Constant-Time Arithmetic for Safer Cryptography

Lúcás Críostóir Meier[1], Simone Colombo[1], Marin Thiercelin[2], and Bryan Ford[1]

[1]DEDIS, EPFL, Switzerland
{lucascriostoir.meier, simone.colombo, bryan.ford}@epfl.ch
[2]ProtonMail, Switzerland
marin.thiercelin@protonmail.com

Abstract

The humble integers, $\mathbb{Z}$, are the backbone of many cryptosystems. When bridging the gap from theoretical systems to real-world implementations, programmers often look towards general purpose libraries to implement the arbitrary-precision arithmetic required. Alas, these libraries are often conceived without cryptography in mind, leaving applications potentially vulnerable to timing attacks. To address this, we present `saferith`, a library providing safer arbitrary-precision arithmetic for cryptography, through constant-time operations. The main challenge was in designing an API to provide this functionality alongside these stronger constant-time guarantees. We benchmarked the performance of our library against Go's `big.Int` library, and found an acceptable slowdown of only 2.56× for modular exponentiation, the most expensive operation. Our library was also used to implement a variety cryptosystems and applications, in collaboration with industrial partners ProtonMail and Taurus. Porting implementations to use our library is relatively easy: it took the first author under 8 hours to port Go's implementation of P-384.

1 Introduction

At CRYPTO '96, Kocher [Koc96] warned the cryptographic community about the risks of timing attacks against public-key cryptosystems. Twenty-five years later, cryptographic implementations are ubiquitous; the risks of these attacks as well. Timing attacks enable an adversary to extract secrets, such as private keys or plaintexts, from a system, by measuring the time it takes to perform sensitive operations, such as decryption. These attacks often arise from the increasingly complex micro-architecture used to implement operations in hardware [GYCH18]. They can even be carried out over a network [BB05, BT11], and have lead to concrete attacks against AES [Ber05, OST06], RSA [AKS06, AKS07, YGH17, CAPGATB19], and Diffie-Hellman key exchange [MBA+19].

To avoid these attacks, it is possible to write *constant-time* programs, in which operations only vary in time based on public values. This involves various techniques [Aum20], such as replacing branches with bitwise operations, avoiding array accesses with secret indices, or making sure that loops have a fixed number of iterations. Constant-time operations are easier to implement with values that have a fixed size. This is the case for many modern cryptosystems, which often use finite fields with fixed parameters. In this case, each set of parameters can have an optimized—and constant-time—implementation.

Unfortunately, some cryptosystems, like RSA [RSA78], or Paillier [Pai99], do not have fixed parameters. Instead, these parameters are part of the public key. In this case, arbitrary-precision arithmetic is necessary, which requires more care to design a constant-time implementation. While most programming languages have popular libraries providing arbitrary-precision arithmetic, they are, alas, not designed with cryptography in mind, leaving them vulnerable to timing attacks.

Go [Aut], in particular, suffers from these issues. Go provides the `big.Int` type for working with arbitrary-precision signed integers. Although the authors provide no guarantees of constant-time operations, `big.Int` gets used for cryptography, even inside of Go's standard library. This is especially concerning, as an increasing amount of cryptographic software is written in Go.

To provide arbitrary-precision arithmetic with the constant-time properties needed for cryptography, we created the `saferith` library [Mei21], implemented in Go. The main challenge of our work was in designing an API providing all the functionality needed from `big.Int`, as well as the additional constant-time guarantees needed for cryptography. This involved synthesizing the variety of timing-attacks into a succinct threat model, as well as determining how to safely manipulate arbitrary-precision integers, even though operations vary in execution time based on the size of numbers. The crux of our solution is to pad numbers to different public sizes. While operations may leak this public size, the underlying value is kept hidden.

To evaluate our library, we created a couple of experiments to compare the constant-time execution of our library with the variable-time execution of `big.Int`. We also ran micro-benchmarks showing that these extra guarantees come only at a performance cost of 2.56× for exponentiation, the most expensive arithmetic operation.

