Proofless Verifiable Computation from Integer Factoring

Alex Dalton, David Thomas, and Peter Cheung

Abstract—Verifiable Computation (VC) schemes provide a mechanism for verifying the output of a remotely executed program. These are used to support computing paradigms wherein a computationally restricted client, the Verifier, wishes to delegate work to a more powerful but untrusted server, the Prover. The Verifier wishes to detect any incorrect results, be they accidental or malicious. The current state-of-the-art is only close-to-practical, usually because of a computationally demanding setup which must be amortised across repeat executions. We present a VC scheme for verifying the output of arithmetic circuits with a small one-time setup, KGen, independent of the size of the circuit being verified, and a insignificantly small constant program specific setup, ProbGen. To our knowledge our VC scheme is the first built from the hardness of integer factoring, a standard cryptographic assumption. Our scheme has the added novelty that the proofs are simply the raw output of the target computation, and the Prover is in effect blind to the fact they are taking part in a VC scheme at all. Although our scheme has worse asymptotic performance than the state-of-the-art it is particularly well suited for verifying one-shot programs and the output of large integer polynomial evaluation.

Index Terms—Verifiable computation, Cloud computing, Delegated computing, Distributed computing

1 Introduction

VC schemes are protocols by which a Prover can convince a Verifier of the correctness of the output of some computation. There are obvious applications to a huge variety of client/server programs. From cloud deployments [1] to distributed volunteer scientific computation [2], the client/server paradigm is a hugely popular and widely accepted standard program architecture.

In programs using this design pattern there is usually a powerful server performing calculations on behalf of a weaker client. Despite how commonplace this style of computation is, there are relatively few usable mechanisms by which the client can generate strong guarantees of the correctness of the output from a computation run on a remote server. The best performing VC schemes to date have extremely efficient verification steps, but expensive setup phases that must be amortised across multiple repeat executions of the same program with different inputs.

Worse still, existing VC schemes assume that there is no bound on the computational power of the Prover. In all the related work the computational complexity of the prover increases by some non-negligible multiplicative factor over unverified computation. In practice the Prover is rarely unbounded, and often the Verifier will be incentivised to minimise their use of the Prover, perhaps because they are paying for the Prover's time.

A compelling application of VC involves verifying work offloaded to a peer-to-peer computation network. There are a variety of projects and frameworks built to support the operation of peer-to-peer computation networks [3] [4] [5]. At most, these protocols may have some protections against independently misbehaving nodes, but rarely build defences against nodes capable of collusion. Work duplication is the validation mechanism of choice, with a majority consensus forming the result. This does not protect against the possibility of collusion, but beyond this, work duplication is clearly wasteful. Each program has to be executed at least twice to see if consensus was reached; more if a dispute has to be resolved.

Without a viable mechanism to check the output of an outsourced computation, one of the few ways to build faith in the validity of a result is through the weaker notions of brand reputation and contract law. Along with the economic benefit of a large scale data centre [1], it must be the case that one of the reasons cloud computing has become so successful is that large brand names can garner more trust than smaller operations or anonymous donors. Trusting a computational task to a cloud provider with a good brand reputation is considerably more compelling than a series of results provided by donated resources of unknowable repute.

The ability to formally verify the correctness of a result is clearly a compelling prospect, if only it could be done computationally cheaply enough. This paper presents a mechanism that allows a computationally restricted Verifier to check the results of a remote computation from a more powerful but untrusted Prover with minimal additional computational complexity to either party. Although our asymptotic complexity is worse in many cases than the related work, there are a number of situations in which our protocol excels. These include programs that will only be executed and verified once, and the evaluation of polynomials over large integers.

We leave a discussion on the related VC work until towards the end of the paper. The work presented here is a completely new approach to the problem of VC, and as such understanding the construction does not require an understanding of the mechanics of the related work.

Alex Dalton is with Imperial College London, Email: akd19@ic.ac.uk.
David Thomas is with the University of Southampton, Email: d.b.thomas@southampton.ac.uk.
Peter Cheung is with Imperial College London, Email: p.cheung@imperial.ac.uk.

1.1 Our Contributions

In this work we present a novel VC scheme. Our contributions can be summarised as:

- Specifying a novel VC scheme, which, to the best of our knowledge, is the first VC scheme to be:
  - Proofless, the raw computational output of the target program is verified without any additional data from the Prover.
  - Built from the hardness of integer factoring.
- Prove our scheme satisfies the standard VC properties of *Completeness*, *Soundness*, and is always *Outsourceable.*

For large composite number $N = \prod P_N$, arithmetic circuit to be verified C_N, our scheme has the following additional properties:

- A small one-time setup KGen, independent of the size of the target computation; $\mathcal{O}(|P_N|)$. Which can be shared between programs to be verified.
- An insignificantly small constant cost $\mathsf{ProbGen}$.
- A verification step, Vf, with computational complexity a constant factor of the computational complexity of the target program, $\frac{\mathcal{O}(|C|)}{|P_N|}$
- A zero additional Prover computational complexity as compared with unverified computation, $\mathcal{O}(|C|)$.
- Zero additional bandwidth as compared with unverified computation.
- Requires only implementing the KGen and Vf functions to add VC functionality to an existing unverified remote computation framework.

The downside of our scheme in relation to the related work is worse asymptotic properties for the verification step. For programs executed many times, our scheme is almost certainly not the most suitable approach. However, where related work requires repeat executions of the same program to be considered *Outsourceable*, we are *Outsourceable* with only a single execution.

1.2 Notation

In this paper we will refer to the local client that has issued the work as the Verifier, and the remote server performing the outsourced execution as the Prover. Standard cryptographic notation is used throughout; pk and sk for any public and secret keys, and a security parameter λ. $\mathsf{negl}(\lambda)$ is any function that grows negligibly in λ. $\mathsf{poly}(x)$ is any polynomial function of x.

