Formal verification for constant-time cryptography

Ján Jančár
jan@neuromancer.sk

Centre for Research on Cryptography and Security

IA072
December 4, 2020

- Cryptography
- Side-channel attacks
- Timing attacks
- Formal verification for constant-time cryptography
    - ctgrind
    - ct-verif
    - SideTrail
    - ct-fuzz

Cryptography

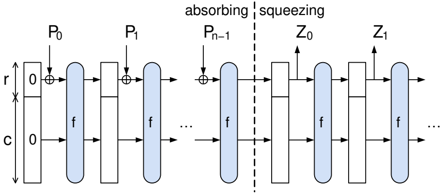
Cryptography

- Symmetric
  - Uses the same key for decryption/encryption
  - Encryption, Hash functions, …
  - **AES**, **SHA1**, **SHA256**, …
- Asymmetric
  - Uses different keys for the operations (private + public = keypair)
  - Encryption, Digital signatures, Key exchange, …
  - **RSA**, **Diffie-Hellman**, **ECC**, …
- Post-quantum
  - Symmetric crypto is ok
  - Asymmetric broken by (future) quantum computers
  - Needs new algorithms
  - **Lattices**, **Codes**, **Isogenies**, …
- Libraries & Protocols

Cryptography

Symmetric

- Bit and byte operations
- `xor`, `and`, `shift`, ...
- Byte permutations
- No number theory
- Rounds: same operations repeated




```
for (let round=1; round<Nr; round++) {
  state = subBytes(state);
  state = shiftRows(state);
  state = mixColumns(state);
  state = addRoundKey(state, round, schedule);
}
```

Cryptography

Asymmetric

- Modular arithmetic
- Number theory ($\mathbb{Z}_n^*$, $\mathbb{F}_p$, ...)
- Only private key is secret
- Huge integers (256 bits for ECC, 4096 for RSA)
- Bignumber libraries

```
N = P
Q = 0
for i from 0 to m do
  if ((d >> i) & 1) == 1 then
    Q = point_add(Q, N)
  N = point_double(N)
return Q
```

```
point_add((X1, Y1, Z1), (X2, Y2, Z2)):
 u = Y2*Z1-Y1*Z2
 v = X2*Z1-X1*Z2
 A = u2*Z1*Z2-v3-2*v2*X1*Z2
 X3 = v*A
 Y3 = u*(v2*X1*Z2-A)-v3*Y1*Z2
 Z3 = v3*Z1*Z2
 return (X3, Y3, Z3)
```

Cryptography

Post-quantum

- Quantum computers break all classical asymmetric algorithms
- Post-quantum cryptography attempts to fix it
- More number theory ($\mathbb{F}_p$, $\mathbb{F}_q$, $\ldots$)
- More linear algebra
- Very large keys (kB)
- Lattices, Codes, Isogenies

Cryptography

Protocols & Libraries

- Basic crypto primitives are used in protocols
- Libraries collect primitives and protocols
- SSL/TLS, Signal, IPSec
- State machines
- Read message, decrypt, verify, process, sign, encrypt, respond
- Most in C, low-level functions in assembly

Side-channel attacks

Side-channel attacks

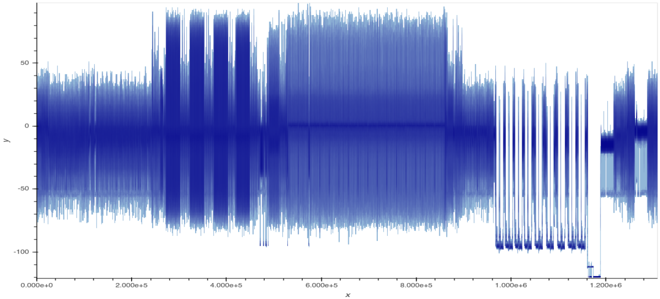
Side-channels

- Power
- Electromagnetic radiation
- Cache
- Errors
- Time
- Sound, …

Side-channel attacks

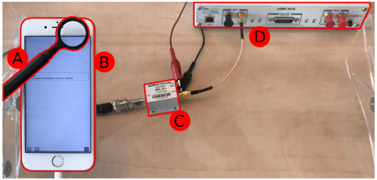
Power

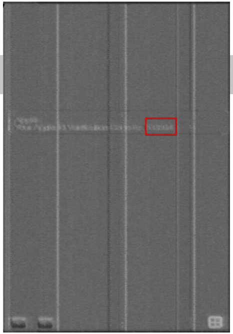
- Transistors take some power to switch
- Switching in a clock cycle is data dependent
- Thus, power consumption is data dependent
- Hamming weight of operand often leaks