To test the practicality of our library, we modified Go's implementations of RSA and NIST's P-384 curve [CMRR19] to use `saferith`. We also worked with industrial partners to port their applications to use our library. ProtonMail ported their implementation of the Secure Remote Password (SRP) protocol [Wu97] away from `big.Int`. With Taurus, we integrated our library into their implementation [HMA21] of the CGG+ Threshold ECDSA protocol [CGG+20]. This state-of-the-art protocol is used by Fireblocks, among others, to secure their implementation of cryptocurrency wallets[1].

In summary, our contributions are:

- A succinct threat-model for timing-attacks against arbitrary-precision arithmetic libraries.
- A language-agnostic API for constant-time execution of arbitrary-precision arithmetic operations.
- An efficient Go library providing constant-time arbitrary-precision arithmetic.
- An evaluation of the library using micro-benchmarks and implementations of real-world protocols with industrial partners.

2 Background

In this section, we review timing side-channel attacks. We focus in particular on attacks targeting arbitrary-precision arithmetic libraries and the Go's standard cryptography library.

[1] https://apnews.com/press-release/pr-newswire/26aab91e254bc254d331ceafc20b9859

2.1 Timing Side-Channels

A timing side-channel [Koc96, KSWH98] arises when an implementation of some algorithm leaks information about the values it processes through observable timing patterns.

When a program performs more operations based on the value of its inputs, it inevitably leaks information about those inputs: their values can be inferred by observing how many operations were performed. Additionally, programs performing a fixed number of operations may have timing leaks because of how the underlying hardware implements these operations. These micro-architectural attacks are covered in more detail in our threat model (Section 3.1), as well as in the survey by Ge et al. [GYCH18].

2.2 Arbitrary-Precision Arithmetic in Practice

Many programming languages have a general-purpose type for arbitrary-precision arithmetic in their standard library, or a prominent one in their ecosystem. For example, Python, Haskell, and Go all have arbitrary-precision arithmetic types in their standard libraries, while Rust has the `num-bigint` library, and C has `GMP`. These types provide no guarantees around constant-time operations. Yet, for lack of better alternatives, they do get used for cryptography. From this perspective, these types suffer from two major issues.

The first issue is that algorithms branch or access memory based on input values. For example, exponentiation may branch on the bits of the exponent. Not using constant-time implementations of these algorithms is understandable: these libraries are not intended to be used in cryptography. The second, more fundamental, issue is that these libraries store numbers without any zero padding. Numbers are represented as a sequence of register sized *limbs*, analogous to bit sized digits. Numbers are stored *normalized*, with leading zero limbs removed. Even if an operation produces an output of a given length, the result is truncated based on its value. Because operations inevitably vary in time based on the number of limbs a value is stored with, removing this padding can make certain values produce considerably different timing patterns.

While timing vulnerabilities like these are concerning in theory, they may not always lead to practical attacks. Certain operations are particularly vulnerable, such as exponentiation. A recent paper by De Feo et al. [FPS21] analyzed, among other things, Go's implementation of DSA with its own `big.Int` type, and found that it was practically exploitable. The problem was that `big.Int`'s exponentiation method uses direct table lookups, making it vulnerable to a cache-based timing attack.

More subtle vulnerabilities can come from the pervasive padding issues mentioned above: the OpenSSL library was affected by a similar issue for over 20 years, as recently exposed by Merget et al. [MBA+19].

2.3 Go: A Case Study

We have chosen Go [Aut] as a poignant example of how vulnerable arbitrary-precision arithmetic libraries see widespread usage.

Go provides a type for arbitrary-precision arithmetic in its standard library: `big.Int`. Despite suffering from the many shortcomings highlighted above, it is widely used in cryptography, including inside of Go's own standard cryptography package: `go/crypto`. Inside this package, there are three cryptographic schemes which make use of `big.Int`.

In DSA, `big.Int` is used for signing and verification. While this implementation is marked as deprecated by the Go authors, it is nonetheless imported by more than two thousand known packages.

