Improved key recovery on the Legendre PRF

Novak Kaluđerović, Thorsten Kleinjung, Dušan Kostić

École polytechnique fédérale de Lausanne
Faculté Informatique et Communications
Station 14
Laboratory for cryptologic algorithms
CH-1015 Lausanne
Switzerland

Abstract. We give an algorithm for key recovery of the Legendre pseudorandom function that supersedes the best known algorithms so far. The expected number of operations is $O(\sqrt{p \log\log p})$ on a $\Theta(\log p)$-bit word machine, under reasonable heuristic assumptions, and requires only $\sqrt[4]{p \log^2 p \log\log p}$ oracle queries. If the number of queries M is smaller, the expected number of operations is $\frac{p \log p \log\log p}{M^2}$. We further show that the algorithm works in many different generalisations – using a different character instead of the Legendre symbol, using the Jacobi symbol, or using a degree r polynomial in the Legendre symbol numerator. In the latter case we show how to use Möbius transforms to lower the complexity to $O(p^{\max\{r-3,r/2\}} r^2 \log p)$ Legendre symbol computations, and $O(p^{\max\{r-4,r/2\}} r^2 \log p)$ in the case of a reducible polynomial. We also give an $O(\sqrt[3]{p})$ quantum algorithm that does not require a quantum oracle, and comments on the action of the Möbius group in the linear PRF case. On the practical side we give implementational details of our algorithm. We give the solutions of the 64, 74 and 84-bit prime challenges for key recovery with $M = 2^{20}$ queries posed by Ethereum, out of which only the 64 and 74-bit were solved earlier.

1 Introduction

The usage of Legendre symbols in a pseudorandom function is an idea originally proposed by Damgård [5]. We model the Legendre PRF as an oracle $\mathcal{O}$ that on input x outputs the Legendre symbol $\left(\frac{k+x}{p}\right)$, where the shift – k – is a secret key. Damgård conjectured that given a sequence of Legendre symbols of consecutive elements it is hard to predict the next one. Similar problems conjectured to be hard were also proposed [8], such as finding the secret key k while being given access to $\mathcal{O}$ and distinguishing $\mathcal{O}$ from a random function. So far no polynomial time algorithms were found for either of these problems and it is believed that they are hard. Until recently practical applications have been limited, primarily due to availability of much faster alternatives.

A recent result by Grassi et al. [8] sparked an interest in the Legendre PRF because it was found suitable as a multi-party computation friendly pseudorandom generator. This is mainly due to the homomorphic property of the Legendre symbol and the possibility of evaluating it with only three modular multiplications in arithmetic circuit multi-party computations, which makes it a very efficient MPC friendly PRF candidate.

There are plans to use this construction as a PRF for a proof of custody scheme in the Ethereum blockchain [7]. The proof of custody scheme requires a "*mix*" function i.e., a pseudorandom function that produces one bit of output. For this purpose the Legendre PRF was shown to be a great candidate because of its efficiency. In comparison, SHA256 requires tens of thousands of multiplications while AES needs 290 in the MPC setting.

In order to raise interest for this construction, Ethereum research [7] posted a number of challenges online. The goal was to recover the secret key given $M = 2^{20}$ consecutive Legendre symbols, for primes of size varying from 64 to 148 bits.

1.1 Contribution

In this paper we give an algorithm that recovers the secret key of the Legendre PRF in $O(\sqrt{p \log\log p})$ operations on a $\Theta(\log p)$-bit architecture and by using only $\sqrt[4]{p \log^2 p \log\log p}$ queries of the PRF. The

main advantages of our algorithm with respect to the previous best algorithm (cf. Khovratovich [10] and [2] that was independently published while this paper was being written) are multiple. Firstly, in [10] and [2] the run time depends linearly in the cost of Legendre symbol evaluations and queries, while for us this cost can be ignored. If t is the number of operations needed to compute a Legendre symbol or query an oracle, then the key extraction effort is lowered from $O(\sqrt{p}\ t \log p)$ in [10] and $O(\sqrt{p} \log p)$ in [2] to $O(\sqrt{p \log \log p})$. Secondly, if the number of oracle calls is bounded by M, we lower the number of operations from $O(\frac{p\ t \log p}{M})$ [10] and $O(\frac{p \log^2 p}{M^2})$[2] to $O(\frac{p \log p \log \log p}{M^2})$. Furthermore we require access only to sequential oracle queries, while [10] asks for short sequential queries in random positions.

An important contribution is the algorithm for solving the higher degree Legendre PRF. We considered the action of the group of Möbius transformations on the set of degree r polynomials which allowed us to lower the complexity of key extraction from $O(p^{\max\{r-1,r/2\}} r \log p)$[10] and $O(p^{\max\{r-2,r/2\}} r^2 \log^2 p)$[2] down to $O(p^{\max\{r-3,r/2\}} r^2 \log p)$ Legendre symbol computations with further improvements on reducible polynomials. In the linear case, however, we only consider the action of a subgroup of the full Möbius group allowing only a quadratic instead of a cubic increase in the number of sequences extracted from the oracle queries.

We give a rigorous analysis of the run time of our algorithm. The main bottleneck is the number of simple operations on $\log p$-bit words, such as word comparisons, shifts, ANDs, ORs and XORs. Therefore we analyse the total number of such operations and ignore the cost of Legendre symbol computations as the amount of work spent on computing them is negligible compared to the rest of the algorithm.

We further add a quantum attack without a quantum oracle, improving some bounds on questions asked in [12]. We also show how the attack applies to different generalisations of the Legendre symbol and finally we give a few ideas that so far we did not manage to exploit, but which might be useful for future research.

We also give the solutions to challenges 0, 1 and 2 of Ethereum research Legendre PRF for 64, 74 and 84-bit primes, the last of which we were the only to solve. In all cases we were given access to $M = 2^{20}$ Legendre symbols.

During the computation of the 84-bit challenge, another, independently written paper on the same subject, was published on e-print [2]. Both ideas are similar, however, with some crucial differences. Our run time is lower and does not depend on the cost of Legendre symbol computation. Because of this property our analysis is in-depth, does not contain any hidden logarithmic factors in big O's and carefully analyses the cost of sequence extraction. We argue further on the sequence correlation properties and act on multiplicative correlation. Implementation-wise, even though in theory the authors note that one can make a table of size M^2/L, their argumentation regarding sequence correlation led to reducing the table size to M^2/L^2 which further led to a loss of a factor of L in the run time. Regarding the higher degree Legendre PRF we go deeper into the theory by studying the Möbius group which fully expands the properties from the linear case. This makes our algorithm better by a factor of $p \log p$, and in the case of a reducible polynomial a factor of $p^3 \log p$.

1.2 Structure

In Section 2 we introduce some notation regarding pseudorandom functions, Legendre symbols and sequences. We then define some hard problems on which the security of the Legendre PRF is based.

In Section 3 we give our algorithm. We define the main algebraic properties of the Legendre symbols and show how they extend to Legendre sequences. The algorithm is divided in two parts - the precomputation stage and the search stage. The algorithm is a birthday attack – we precompute a table and then we choose random elements until we find a hit in the table. The similarity of the precomputation and the search stage can lead to some problems which we address and show how to fix.

In Section 4 we analyse the complexity of the algorithm. The complexity is given in the number of operations on a $\Theta(\log p)$-bit word machine. The costs of Legendre symbols and queries are ignored as they are negligible with respect to the rest of the algorithm. The precomputation and search stage are treated separately and the optimal run time is given under reasonable heuristic assumptions. We also show how the run time changes if a limited number of queries is available.

Section 5 treats the implementation of the algorithm. We show how it differs from the theoretical algorithm, and we explain the implementational details of the precomputation and the search. We also give some implementational tricks that gave very valuable constant time improvements.

In Section 6 we give the results of the experiments done. We give the keys of the 64, 74 and 84-bit prime challenges posted on the Ethereum website [7]. We discuss the difference between the expected and the run time we observed.

In Section 7 we treat some generalisations of the Legendre PRF. Mainly we show that the same algorithm applies to alternative definitions of the Legendre symbol, to the Jacobi symbol and to different characters of the finite field. We also give a quantum attack that does not require a quantum oracle. In the end we treat the higher degree Legendre PRF by showing how to exploit the action of the group of Möbius transformations on the set of monic polynomials.

We give some remarks on using the Möbius group in the standard Legendre PRF, and show why it does not give an improvement with respect to the first algorithm. We also show how the main algorithm can be modelled in the Möbius group scenario.

2 Background

Let p be a prime. Throughout the paper we suppose that the prime is public and everyone has access to it. [1] We denote with $\mathbb{Z}/p$ the ring of integers modulo p.

2.1 Notation

Pseudorandom Functions A pseudorandom function family $\{F_k\}_k$ is a set of functions with the same domain and codomain indexed by the set of keys k such that a function F_k chosen randomly over the set of k-values cannot be distinguished from a random function.

Legendre symbol We define the Legendre symbol by setting

$$\left(\frac{x}{p}\right) = x^{\frac{p-1}{2}} = \begin{cases} 1 \text{ if } x \in (\mathbb{Z}/p)^* \text{ is a square mod } p \\ 0 \text{ if } x = 0 \text{ mod p} \\ -1 \text{ if } x \in (\mathbb{Z}/p)^* \text{ is not a square mod } p. \end{cases}$$

Some authors prefer to set $\left(\frac{0}{p}\right) = 1$, which makes the Legendre symbol a binary function but breaks the multiplicative property. As this occurs for only one out of p elements, we show that by sticking to our definition and ignoring the case $x = 0$, we can cover both definitions and our algorithm still works with high probability.

Legendre PRF The Legendre pseudorandom functions are functions F_k from $\mathbb{Z}/p$ to $\{-1, 0, 1\}$ indexed by $k \in \mathbb{Z}/p$ and defined as

$$F_k(x) = \left(\frac{k+x}{p}\right).$$

Throughout the paper we assume to be given access to a pseudorandom function denoted by $\mathcal{O}$ with unknown key k, and the goal is to find k.

[1] Originally, as proposed by Damgård, the prime was considered secret. We chose only to pursue the case of a public prime, as in the MPC use case.

