Homomorphic decryption in blockchains via compressed discrete-log lookup tables

Panagiotis Chatzigiannis[1], Konstantinos Chalkias[2], and Valeria Nikolaenko[2]

[1] Novi / Facebook / George Mason University, USA pchatzig@gmu.edu
[2] Novi / Facebook, USA {kostascrypto, valerini}@fb.com

Abstract. Many privacy preserving blockchain and e-voting systems are based on the modified ElGamal scheme that supports homomorphic addition of encrypted values. For practicality reasons though, decryption requires the use of precomputed discrete-log (*dlog*) lookup tables along with algorithms like Shanks's *baby-step giant-step* and Pollard's *kangaroo*. We extend the Shanks approach as it is the most commonly used method in practice due to its determinism and simplicity, by proposing a truncated lookup table strategy to speed up decryption and reduce memory requirements. While there is significant overhead at the precomputation phase, these costs can be parallelized and only paid once and for all. As a starting point, we evaluated our solution against the widely-used *secp* family of elliptic curves and show that we can achieve storage reduction by 7x-14x, depending on the group size. Our algorithm can be immediately imported to existing works, especially when the range of encrypted values is known, such as in Zether, PGC and Solidus protocols.

Keywords: discrete log, ElGamal, homomorphic encryption, precomputation

1 Introduction

Over the last few years, we have witnessed an increasing number of decentralized payment systems that use the additively homomorphic ElGamal scheme to offer non-interactive confidential amounts [2,6,8,10,11]. However, for efficiency purposes, this ElGamal variant requires a precomputed discrete-log lookup table. The latter is because the ciphertext carries the message in the exponent of the group's base point, and without caching, it's only decryptable via bruteforcing.

Typically, for most financial applications, the max transaction amount ranges from 2^{32} to 2^{64}, as this is usually enough to encode even the largest reasonable balance. Generating and loading a table for billions and trillions of elements is impractical though, especially for constrained devices such as mobile phones, IoT devices and light clients in general. For instance, when using the 256-bit *secp256k1* elliptic curve where a group point is serialized in 33 bytes (in compressed form), 132 GB are required to store 2^{32} elements in binary format.

While solutions exist that trade storage space for less computations, they increase the overall storage costs by several orders of magnitude or they are not deterministic. In this paper we propose an extra layer of compression that

is compatible with Shanks algorithm (also known as *baby-step giant-step*), that reduces the lookup table size by a constant factor *without* any additional decryption cost. Our methods only require a precomputation phase and produce collision-free tables[3] with size independent of the elliptic curve field size; the larger the size, the better the compression rate. We also leverage the fact that some blockchain systems' encrypted content (e.g., Zether [6]) is accompanied by range proofs, and we use this information to slightly speed up decryption.

In the implementation section, we present a few strategies for precomputations and actually provide complete compressed lookup tables for NIST's secp256r1 curve as a starting point. Protocol designers can follow our recommendations to also create tables for curves of their choice, which will eventually result in a template repository for the most popular curves that will help the adoption and efficiency of additively homomorphic encryption schemes in general.

2 Background

Here we provide background on small *dlog* lookups. We first describe how modified ElGamal encryption works as a common use-case, then review related works in improving lookup table efficiency for recent blockchain-based systems requiring such tables, which can benefit from our work.

2.1 Additively homomorphic ElGamal encryption

Modified ElGamal encryption [19] consists of the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$: On input of security parameter λ, outputs public parameters $\mathsf{pp} = (\mathbb{G}, g, p)$ where g is generator of cyclic group $\mathbb{G}$ of prime order p. We consider these parameters as a default input to all following algorithms and we omit them for simplicity.
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}()$: Outputs a secret-public key pair as $\mathsf{sk} \leftarrow \mathbb{Z}_p$, $\mathsf{pk} = g^{\mathsf{sk}}$.
- $(c_1, c_2) \leftarrow \mathsf{Enc}(\mathsf{pk}, x)$: Samples $r \leftarrow \mathbb{Z}_p$, computes $c_1 = g^r, c_2 = g^x \cdot \mathsf{pk}^r$ and outputs ciphertext $C = (c_1, c_2)$.
- $x \leftarrow \mathsf{Dec}(\mathsf{sk}, (c_1, c_2))$: Compute $g^x = c_2 / c_1^{\mathsf{sk}}$. While x cannot be directly computed from g^x, it can be recovered through a pre-computed lookup table, assuming that the message space is relatively small, i.e., up to 2^{32}.