In RSA, Go embeds `big.Int` as part of the API for the package. Key-generation, encryption, decryption, signing, and verification use `big.Int`. The authors are aware of the shortcomings of `big.Int`, and implemented a common mitigation for RSA decryption (and thus signing): blinding. Instead of calculating $c^d \mod N$, a random nonce r is generated, and $(rc)^d \mod N$ is calculated instead. This mitigation was first suggested in [Koc96], and is effective against the simple attacks detailed in this early paper. Unfortunately, blinding can fail to protect against attacks making use of more granular timing information [SI11].

For elliptic curve cryptography, `go/crypto` provides implementations of various NIST curves. The library also provides a generic interface for curves in Weierstrass form. This interface is specified in terms of the `big.Int` type. Internally, only one curve is implemented using `big.Int`: P-384 [CMRR19]. Not only are the field operations performed using `big.Int`, but the elliptic curve operations use variable-time branching pervasively. Most importantly, scalar multiplication is not performed in constant-time, which is particularly concerning.

3 `saferith`

The `saferith` library provides an alternative to `big.Int`, suitable for cryptography. In this section, we go over the threat model it uses, as well as the resulting API design choices. We also present a practical experiments to test our library's claims of constant-time operation. Finally, we present the performance cost of these additional guarantees.

3.1 Threat Model

We have distilled the concerns outlined in Section 2.1 into a simple list of rules, which constitute the threat model for our library:

1. Any loop leaks the number of iterations taken.
2. Any memory access leaks the address (or index) accessed.
3. Any conditional statement leaks which branch was taken.

We take a pessimistic position, assuming that each individual violation of these rules leaks perfectly to the adversary.

Rule 1 is justified by a trivial observation: a longer loop uses more operations. In practice, it is difficult to observe the duration of each loop in a larger program, making this a pessimistic rule. Rule 2 is justified by various cache based side-channels and attacks [Ber05, YGH17, CAPGATB19]. 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 [Bri11]. Unfortunately, it is possible to perform attacks based on accesses inside of a cache line [BS13, OST06, YGH17]. This is why we take a pessimistic position, and assume that accesses leak their exact address. The justification for Rule 3 is twofold. First, if different branches of a conditional statement execute a different number of operations, we can observe which branch was taken. Second, even if both branches execute identical operations, the CPU's branch predictor can be exploited to leak information about the selected branch [AKS06, AKS07, EPAG16].

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 [Por]. This assumption is reasonable for the platforms targeted by our library.

3.2 Challenges in API Design

The central challenge of our work was in designing an API for arbitrary-precision arithmetic that provided all the functionality programmers needed from `big.Int`, alongside the safety guarantees they were sorely lacking.

The first issue is how to deal with padding. Operations fundamentally vary in time based on the number of bits used to represent values. `big.Int` determines this size based on the significant bits of each value, leading to pervasive timing leaks. At first, it seems difficult to keep values secret if large values inevitably need more storage. Fortunately, while the value of a number needs to be kept secret, there is usually a publicly known bound for how large it can be. For example, in RSA the public modulus N bounds the size of ciphertexts, finite fields are ≈ 256 bits in size, etc. Because of this, we can define this public bound to be the *announced size*. This size determines the number of bits we use to store our values and can be larger than the *true size*, which would be the minimal number of bits needed. By padding numbers to a publicly known size, and only allowing operations to vary in time based on this public size, we can keep their underlying real values secret.

Another issue comes from handling negative numbers. `big.Int` stores integers in $\mathbb{Z}$ as an absolute value in $\mathbb{N}$, along with a boolean sign bit. The problem is that handling this sign involves branching, revealing which numbers are negative. With a few exceptions—notably, the CGG+ protocol—most applications only need numbers in $\mathbb{N}$. Indeed, most cryptographic schemes involve only modular arithmetic, with a few intermediate operations involving standard arithmetic in $\mathbb{N}$. Most of our functionality is therefore provided through a `saferith.Nat` type, providing modular arithmetic and other operations in $\mathbb{N}$.

