Efficient Verification of the Wesolowski Verifiable Delay Function for Distributed Environments

Vidal Attias[1][0000−0001−5095−4740], Luigi Vigneri[1], and Vassil Dimitrov[1,2]

[1] IOTA Foundation, Pappelallee 78/79, 10437 Berlin, Germany
https://www.iota.org
{vidal.attias,luigi.vigneri,vassil}@iota.org
[2] Calgary University, Calgary, AB T2N 1N4, Canada

Abstract. Verifiable Delay Functions (VDFs) are a set of new cryptographic schemes ensuring that an agent has spent some time (evaluation phase) in a unparalleled computation. A key requirement for such a construction is that the verification of the computation's correctness has to be done in a significantly shorter time than the evaluation phase. This has led VDFs to recently gain exposure in large-scale decentralized projects as a core component of consensus algorithms or spam-prevention mechanisms. In this work, due to the increasing relevance and the lack of literature, we will focus on the optimization of the verification phase of Wesolowski's VDF and provide a three-axis of improvement concerning multi-exponentiation computation, prime testing techniques, and hashing tricks. We will show that our optimizations reduce the computation time of the verification phase between 12% and 35% for the range of parameters considered.

1 Introduction

After more than two decades since the emergence of commercial services on the Internet [38] and following an increasing centralization of data by governments and large private companies, decentralized projects promise to grant more control over privacy and personal information. Enabling access to services to a large scale of users, humans, or potentially Internet of Things (IoT) devices, requires managing large streams of messages and inescapable loads of spamming, would it be accidental or mischievous. However, spam-prevention mechanisms in such decentralized settings require novel approaches and it essentially boils down to asking users to pledge a scarce resource they own proportionally to their use of the network. Such resources include money, computational power, time, identity, or a certain notion of reputation in the network [10, 17]. Using time as a spam-prevention mechanism dates back to the late 90s with the Hashcash system [4] that prevented e-mail spamming by requiring senders to solve a small cryptographic puzzle (Proof of Work) consisting in finding the nonce corresponding to a hashing function's output. This idea will be used in the founding paper of the *blockchain* [30] and will be the cornerstone of many Distributed Ledger Technology (DLT) projects.

This digital revolution is hindered by various structural flaws of DLTs, with the major drawback being the non-scalability of most blockchain-based projects[3]. This shortcoming prevents many potential use cases and, additionally, these solutions imply high transaction fees which are not compatible with a future filled with IoT devices: use cases involving high-transaction throughput include *autonomous vehicles* [20, 24] posting updates on their state or paying a toll or a parking, *supply chains* [37] involving multiple partners that need a source of trust in their relations, *digital monetary systems* [14, 23] and so on. Fortunately, some DLTs are moving forward aiming to be the backbone of these future high-performance, decentralized networks: among them, IOTA [35], Ethereum 2.0 [25], Bitcoin's Lightning off-chain protocol [34], Polkadot [40]. These projects aim to eliminate the need for expensive Proof of Work, sometimes replacing it with a *Verifiable Delay Function* (VDF) [6, 13, 33, 39] as a spam-prevention mechanism [2] or as a core component of the consensus protocol [11].

VDFs were introduced in 2018 by Boneh et al. [6] where the authors formally defined functions that provably take some sequential steps to compute, hence some time. At a high level, VDFs can be seen as a Proof of Work that cannot be parallelized. As an example, an RSA-based VDF will require to solve $y = x^{2^\tau} \mod N$, which can only be done efficiently by performing τ squaring and no parallel algorithm is known to solve this problem. In this paper, we will look at one of the best candidates for spam prevention, namely the *Wesolowski's efficient VDF* [39] which ensures small verification times and low communication overhead [19]. Our main contribution consists of a thorough study of the verification phase. Fast verification is critical when VDF is used as an anti-spam mechanism in DLTs since invalid messages have to be detected and discarded rapidly to avoid harming the performance of the system. This study is focused on the optimization of the three most time-consuming parts of the verification algorithm: *i)* modular multi-exponentiation, *ii)* prime testing algorithm and *iii)* hash function computation. In this paper, we will provide a theoretical analysis of each part and will demonstrate, through experiments on real devices, that important improvements can be achieved with realistic parameters. To the best of our knowledge, this is the first work of this kind as the VDF literature is either focused on the cryptographic theory of VDFs or the optimization of the evaluation phase.

The rest of the paper is organized as follows. First, Section 2 we will introduce VDFs and their usage, and describe Wesolowski's construction. Then, in Section 3 we will show how using double-exponentiation algorithms significantly improve the verification time of Wesolowski's construction. Next, Section 4 will be dedicated to prime testing and Section 5 will discuss hash functions to solve a specific part of this operation. Finally, Section 6 concludes the paper.

[3] In 2022, the throughput of the two major blockchains, *Bitcoin* and *Ethereum*, is limited to just a few transactions per second.

2 Verifiable Delay Functions

2.1 Context

Using computer programs to verify that some time has elapsed between two events has been a long-sought-after grail in the cryptography world. This problem stems from the difficulty of trusting foreign hardware and executions and the lack of trusted time beacons. In 1993, Timothy May mentioned ideas of how to use *timed-release cryptographic protocols* [28]. The idea at that time was to be able to send cyphered messages in the future, meaning that uncyphering the said message would take a predictable time. Rivest, Shamir. and Wagner [36] proposed the first real construction of such a function, introducing *time-lock puzzle*. Their construction was based on the RSA cryptosystem, the core of the puzzle was to compute

$$b = a^{2^t} \mod N \tag{1}$$

with N a product of two large prime numbers and a a member of the group $\frac{\mathbb{Z}}{N\mathbb{Z}}^{\times}$. The puzzle issuer can easily compute b by computing $e = 2^t \mod \phi(N)$ and then $b = a^e \mod N$ with knowledge of the factorization of N, with $\phi(N)$ being Euler's totient function. However, an agent who wants to solve the puzzle without knowledge of N factors will have compute iteratively a^{2^1}, a^{2^2}, ..., a^{2^t} using modular squaring's. The security is based on the fact that computing $\phi(N)$ from N is provably as hard as factoring N and the conjecture that there is no faster method of computing $a^{2^t} \mod N$ without knowledge of $\phi(N)$ than iteratively squaring [5], guarantees that solving the puzzle takes at least $T = t \cdot S$ seconds, where S is the number of squaring per second that an agent can process. The novelty of the scheme proposed by Rivest et al. compared to May's proposal is that it removes the need for a trusted third party.

One has to wait for more than twenty years to see substantial evolution in the field of provable time functions when in 2018 Boneh et al. [6] introduced the notion of *Verifiable Delay Functions* (VDFs), based on the seminal work of Rivest et al.

2.2 Definition of Verifiable Delay Functions

Boneh et al. present a class of functions $\mathcal{M} \to \mathcal{Y}$, for an input message space $\mathcal{M}$ and an output space $\mathcal{Y}$, that consists of a set of three algorithms [6]:

- **Setup:** $\mathsf{Setup}(\lambda, \tau) \to \mathbf{pp} = (ek, vk)$ that takes a security parameter λ and a challenge difficulty τ and outputs the public parameters **pp** which consist of the evaluation key ek and the verification key vk. The security parameter λ can be an RSA security (the modulus size), bit-level security, an elliptic curve security strength, etc. The evaluation key and the verification keys will vary greatly depending on the construction, or even be identical; in short, they provide an instance of the underlying cryptographic scheme considered, e.g., an RSA modulus.

- **Evaluation:** $\mathsf{Eval}(ek, m) \rightarrow (y, \pi)$ that takes an input message m from $\mathcal{M}$ and outputs a solution y from $\mathcal{Y}$; depending on the actual construction of the VDF, the output can admit a proof π to speed the verification up. Here again, the input and output spaces $\mathcal{M}$ and $\mathcal{Y}$ will depend on the construction. For an RSA-based VDF for example, given a modulus N, we will have $\mathcal{M} = \mathcal{Y} = [1, N-1]$.
- **Verification:** $\mathsf{Verif}(vk, m, y, \pi) \rightarrow \{\top, \bot\}$ that accepts as an input the verification key vk, the evaluation input message m, a candidate solution y and a potential auxiliary proof π, and deterministically returns $\top$ if $\mathsf{Eval}(ek, m) = (y, \pi)$, and $\bot$ otherwise. In order to be efficient, the Verif algorithm has to run in a time *polylog* of Eval.

