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 [28, 21] . 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** [16], 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 [19] 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 [28], 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 [29] 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 [21] 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 $\text{GF}(2^n)$, 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

When you know the modulus in advance, like for Elliptic Curve Cryptography, where prime modulus is part of the system itself, then you can implement a library for doing arithmetic with that specific modulus. By using a fixed size type, it's easier to provide constant-time operation as well.

The downside is that since different systems require different moduli, a lot of effort has to be expended to implement and support these different systems.

One way to address this is to automatically generate implementations of modular arithmetic, as done by FiatCrypto [14].

In some systems, you need support for dynamic moduli. For example, in RSA. In this case, you do need a Big Number libary of some kind, providing dynamically sized numbers, and a myriad of operations.

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

For RSA [28], 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 [21], 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 [15] 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 [16, 17]. 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 [23]. 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, 31, 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, 31, 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, 22, 31]. 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 [24]. 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 [20], 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 [17]. 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 [20].

3 Implementation

We've implemented a library, called `safenum` [19], 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 [18].

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

3.1 The `safenum` library

Safenum defines a `Nat` type, which is intended to replace `big.Int`. This type represents arbitrary numbers in $\mathbb{N}$. Unlike `big.Int`, we do not handle negative numbers. Handling a sign bit in constant-time is exceedingly tricky. Thankfully, we haven't found this limitation to be restrictive when replacing `big.Int` in Go's cryptography library.

We represent numbers in base $W := 2^{64}$. Concretely, we store a number as a slice of type `[]uint`, in little endian order. We call these the "limbs" of a number. For example, the slice:

```
[]uint{13, 47, 52}
```

represents the number:

$$52 \cdot 2^{128} + 47 \cdot 2^{64} + 13$$

These limbs might be padded, to conceal the true value of a number, as we'll see later.

We provide operations for addition and multiplication of `Nat`s, all other operations are for modular arithmetic. For modular arithmetic, we provide numerous operations, including modular addition, subtraction, multiplication, exponentiation, inversion, reduction, and taking square roots modulo prime numbers. We also provide the usual operations for serializing to and from bytes.

We try and structure the API in a similar way to `big.Int`, where an operation is performed on a buffer `Nat`, which can receive the result. For example, this is the signature for modular addition:

```
func (z *Nat) ModAdd(x *Nat, y *Nat, m *Modulus) *Nat
```

This calculates $z \leftarrow x + y \mod m$, returning z. The advantage of structuring the API this way, instead of simply returning a new value, is that we can reuse the memory of z for the result.

We go one step further, in fact, and use the memory of the buffer `Nat` for all scratch space needed inside of an operation. Structuring our operations this way allows us to limit unnecessary waste of allocation.

3.1.1 Handling Size

Unlike `big.Int`, a `Nat` doesn't truncate its limbs to remove any zero padding. Because of this, we distinguish between the *true size* of a number, which is how many significant bits or limbs it actually has, and the *announced size* of a number, which is how many limbs are actually used to store that number. The announced size is allowed to be leaked, while the true size should be kept secret. The true size is always at most the announced size

Because of this, we need to ensure that there's always a clear announced size to use for the results that we produced. For modular operations, we have an obvious choice: the size of the modulus. When doing a modular operation, the result will always receive the same announced size as the modulus does.

For example, when doing modular addition:

```go
func (z *Nat) ModAdd(x *Nat, y *Nat, m *Modulus) *Nat
```

our result `z` will have the same announced size as `m`. After modular addition, we have that $z \in [0, m-1]$ by definition. Because of this, leaking the fact that the true size of z is at most that of m leaks no information about what value z actually has, beyond what's knowable just by inspecting the call graph in the source code of a program.

When serializing a `Nat`, we respect its announced size, and produce zeros for padding as necessary. This is done without any special handling, because we already store padded limbs anyways.

Similarly, when deserializing a `Nat` from bytes, we respect any padding, unlike `big.Int`. For example, if 32 big endian bytes are deserialized, we will end up with a `Nat` with an announced size of 256 bits, regardless of the value of those bytes.