Since modular arithmetic is so important, one question is whether moduli need to be padded or not. The size of a modulus has a direct impact on the performance of operations, so removing padding is desirable. Fortunately, while the value of a modulus is sometimes secret, there is always a public bound for its size, e.g. the RSA's exponent modulus $\varphi(N)$ has approximately the same as N. Because of this, we introduce a separate modulus type `saferith.Modulus`, which is stored without any padding.

When implementing the P-384 curve [CMRR19] we needed to conditionally choose, in constant-time, between different numbers, by comparing their values. This functionality was already used internally to implement various operations, but this application required us to expose it. We do this by providing a `saferith.Choice` type, representing a boolean value suitable for constant-time choice. We also expose operations for comparing numbers, producing a `saferith.Choice`. We have an additional operation using a `saferith.Choice` to conditionally assign a value to a `saferith.Nat`.

When implementing the threshold ECDSA protocol by Canetti et al. [CGG+20], we realized that we needed negative integers. The CGG+ protocol represents exponents over a symmetric range, including negative numbers. To support them, we have a `saferith.Int` type, representing arbitrary integers in $\mathbb{Z}$ with an absolute value, as a `saferith.Nat`, as well as a sign bit, as a `saferith.Choice`. In addition to arithmetic operations, we also support acting as an exponent,

and converting between modular numbers in the positive $[0, \ldots, N-1]$ to the symmetric range $[-N/2, \ldots, N/2]$.

3.3 Constant-Time Properties

In order to test our claims of constant-time execution, we devised a two simple experiments to observe a difference in the timing patterns of Go's `big.Int` and `saferith`.

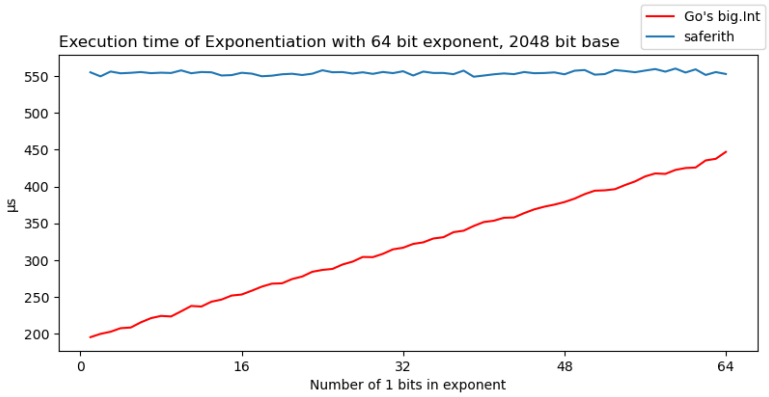

Figure 1: Execution time of exponentiation (`Exp`), averaged over 400 runs, with an increasing number of set bits. For `big.Int`, execution time is clearly correlated with this value. For `Nat`, we observe no such correlation.

In Figure 1, we measured the execution time of exponentiation with a 64 bit exponent, and a 2048 bit modulus. For small exponents, `big.Int` conditionally performs a multiplication for each bit of the exponent. This is clear in the timing pattern: the execution grows linearly with the number of bits set in the exponent. For `Nat`, on the other hand, there is no observable correlation between the weight of the exponent and the execution time. For larger exponents, `big.Int` does not branch based on bits, but looks up elements in a table using multiple bits at a time. This is exploitable through a cache based timing attack [FPS21].

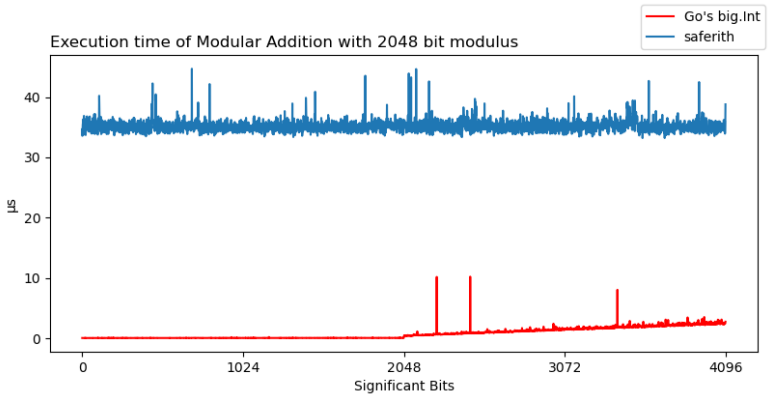

Figure 2: Execution time of modular addition (`ModAdd`) with a 2048 bit modulus, averaged over 400 runs. For `big.Int`, execution time is clearly correlated with size, but not for `Nat`.

