Abstract

We show that Onion ORAM [DvDFR15], a constant bandwith oblivious RAM, is not practical when used in conjuction with the additively homomorphic encryption Damgård-Jurik [DJ01].

In particular, we show that a single read/write operation of a block of size $0.125MB$ *out of Onion ORAM with capacity of 1 GB and with 1024-bit RSA modulus for Damgård-Jurik is expected to take about 60 days under a reasonable Python implementation. Moreover, we show that even if using GNU MP Bignum Library and C++, the same operation is expected to take at least one day. Therefore, we conclude that Onion ORAM is not practical when coupled implemented with Damgård-Jurik as a cryptographic primitive.*

Finally, we hope that the Python implementation is useful for understanding Onion ORAM works, and that it might be found useful as a starting point for implementing Onion ORAM with a different cryptographic primitive.

I. Introduction

Oblivious RAM (ORAM) is a cryptographic primitive that allows client's data to be stored on an untrusted server without any observer learning anything neither about the data, nor about the client's access pattern to the data.

The ORAM model, which began with the work of Goldreich and Ostrovsky [Gol87, GO96], assumed that the server was just a storage that a client could read from and write to, but could not perform any computation. In their seminal paper [GO96], Goldreich and Ostrovsky showed that ORAM of N blocks must incur a $O(\log N)$ lower bound in bandwidth blowup under $O(1)$ blocks of client storage.

However, if we allow server computation, this lower bound no longer applies. In particular, Onion ORAM [DvDFR15] is a construction that uses server-computation to achieve constant bandwidth blowup and poly-logarithmic server computation. Moreover, Onion ORAM relies only on additively homomorphic encryption as opposed to fully homomorphic encryption (FHE) [AKST14] or somewhat homomorphic encryption (SWHE) [GHJR14].

Taking into account the existence of easy-to-implement additively-homomorphic-encryption schemes, such as Damgård-Jurik cryptosystem [DJ01], it's natural to ask if Onion ORAM, in addition to being of theoretical interest, can also be of practical use. This project aims to answer this question by implementing Onion ORAM with Damgård-Jurik in Python, and measuring its performance under different scenarios.

II. The Damgård-Jurik cryptosystem

The Damgård-Jurik cryptosystem is a generalization of the Paillier cryptosystem [Pai99] that performs calculations in $\mathbb{Z}/n^{s+1}\mathbb{Z}$ where n is an RSA modulus, and s is a positive natural number.

The procedure for generating a pair of public and private keys is given in Algorithm 1. Note that we can generate a random $(k/2)$-bit prime number by repeatedly generating a $(k/2)$-bit random integer, and checking whether it is prime by the Miller-Rabin probabilistic primality test [Rab80]. The encryption and decryption procedures are given respectively by Algorithm 2 and 3.

Algorithm 1 Generate-Keys(k, s)

Generate two random $(k/2)$-bit prime numbers p and q independently of each other.
Let $n = pq$ be an RSA modulus.
Let $\lambda = \text{lcm}(p-1, q-1)$.
Compute d such that $d \equiv 1 (\text{mod } n^s)$ and $d \equiv 0 (\text{mod } \lambda)$ via the Chinese Remainder Theorem.
The encryption key is given by (n, s).
The decryption key is given by d.

The encryption algorithm involves two modular exponentiations and one modular multiplication. Modular exponentiation is implemented as $O(\log_2(n))$ modular multiplications through repeated squaring. Noting that Python uses Karatsuba's algorithm with running time $O(b^{\log_2 3})$ [KO63] for multiplying two b-bit numbers, we obtain an implementation of Damgård-Jurik encryption algorithm with running time

$$O((\log_2(n)s)^{1+\log_2 3}) = O((ks)^{2.58}) \tag{1}$$

for an RSA modulus of k bits and base s. Empirical results for the running time of the encryption algorithm for various pairs of k and s are given in Figure 1.

Algorithm 2 Encrypt(m)

Generate $r \in \mathbb{Z}/n^{s+1}\mathbb{Z}$.
Compute the encryption of the message m: $e = (1+n)^m r^{n^s} (\text{mod } n^{s+1})$.