The set of algorithms must satisfy three properties, *correctness*, *soundness* and *sequentiality*, to qualify a function as a VDF:

- **Correctness:** A VDF is said to be correct if $\mathsf{Verif}(vk, m, y, \pi)$ returns $\top$ when $(y, \pi) \leftarrow \mathsf{Eval}(ek, m)$ for any $(ek, vk) \leftarrow \mathsf{Setup}(\lambda, \tau)$, λ, τ, and m from $\mathcal{M}$.
- **Soundness:** A VDF is sound if for any algorithm $\mathcal{A}$ that runs in time $\mathcal{O}(\mathsf{poly}(\tau, \lambda))$,

$$\mathbb{P}[\mathrm{Verif}(vk, m, y, \pi) = \top \ \wedge \ y \neq \mathrm{Eval}(ek, m)] < \epsilon, \qquad (2)$$

  where $\mathbf{pp} = (ek, vk) \leftarrow \mathsf{Setup}(\lambda, \tau)$ and $(m, y, \pi) \leftarrow A(\lambda, \mathbf{pp}, \tau)$. In other words, given the public parameters $\mathbf{pp} = (ek, vk)$ and an output (m, y, π) generated by the algorithm A, if y is not the solution output by $\mathsf{Eval}(ek, m)$, then the probability that the verification accepts on y and π should be negligible in the size of the security level λ.
  This ensures that an attacker cannot forge a fake solution given an input m, in the context of the public parameters λ and τ.
- **Sequentiality:** A VDF consists of a sequence of τ steps that must be performed sequentially, i.e., one cannot compute steps in parallel. However, the steps themselves might be parallelizable; however, they should not give a substantial advantage to an agent with a high parallelization capacity.

One should notice that in this formal definition of a VDF, the public parameters are dependent on the challenge difficulty τ. Hence, in case the challenge difficulty needs to change, all the public parameters have to be recomputed again. This property has been abandoned by the most recent VDF constructions, allowing more flexible use in applications.

2.3 The Wesolowski construction

This paper focuses on a specific VDF construction, namely Wesolowski's VDF introduced [39]. This construction, which is based on sequential modular squaring, offers the best tradeoff between verification time and a lightness of the outputs [7, 19]. Moreover, such features allow the integration of Wesolowski's VDF in DLTs at little cost, for instance as a spam-prevention mechanism [2].

Setup. Wesolowski's VDF setup requires two security parameters: in the rest of the paper, we will overrule the λ parameter mentioned in the general framework with a new λ (typically between 1024 and 2048 bits) which is the RSA modulus size and k (typically between 128 and 256), which represents the bit-level security of the hashing functions used in the protocol. A committee generates an RSA public modulus[4] N of bit length λ and defines a cryptographic hashing function $H : \{0,1\}^* \mapsto \{0,1\}^{2k}$. We then define, for any $\alpha \in \{0,1\}^*$,

$$\begin{cases} H_{prime}(\alpha) = H(\alpha + j) \\ j = min\{i \mid H(\alpha + i) \text{ is prime}\} \end{cases} \tag{3}$$

returning the smallest prime number larger or equal to $H(\alpha)$.

Evaluation. The evaluation takes a challenge $\tau \in \mathbb{N}$ and an input message $m \in \{0,1\}^*$ as inputs, and then computes $x = H(m)$ and solves the challenge $y = x^{2^\tau} \bmod N$. It is important to reiterate that, if the evaluator knows $\phi(N)$, the computation time is drastically decreased as

$$x^{2^\tau} \bmod N = x^{2^\tau \bmod \phi(N)} \bmod N. \tag{4}$$

Proof. The proof begins by computing $l = H_{prime}(x + y)$ and then $\pi = x^{\lfloor 2^\tau / l \rfloor} \bmod N$. This algorithm can be parallelized, and it takes a $\frac{2\tau}{s \log(\tau)}$ time to run if s cores are used. At the end of this phase, the evaluator can publicly use the pair (l, π) as a proof of computation. In Algorithm 1 we present a pseudocode of evaluation and proof algorithms combined.

Algorithm 1: Evaluation and proof of the Wesolowski construction

Input: $m \in \{0,1\}^*$, $\tau \in \mathbb{N}$
Output: $\pi \in [0, N-1]$, l prime $\in [0, 2^{2k} - 1]$

```
x ← H(m)
y ← x
for k ← 1 to τ do
    y ← y^2 mod N
end for
l ← H_prime(x + y)
π = x^⌊2^τ / l⌋ mod N
return (π, l)
```

[4] Multiparty generation of RSA modulus is an actively researched topic [8, 9, 15]. We do not provide further details as this is out of the scope of the paper.

Verification. A verifier takes as an input the 4-tuple (m, τ, l, π). It first gets the hash of the message $x = H(m)$ as in the evaluation algorithm and checks whether

$$H_{prime}(x + y') = l, \tag{5}$$

where

$$\begin{cases} r = 2^\tau \bmod l, \\ y' = \pi^l \cdot x^r \bmod N. \end{cases} \tag{6}$$

The computations described by Eq.(6) are performed to recover the VDF solution y from (m, τ, l, π). Considering that $\pi = x^{\lfloor 2^\tau / l \rfloor} \bmod N$ and $y = x^\tau \bmod N$, then

$$\begin{aligned} \pi^l \cdot x^r \bmod N &= x^{\lfloor 2^\tau / l \rfloor} \cdot x^{2^\tau \bmod l} \\ &= x^{2^\tau} = y \end{aligned}$$

Given the soundness property, we are assured that if the y' that we find is equal to x^{2^τ} then the 4-tuple given in the input is a correct computation of the VDF. However, since we cannot require the verifier to compute again the $y' = x^{2^\tau} \bmod N$ given that verification should be exponentially faster than the evaluation; this is when Eq.(5) is helpful. It works as opening a commitment from the evaluator and ensures that the y' value found by the verifier is the same as the y value computed by the prover.

The verification phase for the Wesolowski's VDF takes a time $O(\lambda^4)$ and is thus independent of T. In Algorithm 2 we present the pseudocode for this phase.

Algorithm 2: Verification of the Wesolowski VDF

Input: x, τ, π, l
Output: $\top$ or $\bot$
1: $x \leftarrow H(m)$
2: $r \leftarrow 2^\tau \bmod l$
3: $y \leftarrow \pi^l \cdot x^r \bmod N$
4: **if** $l = H_{prime}(x + y)$ **then**
5: **return** $\top$
6: **else**
7: **return** $\bot$
8: **end if**

Overhead on the network The output size of a VDF can be of paramount importance, e.g., when the VDF is used as a spam-prevention mechanism. As bandwidth becomes a valuable resource in contested environments, the spam-prevention mechanism's footprint must be limited. A VDF solution in the order of magnitude of megabytes would not be suitable for such an application. For example, as of 2022, an IOTA message can be up to 32 KiB (32*1024 bytes) and

the dedicated space for the PoW proof is 8 bytes. Therefore, we can state that a footprint in the order of kilobytes would be acceptable. Fortunately, Wesolowski's VDF has such a tiny footprint. An evaluation output is composed of elements of the RSA group π which is at most λ bits long and a prime number of size at most $2 \cdot k$. As motivated later on, a conservative estimation can be $\lambda = 2048$ bits and $2 \cdot k = 512$ bits, which make 320 bytes.

2.4 Breakdown of the verification algorithm

In this paper, we will focus on the analysing the performance of the verification algorithm, which plays a critical role in many applications. In particular, we will analyze how one can minimize computation time.

