Hacking Embedded Crypto Implementations using Fault Injection

Johannes Bauer
<johannes.bauer@de.bosch.com>

Bosch Software Innovations GmbH

March 14th 2013

whoami

- Graduated at Uni Erlangen as Dipl. Inf.
- Employed at Bosch SI since 2009 (Immenstaad am Bodensee) as Software Developer
- Ph.D. candidate with Felix Freiling since April 2012
- Area of research: Security of embedded devices

Embedded devices

- By this, we refer to microcontroller/SoC-based systems
- “System on a Chip” $\approx$ “Batteries included – everything is ready to go”
  - Flash-Memory (program storage)
  - EEPROM-Memory (data storage)
  - RAM
  - ALU, FPU
  - A bunch of peripherals

What's so special?

- So then embedded security is just security on smaller devices?
- Why would this be so special?
- Why do they even employ people to look into it?
- This doesn't make sense.

What's so special?

- With embedded devices, we have a fundamental different attack model compared to usual scenarios
- Main difference: our cryptographic material is in the hands of potential attackers 24/7 (i.e. customers)
- The attacker has unlimited time on her hands

Example

- This is a problem when cryptographic secrets are (more or less) sensitive
- Embedded devices doing crypto are ubiquitous
  - Symmetric crypto keys (RFID systems, locking systems)
  - Asymmetric keys (CV certificates, device identification, smart meter signatures)

Noninvasive attack

- Leakage can not only occur via timing channels, but also via radiated emission
- Most important example: differential power analysis
- Idea: measure power consumption over time, correlate with model, calculate key
- CMOS model dictates power consumption peak on a flipping bit, otherwise almost no dissipation

Seminvasive attack

- Timing attacks work a lot better in an environment where the attacker controls the device main clock
- Clock-cycle accurate measurements are no problem for a sophisticated attacker
- And clock can be controlled in a “bullet-time” manner
- See Goodspeed “A Side-channel Timing Attack of the MSP430 BSL”
- Where he exploits a 2-cycle difference (6511 vs. 6513 $\approx$ 400μs at 16MHz)

Invasive attack

- Processors operate only within certain constraints reliably
- The most important constraints are
  - Voltage conditions (supply voltage)
  - Temperature range
  - Clock waveform shape and parameters (frequency, dutycycle)

Invasive attack

- If all parameters are in spec, then the device works reliably (Atmel: 105 ℃ 153 years, 65 ℃ 1929 years)
- ...but if they aren't, then all bets are off
- This is what we'll be exploting in this talk

Maximally invasive attack

- With very sophisticated equipment, it's possible to open the chip physically
- Equipment: focused ion beam, electron microscope, microprober
- Enough to pretty much break all hardware today, but very expensive ($1 Mio.)
- So all in all, rather expensive

Fault injection

- An interesting attack method is to push the operating parameters into an illegal region
- These glitches/faults will cause the CPU to perform undefined behavior
  - Bitflips in the registers (flags!) or on the busses
  - Control flow mishaps (errors during decoding)

How?

- At critical points we induce brownouts (i.e. let the supply voltage drop below the guaranteed limit) for a very short period of time
- Or we modulate fast (nanosecond) pulses on the clock signal
- This will either modify control flow or data transfer

But why?

- Undefined behavior usually means: The program crashes
- This doesn't really help, but it's no real problem either: We just try again
- Try again until what exactly happens?

Try until?

- We try to force the processor into miscalculating a cryptographic operation
- In the hopes that it will spit out the (wrong) calculation result
- With that calculation result and some modular arithmetic, we can do pretty impressive things

Pretty impressive

pretty impressive (coll.)

1. *Making, or tending to make, an impression*
2. *Being able to recover private key material using fault injection :-)*

Keep calm and carry on

- Please bear with me for a moment
- If you don't understand everything comepletely, don't worry — you don't have to understand the whole theory in order to *use* it
- All I want to get across: fault injection isn't just some obscure made-up attack — it's a security developers nightmare and incredibly unintuitive

RSA Revisited

- Choose primes p, q (secret!)
- Calculate modulus $n = p \cdot q$ (public)
- Choose encrypting exponent e (public, usually 65537)
- $d = e^{-1} \mod \varphi(n)$
- Public: (e, n), private: (d, n)
- Encryption: $c = p^e \mod n$
- Signature: $s = m^d \mod n$

RSA

- The basic primitive that is always used in RSA is the modular exponentiation
- $x = a^b \mod c$
- This needs to be calculated often (encryption, decryption, singing, signature verification)
- And it tends to be slow (even more so on embedded systems)

RSA

- So we need a fast way to calculate $m^d \mod n$
- Implementations use an algorithmic trick (chinese remainder theorem)
- Precalculate in advance once:
  - $d_p = d \mod (p-1)$
  - $d_q = d \mod (q-1)$
  - $q' = q^{-1} \mod p$

RSA

- And then at runtime:
  - $s = m^d \mod n$ calculation via detour:
  - $s_1 = m^{d_p} \mod p = s \mod p$
  - $s_2 = m^{d_q} \mod q = s \mod q$
  - $h = (q' \cdot (s_1 - s_2)) \mod p$
  - $s = s_2 + (h \cdot q) \mod n$

Efficient?

- RSA-CRT method uses *twice* as many modular exponentiations than the naive approach
- But they are only of half bitlength
- Amount of operations is approximately quadratic with bitlength (square/multiply)
- i.e. $O(2 \cdot n^2)$ is better than $O((2 \cdot n)^2)$

Does it work?

