An Efficient Hash Function for Imaginary Class Groups

Kostas Kryptos Chalkias, Jonas Lindstrøm, and Arnab Roy

Mysten Labs
{kostas,jonas,arnab}@mystenlabs.com

Abstract. This paper presents a new efficient hash function for imaginary class groups. Many class group based protocols, such as verifiable delay functions, timed commitments and accumulators, rely on the existence of an efficient and secure hash function, but there are not many concrete constructions available in the literature, and existing constructions are too inefficient for practical use cases.
Our novel approach, building on Wesolowski's initial scheme, achieves a staggering 500-fold increase in computation speed, making it exceptionally practical for real-world applications. This optimisation is achieved at the cost of a smaller image of the hash function, but we show that the image is still sufficiently large for the hash function to be secure. Additionally, our construction is almost linear in its ability to be parallelized, which significantly enhances its computational efficiency on multi-processor systems, making it highly suitable for modern computing environments.

Keywords: Imaginary class groups · Class group cryptography · Hash functions · Verifiable delay functions · Accumulators · Timed commitments

1 Introduction

Imaginary class groups have recently become a focal point in cryptographic research due to a unique property: their order remains elusive. Recall that the order of an imaginary class group with a given discriminant is known as the *class number* and it is believed to be difficult to compute for large discriminants, which is why we may assume that the order of a class group, even if we know the discriminant, is unknown.

Assuming the factorization is not known, the order of an RSA group is also unknown, but the benefit of class groups over RSA groups is that sampling a new group with unknown order is easier for class groups, because one can simply sample a sufficiently large negative, prime discriminant Δ and publish it. For RSA groups, sampling a new group is much more difficult to do because it requires sampling a modulus $N = pq$ where p and q are two prime factors, and if you know these you also know the group order. So in RSA groups, for a trusted party to sample a group with unknown order *without*, more sophisticated and computationally expensive protocols are required, such as secure multi-party computation (MPC), making class groups a more practical choice for certain cryptographic applications. See [8] for a recent example of this.

Groups of unknown order are used in many applications, most famously in the RSA signature scheme [1], but also for some implementations of accumulators [6], verifiable delay functions ([22,18]), and polynomial commitments [7]. We also note that specific groups of unknown order, including class groups, cannot replace RSA in applications which require the group order as a trapdoor - some examples are RSA-based time-lock puzzles [19], and random beacons [5].

When class groups are used in cryptography, sampling or hashing to a random element is an important primitive. This is for instance the case for some digital signature schemeas, where the plaintext has to be mapped to a group element, or for verifiable delay functions where a random input has to be sampled for each VDF instance.

Until recently, there have not been many concrete examples of hash functions for class groups. Wesolowski [22] presented a construction where the first coefficient a is restricted to be a prime. Recently, Seres, Burcsi and Kutas [20] have presented a set of new hash function schemes and have also shown that some previous constructions are insecure. Their work focuses in particular on how to construct a hash function with a uniform output in the class group and uses Bach's algorithm [3] to sample quadratic forms with composite a coefficients.

In [22], Wesoloski calls for optimizations to his hash construction, and this paper is a response to this. Our proposed scheme is significantly more efficient than both Wesolowski's construction and the constructions presented by Seres et al.

After a brief introduction to imaginary class groups in Section 2, we present the algorithm in Section 3 and analyze its time complexity. In Section 4 we prove the security of the hash function

by proving that the output is uniformly distributed in a large subset of the class group, and we provide estimates on the size of the subset. The algorithm is implemented as part of the open-source *fastcrypto* Rust library and is used in the implementation of verifiable delay functions on the Sui blockchain. Finally, Section 5 discusses implementation remarks and benchmarks, before presenting future work developments in Section 7.