Looking at Algorithm 2, we can isolate the following components:

1. Lines 1 and 2 operate some initialization of x and r. It is of interest to point out that Line 2 seems to have a computation time exponentially dependent on τ; however, knowing that l is a prime number, one can actually compute
$$2^\tau \bmod l = 2^{\tau \bmod \phi(l)} \bmod l, \tag{7}$$
since $\phi(l) = l - 1$. These two lines will take a time negligible compared to the other ones, so we will not consider them in our analysis.
2. Line 3 computes the modular multi-exponentiation (MME) operation $y \leftarrow \pi^l \cdot x^r \bmod N$. Modular multi-exponentiation is not a trivial operation to compute [2]. For example, one could naively compute $y_1 \leftarrow \pi^l \bmod N$, $y_2 \leftarrow x^r \bmod N$ and then $y \leftarrow y_1 \cdot y_2 \bmod N$ but it is proven to be suboptimal [26], especially considering the sizes of N, l and r.
3. Line 4 consists of the computation of the H_{prime} function, which deterministically returns a prime number that is the output of several iterations of a hash function. This function itself can be broken down into two components, *i)* the primality testing and *ii)* the hashing function. Both have massive literature attached to them, and numerous optimization exists for these computations. We will present in this paper how some of these optimizations fit particularly well for Wesolowski's verification computations.

In Table 1 we display the computation times of the MME, hashing, and prime testing parts of the verification algorithm, considering values for k in $\{256, 512\}$ and for λ in $\{1024, 2048, 4096\}$, these values being the most realistic ones to be used in a real-world setup. We have used the Apple M1 chip for testing, with custom implementation using the OpenSSL library for C++, the same setup that will be used for the experimental section of this article. In addition to the absolute values in milliseconds, we have provided the table with the percentage of each part of the verification, each line adding up to 100%.

This table shows us that the three steps of these calculations take a substantial amount of time, depending on the parameters k and λ. One can observe that the time dedicated to hashing is the most stable one, taking 14–18% for $k = 256$

down to 7–9% for $k = 512$. On the other hand, the multi-exponentiation computation share increases with λ and the prime testing shares increase with k. It is to be also noted that the computation time of multi-exponentiation depends on both k *and* λ, $2k$ being the size of the exponents and λ of the radices. Hashing also increases with k and λ, considering that λ is the size of the input values and $2k$ is the size of the output. However, the prime testing is independent of λ because it only computes prime testing on numbers of size $2k$.

We can see from this breakdown that each part can be optimized independently of the other ones, considering that they are executed in sequentially and the output of the two first is used as an input of the following one. Moreover, the values displayed in Table 1 motivate the need for this study, each part taking substantial time; hence an optimization on each has its own merits. Therefore, we will analyze each part of the verification procedure, namely MME, hashing and prime testing, separately in the next sections.

k	λ	**MME**	**Hashing**	**Prime testing**	**Total time**
256	1024	0.117 (14%)	0.143 (17%)	0.574 (69%)	0.834
	2048	0.407 (35%)	0.205 (18%)	0.551 (47%)	1.16
	4096	1.542 (65%)	0.328 (14%)	0.500 (21%)	2.36
512	1024	0.271 (7%)	0.286 (7%)	3.42 (86%)	3.98
	2048	0.771 (17%)	0.413 (9%)	3.42 (74%)	4.60
	4096	2.94 (42%)	0.66 (9%)	3.41 (49%)	7.01

Table 1: Breakdown of the verification algorithm, describing computation times for the modular multi-exponentiation (MME), hashing, and prime testing parts, for different values of k and λ. Times are given in milliseconds and the percentage of each row adds to 100%.

Algorithm 3: Pseudocode of the function H_{prime}

Input: $m \in \{0,1\}^*$, $k \in \mathbb{N}$
Output: l prime $\in \{0,1\}^{2k}$
1: $i \leftarrow 0$
2: **while** $H_k(x+i)$ is not prime **do**
3: $\quad i \leftarrow i+1$
4: **end while**
5: **return** $H_k(x+i)$

3 Use of double-exponentiation algorithms for verification optimization

In this section, we optimize the computation of the MME part of Wesolowski's VDF verification, corresponding to Line 3 of Algorithm 2. According to Table 1, MME can take up to 42% of the whole verification time, which motivates the need for optimization. In the verification algorithm, it is required to compute $y' \leftarrow \pi^l \cdot x^r \bmod N$ where π, x and N are of size λ bits and l and r are of size $2k$ bits. A naive way to perform such an operation is to compute $y_1 \leftarrow \pi^l \bmod N$ and $y_2 \leftarrow x^r \bmod N$ and then $y \leftarrow y_1 \cdot y_2 \bmod N$. However, this is proven to be suboptimal [26], and we will present in this section some algorithms to speed up the computation time. The above problem is of the following form:

Problem 1 (Double-exponentiation computation). Find the algorithm $A^\star$ that solves $x^a \cdot y^b \bmod N$ in the shortest average time, for x and y random elements of an RSA group of modulus N with size λ and a, b random integers of size K.

Problem 1 is referred to as the *double-exponentiation computation* and is part of a broader area of research named *multi-exponentiation algorithms* which consists in computing $\prod_{i=1}^{n} x_i^{e_i}$ with $e_{i\{i\in[1,n]\}}$ and $x_{i\{i\in[1,n]\}}$ being elements of a cyclic group and n a natural number. Using the same notation as used to define the Wesolowski's VDF, we can say that the radices x_i can be represented using λ_i bits and the exponents are of size K. As we will see in the rest of the section, these parameters largely affect the verification time.

The literature comprises various algorithms, mostly dedicated to solving the general multi-exponentiation problem [29,41], which can be easily reduced to the double-exponentiation problem. For our scenario, two main algorithms can be considered, the windowed 2^w-ary algorithm (we refer to it as 2^w-ARY) and the Simultaneous sliding window algorithm by Yen, Laih and Lenstra (YLL). The two algorithms are similar, the latter being optimization of the former. They are based on the idea of different precomputing combinations of products of small powers of x and y. The evaluation of the multi-exponentiation is reduced to a series of table lookup, modular product, and squaring, in a very similar fashion to the quick exponentiation. The two algorithms have a tuning parameter w that describes the size of the small powers of y and x computed in the precomputation phase. The precomputation computation time grows exponentially with w but the evaluation time is inversely proportional to w. The main practical difference between the two algorithms is that YLL introduces some computational overhead in the evaluation phase of the multi-exponentiation that gets smoothed out when increasing λ, hence increasing the relative weight of modular multiplication concerning to the computational overhead induced by YLL.

In a previous work [3], we performed an implementation study of double-exponentiation algorithms, providing computation time comparisons between the naive approach, 2^w-ARY and YLL. In particular, this paper [3] provides the heatmap referenced in Figure 1 highlighting the algorithm with the shortest computation time for double-exponentiation for different values of λ and k.

Please note that this figure has been experimentally generated using the same setup as described in Section 2.4. Hence, for the set of values we are interested in, i.e., K in $\{128, 256\}$ and λ in $\{1024, 2048, 4096\}$, the best algorithm to use is YLL with $w = 2$ for $K = 256$ or YLL with $w = 3$ for $K = 512$.

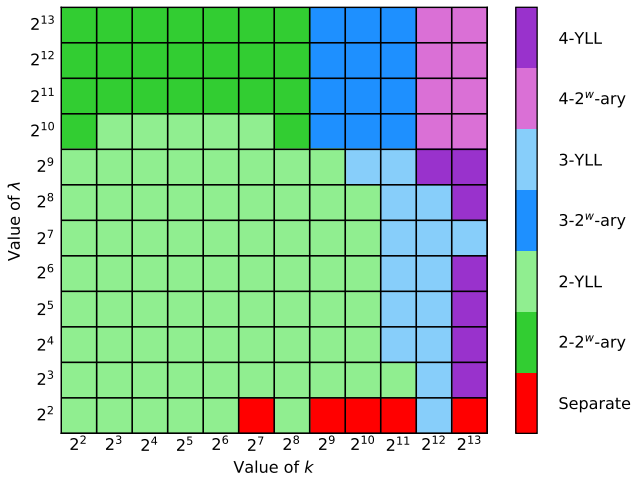


Fig. 1: Best MME algorithm as a function of λ and k on an Apple M1 chip.

