Formally Proved Security of Automatically Protected Software Against Physical Attacks

Pablo Rauzy

rauzy@enst.fr

advisors: *Sylvain Guilley, Jean-Luc Danger*

Telecom ParisTech

LTCI / COMELEC / SEN

July 10, 2013

Physical attacks

There are two main categories of physical attacks:

- side channel attacks, which are passive,
- fault injection attacks, which are active.

Side channel attack

A *side channel attack* is any attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms.

Examples of side channel information:

- timing
- power consumption
- electromagnetic leaks

Fault injection attack

A *fault injection attack* consists in modifying parameters or intermediate values of a cryptosystem's computation in order for the final result of the computation to leak sensitive information about the system, often by comparing the compromised result with a correct one.

There are many form of fault injections:

- invasive / non-invasive
- destructive / non-destructive
- global / precise

Formally proved security

Formally proved security against physical attacks is a relatively new topic.

- Cryptosystems' software should be bug-free and rely as little as possible on hand-written code for critical parts.

$\Rightarrow$ We need tools to formally assess the security of implementations, and where possible automatically generate or insert countermeasures against physical attacks.

Our projects

We are working on two projects:

- Automatically inserted and formally proved countermeasures against side channel attacks.
- Formal proof of security against fault injection attacks.

In both cases, we apply our methods on real-world algorithms and implementations.

Side channel countermeasures

Side channel countermeasures can be classified in two categories:

- Those that use randomness to make the leakage statistically independent from sensitive data (like *masking*).
- Those that make the leakage indistinguishable (like *dual-rail with precharge logic* (DPL)).

Automated masking has already been explored but most efforts have yet to be done for DPL.

Masking

- Needs randomness (hard to formalize).
- Assumes shares are not interfering neither logically (opcode's effect depending on previous ones) nor physically (glitches, cross-coupling).
- Assumes the data and operations in the algorithm are embedded within a group (for instance $(\mathbb{F}_2^n, \oplus)$ for Boolean additive masking).
- Protection depends on the linearity of the operation.
- Masking S-Boxes is difficult in general.

Dual-Rail with Precharge Logic (DPL)

- Assumes that (at least) two equivalent (in terms of leakage) resources exist.
- Protection depends less on the algorithm.
- Algorithms can be bitsliced which leads to a simple model that operates at the bit level.

Since it seems easier, we chose to start working on automatic insertion of countermeasures with DPL.

Formally proven countermeasures

We want to be able to formally prove two properties on automatically applied countermeasures.

- The semantics of the code must not be unaltered by the transformation that adds countermeasures (correctness).

$\Rightarrow$ Exactly what a formally proven compiler does.

- The countermeasure must be efficient (security).

$\Rightarrow$ We need formal models of the possible side channel leakages, and then use them to prove that the obtained code is protected against those leakages.

Moreover, being able to prove the security enable optimization.

Our project against side channel attacks

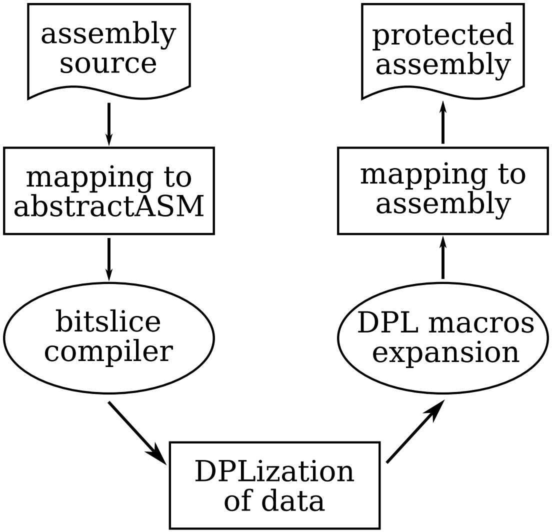
- Automatically insert the DPL countermeasure in some arbitrary assembly code.
- Formally prove that the resulting assembly code is protected.
- Apply our research to a real-world algorithm running on a real-world system.

Automatic insertion of the DPL countermeasure

- We need to be able to manipulate any assembly code. For that we designed a generalist assembly called AbstractASM that our tools manipulate.
- AbstractASM is generalist enough for us to be able to easily map instructions from most assembly one-to-one and back.
- AbstractASM instructions obey the following pattern:

```
opcode destination operand1 operand2
```

Workflow


assembly source
mapping to abstractASM
bitslice compiler
DPLization of data
DPL macros expansion
mapping to assembly
protected assembly

Proving the resulting code leak-free

- We use the hamming distance leakage model.
- Using symbolic evaluation, we prove that each time a register or memory cell is updated, the hamming distance between its old and new values is constant, independent of sensitive value.
- This is done by executing the code on our AbstractCPU, which is equipped to keep track of all possible leakage.

AbstractCPU

- Instead of values, computation are carried using sets of possible values.
- At the beginning each bits of the key and the cleartext can be 0 or 1 (or their DPL encoding).
- For each instruction all possible results given all the possible inputs are computed.
- This allow to keep track of all the possible values for the hamming distance each updates.
- If for each update there is at most one possible value, it means there is no leakage.

Case study

- We used our tools to protect an implementation of PRESENT written in 8 bit AVR assembly.
- The resulting code was proved leak-free and ran successfully on the AVR micro-controller.

Our project against fault injection attacks

- We consider fault injection attacks consisting in changing the value of intermediate variables in the computation.
- Find fault injection attacks on high-level model of cryptographic algorithm.
- Or formally prove their absence.

Finding fault injection attacks

- The idea is to model the computation of the algorithm using datatypes representing the interesting properties of the numbers.
- Define which properties allow an attack.
- Check for each possible fault injection if it results in an effective attack or not.

Case study

- On the CRT-RSA algorithm model we are able to find the Bellcore attack.
- Our goal is to model existing countermeasures and verify them with our tool.

That's it. Questions?