This leaves us with non-modular addition and multiplication of numbers. One approach is to use the maximum possible resulting size for our result's announced length. For example, if we multiply numbers x_1 and x_2, of announced size b_1 and b_2, then our result will need a size of at most $b_1 + b_2$. In situations where we know that our result will be smaller, this size explosion can be undesirable.

Because of this, we opt towards letting users specify exactly how many resulting bits they need in the output. For example, multiplication has the following signature:

```go
func (z *Nat) Mul(x *Nat, y *Nat, cap uint) *Nat
```

Here `cap` is the number of bits that the result should have. We use this to determine the result's announced length. Any output beyond that capacity will simply be discarded.

In summary, the announced size of a `Nat` is always clear based on how it's produced, and results from deserializing a value, from using the same size as a modulus, or from manually deciding on an output size.

3.1.2 Moduli

In our library, we've decided to make a separate type for representing the moduli used in modular arithmetic: `Modulus`. There are several reasons for doing this.

First, various operations in modular arithmetic require different properties of the modulus which can be pre-computed. For example, montgomery multiplication requires us to know $m^{-1} \mod W$, with W our base, and m our modulus. By using a separate type for moduli, we can precompute these values.

Second, the true size of a modulus is considered to be leakable. As a consequence, moduli are stored *without* padding. This is desirable because modular reduction needs access to the most significant bits of a modulus, and fetching this information without leaking padding is exceedingly difficult. Furthermore, by storing moduli without padding, the announced size of numbers produced through modular operations is as tight as possible, which speeds up operation.

This assumption is safe in cryptography. Moduli are often public, like with the public modulus N in RSA. In this case, leaking the true size is fine, since even the exact value is known. There are some cases where a secret modulus is necessary. For example, when generating an RSA key, we use the factorization $N = pq$ of the modulus, and calculate our private key modulo $\varphi(N) := (p-1)(q-1)$:

$$d := e^{-1} \mod \varphi(N)$$

Leaking the value of $\varphi(N)$ would be catastrophic. On the other hand, it's clear that the true size of $\varphi(N)$ is approximately that of N, which is known. In this case, leaking the true size of $\varphi(N)$ is fine.

Using a separate modulus type is necessary to have this weaker constraint on its announced size.

3.2 Constant-Time Operations

The rules we established in our threat model 2.2.2 are quite stringent. For many operations, we want to have conditional behavior depending on the values and results we see. Without access to branching, this would seem impossible. Thankfully, there are workarounds to enable us to have conditional behavior, all while not leaking information about which conditions are selected. The core idea here is that whenever we're faced with a choice, we perform both branches, and then combine the results together without revealing which result we end up using.

For example, an standard algorithm for modular subtraction would look like this (in pseudo-Go):

```go
func (z *Nat) ModSub(x *Nat, y *Nat, m *Modulus) *Nat {
  borrow := z.Sub(x, y)
  if borrow == 1 {
    z.Add(z, m)
  }
}
```

The problem here is that by conditionally adding in m, we reveal whether or not $y > x$, and a borrow occurred. Our solution requires a new primitive:

```go
func (z *Nat) ctCondCopy(v choice, y *Nat) *Nat
```

This function assigns y to z if $v = 1$, and does nothing otherwise. Furthermore, this primitive shouldn't leak any information about whether or not the condition was true.

With this in place, we can implement modular subtraction without leakage:

```go
func (z *Nat) ModSub(x *Nat, y *Nat, m *Modulus) *Nat {
  borrow := z.Sub(x, y)
  scratch := new(Nat).Add(z, m)
  z.ctCondCopy(choice(borrow), scratch)
}
```

We always perform the addition, and copy over the result if necessary, without leaking the value of `borrow`.

This kind of rearrangement is the foundation that allows us to replace branching in algorithms with constant-time operations.

3.2.1 Building Primitives

The question remains: how do you build up the primitives like `ctCondCopy`, which let you choose results without leaking which result was chosen?