Legendre sequence We define a Legendre sequence with starting point a and length L to be the sequence of Legendre symbols evaluated at consecutive elements starting from a. We denote it with $\{a\}_L$.

$$\{a\}_L := \left(\frac{a}{p}\right), \left(\frac{a+1}{p}\right), \left(\frac{a+2}{p}\right), \ldots, \left(\frac{a+L-1}{p}\right)$$

Every a fully determines its sequence of length L, but not vice versa – that property depends on L. For example the sequences with $L = 1$ just gives us information on quadratic residuosity of a, but nothing more. In general, these sequences are as well distributed as one can hope them to be. We know already that when $L = 1$ *"half"* of the elements give 1, and the other *"half"* give -1. Similar properties are true for larger L, and in general, following a theorem of Davenport around one in 2^L elements of $\mathbb{Z}/p$ is a starting point of a given sequence of length L.

Theorem 1 (Davenport, 1933). *Let S be a finite sequence of ± 1's of length L. Then the number of elements of $\mathbb{Z}/p$ whose sequence is equal to S satisfies*

$$\#\Big\{a \in \mathbb{Z}/p \;\Big|\; \{a\}_L = S\Big\} = \frac{p}{2^L} + O(p^\varepsilon)$$

where $0 < \varepsilon < 1$ is a constant depending only on L.

Throughout the paper we assume that L is such that $\{a\}_L$ uniquely defines a, i.e., that the following holds

$$\{a\}_L = \{b\}_L \text{ if and only if } a = b. \tag{1}$$

It is easy to see that if we want this property to hold, we need $L = \Omega(\log_2 p)$. Unfortunately the only provable upper bound we have comes from the Weil bound [14] and is $L = O(\sqrt{p}\log p)$ which is exponential.

On the other side, a result of Bach [1] tells us that if $L = \log_2 p$ and $p = 3 \mod 4$, then at most $\sqrt{p}\log_2 p$ elements can share the same sequence of length L. Russell and Shparlinski [12] show that for $L = (\log_2 p)^2$, at most $O(\sqrt{p}\log p)$ different elements can have the same sequence of length L, for all p.

Furthermore our computational results, together with other statistical data on the distribution of Legendre sequences [5], indicate that on average over all sequences S, there are $\frac{p}{2^L} + O(1)$ elements whose Legendre sequences are equal to S. In other words for a random S and a random j, we have $\{j\}_L = S$ with probability $\frac{1}{2^L}$.

The complexity of our algorithm is given as a function of L such that it satisfies property (1). A good estimate of L in terms of p is $L = 2\log_2 p$.

Complete Legendre sequence We define the complete Legendre sequence to be the sequence of p Legendre symbols of all ordered elements of $\mathbb{Z}/p$ up to rotation, i.e. $\{0\}_p$ where the tail connects to the head

$$\left(\frac{0}{p}\right), \left(\frac{1}{p}\right), \left(\frac{2}{p}\right), \ldots, \left(\frac{p-1}{p}\right) / \sim .$$

The Legendre sequences $\{a\}_L$ for all $a \in \mathbb{Z}/p$ and $L \geq 0$ are subsequences of the complete Legendre sequence.

2.2 Hard Problems

There are three main problems conjectured to be hard, and on which the security of the Legendre PRF is based.

Definition 1 (Shifted Legendre Symbol Problem - SLSP). *Let k be a uniformly random value in $\mathbb{Z}/p$. Given access to an oracle $\mathcal{O}$ that on input $x \in \mathbb{Z}/p$ computes $\mathcal{O}(x) = \left(\frac{k+x}{p}\right)$, find k.*

Definition 2 (Decisional Shifted Legendre Symbol Problem - DSLSP). *Let k be a uniformly random value in $\mathbb{Z}/p$. Let $\mathcal{O}_0$ be an oracle that on input $x \in \mathbb{Z}/p$ computes $\mathcal{O}_0(x) = \left(\frac{k+x}{p}\right)$, and let $\mathcal{O}_1$ be an oracle that on input x outputs a random value in $\{-1, +1\}$. Given access to $\mathcal{O}_b$ where b is an unknown random bit, find b.*

Definition 3 (Next Symbol Problem - NSP). *Given a Legendre sequence $\{a\}_M$ of $M = \text{polylog}(p)$ symbols, find $\left(\frac{a+M}{p}\right)$, or equivalently find $\{a\}_{M+1}$.*

It is easy to see that the SLSP and NSP are at least as hard as DSLSP. In the other direction, following a theorem of Yao [11] on general pseudorandom functions, predicting the next bit of a pseudorandom function is as hard as distinguishing it from a truly random one. Therefore $NSP = DSLSP \leq SLSP$, under polynomial time reductions. We give an $O(\sqrt{p \log\log p})$ attack on $SLSP$ and $DSLSP$, while for NSP the run time depends on M and varies from $O(\sqrt{p \log\log p})$ to $O(\frac{pL \log\log p}{M^2})$.

3 Algorithm

We give our algorithm in the scenario of attacking a Shifted Legendre Symbol Problem. The reader can easily generalise it to DSLSP and NSP. We assume that we are given access to an oracle $\mathcal{O}$ that computes $\left(\frac{k+x}{p}\right)$ on input x, and we want to find k. Let M be the number of oracle calls. It is assumed that M is larger than L.

The general idea is the following - we call the oracle multiple times in order to obtain many Legendre sequences $\{k_i\}_L$ with the property that from k_i we can find k. We store these sequences in a table and then we compute Legendre sequences of random elements j until $\{j\}_L = \{k_i\}_L$ for some k_i. By property (1) $j = k_i$, and from this we can find k.

This simple birthday algorithm is optimal when the table contains $\sqrt{p}$ sequences. However one needs to take into account the cost of creating the table – which depends on the number of oracle calls, the cost of computing $\{j\}_L$ – which depends on the cost of Legendre symbol computations, and the cost of table lookups – which is a couple of operations on an L-bit word machine.

A simple way to proceed is to compute a sequence from the table with L oracle calls, and to compute $\{j\}_L$ with L Legendre symbol computations. However we show that one can do much better, and reduce this cost to $o(1)$ oracle calls per sequence and $o(1)$ Legendre symbol computations per sequence. This transfers the bottleneck to bit operations such as sequence comparisons which are much cheaper than Legendre symbol computations.

By doing this we decrease the number of oracle calls necessary to create a table of a certain size. This is of interest in the cases where we are being given limited access to the oracle. In the Legendre PRF challenge we only have the first 2^{20} symbols following $\left(\frac{k}{p}\right)$. We will show how to create a table of size $O(M^2/L)$ with M oracle calls.

The algorithm is divided in two parts - the precomputation stage and the random guess stage. They both rely on some fundamental properties of Legendre sequences, so we start by explaining those first.

3.1 Sequence properties

Given a Legendre sequence $\{a\}_M$ of length $M \geq L$, we can extract $\{a\}_L$ from it. The main question is - how many other Legendre sequences can we find in $\{a\}_M$? How do they relate to a? There are three main properties that allow us to extract more Legendre sequences from $\{a\}_M$.

Shifting property Each subsequence of $\{a\}_M$ of L consecutive symbols corresponds to the Legendre sequence of $a+i$ for some shift i. As long as $0 \le i \le M-L$, then $\{a+i\}_L$ is a subsequence of $\{a\}_M$.

$$\{a+i\}_L = \{a\}_M \text{ from } i\text{'th to } L-1+i\text{'th element}$$
$$\text{or}$$
$$\{a+i\}_L = \{a\}_{L+i} \text{ from } i\text{'th to last element}$$

This allows us to obtain the sequences $\{a+i\}_L$ for $i = 0, 1, \ldots, M-L$.

Multiplicative property It is well known that the Legendre symbol is a totally multiplicative function, in other words $\left(\frac{a}{p}\right)\left(\frac{d}{p}\right) = \left(\frac{ad}{p}\right)$. This relates to Legendre sequences of a and d in the following way: the sequence of Legendre symbols with starting point ad, common difference $d \ge 1$ and length L, i.e.,

$$\left(\frac{ad}{p}\right), \left(\frac{ad+d}{p}\right), \left(\frac{ad+2d}{p}\right), \ldots, \left(\frac{ad+(L-1)d}{p}\right)$$

is equal to the Legendre sequence of a of length L multiplied by $\left(\frac{d}{p}\right)$, i.e.,

$$\left(\frac{d}{p}\right)\{a\}_L \coloneqq \left(\frac{d}{p}\right)\left(\frac{a}{p}\right), \left(\frac{d}{p}\right)\left(\frac{a+1}{p}\right), \left(\frac{d}{p}\right)\left(\frac{a+2}{p}\right), \ldots, \left(\frac{d}{p}\right)\left(\frac{a+(L-1)}{p}\right).$$

We may also denote this in a dual manner – a sequence of L Legendre symbols starting from a with common difference d is equal to the Legendre sequence of a/d where every element is multiplied by $\left(\frac{d}{p}\right)$.

$$\left(\frac{d}{p}\right)\{a/d\}_L = \left(\frac{a}{p}\right), \left(\frac{a+d}{p}\right), \left(\frac{a+2d}{p}\right), \ldots, \left(\frac{a+(L-1)d}{p}\right).$$

The sequence $\left(\frac{d}{p}\right)\{a/d\}_L$ is a subsequence of $\{a\}_M$ as long as $(L-1)d \le M-1$, in other words $d \le D_M \coloneqq \left\lfloor \frac{M-1}{L-1} \right\rfloor$. This allows us to obtain $\{a/d\}_L$ for $d = 1, 2 \ldots, D_M$ by computing $\left(\frac{d}{p}\right)$ for all d's, and extracting symbols from $\{a\}_M$ at indices $0, d, 2d, \ldots, (L-1)d$.

Reverse sequence property Suppose that we have a Legendre sequence

$$\{a\}_L = \left(\frac{a}{p}\right), \left(\frac{a+1}{p}\right), \left(\frac{a+2}{p}\right), \ldots, \left(\frac{a+L-1}{p}\right).$$

Then, the reverse sequence, after multiplying it element-wise by $\left(\frac{-1}{p}\right)$, is the Legendre sequence of $-(a+L-1) = -a-(L-1)$

$$\{-a-(L-1)\}_L = \left(\frac{-a-L+1}{p}\right), \left(\frac{-a-L+2}{p}\right), \ldots, \left(\frac{-a-1}{p}\right), \left(\frac{-a}{p}\right)$$

We may think of this as of a generalisation of the homomorphic property to negative denominators. This property allows us to obtain one extra sequence gratis for each sequence that we have.

