Catch me if you can

Locating (and fixing) side channel leaks (for dummies)

Elisabeth Oswald

European Research Council

Established by the European Commission

Outline

- Why: There are many more non-crypto experts, than crypto experts!

- What: This talk is about tools and techniques for detecting (and fixing) information leaks that are designed for developers who are not cryptographers.

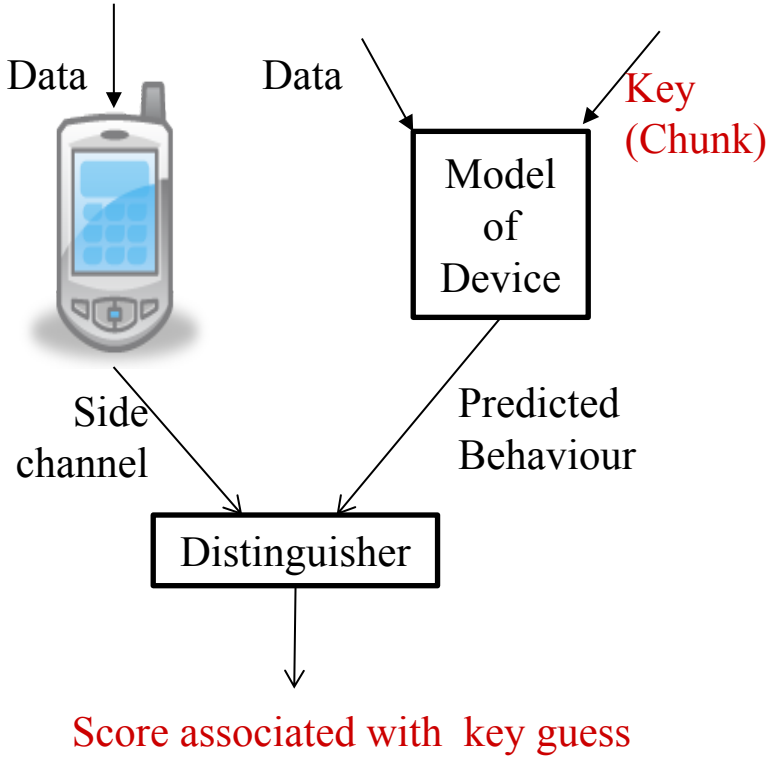
- How: With a lot of effort by developing an appropriate model of the TOE that integrates in some ‘design flow’.

Context

- Developed around 1995, with publications emerging from 1996 onwards, side channel attacks have exploited
  - Execution times
  - Power consumption
  - EM radiation
  - Cache behavior
  - RF emanation
  - Sound
  - Packet length
  - ….

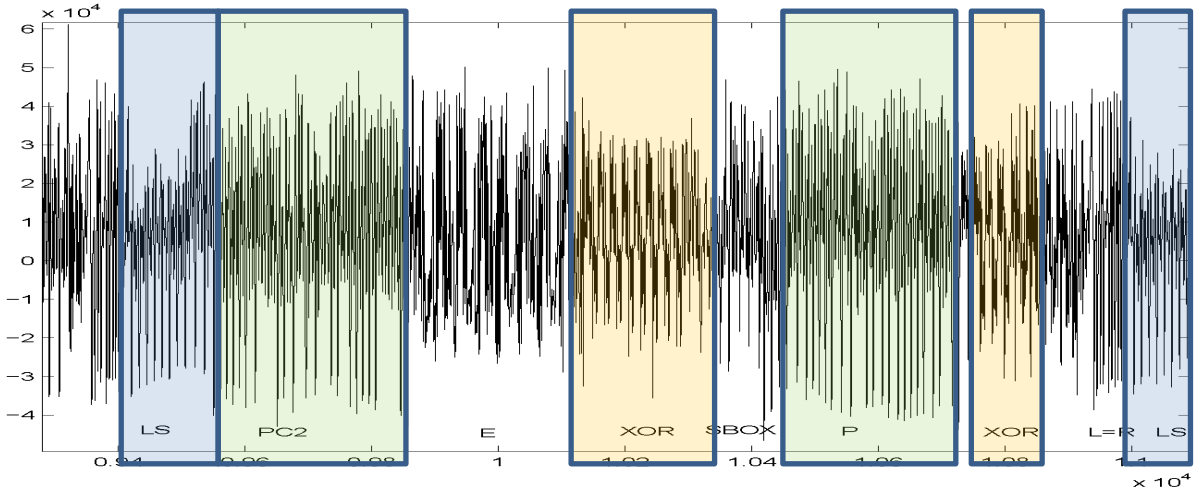
Attacks that Exploit Leakage

- Many attacks recover information about `chunks' of a secret key
  - Stronger attacks tend to have better device leakage models
  - Distinguisher needs to be chosen in conjunction with the device leakage model
  - High quality traces naturally also improve attack outcomes
- Some attacks recover plaintext information



