Fine-Tuned Verification for Verifiable Delay Functions in Blockchains

Vidal Attias[1], Luigi Vigneri[1], and Vassil Dimitrov[1,2]

[1] IOTA Foundation
Berlin, Germany
[2] University of Calgary
Calgary, AB, Canada

Abstract. Verifiable Delay Functions (VDFs) are a set of new cryptographic schemes ensuring that an agent has spent some time (evaluation phase) in an 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-fold 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.

Keywords: Verifiable Delay Function, anti-spam mechanism, optimization, prime testing

1 Introduction

After a score of years since the emergence of commercial services on the Internet [30] and following an increasing centralization of data by governments and large private companies, decentralized projects promise more control over privacy and personal data. 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 [7,12]. Using time as a spam-prevention mechanism dates back to the late 90s with the Hashcash system [3] that prevented e-mail spamming by requiring senders to solve a small cryptographic puzzle, called *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* [22] 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[1]. 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* [14, 17] posting updates on their state or paying a toll or a parking, *supply chains* [29] involving multiple partners that need a source of trust in their relations, *digital monetary systems* [11, 16] and so on. Fortunately, some DLTs are moving forward aiming to be the backbone of these future high-performance, decentralized networks: among them, IOTA [27], Ethereum 2.0 [18], Bitcoin's Lightning off-chain protocol [26], Polkadot [32]. These projects aim to eliminate the need for expensive Proof of Work, sometimes replacing it with a *Verifiable Delay Function* (VDF) [5, 10, 25, 31] as a spam-prevention mechanism [1] or as a core component of the consensus protocol [8]. VDFs were introduced in 2018 by Boneh et al. [5] 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 $y = x^{2^\tau} \bmod N$, which can only be done efficiently by performing τ squarings and no parallel algorithm is known to solve this problem. In this paper, we will look closely at the *Wesolowski's efficient VDF* [31] which ensures small verification times and low communication overhead [13]; these features make this VDF one of the best candidates for spam prevention in DLTs.

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, in 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.

[1] 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, May mentioned ideas of how to use *timed-release cryptographic protocols* [20]. His idea was to send cyphered messages in the future, meaning that uncyphering the said message would take a predictable time. Rivest, Shamir. and Wagner [28] proposed the first real construction of such a function, introducing *time-lock puzzle*. Their construction was based on RSA, the core of the puzzle was to compute

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

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 $e = 2^t \bmod \phi(N)$ and then $b = a^e \bmod N$ with knowledge of the factorization of N, $\phi(N)$ being Euler's totient function. However, an agent wanting to solve the puzzle without knowledge of N factors will have compute iteratively a^{2^1}, a^{2^2}, $\ldots$, a^{2^t} using modular squaring's. The security is based on the equivalent hardness of computing $\phi(N)$ from N and factoring N, and the conjecture that the faster way of computing $a^{2^t} \bmod N$ without knowledge of $\phi(N)$ is via iterative squarings [4], which 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 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 $a^{2^t} \bmod N$ without knowledge of $\phi(N)$ than iteratively squaring [4], 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 this new scheme is that it removes the need for a trusted third party.

The next substantial improvement on provable time functions happened in 2018 when Boneh et al. [5] 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 [5]:

- **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 the verification returns $\top$ for any legitimate evaluation output.
- **Soundness:** the probability that the verification returns $\top$ on inputs that are *not* the output of the evaluation algorithm is negligible in λ.
- **Sequentiatlity:** The evaluation of a VDF consists of a sequence of τ steps that are performed sequentially.

In this definition of VDFs, the public parameters depend on the challenge τ. Hence, if τ needs to change, the public parameters must be computed again. This property has been abandoned by the most recent VDF constructions.

2.3 The Wesolowski construction

This paper focuses on Wesolowski's VDF construction [31] which is based on sequential modular squarings. It offers the best tradeoff between verification time and output lightness [6, 13] which allows use for DLT applications such as a spam-preventoin mechanism.

