Constant-Time Big Numbers (for Go)

Lúcás Críostóir Meier

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab (DEDIS)

BSc Semester Project

June 2021

Responsible and Supervisor
Prof. Bryan Ford
EPFL / DEDIS

Contents

1 Introduction

With the rise of the internet and widespread digital services, the importance of secure communication has never been greater. Thankfully, after 50 years of Public Key cryptography [14], we have good theoretical tools to provide this security.

Most of these systems rely on modular arithmetic with large numbers. For example, RSA [31], or Elliptic Curve cryptography [24]. Working with these numbers is not natively supported by hardware. Instead, we use a "Big Number" software library to provide this functionality.

Unfortunately, although Public Key cryptosystems have been thoroughly scrutinized in theory, implementations often suffer from vulnerabilities in practice.

One important class of vulnerability are timing side-channels [18]. This is when an implementation leaks information about secret values through its execution time. Big Number libraries designed without cryptography in mind suffer from these vulnerabilities.

In particular, Go [3] provides a general purpose Big Number type, `big.Int`, which does not provide constant-time operations. Unfortunately, this library gets used for cryptography [12], including inside of Go's own standard library, in the `go/crypto` package.

We've addressed this issue by creating a library [22] providing Big Numbers with constant-time operations. Our library provides the necessary operations for Public Key cryptography, while avoiding timing side-channels. To demonstrate its utility, we've modified Go's `go/crypto` package, replacing the use of `big.Int` in the DSA and RSA systems, achieving a slowdown of only 2x for the latter.

2 Background

In this section, we explain how Big Numbers are used in Public Key cryptography, what timing side-channels are, how we model their threat, as well as what kind of vulnerabilities are present in Go's `big.Int` type.

2.1 Big Numbers in Cryptography

Most Public Key cryptosystems rely on modular arithmetic.

In RSA [31], a public key (e, N) consists of a modulus $N \in \mathbb{N}$, usually 2048 bits long, and an exponent $e \in [0, \varphi(N) - 1]$. To encrypt a message

$m \in [0, N-1]$, we calculate

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

Since N is much larger than a register, we need a Big Number library to implement this system. Modular exponentiation is also not implemented in hardware, requiring extra support in software.

DSA [32] 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 [24] 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 and polynomial arithmetic are used. For prime fields, Big Number functionality is necessary, because the size of the prime is greater than 200 bits.

2.1.1 Implementing Big Numbers

When a modulus is known in advance, a special purpose library implementing arithmetic with this fixed modulus can be used. This is the case for Elliptic Curve cryptography, where the prime field is a fixed parameter of the system. Using a fixed type makes it easier to provide constant-time operation.

A disadvantage is that different systems require different moduli, so it takes more work to implement and support each system. One way to address this is to automatically generate implementations of modular arithmetic, as done by FiatCrypto [15].

In some systems, you need support for dynamic moduli. Take RSA, for example. In this case, you need a library that provides dynamically sized numbers.

2.1.2 Big Numbers in `go/crypto`

Go provides implementations of the Public Key cryptosystems we've mentioned so far in its `go/crypto` package.

Unfortunately [12], the general purpose `big.Int` is used, in part, to implement these systems, despite its potential vulnerability to timing attacks.

For DSA [32], Go uses `big.Int` for all operations, including key generation, signing, and verification.

For RSA [31], Go embeds `big.Int` as part of the API for this package. Key generation, encryption, decryption, signing, and verification all use `big.Int`.

For ECC [24], 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`. Only `P384` uses `big.Int` for field arithmetic directly. The other curves instead convert `big.Int` to an internal type for their field elements, and perform their operations using that type instead. These internal types vary a lot between these curves. `P224` uses hand-written operations in pure Go for its field, `P256` uses an optimized implementation in assembly, and `P521` uses field arithmetic generated by FiatCrypto [15].

2.2 Timing Attacks

A side-channel [17] leaks secret information indirectly, through the observable effects of a program's execution. Timing side-channels use the execution time of a program to infer information about secret data it handles. A timing attack is the use of a timing side-channel to break the security of some program or cryptographic algorithm.