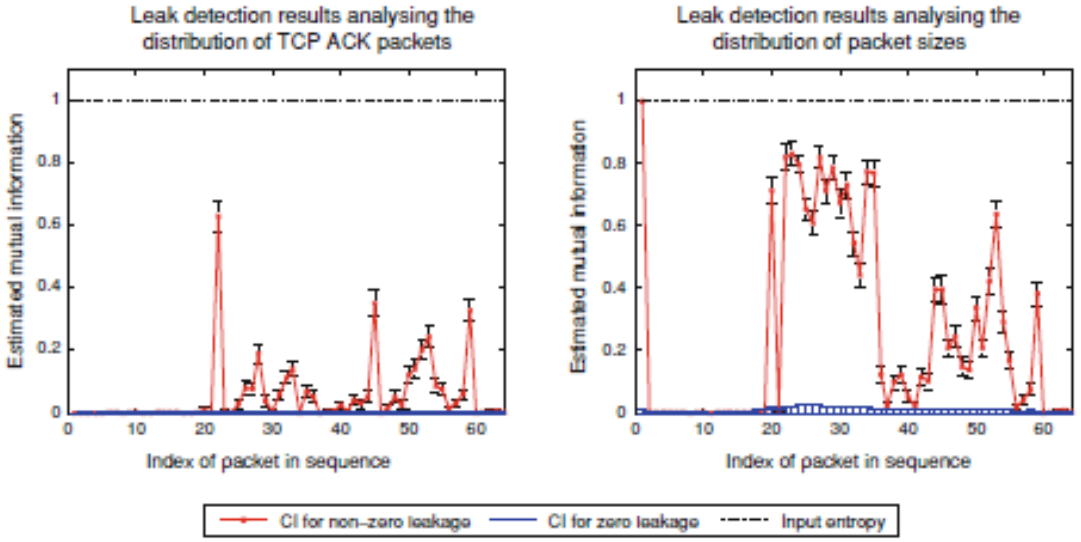
Context, cont.

- Attacks only ever get better



1999: attacks on block ciphers (DES, AES), exploiting physical leaks, simple implementations, trivial to break

2012: attacks on protocols (TLS), exploiting protocol leaks, non trivial attacks requiring profiling



Context, cont.

- Past attacks on real world products
  - PayTV as a ‘market driver’: protecting people from watching too much too bad telly is clearly very important!
  - Standards/evaluation schemes exist to protect chip cards in the context of banking applications (CC protection profiles, EMVCo scheme)
  - But also printer cartridges, and other ‘gadgets’ that have static secret keys embedded are routinely protected

These applications are all somewhat ‘closed’: specialist developers with access to crypto/side channel expertise + labs are available. Code/Implementations/Evaluations remain confidential.

Context, cont.

- But the world has changed:
  - Payments are getting integrated, e.g. in software apps running on mobile phones
  - We have more and more `smart' devices around that interact with us, and sometimes connect us with other devices/apps/institutions/people
  - These systems are much more `open' in the sense that there exist many (small) companies that produce software.

In this context, access to crypto/side channel specialists + lab, can no longer be taken for granted.

Context, cont.

- Research into attacks and mitigation strategies (provable or not) has more or less assumed ‘specialist developer’ so far.
  - We develop ‘Cryptography for Cryptographers only’
  - We need ‘Cryptography for Everybody’, and this includes ways to implement cryptography securely in the real world

- A large part of my research interest is to find ways to `automate’ implementing crypto so as to take away (some of) the burden from developers.

Automation

Let’s now focus on mitigation of physical leaks:

- What leaks?
- (Why does it leak)?
- How can it be fixed?