2 Background

We will only provide a few facts and definitions around imaginary class groups here and refer to Chapter 5 in [10] for a thorough introduction. The imaginary class group $Cl(\Delta)$ with discriminant $\Delta < 0$ where $\Delta \equiv 1 \bmod 4$ and $|\Delta|$ is prime may be represented by all quadratic forms $ax^2 + bxy + cy^2$, which we will write (a, b, c), with $\Delta = b^2 - 4ac$ where the group operation is known as *composition* of forms which was discovered by Gauss [13]. In this representation, two quadratic forms f and g represent the same group element if they are equivalent, e.g. if there is a matrix $U \in SL_2(\mathbb{Z})$ such that $f = g \circ U$. Each equivalence class contains exactly one *reduced* form so we use reduced quadratic forms to represent class group elements:

Definition 1 (Reduced quadratic form). *A quadratic form (a, b, c) with $a > 0$ and discriminant $\Delta = b^2 - 4ac$ is said to be* reduced *if $|b| \leq a \leq c$ and if $a \in \{|b|, c\}$ implies $b \geq 0$.*

As discussed in the introduction, the order of a class group is hard to compute, but we get as a consequence of Dirichlet's class group formula an approximation, namely that $\#Cl(\Delta) \approx \sqrt{|\Delta|}$. The size of the discriminant for cryptographic protocols will typically be at least 1024 bits, but a recent paper by Dobson, Galbraith and Smith [12] suggests that discriminants should be thousands of bits to ensure that it is sufficiently hard to find the order of a class group for a random discriminant of a given size.

The only result we need about quadratic forms is the following lemma which ensures that the output of our algorithm is reduced.

Lemma 1. *If $a < \frac{\sqrt{|\Delta|}}{2}$ and $-a < b \leq a$ then (a, b, c) is reduced.*

Proof. By assumption, we have $|b| \leq a$, so we just need to consider the bound on c, but this is true because

$$c = \frac{b^2 + |\Delta|}{4a} \geq \frac{|\Delta|}{4a} > \frac{a^2}{a} = a.$$

This also proves that $a \neq c$. We may still have $a = b$, but by assumption this only happens when $b \geq 0$ which proves that (a, b, c) is reduced.

3 The Algorithm

3.1 Description

Our algorithm is an extension of Wesolowski's construction from [22]. The original construction samples a uniformly random prime a such that $a < \sqrt{|\Delta|}/2$ and $\left(\frac{\Delta}{a}\right) = 1$. This ensures that we can find a square root b such that $b^2 \equiv \Delta \bmod a$. If b is odd, $b^2 \equiv 1 \bmod 4$, so $4a$ divides $b^2 - \Delta$. If not, we can use the other square root, $a - b$, in place of b and compute the exact quotient $c = (b^2 - \Delta)/4a$ and return a quadratic form (a, b, c) with discriminant Δ. Lemma 1 ensures that the result is reduced.

Our algorithm is a natural extension to this construction where we pick an integer $k \geq 1$ and sample k primes smaller than $(\sqrt{|\Delta|}/2)^{1/k}$ and use their product as a. This method outputs the prime factorization of a and hence an efficient method to compute the square root of a.

The bottleneck for Wesolowski's construction is sampling random primes, and using multiple smaller primes instead allows for a significant speed-up and efficient parallelization at the cost of slightly reducing the size of the image, which we will analyse in depth in Section 4.2.

In the algorithm, we use the following function which samples a uniformly random integer in a range, modeled as a random oracle:

$$\textsc{RandomNumber}_B : \{0,1\}^* \times \mathbb{N} \to \{0, \ldots, \lfloor B \rfloor - 1\}$$

for $B \geq 2$. Realizing this in practice requires a method to sample a random number efficiently in an arbitrary range which is out of scope for this paper but we refer to Chapter 3 in [14] for some methods.

Algorithm 1 SampleModulus$(C, N, x, \mathsf{counter})$

Require: $C > 3$, $N \in \mathbb{Z} \setminus \{0\}$, $x \in \{0,1\}^*$ and $\mathsf{counter} \in \mathbb{N}$.
Ensure: a is prime, $3 \leq a < C$ and $b^2 \equiv N \bmod a$.

```
j ← 0,                                   ▷ Number of times we have incremented the counter
repeat
    a ← RandomNumber_C(x, counter + j)
    j = j + 1
until a is an odd prime and (N/a) = 1.
Let b ← ModSqrt(N, a) mod a              ▷ Efficiently computable because a is prime.
s ← RandomNumber_2(x, counter + j)
j ← j + 1
if s = 1 then
    b ← a − b
end if
return ((a, b), j).
```

Algorithm 2 SampleModuli(k, B, N, x)