The scheme is additively homomorphic because it holds that $\mathsf{Enc}_A(\mathsf{pk}, x_1) \cdot \mathsf{Enc}_B(\mathsf{pk}, x_2) = (c_{1_A} \cdot c_{1_B}\ ,\ c_{2_A} \cdot c_{2_B}) = \mathsf{Enc}(\mathsf{pk}, x_1 + x_2)$.

Also, note that the message space is in $\mathbb{Z}_p$, and thus 0 is not included. There are various methods to support encryption of 0, for instance by mapping the message space $\{0, ..., p-2\}$ to $\{1, ..., p-1\}$ at the application layer.

[3] Similar compression techniques are also discussed in [3], but our work focuses on collision-free tables per group to completely avoid false positives.

2.2 Improving efficiency of discrete log lookup operations

A naive approach for lookup tables would require precomputing all (x, g^x) tuples up to a maximum value and storing them in a file, where $x \in [1, N]$; N is usually a power of 2, thus $N = 2^n$. Typically, generating this file requires resource-intensive bruteforcing and $O(N)$ storage, but the lookup cost is amortized to $O(1)$.

Shanks algorithm [18] enables a space-time tradeoff for the naive solution. It defines a baby-step with stores (x, g^x) for all $x \in [1, 2^\alpha]$ in a lookup table M, and up to 2^β giant-steps, with $\alpha + \beta = n$. To recover the discrete log of $c = g^x$, it computes the giant step as g^{2^α}, initializes counter $i \to 0$, and checks if c/g^{i2^α} exists in M. If the lookup is successful: return $x + i \cdot 2^\alpha$, else increment i and repeat lookup. This algorithm has $O(2^\alpha)$ storage and requires 2^β multiplications (elliptic curve point additions) in the worst case, which can be amortized per application. There are also followup works that improve on the average expected computation costs by a constant factor [4,17].

Existing practical approaches for finding *dlog* in a small interval build on top of Pollard's rho and kangaroo methods [1,3,12,13,16], mainly focusing on improving the average-case complexity. In contrast, we mainly focus on Shanks method, because its practical average-case complexity is within a constant factor of its worst-case complexity, thus giving an algorithm that will perform well in practice for all instances (even for adversarially chosen ones).

2.3 Related blockchain works that require precomputed tables

Solidus [8] is a permissioned, distributed payment system that associates user accounts with Banks, which in turn maintain a private data structure containing their user account public keys and the respective balances. Solidus utilizes modified ElGamal encryption to achieve its needed additive homomorphism, with balances represented in the exponent. The authors apply Shanks algorithm in their implementation to represent up to 2^{32} values.

Zether [6] is a distributed confidential payment system implemented on top of an Ethereum smart contract by converting Ethereum coins to native Zether tokens. These tokens are represented in encrypted form within the contract internal state using the additively homomorphic version of ElGamal encryption scheme, and are accompanied by range proofs. Original Zether's implementation considers a range of 2^{32}, but supporting 64 bits is possible by splitting into 2 amounts of 32 bits each (most and least significant bits, respectively); then encrypt each one of them separately. Interestingly, the authors refer to decryption via bruteforcing without references to optimized solutions or precomputed tables. As they claim, this is not considered an issue because a) decryption is happening off-chain b) bruteforcing would occur only rarely, i.e., when the amount is unknown.

PGC [10] is another distributed, confidential payment system, which uses a customized ElGamal encryption variant with an extra generator h in its public parameters that enables proving relations between ciphertexts encrypted under

different public keys. PGC is one of the few works that recognizes the problem of efficient small *dlog* lookup tables, and while it highlights the greater efficiency of heuristic approaches like *kangaroo*, it still opts for Shanks to enable easy amortization for the time-space tradeoff and parallelization. In their proof of concept implementation [2], the authors assume up to 2^{32} values and utilize a 264MB lookup table by precomputing 2^{23} 33-byte secp256k1 elements; then requiring at most $2^9 = 512$ giant Shanks steps during decryption.

