Constant Time Big Numbers
(For Go)
Lúcás C. Meier
Supervisor: Prof. Bryan Ford
Overview

- Big Numbers?
- Timing Attacks?
- Go?
- Safenum (Our Work)
- Further Work
Big Numbers
Useful in Cryptography

- \( \mathbb{N} \) (Natural Numbers)
- \( \mathbb{Z}/\mathbb{N}\mathbb{Z} \) (Modular Arithmetic)
- \( \mathbb{F}_p \) (Prime Fields)
RSA

Public key \((e, N)\), encrypt \(m\) with:

\[ m^e \mod N \]

\(N \approx 2048\) bits
Too Big!
Elliptic Curve Cryptography
Prime Fields!

\[ \mathbb{Z}/p\mathbb{Z} \]

for example:

\[ p = 2^{255} - 19 \]

Somewhat big
Implementation Strategies

- Hand-written implementation
- Generated (e.g. FiatCrypto)
- Dynamic (big.Int, our library)
Timing Side Channels
Implementations in Theory
Implementations in Practice
Timing
Guessing Passwords

1 2 3 4
1 2 6 4
Side-Channel Overview

Subtle Behavior:

- Caches
- Branch Prediction
- Microcode Pipelines
Further Information

A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware

Qian Ge¹, Yuval Yarom², David Cock¹-³, and Gernot Heiser¹

¹Data61, CSIRO and UNSW, Australia, qian.ge, gernot@data61.csiro.au
²Data61, CSIRO and the University of Adelaide, Australia, yval@cs.adelaide.edu.au
³Present address: Systems Group, Department of Computer Science, ETH Zürich, david.cock@inf.ethz.ch
Threat Model

- Loops leak the number of iterations
- Memory accesses leak addresses
- Branching leaks condition
Constant-Time Computing Base

- Addition +
- Multiplication *
- Logical Operations |, &, ^
- Shifts <<, >>
Go
Package big

import "math/big"

Overview

Package big implements arbitrary-precision arithmetic (big numbers). The following numeric types are supported:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>signed integers</td>
</tr>
<tr>
<td>Rat</td>
<td>rational numbers</td>
</tr>
<tr>
<td>Float</td>
<td>floating-point numbers</td>
</tr>
</tbody>
</table>
go/crypto
We use big.Int for Cryptography
It's constant-time right?
It's constant-time right?
Not Constant-Time

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.
Why? Bad Algorithms

```go
math > big > 🟢nat.go

237 }  
238 if c != 0 {  
239 | subVV(z[:n], z[n:], m)  
240 } else {  
241 | copy(z[:n], z[n:])  
242 }  
243 return z[:n]  
244 }
```
Why? Padding

0  0  5  9

5  9
in go/crypto

- Extensively in **RSA**, and **DSA**
- **ECC**: Elliptic Curve interface uses `big.Int`
- Only **P384** uses `big.Int` for field arithmetic
Mitigations

In RSA: *blinding*:

Instead of:

\[ c^d \mod N \]

Calculate:

\[ \frac{1}{r} (c \cdot r^e)^d \mod N \]
There be Dragons?

```go
crypto > rsa > 🔄 rsa.go > DecryptOAEP

616
617  // We probably leak the number of leading zeros.
618  // It's not clear that we can do anything about this.
619  em := m.FillBytes(make([], byte, k))
```

Our Library

cronokirby/safenum
Constant time big numbers for Go

Contributors: 2
Issues: 0
Stars: 66
Forks: 3

github.com
Operations

- Modular addition, subtraction, exponentation, etc.
- Modular square roots
- “Raw” addition and multiplication
Constant-Time Choice
Performance: Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>op / s (big.Int)</th>
<th>op / s (Nat)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>10,980,842</td>
<td>12,164,599</td>
<td>0.90</td>
</tr>
<tr>
<td>Modular Addition</td>
<td>6,986,739</td>
<td>3,075,188</td>
<td>2.27</td>
</tr>
<tr>
<td>Multiplication</td>
<td>1,316,322</td>
<td>542,385</td>
<td>2.43</td>
</tr>
<tr>
<td>Modular Reduction</td>
<td>454,917</td>
<td>63,253</td>
<td>7.19</td>
</tr>
<tr>
<td>Modular Multiplication</td>
<td>1,000,000</td>
<td>44,596</td>
<td>22.42</td>
</tr>
<tr>
<td>Modular Inversion</td>
<td>1,000,000</td>
<td>621</td>
<td>1610</td>
</tr>
<tr>
<td>Modular Exponentiation</td>
<td>223</td>
<td>86</td>
<td>2.59</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>op / s (big.Int)</th>
<th>op / s (Nat)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sqrt{z} \mod p_3$</td>
<td>40,464</td>
<td>26,886</td>
<td>1.50</td>
</tr>
<tr>
<td>$\sqrt{z} \mod p_1$</td>
<td>-</td>
<td>7,867</td>
<td>-</td>
</tr>
</tbody>
</table>
Performance: Cryptography

<table>
<thead>
<tr>
<th>Operation</th>
<th>op / s (big.Int)</th>
<th>op / s (Nat)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA Decrypt</td>
<td>670</td>
<td>312</td>
<td>2.15</td>
</tr>
<tr>
<td>RSA Sign</td>
<td>675</td>
<td>372</td>
<td>1.81</td>
</tr>
<tr>
<td>RSA Decrypt (3 Prime)</td>
<td>1173</td>
<td>596</td>
<td>1.97</td>
</tr>
<tr>
<td>DSA Sign</td>
<td>6202</td>
<td>2625</td>
<td>2.36</td>
</tr>
<tr>
<td>DSA Parameters</td>
<td>0.89</td>
<td>1.64</td>
<td>0.54</td>
</tr>
</tbody>
</table>
Patching RSA

crypto/rsa: replace `big.Int` for encryption and decryption

Infamously, `big.Int` does not provide constant-time arithmetic, making its use in cryptographic code quite tricky. RSA uses `big.Int` pervasively, in its public API, for key generation, precomputation, and for encryption and decryption. This is a known problem. One mitigation, blinding, is already in place during decryption. This helps mitigate the very leaky exponentiation operation. Because `big.int` is fundamentally not constant-time, it's unfortunately difficult to guarantee that mitigations like these are completely effective.

This patch removes the use of `big.Int` for encryption and decryption, replacing it with an internal nat type instead. RSA signing is also affected, because it depends on encryption.
Timeline

February
- Initial Commit

March
- API Designed
- Serialization
- Add, Mul, Exp
- ModInv

April
- Faster (Montgomery)

May
- Test on RSA

June
- RSA Patch
The most important artifact?
Understanding!
Further Work

- Verifying security properties
- Improving performance: Assembly?
- More scenarios: ECC, PQC?
In Summary

We made an alternative to big.Int for Cryptography. It’s only 2x slower.