3.1 Experimental results

Given the implementations provided in [3], we compare the computation times for the MME part of the verification using an Apple Silicon M1 chip and the OpenSSL library for C++. Table 2 presents the results for the values of K and λ aforementioned. The optimization is pretty stable, around 66% for $K = 256$ and 62–63% for $K = 512$, which represent a substantial improvement. Moreover, the computation time is linear concerning K and λ, which helps with the predictability of the performances. Additionally, the fact that the optimal algorithm for these values is always YLL [3] helps with the implementation.

4 Prime testing

The second important optimization is on the H_{prime} function, described in Section 2.4. It consists of two operations, a hash and the primality testing of its result, to be repeated a certain number of times. However, the number of repetitions is unpredictable as the algorithm stops as soon as it finds a prime output.

K	λ	Separate	Optimized	Factor
256	1024	0.119	0.078 (2-YLL)	x0.66
	2048	0.415	0.270 (2-YLL)	x0.65
	4096	1.546	1.027 (2-YLL)	x0.66
512	1024	0.222	0.143 (3-YLL)	x0.64
	2048	0.788	0.490 (3-YLL)	x0.62
	4096	3.001	1.891 (3-YLL)	x0.63

Table 2: Multi-exponentiation computation times and factor for different values of K and λ, comparing performing multi-exponentiations with Separate (naive approach) or Optimized algorithms.

In this section and in the following one, we will explain how the H_{prime} function can be optimized by using specific tricks for the primality testing of the hash output and then speeding up the hashing itself.

4.1 Number of candidates needed to find a prime in H_{prime}

In the Appendix, we provide a comprehensive list of primality testing algorithms, confirming that this is a very well-studied field and any implementation of industry-grade cryptography libraries is extremely optimized. Thus, in paper will present yet another prime testing algorithm, but rather show how we can optimize their use.

The OpenSSL library uses the Rabin-Miller primality test [32]. We recall that a primality test involving the Rabin-Miller algorithm of a number x works in the following way:

1. The test begins what is called the *trial division* phase, in which x will be tested for the division of a certain amount of the first prime numbers. This amount depends on the size of x. This is done to quickly eliminate numbers instead of immediately going into the Miller-Rabin test, which involves heavy computations.
2. After passing the trial division phase, x will be tested for primality as described in 6. Rabin-Miller's algorithm is a powerful generalization of the primality testing based on the use of Fermat's Little Theorem. Fermat's test simply computes a^{p-1} mod p, for some a larger than 1. If the outcome is different from 1 the number, p, is rejected as a composite one, otherwise the algorithm reports 'probably prime'. The RM test 'removes' the probabilistic nature of Fermat's primality testing by implementing the same computation in a more refined way. Firstly, $p-1$ is represented as $2^a b$ for some positive a and b, b is odd. Then, one computes a^b mod p first, followed by a squarings modulo p. If the final answer is different from one, the candidate - p - is rejected, but if it is equal to one, then we look at the previous a reductions. If any of them is different from 1 or $p-1$, then the number p is still being

rejected as a composite, even though it passes Fermat's primality test. This is the crucial difference between the two tests. According to the computational number theory, if the test is successfully passed for any a less than $2ln^2(p)$, then the number p can be certified as a prime number. The algorithm is indeed very powerful, but it still involves a large number of modular exponentiations.

We have estimated that for an Apple Silicon M1 chip, for a very large number of uniformly drawn integers of size 2048 bits, the time spent in the Rabin-Miller test represents 96% of the accumulated computation time. This shows that the trial division which prunes some candidates is a valuable part of prime testing. It prevents most of them from entering the Rabin-Miller test and then grieve the computation time.

In the case of Wesolowski's VDF verification, we are interested in finding a prime number during the H_{prime} function, in which we run the Rabin-Miller test for a certain number of times. But what number of candidates must be tested until finding a prime one? The more candidates required to find a prime, the more hashing will have to be performed and the more prime testing we will run. So, although not all candidates make it to the Robin-Miller test, some do and all consume some time to compute.

Considering that the H_{prime} function returns a uniform random number of N bits, the probability that it is prime is $\frac{1}{N \ln 2}$ [31], thus the average number of trials to find a prime is $N \ln 2$, so respectively 155, 178, 266 and 354 for N being 224, 256, 384 or 512 bits. The distribution of the number of candidates until finding a prime follows the geometric law. For $p = \frac{1}{N \ln 2}$, we have $P[x = k] = p \cdot (1-p)^k$. In Figure 2, we represent the distribution of the number of candidates required to find a prime for $N = 256$ and $N = 512$ with the dashed lines. While the number of candidates is theoretically unbounded, after one million runs of the H_{prime} function we observe that it takes less than 5000 trials at most to find a prime.

However, number theory teaches us that sieving candidates using small prime numbers effectively increases the probability of finding a prime. Indeed, half of the numbers are divisible by 2; a third is divisible by 3; a fifth is divisible by 5 etc. Thus, if we can guarantee that a number has no small prime divisors, up to a certain threshold, then the probability that it is a prime rises significantly.

De Bruijn [12] gives an estimated formula of the probability that a sieved number is a prime. For a number of size N guaranteed to have no prime numbers up to B, the probability that it is a prime is $e^{\gamma} \frac{\ln B}{N} \left(1 + o\left(\frac{1}{N}\right)\right)$ with γ being the Euler-Mascheroni's constant, approximately 0.57721. For example for $B = 47$, i.e., the 15-th prime number and $N = 512$ bits, the probability of finding a prime is $\frac{1}{51}$. Compared to the probability of $\frac{1}{354}$ for a uniform random number, this is a huge gap.

4.2 Primality testing without trial division part

Fortunately, in our case, we have a way to manipulate the output of the hash function to obtain a sieved number without having to go through trial divisions.

For example, for a random number x and a given number B, it is guaranteed that $\overline{x} = \lfloor \frac{x}{B} \rfloor \cdot B + 1$ is co-prime with B; in other words, it does not share any prime divisor with B. Then, considering y the output of our hash function, if we take B being the product of a certain amount of the first prime numbers, we can build $\overline{y}$ that does not have any of these first prime numbers. Then, we can considerably limit the number of candidates the H_{prime} function has to go test before finding a prime number.

The question now is to understand which number B to consider, i.e., how many small primes are necessary to yield a satisfying reduction of the candidates. In Table 3, we display the following informations:

- We provide a theoretical estimation of the average number of candidates required to find a prime number, for *i)* uniformly random candidates and *ii)* for candidates sieved up to the 15 first small primes, for candidates of size 224, 256, 384 and 512 bits. For uniformly drawn candidates we use the formula $N \ln 2$ and for sieved numbers, we use the De Bruijn estimation.
- We provide an experimental estimation of the average number of candidates required to find a prime for comparison alongside the theoretical estimations.

We have limited our experimentations to the fifteen first small primes because it is the maximum number of small primes whose product fits into a word of 64 bits, which allows us to use the OpenSSL word division function instead of dividing by a `bignum`.

From this table, we can make several observations:

- As predicted by the formula, the average number of candidates to find a prime is linear with the size of the number to test N.
- The average number of candidates to test decreases dramatically by a factor of 7 (approximation from $e^{\gamma}\dot{\ln}43 \approx 6.699$) when the first fifteen primes are considered. This means that we can perform seven times less hash and prime testing for a single word division and a multiplication per candidate.
- De Bruijn's formula precision is pretty weak for a low amount of small primes sieved. For example, when only the prime factor 2 is removed, the difference between the real experimental values and the De Bruijn formula is 40%. However, such a difference decreases below 10% after only eight primes.
- De Bruijn approximation constantly overestimates the average number of candidates before finding a prime, even though it becomes very close for fifteen primes. This is important to observe because it proves that the formula is only an approximation and should be considered carefully.