Setup The setup requires two security parameters: λ (typically between 1024 and 2048 bits), an RSA modulus size and k (typically between 128 and 256), the bit-level security of the hashing functions used in the protocol. A committee generates an RSA public modulus N of bit length λ and defines a cryptographic hashing function $H : \{0,1\}^* \mapsto \{0,1\}^{2k}$. We then define, for any $\alpha \in \{0,1\}^*$,

$$\begin{cases} H_{prime}(\alpha) = H(\alpha + j) \\ j = min\{i \mid H(\alpha + i) \text{ is prime}\} \end{cases} \tag{2}$$

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

$$x^{2^\tau} \mod N = x^{2^\tau \mod \phi(N)} \mod N. \tag{3}$$

Proof The proof first computes $l = H_{prime}(x+y)$ and then $\pi = x^{\lfloor 2^\tau / l \rfloor} \mod N$. 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. Algorithm 1 presents a pseudocode of evaluation and proof algorithms combined.

Algorithm 1: Wesolowski's valuation and proof

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)
```

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

$$H_{prime}(x + y') = l, \tag{4}$$

where

$$\begin{cases} r = 2^\tau \mod l, \\ y' = \pi^l \cdot x^r \mod N. \end{cases} \tag{5}$$

The computations described by Eq.(5) are performed to recover the VDF solution y from (m, τ, l, π). Considering that $\pi = x^{\lfloor 2^\tau / l \rfloor} \mod N$ and $y = x^{2^\tau} \mod N$, then

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

The verification phase for the Wesolowski's VDF takes a time $O(\lambda^4)$ and is independent of τ. Algorithm 2 presents a pseudocode.

Algorithm 2: Wesolowski's verification

Input: m, τ, π, l
Output: $\top$ or $\bot$
1: $x \leftarrow H(m)$
2: $r \leftarrow 2^{\tau} \mod l$
3: $y \leftarrow \pi^{l} \cdot x^{r} \mod N$
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 is of paramount importance: 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 analyzing the performance of the verification algorithm, which plays a critical role in many applications. In particular, we will study how to minimize computation time.

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

1. Lines 1 and 2 initializes of x and r. Line 2 has a computation time exponential iwth τ; however, since l is prime, we have
$$2^{\tau} \mod l = 2^{\tau \mod \phi(l)} \mod l, \tag{6}$$
since $\phi(l) = l - 1$. These lines 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 $y \leftarrow \pi^{l} \cdot x^{r} \mod N$. Modular multi-exponentiation is not a trivial operation to compute [1]. For example, one could try $y_1 \leftarrow \pi^{l} \mod N$, $y_2 \leftarrow x^{r} \mod N$ and then $y \leftarrow y_1 \cdot y_2 \mod N$ but it is suboptimal [19].
3. Line 4 is the H_{prime} function, which returns a prime number that is the output of several iterations of a hash function. This function can be broken down into, *i)* the primality testing and *ii)* the hashing function. Both have been extensively studied, and numerous optimization exits. We will present how some of these fit well for Wesolowski's verification.

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. The hardware is the Apple M1 chip, using OpenSSL, and will also be used in the rest of this paper. In addition to computation time, we have provided the percentages of each part of the verification, each line adding up to 100%.

Each step takes a substantial amount of time, depending on k and λ. Hashing is quite stable, taking 14–18% for $k = 256$ down to 7–9% for $k = 512$. Inversely, the multi-exponentiation computation share increases with λ and the prime testing shares increase with k. The computation time of multi-exponentiation depends on k *and* λ, $2k$ being the exponent size and λ the radix. Hashing increases with k and λ, λ being input values' size and $2k$ the output's. However, prime testing is independent of λ because it only tests numbers of size $2k$.