Usually, capital letters are used to represent sets, with the size of a set A given by $|A|$. There are a few cases where capital letters do not refer to sets, as to better align our work with conventions in related fields. The first case is for the large composite integer N. The second is for B and C which denote binary and arithmetic circuits respectively. Our work uses arithmetic circuits, but we reference binary circuits when discussing related work. We reserve the letter R to represent a special a ring. R_x is the ring defined by the integers modulo x. When an arithmetic circuit C operates over ring R_x we use the notation C_x.

Generic functions are given the letter f, whereas named functions are written in a specific $\mathsf{Function}$ font. In function signatures $\leftarrow$ denotes output, and in pseudo-code descriptions it indicates an assignment. $\xleftarrow{\$}$ is used where a random element should be sampled from a domain. $\equiv$, and $=$ represent congruence and equality respectively, the former usually implying a modular reduction.

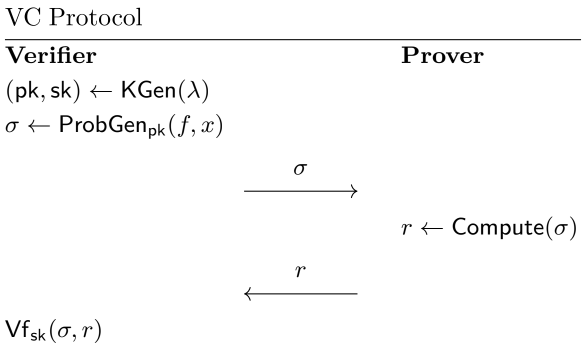

VC Protocol
Verifier
Prover
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$
$\sigma \leftarrow \mathsf{ProbGen}_{\mathsf{pk}}(f, x)$
σ
$r \leftarrow \mathsf{Compute}(\sigma)$
r
$\mathsf{Vf}_{\mathsf{sk}}(\sigma, r)$

Figure 1. VC protocol overview.

2 Background

VC protocols define a set of functions which, when orchestrated into a protocol can be used to guarantee the correctness of the result of a computation. Our scheme is a new approach to VC that is built from the integer factoring problem. In our scheme, programs are expressed as arithmetic circuits. First we establish the VC nomenclature, detail the mechanics of arithmetic circuits, followed by background on the integer factoring problem.

2.1 Verifiable Computation

A comprehensive formal definition of the VC problem can be found as part of [6], many of the definitions to follow are adapted from this work. A VC scheme defines the following functions:

- **Key Generation**: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$; generate a public and secret key pair.
- **Problem Generation**: $\sigma \leftarrow \mathsf{ProbGen}_{\mathsf{pk}}(f, x)$; encode arbitrary function f and an input x into problem σ.
- **Compute**: $r \leftarrow \mathsf{Compute}(\sigma)$; process problem σ, producing a result r.
- **Verification**: $\{\top, \bot\} \leftarrow \mathsf{Vf}_{\mathsf{sk}}(\sigma, r)$; verify the result r, given problem σ. $\top$ is an accepted verification, $\bot$ indicates a rejection.

The functions are modified slightly from those presented in [6], as our scheme has some comparatively simple function definitions. This simplification causes subtle changes in the other function signatures, and in the security proofs that follow in this work.

These functions are orchestrated into a protocol between a Verifier and a Prover. Figure 1 provides an overview of this exchange, as it should work when provided definitions for the function signatures above. Key generation, problem generation, and verification are all executed by the Verifier, and compute is executed by the Prover.

A VC scheme must accept correct results of computation, and reject incorrect results. These properties are called *Completeness* and *Soundness* respectively.

```
$\mathsf{VerifyGame}_1(f\ \lambda)$
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$
2 : $(\sigma, r') \leftarrow \mathcal{A}(\mathsf{pk}, f)$
3 : if $\mathsf{Vf}_{\mathsf{sk}}(\sigma, r') = \top \wedge \mathsf{Compute}(\sigma) \neq r'$
4 :     return 1
5 : else
6 :     return 0
7 : endif
```

Figure 2. Verify security game for VC schemes. (1) Generate key pair, (2) send f and pk to adversary, which returns an encoded problem σ and forgery r', (3) the adversary wins if r' is not the correct result of Compute(σ) but is accepted by the Verifier regardless.

Definition 2.1 (Completeness)**.** An honest Prover can convince a Verifier their result is correct with high probability. For function f, data x, encoding $\sigma \leftarrow \mathsf{ProbGen}(f, x)$, and honestly generated result $r \leftarrow \mathsf{Compute}(\sigma)$

$$\Pr[\mathsf{Vf}(\sigma, r) = \top] \approx 1 \tag{1}$$

When considering *Soundness* we are reasoning about the security of a VC scheme in the face of an adversarial Prover. Consider the security game in Figure 2. In this game an adversary, $\mathcal{A}$, is provided with a public key pk and function f, and wins the game if they are able to construct a valid encoded problem σ and forged output r', such that Vf incorrectly returns $\top$. Throughout this work we use the term *forgery* to refer to an incorrect result provided by a malicious Prover. A VC scheme has the *Soundness* property if the probability of $\mathcal{A}$ winning is negligible.

Definition 2.2 (Soundness)**.** For a VC scheme we define the advantage of an adversary in the VerifyGame, as detailed in Figure 2, as:

$$\mathsf{Adv}_{\mathcal{A}}^{\text{verify}}(\lambda) = \Pr[\mathsf{VerifyGame}(\lambda) = 1] \tag{2}$$

A VC scheme is secure for a function f, if for any probabilistic polynomial time adversary $\mathcal{A}$

$$\mathsf{Adv}_{\mathcal{A}}^{\text{verify}}(\lambda) \leq \mathsf{negl}(\lambda) \tag{3}$$

The central challenge when constructing a VC scheme, is to create a scheme in which the total computational burden on a Verifier is less than the fastest time to compute $f(x)$. A scheme with this property is *Outsourceable*.