Table 3 shows that we do not need an extremely high number of sieved num- bers to reduce the number of candidates significantly. We argue that a cutoff of the first fifteen primes is sufficient as it offers a good tradeoff between division costs and effictively reduces the number of candidates. For example, the OpenSSL library runs the trial division phase for the 64 first primes for numbers up to 512 bits long. The 64-th prime is 311, then according to the De Bruijn formula, the expected number of candidates should be respectively 25 and 50

Primes	Product of primes	N=224		N=256		N=384		N=512	
		DB	Exp	DB	Exp	DB	Exp	DB	Exp
0	1	155	153	177	178	266	265	354	354
1	2	125	78	143	87	215	132	287	176
2	6	79	52	90	57	136	88	181	117
3	30	54	41	61	47	92	71	123	93
4	210	44	36	51	40	76	60	102	81
5	2310	36	52	41	36	62	55	83	73
6	510510	33	29	38	34	58	50	77	68
7	510510	30	28	35	31	52	47	70	63
8	9699690	29	26	33	29	50	45	67	62
9	223092870	27	25	31	29	47	42	63	57
10	6469693230	25	24	29	27	44	41	59	56
11	200560490130	25	24	29	27	43	40	58	54
12	7420738134810	24	22	27	26	41	39	55	52
13	304250263527210	23	22	26	25	40	38	53	51
14	13082761331670030	23	22	26	25	39	37	52	50
15	614889782588491410	22	21	25	24	38	36	51	49

Table 3: Comparison of the average number of candidates required to find a prime number between the De Bruijn formula (DB) and experimental results (exp), with tested numbers of size 224, 256, 384, and 512 bits.

for numbers of size N of 256 and 512 bits. When compared with Table 3, it is absolutely not a significant improvement. However, the product of the 64 first primes is a 417 bits long number which fits in 7 words of 64 bits each, which is the reason why we argue that sieving only up to the 15 first primes is sufficient.

In Figure 2, we show the probability distribution of the number of candidates to be tried before finding a prime number and the values of what happens when we sift through the numbers with the presented technique and have adjusted the x-axis to a logarithmic scale for better visibility. Considering that the distribution follows a geometric law, we observe that it is markedly more likely to find a prime with a low number of candidates than without sieving. The probability of performing more than 100 trials is almost nonexistent. In addition to reducing the average computation time, it also improves the predictability of the computation time by reducing the standard deviation.

4.3 Security analysis

It is fundamental to verify whether the filtering technique described in the previous section can be exploited to affect the security of the VDF concerning the

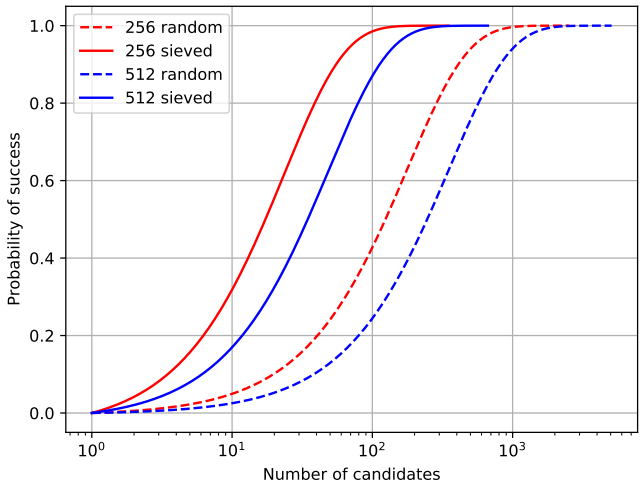


Fig. 2: Probability of finding a prime in function of the number of trials for candidates of size 256 and 512 bits and with comparison with sieved numbers

properties described in Section 2.2. More precisely, we shall investigate the set of primes that can be generated according to our sieving method.

The method that we have presented earlier produces numbers that can be identified to an arithmetic progression of the form $\{a + i \cdot d \mid i \in \mathbb{N}\}$, with $a = 1$ and $d = B$. We denote the prime-counting function in this set with $\pi_{a,d}(x)$, which counts the number of primes smaller than x. This function can be efficiently approximated by $\pi(x) \approx \frac{x}{\ln x}$. For numbers of size respectively 256 and 512 bits, there are approximately 10^{74} and 10^{151} primes respectively. Dirichlet and Legendre conjectured, then proved by de la Vallée Poussin, that

$$\pi_{a,d}(x) \sim \frac{\mathrm{Li}(x)}{\varphi(d)}, \tag{8}$$

with φ being the Euler's totient function, and Li being the *offset logarithmic integral*, that is $\mathbf{Li}(x) = \int_2^x \frac{dt}{\ln t}$. We can make two observations: *i)* the number of primes does *not* depend on the offset a; *ii)* the factor of primes "lost" in the sieving operation only depends on the size of B, not on the size of the numbers sieved.

In Figure 3 we display the number of primes that can be generated after the sieving operation (in blue) as a function of the number of small primes that are used in the sieving, against the total number of primes that can be generated randomly (in red). We show the results for numbers of size 256 and 512 bits. This figure shows that when sieving the fifteen first prime numbers, the number of accessible prime numbers is reduced from 2^{247} to 2^{207} (for the 256 dynamic range). But even this issue can be circumvented with the use of randomization.

We propose the following algorithm. For a given output x of the hash function, we compute

$$\overline{x} = \lfloor \frac{x}{B} \rfloor \cdot B + r, \tag{9}$$

with r being a random number smaller than B and co-prime with $\lfloor \frac{x}{B} \rfloor \cdot B$. In this way, we can generate all the co-prime numbers with B. The main issue is the way to draw this number r in practice. This number r has to be deterministically generated to ensure the correctness of the non-interactive H_{prime} function, the verifier needs to be able to generate the same ones while generating a number with enough entropy to cover all the prime numbers, which can naturally be achieved with a hash function. The coprimality test is easily done by computing the greatest common divisor of $\lfloor \frac{x}{B} \rfloor \cdot B$ and r and ensuring it is equal to 1.

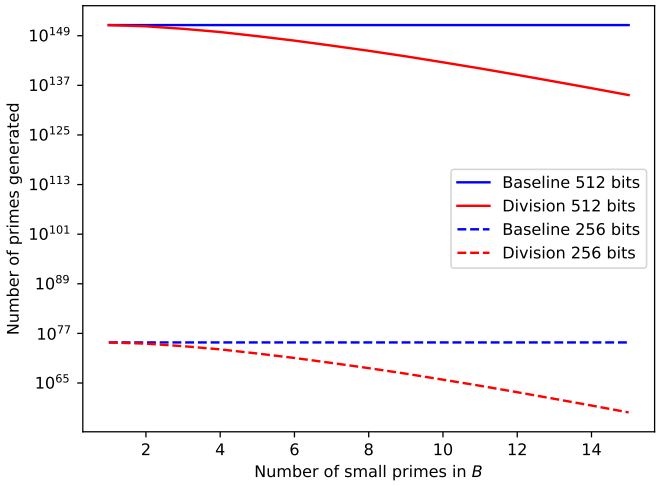


Fig. 3: Comparison of the number of primes reachable after sieving concerning random numbers in the function of the number of small primes sieved and with values for 256- and 512-bits numbers.

4.4 Experimental results

Finally, in Table 4 we show the average time to compute a single primality test, for a randomly chosen number and a sieved number (up to fifteen small primes). Along with these values, we estimate the computation time spent on prime testing during the execution of the H_{prime} function by multiplicating the previous value by the average number of candidates required to find a prime. Finally, in the last column, we display the improvement gained on the H_{prime}

function when using the sieving technique in the last column. We display these values for numbers of size 224, 256, 384 and 512 bits, and the unit of measurement here is in microseconds.

