Using Fault Injection to weaken RSA public key verification

SNE Research Project 2
Ivo van der Elzen
University of Amsterdam
What is Fault Injection?

Simply put:

“Introducing faults in a target to alter its intended behavior”*

*(N. Timmers)*
What is Fault Injection?

- Use **physical** means to induce **logical** faults into a target
  - Electromagnetic
  - Temperature
  - Optical (laser)
  - Voltage
  - Etc.

- Can cause faults in instructions, execution flow, data.
  - Instruction corruption
  - Instruction skipping
  - Data corruption
What can Fault Injection accomplish?

- Some examples:
  - Bypassing PIN/password verification
  - Escalating privileges
  - Bypassing Secure boot
  - Extracting RSA private key, AES keys
  - Firmware extraction
  - Modifying data in memory

- We’ll be using Voltage Fault Injection to modify data

- Some excellent references I recommend to check out
  - Bellcore attack on RSA-CRT, Boneh et al. (1996)
  - Attacking RSA public modulus by Seifert (2005) and Muir (2005)
  - Low-voltage attacks on RSA and AES on ARM9 by Barenghi et al. (2009, 2010)
  - Building fault models for microcontrollers, SNE RP2, Spruyt (2012)
  - Proving the wild jungle jump, SNE RP2, Gratchoff (2015)
  - Controlling PC on ARM using Fault Injection, Timmers et al. (2017)
Attacking RSA’s public modulus

- An RSA public key consists of two values:
  - Public exponent $e$
  - Public modulus $N$
- $N$ is (usually) a product of two large prime integers
- To get the private key, we need the factorization of $N$, but this is infeasible
- If we can modify $N$, we can make it easier to factor (call this modification $N'$)
- With the factorization of $N'$, we can make a private key ($N', e, d'$)
- As long as the target uses the modified $N$, our private key will work
Voltage glitching to induce faults in data

- When copying data, we introduce a **glitch** in the supply voltage
- The processor will execute an instruction incorrectly and introduce a **fault**:

Source data:  C3B5F25715A8D1

Destination data:  C3B5F20055A8D1

We can use this to change values in an RSA public key!
The Attack

- While $N$ is being copied, induce a fault to obtain $N'$
- We factor $N'$ and create a private key $d'$
- Use $d'$ to sign a message, which verifies against $N'$

As long as the target has $N'$ in memory, the signature will be valid.
Attack example - Secure Boot

Inject voltage glitch while key is being copied:
01010110000001101

Glitch causes modification of key

Counterfeit Firmware Image

Signed with private key from factored glitched key

Decrypt signature using glitched key

Signature is validated
Research questions

- Is modifying the RSA public modulus using voltage fault injection a practical means of weakening RSA signature verification?
  - How can an RSA public modulus be modified in a way that is beneficial to an attacker?
  - Which types of modifications reliably yield factorable moduli?
  - Can we create valid private keys from these factorizations?
  - Is it practical to apply this attack against RSA?
Obtaining a fault model - Target Characterization

- Study the effects of V-FI on a memory copy
- Target device: ARM Cortex-M4F 32 bit
- Program target device to:
  - Copy data between buffers
  - Set trigger when copy starts and unset when finished
  - Return result
- Apply voltage glitch after trigger is set
- Record response and classify
  - Normal response, *green* color
  - Correct glitched response, *red* color
  - No response, *yellow* color
Experimental Setup

- Target running our test code (Riscure Piñata)
- Glitcher and glitch amplifier (Riscure Spider and GA)
- Computer
  - Control glitcher over USB
  - Control target over UART
  - Record responses from target
Experimental Setup (cont.)

1. PC oscilloscope
2. UART interface
target <-> PC
3. Target (Piñata)
4. Glitch Amplifier
5. Glitcher (Spider)
Prepare target device

- Prepare two buffers:
  - Fill `source` with 0x55
  - Fill `destination` with 0x44
  - (Normally memory is initialized with 0x00. We use 0x44 to distinguish between faults)

- Initialize unused registers to known pattern
  - C4 F4 B4 D4 for r4, C5 F5 B5 D5 for r5 etc.