Definition 2.3 (Outsourceable)**.** For any function f, the total time required to compute ProbGen and Vf is less than the time required to compute $f(x)$ for any x.

2.2 Arithmetic Circuits

As is common in VC work, we express the computation that will be verified as an arithmetic circuit, C. An arithmetic circuit is a directed graph of arithmetic operations. Nodes in this graph represent arithmetic gates, which perform some arithmetic operation on their input data. Figure 3 is a toy example of an arithmetic circuit for a simple program. For a set of inputs the circuit is evaluated by iteratively processing the input to a gate based on the gate's designated operation.

A related computational model often seen in other VC work is a binary circuit, B. These function similarly to arithmetic circuits, except the gates are binary operations, such as `AND` or `OR`. Binary circuits can express comparison operations in fewer gates than arithmetic circuits, but otherwise arithmetic circuits are more concise than binary circuits. The number of wires required to express a program as an arithmetic circuit is approximately four orders of magnitude smaller than the same program represented as a binary circuit [7].

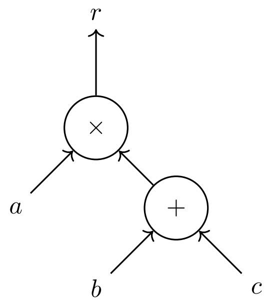


Figure 3. An arithmetic circuit representation of $r = a(b + c)$.

It has been shown that arithmetic circuits are a relatively good model for general computation [8]. In this work programs are expressed as arithmetic circuits over a ring. For many cryptographic operations this is particularly useful as these programs can be naturally expressed as such.

2.3 Integer Factoring Problem

The security of our scheme reduces to security of the integer factoring problem. The difficulty of integer factoring is a widely accepted standard cryptographic assumption. Given large composite integer N, producing any non-trivial factors of N is thought to be computationally intractable [9]. Importantly, while not NP-complete, current estimates place integer factoring as an NP-intermediate problem. There is no polynomial time algorithm for factoring numbers, the current state-of-the-art uses a General Number Field Sieve (GNFS) to achieve subexponential time [10].

Unfortunately integer factoring is not post-quantum secure, as running Shor's algorithm on a quantum computer can factor integers in polynomial time [11]. Current practical limitations on quantum computing [12] indicate a great deal of longevity in cryptosystems built on the difficulty of the integer factoring problem.

The formulation of an instance of the integer factoring problem is based on the current best known algorithms to factor N. To better reason about the security of our scheme we have abstracted the generation of a suitably difficult factoring problem as an oracle,

$$(N, P_N) \leftarrow \mathsf{PrimeFactor}(\lambda) \text{ s.t. } N = \prod_{p \in P_N} p \tag{4}$$

which produces a suitably difficult instance of the integer factoring problem to satisfy security parameter λ.

We use the security game detailed in Figure 4 in later security proofs. The advantage of an adversary in this game is defined as follows:

```
$\mathsf{FactorGame}_1(\lambda)$
1 :  $(N, P_N) \leftarrow \mathsf{PrimeFactor}(\lambda)$
2 :  $a \leftarrow \mathcal{A}(N)$
3 :  if $N/a = b$ for integers $b, a \notin \{1, N\}$
4 :    return 1
5 :  else
6 :    return 0
7 :  endif
```

Figure 4. Factoring security game for the integer factoring problem. (1) Generate suitably hard factorisation problem, (2) send N to the adversary and receive result a, (3) the adversary wins if a is a non-trivial factor of N.

Definition 2.4. For the integer factoring problem we define the advantage of an adversary to FactorGame, detailed in Figure 4, as:

$$\mathsf{Adv}^{\text{factor}}_{\mathcal{A}}(\lambda) = \Pr[\mathsf{FactorGame}(\lambda) = 1] \tag{5}$$

3 New Verifiable Computation Scheme

We now present a new VC scheme built from the integer factoring problem. We start with an intuitive outline of the mechanisms involved, followed by a formal description of the scheme.

3.1 Outline

Given an arithmetic circuit, C_N, over a ring of integers R_N, where N is a large composite number, $N = \prod P_N$. The Prover is sent C_N and some input, x, for evaluation. The Verifier selects one of the products p of N from P_N arbitrarily and independently evaluates the circuit under R_p with the same input. The result of this evaluation is then compared to the result provided by the Prover. The Verifier's result should be congruent to the Prover's result when reduced to the same modulus.

The ability to efficiently fabricate a result that is not the correct output of C_N, yet is still congruent to the correct result modulo an element of P_N, reduces to the ability to efficiently factor N.

3.2 Definition

The Verifier considers a computation they wish to evaluate, C. They generate a set of co-prime numbers $P_N = \{p_1, p_2, \ldots\}$, and their product N, as provided by the integer factoring problem oracle $\mathsf{PrimeFactor}(\lambda)$. P_N is kept secret, and N can be freely shared.

$$(N, P_N) \leftarrow \mathsf{PrimeFactor}(\lambda) \tag{6}$$

The generation of N need only be done once and can be shared between verifications of different arithmetic circuits.

N is used to define the ring R_N, the set of integers modulo N. The Verifier notes that their arithmetic circuit C should be evaluated in R_N, denoted as C_N. Next, the Verifier bundles the arithmetic circuit C_N and the input data x into a tuple σ and sends it to the Prover for evaluation.

$$\sigma \leftarrow (C_N, x) \tag{7}$$

The Prover extracts C_N and x from σ, and calculates the result of evaluating C_N with input x, r. Then sends r back to the Verifier.

$$(C_N, x) \leftarrow \sigma \tag{8}$$
$$r \leftarrow C_N(x) \tag{9}$$

The Verifier selects an arbitrary $p \in P_N$ and evaluates $C_p(x)$.

$$r_p \leftarrow C_p(x) \tag{10}$$

The Verifier checks the congruence $r \equiv r_p \bmod p$ and if true, accepts the result from the Prover.