Each part can be optimized independently since they are executed sequentially and the output of the two first is the last one's input. The values displayed in Table 1 motivate 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, namely MME, hashing and prime testing, in separate sections.

k	λ	**MME**	**Hashing**	**Prime testing**	**Total**
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, for the modular multi-exponentiation (MME), hashing, and prime testing parts, for different values of k and λ. Values are in milliseconds.

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, which can take up to 42% of the verification time according to Table 1. 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 this 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 suboptimal [19], and we present in this section some algorithms to speed it up. 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. The radices x_i have a λ_i-bits representation while the exponents have a K-bits one. 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 [21, 33], which can be easily reduced to the double-exponentiation problem. For our scenario, two relevant algorithms are the *Windowed 2^w-ary* algorithm (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 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, as in 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 time grows exponentially with w but the evaluation time is inversely proportional to w. Pratically, 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.

Attias et al. [2] performed an implementation study of double-exponentiation algorithms, providing computation time comparisons between the naive approach, 2^w-ARY and YLL, using a very similar hardware as ours. In particular, the authors provide a heatmap, referenced in Figure 1, highlighting the algorithm with the shortest computation time for double-exponentiation for different values of λ and k. 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 $K = 512$.

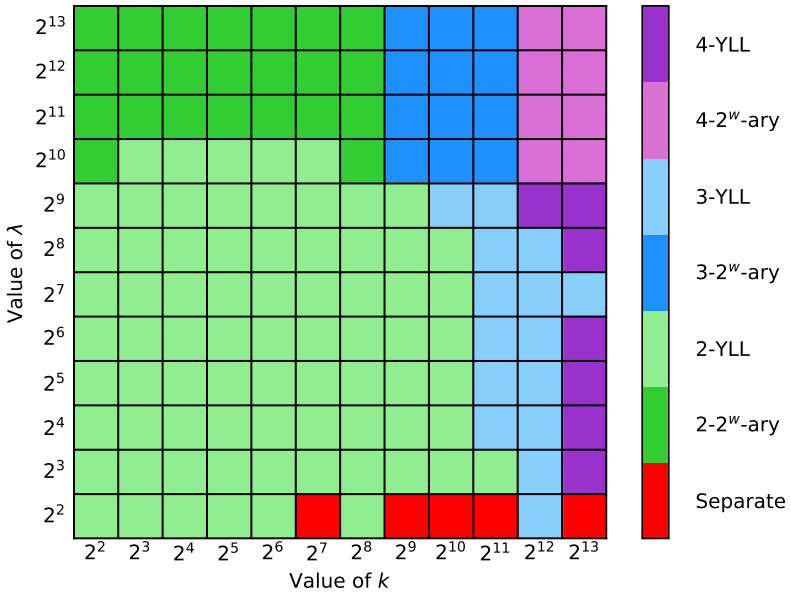


Fig. 1: Best MME algorithm as a function of λ and k on an Apple M1 chip.

3.1 Experimental results

Given the implementations provided in [2], we compare the computation times for the MME part of the verification. Table 2 presents the results for the values of K and λ aforementioned. The optimization is stable, around 66% for $K = 256$ and 62–63% for $K = 512$, which represent a substantial improvement. Moreover, the computation time is linear with K and λ, which helps with predicting performances. Additionally, YLL being constantly the optimal algorithms alleviates implementing.

4 Prime testing

The second important optimization concerns the H_{prime} function, described in Section 2.4. It consists of two operations, a hash and a primality test, repeated a certain number of times. However, the number of repetitions is unpredictable as the algorithm stops as soon when a prime is found. In the two following sections, we explain how H_{prime} can be optimized using specific tricks for primality testing of the hash output and then speeding up hashing.

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 Separate and Optimized algorithms.

4.1 Number of candidates required to find a prime

This paper will not present yet another prime testing algorithm, but rather show how we can optimize their use.

The OpenSSL library uses the Rabin-Miller primality test [24]. We recall that a primality test involving the Rabin-Miller algorithm of a number x works in the following way:

1. First, a *trial division* is performed on x, testing division by a certain amount of small primes. This amount depends on x's size. This quickly eliminates candidates instead of going into the Miller-Rabin test, which involves heavy computations
2. Then, x will be tested for primality. Rabin-Miller's algorithm is a generalization of the primality testing based on the use of Fermat's Little Theorem. Fermat's test simply computes $a^{p-1} \bmod p$, for some a larger than 1. If the outcome is not 1, p is rejected as a composite, 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 $a^b \bmod p$ first, followed by a squarings modulo p. If the final answer is not 1, p is rejected, otherwise, we look at the previous a reductions. If any of them is not 1 or $p-1$, then the number p is rejected as a composite, even though it passes Fermat's primality test. This is the crucial difference between the two tests. 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 very powerful, but it still involves a large number of modular exponentiations.

We have estimated that for our hardware, for a very large number of 2048 bits integers, 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 H_{prime} we run the Rabin-Miller test a certain number of times to find a prime. But how many candidates should we test to find a prime? The more candidates, the more hashing will have to be performed and the more prime we test. Although not all candidates make it to the Robin-Miller test, some do and all increase computation time.

Considering that H_{prime} returns a uniform random number of N bits, the probability that it is prime is $\frac{1}{N \ln 2}$ [23], 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 number of candidates until finding a prime distribution follows a geometric law. For $p = \frac{1}{N \ln 2}$, we have $P[x = k] = p \cdot (1-p)^k$. Figure 2 represents this distribution 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 it takes less than 5000 trials to find a prime.

However, number theory teaches us that sieving candidates using small prime numbers 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 a number has no small prime divisors, up to a certain threshold, then the probability that it is a prime rises significantly.

De Bruijn [9] gives an estimated formula of the probability for a sieved number to be prime. For a number of size N with no prime divisors up to B, the prime probability is $e^{\gamma} \frac{\ln B}{N} \left(1 + o\left(\frac{1}{N}\right)\right)$ with γ 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 to find 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 hash function's output to obtain a sieved number without trial divisions. For a random x and a given B, $\overline{x} = \lfloor \frac{x}{B} \rfloor \cdot B + 1$ is co-prime with B. Then, considering y the output of our hash function, if B is 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 test to find a prime.

We need to understand what B to consider, i.e., how many small primes are necessary to yield a satisfying reduction of the candidates. Table 3 displays the following information:

- A theoretical estimation of the average number of candidates required to find a prime, for *i)* uniformly random candidates and *ii)* 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.
- An experimental estimation of the average number of candidates required to find a prime evaluate 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 64 bits word, allowing us to use the OpenSSL word division function instead of dividing by a `bignum`.

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 numbers of size 224, 256, 384, and 512 bits.