- $s = s_2 + (((q' \cdot (s_1 - s_2)) \mod p) \cdot q) \mod n$
- Modulo p: $s_1 = s \mod p$
- Modulo q: $s_2 = s \mod q$
- $\overset{!}{\rightarrow}$ Modulo n: $s \mod n$ (because of CRT)

If things go wrong

- So assume we know the correct signature of a message
- And we then get the system to sign the same message again, this time we use fault injection
- Our target is to inject a fault so the system miscalculates s_2, i.e. it uses s_2'

If things go wrong

$$
\begin{aligned}
s &= s_2 + (((q' \cdot (s_1 - s_2)) \mod p) \cdot q) \mod n \\
s' &= s_2 + (((q' \cdot (s'_1 - s_2)) \mod p) \cdot q) \mod n \\
s - s' &= s_2 + (((q' \cdot (s_1 - s_2)) \mod p) \cdot q) \\
&\quad - s_2 - (((q' \cdot (s'_1 - s_2)) \mod p) \cdot q) \mod n \\
&= (((q' \cdot (s_1 - s_2)) \mod p) \cdot q) \\
&\quad - (((q' \cdot (s'_1 - s_2)) \mod p) \cdot q) \mod n
\end{aligned}
$$

If things go wrong

$$
\begin{aligned}
s - s' &= ((q' \cdot (s_1 - s_2)) - ((q' \cdot (s_1' - s_2))) \mod p) \cdot q \\
&= ((q' \cdot (s_1 - s_2 - s_1' + s_2)) \mod p) \cdot q \\
&= \underbrace{((q' \cdot (s_1 - s_1')) \mod p)}_{x} \cdot q
\end{aligned}
$$

If things go wrong

Fault injection aesthetics

$$gcd(x \cdot q, n) = gcd(x \cdot q, p \cdot q) \stackrel{!}{=} q$$

RSA

- $p = 101, q = 103$
- $n = p \cdot q = 10403$
- $e = 7, d = e^{-1} \mod \varphi(n) = 8743$
- Public: (e, n), private: (d, n)
- Encryption: $c = p^e \mod n$
- Signature: $s = m^d \mod n$

Correct signature

- Signing $m = 1234$:
- $d_p = 43, d_q = 73, q^{-1} = 51 \mod p$
- $s_1 = 4, s_2 = 62$
- $h = 72$
- $s = 1234^d \mod n = 7478$

RSA Fault Attack

- Then glitched signing: $d'_q = d_q \,\&\, (\sim 1)$:
- $d_p = 43, d'_q = 72, q^{-1} = 51 \mod p$
- $s_1 = 4, s_2 = 72$
- $h = 67$
- $s' = 6973$

RSA Fault Attack

- Message: $m = 1234$
- Correct signature: $s = 7478$
- Wrong signature: $s' = 6973$
- Recovery: $\gcd((s - s') \mod n, n)$
- $\gcd(7478 - 6973, n) = \gcd(505, 10403) =$
  $= 101 \stackrel{!}{=} p$

Practical attack

- Requires hardware to induce faults in our target
- And some controlling logic that times when the faults are injected
- Together with some processing login (in short: another MCU system that evaluates the target's responses)

GoodFET JTAG

A word on other faults

- RSA-CRT is obviously not the only vulnerability
- Elliptic curves have a similar problem (force weak twist of the curve or glitch off-curve point)
- Symmetric ciphers implementations are also vulnerable
- All in all, this is a minefield and very counter-intuitive

What? How?

- Obviously we can't do this here with real hardware
- But we'll try the next best thing:
  - First we generate RSA keys with OpenSSL
  - Then "glitch" OpenSSL to create borked signatures
  - And use *sage* to recover our private key

WIFI Parameters

- ESSID: workshop0815
- Password: notsecure
- `ssh groupx@192.168.123.1`

Generating a certificate

- `./gencrt`
- Creates certificate (`myuser.crt`) and private key (`myuser.key`)
- Key is stored in DER format so we can easily hack it later on

Generating a document to sign

- `echo 'TODO: Think of witty text' >signme.txt`
- ;-)
- What that content is, doesn't matter.

Examining the key

- `./showkey myuser.key`
  - modulus: n
  - publicExponent: e
  - privateExponent: d
  - prime1: p
  - prime2: q
  - exponent1: $d \mod (p-1)$
  - exponent2: $d \mod (q-1)$
  - coefficient: h
- If you like it rough: `openssl rsa -inform der -noout -text -in myuser.key`

Glitching OpenSSL

- Create a copy of your key
- `cp myuser.key myuser_broken.key`
- Now look at the exponent1 again and memorize a part of it
- Then open hexedit and edit that exponent1 (e.g. flip a bit):
- `hexedit myuser_broken.key`
- Exit hexedit with Ctrl-W, Ctrl-X
- Examine both keys again to verify it worked: `./showkey myuser_broken.key`

Glitching OpenSSL

- You now have an intact keypair and a broken one (modified exponent1)
- Let's sign your document with both to gernerate two signatures:
- ```
  ./signdoc
  Signing with proper key
  Signing with broken (fault injected) key
  ```
  ```

Glitching OpenSSL

- Two signatures have been created, one by the proper and one by the broken key
- Dump those signature values:
- `./dumpsig signature.p7`
- `./dumpsig signature_broken.p7`

Glitching OpenSSL

- Copy and paste the signature values (`OCTET STRING` at the end) for use in sage
- Open up sage and do some modular magic!
- Redirect your browser to `https://192.168.123.1:8080`

Are there further...

...questions?