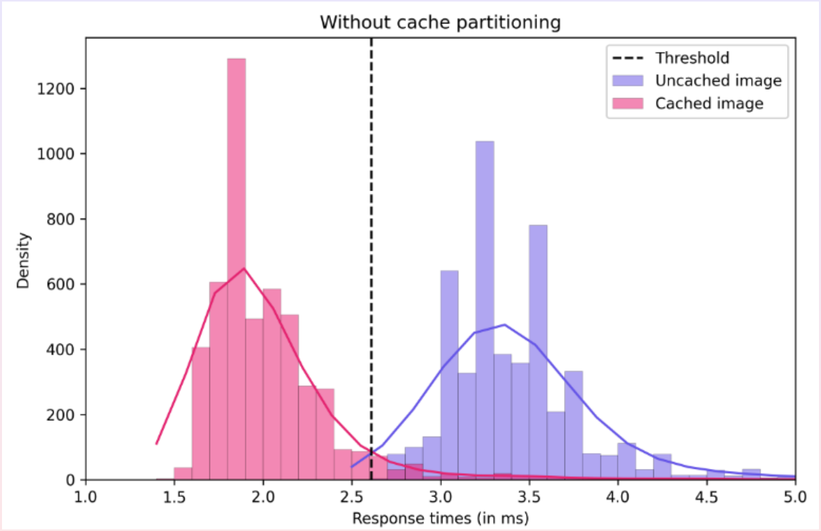
DOTA2024:10 Defense of the Ancients Tenth topic - Side channels, Future?

Hugh Anderson

National University of Singapore
School of Computing

July, 2024

Cached or not?



Outline

1. **Side Channels**
   - Exponentiation attack
   - Indirect timing (OpenSSL) attack
   - Fault injections
2. **Maybe in the future...**
   - Preventing DOS using proof-of-work
   - Quantum computation
   - Quantum “cryptography”

Outline

1. **Side Channels**
   - Exponentiation attack
   - Indirect timing (OpenSSL) attack
   - Fault injections

2. **Maybe in the future...**
   - Preventing DOS using proof-of-work
   - Quantum computation
   - Quantum “cryptography”

Outline

1. **Side Channels**
   - Exponentiation attack
   - Indirect timing (OpenSSL) attack
   - Fault injections
2. **Maybe in the future...**
   - Preventing DOS using proof-of-work
   - Quantum computation
   - Quantum “cryptography”

Timing attack on exponentiation

Efficient exponentiation

The first example of a timing attack on exponentiation (as in RSA) was introduced by Paul Kocher in 1995, while he was an undergraduate at Stanford. The attack attempts to find the key k, and exploits how efficient exponentiation is carried out.

Note that RSA does repeated multiplication, and ECC/Elgamal use point addition, an analog of exponentiation. The algorithms use doubling and look the same:

RSA

```
To compute $c^d \bmod N$ in RSA
Q = 1;
foreach bit in key d {
    Q = Q*Q mod N;
    if (bit==1) {
        Q = Q*c mod N;
    }
}
return Q;
```

ECC

```
To compute $kP$ in ECC
Q = O;
foreach bit in key k {
    Q = 2Q;
    if (bit==1) {
        Q = Q + P;
    }
}
return Q;
```

Attacker's capability and goal

Assumptions on the attacker's capabilities

Consider the RSA algorithm, and multiplication. For any 2 integers x, y, the adversary can determine the time taken to compute $x \times y \bmod N$. The adversary can send any c to the decryption oracle and measure the time taken by the oracle. That is, the time taken to compute $c^d \bmod N$

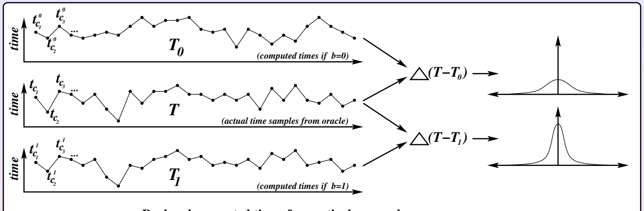
The main idea

First, note that for different values of the bits in the key, the loop takes different times. It takes longer if the bit is a 1.