We also tested modular addition with a 2048 bit modulus in Figure 2. The input varied in its significant bit count, but was always padded to 4096 bits. When the size of the input surpasses the size of the modulus, a modular reduction is performed. This operation varies greatly in time based on the announced size of the input. Because `big.Int` always removes padding, we can clearly see this variation as the true size of the input grows. With `saferith` all inputs behave as slowly as a full 4096 bit input.

In both figures, a noticeable amount of noise can be observed. We posit that this noise stems from cache access patterns, and note that the noise does not seem to be correlated with the value of the inputs.

Both of these experiments are quite crude. While they can show the leakiness of `big.Int`, they do not guarantee the opaqueness of `saferith`.

3.4 Performance

Implementing constant-time operations comes with a performance slowdown. To quantify this, we compared the performance of `saferith.Nat` and `big.Int` for basic arithmetic operations, as presented in Table 1.

Operation	`big.Int`	`saferith.Nat`	slowdown
Modular Addition	141ns	455ns	3.23×
Modular Multiplication	1.01μs	21.93μs	21.66×
Modular Reduction	2.89μs	18.03μs	6.24×
Modular Inversion	0.97μs	355.26μs	366.25×
Exponentiation	5.47ms	14.01ms	2.56×
Modular Square Roots	30.9μs	47.5μs	1.54×

Table 1: Performance of arithmetic operations in `big.Int` and `Nat`, with 2048 bit numbers, averaged over 5 runs. The slowdown imposed by constant-time operations ranges from 1.54× to 366.25×.

These operations are benchmarked with moduli of 2048 bits, and inputs with that many significant bits. We stress that exponentiation is by far the most expensive operation, which makes it encouraging to see that the slowdown on this operation is only 2.56×. Modular inversion has the largest slowdown. To implement this operation, we used Pornin's optimized binary GCD [Por20c], which differs substantially from Go's variable-time algorithm based on Euclidean division. This explains the significant slowdown for modular inversion as compared to other operations. On the bright side, this method is still faster than inversion through exponentiation, which is a popular alternative to avoid writing a separate operation.

3.5 Limitations and Further Work

The main limitation of our work is in the confidence we can provide around constant-time execution. Our code has not undergone an audit, and we have only performed basic tests to get a sense of constant-time execution. Further work could involve performing more robust statistical tests with `dudect` [RBV17], instrumenting our code to detect timing leaks with `ctgrind` [Lan10], or formally verifying our code with `ctverify` [ABB+16].

Our library is written in Go, but the techniques it uses are language-agnostic. We could port it to other languages, enabling them to reap the benefits provided by our library.

The performance of our library could be improved further. While we do reuse some of the safe assembly routines written for Go's `big.Int`, not all of the functionality we needed was available. Rewriting certain routines—like Montgomery multiplication—in assembly would shorten the performance gap between `saferith` and `big.Int`.

4 Results

To evaluate the practicality of `saferith` for real-world applications, we ported a variety of cryptographic systems to use our library instead of Go's `big.Int`. In addition, we collaborated with ProtonMail and Taurus as industrial partners to integrate `saferith` into their applications. We helped patch ProtonMail's implementation of the SRP protocol [Wu97], and Taurus's implementation of the CGG+ Threshold ECDSA Protocol [CGG+20]. In this section, we present these systems, along with benchmarks evaluating the performance cost of our changes.

4.1 RSA

We adapted Go's implementation of RSA to use our library[2]. Go's implementation uses `big.Int` extensively, for both the public API, and for internal operations.