Questions:

- At which point in the design cycle to do this?
- What leaks do matter?
- How to include developers’ decisions?

Automation Approaches

`Closed World' approaches from the past include:

- Hardware level: assume we build a processor/crypto module, have full design details, aim for early mitigation
  - Pros: hope to remove leakage entirely, implies that software developer does not need to care at all
  - Cons: unable to remove leakage entirely, impractical as for most applications the fabrication of a dedicated security IC is not an option
- Software level: assume that crypto runs on a leaky processor, a simplistic leakage model (Hamming weight), and focus on a specific algorithm
  - Pros: does not rely on control/exact knowledge of hardware design, potentially more applicable to a wide range of applications, promise to prove security
  - Cons: unable to capture any leak that does not fit the model

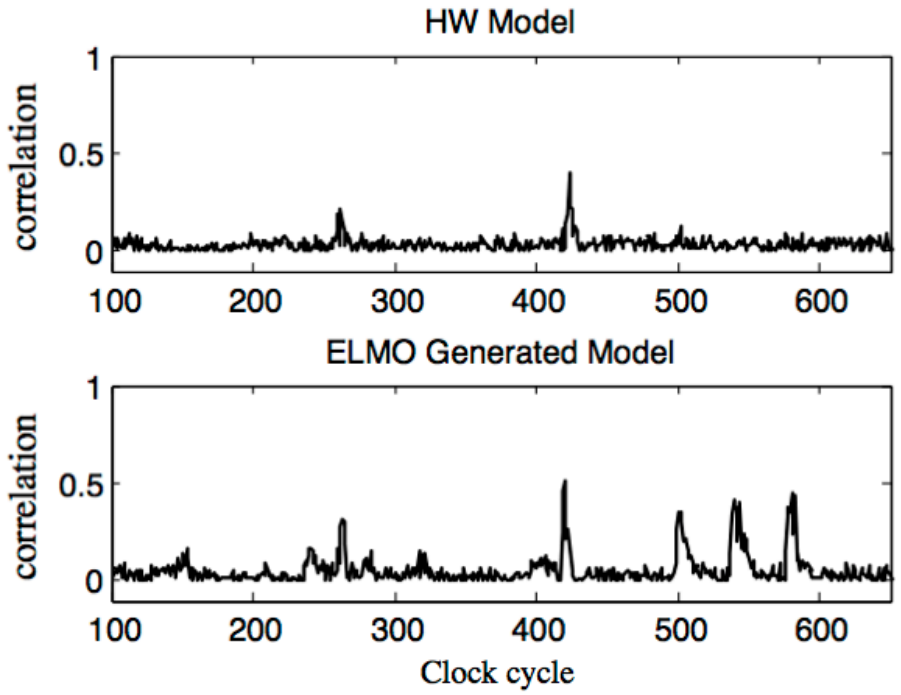
Automation Approaches, cont.

Compiler based approaches:

- 2012: we developed a compiler extension, which required a domain specific language, that was capable of taking a ‘raw’ AES implementation, and translate it into a first-order Boolean masked implementation in Thumb assembly for an ARM7TDMI.
- 2013: Bayrak, and Agosta independently proposed different compiler extensions that ‘identified’ vulnerable instructions and applied some countermeasures
- 2013 onwards: Dupressoir published a series of papers in which formal verification was used to prove leakage properties of code

All approaches relied on very simplistic leakage models.