$\log_2(n) \diagdown s$	1	2	3	4	5
256 bits	0.001852	0.005295	0.010898	0.017616	0.039031
512 bits	0.008663	0.028546	0.067480	0.129059	0.201658
1024 bits	0.052233	0.189566	0.450596	0.836500	1.370912
2048 bits	0.335956	1.281110	3.106418	5.872886	10.464225

Figure 1: *Execution time (in seconds) of Damgård-Jurik encryption (Algorithm 2) averaged over 10 executions.*

The decryption algorithm of the Damgård-Jurik cryptosystem is more involved as can be seen in Algorithm 3. However, it's running time is essentially the same as that of the encryption algorithm, because the single operation that takes the most time is the modular exponentiation with modulus n^{s+1}; the rest of the computations are modular multiplications and subtractions with modulus at most n^{s+1}, and modular exponentiations with modulus at most n^s, all of which are dominated by the modular exponentiation with modulus n^{s+1}. Note that the binomial coefficient $\binom{m}{k}$ involves only $k \in [2, s]$ and can be computed as

$$\binom{m}{k} = \frac{m(m-1)(m-2)\dots(m-k+1)}{k(k-1)\dots 1},$$

which is just the quotient of two products of at most s terms each. We also note that it is possible to precompute the modular inverses for each $k \in [1, s]$ once without affecting the asymptotic complexity. Empirical results for the running time of the decryption algorithm for various pairs of k and s are given in Figure 1.

Algorithm 3 Decrypt(c)

Let $m = 0$.
for $j = 1$ **to** s **do**
  Let $m' = \dfrac{c^d \pmod{n^{j+1}} - 1}{n}$.
  for $k = 2$ **to** j **do**
    Compute $m' = \left(m' - \binom{m}{k} n^{k-1}\right) \pmod{n^j}$.
  end for
  Let $m = m'$.
end for
The decrypted message is contained in m.

$\log_2(n) \backslash s$	1	2	3	4	5
256 bits	0.002635	0.005171	0.010291	0.020757	0.035139
512 bits	0.011517	0.031397	0.081265	0.123845	0.198052
1024 bits	0.066245	0.215878	0.460127	0.821298	1.385053
2048 bits	0.444607	1.355042	3.078118	5.863956	10.308321

Figure 2: *Execution time (in seconds) of Damgård-Jurik decryption (Algorithm 3) averaged over 10 executions.*

One of the crucial properties of the Damgård-Jurik cryptosystem that makes it possible to use as a primitive for Onion ORAM is that it is *additively homomorphic*. In particular, let m_1 and m_2 be two plaintexts from a suitable plaintext space $\mathbb{P}$, and let $\mathcal{E} : \mathbb{P} \to \mathbb{C}$ be the Damgård-Jurik encryption algorithm, where $\mathbb{C}$ is the corresponding ciphertext space. Then

$$\mathcal{E}(m_1) \oplus \mathcal{E}(m_2) \triangleq \mathcal{E}(m_1) \cdot \mathcal{E}(m_2) = \mathcal{E}(m_1 + m_2),$$

where $\oplus$ corresponds to "encrypted addition". This enables an untrusted party to perform encrypted additions with only having access to the encrypted addends and neither having access to the unencrypted counterparties, nor being able to gain access to them. Moreover, it is possible to implement scalar multiplication by 0 or 1. Let b zero or one. Then

$$\mathcal{E}(b) \otimes \mathcal{E}(m) \triangleq \mathcal{E}(b)^{\mathcal{E}(m)} = \mathcal{E}(b\mathcal{E}(m)) = \begin{cases} \mathcal{E}(0), & b = 0, \\ \mathcal{E}(\mathcal{E}(m)), & b = 1. \end{cases}$$

Encrypted scalar multiplication allows an untrusted party to multiply an uncrypted scalar by an encrypted number without learning any of the numbers. However, this operations adds

one complication: the result is contained in a 2-layer encryption. Those layers are called onions in [DvDFR15], which is where the name comes from. Encrypted scalar multiplication is interesting by itself, but its power comes when it is used to implement encrypted selection. In particular, let $b \in \{0,1\}^n$ be an n-dimensional selection vector which contains $n-1$ zeroes, and exactly one 1 (i.e. $\sum_{i=1}^{n} b_i = 1$.), and let m be an n-dimensional vector, where each coordinate is some message. Then, it is easy to verify that

$$\bigoplus_{i=1}^{n} (\mathcal{E}(b_i) \otimes \mathcal{E}(m_i)) = \mathcal{E}(\mathcal{E}(m_i^*)), \text{ where } b_i^* = 1.$$