The attacker sends large numbers of randomly drawn ciphertexts to the oracle, and measures the time taken for each computation. The attacker simulates the behaviour of the oracle for $b_0 = 0, b_0 = 1$ and tries to find timing correlations with the actual measured time. Once it has a best guess for b_0 it moves on to b_1 and so on through the bits, an iterative process.

`https://paulkocher.com/doc/TimingAttacks.pdf`

The attack



Real and computed times for particular samples

Midway through the attack

The attacker knows $b_0, \ldots b_4$, and is trying to discover b_5 by repeatedly querying the oracle with a new c, and getting the time t. The adversary then

- Determines $t_{0\ldots 4}$, with knowledge of the algorithm, and $b_0, \ldots b_4$.
- Determines t_5^0, the time if the bit in the key was 0.
- Determines t_5^1, the time if the bit in the key was 1.

Now either $t = t_{0\ldots 4} + t_5^0 + x$ or $t = t_{0\ldots 4} + t_5^1 + x$ (along with noise). The attacker builds two distributions T_0 and T_1. If the oracle's actual-time distribution T correlates better with T_0, then $b_5 = 0$, otherwise $b_5 = 1$.

The algorithm continues until all bits of the key are retrieved.

What do we do about this attack?

The Montgomery ladder:

Both functions return the same result as before, but now they run in a constant time. Note that RSA still does multiplication, and Elgamal still uses point addition, and the algorithms still use doubling and look the same:

RSA

```
To compute $c^d \bmod N$ in RSA
R0 = c;
R1 = c * c mod N;
foreach bit in key d {
    if (bit==0) {
        R1 = R0 * R1 mod N;
        R0 = R0 * R0 mod N;
    } else {
        R0 = R0 * R1 mod N;
        R1 = R1 * R1 mod N;
    }
}
return R0;
```

ECC

```
To compute $kP$ in ECC
R0 = O;
R1 = P;
foreach bit in key k {
    if (bit==0) {
        R1 = R0 + R1;
        R0 = 2R0;
    } else {
        R0 = R0 + R1;
        R1 = 2R1;
    }
}
return R0;
```

Outline

Attack involves ECC, memory caches, timing...

OpenSSL: An important bit of security software

OpenSSL is used in about 60% of all web based ssl (https) servers. As a result of this we are *very* interested in bugs in openssl. It may be the software most closely examined for security errors - all 700,000 lines of code!

What is the (one second) attack?

From the paper at hrefhttp://eprint.iacr.org/2014/140.pdfhttp://eprint.iacr.org/2014/140.pdf .

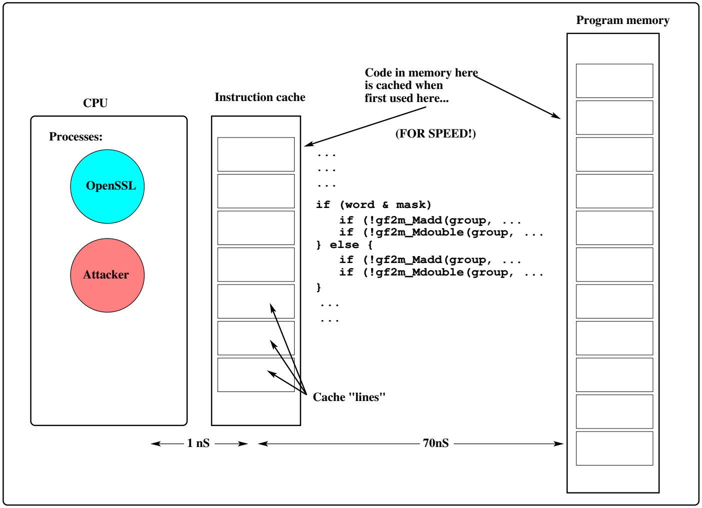
> *"Our attack recovers the scalar k and thus the secret key of the signer and would therefore allow unlimited forgeries."*

The attack is a side channel attack. An attacker can determine if code has been cached or not. If it has been used, then it will be cached.