Combining all properties The three properties can be combined to vastly increase the number of Legendre sequences obtainable from $\{a\}_M$. Consider an arithmetic sequence of length L starting from $a+i$ and of common difference d. Legendre symbols of this sequence are

$$\left(\frac{a+i}{p}\right), \left(\frac{a+i+d}{p}\right), \left(\frac{a+i+2d}{p}\right), \ldots, \left(\frac{a+i+(L-1)d}{p}\right).$$

They can all be obtained from $\{a\}_M$ if $0 \leq i$ and $i + (L-1)d \leq M-1$. Furthermore, this sequence, multiplied (divided) by $\left(\frac{d}{p}\right)$ is equal to

$$\left\{\frac{a+i}{d}\right\}_L = \left(\frac{\frac{a+i}{d}}{p}\right), \left(\frac{\frac{a+i}{d}+1}{p}\right), \left(\frac{\frac{a+i}{d}+2}{p}\right), \ldots, \left(\frac{\frac{a+i}{d}+L-1}{p}\right).$$

Therefore, from $\{a\}_M$ we can obtain the Legendre sequences of $\frac{a+i}{d}$ for $d = 1, 2, \ldots, D_M = \left\lfloor \frac{M-1}{L-1} \right\rfloor$ and $i = 0, 1, \ldots, M-1-(L-1)d$. Furthermore we can also obtain the sequence of $-\frac{a+i}{d} - (L-1)$, by reversing this one and multiplying by $\left(\frac{-1}{p}\right)$. This increases the total number of Legendre sequences obtainable from $\{a\}_M$ up to

$$\sum_{d=1}^{D_M} \sum_{i=0}^{M-(L-1)d} 2 = 2MD_M - (L-1)D_M(D_M+1) = \frac{M^2}{(L-1)} - M + O(L)$$

where the constant in $O(L)$ is at most 2. For all these sequences, if we know their starting points, $\frac{a+i}{d}$ or $-\frac{a+i}{d} - (L-1)$, together with i and d, then we can find a.

3.2 Precomputation stage

The first part of the algorithm is the precomputation stage. It is done in two steps. Firstly we query $\mathcal{O}(x)$ for $x = 0, 1, \ldots, M-1$ in order to obtain $\{k\}_M$. Then we use the above mentioned sequence properties to extract $\frac{M^2}{(L-1)} + O(M)$ Legendre sequences out of $\{k\}_M$. These sequences are of the following two types:

$$\left\{\frac{k+i}{d}\right\}_L \text{ and } \left\{-\frac{k+i}{d} - (L-1)\right\}_L.$$

They are saved in a hash table, together with the corresponding i, d, and one extra bit to differentiate $\frac{k+i}{d}$ from $-\frac{k+i}{d} - (L-1)$. This finishes the precomputation stage.

3.3 Search stage

During the search stage we compute $\{j\}_L$ for many random j's, until we find a hit in the hash table. In that case we have $\{j\}_L = \{\frac{k+i}{d}\}_L$, and by property 1, it follows that $j = \frac{k+i}{d}$, which allows us to find $k = dj - i$. The scenario is the same in the case of $-\frac{k+i}{d} - (L-1)$.

As the table contains $\frac{M^2}{L-1} + O(M)$ sequences, we expect to find a hit after $p\frac{L-1}{M^2}$ trials. However every trial costs L Legendre symbol computations. We can improve this by using the same sequence properties as before, allowing us to lower the number of Legendre sequence computations per trial from L to $o(1)$.

After choosing a random $j \in \mathbb{Z}/p$, we compute the Legendre sequence of length N with starting point j. Given $\{j\}_N$ we can extract $\frac{N^2}{L-1} + O(N)$ sequences of type $\frac{j+a}{b}$ and $-\frac{j+a}{b} - (L-1)$ from it, with a and b satisfying similar constrains as i and d in the precomputation stage. However one needs to be more careful. The sequences that we extract are highly correlated with the sequences extracted from $\{k\}_M$, and they may not be considered as sequences obtained from uniformly random elements in $\mathbb{Z}/p$ when we do hash table checks. There are three main types of correlation.

Reverse sequence correlation If $\frac{j+a}{b} = \frac{k+i}{d}$ then $-\frac{j+a}{b} - (L-1) = -\frac{k+i}{d} - (L-1)$ and vice-versa. Therefore we do not compute the reverse sequences for the j's.

Shifting correlation If $\frac{j+a}{b} = \frac{k+i}{d}$ then $\frac{j+a+b}{b} = \frac{k+i+d}{d}$. In other words if $\frac{j+a}{b} \neq \frac{k+i}{d}$ then there is a lower chance for a hash collision for $\frac{j+a+b}{b}$. To combat this we reduce the number of sequences we extract from $\{j\}_N$ by only considering sequences for $\frac{j+a}{b}$ with $0 \leq a < b$. This way $\frac{j+a+b}{b}$ is never tested. However this reduces the number of sequences to

$$\sum_{b=1}^{D_N} \sum_{i=0}^{b-1} 1 = \frac{N^2}{2(L-1)^2} + O(\frac{N}{L-1}).$$

Multiplicative correlation If $\frac{j+a}{b} = \frac{k+i}{d}$ then $\frac{j+a}{b/f} = \frac{k+i}{d/f}$ for each divisor f of $\text{lcm}(d, b)$. We combat this by not allowing any nontrivial common divisors between the denominators d and b. This is done by changing the range of b's from $1, 2, \ldots, D_N = \lfloor \frac{N-1}{L-1} \rfloor$ to

$$b \in \{D_M + 1, D_M + 2, \ldots, D_N\} \cap \mathbb{P}$$

where $\mathbb{P}$ is the set of prime numbers, giving in total $O(\frac{N-M}{L} / \log(\frac{N}{L}))$ different b-values.

By putting everything together we obtain the following - out of $\{j\}_N$, we extract all sequences of type $\{\frac{j+a}{b}\}_L$ with $b \in \{D_M + 1, D_M + 2, \ldots, D_N\} \cap \mathbb{P}$ and $0 \leq a < b$. This gives rise to a total of

$$\sum_{\substack{b=D_M+1 \\ b \text{ prime}}}^{D_N} b = O\left(\frac{N^2}{L^3}\right)$$

Legendre sequences where we consider $N > 2M$ and $L = O(\log N)$, which is our use case. Therefore by computing N Legendre symbols we obtain $O(\frac{N^2}{L^3})$ sequences, implying that we compute $O(\frac{L^3}{N})$ Legendre symbols per sequence. As N is exponential in L, this cost becomes negligible and the lion's share of the work comes from extracting the sequences out of $\{j\}_N$ and look ups in the hash table.

4 Complexity of the algorithm

In order to give a precise estimate we will measure the run time in number of operations on an L-bit word architecture. The results are the same for any $\Theta(L)$-bit architecture. The following is assumed:

- Accessing a memory location costs $O(1)$;
- Comparing strings costs $O(1)$;
- Copying, shifting or writing a bit in an L-bit string costs $O(1)$;
- Building a hash table of size n costs $O(n)$;
- Hash table look up costs $O(1)$.

4.1 Precomputation stage

In the precomputation stage the following operations are performed:

- $\mathcal{O}$ is queried M times.
  One may assume that every query takes the time of a Legendre symbol computation, or that they are given for free. In either case this cost is negligible with respect to the rest of the algorithm.
- $O(M/L)$ Legendre symbols are computed.
  This is the cost of computing $\left(\frac{d}{p}\right)$ for all denominators $d = 1, \ldots, D_M$. Again, this will be negligible.
- $O(M^2/L)$ sequences are extracted.
  With each sequence of length L, we can extract all of them in $O(M^2)$ L-bit word operations. However one can do better. For each d we can extract $\{\frac{k+i}{d}\}_L$ for $0 \leq i < d$ in L operations each. For sequential values of i, we need only extract one extra bit per sequence because $\{\frac{k+i+d}{d}\}_L = \{\frac{k+i}{d} + 1\}_L$, and this can be obtained from $\{\frac{k+i}{d}\}_L$ by one shift and one extra symbol extraction. Therefore the cost is $O((M^2/L^2)L + M^2/L) = O(M^2/L)$.

– The hash table is made on the fly as the sequences are extracted, so this cost is $O(M^2/L)$.

This finishes the precomputation stage for a total run time of $O(M)$ Legendre symbol computations and $O(M^2/L)$ L-bit word operations.

4.2 Search stage

In the search stage the following operations are done

– A random element $j \in \mathbb{Z}/p$ is selected until we find a collision.
The number of j-values is referred to as c.
– N Legendre symbols are computed.
We obtain $\{j\}_N$ in N Legendre symbol computations.
– The range $\{D_M + 1, \ldots, D_N\}$ is sieved in order to extract the prime denominators b.
A rough estimate of the cost is $O(N \log N \log\log N)$ L-bit word operations, which stays negligible with respect to the last step of the algorithm.
– Legendre symbols $\left(\frac{b}{p}\right)$ are computed for the resulting prime b values.
There are in total $O(N/(L \log N/L)) = O(N/L^2)$ Legendre symbol computations to be done.
– $O(N^2/L^3)$ sequences are extracted and for each sequence one hash table look up is done.
These sequences can be extracted in $O((N^2/L^3)\frac{L \log L}{w}) = O((N^2/L^3) \log L)$ operations where w is the word size. For more information on this procedure see Appendix A.
– In the case of a hit we have $\frac{j+a}{b} = \frac{k+i}{d}$ or $\frac{j+a}{b} = -\frac{k+i}{d} - (L-1)$ from which we extract k in $O(1)$ modular operations.

Summing everything together, the search stage takes $O(c(N^2/L^3) \log L)$ operations on an L-bit word machine.

4.3 Run time hypothesis

We conjecture that each sequence extracted from $\{j\}_N$ has probability of $(M^2/L)/p$ of being inside the hash table, in other words we assume that the sequences extracted from $\{j\}_N$ behave as if they were Legendre sequences of uniformly random elements of $\mathbb{Z}/p$. The heuristic results seem to show that this is indeed the case. Therefore, the number of trials required until a hit is found is $p/(M^2/L)$, and so if the following formula is satisfied

$$\frac{M^2}{L}\frac{cN^2}{L^3} = p$$

we will find a hit with constant probability.

4.4 Optimal run time

The total run time is the sum of run times for both stages of the algorithm

$$\frac{M^2}{L} + \frac{cN^2}{L^3}\log L$$

under the hypothesis that

$$\frac{M^2}{L}\frac{cN^2}{L^3} = p.$$

This run time is optimised for the following values of M and N:

$$M = \sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L},$$

$$N = \sqrt[4]{p}\frac{L\sqrt{L}}{\sqrt{c}\sqrt[4]{\log L}},$$

$$\text{Run time} = \sqrt{p \log L}.$$

In this scenario we additionally determine $O(M + M/L + cN + N/(\log N)^2)$ Legendre symbols using oracle calls or by directly computing them. This is negligible with the above choices of M and N.