This procedure is expressed as definitions for the standard verifiable computation functions in Figure 5. Interestingly, a great deal of the work done by the Verifier is not dependant on the output of the Prover, and therefore can be executed asynchronously. The asynchronous exchange between Prover and Verifier is shown diagrammatically in Figure 6.

```
$\mathsf{KGen}(\lambda)$
  $(N, P_N) \leftarrow \mathsf{PrimeFactor}(\lambda)$
  return $(N, P_N)$
```

```
$\mathsf{ProbGen}_N(C, x)$
  $\sigma \leftarrow (C_N, x)$
  return $\sigma$
```

```
$\mathsf{Compute}(\sigma)$
  $(C_N, x) \leftarrow \sigma$
  return $C_N(x)$
```

```
$\mathsf{Vf}_{P_N}(\sigma, r)$
  $(C_N, x) \leftarrow \sigma$
  $p \in P_N$
  $r_p \leftarrow C_p(x)$
  if $r \equiv r_p \bmod p$ then
    return $\top$
  endif
  return $\bot$
```

Figure 5. The KGen, ProbGen, Compute, and Vf function definitions for our scheme, where KGen has access to an oracle capable of formulating the integer factoring problem for a given security parameter, PrimeFactor(λ).

4 Evaluation

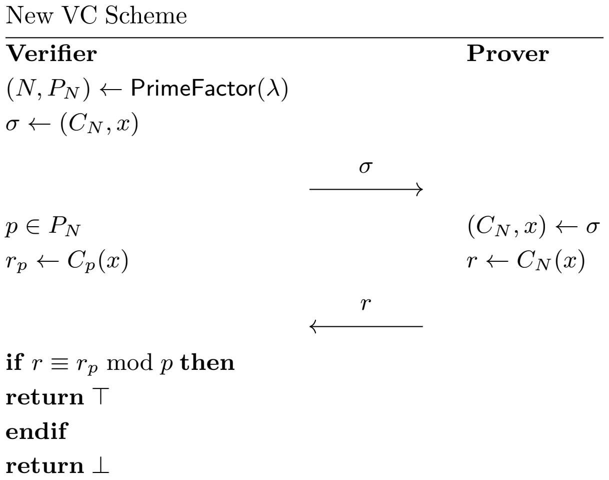
In this section we prove properties for our VC protocol relating to security, and computational and bandwidth complexity.

4.1 Completeness

Theorem 4.1 (Completeness)**.** An honest Prover can convince the Verifier of a correct result r to the computation $C_N(x)$ with probability

$$\Pr[\mathsf{Vf}(r) = \top] = 1 \tag{11}$$

This proof is omitted for brevity. The completeness of our scheme can be trivially derived by assuming both the Prover and Verifier act honestly and conform to the protocol.



```
New VC Scheme
Verifier                                   Prover
$(N, P_N) \leftarrow \mathsf{PrimeFactor}(\lambda)$
$\sigma \leftarrow (C_N, x)$
                      $\sigma$
                 $\longrightarrow$
$p \in P_N$                                $(C_N, x) \leftarrow \sigma$
$r_p \leftarrow C_p(x)$                    $r \leftarrow C_N(x)$
                      $r$
                 $\longleftarrow$
if $r \equiv r_p \bmod p$ then
return $\top$
endif
return $\bot$
```

Figure 6. Our VC protocol overview. This can be derived by applying the function definitions in Figure 5 to the exchange detailed in Figure 1 with some of the Vf functionality pre-calculated by the Verifier as they await r from the Prover.

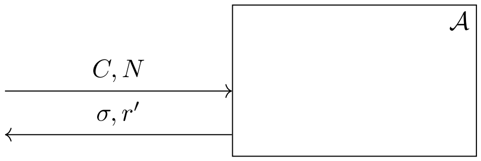


Figure 7. Adversary $\mathcal{A}$; capable of producing forged results r' in our VC scheme to VerifyGame, as defined in Figure 2, with advantage $\mathsf{negl}(\lambda) < \mathsf{Adv}^{\mathsf{vf}}_{\mathcal{A}}(\lambda) < 1$.

4.2 Soundness

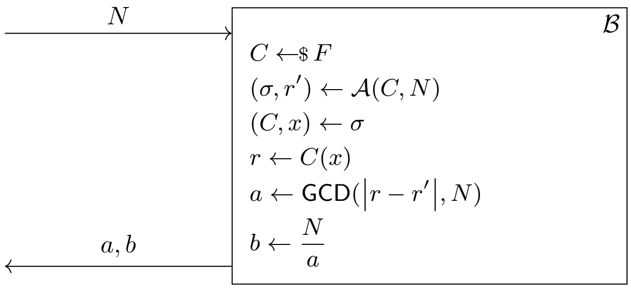
The intuition behind the soundness of our scheme is that fabricating a fake result $r' \neq r$ with a non-negligible probability of being congruent to r under a random element of P_N requires knowledge of how N decomposes into P_N. We prove that given a forgery $r' \neq r$ that is undetectable when reduced by any elements from the set Q, where Q is an arbitrary non-empty subset of P_N, we can extract $\prod Q$, a non-trivial factor of N. Proving that generating a forgery is at least as hard as factoring N.

Theorem 4.2 (Soundness)**.** The advantage of a computationally bounded adversary in the verification game, $\mathsf{Adv}^{\mathsf{vf}}_{\mathcal{A}}(\lambda)$ is at most that of the advantage of a computationally bounded adversary in the integer factoring problem, $\mathsf{Adv}^{\mathrm{factor}}_{\mathcal{B}}(\lambda)$.