One might be surprised to see that a single primality test is, on average, more computationally expensive when sieved numbers are considered. However, this can be clearly explained because a sieved number is 7 times more likely to pass the trial division and make it to the actual Rabin-Miller test, which is more expensive than the trial division by a factor of (almost) 2. One can observe that the ratio of 7 is also observed here. With the sieved approach, we actually save the trial division for the 85% of candidates eliminated (that would have failed the Rabin-Miller test anyway). The improvement observed on the H_{prime} function itself is limited, only up to around 4%. However, we invite the reader to note that the primality test is intertwined with executing a hash function. Hence, the speedup here and the optimization that we will show in the next section contribute to a non-negligible optimization of the Wesolowski's VDF verification.

Size	**Random**		**Sieved**		**Speedup**
224	554	(3.57)	532	(25.3)	3.97%
256	635	(3.57)	609	(25.4)	4.09%
384	2430	(9.17)	2360	(65.5)	2.88%
512	3510	(9.92)	3390	(69.2)	3.42%

Table 4: Total and per-trial (in parenthesis) average computation time in microseconds of primality testing between a uniformly random number and a sieved number up to fifteen small primes, for numbers of size 224, 256, 384, and 512 bits.

5 Optimized hashing

In this section, we discuss about the hash function used in the Wesolowski's VDF. The requirement is that the chosen hash function H must be *cryptographically secure*, that is H satisfies the following properties:

- **Pre-image resistance (PR):** given an output y, it is not feasible to find an input x such that $H(x) = y$.
- **Second pre-image resistance (SPR):** given an input x_1, it is hard to find a second input x_2 such that $H(x_1) = H(x_2)$.
- **Collision resistance (CR):** it is hard to find a distinct pair (x_1, x_2) of inputs such that $H(x_1) = H(x_2)$.

It is important to note that the *CR* property is similar to the *SPR* one, *CR* implying *SPR*. For this reason, if a hash function satisfying *SPR* has an output

size k, then a hash function satisfying CR must have an output size $2k$ because of the existence of birthday attacks [21].

Practically, acceptable bit-level security is 128 bits, whereas a considered very strong bit-level security is 256 bits. Then a hash function would need an output of respectively 256 or 512 bits, hence the values aforementioned. This section, will present a practical point of view on hash functions, using the C++ library of OpenSSL, showcasing different hash function candidates and then an optimization that can be applied for our specific use case of VDF verification.

5.1 Overview of hash functions

First, we provide an overview of the candidate hash function to consider in our study. In theory, any hash function that satisfiessatisfies the properties mentioned above is suitable for the H_{prime} function. However, differences in performance among various hash functions may be huge. Therefore, considering that we are using the OpenSSL library for conducting our experiments, we have restricted the choice of the hash function to the following available hash families:

- **SHA-2:** SHA-2 is the previous generation of NIST-selected hash functions designed in the early 2000s. It encompasses six hash functions, `SHA-224`, `SHA-256`, `SHA-384`, `SHA-512`, `SHA-512/224` and `SHA-512/256`, having respectively an output size of 224, 256, 384, 512, 22 and 256 bits. We will restrict the study to only the `SHA-256` and `SHA-512` functions for the sake of simplicity. It is important to note that for this family, OpenSSL provides two programming interfaces, one is based on a `C`-style implementation with types and functions and the other one is based on a modern hashing API, called `EVP` and is aimed at increasing the modularity of hash, changing a hash function being done by only changing a parameter instead of rewriting the whole code section. We denote the `C`-style API by `SHA-2 (C)` whereas the `EVP` interface will just be denoted as `SHA-2` for consistency with the other families as they are only available through the `EVP` API.
- **SHA-3:** this is the last generation of the NIST-elected hash functions, sometimes referred to as `KECCAK`, its former name before standardization by the NIST destined to replace the `SHA-2` family, although the latter being still deemed secure. `SHA-3` is generally considered to be less performant than its predecessor but it features two important characteristics: *i)* unlike `SHA-2`, it is resistant to *length-extension attacks*; *ii)* it is built on a complete different design, hence industry can safely switch to an alternative, if `SHA-2` design appears to be broken in the future. It encompasses four hash functions, `SHA3-224`, `SHA3-256`, `SHA3-384` and `SHA3-512`, with respective output sizes 224, 256, 384 and 512 bits. We will restrict our study to the `SHA3-256` and `SHA3-512` for the sake of simplicity.
- **SHAKE:** the `SHAKE` family is a hybridisation between `SHA-2` and `SHA-3`. Its main feature is to allow for *sponge hashing*, meaning that one can set arbitrary input and output sizes. We excluded this family from our study because we do not consider this feature to be of interest for the sake of the H_{prime} function.

– **BLAKE2:** the `BLAKE` hash function is an unfortunate finalist of the NIST competition that elected `KECCAK` as the new standard for the `SHA-3` generation, but it has made it to most libraries due to its excellent performances. It comes with two implementations, `blake2s256` and `blake2b512`, with an output size of respectively 256 and 512 bits. The main difference between the two functions is that the former is designed for 32-bits architecture while the latter is designed for 64-bits ones.

We have compared the computation time of `SHA-2`, `SHA-3` and `BLAKE2` families on an Apple Silicon M1 chip using the OpenSSL library for C++ for the sets of parameters we are interested in our VDF study, being the input size λ in $\{1024, 2048, 4096\}$ and with a security level of 128 or 256 bits (i.e., an output size of 256 or 512 bits). Table 5 presents the results. A first observation is that the deprecated `SHA-2 (C)` implementation leads to significantly better performances, from 15% when $k = 512$ and $\lambda = 4096$ to over 50% in computation time reduction when $k = 256$ and $\lambda = 1024$. A second observation is that the `SHA-3` family is substantially slower than the `SHA-2` family, especially for $k = 256$ and the gap increases with λ. Finally, the case of `BLAKE2` is more complex. It is consistently outperformed by `SHA-2` for $k = 256$, even using the EVP interface. For $k = 512$, it is outperformed in the case of $\lambda = 1024$ (outperforms `SHA-2 (C)` with a very slight margin (3%) when $\lambda = 2048$ and definitely takes over for larger values of λ.

Security level	λ	**SHA-2**	**SHA-2(C)**	**SHA-3**	**BLAKE2**
128 bits level security (256 bits output)	1024	566	255	686	702
	2048	740	460	1052	1074
	4096	1197	853	1745	1825
256 bits level security (512 bits output)	1024	875	601	899	635
	2048	1222	956	1464	928
	4096	1960	1689	2633	1486

Table 5: Comparison between `SHA-2`, `SHA-2-C`, `SHA-3` and `BLAKE2` implementations on an Apple Silicon M1 chip using OpenSSL library for C++, for different security level k and input size λ. For `BLAKE2`, we used the `blake2s256` function for the 128 bits level security and `blake2b512` for the 256 bits level security parameter. All values are given in nanoseconds.

The sheer variety of hash functions makes it hard to select one for a given project. We have presented a set of hash functions family that were suitable for VDF verification and their performance. Still, one should keep in mind that these results only indicate that other factors, such as security, have to be considered. For our purpose, we chose to pick the `SHA-2` family for the computation of the H_{prime} function, as the primary goal of the work is about performance. It is

important also to note that the `BLAKE2` family is of great interest too. On 64-bits systems, the `blake2b512` performs well for large values and the `blake2s256` should be promising for 32-bits architectures.

5.2 Context copying optimization

In this section, we will present the optimizations for the H_{prime} function. In Table 1, we showed that the H_{prime} function makes up to 18% of the total computation time of the VDF verification. Looking back to Algorithm 3, we can observe that the input fed into the hash function is $x + i$ with i typically being a value below 5,000, as described in Section 4.1. So i can be described by using only two bytes. On the other hand, x typically has a size of 1024, 2048, or 4096 bits, respectively 128, 256, or 512 bytes, which means that the input $x + i$ of the hash function is mostly the same when the H_{prime} function is called, except for the two last bits being updated at each call.