The variable c can be chosen freely as long as $c < L^2/\log L$.

4.5 Run time with a fixed M

In the case where we are given the possibility of only doing M queries, for some fixed M, the run time becomes a function of M. As seen above, if $M \geq \sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}$, then it is enough to do $\sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}$ queries, discard the rest and achieve a $\sqrt{p \log L}$ run time. Otherwise, if $M < \sqrt[4]{p}\sqrt{L}\sqrt[4]{\log L}$ the run time is dominated by the search stage, and it is

$$O\left(\frac{cN^2}{L^3}\log L\right) = O\left(\frac{pL}{M^2}\log L\right) = O\left(\frac{p\log p \log\log p}{M^2}\right)$$

assuming $L = O(\log p)$, therefore the run time is

$$O\left(\min\left\{\sqrt[4]{p}\sqrt{\log\log p}\,,\ \frac{p\log p\log\log p}{M^2}\right\}\right).$$

5 Implementation details

In this section we explain the concrete implementation and some subtle optimisations of the presented algorithm which we used to break the Legendre PRF challenges [7]. For each challenge, a sequence $\{k\}_M$ of $M = 2^{20}$ bits of output from the Legendre PRF was given. The challenge was to find the correct key k.

The proposed algorithm works in two stages, the *precomputation* and the *search* stage. During the *precomputation* stage a big hash table is generated, containing the short subsequences extracted from the given sequence of M bits. Later in the *search* stage many random short sequences are produced and checked against the entries in the hash table. Every obtained collision gives the correct key with a certain probability. The code can be found at

`https://github.com/nKolja/LegendrePRF` .

5.1 Precomputation stage

Since we are given the same number M of Legendre PRF oracle calls for every challenge, the *precomputation* stage is exactly the same for each instance of the challenge. In all the cases we set the length L of the short subsequences to be $L = 64$. We note that the primes in the challenges we solve have bit length larger than 64, and theoretically L should be set to approximately the bit length of the prime (as explained in Section 2.1), but for practical reasons we opt for $L = 64$ even for larger primes. The advantage of this decision is that we are working with 64-bit processor architectures so naturally all the operations as well as memory storage of the subsequences are much faster if their size is limited to the word size of the architecture. On the other hand, the proportion of false-positive hits is increased for larger p. However, the additional cost of validating the fake hits is negligible and heavily outweighed by the memory and run time savings obtained by choosing this trade-off.

From the given M bits we extract $\approx \frac{M^2}{L} = 2^{34}$ subsequences of length L as explained in Section 3.2. We define the hash table with 32-bit keys (i.e., a hash table key is an integer between 0 and $2^{32} - 1$). The table is then generated by taking the 32 least significant bits of each subsequence as the hash key and storing only the other 32 bits (the most significant bits) in the hash entry with address determined by the key (e.g., `table[seq & (`$2^{32} - 1$`)] = seq » 32`). By storing not the full sequence, but only the last 32 bits we halve the space required for the hash table. Obviously, since the number of sequences we have is larger than the number of keys in the hash table, some of the entries in the table should hold more than one subsequence. In order to minimise the memory usage we generate the table in two passes. In the first one, we extract all the subsequences and only count and store the number of different values for each hash key. After this, we get an array denoted by *positions* where for each key it holds its starting position in the table. Then in the second pass we allocate the required memory for the table, extract the subsequences again and populate the table based on the *positions* array. Each entry in the

table contains one or more values that is compared with many random values later in the *search* stage. Therefore, we have to either sort the values in each individual entry and do a binary search among the sorted values in the *search* stage, or leave the values as is in the table and perform a linear search with the guessed value in the *search* stage. We decided to sort the values. We note that either way does not affect the performance of the algorithm because in the concrete instances of the challenge we have that the average number of values sharing the same hash key is 4, which is small enough that all the values get stored in the CPU cache memory so both linear and binary search run in approximately the same number of operations.

The *positions* array stores 2^{32} pointers each pointing to a number in $[0, 2^{34} - 1]$. The i'th pointer points to roughly $4 \cdot i$ since each hash key holds 4 values on average. In order to save space we only store the last 32 bits of the pointer, and then we choose the full 34 bit value that is closest to $4 * i$ and has the last 32 bit equal to the ones saved. This allows us to fit the *positions* array in 16GB instead of 32GB if we use 64 bit values for pointers.

For the given parameter $M = 2^{20}$ and the chosen $L = 64$, the 2^{34} extracted sequences are stored in the hash table of size about 65GB, while the *positions* array occupies another 16GB. We also note that contrary to the theoretical algorithm given in Section 3.2 where the i and d of an extracted subsequence are stored in the hash table alongside the actual hash value, in practice we do not store i and d to minimise the memory required for the program to run. In the next section we explain how this is handled in the *search* stage. Additionally, if the amount of available memory permits we generate another data structure during the precomputation to further optimise memory accesses, namely a *bitmap*, as explained in the next section.

5.2 Search stage

In the *search* stage we generate random sequences of length L and query the hash table for collisions. As already explained in Section 3.3, computation of a single short sequence of L Legendre symbols is computationally expensive, see Table 1. Therefore, we apply the same technique for sequence extraction as in the *precomputation* stage with some rather important differences, vastly lowering the number of clock cycles per Legendre symbol.

Firstly, a random $j \in \mathbb{Z}/p$ is selected and the Legendre symbols of N consecutive values starting from j are computed. The number N of symbols to be computed is such that $N > 2 \cdot M$ as explained in Section 3.3. We proceed by extracting subsequences from the obtained $\{j\}_N$ sequence. Recall that subsequences extracted from $\{k\}_M$ and $\{j\}_N$ are of the form:

$$\left\{\frac{k+i}{d}\right\}_L \text{ and } \left\{\frac{j+a}{b}\right\}_L$$

respectively, where $d \in \{1, 2, \ldots, D_M\}$ for $D_M = \lfloor \frac{M-1}{L-1} \rfloor$. On the j side the denominators b are chosen from the range $[D_M + 1, D_N]$ such that b is prime and $D_N = \lfloor \frac{N-1}{L-1} \rfloor$. To select prime numbers in the relevant range we implemented a simple sieve. There is no need for a more sophisticated algorithm because the sieving range determined by M is rather small and the the sieving is done only once.

Given the $\{j\}_N$ list as an array of $\frac{N}{8}$ bytes and an (a, b) pair, the subsequence $\left\{\frac{j+a}{b}\right\}_L$ is extracted as shown in Algorithm 1, where we use 0 for quadratic residues, and 1 for non-residues. The same algorithm is used for extracting the subsequences in the precomputation stage. However, in the precomputation stage the extraction of multiple subsequences can be further optimised. Recall from Section 3 that the i and a parameters used for computing the $\left\{\frac{k+i}{d}\right\}_L$ and $\left\{\frac{j+a}{b}\right\}_L$ sequences are such that $i = 0, \ldots, M - 1 - (L-1) \cdot d$ and $a = 0, \ldots, b - 1$. Hence, when computing for example the $\left\{\frac{k+i}{d}\right\}_L$ and $\left\{\frac{k+i+d}{d}\right\}_L$, we note that those two sequences share $L - 1$ Legendre symbols. This allows us to amortise the cost of extraction of "consecutive" sequences by basically extracting only one bit per sequence for each string of such "consecutive" sequences.

We did not choose to further improve Algorithm 1 by implementing the algorithm from Appendix-A. This is due to the fact that at this point the time spent on producing sequences is dwarfed by the time spent on random memory access.

Table 1. Number of clock cycles required to obtain a Legendre symbol by computation and extraction, amortised.

Prime size [bits]	64	74	84	100	148
computing	460	650	780	950	2700
extracting from $\{k\}_M$	0.25	0.25	0.25	0.25	0.25
extracting from $\{j\}_N$	5.95	5.95	5.95	5.95	5.95

Algorithm 1

Input: Byte array j_list, pair (a, b), length L
Output: Sequence $\left\{\frac{j+a}{b}\right\}_L$

```
1: procedure EXTRACT_SEQ(j_seq, a, b, L)
2:     seq = 0
3:     for k = 0 to L − 1 do
4:         idx = a + k ∗ b
5:         byte_idx = idx >> 3
6:         bit_idx = 7 − (idx&7)
7:         bit = (j_list[byte_idx] >> bit_idx)&1
8:         seq = seq|(bit << j)
9:     end for
10:    if legendre(b) equals 1 then
11:        seq = ∼seq
12:    end if
13:    return seq
14: end procedure
```

Each obtained subsequence is checked against the hash table for a potential collision. The check is performed in the following three steps:

1. Compute the hash key of the sequence by taking only the 32 least-significant bits, and the hash value by taking the 32 most-significant bits of the sequence.
2. Read two values from the *positions* array (generated in the *precomputation* stage): value corresponding to the computed hash key and the next value. Recall that these two values are the address of the hash table entry corresponding to the key and the address of the next entry.
3. If the two addresses are different, it means that the table entry of the computed key is not empty. This happens with probability $1 - 1/e^4$. In that case we perform a binary search for the hash value in the range specified by the starting address of the entry and address of the next one.
4. In case of a collision, the subsequence and the variables that determine it (j, a, b) are stored for later validation.

Bitmap For each hash table lookup we do two random memory accesses – to the *positions* array and to the hash table. In order to decrease the number of random memory accesses we generate a bitmap during the *precomputation*. Each bit in the bitmap denotes if a sequence with a certain property appears in the hash table. More precisely, the m least significant bits of the sequence form an address, and the bit at this address in the bitmap signifies the existence of such a sequence in the table. So after the first step from above, we first read the appropriate bit in the bitmap and only when this bit is set we proceed with reading the *positions* array. If we suppose that the size of the bitmap is 2^m and the size of the hash table 2^h, the probability that a sequence produces a hit in the bitmap is 2^{h-m}. Consequently, instead of accessing the memory twice per trial we access the bitmap once per trial and with probability 2^{h-m} continue to access the *positions* array twice more. As a result the number of accesses per trial is reduced from 2 to $1 + \frac{2}{2^{m-h}}$. The size of the bitmap is determined based on the available memory, for example in the solution for the 84-bit prime challenge, we have used a bitmap consisting of 2^{37} bits so in particular we had $m = 37$ and $h = 34$.