OpenSSL is built with security in mind, so (of course) it uses the Montgomery ladder, that runs in constant time. Weirdly, the attack is a *timing* attack (on the cache for the Montgomery ladder code).

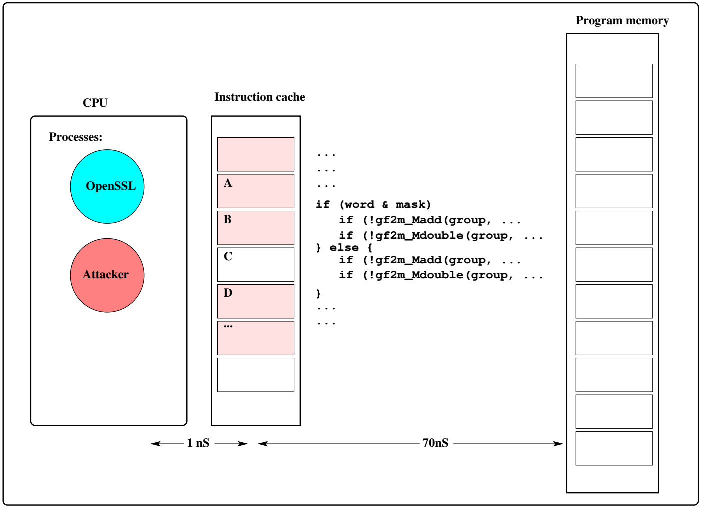
Attack is directed at the instruction cache...

As openssl program runs, it caches the code...



```
...
...
...
if (word & mask)
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
} else {
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
}
 ...
 ...
```

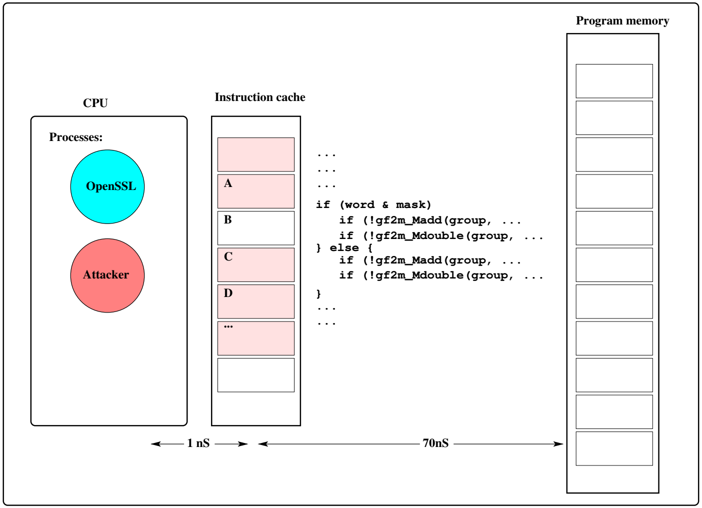
If bit is a “1” THIS is cached...



```
...
...
...
if (word & mask)
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
} else {
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
}
...
...
```

(and the attacker can determine this by seeing how fast the four cachelines A,B,C,D are loaded)

If bit is a “0” THIS is cached...



```
...
...
...
if (word & mask)
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
} else {
    if (!gf2m_Madd(group, ...
    if (!gf2m_Mdouble(group, ...
}
...
...
```

(and the attacker can determine this by seeing how fast the four cachelines A,B,C,D are loaded)

OpenSSL timing

Original openssl-1.0.1f/crypto/ec/ec2_mult.c...

```
if (word & mask) {
   if (!gf2m_Madd(..., x1, ... ) goto err;
   if (!gf2m_Mdouble(..., x2, ... ) goto err;
} else {
   if (!gf2m_Madd(..., x2, ... ) goto err;
   if (!gf2m_Mdouble(..., x1, ... ) goto err;
}
```

Each bit of a key is being tested, and depending on the bit we take two different paths. The paths operate on different (code) memory locations which would be cached, and an attacker can (afterwards) discover that, because that memory location would load faster. An attacker can determine the value of each bit.

