Time in Cryptography: From Clock Cycles to Eternity

rkm0959

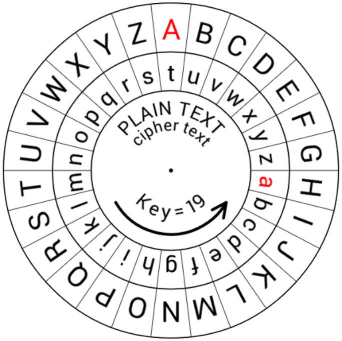
July 24th

Outline

Table of Contents

Introduction

What do you think when you hear the word “cryptography”?


PLAIN TEXT
cipher text
Key = 19
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
t u v w x y z a b c d e f g h i j k l m n o p q r s

Common Themes

The goal here is to send a message securely. Even if an adversary captures the encrypted text, or the *ciphertext*, they should have a hard time trying to figure out the actual text, or the *plaintext*.

How long should the message remain secure? *For an eternity, if possible*.

The Topic of Today's Seminar

We think that cryptography is about *eternal* security, and therefore about *eternal time*. This is usually the case - for example, RSA2048 will take an *eternal time* to attack without a quantum computer.

However, there are other topics in cryptography that deals with shorter periods of time. In this seminar, we will look at

- Attacking Cryptosystems from "Clock Cycle Time"
- Designing Cryptosystems for "Minutes, Hours and Days"

Table of Contents

Textbook RSA

Let $n = pq$ be a product of two large primes.
Set e such that $\gcd(e, \phi(n)) = 1$ and find $d \equiv e^{-1} \pmod{\phi(n)}$.
The public key is (n, e), and the secret key is d.

We encrypt a message m as $c \equiv m^e \pmod{n}$.

We decrypt a message c as $m \equiv c^d \pmod{n}$.

The security depends on the computational hardness of factorization.

Kocher's Timing Attack [Koc96]

Consider we have a decryption oracle. How to find d?

Most use binary exponentiation to compute $c^d \pmod{n}$.
Clearly d is odd. We utilize the following difference.

- If $d \equiv 1 \pmod{4}$, $c \cdot c^2$ is never calculated
- If $d \equiv 3 \pmod{4}$, $c \cdot c^2$ is calculated

We can differentiate these two with statistics. Recover d bit by bit.

Textbook CRT-RSA

Doing everything $\pmod{n}$ is slow. In practice, we utilize CRT for speed.
Let $d_p \equiv d \pmod{p-1}$, $d_q \equiv d \pmod{q-1}$.
We may decrypt c by computing

$$m_p \equiv c^{d_p} \pmod{p}, \quad m_q \equiv c^{d_q} \pmod{q}$$

then combining these with CRT to get m.

Attack on OpenSSL's CRT-RSA [BB05]

Since we do not know p, q the approach of [Koc96] doesn't work. However, there are still two things we can use for a timing attack

- Montgomery Reduction for fast modular reduction
- Karatsuba vs Naive algorithm for multiplication

Note that getting either top half bits or bottom half bits of p or q is enough to factorize N, which is due to Coppersmith's Attack.

Montgomery Reduction

Fix R, a power of 2. The Montgomery form of a is $aR \pmod{N}$.
To multiply a, b written in Montgomery form, we do

$$abR \equiv (aR) \cdot (bR) \cdot R^{-1} \pmod{N}$$

The idea is that multiplication by R^{-1} can be done fast.

REDC Algorithm

Let R, N be coprime integers with $N' = -N^{-1} \pmod{R}$.
Given $T \in [0, RN)$, we may compute $S \equiv TR^{-1} \pmod{N}$ as follows.

REDC Algorithm

```
$m \leftarrow TN' \pmod{R}$
$t \leftarrow (T + mN)/R$
if $t \geq N$ then
    return $t - N$
else
    return $t$
end if
```

The Extra Reduction

The final branch is definitely an attack vector.
We call this final subtraction by N as "extra reduction".
Schindler proved that the probability of this extra reduction is

$$\frac{(g \bmod q)}{2R}$$

when we are computing $g^d \pmod{q}$ via binary exponentiation.

Multiplication Algorithm

OpenSSL uses two algorithms for multiplication.

- If two integers have equal number of words, Karatsuba is used.
- If two integers have different number of words, naive algorithm is used.

We know Karatsuba multiplication is faster, which is an issue.
If $g \pmod q$ is small, naive algorithm will be used many times.

The two effects of $g \pmod q$ on execution time counteracts each other.
Unfortunately, (or fortunately?) the attack works regardless.