Quisquis cryptocurrency [11] has a unique account balance representation, combining public keys and additively homomorphic commitment scheme tuples with balances represented in an exponent within the commitment. The system considers ranges up to 2^{32} without specifying any time-space tradeoff algorithm.

Although blockchains is our primary focus, other areas that use homomorphic encryption would also benefit from our construction, e.g. e-voting schemes [15].

3 Methodology

Our goal is to compute a lookup table for $g^x \ \forall x \in [1, 2^n]$. Assuming an elliptic curve (EC) over a finite field $\mathbb{F}_q$ for a security parameter λ, the uncompressed serialized representation of an EC point is $2\log q$ bits. Typically however, a compressed format of $q' = \log q + 1$ bits is selected, where only one coordinate and additional sign bit are enough to reconstruct the EC point. We define our initial table construction algorithm ComputeTable() with output to file f: Pick some $\tau < q'$, append $g^x = b_1 b_2 ... b_\tau$ to f where $b_i \in (0, 1)$.

Then our second important optimization relies on picking a "truncation" parameter τ. As it involves the birthday paradox, τ needs to be carefully considered. Picking a τ too small (meaning that the "chopped" portion of g^x is large), many collisions among the whole table will occur, which might result in ambiguities during lookup operations (e.g. on ElGamal decryption).

Therefore at this stage we elect a conservative approach similarly to [14]. Note that an approximation of collision probability[4] via Taylor series expansion is $\Pr[n, \tau] \approx 1 - 1/e^{2^{(2n-\tau-1)}}$. One can safely pick $\tau = 80$ for a table with 2^{32} elements, because $\Pr[\exists \text{ collision}] \approx 1/132,000$. ComputeTable() provides an immediate benefit for memory and computation requirements, bringing down storage from $q'2^n$ to $\tau 2^n$.

3.1 Ideal truncation

The first precomputation phase ComputeTable() was a warm-up and required to minimize memory and storage requirements; essentially all our methods for compressing lookup tables rely on truncating (or omitting) redundant information from small *dlog* lookup tables. We describe our generic approach as follows:

Given the precomputed table in f (which is already less than half the size of the naively-precomputed full table), there exists an *ideal* encoding where each

[4] While we assume the binary representation of g^x is random, we can employ a hash function if g has some special property.

value can be *uniquely* represented with n bits of information, thus bringing the total size of the table to $n2^n$ bits. We denote this ideal encoding as $F_{\mathsf{ideal}}(g^x) \rightarrow \tilde{g}^x$ where $|\tilde{g}^x| = n$. Note that this ideal encoding does not depend on q'. For decrypting g^x, we would need to sequentially parse the table until its encoding $\tilde{g}^x$ is found, and return its position in the file as x. Essentially, this ideal encoding F_{ideal} is a compression technique that provides great savings in storage; in particular, the end compression ratio is $\frac{q'}{n}$ and the space cost savings are significant in typical implementation scenarios where $n << q'$. Note that after the compressed table has been computed, there are no additional computation costs during decryption, as the lookup costs remain amortized $O(1)$.

The above technique is compatible with existing algorithms with time-space tradeoffs, as discussed in Section 2.2. For instance, one could implement Shanks algorithm with an even smaller baby-step lookup table. In this case however, uniqueness must still be ensured not just for the values in the baby-step lookup table, but for all possible values generated by the giant steps, else any ambiguity would still make a ciphertext decryption possible into different values. Therefore, the baby-step lookup table would be $n2^\alpha$ bits in size using the ideal encoding F_{ideal}. As an example, Zether with the NIST-P256 curve, $2^{32} - 1$ max value and Shanks parameters $\alpha = 24, \beta = 8$, would normally have a lookup table of size $264 \cdot 2^{24}$ bits or about 528MB. Using an ideal encoding F_{ideal}, its size would only be $32 \cdot 2^{24}$ bits or about 64MB (decreased by a factor of 8.25), which is the best possible compression that can be achieved while ensuring unique representation of all elements.

3.2 Variable-length truncation

In the previous paragraph we described an ideal encoding function F_{ideal} which maps values g^x of length q' to short, unique bit strings a of length n. However, assuming all g^x values have a uniform distribution, constructing such an ideal function that ensures uniqueness among all strings is challenging and potentially impractical to find. Therefore to approximate this ideal encoding as much as possible, we define VarTruncate(), shown in Algorithm 1.

