7.8s	7.9s	-2%
Encode powers of s	28.4s	4.7s	83%
Eval polys at s	5.0s	1.7s	66%
Encode polys	67.2s	27.4s	59%
Compute	691.4s	246.1s	64%
Solve for $h(x)$	252.3s	76.3s	70%
Apply coefficients	391.1s	154.7s	60%
Verify	15.2ms	11.6ms	24%
Process I/O	456.5μs	901.8μs	-98%
Crypto checks	14.8ms	10.7ms	28%
Evaluation Key Size	105.5MB	55.9MB	47%
Verification Key Size	640B	640B	0%
Proof Size	352B	288B	18%

Figure 6: A comparison of Pinocchio and GGPR [GGPR13] VC schemes.

QAP's polynomials in the exponent still dominate, but the overhead of solving for $h(x)$ is nontrivial as well.

6 Conclusions

In this report, we detailed main intuition and efficiency of *Pinocchio* which is an efficient solution for public verifiable computing has presented in 2013 [PHGR13]. We got acquainted with pipeline of the Pinocchio and described different parts of this its pipeline. We saw that how Pinocchio uses quadratic programs combined with a cryptographic verification protocol to brought verification time down 5-7 orders of magnitude. We observed that it produces 288-byte proofs, regardless of the size of the computation, and the proofs can be verified rapidly. Finally we provided a short description of implementation of Pinocchio and its performance in seven different practical application.

References

[BDD+16] Johannes Buchmann, Denise Demirel, David Derler, Lucas Schabhüser, Daniel Slamanig, Reviewers Daniel Slamanig, and Thomas Gross. Overview of verifiable computing techniques providing private and public verification. 2016.

[BN05] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.

[Dam91] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *Annual International Cryptology Conference*, pages 445–456. Springer, 1991.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*, pages 465–482. Springer, 2010.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[IKO07] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short pcps. In *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC'07)*, pages 278–291. IEEE, 2007.

[NNS10] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *International Conference on Cryptology and Information Security in Latin America*, pages 109–123. Springer, 2010.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252. IEEE, 2013.

[SMBW12] Srinath TV Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, volume 1, page 17, 2012.

[SVP+12] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 253–268, 2012.