The methods for constant-time choice are analagous to the standard programming methods for conditional branching. Instead of using `bool` to represent the result of a condition, we use

```
type choice Word
```

The value of a choice is either 1 or 0, but we use the same type as full limbs, to try and avoid having the compiler turning or manipulations back into branches.

From this choice value, we can build a primitive that selects between two limbs without leaking which choice was selected:

```
func ctIfElse(v choice, x, y Word) Word {
  mask := -Word(v)
  return y ^ (mask & (y ^ x))
}
```

This routine returns x if $v = 1$, and y otherwise. Unlike a conditional statement, this is implemented through bitwise operations, and doesn't leak information about what was selected.

If $v = 0$, then `mask` contains only zeros, and we're left with `y`. When `v == 1`, then `mask` only contains ones. The `y`'s cancel eachother out, leaving us with `x`.

We can use this primitive to build up a larger selection primitive, allowing us to assign one slice to another, conditionally:

```
func ctCondCopy(v choice, x, y []Word) {
  for i := 0; i < len(x); i++ {
    x[i] = ctIfElse(v, y[i], x[i])
  }
}
```

These primitives allow us to use `choice` to introduce conditional behavior without leaking our choices, but we also need primitives to create `choice` values in the first place. These are built up in a similar way from small primitives to large primitives.

We can decide whether or not two limbs are equal using some bitwise trickery:

```
func ctEq(x, y Word) choice {
  q := uint64(x ^ y)
  return 1 ^ choice((q|-q)>>63)
}
```

To understand why this trick works, first realize that in two's complement, either the most significant bit of a number is set, or the most significant bit of a number's negation is set, unless that number is zero. Thus, the expression

```
choice((q|-q)>>63)
```

simply checks, using only bitwise operations, that `q` is not zero. Since `q` is zero precisely when `x` and `y` are equal, we can simply negate the non-zero check.

We can use this primitive to compare entire slices of limbs in constant time:

```
func cmpEq(x []Word, y []Word) choice {
  res := choice(1)
  for i := 0; i < len(x) && i < len(y); i++ {
    res &= ctEq(x[i], y[i])
  }
  return res
}
```

We can't exit early, since that would leak information about the value of x and y. Instead, we combine all the results together, using the fact that two slices are equal when every one of their matching limbs are equal.

3.3 Algorithm Choices

While going over how each operation works in detail is outside the scope of this report, describing some of the high-level techniques for certain trickier operations is nonetheless interesting.

Many of operations were inspired by the excellent work of Thomas Pornin in BearSSL [25].

3.3.1 Modular Reduction

To reduce a number a modulo m, we first implement an operation that allows us to shift in a single limb. This lets us reduce a number of the form:

$$z := a \cdot W + b$$

with $a \in [0, m-1]$ and $b \in [0, W-1]$. With this in place, we can reduce an arbitrary number, by shifting in each of its limbs, from most significant, to least significant, reducing modulo m each time.

Implementing this shifting operation is a bit tricky. In the case that m has only a single limb, we can reduce $z = z_1 W + z_0$ by dividing the 128 bit number $z1 : z0$ by m, and then using the remainder as our result. We implement this division using bitwise operations, because Go's equivalent `bits.Div` operation is not constant-time.

In the case that m has more than one limb, we want to estimate the quotient $q := \lfloor z/m \rfloor$, and then calculate $z - q \cdot m$, to get our remainder. A first estimate divides the most significant two limbs of z with the most significant limb of m, using the division operation we've already established. To improve this estimate, we can take the most significant 64 bits of m, ignoring the leading zero bits in m's top limb, and align our 128 bits of z accordingly. It's because of this need to skip past m's leading zeros that we'd like for moduli to not have padding. This gives us an estimate $\hat{q} = q \pm 1$. We can then calculate $z - \hat{q} \cdot m$, and then either add or subtract m based on the result. In practice, this means always adding and subtracting m, and then mixing in the right result in constant-time based on flags.

This technique is further described in [25].

3.3.2 Multiplication

Two multiply two numbers a, and b, we can expand a over its limbs $a_n W^n + \cdots + a_0$, and then calculate the sum:

$$\sum_{i=0}^{n} a_n \cdot b \cdot W^i$$

We can implement the multiplication by W^i by each small multiplication to a different position in our result. Our result fits over double the number of limbs, and can then be truncated or expanded to the desired bitsize.

