Fast Bulk GCD Implementation in Go to Detect Shared Primes in RSA Keys

Best Secr-ity Team: Siqi Chen, Yunfei Lin, Connie Siu, Elise Xue

Massachusetts Institute of Technology

Abstract. The purpose of this project, proposed by Let's Encrypt, is to implement a very fast bulk GCD computation for RSA keys in Go, so that it would be possible to identify weak RSA keys at request time. Using Bernstein's quasi-linear bulk GCD algorithm [Ber08], we obtained RSA moduli with shared primes. Throughout the course of the project, we performed multiple optimizations on our algorithm, eventually being able to identify weak keys amongst millions of RSA keys within a reasonable amount of time.

Keywords: RSA, Weak RSA Keys, Bulk GCD Algorithm

1 Introduction

The effectiveness of RSA depends on the difficulty of factoring the two primes that make up the RSA modulus. If an adversary can retrieve the primes from the RSA modulus, then he can easily decrypt the data. One weakness of RSA is that some of the RSA keys currently in use share prime factors. These keys are particularly vulnerable - if an adversary can factor one of the RSA moduli, he can easily factor others that share the same prime.

For this project, we implemented a fast bulk GCD computation for RSA keys in Go, and constructed a set of known weak RSA keys from the 4.3 million RSA keys collected. In this way, Let's Encrypt, a new certificate authority launched in 2016, can test and reject weak keys at request time.

2 Background

2.1 RSA

RSA is a cryptosystem that encrypts sensitive data to send over the Internet. It uses a public key, which is shared with everyone, to encrypt messages, and a private key, which is only known to the receiver, to decrypt the data. The RSA keys are also used to digitally sign and verify messages.

The key generation step in RSA involves randomly generating two distinct large primes p and q, by the Rabin-Miller primality test algorithm, and computing the modulus $n = pq$ as the public key. Then, it finds an exponent e that is relatively prime to $(p-1)(q-1)$, the totient of n, as the second public key. The private key d is the multiplicative inverse of e with respect to $(p-1)(q-1)$, as determined by the Extended Euclidean algorithm.

2.2 Shared-Prime in RSA Moduli

Many RSA keys are currently in use in the Web. The two prime factors used in the RSA modulus are created using a random prime number generator. However, if the implementation of this generator is incorrect (e.g. the prime numbers are generated with low entropy), keys with common prime factors may be generated. In fact, some keys collected from the Web do share prime factors. These keys are called weak RSA keys [FNI15].

According to Ron was wrong, Whit is right [LHA$^+$12], among the 4.7 million distinct 1024-bit RSA moduli the authors collected, 12720 of them shared prime factors with some other moduli. The trend seems to stay: in their more recent collection of 11.4 million RSA moduli, there were 26965 weak keys, including ten 2048-bit ones.

Weak RSA keys are vulnerable to attacks. Suppose that there are two distinct public keys (n_A, e_A) and (n_B, e_B), and that $\gcd(n_A, n_B) \neq 1$. Since n_A and n_B have only two distinct prime factors, $\gcd(n_A, n_B)$ is one of the secret primes of both keys, then the attacker can easily find the other secret primes by calculating $n_A / \gcd(n_A, n_B)$ and $n_B / \gcd(n_A, n_B)$. With the secret keys, the attacker is able to decrypt any transmitted messages and send malicious messages by pretending to be the sender.

2.3 Existing Models for Detecting Weak RSA Keys

The naive way to detect weak RSA keys is to factor the RSA moduli. However, even with number field sieve, the fastest known factoring method, it would take thousands of years to factor one million RSA keys [KAF$^+$10].

Finding pairwise GCD is efficient, which takes $O(n^2)$ for each n-bit number pair. With fast integer arithmetic, the complexity of pairwise GCD can be improved to $O(n(\log n)^2 \log\log n)$ [Ber08]. Using the GMP library, computing the GCD of two 1024-bit RSA moduli takes approximately 15 μs on a current mid-range computer. Hence, to find weak keys among one million RSA keys, it would take 86 days to finish the calculation of $5 * 10^{11}$ pairwise GCDs.