Fixed openssl-1.0.1g/crypto/ec/ec2_mult.c...

```
BN_consttime_swap(word & mask, x1, x2,...
if (!gf2m_Madd(..., x1, ... ) goto err;
if (!gf2m_Mdouble(..., x2, ... ) goto err;
BN_consttime_swap(word & mask, x1, x2,...
```

A constant time swap to put variables in the same memory locations.

Outline

Fault injection

The idea of getting secrets through injecting faults

This idea was first discussed in D Boneh, RA DeMillo, RJ Lipton, *"On the importance of checking cryptographic protocols for faults"*, CRYPTO, 1997.

The adversary observes the behavior of a device when some components are faulty. From the behavior, the adversary infers a secret.

The fault could be injected by the adversary. Very often, although the adversary can inject a fault, the adversary cannot predict the outcome. For example, the adversary can flip some bits in a particular register but does not know which bits will be flipped and be unable to read the register.

Using CRT in RSA to speed up decryption about $4\times$

Rather than computing $c^d \bmod N$, it is possible to instead compute $e_1 = c^d \bmod p$, and $e_2 = c^d \bmod q$, and then

$$\boxed{c^d \bmod N = (ae_1 + be_2) \bmod N}$$

where $a = 1 \bmod p, a = 0 \bmod q, b = 0 \bmod p, b = 1 \bmod q$ (pre-computed). This is possible because the factorization of N could be kept around.

Fault injection

If exponentiation is using CRT...

Assume the attacker can obtain the output $m = (ae_1 + be_2) \bmod N$.

The attacker can introduce noise/errors during the computation of e_1, and no error during the computation of e_2. Hence the attacker obtains

$$m' = (ae'_1 + be_2) \bmod N$$

Now, the attacker computes $\gcd(m–m', N)$. The gcd is either p or q.

Why does it work?

Note that $m - m' = a(e_1 - e'_1) \bmod N$, and that $a = 0 \bmod q$ (but is not zero). If $m–m'$ is not divisible by p, then

$$\gcd(m - m', N) = q$$

Side channel attacks

Summary

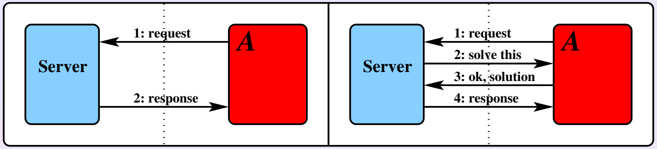
These attacks are very real, and here we have only looked at a few of the known attacks. In general side-channel attacks are implementation or system dependant, not related to the *hardness* of the (crypto/security) problem. Caches, timing, power consumption, sound, light, radio waves, faults and so on - all have been used.

In any case they have been shown to achieve success with vastly smaller number of trials/attacks, than would be expected through (for example) root finding, discrete logs or factorization.

Though the underlying mathematical structures seem good, to implement security systems we must think out of the box, like the attackers, and handle unexpected side channel attacks.

Outline

Proof-of-work based prevention of DOS

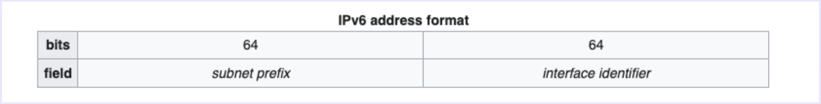


NDSS 1999

In a typical Denial of Service attack, the attacker invests relatively less resource in issuing a request, but the victim has to spend much more resources in answering the request. The puzzle based method tries to flip this asymmetry. An attacker needs to submit a proof-of-work before the server serves the request.