In a nutshell, this algorithm works as follows. Similarly to Shanks scheme, we pick the baby-step giant-step parameters α and β respectively. We also choose truncation parameters τ_{start} and τ_{stop} (where $\tau_{start} > \tau_{stop}$), which respectively direct the algorithm on how many bits should start the binary representation for each element with, and how many bits should try to represent the elements in an unambiguous way. The algorithm also needs as input data a set of uncompressed precomputed tables $\mathrm{A}_1, ..., \mathrm{A}_\mathrm{p}$, which are partitioned to lower RAM requirements. Then the algorithm, after initializing a variable truncation index table C, it starts from the "conservative" truncation parameter τ_{start} and checks uniqueness of truncated elements for a baby-step table A2 against all truncated elements of a (precomputed) full table A1. If a collision is found, we update the respective index in C with the previous collision-free truncation parameter τ. This process is repeated by decrementing that parameter (i.e. truncating more

Input : Generator g, Shanks parameters α, β, truncation start-stop τ_{start},τ_{stop}, p = # of partitions
Data : Precomputed tables for $[1, 2^{\alpha+\beta}]$: $\mathrm{A}_1, ..., \mathrm{A}_\mathrm{p}$
Output : List C that stores where collisions found per index

```
B ← [ ] , C ← [0]^(2^α);
for i = 1, i = 2^α, i++ do
    B.append(g^i)
end
\\ Check for collisions between elements of B and those from 2^α + 1 to g^(2^(α+β))
for τ = τ_start; τ ≥ τ_stop; τ- - do
    for i = 1, i ≤ p, i++ do
        A^τ ← set();
        for elem in A_i do
            \\ we skip the first 2^α elements in A_1 only
            if i ≠ 1 ∨ elem.index > 2^α then
                \\ elem_[τ] = truncated-elem to τ
                A^τ.add(elem_[τ])
            end
        end
        for elem in B do
            if C[elem.index] == 0 then
                if elem_[τ] ∈ A^τ then
                    C[elem.index] = τ
                end
            end
        end
    end
end
\\ Check for "self"-collisions among elements of B
for τ = τ_start; τ ≥ τ_stop; τ- - do
    B^τ ← map(key : elem_[τ], value : boolean); \\ stores (key, value) pairs
    \\ 1st pass to track collisions
    for elem in B do
        if elem_[τ] ∉ B^τ then
            B^τ.add(elem_[τ], false)
        end
        else
            B^τ.set(elem_[τ], true)
        end
    end
    \\ 2nd pass to update C
    for elem in B do
        if B^τ.getValue(elem_[τ]) == true ∧ C[elem.index] < τ then
            C[elem.index] = τ
        end
    end
end
```

Algorithm 1: Variable truncation algorithm VarTruncate()

bits). Note this process is done in two separate phases, one for checking for collisions of baby-step elements against all all values in range $(2^{\alpha}, 2^{\alpha+\beta}]$ (to ensure that no collisions occur even when doing giant steps) and then for self-collisions between baby step elements.

Therefore from C we can serialize the respective table for 2^{α} elements by interleaving $(\lceil \log(\tau_{start} - \tau_{stop}) \rceil)$ bits per element. These bits encode the variable length and is necessary to make serialized parsing possible. Alternatively, we can reduce the number of those encoding bits by assigning them into k groups $G_1, G_2, .., G_k$, therefore needing $\lceil \log k \rceil$ bits per element. For instance, we can decrement τ by 2 bits each time instead of 1, and group G_1 would represent lengths of τ_{start} and $\tau_{start} - 1$ bits, group G_2 would represent lengths of $\tau_{start} - 2$ and $\tau_{start} - 3$ bits etc. Also, depending on the results, we might have these groups contain an uneven number of bit representations. For example, truncating with τ_{start} usually turns out to contain relatively very few elements, and therefore devoting a group for a very small population won't be efficient overall. Still each group G must encode the maximum τ that is included in that group, denoted by $\max \tau_G$. The total size of the serialized lookup table will then be

$$\sum_{i=1}^{k} (\max \tau_{G_i} \cdot |G_i|) + \lceil \log k \rceil \cdot 2^{\alpha}$$

where $\sum_{i=1}^{k} |G_i| = 2^{\alpha}$ and $\max \tau_{G_k} = \tau_{start}$. Note that even though the "grouping" approach reduces the factor $\lceil \log k \rceil \cdot 2^{\alpha}$, the granularity of the algorithm is also reduced, which overall increases the total size. On the other hand, more fine-grained groups will decrease the overall needed storage of the serialized lookup table, but will result in slightly increased computation cost in hashmap lookups[5].