- Copy `source` to `destination`

- Three variants, implemented in ARM assembly:
  - Byte-per-byte using LDRB / STRB
  - Word-per-word (4 bytes) using LDR / STR
  - Multi-word (16 bytes) using LDM / STM

- Output destination buffer over UART, bookended with 0xAA, 0xBB
Loop timing measurement

- We determine the time each loop takes using the oscilloscope
- Select glitch timings to hit the middle third (focus on area highlighted in red)
Glitch characterization, byte-wise, (229815 tests)
Glitch characterization, word-wise, (230123 tests)
Glitch characterization, multi-word-wise (231069 tests)
Refine parameters -> Fix voltage at 2.5V

- Higher success rate
- Multi-word still difficult, but shows a clear area to focus on
- Further refinement is possible
Fault Models observed - some examples

- Early Break: AA5555...55555555555555555555555555555554444444444444444BB
- Skip: AA5555...55555555555555555555555555555554444444444444444BB
- Zeroed: AA5555...555500005555555555555555555555555555555BB
- Registers: AA5555...5555DABAFACA55555555C75555555555BB
- Bitflips: AA5555...555554555555555555555555555555555...55BB (01010100)
- Mixed: AA5555...5555D7B7F7C755550000555555554444BB
- Other: AA5555...55554400230120AD2C0008152D000851...BB
Determine Fault Model

Out of 3,191,236 total tests, we observed 205,366 desired (red) glitches. These glitches are categorized and tallied as follows:

<table>
<thead>
<tr>
<th>Type of fault</th>
<th>Percentage of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early break</td>
<td>63.6%</td>
</tr>
<tr>
<td>Single skip</td>
<td>7.8%</td>
</tr>
<tr>
<td>Zeroed</td>
<td>2.2%</td>
</tr>
<tr>
<td>Other registers</td>
<td>1.5%</td>
</tr>
<tr>
<td>Flipped bits</td>
<td>1%</td>
</tr>
<tr>
<td>Other/mixed</td>
<td>23.9%</td>
</tr>
</tbody>
</table>

Number of observed glitches per type

- Break: 130,621
- Skip: 15,985
- Zeroed: 4,532
- Registers: 2,957
- Bitflips: 2,103
- Other/mixed: 49,168
Most suitable for breaking RSA

- By far the most common is an early break scenario
  - Every byte set to 0 at the end adds $2^8$ as a factor
  - In this scenario, about half of the messages fail to decrypt properly
  - RSA requires that message and $n$ are coprime
  - You could modify the message to make it work
- More suitable is a single skip
  - It’s the second most common
  - It’s predictable
  - Less likely to add repeating factors
But can we hit every single loop iteration?

- Yes, we can incur single skips in every single byte or word
  - More difficult with multi-word
- We can hit a single iteration with a probability of 95% within about 2.5 minutes.
- If a secure boot takes 10 seconds this scales up to once in every 5 hours or so.
- But we only need one hit for this attack to work!
Factoring glitched moduli

- For “normal” RSA General Number Field Sieve is currently the most efficient
- We can expect multiple smaller factors, so there is a better solution
- ECM: Lenstra’s Elliptic Curve Method
- Can find factors up to 128 bits efficiently
- We used SAGE’s implementation of ECM

SAGE: an open-source mathematics framework
Factorization testing method

Based on most suitable fault model of skipping a single loop iteration.