Require: $k \geq 1, B > 3$, $N \in \mathbb{Z} \setminus \{0\}$ and $x \in \{0,1\}^*$.
Ensure: a_i is prime, $3 \leq a_i < B^{1/k}$, $b_i^2 \equiv N \bmod a$ for $i = 1, \ldots, k$ and $i \neq j \implies a_i \neq a_j$.

```
counter ← 0
for i = 1, ..., k do
    repeat
        ((a_i, b_i), j) ← SampleModulus(B^{1/k}, N, x, counter),
        counter ← counter + j
    until a_i ≠ a_l for all l < i
end for
return ((a_1, b_1), ..., (a_k, b_k)).
```

Algorithm 3 Hash to an imaginary class group: $H_{\Delta}^{k}(x)$

Require: $k \geq 1$, $\Delta < 0, \Delta \equiv 1 \pmod 4$, $|\Delta|$ is prime and $x \in \{0,1\}^*$.
Ensure: $(a, b, c) \in Cl(\Delta)$ is a reduced quadratic form.

```
((a_1, b_1), ..., (a_k, b_k)) ← SampleModuli(k, sqrt(|Δ|)/2, Δ, x)
a ← ∏_{i=1}^{k} a_i.
Find b ∈ {0, ..., a − 1} with b ≡ b_i mod a_i for all i.   ▷ Using the Chinese Remainder Theorem.
if b is even then                        ▷ Ensure that b is a square root of Δ mod 4.
    b ← b − a                            ▷ b may be negative.
end if
c ← (b^2 − Δ)/(4a)
return (a, b, c).
```

We also assume there is a function $\textsc{ModSqrt}(k, m)$ for positive integers k and m, m being a prime number, which returns the smallest s such that $s^2 \equiv k \bmod m$ with $0 \leq s < m$, assuming this exists. In particular, it is the smallest s and $m - s$ for any square root $0 \leq s < m$.

Lemma 2. *The output of Algorithm 3 is a reduced quadratic form with discriminant Δ.*

Proof. The sampling method for a ensures that Δ is a quadratic residue modulo a and the Chinese Remainder Theorem ensures that b is a square root of Δ modulo a and that b is odd so $b^2 \equiv 1 \bmod 4$. This ensures that the division when computing c is exact and that (a, b, c) has discriminant Δ. We also see that $a < \sqrt{|\Delta|}/2$ and $-a < b < a$ so (a, b, c) is reduced by Lemma 1.

3.2 Complexity

Assuming that multiplication of two numbers smaller than $|\Delta|$ can be done in $O(\log^2|\Delta|)$ steps[1] we get the following complexity estimate. If we use the Miller-Rabin primality test[2], the complexity of checking a single a_i is $O(\frac{r}{(2k)^3}\log^3|\Delta|)$ where r is the number of rounds. Computing the Legendre symbol can be done in $O(\frac{1}{(2k)^2}\log^2|\Delta|)$ steps using a Euclidean Algorithm-like implementation (see Algorithm 2.3.5 in [11]).

The expected number of a_i's to consider before finding a prime is $O(\frac{1}{k}\log|\Delta|)$ under the Cramér random model, and the probability that the Legendre symbol $= 1$, is $1/2$, so the expected complexity of the entire loop is $O(\frac{r}{(2k)^4}\log^4|\Delta|)$.

The modular square root may be computed using the Tonelli-Shanks algorithm which has expected complexity $O(\frac{1}{(2k)^2}\log^2|\Delta|)$. This is dominated by the loop to sample the a_i's, so the total time complexity of computing H^k_Δ is

$$O\left(\frac{r}{(2k)^4}\log^4|\Delta|\right).$$

Recalling that Wesolowski's construction is equivalent to the case where $k = 1$, we see immediately from the time complexity why increasing k improves performance significantly, even for small choices of k.

3.3 Parallelization

Algorithm 2 may be parallelized by running the loop to sample the a_i's in multiple threads. This requires some alterations to the algorithm:

1. We need to check that no a_i can be sampled more than once. This may be done either by checking it after the loop and resample any duplications or by making the list of a_i's synchronized such that only one thread can write to it at a time. In practice, it is very unlikely it will happen when the discriminant is large.
2. The counter variable needs to be handled such that the same value is not used twice. This may be done by allowing the i'th thread to only use use values for the counter that are of the form $qk + i$ for and then increment q instead.