A faster way is to find weak RSA moduli by computing the GCD of many pairs of RSA moduli. To achieve this, Fujita, Nakano and Itomay [FNI15] presented Approximate Euclidean algorithm, a faster Euclidean algorithm for finding the GCD between all pairs of RSA moduli. The algorithm computes an approximation of quotient by just one 64-bit division and uses it for reducing the number of iterations of the Euclidean algorithm. The experimental results showed that the GPU implementation based on Approximate Euclidean Algorithm is more than 9 times faster than the best known published GCD computation using the same generation GPU.

Heninger, Durumeric, Wustrow and Halderman [HDWH12] adopted Bernstein's quasi-linear bulk GCD algorithm [Ber08] and were able to detect weak keys among the 11,170,883 distinct RSA moduli in around 60 hours using a single core on a machine with a 3.30 GHz Intel Core i5 processor and 32 GB of RAM.

3 Implementation

3.1 Access to RSA Keys

Nadia Heninger recommended us to obtain the RSA keys from Censys.io, a search engine that enables researchers to look up information about hosts and networks on the Internet. Censys.io has a Data Export Tool that allows us to export large amounts of data using SQL. We used the query `select parsed.subject_key_info.rsa_public_key.modulus from certificates.certificates` to find the modulus of the RSA public keys. The result gave us 500 CSV files, which totaled to 980 GB. Since we do not have 980 GB available on our computer, we decided to take the first 13 files for this project. After removing duplicate moduli, we were left with 4.3 million distinct RSA moduli of various lengths.

3.2 Algorithm for Efficiently Computing All-Pairs GCDs

Daniel J. Berstein [Ber08] introduced a quasilinear bulk GCD algorithm that efficiently computes the GCD between each element and the product of all elements. The algorithm involves a product tree and a remainder tree.

As shown below, the algorithm starts out by generating a product tree involving all of the k RSA moduli. Each moduli is multiplied pairwise with another moduli, generating a node in the next level of the tree. This ultimately constructs a binary tree of products with the root node being the product of all the input moduli.

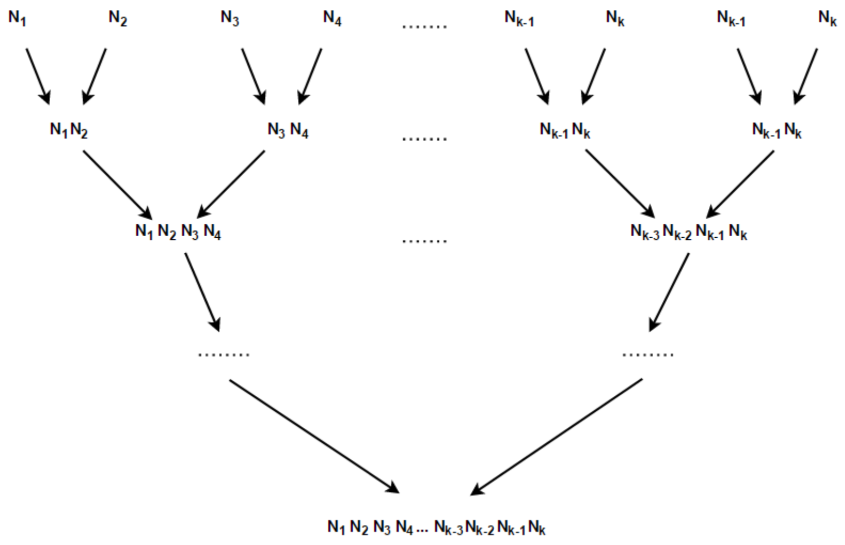

Fig. 1: Product tree

A remainder tree is generated using the product tree. The root node's value is the product of all of the input RSA moduli. As we work down the tree, each node takes the value of the parent modulo the square of the corresponding node in the product tree. This continues for all of the nodes in the tree.