Operation	`big.Int`	`saferith.Nat`	slowdown
Decrypt	1.71ms	3.73ms	2.18×
Sign	1.80ms	4.21ms	2.34×
Decrypt (3 prime Modulus)	0.98ms	1.96ms	2.00×

Table 2: Performance of 2048 bit RSA, averaged over 5 runs.

Table 2 shows the benchmarks of our version's performance, with a 2048 bit public key. We notice that the performance of our library is better in this realistic benchmark as compared to the previous benchmarks for basic operations. This is because of the amortization of operations when implementing the full system.

We also upstreamed a portion of this work, implementing the bare necessities needed for RSA encryption and decryption, in the form of a patch [3] to Go's standard library.

4.2 P-384

Go implements the NIST P-384 curve using `big.Int`, making it the perfect candidate for a `saferith`-based replacement.

We benchmarked the performance of scalar multiplication, the principal operation for ECC schemes, in Table 3. The performance of our implementation predictably incurs a 10× slowdown. We implemented operations using the minimal adjustments necessary to make them constant-time. This entails a performance overhead, because of the extra work in scalar multiplication, as well as in each individual curve operation. We could improve this by implementing complete

[2]https://github.com/cronokirby/saferith-misc
[3]https://go-review.googlesource.com/c/go/+/326012

Operation	`big.Int`	`saferith.Nat`	slowdown
Scalar Base Multiplication	4.20ms	45.40ms	10.79×
Scalar Multiplication	4.37ms	46.05ms	10.53×

Table 3: Performance of P-384 Operations, averaged over 5 runs.

formulas for addition [RCB16], or better ladder formulas [Joy07, Ham20]. But, our goal with this implementation was not to write a more performant implementation, but rather to evaluate the ease of use and speed of porting an existing implementation using `saferith`. The first author ported the implementation from `go/crypto` line-by-line, in under 8 hours.

4.3 CGG+ ECDSA

Recently, Canetti et al. proposed a new state of the art implementation of threshold ECDSA signatures [CGG+20], in collaboration with Fireblocks[4]. This protocol makes heavy use of Paillier encryption [Pai99], Pedersen commitments [Ped92], and a battery of Zero-Knowledge Proofs [GMR89, DSMP88] around their usage. All of these make extensive use of arbitrary-precision arithmetic.

Taurus released[5] an implementation of this protocol [HMA21], initially using Go's `big.Int`. We patched the implementation to use `saferith` where necessary. The needs of this application drove the creation of new functionality in `saferith`, namely, the `saferith.Int` type. This example shows the ability of `saferith` to easily replace `big.Int`, even when used extensively across a codebase of thousands of lines of code.

Paillier Encryption Part of the application uses Paillier encryption, which requires arbitrary-precision arithmetic.

Operation	`big.Int`	`saferith.Nat`	slowdown
Encryption	26.7ms	82.2ms	3.08×
Ciphertext Addition	10.6μs	64.0μs	5.02×
Ciphertext Scaling	7.34ms	26.72ms	3.63×

Table 4: Performance of 2048 bit Paillier operations, averaged over 5 runs.

Given a 2048 bit modulus N, the operations benchmarked here involve multiplication and exponentiation modulo N^2. The results presented in Table 4 match our expectations, based on our benchmarks for the underlying operations.

Zero-Knowledge Proofs The CGG+ protocol makes use of Pedersen commitments, as well as Zero-Knowledge proofs related to Paillier encryption. Both of these are based on modular—and thus arbitrary-precision—arithmetic.

We benchmarked these operations in Table 5. The `ZK Prm` operation creates a proof that a number is a valid Pedersen commitment, and `ZK Mod` creates a proof that a number is a valid

[4]https://apnews.com/press-release/pr-newswire/26aab91e254bc254d331ceafc20b9859
[5]https://www.taurushq.com/en/insights/taurus-mpc-cmp

Paillier-Blum modulus. For a more detailed description of these operations, we refer the reader to the CGG+ paper [CGG+20].

