Constant-Time Big Numbers (for Go)

Lúcás Críostóir Meier

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab (DEDIS)

BSc Semester Project

May 2021

Responsible and Supervisor
Prof. Bryan Ford
EPFL / DEDIS

Contents

1 Introduction

With the Internet cemented as a principal keystone in communication, and ever increasing activity taking place digitially, the importance of secure communication and Cryptography has never been greater. Thankfully, after 50 years of Public Key Cryptography [4], we have good theoretical systems to provide these guarantees.

Most of these systems rely on modular arithmetic with large numbers, such as RSA or Elliptic Curve Cryptography [9, 8] . Working with such numbers is not natively supported by hardware, requiring a "Big Number" software library to provide this functionality.

Unfortunately, even though Public Key Cryptosystems have been heavily scrutinized *in theory*, in practice many vulnerabilities arise in software implementations of these systems.

One particularly pernicious class of vulnerability are **timing attacks** [6], where an implementation leaks information about secret values through its execution time or cache usage, among many side-channels.

Libraries for Big Numbers that are not designed with Cryptography in mind are pervasively vulnerable to this class of attack.

In particular, Go [1] provides a general purpose Big Number type, `big.Int`, which suffers from these vulnerabilities, as we detail later in this report. Unfortunately, this library gets used for Cryptography [2], including inside of Go's own standard library, in `go/crypto`.

We've addressed this issue by creating a library [7] designed to work with Big Numbers in the context of Public Key Cryptography. Our library provides the necessary operations for implementing these systems, all while avoiding the leakage of secret information. To demonstrate its utility, we've modified Go's `go/crypto` package, replacing the use of `big.Int` in the DSA and RSA systems.

2 Background

In this section, we explain how Big Numbers are used in Public Key Cryptography, what timing attacks are, and how they affect our threat model, as well as what kind of side-channels are present in Go's `big.Int` type.

2.1 Big Numbers in Cryptography

As mentioned previously, most Public Key Cryptosystems rely on modular arithmetic.

In RSA [9], for example, a public key (e, N) consists of modulus $N \in \mathbb{N}$, usually 2048 bits long, and an exponent taken modulo $\varphi(N)$. To encrypt a message $m \in \mathbb{Z}/N\mathbb{Z}$, we calculate

$$c := m^e \mod N$$

The typical word size on computers is now 64 bits. Because of this, to do arithmetic modulo N, we need a Big Number library to work with these large numbers, as well as to provide optimized implementations of opereaitons like modular exponentiation, which aren't natively supported.

DSA [10] also relies on modular arithmetic, this time using a large prime p of around 2048 bits, and working in the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$.

Elliptic Curve Cryptography [8] relies on complex formulas for adding points on an elliptic curve, built over a finite field K. This field is usually either a prime field $\mathbb{Z}/p\mathbb{Z}$, in which case arithmetic modulo p is used, or a binary extension field GF(2^n), in which case binary arithmetic in combination with polynomial addition and multiplication are used.

For prime fields, a Big Number library is once again necessary, because the size of the prime is greater than 200 bits.

In summary, large modular arithmetic is a cornerstone of Public Key cryptosystems, requiring Big Numbers in some form.

2.1.1 Big Numbers in `go/crypto`

Go [1] provides implementations of numerous cryptographic algorithms as part of its standard library, including the aforementioned Public Key systems, in the `go/crypto` package.

Unfortunately [2], the general purpose `big.Int` type gets used in this package for Cryptography, despite its potential vulnerability to timing attacks.

For DSA [10], Go uses `big.Int` for all operations, including key generation, signing, and verification.

For RSA [9], Go uses `big.Int` for all operations, and fixes this type as part of the API for this package. Key generation, encryption, decryption, signing, and verification all use `big.Int`.

For ECC [8], Go defines a general interface for Elliptic Curves, requiring operations like point addition, scalar multiplication, etc. All of these are defined in terms of `big.Int`. Some of the curves have specialized implementations of their prime field, only converting from `big.Int` to satisfy the interface curves. The remaining curves directly make use `big.Int` for their field arithmetic.