As a special case of VarTruncate(), we can define FixedTruncate() (equivalently to [14]) where the algorithm stops when at least one collision is found. This essentially implies having a single group G_1 for all bit truncations we consider. In this case, no length encoding bits are needed, however the overall space savings are not optimal.

3.3 Optimizations

Although FixedTruncate() is a great step for decreasing lookup table costs, there is still room for further compression. In a concrete example with $n = 32$, the probability of no-collision for $\tau = 62$ is about 13% so attempting to truncate to a set with non-colliding elements might not be easy. However we can pick τ random bit positions instead and make non-collision tests, and keep testing combinations of those positions until we successfully get a non-colliding set. Then, the resulting truncated table needs to be accompanied by the subset of those indices that it

[5] Note that the cost of a hashmap lookup is insignificant compared to elliptic curve (EC) point addition (about 40 times in our implementation), while a scalar to EC point multiplication is around 32 times more expensive than EC point addition using the *double-and-add* method for small 32-bit scalars.

Input : Generator g, truncation paramenters τ_{start},τ_{stop} and encryption c
Data : Hashmap M for variable-length truncated elements in the range $[1, 2^{\alpha}]$
Output : Value x where $g^x = c$
$s \leftarrow g^{2^{\alpha}}$;
for $i \leftarrow 0; i < 2^{\beta}; i++$ **do**
  for $\ell = \tau_{stop}, \ell = \tau_{start}, \ell++$ **do**
    if $c_{[\ell]} \in$ M **then**
      return M.idx($c_{[\ell]}$) + $i2^{\alpha}$
    end
  end
  $c \leftarrow c/s$;
end
return "Not found"

Algorithm 2: Lookup decryption algorithm

corresponds to, and only these indices would need to be looked up on receipt of g^x. We refer to these bits as the *subset*. Note that in the above example, for $\tau = 60$ the probability of success becomes very small and given the computationally intensive process of finding a non-colliding *subset* makes further compression using this method practically infeasible. As an alternative and more practical approach, one can try hashing all points with some salt/counter, then truncate and check for collisions, instead of looking for optimized subsets [3].

3.4 Relaxed collisions and template distribution

In the algorithms previously discussed, we require zero collisions among truncated elements. However we could tolerate a *few* collisions among those elements (say κ collisions), which would reduce the table size even more as discussed in [3]. In case of ambiguity due to collisions in the lookup operations, up to κ exponentiations will be needed (worst case) to resolve this. As that these collisions imply additional exponentiations during lookup, this approach is essentially a space-computation tradeoff. For instance, in VarTruncate() group G_k (or even group G_{k-1}) typically contain relatively few elements, so one could discard these groups entirely to save 1 bit per element of encoding overhead, and provide those elements in a separate list L; then, one exponentiation would be needed in those (rare) cases.

Note that in some environments (e.g. limited connectivity, expensive cellular data etc.), downloading lookup tables, even if compressed, might still be prohibitive. However, one could construct a *template* using the output C from VarTruncate() or even simply τ from FixedTruncate(), and distribute these values only, which can be used as "advice" for the decrypting party to run the respective sub-algorithms directly with much less heavy computations.

4 Implementation

As discussed, the precomputation phase for compressing lookup tables is the most computationally-intensive process. The lookup table would need to be precomputed first using ComputeTable(), then further compress it by VarTruncate() or FixedTruncate(). However as shown from the existing applications in Section 2.3, a typical table decrypts up to 2^{32} values, and further optimizations are required to perform this step in a reasonable time with consumer-grade hardware. First, given that exponentiations are more expensive computationally, instead of computing g^x for all $x \in [1, n]$ separately, we can iteratively compute each $g^x = g^{x-1} \cdot g$ as multiplications are cheaper operations. Also, naively loading a huge table in memory and checking for collisions might be prohibitive in terms of computation and RAM. Therefore, we divide the table into parts (similar to our approach in Algorithm 1), then cross-check for collisions among all of them.