Operation	`big.Int`	`saferith.Nat`	slowdown
Pedersen Commit	1.51ms	7.26ms	4.81 ×
Pedersen Verify	2.34ms	2.25ms	0.96 ×
ZK Mod	929ms	3031ms	3.26 ×
ZK Prm	437ms	1066ms	2.43 ×

Table 5: Performance of Pedersen and ZK Proofs, averaged over 5 runs.

We note that once again, the performance slowdown is reasonable, and matches our expectations based on the performance of basic arithmetic operations. Our implementation uses `saferith` only for proof generation, and not verification, because the latter step only operates on public values.

4.4 SRP at ProtonMail

ProtonMail's security model requires that the servers never gain access to any private user data. This means that a user's password, which can unlock private keys, must never leave the client. ProtonMail uses the SRP protocol [Wu97] to authenticate users while ensuring their passwords are never revealed to the server. Mobile applications rely on `go-srp`[6], a Go implementation of this protocol. The SRP protocol requires computing a modular exponentiation of a fixed base with the password. This operation was originally done using `big.Int`, making `go-srp` potentially vulnerable to the timing attacks we have previously mentioned. To move away from `big.Int`, `saferith` offered a drop-in solution. `saferith.Nat` is easily convertible from and to `big.Int`, which allowed swapping the type only for operations involving the password.

Operation	`big.Int`	`saferith.Nat`	Slowdown
Verifier generation	63.5ms	86.8ms	1.37×
Proof generation	3.70ms	11.65ms	3.14×

Table 6: Performance of `go-srp`, averaged over 5 runs.

As shown in Table 6, using `saferith` instead of `big.Int` does have a cost in terms of performance. However, in the case of `go-srp`, it is used only during sign-up (verifier generation) and log-in (proof generation), which are rare client operations. The added security for a relatively small performance cost—particularly in absolute terms—constituted an acceptable trade-off in this case.

5 Related Work

Other prominent libraries such as OpenSSL or BearSSL have their own implementations of arbitrary-precision arithmetic. Pornin's BearSSL also has extensive documentation on the design and algorithms behind their implementation [Por20b, Por20a, Por20c]. Other references

[6] https://github.com/opencoff/go-srp

on arbitrary-precision arithmetic [Doc06b, MVOV18, CP06, Coh13] as well as finite field arithmetic [Doc06a, Jun93, LN97, Shp13] were invaluable to our own work.

A recent approach to render integrating finite-field arithmetic easier is FiatCrypto [EPG+19], which can generate formally verified implementations for individual fields. This builds on work using formal methods to verify constant-time properties [ABB+16, BPT19]. Dynamic analysis is another approach to check for variable-time execution [Lan10, RQPA16]. Statistical tests, like those done in `dudect` [RBV17], are an even simpler method to discover timing leaks. These build on the leakage detection tests pioneered by Coron et al. [CKN00, CNK04].

Our work also builds upon the seminal papers on timing—and other side-channel—analysis [Koc95, Koc96, KSWH98]. These have flourished into practical demonstrations [BB05, BT11], as well as concrete attacks against AES [Ber05, OST06], RSA [AKS06, AKS07, YGH17, CAPGATB19], and Diffie-Hellman key exchange [MBA+19], to only mention a few.

6 Conclusion

In this paper, we presented `saferith`, a library providing arbitrary-precision arithmetic with constant-time operations. We benchmarked its performance, and found it to be 2.56× slower than Go's `big.Int`, for modular exponentiation, the most expensive operation. This is an acceptable trade-off for the stronger security guarantees our library tries to provide. Furthermore, we evaluated its practicality by implementing several real-world cryptosystems, including SRP and a Threshold ECDSA library, in collaboration with ProtonMail and Taurus, respectively, as industrial partners.

References

[ABB+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, 2016.

[AKS06] Onur Aciçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In *CT-RSA*, 2006.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ASIACCS*, March 2007.

[Aum20] Jean-Philippe Aumasson. cryptocoding. `https://github.com/veorq/cryptocoding`, 2020.

[Aut] The Go Authors. The Go Programming Language Specification - The Go Programming Language.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, August 2005.

[Ber05] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[BPT19] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27(1):137–163, 2019.