The actual performance benefit of parallelizing the algorithm is analysed in Section 5.3.

4 Security

4.1 Security proof

Throughout this section we fix a discriminant Δ, a $k \geq 1$ and let $B = \frac{\sqrt{|\Delta|}}{2}$ be the upper bound for the a coefficient. We consider the output of Algorithm 3 as a hash function, $H^k_\Delta : \{0,1\}^* \to Cl(\Delta)$.

Recall that a function is a *random oracle* if it behaves like a uniformly distributed random variable on its range, e.g. that on any input it samples a uniformly random output from its range and returns it.

In this section we will prove the following theorem:

[1] This may be optimised to $O(\log|\Delta|\log\log|\Delta|)$ using FFT-based multiplication, but his is rarely efficient in practice so we use the slower estimate instead.

[2] In practice, the Miller-Rabin test should be combined with the Lucas primality tests to ensure security (see section 5.1), but for the complexity analysis here we will just consider the Miller-Rabin test.

Theorem 1. *If* $\textsc{RandomNumber}_X$ *is a random oracle for all* $X \geq 2$ *then* $H^k_\Delta : \{0,1\}^* \to \Omega^k_\Delta$ *is a random oracle where*

$$\Omega^k_\Delta = \{(a,b,c) \in Cl(\Delta) \mid a = \prod_{i=1}^{k} a_i, a_i \text{ prime}, 2 < a_i < B^{1/k}, |b| < a\}. \tag{1}$$

We consider a hash function to be secure if it is a random oracle and is collision-resistant. The security of the hash function H^k_Δ now follows from Ω^k_Δ being sufficiently large, which will be proven in section 4.2.

The proof of Theorem 1 follows from the fact that rejection sampling preserves a random oracle and because there is a one-to-one correspondence between the sampled a_i's and b_i's in $\textsc{SampleModuli}$ and the output of H^k_Δ. More formally, it will follow from Lemma 1 and the following two lemmas regarding the $\textsc{SampleModulus}$ and $\textsc{SampleModuli}$ functions.

Lemma 3. *If* $\textsc{RandomNumber}_X$ *is a random oracle for all* $X \geq 2$ *then*

$$\textsc{SampleModulus}'_c : \{0,1\}^* \to A^k_\Delta$$

defined by

$$\textsc{SampleModulus}'_c(x) = \textsc{SampleModulus}(B^{1/k}, \Delta, x, c)_1$$

and

$$A^k_\Delta = \{(a,b) \in \mathbb{N}^2 \mid a \text{ odd prime}, a < B^{1/k}, 0 < b < a, b^2 \equiv N \bmod a\} \tag{2}$$

is a random oracle for any $c \in \mathbb{N}$.

Proof. Since $\textsc{RandomNumber}_{B^{1/k}}$ is a random oracle, the rejection sampling in the loop ensures that a is uniformly random among the odd primes smaller than $B^{1/k}$ where Δ is a quadratic residue. For each of these a, we have that since a is prime and Δ is a quadratic residue modulo a, Δ has exactly two square roots modulo a, and since $\textsc{RandomNumber}_2$ is a random oracle we choose b to be one of these uniformly at random.

Lemma 4. *If* $\textsc{RandomNumber}_X$ *is a random oracle for all* $X \geq 2$ *are random oracles then* $\textsc{SampleModuli}' : \{0,1\}^* \to B^k_\Delta$ *is a random oracle where*

$$\textsc{SampleModuli}'(x) = \textsc{SampleModuli}(B, \Delta, x)$$

and

$$B^k_\Delta = \{((a_1,b_1),\ldots,(a_k,b_k)) \mid a_i \text{ odd prime}, a_i < B^{1/k}, 0 < b_i < a_i, b_i^2 = \Delta \bmod a_i \\ \text{and } i \neq j \implies a_i \neq a_j\}. \tag{3}$$

Proof. This follows immediately from Lemma 3 since all calls to $\textsc{SampleModulus}$ are done with different values for the counter value, so the distributions of the (a_i, b_i)'s are independent.