The puzzles should be easy to construct, easy to verify, but difficult to solve. One way to construct a puzzle might be to choose a random string s, and force the attacker to find a string r s.t. the first (or last) 20 bits of the digest $\mathscr{H}(r + s)$ are all zeros.
The attacker is forced to carried out an expected number of 2^{20} hash operations (?) to find such a choice of r.

Cryptographically Generated Addresses: CGA

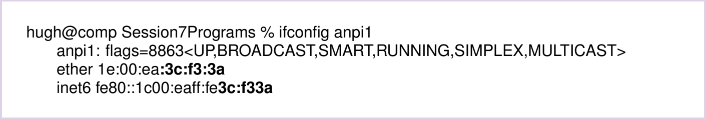


IPv6 address format

bits	64	64
field	*subnet prefix*	*interface identifier*

Cryptographically generated addresses in IPv6

The interface identifier is either automatically generated from the interface's MAC address, obtained from a DHCPv6 server, automatically established randomly, or assigned manually.



```
hugh@comp Session7Programs % ifconfig anpi1
    anpi1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST>
    ether 1e:00:ea:3c:f3:3a
    inet6 fe80::1c00:eaff:fe3c:f33a
```

`fe80::1c00:eaff:fe3c:f33a` is shorthand for `fe80:0000:0000:0000:1c00:eaff:fe3c:f33a`.

RFC3972 (CGA) binds a public key to the interface identifier, with the idea that In order for an attacker to masquerade as a host with that CGA, the attacker must provide a proof-of-work: in particular, find a hash collision!

For RFC3972/CGA, see `http://www.rfc-base.org/rfc-3972.html`
or (Wikipedia) `https://en.wikipedia.org/wiki/Cryptographically_Generated_Address`

Outline

Quantum physics

First interest: Quantum computing...

1. Quantum *computers* may be able to compute HARD problems quickly (such as factorizing large composites).

How? The underlying data elements are quantum bits (qubits), not limited to just 0,1 states - instead considered to be a superposition of states. An operation performed on a qubit is performed on all the states simultaneously. Shor's algorithm.

DWave systems have sold the first commercial quantum computer, with (evidently) 128 qubits. However, it is unable to perform Shor's algorithm: `http://www.dwavesys.com/`
It is likely that no effective quantum computer has yet been built that could factor a large composite.

Shor's Algorithm

A very slow way to find factors of $p \times q$?

Choose some number a, with no factors in common with pq[a]. Imagine you calculate a^2, a^3, a^4, all modulo pq, until the sequence repeats for the first time (perhaps after r steps). The repetition value r evenly divides $(p-1)(q-1)$. Have a look at the Euler table in slide set 9 - it shows the powers modulo $15 = pq = 3 \times 5$, and all repetitions divide $(p-1)(q-1) = 8$.

In addition, $a^r \equiv 1 \mod pq$. , and since this was the first repetition, $b = a^{\frac{r}{2}} \not\equiv 1 \mod pq$. But, $b^2 \equiv 1 \mod pq$.

We have four square roots of 1: ± 1 and $\pm b$. If we know b, we can calculate the factors of pq (as we will see again in oblivious transfer).

[a]Of course if your number a does have factors in common with pq, you have either discovered p or q.

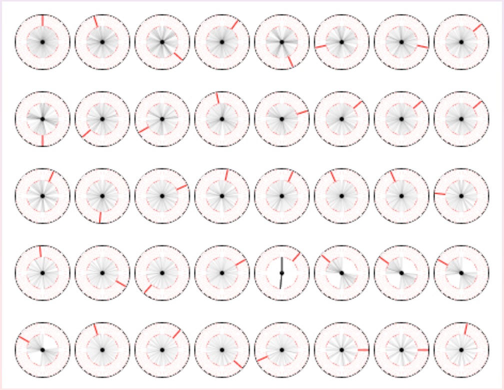
It is slow because...
...the sequences may be very very long. So this is not some secret fast algorithm if you have to calculate long sequences a^2, a^3, a^4, all modulo pq.