[Bri11] Ernie Brickell. Technologies to improve platform security. In *CHES*, 2011.

[BS13] Daniel J Bernstein and Peter Schwabe. A word of warning. In *CHES*, 2013.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *ESORICS*, 2011.

[CAPGATB19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *TCHES*, August 2019.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. In *ACM SIGSAC*, October 2020.

[CKN00] Jean-Sébasticn Coron, Paul Kocher, and David Naccache. Statistics and secret leakage. In *International Conference on Financial Cryptography*, pages 157–173. Springer, 2000.

[CMRR19] Lily Chen, Dustin Moody, Andrew Regenscheid, and Karen Randall. Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters. Technical Report NIST Special Publication (SP) 800-186 (Draft), National Institute of Standards and Technology, October 2019.

[CNK04] Jean-Sébastien Coron, David Naccache, and Paul Kocher. Statistics and secret leakage. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):492–508, 2004.

[Coh13] Henri Cohen. *A course in computational algebraic number theory*, volume 138. Springer Science & Business Media, 2013.

[CP06] Richard Crandall and Carl B Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006.

[Doc06a] Christophe Doche. Finite field arithmetic. In *Handbook of elliptic and hyperelliptic curve cryptography*, pages 201–237. CRC Press, Taylor & Francis Group, 2006.

[Doc06b] Christophe Doche. Integer arithmetic. In *Handbook of elliptic and hyperelliptic curve cryptography*, pages 169–199. CRC Press, Taylor & Francis Group, 2006.

[DSMP88] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-Interactive Zero-Knowledge Proof Systems. In *CRYPTO*, 1988.

[EPAG16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, October 2016.

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE SP*, May 2019.

[FPS21] Luca De Feo, Bertram Poettering, and Alessandro Sorniotti. On the (in)security of elgamal in openpgp. Cryptology ePrint Archive, Report 2021/923, 2021.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18(1):186–208, February 1989. Publisher: Society for Industrial and Applied Mathematics.

[GYCH18] 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.

[Ham20] Mike Hamburg. Faster Montgomery and double-add ladders for short Weierstrass curves. Technical Report 437, 2020.

[HMA21] Adrian Hamelink, Lúcás Críostóir Meier, and J.-P. Aumasson. multi-party-sig, 6 2021.

[Joy07] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In *CHES*. 2007.

[Jun93] Dieter Jungnickel. Finite fields: structure and arithmetics. 1993.

[Koc95] Paul C Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks. page 6, 1995.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *ESORICS*, 1998.

[Lan10] Adam Langley. Checking that functions are constant time with valgrind, 2010.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite fields*. Number 20. Cambridge university press, 1997.

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

[Mei21] Lúcás Críostóir Meier. cronokirby/saferith. `https://github.com/cronokirby/saferith`, May 2021.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[Pai99] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*, 1999.

[Ped92] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *CRYPTO*. Springer, 1992.

[Por] Thomas Pornin. BearSSL - Constant-Time Mul.

[Por20a] Thomas Pornin. BearSSL - Big Integer Design, 2020.

[Por20b] Thomas Pornin. BearSSL - Constant-Time Crypto, 2020.

[Por20c] Thomas Pornin. Optimized Binary GCD for Modular Inversion. page 16, 2020.

[RBV17] 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.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete Addition Formulas for Prime Order Elliptic Curves. In *EUROCRYPT*. 2016.

[RQPA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *ICCC*, 2016.

[RSA78] 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.

[Shp13] Igor Shparlinski. *Finite Fields: Theory and Computation: The meeting point of number theory, computer science, coding theory and cryptography*, volume 477. Springer Science & Business Media, 2013.

[SI11] Werner Schindler and Kouichi Itoh. Exponent Blinding Does Not Always Lift (Partial) Spa Resistance to Higher-Level Security. In Javier Lopez and Gene Tsudik, editors, *Applied Cryptography and Network Security*, volume 6715, pages 73–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[Wu97] Thomas Wu. The Secure Remote Password Protocol. 1997.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. page 21, 2017.