Note that the precomputation phase can be further improved by parallelization, e.g. assign threads to different truncation factors and merge the computed results after. Also, although a hashmap is usually preferred, for variable-length encoding, a trie structure might also be used for storing the needed compressed elements in memory, otherwise a few truncate-retries in the hashmap might be required. Again we stress that this computationally-intensive phase only needs to be performed once per curve and parameter set.

4.1 Truncation algorithm evaluation

We first implement FixedTruncate() for the secp256r1 and secp256k1 curves. We truncated from the MSB side by omitting the sign bit and we derived tables of unique size for 2^{32} with 64 bits per value, which results from 141GB to a compressed 34GB table in binary format, providing a compression factor over 1:4.

As discussed, the compression factor increases with the security parameter, e.g. for the secp521r1 curve it would be 1:8.35. As a starting point, we provide a precomputed table for the secp256r1 curve and 2^{28} values [9] of 2GB size. This table can be safely used in conjunction with Shanks up to 2^{32} values, as it is part of a 2^{32} truncated table where we ensure that no collisions exist. Normally, a final exponentiation would be needed to ensure that no value outside this range (or even a "garbage" value) has been encrypted with same prefix (or suffix), but we leverage existing range proofs in many blockchain systems to ensure this is not a possibility. However, if no indication about the max value exists, the final exponentiation can be performed by utilizing the faster "windowed" methods, such as w-ary non-adjacent

Table 1. Variable length truncation for the secp256k1 curve with $n = 32$, $\alpha = 20$, $\beta = 12$, $k = 4$.

Bits	Bits +encoding	# elements	Total Size
32	35	385479	13491765
36	39	599610	23384790
40	43	59450	2556350
44	47	3786	177942
48	51	238	12138
52	55	12	660
56	59	1	59
Size: 4.72MB		Compress: 1:6.98	

form (wNAF), which require small cache tables to speed up the exponentiation cost by a large degree [5].

For VarTruncate(), we evaluate our compression factor with the secp256k1 curve for $n = 32$, $\alpha = 20$, $\beta = 12$ and $k = 7$. These parameters are consistent with existing implementations in the blockchain domain as we discussed in Section 2.3, and our results are shown in Table 1. Note this requires $\lceil \log k \rceil = 3$ bits of length encoding per element, which overall results in about 393KB encoding overhead. We also performed tests for $n = 20, \alpha = 20, \beta = 0$ and $k = 7$ for the secp256k1 and secp521r1 curves, where we achieved a compression factor of 1:10 and 1:20 respectively. For $k = 4$ in the secp256k1 curve shown in Table 1, the compression factor slightly reduces to 1:6.85.

4.2 Complexity analysis and comparison

For generating a compressed lookup table g^x for $x \in [1, 2^n]$, our algorithm VarTruncate() includes a computationally intensive overhead in addition to just generating the table, similar to [14]. Specifically, [14] has an $O(2^n)$ computational overhead, while our algorithm has $O(k2^n)$ complexity. However this additional cost is paid only once for a specific set of parameters, while our algorithm achieves a) a significant improvement in compression factor and b) collision free encoding which simplifies used data structures.

For recovering the discrete log of g^x, the probabilistic [3] has $O(c2^{n/3})$ computation and storage complexity on average (for a very small $c < 2$), while our decryption computational cost involves $O(2^\beta)$ multiplications and $O(k2^\beta)$ map lookups in the worst case, where k is the number of utilized truncation groups. Note that that regular Shanks has a lookup multiplication complexity of $O(2^{\beta_2})$, for $\beta_2 > \beta$ (typically, for 256-bit curves $\beta_2 \approx \beta + 3$), when using the same precomputed table size with our scheme.