Using the above lemmas, we can finally prove that the hash function acts as a random oracle on its image: The theorem follows from Lemma 1 and Lemma 4 because there is a one-to-one correspondence between B^k_Δ (modulo the 2^k different orderings of the elements) and Ω^k_Δ given by the steps in Algorithm 3.

4.2 Image size

It is crucial for the security of the hash function H^k_Δ that the output has sufficient entropy such that an adversary cannot simply guess the output. In practice we will need the size of the image of H^k_Δ to be at least $2^{2\lambda}$ to ensure that the computational effort to find a collision for H^k_Δ is at least 2^λ.

In the following approximation of the image size, we will ignore the error terms coming from the Prime Number Theorem and from counting the number of quadratic residues modulo a given prime in an interval, but we will give a heuristic estimation of the error terms at the end of this section.

Now, if we pick $\lambda > 0$ such that

$$\frac{kB^{1/k}}{2\ln B} > 2^{2\lambda/k} + k, \tag{4}$$

then the image size of the hash function satisfies

$$\#\Omega_{\Delta}^{k} > 2^{2\lambda}. \tag{5}$$

We show this as follows: The Prime Number Theorem states that there are approximately

$$\frac{kB^{1/k}}{\ln B}$$

candidates for each a_i chosen in Algorithm 2 which are prime. Half of these have Δ as a quadratic residue. No more than k of the candidates may have already been chosen for other a_i's, so we get that there are at least

$$\left(\frac{kB^{1/k}}{2\ln B} - k\right)^{k} > 2^{2\lambda}$$

different ways to pick a_i's. Since a is the product of these, we get by symmetry that the number of a's is the above divided by 2^k, but since each a has exactly 2^k different square roots, we get that $\Omega_{\Delta}^{k} > 2^{2\lambda}$ as desired.

Since B is typically a very large number, we may use the following inequality instead which is easier to use in practice: Note that (4) is satisfied if

$$k < \frac{\log B - 2\lambda}{\log\ln B + 1}.$$

To see this, first rearrange the inequality as

$$\log B - k - k\log\ln B > 2\lambda,$$

divide by k and add $\log k$ to both sides to get

$$\log k + \log B^{1/k} - 1 - \log\ln B > \frac{2\lambda}{k} + \log k.$$

Now, using that the logarithm is concave we get that the RHS is larger than $\log(2^{2\lambda/k}+k)$. Raising both sides to the power-of-2 and rearranging gives (4) as desired. Using this bound, we see that to get an image size of at least 2^{256} we may use values for k as shown in Figure 1.

Recall that we did not consider error terms in this approximation. Using the above approximation we get that $\log \#\Omega_{\Delta}^{k}$ is at least

$$B\left(\frac{k}{2\ln B}\right)^{k} + O\left(B\left(\frac{k}{2\ln B}\right)^{k+1}\right)$$

because the error term for each choice of a_i is $O(\frac{B^{1/k}}{\ln^2 B^{1/k}})$, assuming the Riemann Hypothesis [15].

Benchmarks suggests that $k \sim 32$ is near optimal in implementation for sufficiently large discriminants, while also ensuring that the range of the hash function is large – larger than 2^{1300} for a 3072 bit discriminant.

5 Implementation and Benchmarks

The algorithm has been implemented as part of the Rust language based high-efficiency fastcrypto library [17]. In this section we provide comments and considerations for a concrete implementation.

5.1 Primality testing

Both the theoretical complexity analysis and profiling of the actual application suggests that the main bottleneck of computing the hash function presented in this paper is primality testing, so this has to be designed carefully.

In our implementation, we use the Baillie-PSW probabilistic primality test [4]. This was chosen over using just a Miller-Rabin test because the latter is vulnerable to an attack when used on candidates that may have been chosen by an adversary [2].