After all the subsequences produced by the randomly selected j are checked against the hash table and all the collisions are saved, we proceed to the last step where the collisions are validated. As already noted above, the hash table holds only the actual sequences $\left\{\frac{k+i}{d}\right\}_L$ and not the (i,d) values. On the other hand, for each collision we have saved the sequence, the j and (a,b), which are furthermore sorted by the sequence value. Then again, as in the precomputation stage, we sequentially produce the $\left\{\frac{k+i}{d}\right\}_L$ and $\left\{-\frac{k+i}{d}-(L-1)\right\}_L$ subsequences and compare them to the sorted colliding sequences. Once a collision is obtained we can compute the guessed key with:

$$k = \frac{(j+a)\cdot d}{b} - i \quad \text{or} \quad k = -\frac{(j+a)\cdot d}{b} - d\cdot(L-1) - i$$

and check if it is indeed the correct key by computing $\{k\}_{192}$ and comparing to the first 192 symbols given in the challenge. If the key is not found, the algorithm simply chooses the next random j and repeats all the steps above.

6 Results

In this section we present the results of our attempt to break the Legendre PRF challenges posed by the Ethereum foundation [7]. In each challenge we are given a prime p and $M = 2^{20}$ bits of the sequence $\{k\}_M$ as defined in Section 2. The challenge is to recover the key k. The five challenges and their corresponding security levels are shown in Table 2. We note that security levels in the table are computed based on the complexity of the attack by Khovratovich [10], and that the algorithm proposed in this paper lowers those bounds. We have successfully solved the challenges #0, #1 and #2.

Table 2. Legendre PRF challenges with security levels estimated based on [10] and the new security estimates.

Challenge	Prime size [bits]	Security old [bits]	Security new [bits]
0	64	44	32
1	74	54	40
2	84	64	50
3	100	80	66
4	148	128	114

The algorithm was implemented in C and compiled with the gcc compiler. All the parallelisation was done with OpenMP primitives. Testing and experiments were conducted on a desktop PC equipped with an Intel Xeon E5-1650 processor with 6 cores running at 3.5GHz, and 128GB of RAM. The first two challenges (#0 and #1) were solved on this PC, while for the third challenge (#2) we used 16 nodes of the EPFL IC cluster. Each node has two Intel Xeon E5-2680 v3 processors with 12 cores each running at 2.5GHz, and 192GB of RAM.

The precomputation stage in all three cases took about 18 minutes on the desktop PC. During the precomputation we produce three files for each challenge:

- 65GB file containing the hash table,
- 16GB file containing the positions array,
- 16GB file containing the bitmap which we set at 2^{37} bits.

In the search stage, all three files are loaded in RAM so the program requires in total almost 100GB of memory.

In Table 3 we show the results of the experiment. The complexity of the search stage of our attack expressed as the expected number of trials that need to be done before the solution is found is $\frac{p\cdot L}{M^2}$, where by trial we denote a single hash table collision check with a random subsequence generated as

explained in 5.1. We show the expected number of trials for each challenge in the second column of Table 3, while the third column shows the actual number of trials performed by the algorithm before the solutions are found. For the first two challenges we run the algorithm several times with different seeds in order to record the required number of trials and validate the expected numbers given in the complexity analysis in Section 4. The numbers shown in the third column are the average of all the conducted experiments ($2^{30.78}$ and $2^{39.81}$ for challenges #0 and #1 respectively). We note that the observed variance is considerable, as expected. The expected number of trials for a prime p is pL/M^2 and the variance is $\sim p^2L^2/M^4$.

Table 3. Results and estimates for solving the Legendre PRF challenges. The expected and actual number of core-hours for challenges #0 and #1 is based on measuring the performance of the implementation on our desktop PC with Intel Xeon E5-1650 at 3.5GHZ, while the numbers for the other three challenges are based on performance of the cluster with Intel Xeon E5-2680 v3 at 2.5GHz.

Challenge	Expected # trials	Observed # trials	Expected core-hours	Observed core-hours	k
0	2^{30}	$2^{30.78}$	290 sec	490 sec	650282827113560997
1	2^{40}	$2^{39.53}$	82	59	16619470924565960259133
2	2^{50}	$2^{46.97}$	1.4e5	1.72e4	187320452088744099523844
3	2^{66}	-	9.1e9	-	-
4	2^{114}	-	2.5e24	-	-

The most interesting is of course challenge #2 since no other solutions have been published. As shown in Table 3, the actual number of trials that were done before the key was found is $2^{46.97} = 1.38e14$ which is far less than expected. This can be explained by the large variance and by sheer luck. The implementation version that was used for challenge #2 can perform $2.2e6$ trials per second on a single core of a processor in the EPFL IC cluster. The number of trials per second is slightly higher on the desktop PC since its CPU is working at a higher frequency. In Table 3 we also give estimates for the two most difficult challenges (#3 and #4), which are out of reach with the proposed attack and its implementation.

7 Generalizations

There are multiple ways in which the Legendre PRF can be generalised. We will show how the algorithm adapts to these use cases.

7.1 Alternative definition of the Legendre symbol

In the definition of the Legendre symbol we noted that some authors define it differently, by setting

$$\left(\frac{x}{p}\right) = \begin{cases} 1 \text{ if } x \text{ is a square mod } p \\ -1 \text{ if } x \text{ is not a square mod } p. \end{cases}$$

The only difference being that instead of our $\left(\frac{0}{p}\right) = 0$ some use $\left(\frac{0}{p}\right) = 1$. This makes the Legendre symbol a binary mapping at the cost of breaking the multiplicative property. Our algorithm works identically in both cases as long as we never compute $\left(\frac{0}{p}\right)$.

For every denominator $d = 1, 2, \ldots, D_M$ in all the $\{\frac{k+i}{d}\}_L$ sequences with $0 \leq i \leq M - (L-1)d$ there can be at most L sequences that contain $\left(\frac{0}{p}\right)$. By using the original definition, if we find $\left(\frac{0}{p}\right) = 0$ in the precomputed table, we immediately find k. With the alternative definition however, every sequence that contains $\left(\frac{0}{p}\right)$ could be incorrect. Furthermore we cannot be aware of that in the precomputed stage. We call such sequences *bad*. In the case of a table hit with such sequence, the key that we find will be wrong, but we can confirm that by checking if the first L symbols following that key correspond to the first L symbols of $\{k\}_M$. In total there are $O(\frac{M}{L}L) = O(M)$ *bad* sequences. For every j there is at most one *bad* sequence per denominator. In total the number of *bad* sequences is negligible with respect to the total number of sequences. Therefore, the algorithm terminates in the same expected number of steps.

7.2 Bad keys

If the key k is small, then an alternative algorithm can find the solution faster. We only precompute the sequences with denominator $d = 1$

$$\{k\}_L, \{k+1\}_L, \ldots, \{k+M-L\}_L$$

and then compare $\{j\}_L$ for $j = 0, s, 2s, 3s \ldots$ where $s = M - L + 1 = O(M)$. We will find a hit in $O(\frac{k}{M})$ steps, i.e., after $O(\frac{k}{M}L)$ Legendre symbol computations.

If we know a bound on k, for example $k + M < K$, then by trying random $j \in [0, K]$ we can lower the run time to expected $O(\frac{K}{M \log(M/L)}L) = O(\frac{K}{M})$ Legendre symbol computations. This is because in that case $M \log(M/L)$ of all precomputed elements are smaller than K. One can see this by noticing that for each d, the number of $\frac{k+i}{d}$'s that are contained in $[0, \lfloor \frac{k}{d} \rfloor + \frac{M}{d}] \subseteq [0, K]$ is $\frac{M}{d} - L + O(1)$. Summing over all the d's gives the above mentioned result.

Some keys can be extracted from the hash table during the precomputation stage. If there is a collision in the hash table, we have

$$\frac{k+i}{d} = \frac{k+a}{b}$$

and therefore

$$k = \frac{b \cdot i - d \cdot a}{d - b}.$$

Therefore for some bad keys – at most $O(M^3/L^2)$ of them – the key k will be found during the precomputation stage. These keys that are inside intervals of size $O(M^2/L)$ containing $\frac{i}{d} \mod p$ where $0 \leq i < d$, and almost all such elements will be found in the precomputation stage.

7.3 Different roots of unity

We can generalise the Legendre symbol by setting, for some $c \mid p-1$ and $\xi \in \mathbb{Z}/p$ a primitive c'th root of unity

$$\chi(x) := x^{\frac{p-1}{c}} = \begin{cases} \xi^i \text{ for some } i \in \mathbb{Z}/c \text{ if } x \in (\mathbb{Z}/p)^* \\ 0 \text{ if } x = 0 \text{ mod p.} \end{cases}$$

In this case the oracle is modelled as $\mathcal{O}(x) = \chi(k+x)$. Computing $\chi(x)$ gives $\log_2 c$ bits of information on x. For example, if $c = p-1$ we get the whole value of x by computing $\chi(x)$, making this construction a pretty bad PRF, and there are straightforward attacks if $\frac{p-1}{c}$ small.

In order to expect some resistance from subexponential attacks the minimum we have to ask is that $\frac{p-1}{c}$ is exponential in $\log p$.

For any value of c our algorithm works as before, as the sequences of χ-symbols satisfy all the required sequence properties. The complexity changes since we need $L = \Omega(\log_c p)$, and with the same heuristic assumptions, which we did not test, one expects that $L = 2\log_c p$ is enough.

7.4 Jacobi symbol

The Jacobi symbol is defined, for $n = \prod_i p_i$ and $x \in \mathbb{Z}/n$ as

$$\left(\frac{x}{n}\right) = \prod_i \left(\frac{x}{p_i}\right) = \begin{cases} \pm 1 \text{ if } x \in (\mathbb{Z}/n)^* \\ 0 \text{ if } x \notin (\mathbb{Z}/n)^*. \end{cases}$$

However computing it does not require the factorisation of n, and the evaluation of the symbol can be done in polynomial time with a gcd-like algorithm. The pseudo-random function is defined as in the Legendre symbol case $\mathcal{O}(x) = \left(\frac{k+x}{n}\right)$. Furthermore we may suppose that there are no small divisors of n since this would break the pseudo-randomness of $\mathcal{O}$.

As in the Legendre case, the Jacobi symbol is multiplicative, and sequences obtained from it satisfy the sequence properties from 3.1, therefore our algorithm works as intended. Further improvements in the case of a factorable n are given in [2] where the authors show how to extract the key modulo each factor and glue it with the Chinese remainder theorem.

