Formal verification of side-channel attacks

Gilles Barthe
MPI-SP & IMDEA Software Institute

November 12, 2019
Motivation

- Cryptographic algorithms are provably secure
- But many cryptographic libraries are broken
  - implementation bugs
  - bad randomness
  - side-channels
  - ...
Formal verification of side-channels

- Writing secure implementations is notoriously hard
- Empirical evaluations are useful but insufficient
- Difficult to interpret theoretical approaches

Objective:

(automated) formal guarantees for real implementations

Case studies:
- Cache-based timing attacks
- Differential power analysis

Commonalities:
- modelling approach
- (relational) program verification
- non-trivial interactions with provable security
Modelling

- Precise modelling of CPU is not desirable
- We need good trade-offs between accuracy and tractability

\[
\begin{align*}
\text{model is too simple} & \implies \text{missed attacks} \\
\text{model is too complex} & \implies \text{verification unfeasible}
\end{align*}
\]

Standard warnings:

\[
\begin{align*}
\text{models are constructed} & \implies \text{attacks outside the model} \\
\text{proof fails} & \not\implies \text{practical attack}
\end{align*}
\]

Formal models should match practitioners’s view:

- Likely to yield tractable models
- Do not roll your own models

Ideally, formal models can be validated
Constant-time cryptography

- Control flow does not depend on secrets
  
  \[
  \text{if } H \text{ then } s_1 \text{ else } s_2
  \]

- Memory accesses do not depend on secrets
  
  \[a[H]\]

(array is public)

Why care?

- Best practice against cache attacks
- Non-constant-time implementations are often easily broken
- No panacea: execution time of instruction may depend on operands, does not account for micro-architectural attacks
## Sanity check: language-level vs system-level

### Language-level security

Constant-time is a (non-standard) information flow policy: leakage does not depend on secrets

### System-level security

Constant-time program is protected against adversary
- executing on same virtualized platform
- controlling the cache
- controlling the scheduler
- under all realistic replacement policies

(Mechanized) proof based on idealized model of virtualization:
- no branch prediction, no interrupt
Formalizing constant-time security

Observational non-interference

- Define leakage model
- Show that leakage is independent of secret
  Executions (with different secrets) have equal leakage

Variant with public outputs
Verifying constant-time security

- Build product program

  \[ \text{if } e \text{ then } s \text{ else } s' \leadsto \text{assert } e_1 = e_2; \text{if } e_1 \text{ then } p \text{ else } p' \]

- Check

  \[ m_1 =_L m_2 \implies p, m_1 \uplus m_2 \not\downarrow \]

- Flexible, compatible with off-the-shelf verifiers
- Sound and relatively complete
- Extensively evaluated
Enforcing constant-time security

- Start from information flow secure program
  - no high loop
  - no secret dependent memory access
- Eliminate high conditionals, early termination, etc.
- Flexible, allows programmers to write readable code
- Seriously evaluated
Constant-time security: challenges

- Post-quantum cryptography
- Secure compilation
- Constant-time security under speculative execution
Constant-time and post-quantum cryptography

\[
b \leftarrow \text{tt};
\]
\[
\text{while } b \text{ do } r \leftarrow \mu; y \leftarrow f(x, r); b \leftarrow P(y)
\]
return \(y\)

- Challenges: control-flow, non-uniform distributions
- One approach (for control-flow): leak guards
- But: security proof must be strengthened
- Another approach: use an alternative algorithm (GALACTICS)
Preservation of software-based countermeasures

Does my optimization preserve constant-time?

- Some optimizations break constant-time
- However many optimizations don’t

Techniques and case studies:
- CT-simulations (and simplifications)
- Jasmin (on paper) and CompCert (in Coq)
CT simulations

- Each target step is related to a source step (simulation proof)
- Prove that target leakages are equal for every two instances of the simulation diagram with equal source leakage
- Therefore source-level CT implies target-level CT
- Three variants: lockstep, one to several, one to any (number of steps must be explicit and uniform)
Simpler approaches