Delving into the inner workings of hashing functions will help us understand how this particular phenomenon can be leveraged to optimize the performances of the H_{prime} function. From a high-level perspective, a hash function is called in OpenSSL as follows:

- **Creation.** A *hashing context* holding the interna hashing state gets created.
- **Update.** The state of the hashing context is updated with a memory area and a length in bytes. This will prepare the internal hashing states with the input memory.
- **Finalization.** The hash output is written into a memory area, ready to be used.

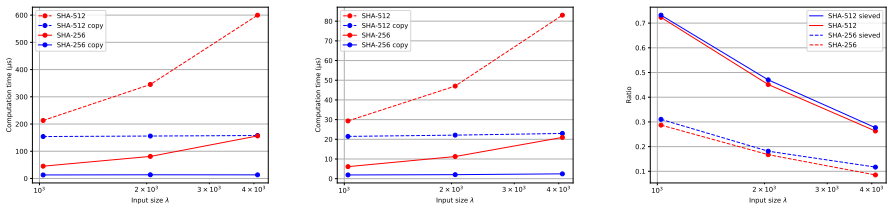
It is important to note that the updating part of the hash primitive can take arbitrary long memory size. It is possible to deterministically update multiple times a *hashing context*, with different memory areas and memory lengths, the smallest unit being one byte. Our idea is to use a *hashing context* copying the mechanism provided by OpenSSL allows saving the context's internal state to be independently updated multiple times. Considering that our number x is of size n bytes, then we start by initializing a hash context c that we will update with the $n - 2$ left-most bytes of x. Then for each iteration performed, we will create a new hash context c' that is a deep copy of c, and then we will update c' with the 2 right-most bytes of x, finalize the hash of the context c' and write the result in a number r that we will test for primality. If not a prime, we set x to be $x + 1$ to get the next candidate deterministically. The key idea is that the primary hash context c is never changed after the first update of $n - 2$ bytes, so each iteration involves an update of 2 bytes instead of n bytes.

5.3 Experimental results

In this section, we will only consider the C-style interface of `SHA-2` as motivated earlier. We are interested in determining whether and how the context copying method can reduce the hashing computation time with our experimentation.

In the presented optimization, the total hashing time is *practically* independent of the input size λ. We say *practically* because we are limited to 2 bytes of increment, i.e., 65536 trials, which are enough for our purposes. Bur, more generally, if m (instead of 2) indicates the number of bytes left for the increment and $k \ll m$ is the number of bytes of the input, then one has to hash $k - m$ bytes in the first phase and then has 256^m trials.

Figure 4a depicts the computation time spent hashing in H_{prime} with and without the copy context technique (Figure 4b) for `SHA-256` and `SHA-512`, as a function of the input size λ. Figure 4b shows the same computation time (please note the different scale of the y-axis) when the input is sieved using the technique described in Section 4 Finally, Figure 4c displays the ratio between the original and the copy-context techniques for the `SHA-256` and `SHA-512` hash functions, and with and without the sieving technique applied. First, we observe that, even for a small number of trials, the computation time is (almost) independent of the input size in the case of the copy-context technique while the original method has a super-linear behavior. We can also see that there is a ratio of 7 between the case when it is sieved and when it is not, which is consistent with the results found in Section 4. Finally, Figure 4c indicates that the ratio between the original and the copy-context techniques does not depend much on whether the input is sieved or not, i.e., whether there are many trials.



(a) Non-sieved input. (b) Sieved input. (c) Ratio of computation time of copy-context over original method.

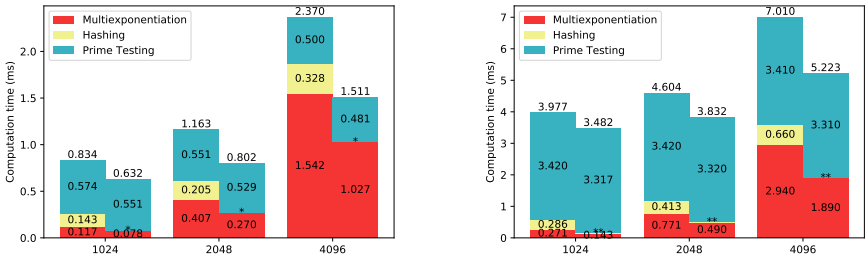
Fig. 4: Comparison of computation time dedicated to hash in the verification algorithm in function of the input size λ.

6 Conclusion

In this paper, we have conducted a thorough analysis of the verification algorithm of Wesolowski's VDF construction. We have broken down the algorithm into three main bricks, *i)* modular multi-exponentiation computation, *ii)* prime testing and *iii)* hashing. We have conducted a theoretical analysis of the underlying problem for each of these components, isolated some optimizations that

work particularly well for our specific situation, and supported theoretical findings with experimental results.

For the modular multi-exponentiation part, we have shown that using dedicated algorithms such as the *Simultaneous sliding window* algorithm can reduce by 33% the computation time of a modular multi-exponentiation. For the prime-testing algorithm, we have provided an analysis of how to generate numbers that are already sieved out of the H_{prime} function with minimal computational overhead and how it reduces the number of trials significantly to find a prime in the H_{prime} function by a factor of 7 with very little overhead which finally yields about a 5% speedup. Finally, we have demonstrated how to leverage the structure of the hashing function inputs to dramatically decrease the computation time, which becomes negligible. To sum up the results of this work, we display Figure 5 which shows the *total* computation time when $k = 128$ (Figure 5a) and $k = 256$ (Figure 5b): our optimizations reduce the computation time of the verification of the Wesolowski's VDF between 12% and 35% for the range of parameters considered in this work.



(a) $k = 128$
∗ indicates 0.003ms for optimized hashing

(b) $k = 256$
∗∗ indicates 0.023ms for optimized hashing

Fig. 5: Comparison of computation time dedicated to each component between non-optimized and optimized implementations for bit-level security k equal to 128 bits in (a) and 256 bits in (b)

References

1. W.R. Alford et al. There are Infinitely Many Carmichael Numbers. *The Annals of Math.*, 1994.
2. V. Attias et al. Preventing Denial of Service Attacks in IoT Networks through Verifiable Delay Functions. In *GLOBECOM 2020*, 2020.
3. Vidal Attias, Luigi Vigneri, and Vassil Dimitrov. Rethinking modular multi-exponentiation in real-world applications. *Journal of Crypto. Eng.*, 2022.
4. Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