5 Conclusion

We presented and implemented several methods for *dlog* table compression by wisely truncating the serialized bytes of elliptic curve points, while guaranteeing no collisions. The proposed deterministic algorithm results in faster table lookups in addition to reduced storage and memory requirements. Although our technique relies on a potentially expensive precomputation phase, it is performed once per curve group and desired table size. Concretely, we show how we compress such tables for small ranges by up to a factor of 7 to 14 for 256 and 521-bit curves, respectively. For instance, the PGC [10] protocol currently implements a precomputed table of 264MB table size (using $\alpha = 23$ and $\beta = 9$) for a 256-bit curve and 2^{32} max amount range [2]. With our algorithm, that could be improved by requiring a variable-length table of roughly 38MB, which would make it applicable to mobile wallets and blockchain IoT devices.

Our work has additional advantages when encrypted content is accompanied by range proofs [7] or when faster windowed exponentiations are applied [5].

We also pave the way for the cryptography and blockchain community to create publicly-available compressed tables for commonly-used elliptic curves, by providing the first compressed 2^{32} lookup table for the secp256r1 curve [9]. Finally, we introduce the concept of optimized compression *templates* per curve, which will help developers and apps to recompute and verify such tables faster, especially when downloading is expensive or if bigger ranges (i.e. up to 2^{40}) are required.

References

1. Cube-root discrete-logarithm algorithms for secure groups, `http://cr.yp.to/dlog/cuberoot.html`
2. libpgc: a c++ library for pretty good confidential transaction system, `https://github.com/yuchen1024/libPGC/tree/master/PGC_openssl/PGC`
3. Bernstein, D.J., Lange, T.: Computing small discrete logarithms faster. In: INDOCRYPT 2012. LNCS, vol. 7668, pp. 317–338. Springer, Heidelberg (Dec 2012)
4. Bernstein, D.J., Lange, T.: Two grumpy giants and a baby. Cryptology ePrint Archive, Report 2012/294 (2012), `http://eprint.iacr.org/2012/294`
5. Blake, I.F., Murty, V.K., Xu, G.: A note on window τ-naf algorithm. Information Processing Letters **95**(5), 496–502 (2005)
6. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. In: FC 2020. LNCS, vol. 12059, pp. 423–443 (Feb 2020)
7. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: IEEE S&P (2018)
8. Cecchetti, E., Zhang, F., Ji, Y., Kosba, A.E., Juels, A., Shi, E.: Solidus: Confidential distributed ledger transactions via PVORM. In: ACM CCS 2017 (2017)
9. Chatzigiannis, P.: Compressed small discrete-log table python code and secp256r1 precomputed table, `https://www.dropbox.com/sh/8x8dvy3fv1cg83z/AABg7xChPRgVcQ3ApH1MC3vra`
10. Chen, Y., Ma, X., Tang, C., Au, M.H.: PGC: Decentralized confidential payment system with auditability. In: ESORICS 2020, Part I. vol. 12308 (Sep 2020)
11. Fauzi, P., Meiklejohn, S., Mercer, R., Orlandi, C.: Quisquis: A new design for anonymous cryptocurrencies. In: ASIACRYPT 2019, Part I. pp. 649–678 (2019)
12. Galbraith, S.D., Gaudry, P.: Recent progress on the elliptic curve discrete logarithm problem. Cryptology ePrint Archive, Report 2015/1022 (2015)
13. Galbraith, S.D., Wang, P., Zhang, F.: Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm. Cryptology ePrint Archive, Report 2015/605 (2015), `http://eprint.iacr.org/2015/605`
14. Mavroudis, V.: Computing small discrete logarithms using optimized lookup tables (2015), USCB, Koç Lab
15. Peng, K., Aditya, R., Boyd, C., Dawson, E., Lee, B.: Multiplicative homomorphic e-voting. In: INDOCRYPT 2004. LNCS, vol. 3348, pp. 61–72 (Dec 2004)
16. Pollard, J.M.: Monte Carlo methods for index computation mod p. Mathematics of Computation **32**, 918–924 (1978)
17. Pollard, J.M.: Kangaroos, monopoly and discrete logarithms. Journal of Cryptology **13**(4), 437–447 (Sep 2000). https://doi.org/10.1007/s001450010010
18. Shanks, D.: Five number-theoretic algorithms (1973)
19. Ugus, O., Hessler, A., Westhoff, D.: Performance of additive homomorphic ec-elgamal encryption for tinypeds. 6. Fachgespräch Sensornetzwerke (2007)