Computer-Assisted Cryptographic Proofs for Low-Level Implementations

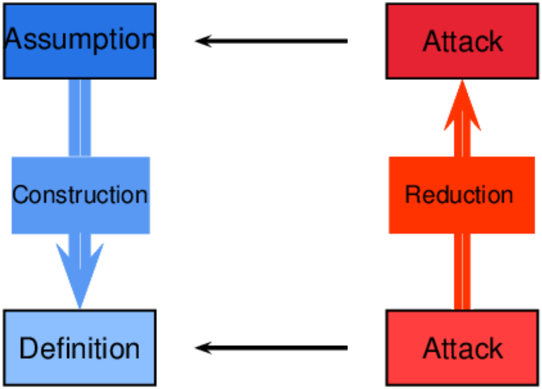
François Dupressoir
IMDEA Software Institute, Madrid, Spain

September 11, 2015 – Paris
Workshop on Implementation: Security and Evaluation

Objectives

- Obtain cryptographic security proofs:
    - for low-level implementations (*software* or hardware);
    - in low-level adversary models.
- Use formal methods to automate:
    - proof checking (heavy-duty/general formal methods: proof assistants, semi-automated program verification);
    - proof finding (lightweight formal methods: specialized tools for specialized problems);
    - code generation (semantics-preserving compilation, security-aware compilation, leakage protection...).

Cryptographic Proofs: A Reminder



Forall adversary against the construction, there exists an adversary against the primitive that has similar complexity and succeeds with almost the same probability.

The Problems with Cryptographic Proofs

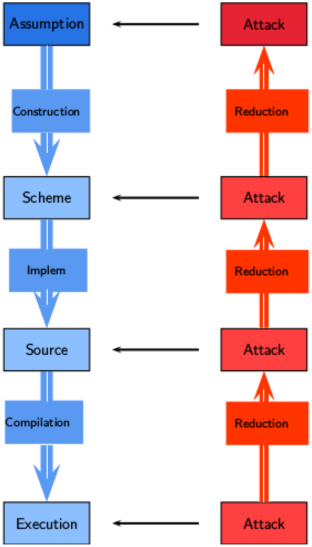
Implementing even good standards securely remains a monumental challenge.
(Schneier et al., 2015)

Assuming a *good* cryptographic specification/standard, three sources of problems in practice:

- algorithm to standard
  - elegant mathematics tend to fall apart when applied to bitstrings;
  - performance, compatibility, efficiency, flexibility, . . .
- standard to implementation
  - bugs when implementing crypto, protocol, or more . . .
- implementation to executable
  - compiler bugs, compiler optimizations, . . .
  - side-channels, ill-adapted adversary model, . . .

Our Plan: Stacking Reductions

- Computer-aided cryptographic proof yields security of standard algorithm;
- Functional equivalence proof yields black-box implementation security;
- Show that *specific* non-black-box capabilities do not help low-level adversaries against a *specific* implementation.



Verifiable Cryptographic Security for Standard Algorithms

- EasyCrypt: an interactive proof assistant...
- ... that includes program logics for reasoning about probabilistic programs...
- ... and in particular to formalize cryptographic proofs:
  - equivalence, equivalence up to failure (relational logic);
  - probability bounding (probabilistic logic);
  - ...
- Other tools provide some automated proof finding in specialized cases:
  - ZooCrypt

Verifiable Black-Box Cryptographic Security for Standard Implementations

- If you have the time and manpower and are willing to work with a high-level language, you can try direct proofs of security at the implementation level:
  - on F# implementation: Bhargavan et al.'11-??
- Otherwise, use one of these techniques to turn a security proof on the standard into a *black-box* security proof on the implementation:
  - Code extraction: Blanchet and Cadé'12, Almeida et al.'14
  - Model extraction: Aizatulin et al.'12
  - By total correctness: D'13, Almeida et al.'13, Beringer et al.'15

How far can we take this?

- Existing tools more than adequate to prove total functional correctness for this kind of algorithms. This brings us to C.
  - Compositionality helps a lot: proofs of component correctness can be reused.
  - Constructing such proofs is still a lot of work, and some care is needed to make sure the implementation adversary model makes sense, but they are well understood.
- Compiler correctness guarantees semantics preservation for valid programs. Program validity is often discharged in the previous step. This gives you *black-box* security of the executable code.
  - Trust your compiler and you should be ok.
  - Better yet, prove your compiler: CompCert (Leroy et al., 05-??)

Verifiable Side-Channel Cryptographic Security of Low-Level Standard Implementations

Observation: It is sufficient to prove functional equivalence of the executable with the standard (e.g. by semantic preserving compilation)
AND to prove that the leakage produced by the executable code can be perfectly simulated using only the adversary's view (public inputs and outputs).