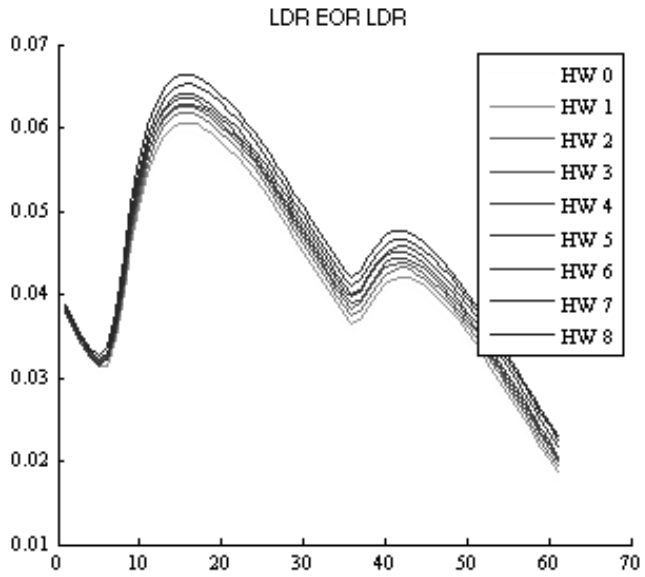
Importance of leakage models

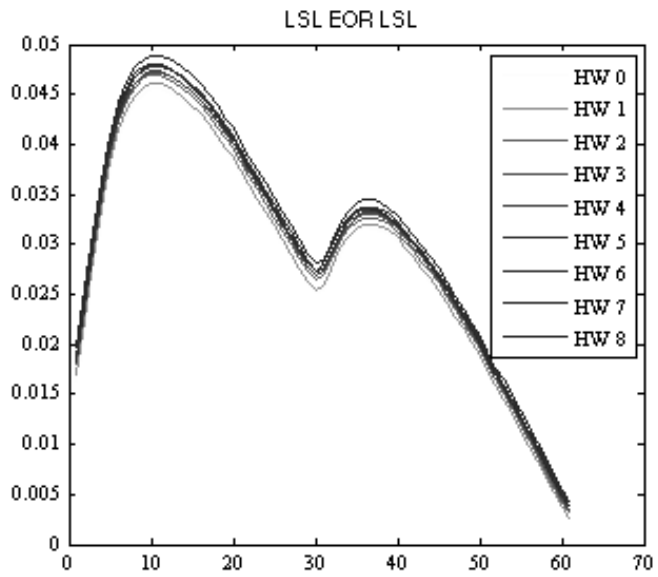


Side channel attack outcomes using the HW assumption (top), and a statistically estimated leakage model (bottom).

- Simplifications are good if they remove unnecessary complexity only
- We have seen many times (in the side channel community) that simplified assumptions render ‘proofs’ (arguments for security) useless
  - Even provably secure schemes such as ISW99 fail miserably in practice due to glitches
  - TI schemes equally make strong independence assumptions on small components

Importance of leakage models, cont.





Pictures showing power traces of an XOR operation: surrounded by LDR (left) and LSL (right).

- Leakage behavior can be very complex:
  - It depends on the state that the processor is in prior to a (target) instruction as well as what the next instruction will be
  - It depends on the pipeline architecture, functional components, busses, etc.
  - Thus modelling `an' instruction requires a sequence of instructions.

Automation challenges

- What leaks: without a sophisticated understanding of the target architecture's leakage 'reasoning' about implementations is pointless
- Why does it leak: without a 'white box' this cannot really be answered, but a good leakage model can potentially describe how the leakage functionally looks
- How can you reliably detect leakage in a new piece of code without having to instrument everything all the time
- How can you mitigate arbitrary leaks

Leakage detection

We now focus more on the `finding' than the `fixing'.

- Detecting information leaks is not a new topic: detecting `points of interest' has been a topic for discussion since the advent of `higher-order' (in this case meaning multivariate) DPA attacks
  - Methods that are easy to use tend to be based on the t-test (leakage model assumes individual bits' leakage differ), and correlation analysis (requires a power model), which are moment based statistics that produce unreliable results when using in a multivariate setting.
  - Statistically rigorous methods were developed by Chothia et al. based on Mutual Information (no power model required, cope better in a multivariate setting)

Leakage modelling

- Modelling has been done under the disguise of template matching for a long time in the community
  - Templates consist of the mean (vector) and (co)variance(matrix) of a (multivariate) Gaussian that represent (a) leakage point(s)
    - Pro: captures potentially the full leakage, Cons: lots of traces, matrix not invertible
  - This is equivalent to a multinomial representation in which one includes all interaction terms
    - Pro: can test which interaction terms are statistically significant, and thus remove all others, requires potentially fewer traces for a very good estimation of the relevant terms using regression

Leakage modelling, cont.

- Beyond the choice of statistical technique, there is a big question about the ‘level of abstraction’, and how to integrate models into a design flow
  - Bayrak et al.’s approach requires to instrument each new piece of code before it can be analysed
- (Maybe) a much better idea: choose Assembly level code snippets to determine and model leakage
  - Length and composition of sequences, choice of leakage points within the corresponding traces, what potential effects to include, etc.