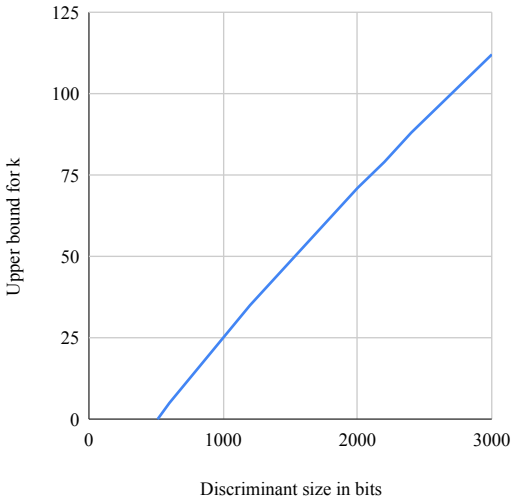

Fig. 1. Plot of the largest possible k while ensuring that the image size of H_{Δ}^{k} is at least 2^{256}. The horizontal axis shows $\log|\Delta|$.

5.2 Legendre symbol

In Algorithm 1, the Legendre symbol $\left(\frac{\Delta}{a}\right)$ is computed to ensure that Δ is a square modulo a. First note that this is indeed a Legendre symbol because a is a prime, so it may be computed as

$$\left(\frac{\Delta}{a}\right) = \Delta^{\frac{a-1}{2}} \bmod a. \tag{6}$$

Assuming that a multiplication of two numbers not larger than a takes $O(\log^2 a)$ operations, this formula gives an algorithm which runs in $O(\log^3 a)$ operations, but there is a faster algorithm, similar to the Euclidean Algorithm (see e.g. [10, pp. 29–31] or [11, p. 98]), which takes $O(\log^2 a)$ operations. The latter is the approach used in our implementation.

There is an alternative way to compute the Legendre symbol which is faster but reduces the range of the hash function slightly: Recall that $\Delta < 0$ is the fixed discriminant and that $-\Delta$ is prime, so if we restrict a to $a \equiv 3 \bmod 4$ we get from the Law of Quadratic Reciprocity that

$$\left(\frac{\Delta}{a}\right) = \left(\frac{a}{-\Delta}\right) = a^{\frac{-\Delta-1}{2}} \mod (-\Delta).$$

Now the modulus is fixed, so we can use the Montgomery Exponentiation [16] to compute the exponent which avoids modular reduction in each step in the exponentiation loop.

5.3 Parallelization

Theoretically, the bottleneck of the algorithm is the sampling of the a_i, so running that loop in parallel should give close to a linear improvement for large k. However, in practice the performance gain is only up to $2\times$ better on 6 cores for our implementation (see Figure 2). We suspect that this is due to contention and because instantiating $k-1$ more RNG's takes extra time.

5.4 Benchmarks

The implementation has been benchmarked on a MacBook Pro Laptop with an M1 Pro processor with 8 cores and 16 GB RAM, and the results are shown in Figure 2.

As we also expected from the theoretical complexity analysis, even with small k, the performance difference is significant. For a 2400 bit discriminant, setting $k = 64$ gives a $273\times$ improvement in performance. We also highlight that for larger k, the relative performance improvement gets smaller, and hence the performance degrades for very large k, so some fine tuning is needed to get optimal performance.