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 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. It proves that the formula is only an approximation and should be considered carefully.

Table 3 shows that we do not need a high number of sieved numbers 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 for numbers of size N of 256 and 512 bits. When compared with Table 3, it is 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, supporting that sieving only up to the 15 first primes is sufficient.

Figure 2 shows the number of candidates to a prime probability distribution and the effect of sifting through the numbers with the presented technique with a logarithmic x-axis scale. Since the distribution follows a geometric law, it is markedly more likely to find a prime with a low number of candidates than without sieving. The possibility 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.

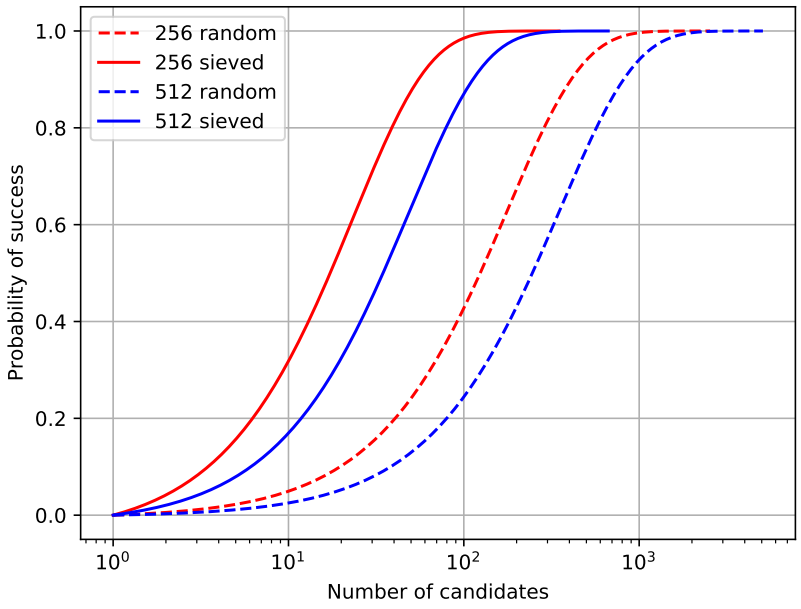


Fig. 2: Probability of finding a prime in function of the number of trials for candidates of size 256 and 512 bits and with comparison with sieved numbers

4.3 Security analysis

It is fundamental to check that the filtering technique described in the previous section cannot be exploited to affect the security of the VDF. Here we investigate the set of primes that can be generated using the sieving method.

This method 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

$$\pi_{a,d}(x) \sim \frac{\mathrm{Li}(x)}{\varphi(d)}, \tag{7}$$

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 see that *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.

Figure 3 displays the number of primes that can be generated after sieving (in blue) as a function of the number of small primes used to sieve, against the total number of primes that can be generated (in red) for numbers of size 256 and 512 bits. It 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 can be circumvented with the use of randomization.

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

$$\overline{x} = \lfloor \frac{x}{B} \rfloor \cdot B + r, \tag{8}$$

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 how to draw this number r in practice. r has to be deterministically generated to ensure H_{prime} correctness.

4.4 Experimental results

Finally, Table 4 shows the average time to compute a single primality test, for randomly chosen numbers and sieved numbers (up to fifteen small primes). We also estimate the computation time spent on prime testing in H_{prime} by multiplying the previous value by the average number of candidates required to find a prime. Finally, in the last column we display the H_{prime} speedup when using the sieving technique. We display these values for numbers of size 224, 256, 384 and 512 bits.

A single primality test is on average more computationally expensive when sieved numbers are considered because a sieved number is 7 times more likely to

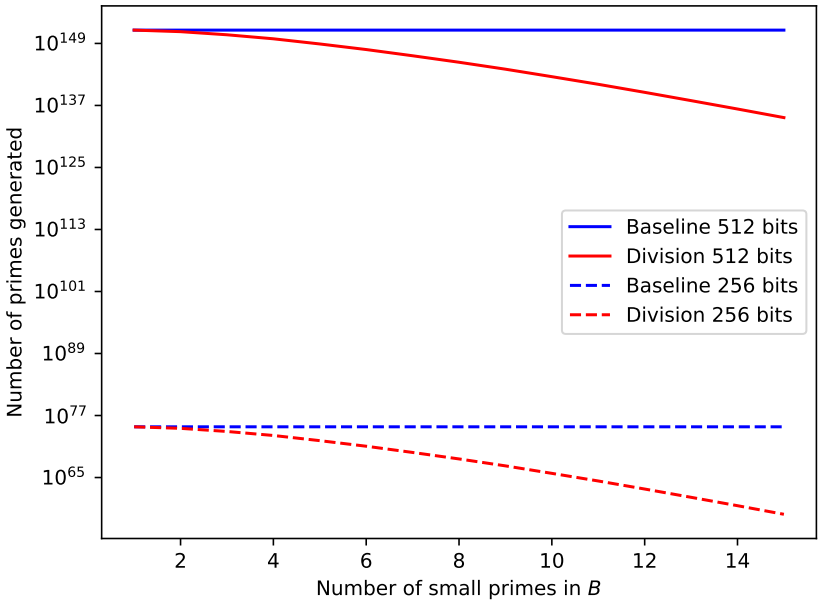


Fig. 3: Comparison of the number of primes reachable after sieving concerning random numbers in the function of the number of small primes sieved and with values for 256- and 512-bits numbers.

pass the trial division and make it to the Rabin-Miller test, which is more expensive than trial division by a factor of (almost) 2. The ratio of 7 is also observed here. When sieving, we save the trial division for the 85% of candidates eliminated (that would have failed the Rabin-Miller test anyway). The improvement observed on H_{prime} is limited, only up to around 4%. However, 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 verification optimization.

5 Optimized hashing

In this section, we discuss about the hash function H used in H_{prime}. We require H to be *cryptographically secure*, i.e, to satisfy:

- **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 $H(x_1) = H(x_2)$.

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.

– **Collision resistance (CR):** it is hard to find a distinct pair (x_1, x_2) of inputs such that $H(x_1) = H(x_2)$.

CR implying *SPR*. Thus, if a hash function satisfying *SPR* has an output size k, then it must have an output size $2k$ to satisfy *CR* because of the existence of birthday attacks [15].

Practically, an acceptable bit-level security is 128 bits, whereas a considered strong bit-level security is 256 bits. We present in this section an optimization that can be applied to VDF verification.

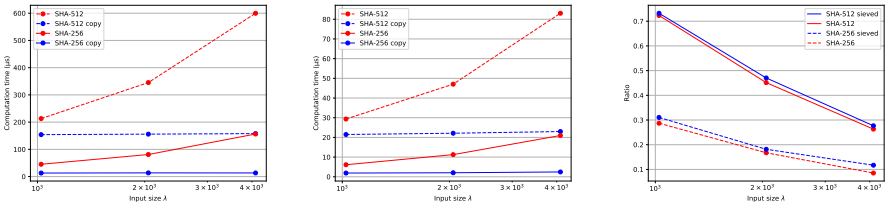
5.1 Context copying optimization

This section presents the optimizations for H_{prime}. Table 1 shows that H_{prime} makes up to 18% of the total verification time. The input fed into the hash function is $x + i$ with i typically being a value below 5,000, as in Section 4.1. So i can be described with only two bytes. On the other hand, x typically has a size of 128, 256, or 512 bytes, meaning that the hash function's input $x + i$ is mostly the same when the H_{prime} function is called, except for the last byte updated at each call.

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

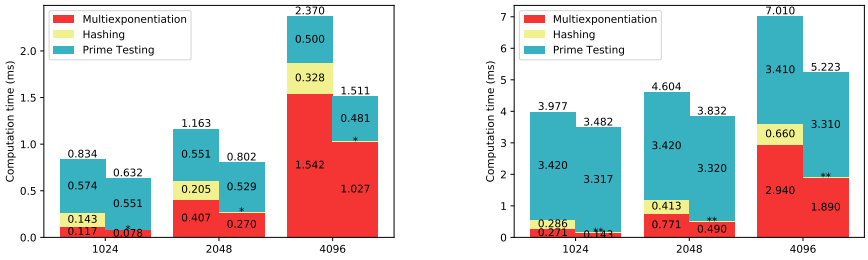
– **Creation.** A *hashing context* holding the internal hashing state gets created.
– **Update.** The state of the hashing context is updated given a memory area and a length in bytes. This will prepare the internal hashing states with the input memory.
– **Finalization.** The hash output is written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

The updating step of the hash primitive accepts arbitrary long memory size. It is possible to deterministically update multiple times a *hashing context*, with



(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.



(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)

different memory areas and memory lengths, the smallest unit being one byte. Our idea is to copy the *hashing context* at each loop in order to independently update it each time. Since x is of size n bytes, we start by initializing a hash context c that we will update with the $n-2$ left-most bytes of x. Then, at each iteration, we create a new hash context c' that is a deep copy of c, and we update c' with the 2 right-most bytes of x, finalize the hash of the context c' and write the result in r that we will be prime tested. If not a prime, we set $x \leftarrow x + 1$ to get the next candidate. The idea is that the original hash context c is never changed, so each iteration involves updating 2 bytes instead of n.

5.2 Experimental results

In this section, we are interested in determining whether and how the context copying method can reduce the hashing computation time with our experimen-

tation. 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. But, more generally, if m (instead of 2) is the number of bytes left for the increment and $k \ll m$ is the input's bytes size, then one has to hash $k-m$ bytes in the first phase and then can try 256^m times.

Figure 4a depicts the time spent hashing in H_{prime} with and without the copy context technique for `SHA-256` and `SHA-512`, as a function of λ. Figure 4b shows the same time when the input is sieved using the technique 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, 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. There is a ratio of 7 between when we sieve or not, which is consistent with the results in Section 4. Finally, Figure 4c indicates that the ratio between the original and the copy-context techniques does not depend on whether the input is sieved or not, i.e., the amount of trials.