TPM-FAIL (CVE-2019-11090, CVE-2019-16863)

In ECDSA, we choose a nonce k and compute kG - an elliptic curve scalar multiplication. Unless implemented in constant time, small k implies faster runtime. However, biased nonce can lead to *Lattice Attacks*.

See rkm0959's Lattice Survey for more info. Details omitted.

Also interesting is the Raccoon Attack, if you want to check out more.

CacheBleed [YGH16]

Some attacks utilize the *cache* for timing attacks.
The cache is divided into banks, and conflict & delay occurs when multiple requests are concurrently made onto a single cache bank.

This can be used to find the exponents in windowed exponentiation.

Note that knowing high ratio of random bits in RSA private key is enough to recover the full private key, which is due to [HS09].

- My recent challenge on WACon Finals 2022 is also on this attack!

Constant Time Algorithms

To stop these timing attacks, we need methods that take constant time. We will take a brief look at the following algorithms.

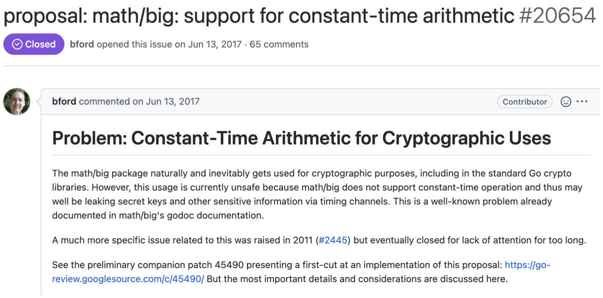
- Constant Time Arithmetic over $\mathbb{Z}$.
- Bit-Slicing Techniques for Constant-Time Cryptography
- Constant Time Sorting Algorithms
- Constant Time Discrete Gaussian Sampling

Constant Time Arithmetic over $\mathbb{Z}$

Clearly important, as $\mathbb{Z}$ operations are done everywhere.
In this seminar, we'll look at *saferith*, a constant time $\mathbb{Z}$ in Go.

$\mathbb{Z}$ in Golang

The problem was that Golang’s cryptography library used its bigint library, which was focused more on performance, hence not implemented in constant time. This was brought up in issue #20654 in golang’s github.



proposal: math/big: support for constant-time arithmetic #20654

Closed bford opened this issue on Jun 13, 2017 · 65 comments

bford commented on Jun 13, 2017 Contributor

Problem: Constant-Time Arithmetic for Cryptographic Uses

The math/big package naturally and inevitably gets used for cryptographic purposes, including in the standard Go crypto libraries. However, this usage is currently unsafe because math/big does not support constant-time operation and thus may well be leaking secret keys and other sensitive information via timing channels. This is a well-known problem already documented in math/big's godoc documentation.

A much more specific issue related to this was raised in 2011 (#2445) but eventually closed for lack of attention for too long.

See the preliminary companion patch 45490 presenting a first-cut at an implementation of this proposal: https://go-review.googlesource.com/c/45490/ But the most important details and considerations are discussed here.

Design Choices of *saferith*

saferith authors assume the following

- loops leak the number of repetitions
- conditionals leak which branch was taken
- memory accesses leak the index accessed

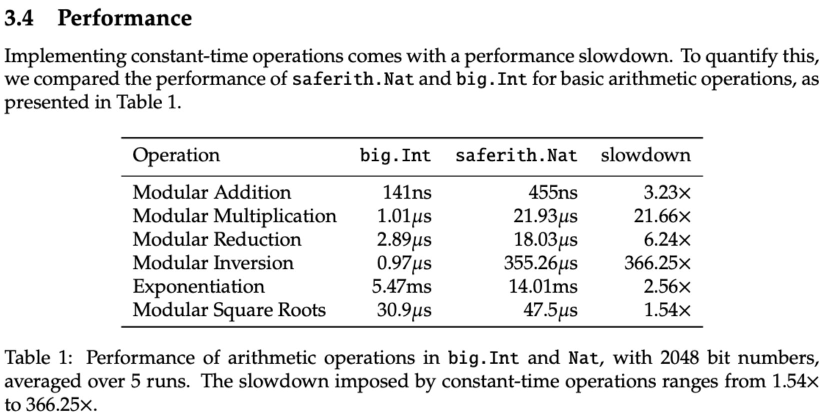
Each number has an *announced length*, practically a bound on its bit length, which is public. In some cases, a number may publicly show their *true length*, especially when their bit length is public.

For example, in RSA, both N and $\phi(N)$ has public bit length.