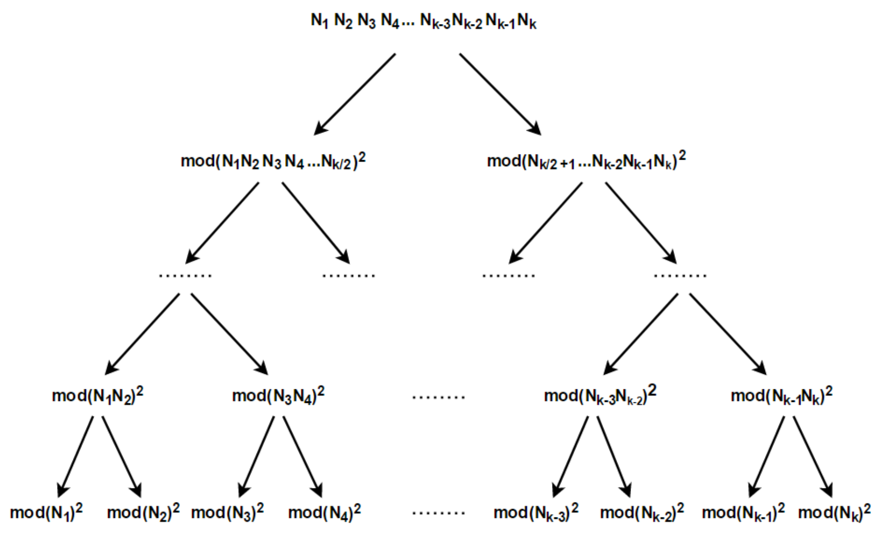

Fig. 2: Remainder tree

Taking the values stored in the leaf nodes, we divide them by the corresponding RSA moduli, then take the resulting quotient and calculate the GCD between that value and the corresponding RSA moduli. If the GCD is not 1 (i.e. the two numbers are not relatively prime), we have managed to factor the RSA moduli and denote that key as weak.

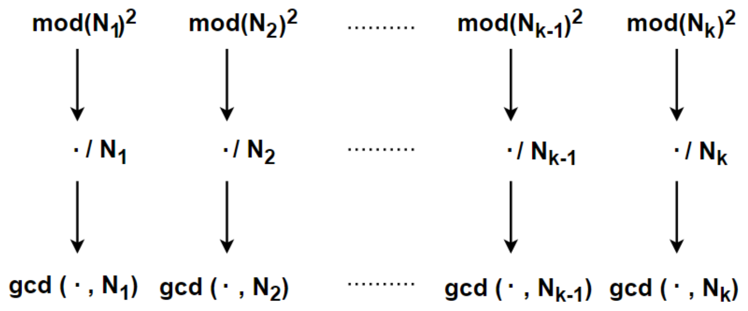

Fig. 3: Computing bulk GCDs

3.3 Go Implementation of Bulk GCD

Generating Product Tree Our algorithm takes in a file of newline separated RSA moduli in the hex form and processes them into an array of integers, which is stored in the file "p0.txt". Every two integers in the array are then multiplied together and the product is put into a second array of products. When this second array representing a new "level" of the product tree is completed, it is then written to a text file labeled with "p" and the level number. After $\log k$ levels, the $\log k$th product file contains the product of all the RSA moduli multiplied together.

Generating Remainder Tree We start by reading $\log k$th product file, which contains final RSA product. We will call this product N. We then read in the file for the $\log k - 1$ product level. Then, for each product that represents a "child" of the previous node (the node containing N), we take the result of N modulo the product squared and append that to a remainder list. When all of the remainders for that level are calculated, the array representing that level is written to a text file labeled with "r" and the level number (level number corresponds to the level in the product tree). The leaves of the remainder tree are thus written to "r0.txt".

Calculating GCD We use the values inside of "r0.txt" for the final step of the algorithm. The value of each leaf of the remainder tree is first divided by the corresponding RSA moduli in "p0.txt". We then take the GCD of the resulting quotient and its corresponding RSA moduli. If the GCD is not equal to 1, we know that the key is weak. The weak RSA moduli are written to "vulnerable.txt", and the non-trivial GCDs are written to "gcds.txt".