For modular multiplication, we can calculate this result, and then reduce it modulo m. There is also a technique, called Montgomery Multiplication [**?**], which allows us to calculate a slightly different result faster.

Let's say that a and b are both reduced mod m, which has n limbs. The small multiplication $a_i b$ fits over $n+1$ limbs. To reduce this down to n limbs, we could do a single reduction step, but estimating a quotient is expensive. Instead, we can multiply m by a fudge factor f, in such a way that $a_i b + fm$ has its lowest limb cleared. Then, we can simply shift this result right by one limb, obtaining:

$$\frac{a_i b}{W} \mod m$$

The right value for this fudge factor turns out to be:

$$-m_0^{-1} a_i b_0 \mod W$$

Because of the need to invert the least significant limb modulo our basis, which is a power of 2, we need m to be an odd number.

If we calculate the sum:

$$\sum_{i=0}^{n-1} \frac{a_i b}{W} \mod m$$

then our final result is just:

$$\frac{(a_{n-1}W^{n-1} + \cdots + a_o)b}{W^n} = \frac{ab}{W^n} \mod m$$

This isn't exactly the product $ab \mod m$, that we wanted, but is much faster to calculate.

If prior to this, we calculate $aW^n \mod m$, then performing Montgomery multiplication with b will yield the result $ab \mod m$. This is our approach to defining a modular multiplication. $aW^n \mod m$ is called the Montgomery representation of a. We can calculate this representation by using the modular shifting operation we defined for reduction. If we have many multiplications to perform, like in exponentiation, the cost of converting to this representation is amortized over the number of multiplications we do, yielding significant savings.

3.3.3 Exponentiation

For exponentiation, we use left-to-right exponentiation, with a window size of 4 bits. In order to do window lookups in constant-time, we do a constant-time conditional assignment over each of the possible operands. Even though this requires traversing the entire window table every time, we've found that this method is still faster than using a window size of 2 bits.

3.3.4 Inversion

For modular inversion, we use a variant of the binary GCD algorithm, as described in [26]. We have yet to implement the most optimized version, which accumulates intermediate results into single limb registers, and instead perform full width operations at each iteration.

3.3.5 Even Inversion

To calculate $x^{-1} \mod m$, when m is an even integer, we employ a standard trick. First, we calculate $u := m^{-1} \mod x$, and then we calculate our desired inverse as:

$$\frac{um - 1}{x}$$

3.3.6 Modular Square Roots

To calculate $\sqrt{z} \mod p$, we assume that p is prime. Which algorithm we use depends on whether $p = 3 \mod 4$, or $p = 1 \mod 4$.

In the first case, we can calculate a root as:

$$\sqrt{z} = z^{(p+1)/4} \mod p$$

In the second case, we use a constant-time variant of the Tonnelli-Shanks algorithm, as described in [30].

3.4 Implementation Techniques

In this section, we describe a few remaining implementation choices and techniques of interest.

3.4.1 Saturated or Unsatured Limbs

As mentioned previously, we store numbers in base $W := 2^{64}$. This means that we use the full width of a register to store each limb. Because of this, we say that are limbs are *saturated*. It's also possible to store limbs *unsaturated*, by using fewer than 64 bits. BearSSL [25] takes this approach, using only 31 bits of the available 32 bits in the integers it uses. For 64 bit registers, using 63 bit unsaturated limbs would be the option of choice.

There are two compelling reasons for using unsaturated limbs.

The first is that this leaves an extra bit of space to hold a carry or borrow after an addition or subtraction, respectively. This allows us to chain together carries to implement operations over multiple limbs, without having to use assembly instructions. In Go, this isn't really an issue, because `bits.Add` and `bits.Sub` are provided to implement this intrinsics in a cross-platform way.