Techniques of *saferith*

- Implement selectors with bit operations
- If a branch leads to two different computation with varying time, do **both** then select which to use *afterwards*
- Loop only on **announced length**, and pad with zeroes if neccesary
- For modular inverses, use the optimized binary GCD by [Por20]

Benchmarks 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×.

Taken directly from the paper from the authors of *saferith*.

Bit-Slicing for Constant Time AES

Idea: express everything with 1 bit logical operations.
Bitslice each variable to achieve parallel computation and constant time.

For example, in a block cipher *AES*, there are *SubBytes* operations which are practically substitutions, with *S-box* over $\{0, 1, \cdots, 255\}$.

If we precompute this substitution table, cache attacks are a threat.

Bit-Slicing for Constant Time AES

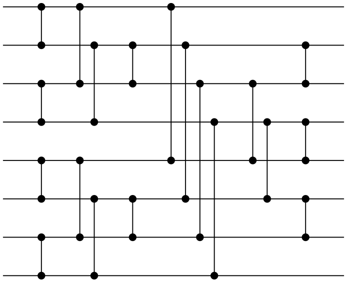
Fortunately, the S-box is composed of inverses over $GF(2^8)$ and affine transformations over $\mathbb{F}_2^8$, which allow efficient bit-slicing.

For details, check out [RSD06, MN07, Kön08, KS09, AP20].

djbsort: A Constant Time Sorting Algorithm

Goal: execution time is independent of the contents of the array.

Idea: Write a comparator of two integers. Utilize sorting networks.



For example, batcher's odd-even mergesort does this in $\mathcal{O}(n \log^2 n)$.

djbsort: Benchmarks & Security

With no branches, djbsort is constant time and has no secret branches.
djbsort is a part of several NIST submissions for PQC.

djbsort is quite fast as well, with 4929 CPU cycles to sort 768 int32 values.
Detailed benchmarks can be found at **djbsort's documentation**.

djbsort: Software Verification

Symbolic Execution with **angr** and *peephole optimizer* to reduce the software into a series of min/max operations and comparisons.

Now it suffices to prove that these operations can sort values properly. This is a hard problem for sorting networks, see e.g. zero-one principle.

djbsort: Software Verification

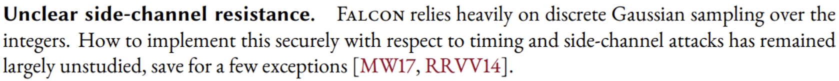
We use that djbsort is based on *2-way-merging* for an efficient proof.

First, decompose the operations into a series of 2-way-merging programs.
Then we verify each 2-way-merging programs efficiently.
This can be done with $\mathcal{O}(n)$ input cases, due to Even-Levi-Litman (2009).

Constant Time Discrete Gaussian Sampling of [HPRR20]

Sampling $\mathbb{Z}$ over discrete Gaussian is a key subroutine in Lattices.
For example, FALCON, a PQC signature scheme, uses this extensively.

In the early days of FALCON, constant-time sampling was still open.


Unclear side-channel resistance. FALCON relies heavily on discrete Gaussian sampling over the integers. How to implement this securely with respect to timing and side-channel attacks has remained largely unstudied, save for a few exceptions [MW17, RRVV14].

Constant Time Discrete Gaussian Sampling of [HPRR20]

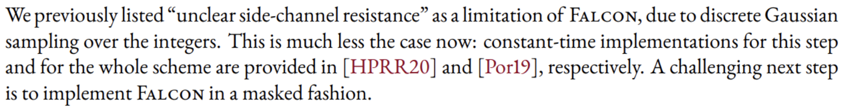
Roughly speaking, [HPRR20] combines the following ideas.

- Bound $\sigma \in [\sigma_{\text{min}}, \sigma_{\text{max}}]$. This can be done due to context of FALCON.
- Compute $\exp(x)$ with polynomial approximations in constant time.
- Sample over $\mathcal{D}_{\mathbb{Z}^+, \sigma_{\text{max}}}$ with cummultative distribution table.
- Transform from $\mathcal{D}_{\mathbb{Z}^+, \sigma_{\text{max}}}$ to $\mathcal{D}_{\mathbb{Z}, \sigma, \mu}$ via rejection sampling.
- Renyi Divergence argument for security proofs of the sampler.

We note that $\sigma_{\text{max}} \approx 1.8205$ in the context of FALCON.

Happy Ending for FALCON!

Now constant-time sampling is studied, and FALCON is a NIST standard!


We previously listed "unclear side-channel resistance" as a limitation of Falcon, due to discrete Gaussian sampling over the integers. This is much less the case now: constant-time implementations for this step and for the whole scheme are provided in [HPRR20] and [Por19], respectively. A challenging next step is to implement Falcon in a masked fashion.

