Computing small discrete logarithms using optimized lookup tables

Vasilios Mavroudis
Department of Computer Science, University of California, Santa Barbara, CA 93106.
E-mail: mavroudisv@cs.ucsb.edu

***Abstract*—In our previous work Crux [1], we used an additively homomorphic elliptic-curve cryptosystem, based on El Gamal, to compute privacy-preserving statistics for the Tor network [4].**

The decryption algorithm of our cryptographic scheme required the computation of a small discrete logarithm (DL). For small values, the computation of the DLP should have been trivial as an exhaustive search would quickly retrieve the correct result. However, in our case the discrete logarithms ended up being quite large and an exhaustive search proved to be quite slow for real-life applications.

In this work, we investigate an alternative approach, which uses an optimized lookup table of precomputed discrete logarithms. More specifically, we use a non-exhaustive lookup table to assist and speed up the computation of the small discrete logarithms. Similar approaches have been also been considered in the past [5], [3], [2]. Initially, we examine the related work in the field, we then outline our method, and finally we experimentally evaluate its performance. Based on our findings, we provide a python implementation, which we also incorporate in Crux.

I. Introduction

In recent years, homomorphic schemes are being applied increasingly often in real-life scenarios. This imposes the additional requirement of speed and efficiency. More specifically, in a research setting the complexity of an algorithm matters more than the actual runtime, whereas users often are not willing to wait more than very few seconds for an operation to complete (even if the complexity of the algorithm is outstanding). Two such examples of (partially) homomorphic schemes are [7] [6] where the authors apply their protocol in a variety of scenarios ranging from smart-metering to recommendation systems. However, in both schemes the decryption process requires the computation of a small discrete logarithm. Normally, this would not be a problem, since the encrypted value is small enough and an exhaustive search is enough to solve the problem. However, since those schemes are homomorphic, the result of the computations is not necessarily small. For instance, when the measurements of a large number of users/meters are aggregated. This can be a problem, as it can severely delay the decryptin operation and thus inhibiting the user-friendliness of the implementation. In this work, we propose a method which allows the rapid computation of the DLP for a wide enough range of values, especially designed to be applied in real-life uses.

[1]The source code of the implementation can be found at: https://github.com/mavroudisv/Crux

II. Related Work

The problem of solving small discrete logarithms has been studied by a number of researchers in recent years. One of the first papers in the area is [2] from Bernstein et. al in which they explain why the problem is of interest, its practical applications and introduce a novel algorithm which uses precomputation tables to reduce the runtime. Their suggested algorithm focuses on the Boneh, Goh, and Nissim protocol and uses a precomputation table to reduce the runtime of the decryption process, while they consume only limited storage space. Additionally, they also work on the problem of computing discrete logarithms in small groups, suggest algorithms which speedup the process, and discuss their practical applications in schemes and protocols. Moreover, Bernstein et. al used precomputations to attack popular cryptographic schemes [3]. In their paper, they clarify that their attacks are not practical and they do not pose a real threat in real-life settings, however, the paper shows that precomputations are a powerful tool which has many different application in modern cryptography. Finally, Galbraith's [5] work summarize the current state of the art for algorithms capable of solving the discrete logarithm problem for elliptic curves (ECDLP). Furthermore, they explain the groups under which each algorithm performs better, their drawbacks and eventually examine potential combinations.

III. Cryptosystem & Problem definition

Crux utilizes an Additively Homomorphic Elliptic-Curve cryptosystem based on a variant of El Gamal [1]. The cryptosystem comprises of the following three algorithms:

KeyGen(1^n): Given a security parameter n, elliptic curve E and a generator g are selected forming a group with order q. For the generation of the key pair, we choose $priv \in \mathbb{Z}_q$ and then we compute the public key as $pub = priv \cdot g$. The output consists of (E, g, pub) which is public and the private key $priv$.

Encryption(pub, m): Given the public key and a plaintext message m the algorithm computes the ciphertext $Ct = (A, B)$, where $A = k \cdot g$, $B = k \cdot pub + m \cdot g$ and $k \in \mathbb{Z}_q$ is randomly chosen.

Decryption(Ct, priv): During the decryption the two parts of the ciphertext and the private key are combined as such: $B - x \cdot A = m \cdot g$. This gives us $m \cdot g$ but not m itself. To extract m

we need to solve a discrete logarithm problem, however, this is not computationally hard since the range of m is limited and hence we can compute $i \cdot g$ for all the possible i's.

However, as discussed in the previous sections, since the cryptosystem is used for privacy-preserving statistics the encrypted values end up being quite large. This often happens when executing a protocol for the computation of a metric/statistic (e.g. mean, median, variance) between a large number of entities. The individual values that each entity is encrypting may be relatively small. However, the decryption of the sum of a large number of ciphertexts or their squares, is not trivial and the inefficiency of the exhaustive search will result in delays. In the following section we present a more efficient alternative to solve this problem.