When a program takes a different number of steps based on the value of some secret, this is an obvious timing side-channel. For example, a naive algorithm for comparing inputs with a secret password might stop as soon as a mismatch is found. This algorithm has an inherent timing side-channel. This can be exploited, allowing the secret password to be guessed byte-by-byte.

Not all timing side-channels are this simple. Algorithms that always take the same number of steps can still having timing side-channels because of the underlying hardware. 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 [13] for a survey of these vulnerabilities.

2.2.1 Actual Attacks

Although the presence of a side-channel does not directly lead to attacks, Paul Kocher demonstrated the potential for timing side-channels to break crypto-systems as early as 1995 [18, 19]. These attacks relied on algorithms that perform a varying number of operations based on secret data.

One objection to timing attacks is that while a timing side-channel is catastrophic in theory, in practice this channel is too noisy to exploit. Unfortunately, it's possible to exploit these attacks, even across a network [8, 7]. Noise only makes the channel more difficult to exploit, requiring more samples to detect the underlying signal.

The use of caches as a potential side-channel was identified early on as well [26]. Accessing data takes longer when that data is outside of the cache. If data accesses depend on a secret value, the observed execution time will also depend on this value. Additionally, an attacker located on the same machine can place data into the cache and probe it themselves, learning more precise information about the program's access patterns. This kind of colocation is increasingly common, as more applications are run on cloud servers.

A wide variety of attacks involving caches have been mounted against various cryptosystems [4, 34, 10] : accessing data based on secret values should be avoided in cryptographic code.

2.2.2 Our Threat-Model

These side-channels can be distilled 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: a longer loop uses more operations. In practice, it's difficult to observe the iterations of each loop in a program from a global timing signal, making this a pessimistic rule.

Rule 2 is justified by various cache based side-channels and attacks [4, 34, 10]. Since caches only load information an entire line at a time, this rule may seem too pessimistic. Perhaps only which cache line was accessed should be kept secret [6]. Unfortunately, it's possible to perform attacks on a much finer level [5, 25, 34]. This is why we take a pessimistic position, and assume that accesses leak their exact address.

Rule 3 is justified in two ways. First, if different branches of a conditional statement execute a different number of operations, we can inherently observe which branch was taken. Second, even if both branches execute identitical operations, the CPU's branch predictor can be exploited to leak information about which branch was taken [2, 1, 11].

In addition to these rules, we need a basic set of trusted operations to build our programs. 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 [27]. This assumption is reasonable for the platforms targeted by Go and our library.

2.3 Vulnerabilities in `big.Int`

Go provides a general purpose type for Big Numbers: `big.Int`. This type focuses on being optimized, and useful in various situations. It does not focus on protecting against timing side-channels. Unfortunately, out of convenience, and for lack of better alternatives, it gets used in cryptography, even inside of Go's standard library.

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

2.3.1 Padding

The `big.Int` type 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. By discarding them, operations on this number will have fewer limbs to process, and will be faster.

Unfortunately, this means that `big.Int` pervasively leaks information about the padding of numbers. Operations take more time when a number has more limbs, thus leaking the padding of numbers. This has been exploited in OpenSSL [23], and might potentially be a vulnerability in Go's cryptography library.

2.3.2 Leaky Algorithms

Because `big.Int` is not written with cryptography in mind, its operations violate the rules in 2.2.2. Many methods take a different number of iterations , branch conditionally, or access memory differently depending on their 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 its 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 [19].

To decrypt a ciphertext $c = m^e \mod N$, we would 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 $r \in (\mathbb{Z}/N\mathbb{Z})^*$. 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 effectively mitigates the simplest attacks against exponentiation, a very leaky operation, other methods are left unprotected, and may have subtle exploits. For example, we there are fundamental issues with padding, which has lead to attacks in OpenSSL [23].

3 Implementation

We've implemented a library, called `safenum` [22], intended to provide a replacement for `big.Int`, suitable for cryptography. To test its utility, we've replaced some of `go/crypto`'s usage of `big.Int` with our own library, in a separate repository [21].