Proof. We can construct an efficient adversary to the integer factoring problem, given an efficient adversary against our VC scheme. Figure 7 depicts a hypothetical black-box adversary $\mathcal{A}$, capable of efficiently creating forgeries for our VC scheme. For a given configuration, the public components are passed in, (C, N), and the adversary returns (σ, r'), an encoded problem and forged result, with advantage

$$\mathsf{negl}(\lambda) < \mathsf{Adv}^{\mathsf{vf}}_{\mathcal{A}}(\lambda) < 1 \tag{12}$$

As an aside, constructing a proof that a perfect adversary against our scheme is impossible is trivial given the chinese remainder theorem. Therefore $Q \neq P_N$, and $\mathsf{Adv}^{\mathsf{vf}}_{\mathcal{A}}(\lambda) < 1$.



```
N →
$\mathcal{B}$
$C \leftarrow\!\$\; F$
$(\sigma, r') \leftarrow \mathcal{A}(C, N)$
$(C, x) \leftarrow \sigma$
$r \leftarrow C(x)$
$a \leftarrow \mathsf{GCD}(|r - r'|, N)$
$b \leftarrow \frac{N}{a}$
← a, b
```

Figure 8. Adversary $\mathcal{B}$; capable of calculating two non-trivial factors to any large integer, given the existence of Adversary $\mathcal{A}$.

Given adversary $\mathcal{A}$, it is straightforward to construct adversary $\mathcal{B}$, detailed in Figure 8. Which, with a single query to $\mathcal{A}$, can efficiently recover two non-trivial factors of N. In detail, when provided with a number N we wish to factorise, $\mathcal{B}$ chooses a suitable arbitrary arithmetic circuit C from the space of possible circuits F. The circuit chosen does not matter, so it is in our best interests to select a simple one. C and N are passed into adversary $\mathcal{A}$ which returns σ and r'. x is extracted from σ and used to calculate $r \leftarrow C_N(x)$ from which we extract two non-trivial factors of N by finding the greatest common divisor of N and $|r - r'|$. A complete list of prime factors can later be found with recursive calls to adversary $\mathcal{B}$.

Using this construction, when provided with any viable adversary to our VC scheme, it is possible to construct an adversary in the integer factoring game with advantage $\mathsf{Adv}^{\mathrm{factor}}_{\mathcal{B}}(\lambda) = 1$.

$$\mathsf{negl}(\lambda) < \mathsf{Adv}^{\mathrm{verify}}_{\mathcal{A}}(\lambda) \Rightarrow \mathsf{Adv}^{\mathrm{factor}}_{\mathcal{B}}(\lambda) = 1 \tag{13}$$

$$\therefore \mathsf{Adv}^{\mathrm{verify}}_{\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathrm{factor}}_{\mathcal{B}}(\lambda) \tag{14}$$

□

4.3 Outsourceable

We will now consider the computational complexity of our VC scheme and show it is *Outsourceable.*

4.3.1 Setup

The one time setup cost of the scheme involves generating an instance of the integer factoring problem. Assuming suitable Ns are not overly sparse this step has computational complexity

$$\mathcal{O}(|P_N|) \tag{15}$$

This is derived from making a linear number of calls to an oracle $\mathsf{PrimeFactor}(\lambda)$, until a suitable N is generated. The main cost being the generation of prime numbers and their subsequent multiplication together. The setup for this scheme is decoupled from the program being validated. As long as the secret key P_N is not leaked there is no need to rerun this step.

4.3.2 Prover

As per the definition of $\mathsf{Compute}$, the Prover executes programs as if they were not being verified, without any change.

$$\mathcal{O}(|C|) \tag{16}$$

It is worth noting here that related schemes require a Prover does an amount of work that is a multiplicative factor of the effort required to evaluate circuit C. In our scheme the Prover need only evaluate the vanilla circuit C and so does dramatically less work than in the related work.

4.3.3 Verifier

Our ProbGen does nothing except pack the arithmetic circuit C_N, and the input data x into a tuple. This is only done to conform with the related work. Therefore, it is safe to assume this step has an insignificantly small constant cost.

The more sizeable contribution to the computational complexity of the Verifier is the Vf function. It is common practice to use a Residue Number System (RNS) to accelerate modular arithmetic [13]. This would suggest that the computational complexity of an RNS can be no worse than computing the same operation in a more conventional number system.

The work done during Vf amounts to performing a a tiny subset of the computations required to evaluate the arithmetic circuit via RNS. This is because the work done is the exact same as if we were evaluating a single element of the circuit decomposed into the RNS. It would be safe to assume only performing a subset of the calculations will require computational effort proportional to the size of the subset. The computational burden of Vf is therefore a factor of $|P_N|$ smaller than the cost to evaluate the entire circuit.

$$\frac{\mathcal{O}(|C|)}{|P_N|} \tag{17}$$

As with the complexity of Compute it is worth noting here that the complexity of Vf is derived from the raw computational complexity of the circuit C, with no additional multiplicative factors.

Theorem 4.3 (Outsourceable)**.** As long as $|P_N| > 1$ the time complexity for the Verifier is less than the time taken to evaluate arithmetic circuit C.

The proof is trivial, given insignificant ProbGen complexity, and the Vf complexity in Equation 17, and is therefore omitted.

4.4 Performance Implications of the Factoring Problem

The belief that integer factoring is computationally intractable is a standard cryptographic assumption. However, algorithms for factoring integers are continuously being developed with incremental gains in factoring performance. Because of this instability we intentionally keep references to the underlying factoring problem vague, and abstract the problem generation to an oracle. This abstraction protects us from any minor developments in integer factoring.

That being said, the properties of the factoring problem have a direct impact upon our VC scheme. As shown in Theorem 4.3, as long as N has more than one factor our scheme is *Outsourceable* and the amount of work done by the Verifier shrinks as the number of factors of N increases. This relies on the fact that the individual complexity of evaluating the arithmetic gates is simpler if the operands are smaller. The gain in computing C_p over C_N is only valuable if p is substantially smaller than the expected intermediate results. Otherwise, it is identical to calculating C_N. In which case our scheme is no better than naively double checking the result.