5. N. Bitansky et al. Time-lock puzzles from randomized encodings. In *ITCS 2016*, pages 345–356, 2016.
6. D. Boneh et al. Verifiable delay functions. In *CRYPTO 2018*.
7. D. Boneh et al. A survey of two verifiable delay functions. *IACR Cryptol. ePrint Arch.*, page 712, 2018.
8. M. Chen et al. Diogenes: Lightweight scalable rsa modulus generation with a dishonest majority. In *IEEE SP21*.
9. M. Chen et al. Multiparty generation of an rsa modulus. *Journal of Cryptology*, 2022.
10. A. Chepurnoy et al. A Systematic Approach to Cryptocurrency Fees. In *Financial Cryptography and Data Security*. 2019.
11. B. Cohen and K. Pietrzak. The chia network blockchain, 2019.
12. N. G. de Bruijn. On the Number of Uncancelled Elements in the Sieve of Eratosthenes. In *Reviews in Number Theory*. 1974.
13. L. De Feo et al. Verifiable delay functions from supersingular isogenies and pairings. In *ASIACRYPT 2019*, 2019.
14. César A. Del Río. Use of distributed ledger technology by central banks: A review. *Enfoque UTE*, 8(5):1–13, 2017.
15. Cyprien Delpech de Saint Guilhem, Eleftheria Makri, Dragos Rotaru, and Titouan Tanguy. The return of eratosthenes: Secure generation of rsa moduli using distributed sieving. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 594–609, New York, NY, USA, 2021. Association for Computing Machinery.
16. Adina Di Porto and Piero Filipponi. A probabilistic primality test based on the properties of certain generalized Lucas numbers. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 211–223. Springer, 1988.
17. John R. Douceur. The sybil attack. In Druschel, Peter, Kaashoek, Frans, Rowstron, and Antony, editors, *Peer-to-Peer Systems*, volume 2429, pages 251–260. Springer Berlin Heidelberg, 2002.
18. Paul Erdős and Carl Pomerance. On the number of false witnesses for a composite number. *Mathematics of Computation*, 46(173):259–279, 1986.
19. Attias et al. Implementation Study of Two Verifiable Delay Functions. In *Tokenomics*, pages 1–6, 2020.
20. Pietro Ferraro, C. King, and Robert Shorten. Distributed ledger technology for smart cities, the sharing economy, and social compliance. *IEEE Access*, 6:62728–62746, 2018.
21. Philippe Flajolet and Andrew M. Odlyzko. Random Mapping Statistics. *Advances in Cryptology — EUROCRYPT '89*, pages 329–354, 1990.
22. Paul Garrett and Daniel Lieman. Public-key cryptography: Baltimore (proceedings of symposia in applied mathematics). *American Mathematical Society, Boston*, 2005.
23. Fred Huibers. Distributed Ledger Technology and the Future of Money and Banking: Banking is Necessary, Banks Are Not. Bill Gates 1994. *Accounting, Economics and Law: A Convivium*, pages 1–37, 2021.
24. Saurabh Jain, Neelu Jyothi Ahuja, P. Srikanth, Kishor Vinayak Bhadane, Bharathram Nagaiah, Adarsh Kumar, and Charalambos Konstantinou. Blockchain and Autonomous Vehicles: Recent Advances and Future Directions. *IEEE Access*, 9:130264–130328, 2021.
25. Christine Kim. Ethereum 2.0: how it works and why it matters, 2020.
26. V. V. Kochergin. On Bellman's and Knuth's Problems and their Generalizations. *Journal of Mathematical Sciences (United States)*, 233(1), 2018.

27. Rudolf Lidl, Winfried B. Müller, and Alan Oswald. Some Remarks on Strong Fibonacci Pseudoprimes. *Appl. Algebra Eng., Commun. Comput.*, 1(1):59–65, March 1990.
28. Timothy C May. Timed-Release Crypto. `http://cypherpunks.venona.com/date/1993/02/msg00129.html`, 1993.
29. Bodo Möller. Algorithms for Multi-exponentiation. In Serge Vaudenay and Amr M Youssef, editors, *Selected Areas in Cryptography*, pages 165–180, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
30. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.
31. Władysław Narkiewicz. *The Development of Prime Number Theory.* Springer, Berlin, Heidelberg, 2000.
32. OpenSSL. Openssl primality checking documentation, 2022. `https://www.openssl.org/docs/man3.0/man3/BN_check_prime.html`, Last accessed on 2022-04-24.
33. Krzysztof Pietrzak. Simple verifiable delay functions. In *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*, volume 124, pages 60:1—-60:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.
34. Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.
35. Serguei Popov, Hans Moog, Darcy Camargo, Angelo Capossele, Vassil Dimitrov, Alon Gal, Andrew Greve, Bartosz Kusmierz, Sebastian Mueller, Andreas Penzkofer, Olivia Saa, William Sanders, Luigi Vigneri, Wolfgang Welz, and Vidal Attias. The Coordicide. *IOTA Foundation*, 2020.
36. R. L. Rivest et al. Time-lock puzzles and timed-release Crypto 1 Introduction. *Cryptologia*, 1996.
37. D. Roeck et al. Distributed ledger technology in supply chains: a transaction cost perspective. *International Journal of Production Research*, 2020.
38. V. Tabora. The Evolution of the Internet, From Decentralized to Centralized, 2018.
39. B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019*, 2019.
40. G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 21:2327–4662, 2016.
41. S. M. Yen et al. Multi-exponentiation. *IEE Proceedings: Computers and Digital Techniques*, 1994.

Appendix

Overview of primality testing algorithms

Primality testing is one of the most important problems in computational number theory:

Problem 2 (Primality testing). Given a large integer p, determine whether p is a prime or a composite number.

For large prime numbers, it is clear that the exhaustive search algorithm that tests all the potential prime divisors of p is computationally infeasible. Indeed, correct algorithms allow to determine if a number is a prime with an absolute certainty but are not practical for large numbers that are commonly used; conversely, probabilistic algorithms determine whether a number is prime with bounds on the probability of giving a wrong answer. In this subsection we review the most relevant probabilistic algorithms used to test primality.

Fermat's Little Theorem. One can test if

$$2^{p-1} \equiv 1 (\text{mod } p)$$

and, if so, then either p is a prime or p is a 2-pseudoprime according to the Fermat's Little Theorem (FLT). The smallest composite number, for which this test fails is 341. One can substitute 2 with larger values, but still there is a set of composite numbers, called Carmichael numbers, for which the test produces an incorrect answer. The fact that the set of Carmichael numbers is infinite has been established in 1994 [1].

Rabin-Miller primality test. So, instead of using FLT-based tests, we can use more precise Rabin-Miller primality test. If in computing $a^{p-1} (\text{mod } p)$ one gets "1" as an answer, the algorithm performs a "forensic" investigation on how exactly this outcome 1 has been obtained. In this case, the one can use only a very small number of witnesses in order to test the primality of p, but the proof that only small number of witnesses is sufficient depends on the correctness of the extended Riemann hypothesis.

Solovay-Strassen primality test. The Solovay-Strassen primality testing algorithm is based on a very simple idea: to test if p is a prime number, one computes $a^{\frac{p-1}{2}}$ and compares this to the value of the Jacobi symbol $\left(\frac{a}{p}\right)$. If p is a prime number, the value of the Jacobi symbol is the same as the value of the Legendre symbol $\left(\frac{a}{p}\right)$. If p is not a prime, then these two values are the same with at most 50% probability. The entire point of the algorithm is that there is no need to factorize p in order to evaluate the Jacobi symbol. So, if the algorithm is executed for, say, 100 values of a and in all the cases

$$a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) (\text{mod } p), \tag{10}$$

then we can claim that p is a prime with probability at least $1 - 2^{-100}$ [18]. The biggest drawback of this algorithm is the necessity to compute the Jacobi

symbol, which involves a large number of GCD computations, and is the chief reason why it is rarely used in practice.

Generalized Fibonacci-based primality test. A similar algorithm is based on the following interesting property of Fibonacci numbers: *For every prime number, except 5, the following congruence holds:*

$$F_{p^2-1} \equiv 0 (\text{mod } p).$$

Since the value of $F_{p^2-1} (\text{mod } p)$ can be obtained in $\mathcal{O}(\log p)$ operations [16, 27], the algorithm is attractive. Again, it fails for very few, specific composite numbers, called Fibonacci pseudo-primes – the smallest one being 161.

In Table 6, we evaluate the computational complexities to test the primality of p for the methods described above. According to our analysis, it is clear that Rabin-Miller's approach is superior:

- When comparing Rabin-Miller and Solovay-Strasses tests, we notice that the latter technique requires the *same* number of modular multiplications plus $\ln p$ evaluations of the Jacobi symbols, which requires approximately the same computational time.
- Fibonacci-based primality testing is implemented by exponentiating the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ to the power of p. The constant in our estimation, 10.5, is based on the assumption that one uses Strassen's matrix multiplications algorithm[5].

Therefore, Rabin-Miller is about twice faster than Solovay-Strasses test and about seven times faster than generalized Fibonacci-based primality test. This basically makes Rabin-Miller's test as the de-facto standard in the primality testing field. A similar analysis can be found in the article [22].

Table 6: Complexity to test the primality of p (MM stands for modular multiplications).

Primality test	**Complexity**
Rabin-Miller	$1.5 \cdot \ln p \cdot \log_2 p$ (MM)
Solovay-Strasses	$1.5 \cdot \ln p \cdot \log_2 p$ (MM) + $\ln p$ (Jacobi symbols estimation)
Fibonacci-based	$10.5 \cdot \ln p \cdot \log_2 p$ (MM)

[5] The use of standard matrix multiplications algorithm will increase this constant to 12.