# 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 [13], 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 [24, 20] . 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** [15], 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 [3] 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 [11], including inside of Go's own standard library, in `go/crypto`.

We've addressed this issue by creating a library [18] 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 [24], 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 [25] 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 [20] 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 Implementing Big Numbers

Describe different strategies in place for providing numbers for Cryptography.

### 2.1.2 Big Numbers in `go/crypto`

Go [3] provides implementations of numerous cryptographic algorithms as part of its standard library, including the aforementioned Public Key systems, in the `go/crypto` package.

Unfortunately [11], the general purpose `big.Int` type gets used in this package for Cryptography, despite its potential vulnerability to timing attacks.

For DSA [25], Go uses `big.Int` for all operations, including key generation, signing, and verification.

For RSA [24], 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 [20], 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 [14] 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 [12] for a survey of these vulnerabilities.

### 2.2.1 Actual Attacks

The presence of a side-channel does not directly lead to vulnerabilities. As early as 1995, Paul Kocher demonstrated the potential for timing attacks against cryptographic algorithms [15, 16]. These specific attacks rely on algorithms that perform a varying number of operations based on secret data.

One common objection to timing attacks more generally is that they while a timing side-channel is catastrophic in theory, in practice this channel is too noisy to exploit. Unfortunately, through gathering many samples, it's possible to exploit these attacks even across a network [8, 7].

The use of caches as a potential side-channel was identified early on as well [22]. The idea is that accessing data that is not present in the cache takes longer than accessing data inside of the cache. If what data is being accessed depends on a secret value, the observed execution time will thus also depend on this secret value. If an attacker is located on the same machine they can place data into the cache as well, and probe the cache themselves, to learn fine-grained information about the program's access patterns. While seemingly far-fetched, this is easy to achieve now that so many applications are run on cloud computing.

A wide variety of attacks involving caches have been mounted against

various cryptosystems [4, 26, 9] , making accessing data based on secret values fraught with peril.

### 2.2.2 Our Threat-Model

Although the variety of potential timing side-channels is quite daunting, we can distill them into a simple, albeit pessimistic set of rules:

1. Any loop leaks the number of iterations taken.
2. Any memory access leaks the address accessed.
    (a) As a consequence, accessing an array leaks the index accessed.
3. Any conditional statement leaks which branch was taken.

Rule 1 is justified by theoretical concerns, since a longer loop requires more operations. In practice, it's difficult to observe the iterations of each loop in a program from an overall timing signal, making this a pessimistic rule.

Rule 2 is justified by various cache based side-channels and attacks [4, 26, 9]. Since caches only load information an entire line at a time, it might seem that our rule is too pessimistic, and that only which cache line was accessed should be kept secret [6]. Unfortunately, the potential for attacks on much finer grained level has been demonstrated [5, 21, 26]. Because of these concerns, we take a pessimistic position, and assume that accessing an array leaks the exact index accessed.

Rule 3 is justified in two ways. First, if different branches of a conditional statement execute a different number of operations, this leaks information about which branch was taken in a fundamental way. Second, even if both branches execute identitical operations, the CPU's branch predictor can be exploited, leaking information about which branch was taken [2, 1, 10].

In addition to these rules, we assume that addition, multiplication, logical operations, and shifts, as implemented in hardware, are constant-time in their inputs. This is the case on most processors, one notable exception being microprocessors [23]. For the platforms which Go, and thus our library, targets, this assumption is reasonable.

## 2.3 Vulnerabilities in `big.Int`

Go provides a general purpose type for Big Numbers: `big.Int`. This implementation is concerned with being broadly applicable, and well-optimized. It does not focus on security, or on hardening itself against timing side-channels.

Unfortunately, its broad applicability makes it useful for cryptography, and it gets used throughout Go's standard cryptography library, as we've seen previously.

In this section, we look at some of the important implementation aspects of `big.Int`, and how they might be potentially vulnerable according to our threat model.

### 2.3.1 Padding

The `big.Int` type always normalizes numbers internally, removing any leading zero limbs. Even if you initialize a number using bytes zero-padded to a certain length, the resulting value will immediately chop off these zeros. These extra zeros don't change the value of a number. By discarding them, operations on this number will have fewer limbs to process, and will be faster.

Unfortunatelly, this means that `big.Int` leaks information about the padding of numbers pervasively. Since every operation on a number takes more time the more limbs the number uses, the removal of padding leaks the true sizes of numbers at every operation. Leaking this padding has been exploited in OpenSSL [19], and might potentially be a vulnerability in Go's cryptography library, because of `big.Int`.