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

3.1 The `safenum` library

Safenum defines a `Nat` type, 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 `Nats`. We also provide numerous operations for modular arithmetic, including modular addition, subtraction, multiplication, exponentiation, inversion, reduction, and taking square roots modulo prime numbers. We also provide operations for serializing to and from bytes, as well as converting to and from `big.Int`.

We try to structure the API in a similar way to `big.Int`, where an operation is performed on a separate `Nat` receiving 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 receiving `Nat` for all scratch space needed inside of an operation. Structuring our operations this way limits memory waste.

3.1.1 Handling Size

A `big.Int` always stores a value using as few limbs as possible. Its *true size*, the number of limbs necessary to store a value, always matches its *announced size*, the number of limbs actually stored. When creating a `big.Int`, any padding is immediately stripped away, to satisfy this invariant.

For `Nat` however, we allow values to be padded. This means that the announced size of a number can be larger than its true size. Because the announced size is what actually affects the runtime of our operations, we can make sure that this size depends only on public information, allowing us to keep the true size secret.

This means that every result we produce needs a clear announced size. 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.

For example, when doing modular addition:

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

our result z will have the same announced size as m. This implies that the true size of z is at most that of m. After modular addition, we know that $z \in [0, m-1]$, so this fact about the true size of z does not reveal anything new.

When serializing a Nat, we respect its announced size, and produce zeros for padding as necessary. This is done without any special handling, since extra zeros are already stored to match our announced size.

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 whatever size necessary to store a maximal result. For example, if we multiply numbers x_1 and x_2, of announced sizes b_1 and b_2, then our result will need a size of at most $b_1 + b_2$.

The disadvantage is that we use more and more space for each operation, and we're unable to shrink the announced size, even when we know that our result would fit. 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:

```
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. This size comes from deserializing a value, from matching the size of a modulus, or from manually deciding on an output size.

3.1.2 Moduli

In our library, we use a different type for the moduli used in modular arithmetic: Modulus. There are several reasons for doing this.

First, some modular operations use certain properties of the modulus which can be pre-computed. For example, montgomery multiplication uses $m^{-1} \bmod W$, with W our base. Using a separate type for moduli lets us avoid recomputing these properties, by storing them alongside the value of our modulus.

Second, we allow the true size of a modulus to be leaked. 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, making every operation faster.

This assumption is safe in cryptography. Moduli are often public, like with the public modulus N in RSA. Since the exact value is known, leaking the true size is fine. There are other cases where a private modulus is necessary. Even then, the true size of that modulus remains known. For example, when generating an RSA key, we know 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.

Finally, moduli are also allowed to leak whether or not they are even. The motivation behind this is that certain modular operations have faster variants for odd moduli, or require different algorithms to support even moduli. Rather than providing different methods for these different cases, we choose to select the correct variant internally, allowing all modular operations to work without exception. This requires checking whether or not a modulus is odd or even, in order to dispatch the right operation, leaking this bit of information in the process.

Fortunately, we're not aware of any realistic situation where the evennness of a modulus needs to be kept secret. In fact, for the cryptographic algorithms we know of, the evenness of a modulus is known statically, making this check fine. In RSA, for example, the modulus N is odd by construction, and $\varphi(N)$ is even as a consequence.

Since moduli are stored without padding, we require a separate type to not violate `Nat`'s contract around size. `Nat` also has stronger constraints on information leakage, so providing a separate type helps to avoid accidentally violating them.

3.2 Constant-Time Operations

The rules we established in our threat model 2.2.2 are quite stringent. For most operations, we need to have conditional behavior depending on the values we process. Without access to branching, this seems difficult. Thankfully, we can emulate branching without leaking information. The idea is simple: to choose between two branches, we perform both, and then combine the results together, without revealing which result is produced.

For example, a standard algorithm for modular subtraction would look like this (in pseudo-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 is that by conditionally adding in m, we reveal whether or not $y > x$, and a borrow occurred. Our solution uses a new primitive:

```
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 should leak no information about the condition.