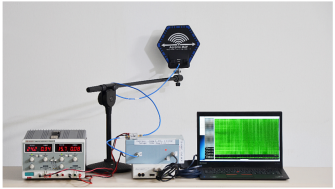
Side-channel attacks

Electromagnetic radiation





- Power also influences EM radiation from the circuit
- Get a good probe and record trace
- Can be localized to a part of a chip



Side-channel attacks

Cache

- Processors have several layers of memory cache
- Cache organized into cache lines
- Cache evicted in a Least Recently Used-like fashion
- *Prime+Probe* cache attack:
  - Malicious process accesses memory to prime all cache lines
  - Target process executes for a bit
  - Malicious process regains execution and checks the cache lines by timing how long a cache access takes
  - Cache hit: Target process **did not** touch cache line
  - Cache miss: Target process **did** touch cache line

Timing attacks

Timing attacks

```
function checkPasswordVarTime(password) {
  let correct = "hunter2";
  for (let i of correct) {
    if (i >= password.length || password[i] !== correct[i]) {
      return false;
    }
  }
  return true;
}
```

Timing attacks

```
R = P
Q = 2P
for i from bit_length(k) to 0 do
  if ((k >> i) & 1) == 1 then
    R = R + Q; Q = 2Q
  else
    Q = R + Q; R = 2R
return R
```

Minerva

Timing attacks

Leakage models

- Remote attacker
  - Wall clock time
- Local attacker (different process or VM)
  - Branching
  - Memory-access
  - Operands to some instructions
  - Instruction count

Formal verification for constant-time cryptography

Formal verification

- Want to somehow verify that implementations are constant-time
- What does that mean? Different for each tool
- **ctgrind**
- **ct-verif**
- **SideTrail**
- **ct-fuzz**
- + 23 more

ctgrind

Github

- Not really formal analysis
- Valgrind's memcheck can warn on uninitialized memory use
- Use Valgrind to track branching and memory-accesses on secret values
- `VALGRIND_MAKE_MEM_UNDEFINED` (memcheck client_request)
- Can be included in tests and CI
- Has false positives and false negatives

ct-verif

Github Github paper

- Formal foundation on what "constant-time" means
- Sound and complete reduction-based approach to verifying constant-timeness
- Prototype implementation based on SMACK, Bam-bam-boogieman and Boogie
- Case studies using the prototype

$$\begin{aligned} p ::= \;& \mathbf{skip} \mid x[e_1] \coloneqq e_2 \mid \mathbf{assert}\ e \mid \mathbf{assume}\ e \mid p_1 ; p_2 \mid \\ & \mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 \mid \mathbf{while}\ e\ \mathbf{do}\ p \end{aligned}$$

- Defines constant-timeness on *while programs*, with arrays and **assert**/**assume**
- x are program variables
- e are expressions

ct-verif

Constant-time implementations

- A **state** s maps variables x and indices $i \in \mathbb{N}$ to values $s(x, i)$, and we write $s(e)$ to denote the value of expression e in state s. The distinguished **error state** $\perp$ represents a state from which no transition is enabled.
- A **configuration** $c = \langle s, p \rangle$ is a state s along with a program p to be executed, and an **execution** is a sequence $c_1, c_2, \ldots, c_n$ of configurations such that $c_i \to c_{i+1}$ for $0 < i < n$.
- **safe** execution: $c_n \neq \langle \perp, \_ \rangle$; **complete** execution: $c_n = \langle \_, \textbf{skip} \rangle$
  execution of program p: $c_1 = \langle \_, p \rangle$, program is **safe** if all executions are safe

$$\frac{}{\langle s,\ \texttt{skip;}\ \ p\rangle \to \langle s, p\rangle} \qquad \frac{i = 1 \text{ if } s(e) \text{ else } 2}{\langle s,\ \texttt{if}\ \ e\ \ \texttt{then}\ \ p_1\ \ \texttt{else}\ \ p_2\rangle \to \langle s, p_i\rangle} \qquad \frac{s' = s[\langle x, s(e_1)\rangle \mapsto s(e_2)]}{\langle s,\ x\texttt{[}e_1\texttt{]}\ \ \texttt{:=}\ \ e_2\rangle \to \langle s', \texttt{skip}\rangle} \qquad \frac{s' = s \text{ if } s(e) \text{ else } \perp}{\langle s,\ \texttt{assert}\ \ e\rangle \to \langle s', \texttt{skip}\rangle}$$