- Preserving, erasing or renaming leakage
- Case study: CompCert

<table>
<thead>
<tr>
<th>Compiler pass</th>
<th>Uses</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cshmgen</td>
<td>Leakage pres.</td>
<td>Type elaboration, simpl. of control Stack allocation</td>
</tr>
<tr>
<td>Cminorgen</td>
<td>Memory inj.</td>
<td>Sel. of operators and addr. modes</td>
</tr>
<tr>
<td>Selection</td>
<td>Leakage erasing</td>
<td>Generation of CFG and 3-address code</td>
</tr>
<tr>
<td>RTLgen</td>
<td>Leakage pres.</td>
<td>Function inlining</td>
</tr>
<tr>
<td>Inlining</td>
<td>Leakage transf.</td>
<td>Constant propagation</td>
</tr>
<tr>
<td>ConstProp</td>
<td>Leakage transf.</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>CSE</td>
<td>Leakage erasing</td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Leakage erasing</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Allocation</td>
<td>Leakage erasing</td>
<td>Branch tunneling</td>
</tr>
<tr>
<td>Tunneling</td>
<td>Leakage erasing</td>
<td>Linearization of CFG</td>
</tr>
<tr>
<td>Linearize</td>
<td>CT-simulation</td>
<td>Laying out stack frames</td>
</tr>
<tr>
<td>Stacking</td>
<td>Memory inj.</td>
<td>Emission of assembly code</td>
</tr>
<tr>
<td>Asmgen</td>
<td>Leakage transf.</td>
<td></td>
</tr>
</tbody>
</table>
Power analysis

Recover secrets from measuring power consumption

- SPA: single trace
- DPA: multiple traces

Serious threat for embedded systems
Masked implementations

- Values $x$ encoded as probabilistic $t+1$-tuples $(x_0 \ldots x_t)$ s.t.
  - $x_0, \ldots, x_t$ are i.i.d. w.r.t. to uniform distribution
  - $x = x_0 + \ldots + x_t$

- Operations operate on probabilistic values:
  - linear operations: apply the function to each share
  - non-linear operations: harder

```
Function SecMult(a,b)
    ab0,0 ← a0.b0; ab0,1 ← a0.b1; ab0,2 ← a0.b2;
    ab1,0 ← a1.b0; ab1,1 ← a1.b1; ab1,2 ← a1.b2;
    ab2,0 ← a2.b0; ab2,1 ← a2.b1; ab2,2 ← a2.b2;
    r0,1 $\leftarrow \mathbb{F}_{256}$; r0,2 $\leftarrow \mathbb{F}_{256}$; r1,2 $\leftarrow \mathbb{F}_{256}$
    r'0,0 ← (r0,1 + ab0,1) + ab1,0
    r'0,2 ← (r0,2 + ab0,2) + ab2,0
    r'2,1 ← (r1,2 + ab1,2) + ab2,1
    c0 ← (ab0,0 + r0,1) + r0,2
    c1 ← (ab1,1 + r1,0) + r1,2
    c2 ← (ab2,2 + r2,0) + r2,1
    return (c0, c1, c2)
```
Probing security, formally

Program $c$ is secure at order $t$ iff
- every set of observations of size $\leq t$ can be simulated with at most $\leq t$ shares from each input;
- the joint distribution for a set of observations of size $\leq t$ is independent from secrets

Relation to information flow
- Independence from secrets $\approx$ non-interference
- Opportunity to leverage programming language techniques

Validating the security model:
- equivalence with noisy leakage model
- experimentally
Challenges

Verification:

- Independence from secrets
- Combinatorial explosion
  - First-order masking:
    100 observation sets for a program of 100 lines
  - Second-order masking:
    4,950 observation sets for a program of 100 lines
  - Fourth-order masking:
    3,921,225 observation sets for a program of 100 lines

Moreover, size of programs grows quadratically with order

- Composition

Optimization

- Randomness complexity
- Parallelization
- etc
Checking independence from a secret $s$

Sets of observations is modelled by tuple $e$ of expressions

- **Rule 1:** If $e$ does not use $s$ then it is independent
- **Rule 2:** If $e$ can be written as $C[f \oplus r]$ and $r$ does not occur in $C$ and $f$ then it is sufficient to test the independence of $C[r]$
- **Rule 3:** Apply decision procedure, or compute distribution

**Benefits**

- easy to automate
- extends to large sets
- works on individual gadgets up to small orders
Composition