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

```
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 pattern is what allows us to replace branching with constant-time operations throughout our algorithms.

3.2.1 Building Primitives

How do you make primitives like `ctCondCopy`, which let you choose results without leaking which choice was made?

The methods for constant-time choice are analogous to those used for variable-time choice. 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 avoid having the compiler re-insert branches into our code.

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. The bitwise operations don't leak information about our choice, unlike a conditional statement.

If v == 0, then mask contains only zeros, and we're left with y. When v == 1, then mask only contains ones. The ys cancel each other out, leaving us with x.

We can use this primitive to build up a larger selection primitive, allowing us to conditionally assign an entire slice of limbs to another:

```
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 a way to create choice values. These are built up in a similar way, from small functions to larger ones.

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, if you take any number $x \neq 0$, then the most significant bit of either x or $-x$ is set. For 0, neither are set. We can check that q is non-zero by seeing if either bit is set:

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

Since q is zero precisely when x and y are equal, we negate this non-zero check.

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

```go
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
}
```

Two slices are equal when each of their limbs match. Since we can't exit early without leaking information about x and y, we implement this formula using bitwise operators, without any short-circuiting.

3.3 Algorithm Choices

While going over how each operation works in detail is outside the scope of this report, describing some the high-level techniques used for these operations is still valuable.

Many of these operations were inspired by the excellent work of Thomas Pornin in BearSSL [28].

3.3.1 Modular Reduction

To reduce a number 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 $b \in [0, W-1]$. We can use this operation to reduce an arbitrary x. First, we can write out x in base W:

$$x = x_n W^n + x_{n-1} W^{n-1} + \cdots + x_0$$

Then, we can rewrite this expression to see how shifting plays a role:

$$x = ((x_n W + x_{n-1})W + \cdots)W + x_0$$

Having written x in this form, it becomes clear that to reduce x modulo m, we can use our shifting primitive, folding in the limbs of x from most significant to least significant.

Implementing the shifting operation needs the quotient $q := \lfloor z/m \rfloor$, in order to calculate the remainder $z - qm$. Instead of calculating q directly, we instead produce an estimate $\hat{q}$, by using the most significant 64 bits of m to divide 128 corresponding bits of z. This estimate satisfies $\hat{q} = q \pm 1$, so we then conditionally add and subtract m to correct for this error. This technique is further described in [28].

3.3.2 Inversion

For modular inversion, which method we use depends on whether our modulus is odd or even. As mentioned previously, we're allowed to choose the right method dynamically.

For odd moduli, we use the binary GCD algorithm, as described in [29]. We have yet to implement the most optimized version, which accumulates intermediate results into single limb registers. Instead, we perform full width operations at each iteration.

For an even modulus m, there's a standard trick to calculate $x^{-1} \mod m$. First, we calculate $u := m^{-1} \mod x$, using our method for odd moduli, and then we calculate our desired inverse as:

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

This requires implementing a division operation that works with padded numbers. We can't reuse the limb-by-limb modular reduction routine we normally use, and instead adapt a bit-by-bit reduction routine, as described in [28].

3.3.3 Exponentiation

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

3.3.4 Multiplication

For modular multiplication, we calculate the full product ab, and then reduce this result modulo m. This works for both odd and even moduli.

In the case of odd moduli, using Montgomery multiplication [16, 28] is faster, but requires converting inputs into their Montgomery representation, producing outputs in this representation. For exponentiation, the cost of converting to and from this representation is amortized, since we do many multiplications between conversions, making it considerably faster to use this technique. Unfortunately, Montgomery multiplication doesn't work for even moduli, so exponentiation is slower in that case.

3.3.5 Modular Square Roots

To calculate $\sqrt{z} \mod p$, we assume that p is prime. Calculating square roots with a composite modulus is, in general, as hard as factoring it [9], which is why we only support prime moduli. 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 [33].

3.4 Implementation Choices

In this section, we describe a few more details about how numbers are stored, and the motivation behind these choices.

3.4.1 Saturated or Unsatured Limbs

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 [28] 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 analogous choice.

There are two compelling reasons for using unsaturated limbs.