$$\frac{p' = (p\texttt{;}\ \ \texttt{while}\ \ e\ \ \texttt{do}\ \ p) \text{ if } s(e) \text{ else } \texttt{skip}}{\langle s,\ \texttt{while}\ \ e\ \ \texttt{do}\ \ p\rangle \to \langle s,\ p'\rangle} \qquad \frac{s(e) = \texttt{true}}{\langle s,\ \texttt{assume}\ \ e\rangle \to \langle s, \texttt{skip}\rangle} \qquad \frac{\langle s, p_1\rangle \to \langle s', p_1'\rangle}{\langle s,\ p_1\texttt{;}\ \ p_2\rangle \to \langle s',\ p_1'\texttt{;}\ \ p_2\rangle}$$

ct-verif

Constant-time implementations

- A **leakage model** L maps program configurations c to observations $L(c)$, and extends to executions, mapping $c_1, \ldots, c_n$ to the observation $L(c_1, \ldots, c_n) = L(c_1)L(c_2) \cdots L(c_n)$.
- Two executions α and β are **indistinguishable** when $L(\alpha) = L(\beta)$
- **Branching model**:

$$\langle s, \textbf{if } e \textbf{ then } p_1 \textbf{ else } p_2 \rangle \mapsto s(e)$$

$$\langle s, \textbf{while } e \textbf{ do } p \rangle \mapsto s(e)$$

- **Memory-access model**:

$$\langle s, x_0[e_0] \coloneqq e \rangle \mapsto s(e_0)s(e_1) \cdots s(e_n)$$

- **Operand model**, for example:

$$\langle s, x[e_1] \coloneqq e_2 / e_3 \rangle \mapsto S(e_2, e_3)$$

- Given a set X of program variables, two configurations $\langle s_1, \_\rangle$ and $\langle s_2, \_\rangle$ are **X-equivalent** when $s_1(x, i) = s_2(x, i)$ for all $x \in X$ and $i \in \mathbb{N}$.
- Executions $c_1 \ldots c_n$ and $c'_1 \ldots c'_n$ are **initially X-equivalent** when c_1 and c'_1 are X-equivalent, and **finally X-equivalent** when c_n and c'_n are X-equivalent.
- X_i is the set of public inputs.
- X_o is the set of publicly observable outputs.

Definition 1 (Constant-Time Security). A program is secure when all of its initally X_i-equivalent and finally X_o-equivalent executions are indistinguishable.

ct-verif

Reducing Security to Safety

- **General idea**: Create a new program *Q* by product of the program *P* with itself, then assert equality of leakage of the two instances
- Simpler output-insensitive product
  - Assume equality of public inputs X_i
- Complex output-sensitive product
  - Handle publicly observable outputs X_o

$$\begin{array}{rl}
\mathsf{product}(p) & \texttt{assume } x{=}\hat{x} \texttt{ for } x \in X_{\mathsf{i}}\,\texttt{;} \\
 & \mathsf{together}(p) \\[1em]
\mathsf{together}(p) & \mathsf{guard}(p)\,\texttt{;} \\
 & \mathsf{instrument}[\lambda\, p.(p\,\texttt{;}\,\hat{p}), \mathsf{together}](p) \\[1em]
\mathsf{guard}(p) & \texttt{assert } L(p){=}L(\hat{p})
\end{array}$$

$\_$	$\mathsf{instrument}[\alpha, \beta](\_)$
`skip`	`skip`
$x\texttt{[}e_1\texttt{]} \texttt{ := } e_2$	$\alpha(x\texttt{[}e_1\texttt{]} \texttt{ := } e_2)$
`assert` e	`assert` e
`assume` e	`assume` e
$p_1\texttt{;}\ p_2$	$\beta(p_1)\texttt{;}\ \beta(p_2)$
`if` e `then` p_1 `else` p_2	`if` e `then` $\beta(p_1)$ `else` $\beta(p_2)$
`while` e `do` p	`while` e `do` $\beta(p)$

ct-verif

Implementation

- On the LLVM IR level
- Needs sources for annotation (public input/output, ...)
- Based on the SMACK toolchain, using the Boogie verifier

ct-verif

Discussion

- **Sound** and **complete**
  - **Sound**: Flags all insecure programs
  - **Complete**: Accepts all secure programs
- Needs source code annotation
- Complicated toolchain setup, outdated versions
- Usability?