Computational Complexity Our code utilizes a fast arithmetic library for operations on large integers (we will describe this library in details in the next section). For an input with k n-bit integers, this algorithm's computational complexity is $O(kn \log k \log(kn) \log\log(kn)$ for the product and remainder trees and $O(kn(\log n)^2 \log\log n$ for calculating GCDs.[HDWH12]

3.4 Optimizations

GMP We started out using Go's "math/big" library for big number operations in our code. However, this library is comparatively slow to the GNU Multiple Precision Arithmetic library (GMP) in C. We found an implementation of GMP in Go[1], which provided methods that were almost one-to-one substitutions of the operations in "math/big". The GMP library was thus very easy to integrate into our code, and we saw significant improvements in our runtime compared to when we used "math/big". We ran into problems because the big integer datatype in

[1] Repository for this library can be found at `https://github.com/ncw/gmp`

the GMP Go library is not copy-safe, but this problem was remedied by changing how some of our datatypes were implemented. This issue has been reported to the author of the Go library and is in the process of being patched.

Multi-Processing One important feature of Go is concurrency. Using goroutines and channels, many operations can be performed concurrently. Our initial thought was to take advantage of concurrency and generate a goroutine for every RSA key multiplication in the product tree. This way, all the multiplications will be performed concurrently, and same goes for the modulo operations in the remainder tree. But after we successfully converted to multi-processing, we found that the runtime reduced by only a small amount. After careful research, we realized that our implementation with multiple goroutines only ensured concurrency, not parallelism. Parallelism occurs when operations are running at the same time, but it is limited by the number of CPUs a machine has. Concurrency here ensures that if parts of a program are running on different threads, then the order of execution does not affect the final result. Since the most recent version of Go already sets the number of CPUs to use to be the number of CPUs on the machine, adding goroutines increases the number of threads, but cannot make all of them run in parallel. Therefore, we stopped using goroutines since we saw no improvements to our runtime.

4 Challenges

4.1 Fast Big Number Arithmetic

The implementation we referenced in Heninger et al.'s paper used C's GNU Multiple Precision Arithmetic library (GMP) to do arithmetic on large integers very quickly. Because Go does not have a standard GMP library, we started out using the "math/big" library to perform these operations. Since C's GMP library has been developed over many years, it is optimized to be very precise and fast. Fortunately, we found an implementation of the GMP library in Go, which significantly improved our runtime, even though the Go library is still undergoing development.

4.2 Storage/Processing Power

Implementations of the bulk GCD algorithms in literature often use high processing power machines with parallel computing architectures. Since we write to disk multiple times during our algorithm's implementations, we quickly run out of disk space on our own machines. We have taken the TA's advice in using Google Cloud Platform[2] to acquire some computing power and storage, but the free trial only had access to a limited set of features (unfortunately, this feature

[2] For more information about Google Cloud Platform, see `https://cloud.google.com/`

set excluded GPUs), and we were not able to acquire enough storage to run the algorithm on the full dataset. We also did not have the hundreds of dollars needed for a decent machine with GPUs.

4.3 C Implementation

Heninger et al.'s paper provided a C implementation of their bulk GCD algorithm, also based on Bernstein's GCD algorithm. They used an older version of GMP (ver 5.0.5) in addition to patched versions of GMP methods, which differs from our Go implementation because they were working with the native C code. We wanted to compare the performance of our Go implementation with their version. However, due to their GMP patch, we ran into many problems when building their code and could not compare their runtimes with ours. We contacted the paper authors about this issue, however, they reiterated that they are not providing technical support for their code.

5 Algorithm Performance and Contributions

5.1 Performance

The runtime analysis on our implementation running on multiple datasets is shown in Table 1. In almost all cases, we performed significantly better than the naive pairwise GCD algorithm for bulk GCD.

Time for Bulk GCD Algorithms to Run to Complete			
	Pair-wise GCD using GMP	Bernstein's Bulk GCD with math/big	Bernstein's Bulk GCD with GMP
200 128-bit moduli	20ms	974ms	291ms
10000 128-bit moduli	50s	4m59s	14s
5000 2048-bit moduli	3m14s	5m1s	15s
100000 2048-bit moduli	41h40m	26h	10m05s

Table 1: Runtime comparison for bulk GCD algorithms on varying numbers of moduli and moduli sizes.