First, this leaves an extra bit of space to hold a carry or borrow after an addition or subtraction. This allows us to chain together carries to implement operations over multiple limbs, without having to use assembly instructions. In Go, this isn't a concern, since `bits.Add` and `bits.Sub` can be used to implement these intrinsics in a cross-platform way.

Second, 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.

One disadvantage of using unsaturated limbs comes when converting numbers to and from bytes. With fully saturated limbs, our 64 bit limbs are composed of exactly 8 bytes. With 63 bit limbs, this isn't the case, making conversion 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, as we do now.

Finally, by using saturated limbs, we can use the assembly routines already implemented for low level operations in `big.Int`, which uses saturated limbs. These low-level routines are constant-time, and speed up basic operations.

Interoperability with these routines is ultimately the main reason why we opted for using saturated limbs.

3.4.2 Redundant Reductions

Our library tries 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 are already in range is not desirable, since this operation is relatively expensive. Ideally, we'd like to avoid reducing inputs when we know this reduction is redundant.

To implement this, each `Nat` stores a pointer to a modulus, indicating that it has been reduced by this modulus. When we reduce a number modulo m, we check this pointer, and skip the reduction if it matches m. If we modify the value of a number, we update the modulus it points to accordingly. For example:

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

will set `z`'s modulus to `m`, and calling:

```
z.SetBytes(data)
```

will clear `z`'s modulus, making it `nil`.

We modify this pointer based strictly on what methods are called, never on the actual value of a result. Thus, the dynamic checks of this pointer only depend on the call-graph 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 (odd) 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	10,980,842	12,164,599	0.90
Modular Addition	6,986,739	3,075,188	2.27
Multiplication	1,316,322	542,385	2.43
Modular Reduction	454,917	63,253	7.19
Modular Multiplication	1,000,000	44,596	22.42
Modular Inversion	1,000,000	621	1610
Modular Exponentiation	223	86	2.59

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 slower compared to `big.Int` for exponentiation, although some operations are much slower.

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:

Operation	op / s (`big.Int`)	op / s (`Nat`)	ratio
$\sqrt{z} \mod p_3$	40,464	26,886	1.50
$\sqrt{z} \mod p_1$	-	7,867	-

We couldn't find a large value where Go's Tonneli Shanks routine managed to find a square root without hanging, although we expect the ratio to be similar to the other case.

4.2 Comparison with `go/crypto`

We've created a forked package [21] of `go/crypto`, where we've replaced `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 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 using 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 [30] 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 variable-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 these 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 [20].

5.3 Upstreaming to `go/crypto`

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

The most likely path towards removing `big.Int` from `go/crypto` is 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, because RSA requires dynamic moduli, we need a Big Number library of some kind. Ideally, this library would be internal to RSA, allowing constant-time operation, and severing the bridge between Go's cryptography package, and `big.Int`.

We've submitted a patch [1] for Go's implementation of RSA, replacing `big.Int` with an internal number type, using the minimal amount of code necessary to implement encryption and decryption. The public API, as well as key generation, still use `big.Int`.

Using unsaturated limbs, we've found that our version of RSA suffers only a 1.7x slowdown, while implementing encryption and decryption in constant-time.

6 Conclusion

In summary, we have shown why Go's general purpose big number type, `big.Int`, is not suitable for Cryptography. Unfortunately, this type gets used out of convenience, and for lack of better alternatives, 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.

[1] `https://go-review.googlesource.com/c/go/+/326012`

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 Professor Bryan Ford for supervising this work. I'd like to thank Pierluca Borsò as well, 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] Joe Buhler and Stan Wagon. Basic algorithms in number theory. page 44, 2008.

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] C. Kaya Koc, T. Acar, and B.S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996. Conference Name: IEEE Micro. doi:10.1109/40.502403.

[17] 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.

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

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

[20] Michael McLoughlin. mmcloughlin/avo, June 2021. original-date: 2018-11-04T17:44:21Z. URL: https://github.com/mmcloughlin/avo.

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

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

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

[24] 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.

[25] 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.

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

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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