SideTrail

Github paper

- Verification of **time-balancedness**
  - Weakening of constant-time notion
  - Leakage below some bound δ
  - Equivalent to constant-time for $\delta = 0$
- Uses time counter + instruction timing model
- For remote attackers

SideTrail

Time-Balancing

- δ**-secure**: For every possible public-input value, the timing difference between every pair of executions with different secrets is at most δ.
- Good for remote attackers (network jitter)

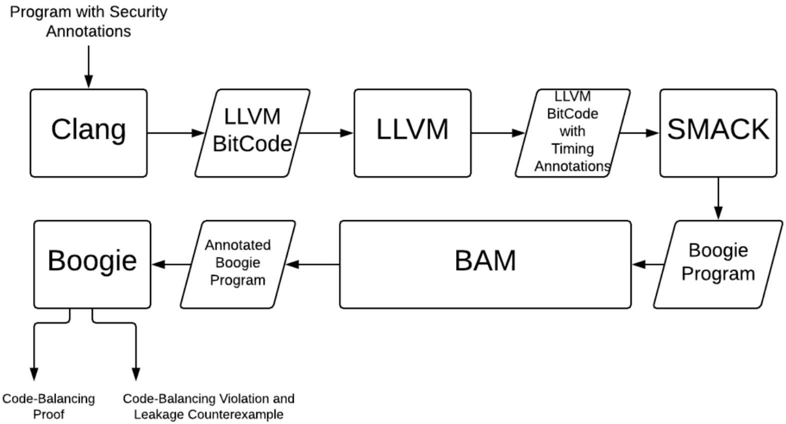
SideTrail

Verifying time-balancedness

- Similar to **ct-verif**
- Instrument program with timing counter
  - Leakage function $l(c)$ mapping configurations c with state s to timing
  - To keep track of the total cost of an execution we extend the set of variables with a time counter $\mathtt{l}$ as $V_L = V \cup \{\mathtt{l}\}$ and write the time counter instrumented program P_L as $l_1; p_1; l_2; p_2 \ldots; l_n; p_n$, in which each instruction l_i updates the time counter variable as $\mathtt{l} := \mathtt{l} + l(s, p_i)$.
- Compose P_L with its renaming $\hat{P}_L$ over variables $\hat{V}_L$ to construct $P_L; \hat{P}_L$
- Assert the equality of timing leakages in P_L and $\hat{P}_L$ at the end

SideTrail

Implementation



ct-fuzz

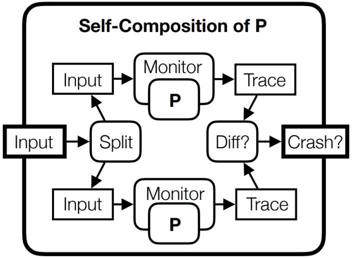
Github paper

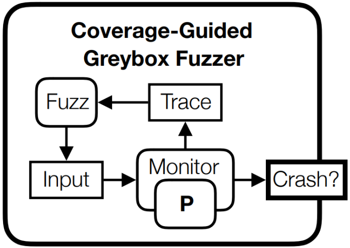
- Uses self-composition to reduce testing two-safety properties into testing safety properties
- Then uses the **afl-fuzz** fuzzer to test

ct-fuzz

Secure Information Flow

- Program splitting via forking
- Derive inputs from fuzz input
  - Split into one public input
  - and into two secret inputs
- Record observations
  - Instrument to record memory-access and branches
  - Hash traces to save memory
- Compare and abort on inequality





ct-fuzz

Discussion

- Uses fuzzing, so not sound
- Uses fuzzing, so setup already done in CI

Summary & Conclusions

- Cryptographic code is complex and small issues can lead to vulnerabilities
- Side-channels create hard to eliminate vulnerabilities
- There is an abundance of tools for verifying constant-timeness (collected 27, presented 4)
- Almost none of the tools are actually used
- Practical usability on real-world implementations is a concern

Thanks!

J08nY | neuromancer.sk | jan@neuromancer.sk

Icons from **Noun Project** & **Font Awesome**

References

- Stefan Mangard, Elisabeth Oswald, Thomas Popp: **Power Analysis Attacks - Revealing the Secrets of Smartcards**, ISBN 978-0-387-30857-9
- Adam Langley: **ctgrind**
- José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Michael Emmi: **Verifying Constant-Time Implementations**, USENIX Security 2016
- Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCarthaigh, Daniel Schwartz-Narbonne, Serdar Tasiran: **SideTrail: Verifying Time-Balancing of Cryptosystems**, VSTTE 2018
- Shaobo He, Michael Emmi, Gabriela Ciocarlie: **ct-fuzz: Fuzzing for Timing Leaks**, ICST 2020