HertzBleed: Clock Cycle is Not Enough

Recently, a new side channel attack was revealed.

DVFS system for CPU may adjust frequency based on the power used.
Therefore, we can turn a power-based side-channel to a timing-based one.

HertzBleed authors used this to attack some implementations of SIKE.
This is due to the fact that SIKE was vulnerable to zero-value attacks.

Table of Contents

Introduction: A Brief Look in MEV

In Ethereum, transactions have the following properties

- miners may order transactions however they want
- transactions that are not yet mined are still public in mempool

Therefore, if A wants to buy ETH, an attacker B may *sandwich* A.

- B sees that A is buying ETH by watching the mempool.
- B buys ETH *before* A buys, then sells *after* A buys.
- B does this by paying a large *tx fee* to the miner.

Verifiable Delay Functions: A Proof of Time

To prevent this, it would be nice to check that B wanted to buy ETH *before* A’s buy transaction joined the mempool. How do you even prove that?

Verifiable Delay Functions is a cryptographic solution. We need

- A computational task on an input that takes time T to find the answer
- An efficient algorithm for the *verification* of the solution

Now we force B to submit the VDF of the tx data and verify it on-chain. For an implementation of this idea, check SlowSwap (ETH Denver 2022).

Design Consideration 1: Parallel Computation

This is not *proof of work*, this is *proof of time*.

Therefore, we cannot use standard hash-based PoW used in bitcoin.
We want our task to be *sequential*, and it should "resist" parallelism.

Design Consideration 2: Optimized Software & Hardware

In competitive programming, people have hard time distinguishing model solution from unintended solution due to constant optimization techniques.

Similar ideas apply here - if we want T to be 1 hour, then it should be 1 hour, not off by a constant. Therefore, our task must be based on operations that are already optimized in the software & hardware level.

VDF based on Groups of Unknown Order [BBF18]

Note: [BBF18] is a *survey paper*.

We use multiplication over $\mathbb{Z}_N^*$ as our base operation.

Consider an RSA modulus N. Now given g, computing

$$h = g^{2^T} \pmod N$$

is a task that requires T multiplications.

This does require *trusted setup* since no one should know the factors of N. To avoid this, we may use class groups of imaginary quadratic fields.

VDF based on Groups of Unknown Order [BBF18]

In Wesolowski's argument, verifier selects random prime ℓ.
The prover also calculates $\pi = g^{\lfloor 2^T/\ell \rfloor}$, and verifier checks

$$h = \pi^\ell g^r, \quad r = 2^T \bmod \ell$$

VDF based on Groups of Unknown Order [BBF18]

In Pietrzak's argument, we compute $v = g^{2^{T/2}}$ and prove

$$h = v^{2^{T/2}}, \quad v = g^{2^{T/2}}$$

Now verifier chooses random r, then we now simply prove

$$(v^r h) = (g^r v)^{2^{T/2}}$$

This "random combination" idea is very common in cryptography.

VDF based on Hash Functions

An alternate method, albeit a bit hard to verify, is a hash chain.
We base our VDF on hash functions which have very optimized hardware.

We simply compute iterative hashes of the given input, i.e. $h = H^k(g)$.

This is used in Solana for “proof of history”, i.e. decentralized time.

Short-Lived Zero Knowledge & Signatures [ABC22]

One may use VDFs to create a signature that loses its value after time T.

The rough idea is to give to prove that the creator of the signature

- knew the private key of A, or
- spent time T to compute a VDF on a value released at time t

The signature can be a proof that A signed the message during $[t, t+T]$, but not afterwards. Hence the name "Short-Lived Signatures".

This can be used to create deniable messaging - for more info see DKIM.

Table of Contents

References

- [Koc96]: `https://paulkocher.com/doc/TimingAttacks.pdf`
- [BB05]: `https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf`
- TPM-FAIL: `https://tpm.fail/`
- [YGH16]: `https://eprint.iacr.org/2016/224.pdf`
- saferith: `https://eprint.iacr.org/2021/1121.pdf`
- [Por20]: `https://eprint.iacr.org/2020/972.pdf`
- [AP20]: `https://eprint.iacr.org/2020/1123.pdf`
- djbsort: `https://sorting.cr.yp.to/index.html`
- [HPRR20]: `https://eprint.iacr.org/2019/1411.pdf`
- SlowSwap: `https://github.com/SlowSwap`
- [BBF18]: `https://eprint.iacr.org/2018/712.pdf`
- [ABC22]: `https://eprint.iacr.org/2022/190.pdf`