Leakage Simulation the Easy Way

- We can check (or enforce) that a leakage simulator exists very easily in some cases using (probabilistic) *non-interference*
  - executing the program on any two initial memories that agree on (the marginal distribution of) their public variables yields two final memories that agree on (the marginal distribution of) their public variables;
  - the simulator is then trivial: given the public inputs, sample the private ones at random and run the program.
- Three examples of "easy" leakage simulations:
  - *constant-time* crypto (timing countermeasure);
  - two tools for reasoning about the deployment of masking.

Constant-Time Cryptography

with J.B. Almeida, M. Barbosa and G. Barthe

- Timing gives information about secrets;
- Adversary gets:
  - list of program counters (branch prediction, instruction cache...); and
  - list of memory addresses read or written (cache misses);
- Countermeasure: ensure that these never depend on secret inputs.
  - Simple non-interference property;
  - Some refinements needed for best precision (Encrypt-then-MAC).
- A lot of formal tools exist to verify this kind of properties.
  - Type-systems (applied to primitives, MEE-(TLS)-CBC...),
  - Product programs (ongoing work).
- Easy to generalize to any compositional leakage model that leaks more than program counters.

Modelling DPA Adversaries

with G. Barthe, S. Belaïd, P.-A. Fouque, B. Grégoire and P.-Y. Strub

Noisy Leakage model

- Adversary receives responses and a *noisy leakage trace*.
- Security is entropy-based.
- Difficult to reason about automatically.

t-threshold probing model

- Adversary (adaptively or non-adaptively) chooses at most t locations (variables, nodes, wires) in the circuit to probe;
- Security is simulation-based: any set of probes of size at most t can be simulated using at most $m - 1$ shares of the secrets;
- Formal methods apply nicely.
  - "can be simulated" $\sim$ "is independent of the secret"
  - (probabilistic) non-interference... a lot of it

(Duc et al., 14) show that security in the noisy leakage model is implied* by security in the t-threshold probing model.

Enter Masking

Masking uses secret-sharing schemes to protect implementation against DPA and other side-channel attacks.
For example, using an additive secret-sharing scheme:

- A secret x is uniformly split into m shares $\vec{x} = (x_0, \ldots, x_{m-1})$.

$$\begin{aligned} x_0 &\overset{\$}{\leftarrow} \mathbb{F}_{256} \\ x_1 &\overset{\$}{\leftarrow} \mathbb{F}_{256} \\ x_2 &\leftarrow x \oplus x_0 \oplus x_1 \end{aligned}$$

- Intuitively, splitting secrets into m shares protects against adversaries that can set up $m - 1$ probes.
  - But multiplication is an issue.

Verifying Masked Algorithms

- The sheer size of *each* problem makes it unrealistic to expect flawless pen and paper proofs.
- What can we do?
  - Security in the t-threshold probing model is probabilistic non-interference.
  - A lot of it: we need to prove that each t-tuple of intermediate variables is distributed independently from the secret inputs.
  - That's $O(|\mathbb{F}|^r t^{2t})$ solutions to count (ESW'13, ...)...
  - ... or $O(t^{2t})$ automorphisms to find.

Verifying Masked Algorithms

- The sheer size of *each* problem makes it unrealistic to expect flawless pen and paper proofs.
- What can we do?
  - Security in the t-threshold probing model is probabilistic non-interference.
  - A lot of it: we need to prove that each t-tuple of intermediate variables is distributed independently from the secret inputs.
  - That's $O(|\mathbb{F}|^r t^{2t})$ solutions to count (ESW'13, ...)...
  - ... or $O(t^{2t})$ automorphisms to find.

Proving Probabilistic Non-Interference

- Program + Set of Probes = Set of probabilistic field/ring/group expressions
- Identify a large sub-expression e, where a random sampling r appears in an invertible position, and nowhere else. Replace e with r.
- Iterate until all trace of the secret disappears (Success!) or a fixpoint is reached (Failure!).

What about that 2^{2t}?

Idea: Instead of trying to prove $\binom{n}{t}$ t-tuples non-interferent, why don't we try to prove many less, much larger sets non-interferent?

- When proving probabilistic non-interference, keep track of the successive substitutions.
- Once we know that a t-tuple is non-interferent, we add other probes to it and try to replay the substitutions in order.
- The difficulty is now in recombining the various chunks and making sure we haven't forgotten any.
- No chance of being generally efficient (NP-hard).

And in practice?