- Leakage modelling methodology
  - Initial scouting of individual instructions with the aim of clustering instructions – verification by cross-checking with known architectural information (grey box modelling)
  - Generation of controlled sequences of specifically designed instruction triplets (with the target instruction in the middle) to produce data for modelling
  - Model: $\mathbf{y} = \delta + [\mathbf{I_p} | \mathbf{I_s} | \mathbf{D} | \mathbf{DxI_p} | \mathbf{DxI_s}]\,\boldsymbol{\beta} + \boldsymbol{\varepsilon}$

$$\mathbf{D} = [\mathbf{O_1} | \mathbf{O_2} | \mathbf{T_1} | \mathbf{T_2}]$$
$$\mathbf{DxI_p} = [\mathbf{O_1xI_p} | \mathbf{O_2xI_p} | \mathbf{T_1xI_p} | \mathbf{T_2xI_p}]$$
$$\mathbf{DxI_s} = [\mathbf{O_1xI_s} | \mathbf{O_2xI_s} | \mathbf{T_1xI_s} | \mathbf{T_2xI_s}]$$

  - We test significance for the terms with F-test and look at R^2
  - We also test for effects of board, register choices, and the potential of higher order terms (included then for some instructions)

- Leakage modelling methodology
  - Model:
    - I_p (previous instruction), Is (subsequent instruction)
    - D (dummies for bits and transitions of operands)
    - DxI_p (HW and HD terms plus interactions with previous instruction), DxI_s

$$\mathbf{y} = \delta + [\mathbf{I_p} | \mathbf{I_s} | \mathbf{D} | \mathbf{DxI_p} | \mathbf{DxI_s}]\,\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

$$\mathbf{D} = [\mathbf{O_1} | \mathbf{O_2} | \mathbf{T_1} | \mathbf{T_2}]$$

$$\mathbf{DxI_p} = [\mathbf{O_1xI_p} | \mathbf{O_2xI_p} | \mathbf{T_1xI_p} | \mathbf{T_2xI_p}]$$

$$\mathbf{DxI_s} = [\mathbf{O_1xI_s} | \mathbf{O_2xI_s} | \mathbf{T_1xI_s} | \mathbf{T_2xI_s}]$$

ELMO

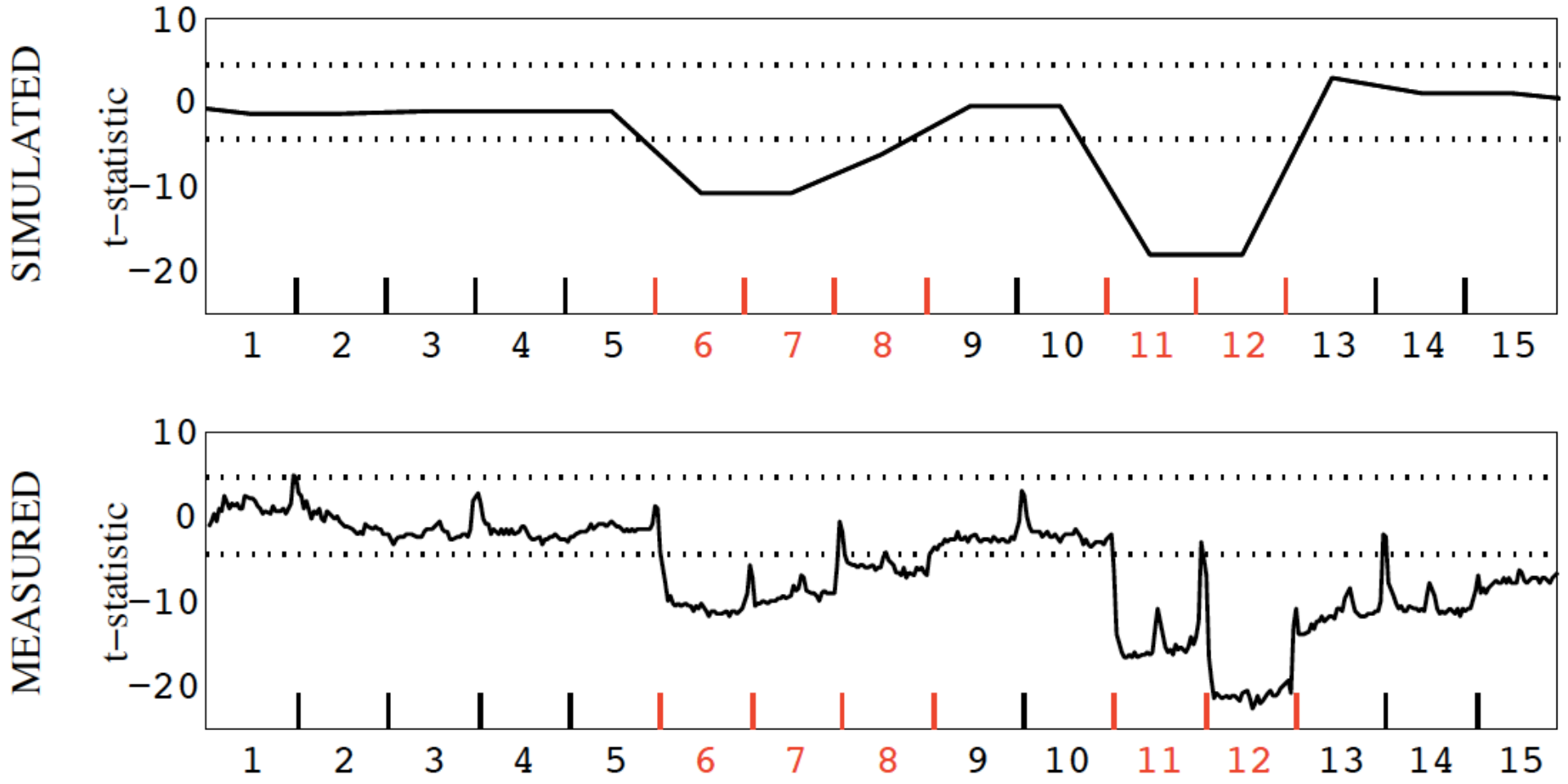
These models were integrated in an open source, instruction set emulator for the target architecture (an M0):