IV. Proposed Solution

Our proposed solution is based on the observation that a lookup table (with precomputed pairs of values) can be used to speed up the slow decryption process. Such a table contains all possible key-value pairs within a user defined range of values. More specifically, each row contains a key $i \cdot g$ and the value of the corresponding i. The algorithm for the construction of the table is in algorithm 1.

```
1  func gen_lookup (lower_bound, upper_bound, limit)
   Input : Two integers (lower_bound, upper_bound)
           defining the range of the secrets and a third
           integer which defines the number of bytes kept
           from each curve point.
   Output: table
2  G ← EcGroup(nid=713);
3  g ← G.generator();
4  o ← G.order();
5  table = {}
6  ix = lower_bound · g;
7  for i=lower_bound to upper_bound do
8  |   trunc_ix ← substring(ix, trunc_limit);
9  |   if trunc_ix in i_table then
10 |   |   table[trunc_ix].append(i);
11 |   else
12 |   |   table[trunc_ix] ← i;
13 |   end
14 |   ix ← ix + g
15 end
16 return table;
```

Algorithm 1: Algorithm for generating a non-exhaustive lookup tables used when solving small instances of the discrete logarithm problem.

Even though the approach with the exhaustive lookup table works as expected it essentially offloads the computational overhead to the storage. Since storage is quite cheap this is preferable, however, we were able to further improve our algorithm to get the best of both worlds. Namely, reduce the computation time and minimize the table size. Our optimization is based on the observation that the stored $i \cdot g$ takes up most of the space compared to i. For this reason, we designed a simple mechanism which allows us to:

- Reduce the number of bytes of $i \cdot g$ that we store
- Avoid storing all $i \cdot g$, but keep storing all i's which are relatively small

More specifically, we truncate $i \cdot g$ and instead of storing all its bytes we store only some of them. If this results in a collision with another element then we store both values (i.e. i's) under the same key. In practice this means that we no longer have an 1-1 pairing between the keys and the values, but each key holds multiple i's. The number of bytes to be stored is parametrized by the user.

Of course, since the pairing in the table is no longer 1-1, the retrieval mechanism should be adapted accordingly. In particular, the algorithm works as follows: given the value $i \cdot g$ and the number of bytes kept when the lookup table was generated, it truncates the value and then uses it as a key to retrieve the record from the table. Since each record may contain multiple i's, the algorithm evaluates each one of them and returns the one that solves the instance. This can be seen in a more formal format in algorithm 2.

```
1  func small_dlp_solver (point, limit)
   Input : The curve point for which we want to solve
           ECDLP (point) and integer which defines the
           number of bytes kept from each curve point
           (limit).
   Output: The solution x to the given ECDLP instance.
2  xs ← table[point[:trunc]].split(',');
3  G ← EcGroup(nid=713);
4  g ← G.generator();
5  for x in xs do
6  |   if x · g == point then
7  |   |   return x;
8  |   end
9  end
10 return None;
```

Algorithm 2: Algorithm for generating the lookup tables used when solving small instances of the discrete logarithm problem.

As seen in section V the proposed technique reduces the runtime significantly, while it requires relatively small storage space.

V. Experiments

In this section, we design and run experiments to determine the efficiency of our proposed technique and evaluate the relationship between the various parameters. More specifically, the first experiment evaluates the speed-up obtained when using an exhaustive lookup table compared to an algorithm which solves the ECDLP by exhaustively scanning all possible values. For the experiment, the two DLP solving algorithms were implemented and then used to solve 100 random instances of the ECDLP, with i being randomly chosen within the range (-200000, 2000000). As seen in table II the exhaustive search method is very slow and this would make it impractical in real-life uses. On the other hand the method using the lookup table is rapid but has the drawback of a fast growing filesize.

Method	Time (sec)	Filesize (MB)
Bruteforce	297.508512	0
Lookup	0.00248408	104

TABLE I
ALGORITHM FOR GENERATING THE LOOKUP TABLES USED WHEN SOLVING SMALL INSTANCES OF THE DISCRETE LOGARITHM PROBLEM.

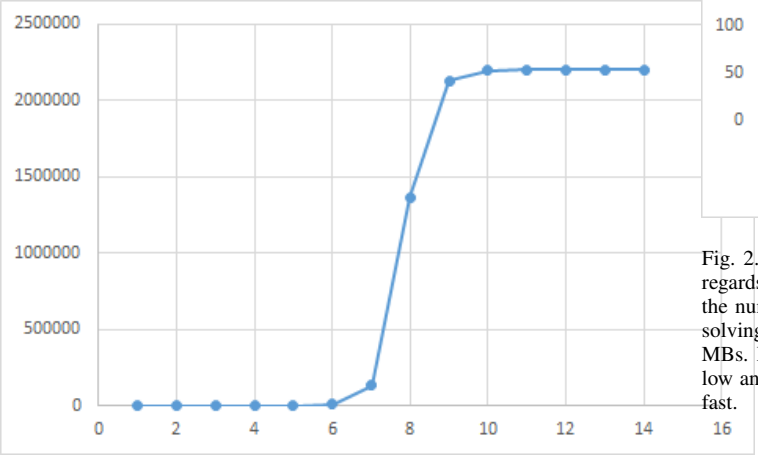

Fig. 1. The figure demonstrates how the number of bytes correlates with the number of keys/items in the lookup table. The x axis shows the number of bytes stored, while the y axis shows the number of unique items. We observe that the number of unique keys is very low and relatively stable when storing up to 6 bytes, however it then grows very fast until it stabilizes around 10 bytes.