### 2.3.2 Leaky Algorithms

Because `big.Int` is not written with Cryptography in mind, its methods violate the rules set in 2.2.2. Many methods take a different number of iterations based on their values, branch conditionally on values, and access memory depending on values. Because `big.Int` is designed for general purpose use, this problem should only get worse as the library is further developed and optimized.

Ultimately, the problem is not the existence of `big.Int`, but it's use in Go's cryptography library, and in the broader ecosystem.

### 2.3.3 Mitigations

Although `big.Int` gets used in Go's cryptography library, the authors are aware of its shortcomings, and have implemented several mitigations to try and make its timing side-channels harder to exploit.

One of the most important ones is a mitigation for RSA: blinding [16]. The decrypt a ciphertext $c = m^e \mod N$, we normally calculate:

$$c^d \mod N$$

with $d$ our private key, and $(e, N)$ our public key. When exponentiation is not implemented in a constant-time way, like with `big.Int`, this process

can leak information about $m$. If an adversary can choose $c$, then this can leak information about $d$ as well.

To mitigate this, instead of decrypting $c$ directly, we first generate a random integer $r \in [0, N-1]$, and make sure it has an inverse $r^{-1} \bmod N$. Then we decrypt $r^e \cdot c$. This gives us the value $r \cdot m$, and we can recover $m$ by multiplying by $r^{-1}$.

While this does effectively mitigate the simplest attacks against exponentiation, a very leaky operation, the other operations involving these values are still left unprotected, and may have exploitable leakages in more subtle ways. We also have the unaddressed issue of padding, which has lead to attacks in OpenSSL [19].

# 3 Implementation

We've implemented a library, called `safenum` [18], intended to provide a replacement for `big.Int`, suitable for use in Cryptography. In order to demonstrate its utility, we've replaced some of `go/crypto`'s usage of `big.Int` with our own library, in a separate repository [17].

In this section, we go over the design and implementation of our library.

## 3.1 The `safenum` library

Describe at a high level what the library provides.

### 3.1.1 Handling Size

Describe how we handle sizing of numbers.

### 3.1.2 Moduli

Describe how moduli are handled

## 3.2 Limbs

## 3.3 Constant-Time Operations

Describe some basic techniques for constant-time operation.

## 3.4 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] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, ASIACCS '07, pages 312–320, New York, NY, USA, March 2007. Association for Computing Machinery. doi:10.1145/1229285.1266999.

[2] Onur Aciçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, Lecture Notes in Computer Science, pages 225–242, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11967668_15.

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

[4] Daniel J Bernstein. Cache-timing attacks on AES. page 37, 2005.

[5] Daniel J Bernstein and Peter Schwabe. A word of warning. In *Workshop on Cryptographic Hardware and Embedded Systems*, volume 13, 2013.

[6] Ernie Brickell. Technologies to improve platform security. In *Workshop on Cryptographic Hardware and Embedded Systems*, volume 11, 2011.

[7] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security ESORICS 2011*, Lecture Notes in Computer Science, pages 355–371, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-23822-2_20.

[8] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, August 2005. URL: http://www.sciencedirect.com/science/article/pii/S1389128605000125, doi:10.1016/j.comnet.2005.01.010.

[9] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 213–242, August 2019. URL: https://tches.iacr.org/index.php/TCHES/article/view/8350, doi:10.46586/tches.v2019.i4.213-242.

[10] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, October 2016. doi:10.1109/MICRO.2016.7783743.

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

[12] 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.

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

[14] 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.

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

[16] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. page 10, 1996.

[17] Lúcás Críostóir Meier. cronokirby/ctcrypto, May 2021. original-date: 2021-04-17T14:26:20Z. URL: https://github.com/cronokirby/ctcrypto.

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

[19] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, and Johannes Mittmann. Raccoon Attack: Finding and Exploiting Most-Signicant-Bit-Oracles in. page 19, 2019.

[20] 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.

[21] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *Topics in Cryptology CT-RSA 2006*, Lecture Notes in Computer Science, pages 1–20, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11605805_1.

[22] D. Page. *Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel*. 2002.

[23] Thomas Pornin. BearSSL - Constant-Time Mul. URL: https://www.bearssl.org/ctmul.html.

[24] 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.

[25] 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.

[26] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. page 21, 2017.