The difficulty of a given instance of the integer factoring problem is largely dictated by the size of N's smallest factors. The smaller the smallest factors the easier it is to find it's factors. In order to speculate on the real-world performance of our system, consider the RSA cryptosystem. RSA is built on the same assumption as our scheme, the computational intractability of integer factoring. In RSA N is thousands of bits long. Similarly p must be many hundreds of bits long. For many conventional computations evaluating C_N and C_p is identical.

4.5 Proof Size

As compared with unverified computation, our scheme requires no extra communication. The only messages passed between the Prover and the Verifier are a description of the arithmetic circuit, C_N, it's input, x, and the result, r. As such nothing extra is sent as compared with unverified computation.

$$|C| + \log x + \log r \tag{18}$$

This leads to an interesting quality of our scheme; because the workload sent to the Verifier is unchanged, when compared with unverified computation, there is no reason that the Verifier would know they are engaged in a VC scheme. The prover is effectively blind to their participation in a VC scheme.

To the best of our knowledge this is the first verifiable computation scheme to be proofless. In that, no additional information need be sent from the Prover to the Verifier other than the raw output of the target computation.

5 Related Work

A great deal of work has already been done attempting to create a suitable mechanism to check the results from a remote computation. These solutions range from concrete number theoretic constructions through to economic solutions wherein the cost of sending incorrect results is set discouragingly high [14] [15].

5.1 Auditing

The first concrete anti-cheat mechanisms were built around the idea of performing an audit on the work process. During execution, regular stack traces can be saved and returned with the results. Random pairs of consecutive stack traces can then be validated by executing the program between these two checkpoints and validating the latter stack trace as correct given the former [16]. Faith is built in the correctness of the answer by checking enough of the pairs provided by the Prover.

An interesting idea for detecting if a program was run at all involves introducing random interactive processes at compile time [17]. In this work automated randomised interactive elements are added to a program during compilation. At points known to the Verifier during the program execution the Prover will interact with the Verifier in a way that suggests they have reached these checkpoints. Such a tool is not designed to defend against a powerful malicious adversary capable of running the program and sending falsified results regardless. Such an adversary may have motives beyond circumventing the cost of executing the program.

5.2 Trusted Platform Modules

A pragmatic solution to verifying the execution of a program is to produce a golden hardware and software state and then create a mechanism for a remote computer to prove that they are in the golden state when performing a computation. Trusted Platform Modules (TPMs) have allowed this style of validation to be possible, with attestation procedures that can provide witness to the entire hardware/software stack [18]. Although this is a viable solution, it does not provide much flexibility in it's implementation. To build a solution where remote parties are attesting with TPMs requires a sufficiently benevolent and trusted device vendor willing to build these modules on the behalf of participants, or someone with enough resources at their disposal to do it themselves. The creation of a golden state requires trusting every party in the hardware and software development life cycle.

5.3 Number Theoretic Solutions

A more rigorous approach to VC involves formulating programs in such a way that the capability to cheat without detection is reduced to known hard problems.

5.3.1 Program Checking

Early number theoretic solutions concerned themselves with automated checking of results given some possibility of non-malicious errors. The field of *Program Checkers* builds this functionality for a suite of standard problems. For specifically chosen problems, such as numerical functions [19], or sorting and greatest common divisor [20], the authors construct bespoke checking programs that certify a program was executed correctly without bugs. Program checkers are designed to detect accidental errors in a computation and so are not secure against a malicious adversary willing to invest computational effort to create forgeries.

5.3.2 Efficient but Problem Specific Solutions

Some problems have specific VC constructions which can provide a high guarantee of correctness, with a high degree of security and approachable resource requirements for both the Verifier and the Prover. Unfortunately, these constructions are bespoke and only work for a specific problem, or subset of problems.

By pairing the somewhat-homomorphic BGN encryption scheme [21] with a publicly verifiable computation scheme across polynomials [22] a privacy preserving construction can be devised that allows users to validate the results of cloud computations across their health data [23].

Similar constructions exist for verifying searches over encrypted data [24], modular exponentiation operations [25], polynomial function evaluation [26], and matrix multiplication [27].

5.3.3 Probabilistically Checkable Proofs (PCPs)

An early attempt at a system for verifying general computation involved using PCPs [28] [29]. PCPs are verifiable in poly-logarithmic time in the size of the proof; unfortunately proofs tend to be very long. There has been some work modifying these schemes to be more efficient by tailoring them to specific given problems [30]. Even with more modern cryptographic tools the computational break-even point is rarely reached. One modern example uses argument systems, a variant of PCPs in which the Prover answers queries from the Verifier interactively [7]. Modern PCPs have a great deal of implementation complexity, deterring adoption and making a bug-free implementation hard.

Another of the first fully actualised general VC schemes involved combining Yao's Garbled Circuit construction [31] with a fully homomorphic encryption scheme to allow reuse of the previously-single-use VC properties of the garbled circuits [6] [32]. By decoupling the setup from the input data, the expensive setup can be amortised across multiple executions of the program with different input data.

Multi-Party Computation (MPC) is a closely related field of inquiry to VC, in which two groups work together to securely compute a function on a shared input without revealing the input data to the other party. A natural question is, can we construct VC from MPC. The construction involves using an MPC scheme to compute the signature of the output of an outsourced function [33].

Some PCP-based VC schemes are close to practical for a reduced domain of possible programs [34], albeit with some non-standard assumptions. Verification is cheap, but the setup costs are very close to the computation cost of the target program. As such program encodings must be reused with new inputs to reduce the average computational burden on the Verifier per execution.

Interactive Oracle Proofs (IOPs) are a generalisation of PCPs. These interactive proof systems have been shown to allow for much more efficient program verification of boolean circuits [35]. In this case, more succinct arguments are created by the removal of a requirement to expensively encode the entire computation.