In the second experiment we wanted to study how the key grouping optimization affects the filesize and the time needed to solve an instance of the problem. The setup was similar with the previous experiment. 100 instances of the ECDLP problem were solved and the median of the runtimes is reported in figure V. As seen the size of the table gradually grows as we store more bytes, while the time needed to solve an ECDLP instance gets reduced very fast. From the graph it becomes apparent that 3 or 4 bytes offer the best trade off between filesize and runtime. A more detailed overview of the data can be seen in table **??**.

Bytes	Time (sec)	Filesize (MB)	Unique Keys
1	157.4811983	16	1
2	84.91564391	16	3
3	5.73622117	16.5	33
4	0.40430955	17	513
5	0.02280284	31	8193
6	0.00366381	32	131073
7	0.00258743	65	1362184
8	0.00245893	88	2129442
9	0.0024523	89	2195549
10	0.00244919	104	2199730
11	0.00245336	104	2199982
12	0.00245889	104	2200000
13	0.00248408	104	2200000

TABLE II
THE TABLE SIZE AND THE AVERAGE RUNTIME FOR THE DIFFERENT NUMBERS OF BYTES.

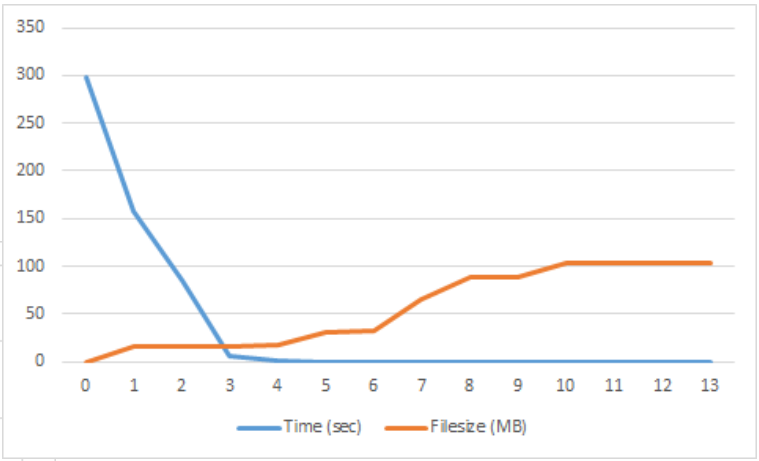

Fig. 2. The figure demonstrates how the runtime and the filesize change in regards to the number of bytes stored in the lookup table. The x axis shows the number of bytes stored, while the y axis shows both the time needed for solving an instance of the ECDLP (in seconds) and the size of the table in MBs. More specifically, we observe that the number of unique keys is very low and relatively stable for the first 6 characters, however it then grows very fast.

VI. CONCLUSIONS & FUTURE WORK

In this work, we attempted to solve the problem of finding small discrete logarithms by combining cryptographic tools with a technique from the realm of ”systems”. The problem of solving small instances of the discrete logarithm problem for elliptic curves has immediate practical implications as many cryptographic schemes with homomorphic properties require the computation of a small DL. In normal end-to-end encryption scenarios the problem is trivial as the range of the plaintext is very limited. However, in cases were the homomorphic properties of the scheme are utilized to do computations between many parties it is likely that the final ciphertext does not fall within the restricted range, making the discrete logarithm problem easy to solve but not trivial. This suggests that the computation is not instant any more, and instead takes few seconds to complete, hence possibly impairing the user experience. Our proposed technique tackles this problem and provides a way to solve the ECDLP in a much large range instantly. The results of our experiments show that our method was successful and we were able to both minimize the runtime of the algorithm and prevent the tables from growing too much when the range increases. An interesting direction for further research would be to examine if this or other ”system” approaches would be applicable on top of the techniques proposed by Bernstein in [3], [2]. Additionally, a generalized approach for various types of schemes would also be of interest.

REFERENCES

[1] Josh Benaloh. Dense probabilistic encryption.
[2] Daniel J Bernstein and Tanja Lange. Computing small discrete logarithms faster. In *Progress in Cryptology-INDOCRYPT 2012*, pages 317–338. Springer, 2012.
[3] Daniel J Bernstein and Tanja Lange. —non-uniform cracks in the concrete: the power of free precomputation. In *Advances in Cryptology-ASIACRYPT 2013*, pages 321–340. Springer, 2013.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.
[5] Steven D Galbraith and Pierrick Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *Designs, Codes and Cryptography*, pages 1–22, 2015.
[6] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *Privacy Enhancing Technologies*, pages 175–191. Springer, 2011.
[7] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches, 2015.