1. Generate a random RSA key, selecting a size between 512 and 4096 bits
2. Apply glitch to each unit of data in the key separately
3. Attempt factoring of all resulting moduli using ECM
   ○ Divide ECM threads over each core
   ○ Use a timeout to keep things manageable
4. Repeat many times with a freshly generated key each time
Results

- 1234 unique RSA keys were tried:
  - 339 512-bit keys
  - 319 1024-bit keys
  - 307 2048-bit keys
  - 269 4096-bit keys

- In total 146512 perturbations of these 1234 keys were attempted!
- Of those, 11150 were factored successfully within 60 seconds, or 7.6%
- But, **ALL** keys had at least one successfully factored perturbation
- Including every single 4096 bit key!
Factorization success rates by fault model

Please note the scale difference. Timeout used: 60 seconds.
Creating private keys from factorizations

- Private key: \( d \equiv e^{-1} \mod \phi(n) \)
- \( \phi(n) \) is easy to calculate if we know the factorization
- Usually more than two primes, different from “textbook” RSA
  - Ask me later for details if you’re interested!
- No further alterations to RSA are needed
- Also implemented this using SAGE
Key takeaways

- We’ve shown that it’s possible to reliably modify a public key using V-FI.
- Even though RSA public values don’t have to be kept secret, they should be protected against modification!
- We can factor all keys efficiently and create a private key.
- All keys, even of 4096 bit size, have at least one easily factored modification.
- With careful timing, this attack can succeed in minutes.

Weakening the public modulus using Voltage Fault Injection is a practical means of attacking RSA signature verification.
Discussion / Future work

- Specialized equipment was used in our experiments
  - But this attack should also work with cheaper, open source hardware, such as a ChipWhisperer

- We had control over the target’s code, allowing easy triggering
  - For targets not under our control, Side Channel Analysis can be used to determine timings

- Signature verification was not tested on target
  - Suggest implementing

- We suggest applying this to a secure boot implementation

- Suggest looking into the effect on various signing schemes
  - PKCS#1 v1.5, RSA-PSS, RSA-OAEP, etc.
  - RSA-CRT signature generation will not work with these keys
Thank You!

GitHub https://github.com/ivovanderelzen/GlitchRSA/

Questions?
Extra bits
Odds of hitting a single byte

- If we target a single byte we can hit it about 1.7/1000 or 0.17% of the time
- We need to do 1761 tests to get a 95% chance of hitting this byte at least once
  - \[ \frac{\ln(1 - 0.95)}{\ln(1 - 0.0017)} = 1760.7 \]
- With a glitch rate of 12 per second, this will take 147 seconds, about 2.5 minutes
- With a (conservative) rate of one every 10 seconds
  - \[ 1761 \times 10 / 360 = 292 \text{ minutes, or about 5 hours} \]
Calculating Euler’s totient

- Generalized formula:
  - \( \phi(n) = \phi(p_1) \cdot \phi(p_2) \cdots \phi(p_n) \)
- Normally RSA works with two prime factors
  - \( \phi(n) = \phi(p) \cdot \phi(q) \)
  - \( \phi(n) = (p - 1) \cdot (q - 1) \)
- More than two factors
  - \( \phi(n) = (p - 1) \cdot (q - 1) \cdot (r - 1) \cdots \)
- Prime power factors
  - \( \phi(p^k) = p^{k-1} \cdot (p - 1) \) (Where \( p \) is the prime factor and \( k \) is its exponent)
- If \( N \) is prime
  - \( \phi(n) = n - 1 \)
Message Coprimality

- RSA states that the message should be coprime with the modulus
  - \(gcd(m, n) = 1\)

- Other situations also work
  - Let \(p, q, r\) be prime (power) factors of \(n\)
  - \(gcd(m,n) = p\) (a factor of \(n\) divides the message)
  - \(gcd(m,n) = p \cdot q \cdot r\), etc... (product of any of the factors)

- With prime power factors, we can run into an issue
  - Let \(p^k\) be a prime power factor of \(n\)
  - \(gcd(m,n) = p^k\) decrypts correctly
  - \(gcd(m,n) = p^x\) where \(x \neq k\), does not decrypt correctly