The second comes that if we use w bits for each limb, then montgomery multiplication needs to work with a value of size $2w + 1$ bits. With a fully saturated limb of 64 bits, we need 129 bits. This uses an extra register compared to unsaturated limbs of 63 bits. Because montgomery multiplication is called very often during exponentiation, this can yield considerable savings.

The disadvantage of using unsaturated limbs comes when converting numbers to and from bytes. With fully saturated limbs, our 64 bit limbs are nicely composed of 8 bytes. With 63 bit limbs, this isn't the case, making conversion to and from bytes more complicated, and expensive. Using unsaturated limbs would also require storing additional information about the exact announced size of a number, instead of being able to use the number of limbs directly.

Ultimately, we opted to use saturated limbs, in order to reuse the assembly routines already implemented for low level operations in `big.Int`. These were designed with saturated limbs in mind, and thankfully, are constant-time.

3.4.2 Redundant Reductions

Our library is defined to prevent misuse. Because of this, modular operations work even if their inputs are not already reduced. For example, addition modulo m should return the right result, even if the inputs are greater than m. Unfortunately, the cost of reducing inputs modulo m when they were already in range is not desirable, since this operation is relatively expensive. Ideally, we'd like to avoid reducing inputs when we already know that they're correctly reduced.

To implement this, each number stores a pointer to a modulus, indicating that it is already reduced by this modulus. When we reduce a number modulo m, we check this pointer, and skip this reduction if it matches m. If we modify the value of a number, we update the modulus it points to accordingly, based on what method was called. For example:

```
z.ModAdd(x, y, m)
```

will set `z`'s modulus to `m`.

We only set this modulus pointer based on what methods are called, never on the actual value of a result. Because of this, the dynamic checks of this pointer only depend on the callgraph of our program. Since this graph is statically determined, these redundant reduction checks don't impact the constant-time properties of our library.

4 Results

We've compared the performance of our library with `big.Int` operation by operation, as well as in the context of the `go/crypto` package. Overall, our library is about 2.6x slower than using `big.Int` for most operations, but only 2x slower in realistic situations. In this section, we present these results in detail.

4.1 Comparison with `big.Int`

We've set up a series of benchmarks to compare the performance of `Nat` compared to `big.Int` on various operations.

The following operations are all implemented on values, exponents, and moduli of 2048 bits. For raw addition and multiplication, we use the full size necessary to represent the result in our benchmarks.

Operation	op / s (`big.Int`)	op / s (`Nat`)	ratio
Addition	7,811,989	10,622,287	0.74
Multiplication	1,282,808	434,802	2.95
Modular Addition	9,088,922	3,121,003	2.91

The most expensive operation, by far, is exponentiation. Because of this, it's fair to compare the performance on these two types mainly on this operation. We can see that `Nat` is 2.6x compared to `big.Int` for exponentiation, although varies in speed for other operations.

For comparing modular square roots, we used the primes $p_3 = 2^{244} + 79$, which is $3 \mod 4$, and $p_1 = 2^{244} + 153$ which is $1 \mod 4$. We use different primes to test the various codepaths for modular square roots.

4.2 Comparison with `go/crypto`

We've created a forked package [18] of `go/crypto`, where we've replaced the usage of `big.Int` with our own `Nat` type for both RSA and DSA. All of the code using `big.Int` has been replaced, with the exception of primality checking. This endeavour demonstrates the utility of our package for writing cryptographic code.

We've also run benchmarks to assess the performance impact, as we show in the following table:

Operation	op / s (`big.Int`)	op / s (`Nat`)	ratio
RSA Decrypt	670	312	2.15
RSA Sign	675	372	1.81
RSA Decrypt (3 Prime)	1173	596	1.97
DSA Sign	6202	2625	2.36
DSA Parameters	0.89	1.64	0.54

We use a 2048 bit modulus for both RSA and DSA. For RSA, we use the CRT optimization, instead of use exponentiation directly. Our benchmarks for `big.Int` use blinding, but our benchmarks for `Nat` do not. Because `Nat` is constant-time, we don't need to use blinding to mitigate timing attacks. We don't include DSA verification, since this can be safely done with `big.Int`.