7.5 Quantum

The problem of key extraction of the Legendre PRF with a quantum computer was studied by many authors [3] [9] [4] [12] and polynomial time quantum algorithms are given. However they all rely on the oracle being queried on state in superposition, i.e., a *quantum* oracle. We answer one of the questions asked - find a better algorithm without a quantum oracle.

In our algorithm we do not rely on the oracle to obtain Legendre sequences. We use it to gather a large number of Legendre symbols, from which we extract sequences. If the symbol extraction and birthday collision search is done on a quantum computer, complexity improvements can be achieved. Using Tani's claw finding algorithm [13], the precompute and search stage can be done in $O(\sqrt[3]{p})$ quantum operations instead of $O(\sqrt{p})$ classical operations. We have the following results:

If $M \geq \sqrt[6]{p}\sqrt{L}$ then key extraction complexity is

$$O(\sqrt[3]{p}) \text{ quantum operations and } O\left(\sqrt[3]{p}L\sqrt{L}\right) \text{ Legendre symbol computations.}$$

If $M \leq \sqrt[6]{p}\sqrt{L}$ then key extraction complexity is

$$O\left(\sqrt{\frac{pL}{M^2}}\right) \text{ quantum operations and } O\left(\sqrt{\frac{pL^4}{M^2}}\right) \text{ Legendre symbol computations.}$$

7.6 Higher degree Legendre PRF

Another way to generalise the Legendre PRF is to consider oracles where the polynomial in the numerator of the Legendre symbol is not a linear polynomial but a general polynomial of degree r, so instead of the secret key being an element k of $\mathbb{Z}/p$, it is sampled from the space of polynomials of degree r. Querying the oracle gives

$$\mathcal{O}_f(x) = \left(\frac{f(x)}{p}\right) \text{ where } f(x) = k_0 + k_1 x + \ldots + k_{r-1}x^{r-1} + k_r x^r.$$

It should be noted that we can consider $f(x)$ to be monic, as $\mathcal{O}_f(x)$ and $\mathcal{O}_{f/k_r}(x)$ are the same up to multiplication by $\left(\frac{k_r}{p}\right)$. The case of linear $f(x)$ reduces to the standard Legendre PRF.

Secondly, the polynomial $f(x)$ is considered up to multiplication by a square since the Legendre symbol kills the square factors of $f(x)$. This does not apply in the linear case as a degree one polynomial cannot have a square factor.

This means that the secret key space, i.e., the space from which we choose $f(x)$ is the space of monic polynomials modulo squares. The number of such polynomials equals $p^r - p^{r-1}$ for $r > 1$.

We suppose that the polynomial $f(x)$ is uniquely determined by

$$\{f\}_L := \left(\frac{f(0)}{p}\right), \left(\frac{f(1)}{p}\right), \left(\frac{f(2)}{p}\right), \ldots, \left(\frac{f(L-1)}{p}\right),$$

and in this case we have $L = \Omega(r \log p)$. We will assume that $\{f\}_L$ determines $f(x)$ uniquely for $L = O(r \log p)$.

Let $\mathcal{M}$ be the rational Möbius group, i.e., the group of rational automorphisms of $\mathbb{P}^1$ which is isomorphic to $\mathrm{PGL}_2(\mathbb{F}_p)$. Given a matrix $m = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) \in \mathrm{PGL}_2(\mathbb{F}_p)$ there is a unique Möbius transformation φ_m given by

$$\begin{aligned} \varphi_m : \mathbb{P}^1 &\longrightarrow \mathbb{P}^1 \\ x &\longmapsto \tfrac{ax+b}{cx+d}. \end{aligned}$$

Furthermore function composition respects matrix multiplication, in other words $\varphi_{m_1 m_2} = \varphi_{m_1} \circ \varphi_{m_2}$.

Consider the elements of the function field of $\mathbb{P}^1$ – these are rational morphisms H from $\mathbb{P}^1$ to $\mathbb{P}^1$ defined as

$$\begin{aligned} H : \mathbb{P}^1 &\longrightarrow \mathbb{P}^1 \\ [x:y] &\longmapsto [F(x,y) : G(x,y)] \end{aligned}$$

where F and G are two homogeneous polynomials in $\mathbb{F}[x,y]$ of the same degree.

Given that Möbius transforms are automorphisms, they give rise to automorphisms of function fields. In this case there is a right (contravariant) action of $\mathcal{M}$ to rational maps $H = [F(x,y) : G(x,y)]$ given by $\varphi_m H = H \circ \varphi_m = [F(\varphi_m[x:y]) : G(\varphi_m[x:y])]$ and therefore $\varphi_m(\varphi_n H) = (\varphi_n \varphi_m)H = \varphi_{nm} H$.

Going back to higher degree Legendre PRF, one may think of the keyspace of polynomials of degree r as rational morphisms of $\mathbb{P}^1$ where

$$\begin{aligned} f : \quad \mathbb{P}^1 &\longrightarrow \mathbb{P}^1 \\ [x:y] &\longmapsto [F(x,y) : y^r] \end{aligned}$$

where $F(x,y) = f(x/y)y^r$ is the homogenisation of f. Möbius transforms act on this space by sending polynomials f to $\tilde{f}_m := f \circ \varphi_m$. In the affine plane $y = 1$ we have $\tilde{f}_m : x \mapsto f(\frac{ax+b}{cx+d})$. Therefore, given an oracle that on input $x \in \mathbb{F}_p$ gives $\mathcal{O}(x) = \left(\frac{f(x)}{p}\right)$, we can obtain an oracle associated to $\tilde{f}_m$ that outputs $\tilde{\mathcal{O}}(x) = \left(\frac{\tilde{f}_m(x)}{p}\right) = \left(\frac{f((ax+b)/(cx+d))}{p}\right)$. However $\tilde{f}_m$ is not a monic polynomial. As such it is not in our primary interest as it is not an element of the keyspace. We can easily fix this by analysing $\tilde{f}_m$ further:

$$\tilde{f}_m(x) = f\left(\frac{ax+b}{cx+d}\right) = \frac{\hat{f}_m(x)}{(cx+d)r}$$

where $\hat{f}_m(x)$ is a polynomial of degree r. However $\hat{f}_m(x)$ is not in the keyspace again as in general it is not monic. The leading coefficient of $\hat{f}_m$ is $f(a/c)c^r$ which can be obtained from one query to the oracle $\mathcal{O}(a/c)$ and one Legendre symbol computation $\left(\frac{c}{p}\right)$. Therefore

$$f_m(x) := f\left(\frac{ax+b}{cx+d}\right)\frac{(cx+d)^r}{f(\frac{a}{c})c^r}$$

is a monic polynomial, and we can compute Legendre symbols of $f_m(x)$ because

$$\left(\frac{f_m(x)}{p}\right) = \mathcal{O}\left(\frac{ax+b}{cx+d}\right)\left(\frac{cx+d}{p}\right)^r \mathcal{O}\left(\frac{a}{c}\right)\left(\frac{c}{p}\right)^r.$$

We can obtain $\{f_m\}_L$ by computing $L+1$ Legendre symbols and querying the oracle $L+1$ times. Both the Legendre symbols and oracle queries may be precomputed and saved in two tables of size p each, from which we just extract the symbols that we need. The cases of f_m such that $cx+d=0$ for some $x = 0, \ldots, L-1$ may be ignored as their number is negligible with respect to the total number of m's that we consider. The case $c=0$ is not problematic as in that case $f(\frac{a}{c})c^r = a^r \neq 0$.

Note further that f_m satisfies the following property - if f factors as $f(x) = \prod_{i=1}^{r}(x-\alpha_i)$ then

$$f_m(x) = \prod_{i=1}^{r}(x - m^{-1}\alpha_i) = \prod_{i=1}^{r}(x - \frac{d\alpha_i - b}{-c\alpha_i + a}) \tag{2}$$

where m^{-1} is the inverse of the Möbius transform m given by $m^{-1} = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)^{-1} \equiv \left(\begin{smallmatrix} d & -b \\ -c & a \end{smallmatrix}\right)$. Therefore the group $\mathcal{M}$ of Möbius transformations has left (covariant) action on the roots of polynomials in $\mathbb{F}_p[x]$.

The algorithm for finding $f(x)$ now works as follows. We precompute $\{f_m\}_L$ for all f_m in the orbit $\mathcal{M}\cdot f$. Then we try random monic polynomials g of degree r until $\{g\}_L$ is in the precomputed set. In that case we have $g = f_m$ for some $m \in \mathcal{M}$, and so we can find $f = g_{m^{-1}}$. The expected number of trials depends on the size of the orbit $\mathcal{M}\cdot f$ which we can bound by the following lemma.

Lemma 1. *Let $\mathcal{M} = \mathrm{PGL}_2(\mathbb{F}_p)$ and $f \in \mathbb{F}_p[x]$ an irreducible polynomial of degree $r \geq 3$. Then the size of the orbit of f is $\frac{p^3-p}{r'}$ for some $r' \mid r$.*

Proof. By the orbit-stabiliser theorem, the size of the orbit is equal to $\frac{\#\mathcal{M}}{\#\mathcal{M}_f}$ where $\mathcal{M}_f = \{m \in \mathcal{M} | f = f_m\}$ is the stabiliser of f. Therefore it is enough to find the stabiliser of f. Let $m \in \mathcal{M}_f$. By property (2) the roots of f_m are $m^{-1}\alpha_i$ meaning that m permutes the roots of f. Let $\mathrm{Gal}(f) = \mathbb{Z}/r = \{\phi_i := x \mapsto x^{p^i} | i \in \mathbb{Z}/r\}$ be the Galois group of f, and let α be any root of f. Then $m\alpha = \phi_i(\alpha)$ for some $i \in \mathbb{Z}/r$. Furthermore $m(\phi_j(\alpha)) = \phi_j(m\alpha) = \phi_j(\phi_i(\alpha)) = \phi_i(\phi_j(\alpha))$ since m is rational and it commutes with the Frobenius. Therefore each element of the stabiliser acts on the roots as an element of $\mathrm{Gal}(f)$. This gives rise to a homomorphism from $\mathcal{M}_f$ to $\mathrm{Gal}(f)$. Furthermore this homomorphism is injective since two Möbius transformations with the same action on a set of $r \geq 3$ points have to be equal. This means that $r' := \#\mathcal{M}_f \mid \#\,\mathrm{Gal}(f)$, and therefore the orbit size is

$$\frac{\#\mathcal{M}}{\#\mathcal{M}_f} = \frac{p^3-p}{r'}$$

for some $r' \mid r = \#\,\mathrm{Gal}(f)$. In particular the order of the orbit is at least $\frac{p^3-p}{r}$. □

It follows that the number of expected trials until we find f is

$$O\left(\frac{p^r}{(p^3-p)/r}\right) = O(p^{r-3}r).$$

For each trial we need to compute L Legendre symbols. Unlike in the linear case, here it does not help us to compute additional "free" polynomials from a random g, i.e., the g_m's. They are highly correlated, and in fact g will give us a solution if and only if g_m gives one. Therefore it is necessary to compute L Legendre symbols at every trial, giving rise to a

$$O(p^{\max\{r-3,r/2\}}r^2 \log p)$$

algorithm. Since the precomputed table is of size $O(p^3)$ and we do $O(p^{r-3})$ trials, we cannot do better than $O(p^{r/2})$ which is the reason for the $\max\{r-3, r/2\}$ in the exponent of p.