5.3.4 Succinct Non-Interactive Proofs

Succinct non-interactive proofs are a subset of VC. The aim to improve the asymptotic properties of existing Zero Knowledge Proof (ZKP) schemes to provide the performance properties required by a VC scheme. One of the most prolific of these is Bulletproofs [36]. Bulletproofs have admirable asymptotic properties, which are linear in the circuit size, but expensive cryptographic operations mean that in practice they are rarely viable, often taking orders of magnitude longer than vanilla computation. Some more recent work, Libra [37], has massively improved practical performance, but only produces succinct arguments for specifically structured arithmetic circuits.

5.4 Identified Problems

At best, the existing solutions front-load the additional computational complexity to a setup phase, and then have very efficient verification. This allows for amortised complexity over multiple executions of the program, with different input data.

As shown in Table 1, schemes tend to have a large computational cost for the Prover. All schemes add, at minimum, a multiplicative factor to the Prover workload over vanilla computation. All related work, save Pinocchio [34], have very large proofs. There is a clear gap to be explored with a VC scheme that is extremely cheap to setup with tiny proofs, possibly at the cost of being more expensive to verify. The development

Table 1
Our VC scheme compared with the related work. x is the input to arithmetic circuit C with depth d and result r. Schemes designed to work with boolean circuits use B instead of C.

	Setup	Prover	Verifier	Proof Size
[34]	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(\log x + \log r)$	$\mathcal{O}(1)$
[35]	$\mathcal{O}(\lvert B\rvert)$	$\mathcal{O}(\lvert B\rvert)$	$\mathcal{O}(\log x + \log\lvert B\rvert)$	$\mathcal{O}(\log\lvert B\rvert)$
[36]	0	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(\log\lvert C\rvert)$
[37]	$\mathcal{O}(\log x)$	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(d\log\lvert C\rvert)$	$\mathcal{O}(d\log\lvert C\rvert)$
None	0	$\mathcal{O}(\lvert C\rvert)$	0	0
Naive	0	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(\lvert C\rvert)$	0
This	$\mathcal{O}(\lvert P_N\rvert)$	$\mathcal{O}(\lvert C\rvert)$	$\frac{\mathcal{O}(\lvert C\rvert)}{\lvert P_N\rvert}$	0

of a scheme with minimal additional Prover overhead would likewise offer a viable alternative to the existing work for settings in which the computational burden on the Prover need be as small as possible; such as in cloud computing workloads.

Despite the rich body of existing work on VC, the most telling evidence that more work is required is that modern volunteer distributed computation deployments, such as Folding@home [38], use naive work duplication to validate outsourced computation.

5.5 Comparison To Related Work

From Table 1 our biggest improvement over the related work is the massively reduced initial setup, which is decoupled from the computation being verified. This, as well as the proof size of zero.

We forgo some additional properties present in related work; such as program, input, or output secrecy; zero knowledge; or public verification; but our scheme has some interesting properties in it's own right. First, the Prover is blind to the fact the computation is being verified as part of a VC scheme. This is because no additional information passes between the Prover and Verifier than would otherwise occur when engaged in a non-verified remote computation protocol. Our scheme is completely proofless. As the Prover operates on unmodified arithmetic circuits, it is impossible for them to do any less work. Therefore, our scheme is Prover-optimal in the arithmetic circuit setting.

Another notable property is the extreme low latency of our system. The evaluation of $C_p(x)$ during Vf can be done before r is received from the Prover. This approach is shown in Figure 6. As such, the time taken to output a verified result is the one-time cost to execute KGen, plus $\mathsf{Compute}$, plus the time taken to perform a single modular reduction by p. It is hard to imagine a VC protocol which could take less time from start to finish. This is in comparison to the related work where the Verifier must await the results and the proof from the Prover before engaging in Vf.

Thirdly, our scheme is relatively easy to implement. An existing client/server computation system only need implement KGen and Vf, as $\mathsf{Compute}$ is unchanged and $\mathsf{ProbGen}$ only packs data into a tuple.

6 Conclusion

We described the first VC construction built from the hardness of factoring integers. This construction only used standard cryptographic assumptions and, as per Theorems 4.1, 4.2, and 4.3 the scheme presented in this work has the *Completeness* and *Soundness* properties and is *Outsourceable*. Vf is more expensive than in related work but our scheme has a one-time setup independent of the computation. Despite the more expensive Vf our setup cost savings mean our scheme is always *Outsourceable* even for only a single execution. Unfortunately, the degree of outsourceableness of our scheme is intrinsically linked to the approximate operand size of the programs being verified and the security of the scheme. In the worst possible case the Verifier does as much work as if they were duplicating the work to check the program output. As such the best performance of the system is when operating with large integer operands much larger than p.

While other schemes have better asymptotic properties, it is plausible that our scheme has better computational cost when verifying low-volume programs due to the zero cost setup phase. Our scheme provides some novel properties, such as prooflessness, low latency, and relative implementation ease.

References

[1] "Cloud computing services - amazon web services (aws)." [Online]. Available: https://aws.amazon.com/

[2] D. Anderson, "Boinc: a system for public-resource computing and storage," in *Fifth IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 4–10.

[3] A. Baratloo, M. Karaul, Z. Kedem, and P. Wijckoff, "Charlotte: Metacomputing on the web," *Future Generation Computer Systems*, vol. 15, no. 5, pp. 559–570, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X99000096

[4] T. Brecht, H. S. Sandhu, M. Shan, and J. Talbot, "Paraweb: towards world-wide supercomputing," in *EW 7*, 1996.

[5] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer, "Atlas: an infrastructure for global computing," in *EW 7*, 1996.

[6] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 465–482.