Overall, we can see that in a real world scenario, the use of `Nat` is only 2x slower. This can surely be improved, but is already an encouraging result.

5 Further Work

While we're happy with the utility of our library, and the performance results we've managed to achieve, it's of course still possible to improve on this front.

5.1 Verifying Constant-Time Properties

Ultimately, we would like to have more assurance about the constant-time properties of our library. Our code hasn't undergone an audit, nor have we verified the assembly output produced by the Go compiler to ensure that it meets our demands.

Ideally, it would be nice to incorporate some kind of automated analysis of our code to detect timing side-channels. An approach similar to dudect [27] might be an interesting way to provide a form of fuzz testing to detect unwanted time-variation.

5.2 Optimizing Assembly Routines

Currently, we rely on some assembly routines pulled from `big.Int`, slightly modified to avoid jumping to non-constant-time routines. Unfortunately, not all of the primitive operations we would like to have are present. Furthermore, we could reduce memory usage in some places, by having this operations present a "conditional" variant. For example, we could have an add operation taking a `choice` flag, allowing us to choose whether or not to perform an addition without leaking information. This would avoid having to use a scratch buffer and a conditional copy.

To gain similar speed to the other primitives, these new primitives would also need to be implemented in assembly. This would be time-consuming, but likely worth the effort. There are also new solutions to help with writing assembly routines in Go, such as the Avo library.

5.3 Upstreaming to `go/crypto`

While we believe our library is immediately useful for the broader ecosystem, it's not realistically going to be replacing the use of `big.Int` in Go's cryptography library any time soon.

The most likely path towards removing `big.Int` from `go/crypto` is likely to move towards specialized arithmetic implementations for each prime field involved in ECC. DSA is a legacy algorithm, where the security flaws introduced by `big.Int` are not of major concern.

This leaves RSA. Unfortunately, the nature of RSA, requiring dynamic moduli, makes it so that a big number library of some kind is necessary. Ideally, this library would be internal to RSA, allowing constant-time operation, and severing the bridge between Go's cryptography package, and `big.Int`.

As a proof of concept, we've implemented a fork of Go's RSA implementation, replacing `big.Int` for encryption and decryption, with an internal number type, using the minimal amount of code necessary to accomplish this.

Using unsaturated limbs, we've found that our fork of RSA suffers only a 1.67x slowdown, while implementing encryption and decryption in constant-time.

We hope to prepare a patch for Go's RSA implementation to merge in this work soon.

6 Conclusion

In summary, we have shown why Go's general purpose big number type, `big.Int`, is not suitable for Cryptography, for various reasons. Unfortunately, this type gets used out of convenience, even in Go's own cryptography library.

To address this, we've created a replacement library for `big.Int`, achieving a slowdown of only 2.6x for most operations, while attempting to provide constant-time operation.

To test the utility of this library, we've replaced the usage of `big.Int` in Go's implementation of RSA, and DSA, and found only a slowdown of 2x.

Acknowledgements

Firstly, I'd like to thank Prof. Bryan Ford for supervising this work. I'd like to thank Pierluca Borsò, for letting me work on this project. Finally, I'd also like to thank Daniel Huigens, and Marin Thiercelin, from ProtonMail, for their advice and industry perspective on this work.

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] Benjamin S Hvass, Diego F Aranha, and Bas Spitters. High-assurance eld inversion for curve-based cryptography. page 23.

[15] 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.

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

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

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

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

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

[21] 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.

[22] 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.

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

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

[25] Thomas Pornin. BearSSL - Big Integer Design, 2020. URL: https://www.bearssl.org/bigint.html.

[26] Thomas Pornin. Optimized Binary GCD for Modular Inversion. page 16, 2020.

[27] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1697–1702, Lausanne, Switzerland, March 2017. IEEE. URL: http://ieeexplore.ieee.org/document/7927267/, doi:10.23919/DATE.2017.7927267.

[28] 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.

[29] 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.

[30] Riad Wahby, Christopher Wood, Armando Faz-Hernández, Sam Scott, and Nick Sullivan. Hashing to Elliptic Curves. URL: https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-09.

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