Such an operation allows an untrusted party to select one out of n encrypted messages without knowing what message was selected since the server only learns the encrypted scalar multiples $\mathcal{E}(b_i)$. Moreover, after performing the computation, the result is encrypted in additional layer, and in general the resulting encrypted selected message will not be equal to any of the initial n encrypted messages due to the additional layer of encryption. This basic operation is the building block of Onion ORAM [DvDFR15], because it allows the server to perform all necessary computations only over encrypted data.

The running time of all server computations in Onion ORAM can be approximated as a multiple of modular exponentiation, because all other computations are dominated by the running time of modular exponentiation. Therefore, it is important to take a look at how long it takes to perform a modular exponentiation. Figure 3 shows the running time of modular exponentiation of n-bit numbers. Note that the underlined values are estimated with the best fit of the class $an^{2.58}$ for $a \in \mathbb{R}$ via linear regression, where the 2.58 exponent matches the exponent of the running time for derived in equation (1). For a comparison, the third column shows the running time of modular exponentiation from the GNU MP Bignum Library. The last column corresponds to the expected speedup from switching from Python to GMP arbitrary-precision integer computations.

n	t_{Python} (s)	t_{C-GMP} (s)	t_{Python}/t_{C-GMP}
512	0.001	0.0003	3.333
1024	0.010	0.001	10.000
2048	0.059	0.008	7.374
4096	0.405	0.056	7.232
8192	2.743	0.312	8.791
16384	21.183	1.179	17.966
32768	158.025	6.562	24.081
65536	<u>946.214</u>	32.904	28.756
131072	<u>5663.375</u>	165.053	34.312
262144	<u>33869.157</u>	825.8131	41.013

Figure 3: *Execution time (in seconds) of a modular exponentiation with n-bit base, exponent and modulus. The third column shows the running time of* ***mpz_powm****: GMP's implementation of modular exponentiation.*

III. Onion ORAM

A complete Python implementation of Onion ORAM with Damgård-Jurik as a cryptographic primitive can be found by following this link to a Dropbox folder: it contains 694 lines of source code, and 234 lines of testing code. Note that the implementation follows closely [DvDFR15]. However, using it for any real-world scenario turns out to be prohibitively slow, which doesn't come as a surprize considering how long it takes to perform a modular exponentiation of big numbers as seen in Figure 3.

We are going to create a simple model that approximates the actual (in seconds) running time (as opposed to asymptotic running time) of a read and write operations of Onion ORAM by considering only the number of modular exponentiations involved in the computation. This is a reasonable approximation because all client and server computations involve modular exponentiations which take significantly more time than any of the other computations do.

Before constructing the model, let's consider the parametrization required for setting up an instance of Onion ORAM. This parametrization is in line with that of original paper [DvDFR15].

The security parameter λ signifies that the Onion ORAM system can fail with probability of at most $2^{-\lambda}$. For practical purposes, we choose $\lambda = 80$. In addition to the security parameter λ, Onion ORAM requires a single pair of public/private keys, which we generate according to Algorithm 1 due to using Damgård-Jurik; we denote the bit-length of the RSA modulus n as $\text{RSA\_}\log n$. The Onion ORAM data structure consists of a complete binary tree of $L+1$ levels, where each node is called a *bucket*. Each bucket contains $\Theta(\lambda)$ *blocks* where a *block* denotes the smallest amount of information that the client has access to. Note that even though [DvDFR15] recommends having $\Theta(\lambda)$ blocks per bucket, taking constant terms into consideration, a reasonable value to choose is $c\lambda$ blocks per bucket, for $c = 5$ and $\lambda = 80$, as chosen above, or 400 blocks per bucket in total. In order to achieve constant time bandwidth ORAM, it becomes necessary to divide each block in the system into equally-sized k chunks, where the size of each chunk is given by $s_0 \cdot \text{RSA\_}\log n$, where s_0 corresponds to the parameter s of Damgård-Jurik for a single chunk of a block at the root of the binary tree. Let the size of a single block be given by