6 Conclusion

In this paper, we have conducted an analysis of the verification of Wesolowski's VDF construction. We have divided the algorithm into three parts, *i)* modular multi-exponentiation, *ii)* prime testing and *iii)* hashing. We conducted a theoretical analysis of the underlying problem for each of these parts, fine-tuned some optimizations, and supported theoretical findings with experimental results.

For the modular multi-exponentiation, 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 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.

References

1. V. Attias et al. Preventing Denial of Service Attacks in IoT Networks through Verifiable Delay Functions. In *GLOBECOM 2020*, 2020.

2. Vidal Attias, Luigi Vigneri, and Vassil Dimitrov. Rethinking modular multi-exponentiation in real-world applications. *Journal of Crypto. Eng.*, 2022.
3. Adam Back et al. Hashcash-a denial of service counter-measure, 2002.
4. N. Bitansky et al. Time-lock puzzles from randomized encodings. In *ITCS 2016*, pages 345–356, 2016.
5. D. Boneh et al. Verifiable delay functions. In *CRYPTO 2018*.
6. D. Boneh et al. A survey of two verifiable delay functions. *IACR Cryptol. ePrint Arch.*, page 712, 2018.
7. A. Chepurnoy et al. A Systematic Approach to Cryptocurrency Fees. In *Financial Cryptography and Data Security*. 2019.
8. B. Cohen and K. Pietrzak. The chia network blockchain, 2019.
9. N. G. de Bruijn. On the Number of Uncancelled Elements in the Sieve of Eratosthenes. In *Reviews in Number Theory*. 1974.
10. L. De Feo et al. Verifiable delay functions from supersingular isogenies and pairings. In *ASIACRYPT 2019*, 2019.
11. César A. Del Río. Use of distributed ledger technology by central banks: A review. *Enfoque UTE*, 8(5):1–13, 2017.
12. 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.
13. Attias et al. Implementation Study of Two Verifiable Delay Functions. In *Tokenomics*, pages 1–6, 2020.
14. 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.
15. Philippe Flajolet and Andrew M. Odlyzko. Random Mapping Statistics. *Advances in Cryptology — EUROCRYPT '89*, pages 329–354, 1990.
16. 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.
17. 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.
18. Christine Kim. Ethereum 2.0: how it works and why it matters, 2020.
19. V. V. Kochergin. On Bellman's and Knuth's Problems and their Generalizations. *Journal of Mathematical Sciences (United States)*, 233(1), 2018.
20. Timothy C May. Timed-Release Crypto. `http://cypherpunks.venona.com/date/1993/02/msg00129.html`, 1993.
21. 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.
22. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.
23. Władysław Narkiewicz. *The Development of Prime Number Theory*. Springer, Berlin, Heidelberg, 2000.
24. OpenSSL. Openssl primality checking documentation, 2022. `https://www.openssl.org/docs/man3.0/man3/BN_check_prime.html`, Last accessed on 2022-04-24.

25. 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.
26. Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.
27. 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.
28. R. L. Rivest et al. Time-lock puzzles and timed-release Crypto 1 Introduction. *Cryptologia*, 1996.
29. D. Roeck et al. Distributed ledger technology in supply chains: a transaction cost perspective. *International Journal of Production Research*, 2020.
30. V. Tabora. The Evolution of the Internet, From Decentralized to Centralized, 2018.
31. B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019*, 2019.
32. G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 21:2327–4662, 2016.
33. S. M. Yen et al. Multi-exponentiation. *IEE Proceedings: Computers and Digital Techniques*, 1994.