If the secret polynomial is reducible one can do better. Let $f(x) = l(x)h(x)$ be a factorisation of $f(x)$ with r_l and r_h the degrees of $l(x)$ and $h(x)$ respectively and r_h the lowest number such that $r_l \leq r_h$. Note that the action of $\mathcal{M}$ is multiplicative on polynomials, i.e. $f_m(x) = l_m(x)h_m(x)$. Then we can achieve an $O(p^{\max\{r_h-3,r/2\}}rL)$ run time in the following manner. First we precompute at random $O(p^{r-3}r)$ polynomials of degree r_h and make a table containing the length L Legendre sequence for each of them. This set will contain $\{h_n(x)\}_L$ for some $n \in \mathcal{M}$ with constant probability. Then we compute all the $O(p^{3+r_l})$ sequences of type $\{f_m(x)g(x)\}_L$ where $m \in \mathcal{M}$ and $g(x) \in \mathbb{F}_p[x]$ of degree r_l, and look up each of them in the precomputed table. Eventually we will find $\{f_m(x)g(x)\}_L = \{h_n\}_L$ meaning that $f(x) = g_{m^{-1}}(x)h_{nm^{-1}}(x)$. With a bit more care $O(p^{r/2}rL)$ is achievable with constant probability even if $3 + r_l > r/2$. Note that we can go through all the reducible polynomials in time

$$O\left(\sum_{r_h=r/2}^{r-1} p^{\max\{r_h-3,r/2\}}rL\right) = O(p^{\max\{r-4,r/2\}}rL)$$

implying that reducible polynomials provide less security than irreducible ones.

7.7 Remarks

We give a couple of remarks regarding the structure of the Möbius group and its use the the linear Legendre PRF case.

The Möbius group in the linear case In the higher degree Legendre PRF we make full use of the group of Möbius transformations $\mathcal{M} \cong \mathrm{PGL}_2(\mathbb{F}_p)$ by obtaining from one polynomial $f(x)$ the full orbit of f that is

$$\mathcal{M} \cdot f = f\left(\frac{ax+b}{cx+d}\right)$$

which is of size $\Omega(\frac{p^3-p}{r})$. Doing this however requires querying f or the associated oracle on the full field $\mathbb{F}_p$. Since the Möbius group allows us to obtain $\sim p^3$ Legendre sequences from p queries in the high degree case, one would hope to be able to obtain $\sim M^3$ sequences from M queries in the linear case which, in theory, would reduce the number of queries necessary to reach a $\sqrt{p}$ attack from $\sqrt[4]{p}$ down to $\sqrt[6]{p}$. However, in the linear case we have to be more careful with respect to which queries we do as we are allowed to do at most $\sqrt[4]{p}$ of them, otherwise we may as well use the algorithm given above.

In our algorithm we query $\mathcal{O}(x) = \left(\frac{x+k}{p}\right)$ at points $i + xd$ such that $0 \leq i + xd \leq M$ for $i = 0, 1, \ldots, L-1$. A natural way to extend this to the full Möbius group is to query

$$\mathcal{O}\left(\frac{ax+b}{cx+d}\right) \text{ with } \begin{cases} 0 \leq ax+b \leq M \\ 0 < cx+d \leq N \end{cases} \quad \text{for } x = 0, 1, \ldots, L-1.$$

For each $m = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ we can obtain $\{\frac{dk+b}{ck+a}\}_L$ from the above queries. Since we have M^2/L choices for (a, b) and N^2/L choices for (c, d) this gives rise to a total of $\frac{M^2N^2}{L^2}$ matrices, and thus Legendre sequences. However matrices are defined up to multiplication by a nonzero scalar. This reduces the number of sequences, but arguably their number stays $O(\frac{M^2N^2}{L^2})$ for small enough values of M, N.

The number of queries that we need to do in order to obtain these Legendre sequences depends on the number of different values of $\frac{ax+b}{cx+d}$ for the above constraints on a, b, c, d. We essentially have numbers of type $\frac{\tilde{m}}{\tilde{n}}$ with $0 \leq \tilde{m} \leq M$ and $0 \leq \tilde{n} \leq N$. One can write this as $\frac{m'}{n} + m''$ with $m' \in \{0, 1, \ldots, n-1\}$ and $m'' \in \{0, 1, \ldots, \frac{M}{n}\}$. Therefore we need query

$$\mathcal{O}\left(\frac{m'}{n} + m''\right) \text{ for } m' \in \{0, 1, \ldots, n-1\} \text{ and } m'' \in \{0, 1, \ldots, \frac{M}{n}\} .$$

Due to possible common factors in m' and n some of these points repeat. We only need to count the m' and n such that $\gcd(m', n) = 1$. This gives rise to $\frac{M}{n}$ queries for each m', n, the total being

$$\sum_{n=1}^{N} \varphi(n)\frac{M}{n} = M\sum_{n=1}^{N} \frac{\varphi(n)}{n} = \Theta(MN).$$

Therefore we can obtain $\frac{M^2N^2}{L^2}$ different Legendre sequences from MN queries, which is worse than $\frac{M^2N^2}{L}$ which can be obtained by the original algorithm that considers only linear transformations. Choosing different bounds for a, b, c, d might allow better sequence extraction, however we did not pursue the search for better bounds further as it seems highly improbable that they exist.

The Möbius group structure It should be noted that the algorithms for the linear and higher degree Legendre PRF do not fully exploit the group properties of $\mathcal{M} = \mathrm{PGL}_2(\mathbb{F}_p)$. We use the fact that inverses exist when we find a hit in the table $f_m(x) = g(x)$ and we use that to compute the polynomial $f(x) = g_{m^{-1}}(x)$. However that is the only place where we use group operations. It would be interesting to see if composing group actions on polynomials can lead to improvements elsewhere. It is known that the generators of $\mathrm{PGL}_2(\mathbb{F}_p)$ are

$$\begin{pmatrix} g & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

where g is a generator of $\mathbb{F}_p^*$. We can write every matrix as a product of the generators easily by following the LU decomposition. The generating set can be reduced to just

$$\begin{pmatrix} g & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} -1 & 1 \\ -1 & 0 \end{pmatrix}.$$

For the linear case we do not use the full Möbius group, but a subgroup. This subgroup can be characterised in the following way

$$G = \mathbb{Z}/p \rtimes (\mathbb{Z}/p)^* \text{ with } \begin{cases} \text{identity } (0, 1), \\ \text{group operation } (i, d)(j, b) = (i + dj, db), \\ \text{inverse } (i, d)^{-1} = (-\frac{i}{d}, d^{-1}). \end{cases}$$

The group has right (contravariant) action on polynomials of degree r by $(i, d) \cdot f(x) = \frac{f(i+dx)}{d^r}$ or left (covariant) action on the roots by sending α to $\frac{\alpha - i}{d}$. In the linear Legendre PRF case this corresponds to sending k to $\frac{k+i}{d}$. The group elements can be represented in matrix form as $\left(\begin{smallmatrix} d & i \\ 0 & 1 \end{smallmatrix}\right) \in \mathrm{PGL}_2(\mathbb{F}_p)$. Another important property is that the group has two generators, $\left(\begin{smallmatrix} g & 0 \\ 0 & 1 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ where g is a generator of $\mathbb{F}_p^*$. In fact

$$(i, d) = \begin{pmatrix} d & i \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} g & 0 \\ 0 & 1 \end{pmatrix}^{\log_g d} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{i/d}.$$

However these properties are not exploited by the algorithm.

8 Conclusion

In this paper we give an improved algorithm for solving the Shifted Legendre Symbol Problem. Our algorithm is an improvement with respect to the previous best known algorithm [10] and the recently published [2] in four ways. Firstly our algorithm solves SLSP in $O(\sqrt{p \log \log p})$ operations on a $\log p$-bit machine, while Khovratovich requires $O(\sqrt{p} \log p)$ Legendre symbol computations, which are more costly than simple bit-operations such as sequence comparisons and bit manipulations. Secondly we achieve the optimal run time with only $M = \sqrt[4]{p}\sqrt{\log p}\sqrt[4]{\log \log p}$ queries to the oracle, while [10] requires $M = \sqrt{p} \log p$ queries. Thirdly, in our algorithm queries are sequential, while [10] needs access to short sequences in random positions. Lastly, if the number of queries is limited to M, we achieve a run time of $O(\frac{p \log p \log \log p}{M^2})$ while the algorithm in [10] achieves $O(\frac{p \log^2 p}{M})$. For the comparison with respect to the algorithm in [2] the reader can refer to Table 4.

On the practical side we explain our implementation, which varies slightly from the theoretical algorithm due to the fact that the bottleneck in practice is the cost of memory access. We give the solutions of the 64, 74 and 84-bit challenges posed by the Ethereum foundation.

We showed that our algorithm extends naturally to all generalisations of the Legendre symbol because they all satisfy the necessary algebraic properties such as the shifting and multiplicative property. We furthermore show that there is an improved quantum algorithm that does not require access to a quantum oracle – this is a natural extension as the bottleneck of our algorithm is the number of L-bit word operations done in a birthday attack, and which can be done more efficiently with a quantum *claw* algorithm. In the higher degree Legendre PRF we explain how the attack generalises and how the Legendre sequences and the oracle can be acted on by the group of Möbius transformations. This gives an improved attack in the higher degree PRF, which in some cases can reach the birthday bound. We also show that reducible polynomials are less secure than irreducible ones. In the linear case we cannot exploit the full Möbius group, but we only use the subgroup consisting of linear transformations.

Table 4. Comparisons of best known algorithms for solving the Legendre PRF challenge. We denote with t the time to compute a Legendre symbol.