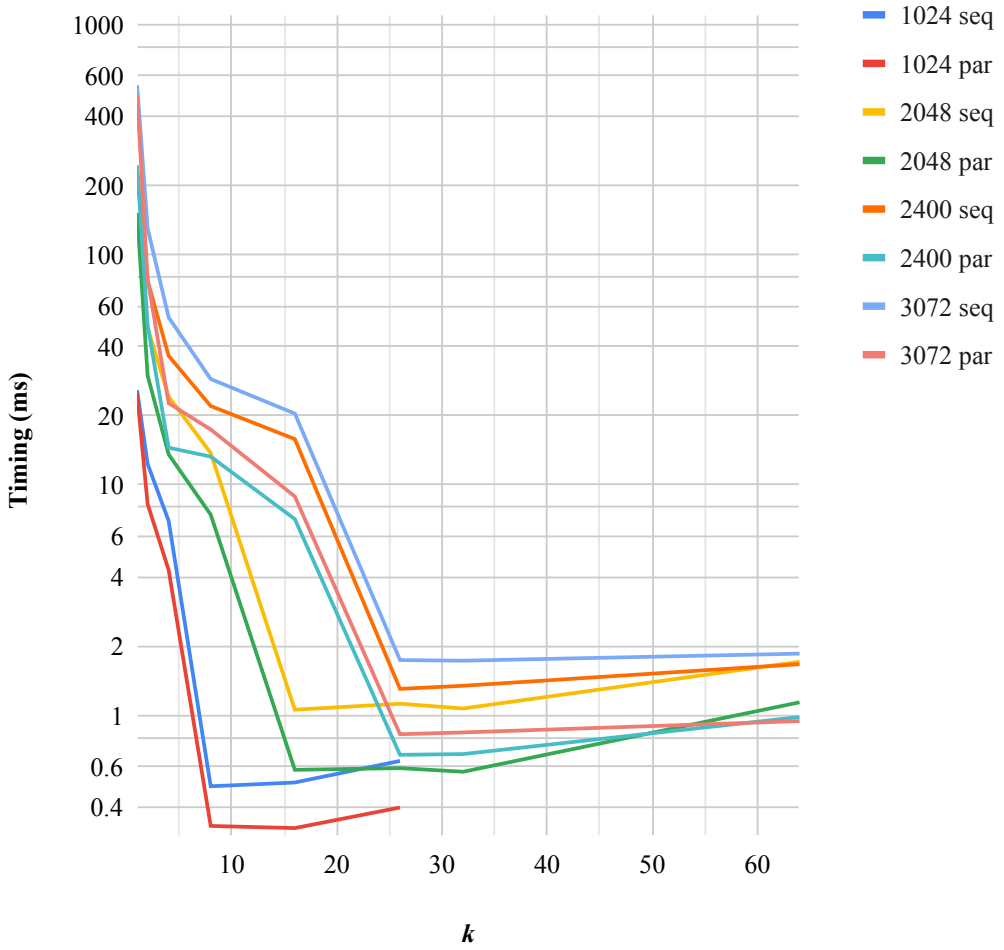

Fig. 2. Plots of performance for different discriminant sizes and k. Note that the vertical axis is on a logarithmic scale, and we cannot use $k > 26$ for 1024 bit discriminants because this will give a too small range. For a 3072 bit discriminant we get a 582× performance improvement from $k = 1$ to $k = 26$. To ensure a sufficiently large image size of the hash function, the implementation in fastcrypto ensures that k is not chosen too large and by default it sets $k = 26$ because it ensures that the image is sufficiently large and that the performance is optimal.

6 Applications

Imaginary class groups are used in a wide array of applications, and for a lot of these, for example timed commitments [21], accumulators [6], and, perhaps most notably, verifiable delay functions (VDFs) [22,18], a secure and efficient hash function to the class group is an important primitive.

Recall that a VDF is a function $F : G \to G$ defined by $g \mapsto g^{2^T}$ for some large T. Computing this function takes T group operations if the order of the group is unknown, but it is also possible to derive a proof that the computation was done correctly, which is fast to verify.

As noted in Remark 3 in [22], using a hash function to pick the input of the VDF is important, because knowing the result of $F(x) = x^{2^T}$ makes it easy to compute, for example,

$$F(x^a) = (x^a)^{2^T} = (x^{2^T})^a = F(x)^a.$$

At the time of writing, the only VDF in production is run by the Chia Network [9], where VDFs are used in a proof-of-time consensus protocol. However, Chia Networks' deployment does not use a hash function to generate a random input to a VDF. Instead it uses a fixed input to the VDF and samples a new random discriminant for each VDF instance.

The Sui blockchain also has a VDF implementation, and while implementing this we have found that sampling sufficiently large (according to [12]) discriminants at random may take several

seconds, which is too slow for on-chain usage. Due to this limitation, a fixed discriminant is used instead so the input to the VDF must be sampled for each instance instead based on user-provided randomness. This requires a hash function to which maps to a class group element, which is fast enough for it to be computed on-chain, but Wesolowski's hash function construction takes up to a second to compute, which is too slow for on-chain usage, so we use the construction presented in this paper which computes a hash in as little as 2 ms (see figure 2) for a large 3072 bit discriminant.

7 Future work

In certain applications, it is imperative to ensure that hash functions operate in constant time to prevent any leakage of information about the input. The challenge with the algorithm described in this paper lies in its dependency on probabilistically sampling random primes from a range, which inherently varies in time. Interesting lines of research could include:

(a) uniform sampling techniques for developing an efficient, constant time algorithm,
(b) pre-computations by preparing a pool of random primes in advance,
(c) time-padding methods where the execution time is artificially extended to be fixed.