[7] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making argument systems for outsourced computation practical (sometimes)," in *Network & Distributed System Security Symposium (NDSS)*, February 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/making-argument-systems-for-outsourced-computation-practical-sometimes/

[8] J. von zur Gathen, "Algebraic complexity theory," *Annual Review of Computer Science*, vol. 3, no. 1, pp. 317–348, 1988. [Online]. Available: https://doi.org/10.1146/annurev.cs.03.060188.001533

[9] A. K. Lenstra, *Integer Factoring*. Boston, MA: Springer US, 2011, pp. 611–618.

[10] J. P. Buhler, H. W. Lenstra, and C. Pomerance, "Factoring integers with the number field sieve," in *The development of the number field sieve*, A. K. Lenstra and H. W. Lenstra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 50–94.

[11] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[12] M. Amico, Z. H. Saleem, and M. Kumph, "Experimental study of shor's factoring algorithm using the ibm q experience," *Phys. Rev. A*, vol. 100, p. 012305, Jul 2019. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.100.012305

[13] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, *Residue number system arithmetic: modern applications in digital signal processing*. IEEE press, 1986.

[14] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya, "Incentivizing outsourced computation," in *Proceedings of the 3rd International Workshop on Economics of Networked Systems*, ser. NetEcon '08. New York, NY, USA:

Association for Computing Machinery, 2008, p. 85–90. [Online]. Available: https://doi.org/10.1145/1403027.1403046

[15] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Topics in Cryptology — CT-RSA 2001*, D. Naccache, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 425–440.

[16] F. Monrose, P. Wyckoff, and A. D. Rubin, "Distributed execution with remote audit," in *NDSS*, 1999.

[17] S. Yang, A. R. Butt, Y. C. Hu, and S. P. Midkiff, "Lightweight monitoring of the progress of remotely executing computations," in *Languages and Compilers for Parallel Computing*, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 319–333.

[18] S. Alsouri, Ö. Dagdelen, and S. Katzenbeisser, "Group-based attestation: Enhancing privacy and management in remote attestation," in *Trust and Trustworthy Computing*, A. Acquisti, S. W. Smith, and A.-R. Sadeghi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 63–77.

[19] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 549–595, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/002200009390044W

[20] M. Blum and S. Kannan, "Designing programs that check their work." *J. ACM*, vol. 42, pp. 269–291, 01 1995.

[21] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of Cryptography*, J. Kilian, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 325–341.

[22] C. Papamanthou, E. Shi, and R. Tamassia, "Signatures of correct computation," in *Theory of Cryptography*, A. Sahai, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 222–242.

[23] L. Guo, Y. Fang, M. Li, and P. Li, "Verifiable privacy-preserving monitoring for cloud-assisted mhealth systems," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 1026–1034.

[24] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li, "Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 3025–3035, 2014.

[25] M. Bellare, J. A. Garay, and T. Rabin, "Fast batch verification for modular exponentiation and digital signatures," in *Advances in Cryptology — EUROCRYPT'98*, K. Nyberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 236–250.

[26] D. Boneh and D. M. Freeman, "Homomorphic signatures for polynomial functions," in *Advances in Cryptology – EUROCRYPT 2011*, K. G. Paterson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 149–168.

[27] M. J. Atallah and K. B. Frikken, "Securely outsourcing linear algebra computations," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 48–59. [Online]. Available: https://doi.org/10.1145/1755688.1755695

[28] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy, "Checking computations in polylogarithmic time," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 21–32. [Online]. Available: https://doi.org/10.1145/103418.103428

[29] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," *Proceedings of the fortieth annual ACM symposium on Theory of computing*, 2008.

[30] G. Cormode, M. Mitzenmacher, and J. Thaler, "Practical verified computation with streaming interactive proofs," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 90–112. [Online]. Available: https://doi.org/10.1145/2090236.2090245

[31] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, 1982, pp. 160–164.

[32] K.-M. Chung, Y. Kalai, and S. Vadhan, "Improved delegation of computation using fully homomorphic encryption," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 483–501.

[33] B. Applebaum, Y. Ishai, and E. Kushilevitz, "From secrecy to soundness: Efficient verification via secure computation," in *Automata, Languages and Programming*, S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 152–163.

[34] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2013, best Paper Award. [Online]. Available: https://www.microsoft.com/en-us/research/publication/pinocchio-nearly-practical-verifiable-computation/

[35] N. Ron-Zewi and R. D. Rothblum, "Proving as fast as computing: Succinct arguments with constant prover overhead," in *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1353–1363. [Online]. Available: https://doi.org/10.1145/3519935.3519956

[36] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," Cryptology ePrint Archive, Paper 2017/1066, 2017, https://eprint.iacr.org/2017/1066. [Online]. Available: https://eprint.iacr.org/2017/1066

[37] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Advances in Cryptology – CRYPTO 2019*, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 733–764.

[38] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–8.

Alex Dalton received an MEng degree in Computer Science from the University of Bristol, and is currently a PhD student in the Circuits and Systems research group in the department of Electrical & Electronic Engineering at Imperial College London. His research interests are primarily in the fields of cryptographic protocols and distributed computation.

David Thomas studied Computer Science as an undergrad at the Dept. of Computing in Imperial College London, then did his PhD in digital architectures in the same department. After 5 years as a researcher associate and then research fellow, in 2010 he moved to the Dept. of Electrical and Electronic Engineer at Imperial as a Lecturer, then Senior Lecturer. In 2021 he joined the Electronics and Computer Science Dept. at the University of Southampton as a Professor. Both his research and teaching interests are at the intersection of software and hardware, particularly in the interaction and relationships between programming languages, algorithms, computer architecture and digital implementation.

Peter Cheung is Professor of Digital Systems at Imperial College London, splitting his time between the Department of Electrical & Electronic Engineering, and Dyson School of Design Engineering. Together with Professor Wayne Luk in Department of Computing, he established one of the strongest research groups in the area of Field Programmable Gate Arrays (FPGAs) in the UK. His research in reconfigurable systems and technology include architecture, variability mitigation, reliability issues, high-level synthesis and tools, and various application area for FPGAs.