Reference	Target	# tuples	Result	Complexity # sets	Complexity time
First Order Masking					
RP-CHES10	$\odot$	13	✓	7	ε
CPRR-FSE13	Sbox	63	✓	17	ε
CPRR-FSE13	full AES	17,206	✓	3,342	128s
Second Order Masking					
SP-RSA06	Sbox	1,188,111	✓	4,104	1.649s
RP-CHES10	$\odot$	435	✓	92	0.001s
RP-CHES10	Sbox	7,140	X	866	0.045s
RP-CHES10	AES KS	23,041,866	✓	771,263	340,745s
CPRR-FSE13	AES (2 rnds)	25,429,146	✓	511,865	1,295s
CPRR-FSE13	AES (4 rnds)	109,571,806	✓	2,317,593	40,169s
Third Order Masking					
RP-CHES10	$\odot$	24,804	✓	1,410	0.033s
CPRR-FSE13	Sbox	4,499,950	✓	33,075	3.894s
CPRR-FSE13	Sbox*	4,499,950	✓	39,613	5.036s
Fourth Order Masking					
SP-RSA06	Sbox	4,874,429,560	X	35,895,437	22,119s
RP-CHES10	$\odot$	2,024,785	✓	33,322	1.138s
CPRR-FSE13	Sbox	$2,277,036,685$	✓	3,343,587	879s
Fifth Order Masking					
RP-CHES10	$\odot$	216,071,394	✓	856,147	45s

And in practice?

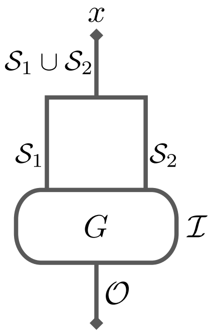
- Full AES at order 2 ran for 12 days (before we stopped it).
- An optimistically linear estimate put completion somewhere in the 4 year range...
- We need to do something about that compositional feeling.

The Problem with Composition

Definition (Tight Simulation (RP'10 Security))

A gadget G is t-tight simulatable whenever, any $d \leq t$ probes in G can be simulated using at most d shares of each of its inputs.

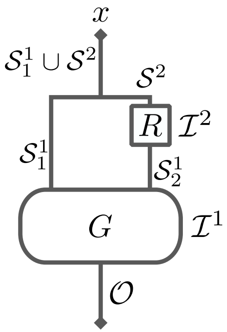
- Only implies t-threshold probing security when gadget is applied to inputs that are independently shared.



Strong Simulation

Definition (Strong Simulation)

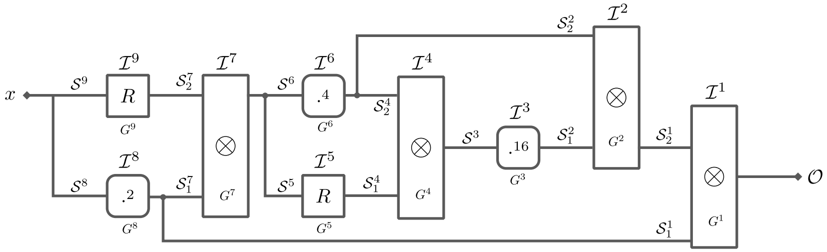
A gadget G is t-strongly simulatable whenever every set of $d_i + d_o < t$ probes on G that are split between internal (d_i) and output (d_o) wires can be simulated using at most d_i shares of each of the gadget's inputs.



Compositional t-Threshold Probing Security

- Given security properties of individual gadgets in the circuit...
  - Strong Simulation (Refresh, SecMult, . . .)
  - Affine Simulation (for affine and linear gadgets)
  - Tight Simulation (maybe some user-defined gadgets?)
- ... and a universallly quantified set of at most t adversary observations (split between the core gadgets)...
- ... starting from the last gadget in the circuit:
  - check that the number of observations on the gadget is $\leq t$;
  - simulate the observations on that gadget;
  - add the set of input shares required to build the gadget's simulators as output observations on the parent gadgets;
  - repeat on next gadget; ...
- ... and finally check that the number of shares of each secret input required to simulate the whole circuit is $\leq t$.

A Compositional Proof Example



- The S-box is checked once, and its “type” is obtained;
- This type can be used as is to prove, say, full AES.
- Bonus: a failure in the proof indicates the need for a Refresh.

Compositional Results

Scheme	# Refresh	Time	Memory
AES ($\odot$)	2	0.09s	4Mo
AES ($x \odot g(x)$)	0	0.05s	4Mo
Keccak	0	121.20	456Mo
Keccak (gen)	600	2728.00s	22870Mo
Simon	67	0.38s	15Mo
Speck	61	6.22s	38Mo

Table : Time taken to verify masked implementation (at any order)

Conclusion

- Combination of techniques to obtain low-level provable security results.
  - Separate scheme security from implementation security;
  - Cut the task into subtasks for which good formal tools exist.
- Proofs are done on a particular program.
  - All techniques are language-independent (C, ASM, HDL...);
  - The leakage models are not.
- Proofs are done with respect to a model.
  - Choice/Adequacy of model remains expert knowledge;
  - Tools illustrated in simple models.

What next?

- Capture lower-level leakage models:
  - Transitions and glitches can be easily captured in the first tool;
  - Extending the second tool to scenarios where a single probe may reveal several values is feasible.
- Extensions to non-compositional leakage:
  - Take pipelines, caches etc. into account...
- Potential use in evaluation processes?
  - Evaluate device leakage, identify PoI;
  - Give some context of the PoI to the verification tool;
  - Get much reduced list of potential flaws that can be used to identify an attack or dismiss false positives.