$$B = ks_0\text{RSA\_}\log n.$$

Having in mind all of the above parameters, it is possible to get specific values for all of them by choosing values for only three of them: total storage of the system, size of a single block, and size of the RSA modulus ($\text{RSA\_}\log n$), and deriving the rest of the parameters by optimizing for some metric. In particular, we are going to optimize for bandwidth. A Python script that performs this optimization was written by Christopher Fletcher and Ling Ren, who are authors of Onion ORAM [DvDFR15], and this script was used to derive the parametrizations that follow. We are going to approximate the running time of Onion ORAM with total storage of $S = 1\,GB = 2^{33}$ bits with a single block size of $B = 0.125MB = 2^{20}$ bits under four different RSA moduli: 128, 256, 512, and 1024 bits. Note that to have a reasonably secure system, we need to use a big RSA modulus considering the attacks on factorization that already exist. In particular, using 1024-bit RSA modulus might be sufficient, whereas 128-bit modulus is not recommended for real-world setup. By using the optimization script, we get that $L = 5$, and the rest of the parameters depend

on RSA_$\log n$ as given in Figure 4.

RSA_$\log n$	128	256	512	1024
s_0	82	57	41	27
k	99	71	49	37

Figure 4: *Parametrization for s_0 and k given size of RSA modulus.*

Now, we are going to approximate running time of Onion ORAM by focusing only on reads and write. Note that evictions along a path can be ignored, since their running time is amortized over a number of reads or writes. A single read or write in Onion ORAM is performed by considering all the data along a single path in Onion ORAM tree. In particular, to read a single chunk of a block, we need to perform one select operation with $(L+1)c\lambda$ arguments of bit-length (s_0+L)RSA_$\log n$. A single select operation consists $(L+1)c\lambda$ modular exponentiations. Therefore, a single read/write in Onion ORAM requires $k(L+1)c\lambda$ modular exponentiations of arguments of bit-length (s_0+L)RSA_$\log n$. The running time of a single scalar multiplication with appropriate arguments can be timed by running a simple Python script.

RSA_$\log n$	128	256	512	1024
s_0	82	57	41	27
k	99	71	49	37
(s_0+L)RSA_$\log n$	11136	15872	23552	32768
$k(L+1)c\lambda$	237600	170400	117600	88800
time for 1 $\otimes$ (in s)	10.701	26.051	80.297	158.025
time for 1 read/write (in h)	33.101	114.855	525.320	1438.37

Figure 5: *Approximate running time for a single read/write Onion ORAM operation.*

Taking all of this into account, we can deduce the running time of a single read/write Onion operation as the time it takes to perform $k(L+1)c\lambda$ scalar multiplications. These results can be found in the last row of the table in Figure 5. Note that the running time for 1 read/write operation is given in hours as opposed to seconds. Taking into account that GNU MP Bignum Library can give us a speed of at most 24.081 on the underlying modular exponentiations, we observe that a single read/write operation is expected to take at least an hour on a commodity computer even under optimized C++ arbitrary-precision arithmetic libraries, and under a choice of 128-bit RSA modulus. On the other hand, under the choice of a 1024-bit RSA modulus, the approximate running time is measured in days.

IV. Discussion

Needless to say, Onion ORAM is prohibitively slow when combined with Damgård-Jurik on real-world scenarios. Moreover, it's likely that using some highly-performant modular exponentiation

would change the negative results above. The resulting Python implementation of Onion ORAM might be of use if another additively homomorphic primitive is used instead of Damgård-Jurik.

V. Acknowledgements

I would like to thank Prof. Srini Devadas for supervising this 6.UAP project. I am grateful to Ling Ren and Christopher Fletcher who helped me get up to speed and understand the intricacies of Onion ORAM.

References

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 131–148, 2014.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, pages 119–136, 2001.

[DvDFR15] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, and Ling Ren. Onion ORAM: A constant bandwidth and constant client storage ORAM (without FHE or SWHE). *IACR Cryptology ePrint Archive*, 2015:5, 2015.

[GHJR14] Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. *IACR Cryptology ePrint Archive*, 2014:345, 2014.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194, 1987.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. 1963.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 223–238, 1999.

[Rab80] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128 – 138, 1980.