Algorithm	Optimal run time	Queries required	Memory	run time with $M \leq$optimal	Memory
Khovratovich	$O(\sqrt{p}t \log p)$	$O(\sqrt{p} \log p)$	$O(1)$	$O(\frac{pt \log^2 p}{M})$	$O(M \log p)$
Beullens et al.	$O(\sqrt{p} \log p)$	$O(\sqrt[4]{p \log^2(p)})$	$O(\sqrt{p} \log p)$	$O(\frac{p \log^2 p}{M^2})$	$O(M^2)$
Our algorithm	$O(\sqrt{p \log \log p})$	$O(\sqrt[4]{p \log^2(p) \log \log(p)})$	$O(\sqrt{p \log \log p} \log p)$	$O(\frac{p \log p \log \log p}{M^2})$	$O(M^2)$

Table 5. Comparisons of best known algorithms for solving the degree $r \geq 2$ Legendre PRF challenge in the irreducible and composite case with a factor of degree $r_h \geq r/2$. Complexity is given in number of Legendre symbols computed. In all cases we need p queries and the run time cannot go below $O(p^{r/2} r \log p)$.

Algorithm	Irreducible run time	Memory	Composite run time	Memory
Khovratovich	$O(p^{r-1} r \log p)$	$O(r \log p)$	$O(p^{r-1} r \log p)$	$O(r \log p)$
Beullens et al.	$O(p^{r-2} r^2 \log^2 p)$	$O(p^2)$	$O(p^{r_h} r \log p)$	$O(p^{r-r_h} r \log p)$
Our algorithm	$O(p^{r-3} r^2 \log p)$	$O(p^3 r \log p)$	$O(p^{r_h-3} r^2 \log p)$	$O(p^{r-r_h} r \log p)$

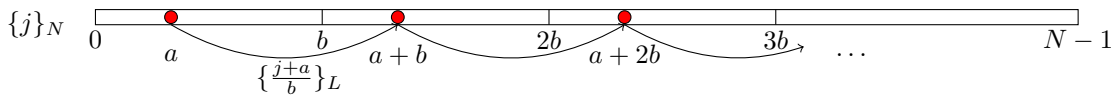
References

1. Bach, E.: Realistic analysis of some randomized algorithms. J. Comput. Syst. Sci. **42**(1), 30–53 (Jan 1991). https://doi.org/10.1016/0022-0000(91)90038-7, http://dx.doi.org/10.1016/0022-0000(91)90038-7
2. Beullens, W., Beyne, T., Udovenko, A., Vitto, G.: Cryptanalysis of the legendre prf and generalizations. Cryptology ePrint Archive, Report 2019/1357 (2019), https://eprint.iacr.org/2019/1357
3. van Dam, W., Hallgren, S.: Efficient quantum algorithms for shifted quadratic character problems (2000)
4. van Dam, W., Hallgren, S., Ip, L.: Quantum algorithms for some hidden shift problems (2002)
5. Damgård, I.: On the randomness of Legendre and Jacobi sequences. In: Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology. pp. 163–172. CRYPTO '88, Springer-Verlag, London, UK, UK (1990), http://dl.acm.org/citation.cfm?id=646753.704912
6. Davenport, H.: On the distribution of quadratic residues (mod p). Journal of the London Mathematical Society **s1-8**(1), 46–52 (1933). https://doi.org/10.1112/jlms/s1-8.1.46, https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/jlms/s1-8.1.46
7. Feist, D.: Legendre pseudo-random function (2019), https://legendreprf.org/
8. Grassi, L., Rechberger, C., Rotaru, D., Scholl, P., Smart, N.P.: MPC-friendly symmetric key primitives. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 430–443. CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978332, http://doi.acm.org/10.1145/2976749.2978332
9. Ip, L.: Solving shift problems and hidden coset problem using the Fourier transform (2002)
10. Khovratovich, D.: Key recovery attacks on the Legendre PRFs within the birthday bound. Cryptology ePrint Archive, Report 2019/862 (2019), https://eprint.iacr.org/2019/862
11. Kranakis, E.: Primality and Cryptography. John Wiley & Sons, Inc., New York, NY, USA (1986)
12. Russell, A., Shparlinski, I.: Classical and quantum function reconstruction via character evaluation. Journal of Complexity **20**(2-3), 404–422 (4 2004). https://doi.org/10.1016/j.jco.2003.08.019
13. Tani, S.: Claw finding algorithms using quantum walk. Theoretical Computer Science **410**(50), 5285–5297 (Nov 2009). https://doi.org/10.1016/j.tcs.2009.08.030, http://dx.doi.org/10.1016/j.tcs.2009.08.030
14. Weil, A.: On some exponential sums. Proceedings of the National Academy of Sciences **34**(5), 204–207 (1948). https://doi.org/10.1073/pnas.34.5.204, https://www.pnas.org/content/34/5/204

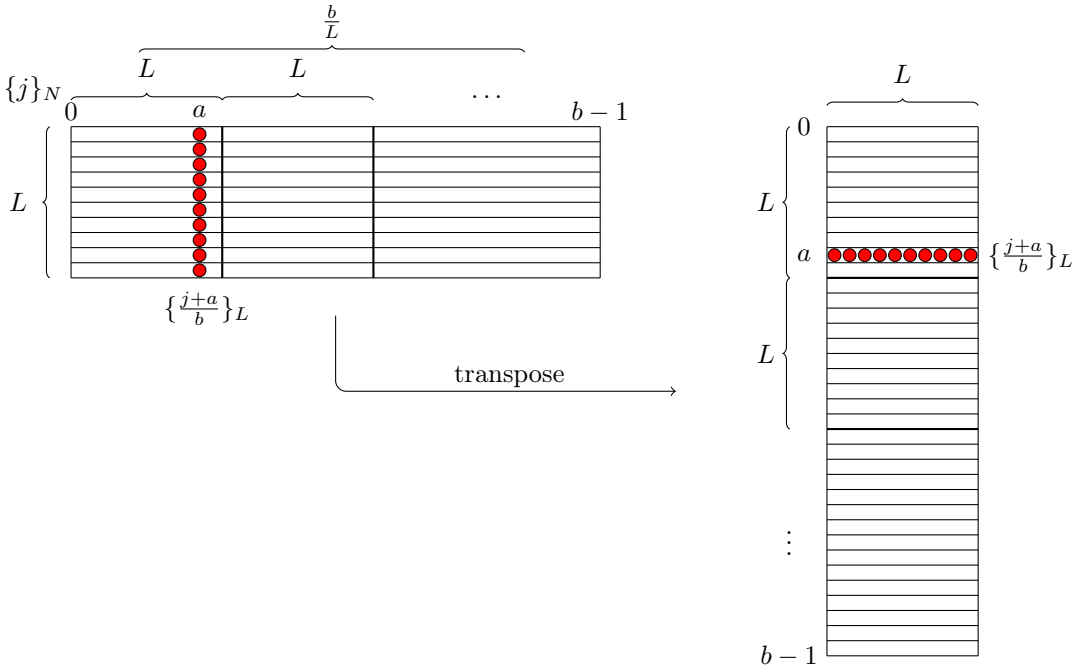
novak.kaluderovic@epfl.ch
thorsten.kleinjung@epfl.ch
dusan.kostic@epfl.ch

A Extracting sequences in the search stage

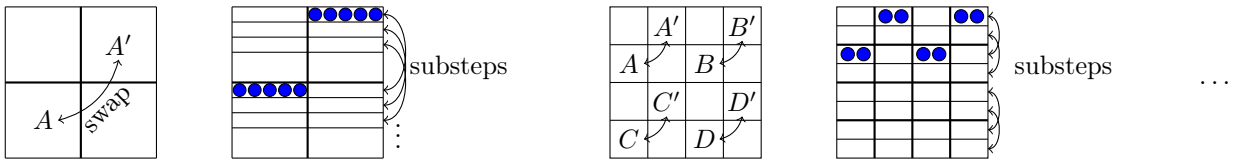
We show how to extract sequences in logarithmic amortised time. The goal is to obtain $\{\frac{j+a}{b}\}_L$ from $\{j\}_N$ for a, b as in Section 3.3. The straightforward manner is to extract the elements of $\{j\}_N$ at indices $a, a+b, a+2b, \ldots, a+(L-1)b$, and multiply everything by $\left(\frac{b}{p}\right)$. This would take a total of $O(L)$ operations on an L-bit word machine. However one can lower this number to $O(\log L)$ by extracting elements in batches. We show how to do this for any denominator b of $\{\frac{j+a}{b}\}_L$. Given a fixed b we need to extract $\{\frac{j+a}{b}\}_L$ for $a = 0, 1, \ldots, b-1$. The required Legendre symbols are represented as red dots:



If we look at $\{j\}_N$ from a different angle by dividing it into chunks of size b and placing them into rows of a matrix, then the sequences $\{\frac{j+a}{b}\}_L$ are exactly the columns of the obtained matrix, up to multiplying everything by the symbol of b. This matrix is written into memory as L consecutive chunks of $\frac{b}{L}$ words of length L. By computing the transpose of this matrix we obtain b consecutive words of length L, each one corresponding to a sequence $\{\frac{j+a}{b}\}_L$.



The transposition is done by transposing $\frac{b}{L}$ submatrices of size $L \times L$. It is known that an $L \times L$ transpose can be done in $O(L \log L)$ operations on an L-bit word machine. This is done in $\log_2 L$ steps, where in step $i = 0, \ldots, \log_2(L) - 1$ we swap $2 \cdot 4^i$ submatrices. The swaps are done in $L/2$ substeps. A substep is a swap of two substrings between two machine words of length L. As a function of two L-bit words, it depends only on the index of step i. All $\log_2 L$ of these functions can be precomputed once and then executed at a cost of $O(1)$ operations on L-bit words per substep. One substep is represented with a swap of consecutive blue dots in the following diagram:



Step 0 Step 1

In total there are b/L transposes of $L \times L$ matrices, each of them in $O(L \log L)$ operations. That is $O(b \log L)$ operations to extract b Legendre sequences, therefore $O(\log L)$ operations per sequence.

In the case of a w-bit word machine, we can transpose w/L matrices in parallel in $O(L \log L)$ operations – just think of their rows being parts of the same word and apply the same operations in parallel. This lowers the complexity to $\frac{L \log L}{w}$ operations per sequence. However at least one additional operation is needed to isolate a sequence of L bits from an w-bit word, therefore the complexity in this case is $O(\max\{\frac{L \log L}{w}, 1\})$ operations per sequence.