Constraint:
\[ t_0 + t_1 + t_2 + t_3 \leq t \]
Strong non-interference

- distinguish between output and internal variables
- show that any set of $t$ intermediate variables with
  - $t_1$ on internal variables
  - $t_2 = t - t_1$ on the outputs

  can be simulated with at most $t_1$ shares of each input
Secure Composition

Constraint: $t_0 + t_1 + t_2 + t_3 + t_r \leq t$

$t_0$ observations

$t_2$ observations

$t_1$ observations

$t_r$ internal observations

$t_3$ observations
Tools

MaskVerif
- Check probabilistic non-interference for large sets
- Probing security, NI, SNI, glitches
- Synthesis of refreshing gadgets

MaskComp
- Type-based information flow analysis
- Automated insertion of refresh gadgets
- Generate code at arbitrary orders
- Reasonably efficient at small orders
## Execution times

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>unmasked</th>
<th>Order 1</th>
<th>Order 2</th>
<th>Order 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>0.078s</td>
<td>2.697s</td>
<td>3.326s</td>
<td>4.516s</td>
</tr>
<tr>
<td>Keccak</td>
<td>0.238s</td>
<td>1.572s</td>
<td>3.057s</td>
<td>5.801s</td>
</tr>
<tr>
<td>Simon</td>
<td>0.053s</td>
<td>0.279s</td>
<td>0.526s</td>
<td>0.873s</td>
</tr>
<tr>
<td>Speck</td>
<td>0.022s</td>
<td>4.361s</td>
<td>10.281s</td>
<td>20.053s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Order 5</th>
<th>Order 10</th>
<th>Order 15</th>
<th>Order 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>8.161s</td>
<td>21.318s</td>
<td>38.007s</td>
<td>59.567s</td>
</tr>
<tr>
<td>Keccak</td>
<td>13.505s</td>
<td>42.764s</td>
<td>92.476s</td>
<td>156.050s</td>
</tr>
<tr>
<td>Simon</td>
<td>1.782s</td>
<td>6.136s</td>
<td>11.551s</td>
<td>20.140s</td>
</tr>
<tr>
<td>Speck</td>
<td>47.389s</td>
<td>231.423s</td>
<td>357.153s</td>
<td>603.261s</td>
</tr>
</tbody>
</table>
Masking: challenges

- More security models
- More composition results
- Secure compilation
- Post-quantum cryptography
Beyond side-channel verification

- High-speed cryptography
  - low-level optimizations
  - partially written in assembly
  - no formal guarantees (mostly)

- High-assurance cryptography
  - functional verification (mainly)
  - side-channel (sometimes)
  - cryptographic strength (maybe)
  - written in C-like languages
    - compiler is in the TCB
    - reasonably efficient, but no match for high-speed crypto

Goal: high-assurance and high-speed cryptographic libraries
Fast and verified assembly implementations
A holistic approach

- Algorithm is provably secure
- Reference implementation is safe and functionally correct
- Optimized implementation is functionally equivalent to reference implementation and (co-)safe
- Optimized implementation is leakage-free

Optimized implementation is functionally correct and provable secure against implementation-level adversary

A recent case study: SHA3
- reference and vectorized assembly implementation of SHA3
- functionally equivalent and correct
- indifferentiable from RO
EasyCrypt

Domain-specific proof assistant for

- tailored to relational and game-hopping proofs
- control and automation from state-of-art verification
  - interactive proof engine and mathematical libraries
    (a la Coq/ssreflect)
  - back-end to SMT solvers

Many case studies:

- Encryption, signatures, key exchange, zero-knowledge, multi-party and verifiable computation, SHA3, voting, KMS
- Private Statistics, Smart Sum, Vertex Cover, Sparse Vector
- SGD, Glauber dynamics, population dynamics, card shufflings
Jasmin

- “Assembly in the head”: mix of high-level constructs (variable names, global parameters, loops, functions) and low-level instructions and intrinsics
- Predictable and formally verified compiler
- Verification-friendly: safety, constant-time, functional correctness and equivalence checking (via EasyCrypt back-end)

Directions

- Support for other platforms
- Cautiously enrich language
Conclusions

- Practical tools for (specific) side-channels
- Interesting interactions with provable security
- “Practical” tools for correctness and provable security
- The future is fast and verified!