The runtime for pair-wise GCD with GMP was estimated from data in the GMP documentation and work done by Granlund et al.[Gra]. Since pairwise GCD is calculated assuming an ideal machine, the runtime seems fast for a naive implementation compared to Bernstein's algorithm. However, the naive implementation does not scale well and Bernstein's GCD algorithm performs significantly better with larger inputs. We also include comparison with our initial implementation with Go's "math/big" library, to emphasize how our implementation was significantly optimized after switching to GMP.

Due to the limit on GMP's maximum integer size (2^{31} bytes $\approx$ 2.15 GB), the algorithm cannot handle an input file with size larger than 500 MB, otherwise, the product of all RSA moduli exceeds 1 GB, and to perform the remainder tree, GMP needs to allocate more than 2 GB of memory. Thus, we divided the 4.3 million RSA moduli into eight files each with 500,000 moduli and a ninth file with 315,000 moduli. Meanwhile, because we separated the keys into different files, all the weak keys we extracted are those that share prime within a file, but not across all input RSA moduli. This is why the number of weak RSA keys we found is significantly smaller than the previous research (85,988 weak RSA keys among 11,170,883 distinct RSA moduli) [HDWH12].

Running our algorithm on the nine moduli files (with 4.3 million RSA moduli in total) took approximately 10 hours on a 4-core machine with a 2.60 GHz Inter Core i7 processor and 16 GB of RAM, and we were able to extract 819 weak factorable RSA keys.

5.2 Key-Check Service

Using the 819 weak factorable RSA keys, we created a key-check service. Our key-check service takes in a RSA modulus and its encoding (e.g hex or decimal). It checks the provided key against our set of known factorable keys by computing pairwise GCDs. However, due to the limited size of the collected factorable keys, we cannot guarantee that a provided key is safe, but we can immediately identify keys that share a prime with the factorable keys we collected.

6 Future Work

6.1 Patching Go's GMP Library

If we were to eventually read in all 4.3 million (and possibly more) RSA keys at the same time, we would inevitably run into GMP's maximum size limit for it's raw-integer I/O format (2^{31} bytes). We would therefore have to patch the library to handle larger numbers.

Once we patch the GMP library, the algorithm should be able to handle input file of any size. Running the algorithm on 4.3 million (and possibly more) RSA keys at the same time will generate a lot more weak keys, with which the key-check service will achieve a better performance at recognizing weak input keys.

6.2 Increasing Computation Power

Running the code on our local machines on the millions of RSA keys was inefficient and costly in terms of storage and CPU power. Using the free trial of Google Cloud Platform also was not as fruitful as anticipated. In the future, with enough resources, we would like to optimize the performance of our algorithm on even larger size datasets by utilizing GPUs, increasing the number of cores used, and other general improvements to computation power. We will also need adequate storage space needed for all of our generated files if we were to run to the full dataset.

7 Conclusion

We implemented a fast bulk GCD computation algorithm for RSA keys in Go and generated a set of known weak RSA keys. With our key-check service, we are able to identify weak RSA keys at request time by computing pairwise GCDs with our set of known weak RSA keys. Given more time and computation power, we hope to create a more efficient and better performing algorithm that can take in larger input files, and therefore generate more weak RSA keys.

8 Appendix

8.1 Repository

Our code can be found at `https://github.com/eyxue/fastgcd`

Acknowledgements

Our project is suggested by Let's Encrypt in order to decrease the likelihood of weak RSA keys to encrypt potentially sensitive information. We would like to thank Prof. Ronald L. Rivest, Prof. Yael Kalai, and the 6.857 TAs for their insights that influenced the design and implementation of this project. We would also like to thank Prof. Nadia Heninger for her guidance on this project.

References

Ber08. Daniel J. Bernstein. Fast multiplication and its applications, 2008.

FNI15. T. Fujita, K. Nakano, and Y. Ito. Bulk gcd computation using a gpu to break weak rsa keys. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 385–394, May 2015.

Gra. Et Al. Granlund, T. The gnu multiple precision arithmetic library.

HDWH12. Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

KAF$^+$10. Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit rsa modulus. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 333–350, Berlin, Heidelberg, 2010. Springer-Verlag.

LHA$^+$12. Arjen K. Lenstra, James P. Hughes, Maxime Augier, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. Technical report, 2012.