The algorithm can be further optimised by using a smaller bound $C = B' < B^{1/k}$ for the a_i's in Algorithm 2. This bound has to be chosen such that the range of the hash function is still larger than $2^{2\lambda}$. Adding extra parameters complicates the range estimate and it will require a careful analysis to ensure that the resulting hash function achieves a sufficient security level.

References

1. Adleman, L.M., Rivest, R.L., Shamir, A.: Cryptographic communications system and method. US Patent No. 4,405,829. (Sep 1983), https://www.google.com/patents/US4405829, patent filed 14 September 1977.
2. Albrecht, M.R., Massimo, J., Paterson, K.G., Somorovsky, J.: Prime and prejudice: Primality testing under adversarial conditions. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 281–298. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243787, https://doi.org/10.1145/3243734.3243787
3. Bach, E.: How to generate factored random numbers. SIAM Journal on Computing **17**(2), 179–193 (1988). https://doi.org/10.1137/0217012, https://doi.org/10.1137/0217012
4. Baillie, R., Wagstaff, S.S.: Lucas pseudoprimes. Mathematics of Computation **35**(152), 1391–1417 (1980), http://www.jstor.org/stable/2006406
5. Beaver, D., Chalkias, K., Kelkar, M., Kokoris-Kogias, L., Lewi, K., de Naurois, L., Nikolaenko, V., Roy, A., Sonnino, A.: Strobe: Streaming threshold random beacons. In: 5th Conference on Advances in Financial Technologies (AFT 2023). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2023)
6. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 561–586. Springer International Publishing, Cham (2019)
7. Bünz, B., Fisch, B., Szepieniec, A.: Transparent snarks from dark compilers. In: Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39. pp. 677–706. Springer (2020)
8. Chen, M., Doerner, J., Kondi, Y., Lee, E., Rosefield, S., Shelat, A., Cohen, R.: Multiparty generation of an rsa modulus. Journal of Cryptology **35**(2), 12 (2022). https://doi.org/10.1007/s00145-021-09395-y, https://doi.org/10.1007/s00145-021-09395-y
9. Cohen, B., Pietrzak, K.: The chia network blockchain (2019), https://api.semanticscholar.org/CorpusID:209373416
10. Cohen, H.: A Course in Computational Algebraic Number Theory. Springer Publishing Company, Incorporated (2010)
11. Crandall, R., Pomerance, C.: Prime numbers. A computational perspective. Springer-Verlag, New York (2001)
12. Dobson, S., Galbraith, S., Smith, B.: Trustless unknown-order groups. Mathematical Cryptology **1**(2), 25–39 (Mar 2022), https://inria.hal.science/hal-02882161, https://eprint.iacr.org/2020/196.pdf
13. Gauss, C., Waterhouse, W.: Disquisitiones Arithmeticae. Springer-Verlag (1986), https://books.google.dk/books?id=Y-49PgAACAAJ
14. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA (1997)
15. Montgomery, H.L., Vaughan, R.C.: Multiplicative Number Theory I: Classical Theory. Cambridge Studies in Advanced Mathematics, Cambridge University Press (2006)

16. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985), https://api.semanticscholar.org/CorpusID:119574413
17. Mysten Labs: fastcrypto, https://github.com/MystenLabs/fastcrypto
18. Pietrzak, K.: Simple verifiable delay functions. In: 10th innovations in theoretical computer science conference (itcs 2019). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2019)
19. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
20. Seres, I.A., Burcsi, P., Kutas, P.: How (not) to hash into class groups of imaginary quadratic fields? Cryptology ePrint Archive, Paper 2024/034 (2024), https://eprint.iacr.org/2024/034, https://eprint.iacr.org/2024/034
21. Thyagarajan, S.A.K., Castagnos, G., Laguillaumie, F., Malavolta, G.: Efficient cca timed commitments in class groups. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 2663–2684. CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484773, https://doi.org/10.1145/3460120.3484773
22. Wesolowski, B.: Efficient verifiable delay functions. J. Cryptol. **33**(4), 2113–2147 (oct 2020). https://doi.org/10.1007/s00145-020-09364-x, https://doi.org/10.1007/s00145-020-09364-x