2.2 Timing Attacks

A side-channel [5] leaks secret information about a program not directly, but indirectly, through the observable properties of its execution. For example, timing side-channels use the execution time of a program to infer properties

of the secret data it processes. A timing attack is the use of a timing side-channel to break the security of some program or cryptographic protocol.

If a program takes a different number of steps based on the value of some secret, then this constitutes an obvious timing side-channel. For example, if a naive program for comparing inputs with a secret password stops as soon as a mismatch is found, then the algorithm itself has a timing side-channel. This side-channel can be exploited, to allow the secret password to be guessed byte by byte.

Not all timing side-channels are this simple, however. Algorithms that take the same number of steps regardless of the value of secret data can still having timing side-channels because of how the underlying hardware executing the program works. For example, a processor may execute an operation faster for some inputs, or the presence of a cache could be used to infer what addresses are being accessed. These microarchitectural timing side-channels are also of concern. See [3] for a survey of these vulnerabilities.

2.2.1 Actual Attacks

Explain how these concerns actually lead to vulnerabilities in systems.

2.2.2 Our Threat-Model

Explain the threat model we have, and what side-channels we aren't concerned about.

2.3 Vulnerabilities in `big.Int`

Explain what vulnerabilities are potentially lurking in bigInt.

2.3.1 Padding and Truncation

Explain how big.Int truncates numbers internally.

Explain potential issues with padding in cryptography.

2.3.2 Leaky Algorithms

Explain how most algorithms are potentially leaky.

2.3.3 Mitigations

Explain what mitigations are deployed in Go.

3 Implementation

Describe at a high level what we've done.

3.1 Strategies for numbers in Cryptography

Describe different strategies in place for providing numbers for Cryptography.

3.2 The `safenum` library

Describe at a high level what the library provides.

3.2.1 Handling Size

Describe how we handle sizing of numbers.

3.3 Some Basic Techniques

Describe some basic techniques for constant-time operation.

3.4 Some Algorithm Choices

Describe the algorithm choices we've used for different things.

4 Results

Describe what results we've managed to perform.

4.1 Comparison with `big.Int`

Describe the final performance results we've managed to achieve.

4.2 Comparison with `go/crypto`

Describe the benchmarks on actual code.

5 Further Work

5.1 Upstreaming to `go/crypto`

Describe our work in providing a patch for RSA, and what results we've managed to achieve.

6 Conclusion

Summarize the things we put in the introduction.

Acknowledgements

References

[1] The Go Authors. The Go Programming Language Specification - The Go Programming Language. URL: https://golang.org/ref/spec.

[2] Bryan Ford. proposal: math/big: support for constant-time arithmetic - Issue #20654 - golang/go, 2017. URL: https://github.com/golang/go/issues/20654.

[3] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018. URL: http://link.springer.com/10.1007/s13389-016-0141-6, doi:10.1007/s13389-016-0141-6.

[4] Martin E Hellman. An Overview of Public Key Cryptography. *IEEE COMMUNICATIONS SOCIETY MAGAZINE*, page 9, 1978.

[5] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security ESORICS 98*, Lecture Notes in Computer Science, pages 97–110, Berlin, Heidelberg, 1998. Springer. doi:10.1007/BFb0055858.

[6] Paul C Kocher. Cryptanalysis of Di e-Hellman, RSA, DSS, and Other Systems Using Timing Attacks. page 6, 1995.

[7] Lúcás Críostóir Meier. cronokirby/safenum, May 2021. URL: https://github.com/cronokirby/safenum.

[8] Victor S. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology CRYPTO 85 Proceedings*, Lecture Notes in Computer Science, pages 417–426, Berlin, Heidelberg, 1986. Springer. doi:10.1007/3-540-39799-X_31.

[9] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. doi:10.1145/359340.359342.

[10] National Institute of Standards and Technology. Digital Signature Standard (DSS). Technical Report Federal Information Processing Standard (FIPS) 186 (Withdrawn), U.S. Department of Commerce, May 1994. URL: https://csrc.nist.gov/publications/detail/fips/186/archive/1994-05-19, doi:10.6028/NIST.FIPS.186.