Shor’s Algorithm

Find repetition values for all powers simultaneously:

A Quantum register holds all possible values at the same time, and we perform an amplifying operation, that only leaves stable repetition values.

Mark the same times every day... In the simulation, only the clock with a repetition rate that we are interested in remains:



Outline

Quantum “cryptography”

Second interest: Quantum “cryptography”

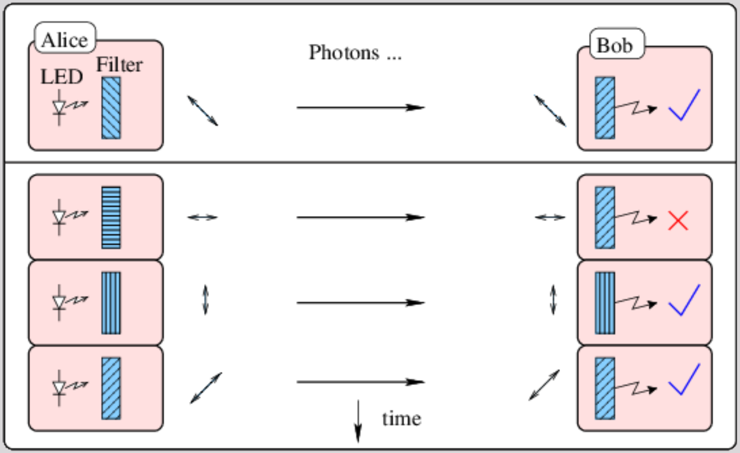
2. Quantum *“cryptography”* uses laws of quantum mechanics - Heisenberg Uncertainty applies to some pairs (of properties) of (atomic) particles. Measuring one property affects another.

A snooper is easily detected, and there are various protocols for using quantum effects to share keys.

BB84: Key sharing protocol
Alice randomly chooses one of four polarizations: rectlinear: (0,90), or diagonal (45,135) (degrees).

Alice transmits 10000 photons

BB84: Encoded using different polarizations:



Alice and Bob’s protocol

BB84: choose bits - no reveal

1. Alice records what she has sent. Bob randomly chooses polarizations, and for each one reads the resultant value. (If he chooses correctly, gets a valid 1 or 0, if not randomly gets 1 or 0).
2. Bob tells Alice the polarizations he has used: diag, diag, rectilinear, diag
3. Alice replies by telling Bob which ones were correct. (1,3,4, 8,9,10 12,17)
4. They now have 5000 (approximately) bits in common.

Harry the hacker listens in, but

BB84: Harry has a problem

1. If Harry the hacker senses some of the photons, he must choose which polarization to use, and will affect the photon.
2. Bob and Alice compare a subset of the bits that they think they know to detect snooping.
3. If no snooping, then rest of bits are likely to be OK.

Current state

Quantum “cryptography” systems are now commercially available, operating over reasonably long (40km) fibre.
Note the probabilistic nature of the algorithm. By choosing bit length can get any degree of assurance.

Where did we go?

The Attack Surfaces...

Introduction (in May):	Warfare, with an indistinct enemy.
Social Engineering (in May):	Fight human weakness with awareness.
System Complexity:	Difficulty constructing complex systems. Fight back with design principles, standards and formal methods.
Crypto:	Pre/post 1976. Strength from computational complexity.
IP Networks:	For resilience, not security. Hardening is a bit ad-hoc.
OtherNets:	Other networks that rely on security-by-obscurity.
Web Applications:	XSS, injections. Fight back with awareness/tools.
Hash functions:	Rainbow tables and collisions. Use good hash functions!
Machines:	Memory/cache issues. Fight back with DEP, ASLR etc.
Side Channels:	Timing, faults. Fight back with constant time functions and...

Quantum futures: Quantum, and the post-quantum world.

End statement...

It has been a privilege to have taught you, and I hope I have shown you a few things that are interesting to you, and help you in the future...