- We piggy back on the ‘Thumbulator’ data flow graph to extract the input and output data for each instruction as it is executed on the target architecture
- We analyse triplets to `plug in’ the corresponding leakage model from our database of models
- This enables us to produce instruction (or cycle) accurate leakage traces for **arbitrary code**

ELMO

- ELMO can thus produce nearly `best case' leakage traces
  - They are noise free, but limited by our model choices
- ELMO has functionality to automate leakage detection
  - At present we only facilitate a t-test
    - ELMO however enables developers to unanimously attribute leaks to instructions
    - It instruments leakage detection according to best practice by interleaving `acquisitions' to avoid any potential statistical bias
    - It can (in principle) select the appropriate number of `acquisitions' to achieve a specific power of a test
    - It significantly speeds up second-order leakage detection because it can attribute masks to instructions, and thus `knows' which pairs of points to select

ELMO traces


SIMULATED
t−statistic
10
0
−10
−20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
MEASURED
t−statistic
10
0
−10
−20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ELMO

- ELMO can also trace `masks' through assembly code and can thus point out if some instructions are unmasked or if masks get taken off
- In principle (tested on the AES in mBed TLS) one can write C code, compile to ARM Thumb, and then analyse this via ELMO
- In principle the tool can be used to randomly insert instructions that foil HW leakage and lower other leakage (certain sequences can enhance or worsen leakage of a target) (tested on AES)
- In principle any of the published work would be much facilitated by ELMO

Next steps

- What's missing?
  - We did not profile address leakage
  - We did not exhaust all Thumb instructions
  - We made no effort to investigate if there is leakage from within the multiplier
  - We did not entertain how to even decide if longer sequences would be more adequate

  Clearly this is not an industrial tool, it is no more than a promising first step.

Next steps

- What else is missing?
  - ELMO is a ‘standalone’ tool and not part of a compiler toolchain, thus it assumes that a developer can identify potentially critical pieces of code and run it through ELMO
  - The ideal solution for the non-expert would be to be able to annotate higher level code (i.e. C for most embedded systems), and then for a tool to do the rest
- We have an ongoing collaboration with Embecosm, the one and only (UK) compiler company that has realised the disruptive power that a security aware compiler could have

Embecosm

- They currently work on some ideas re automating techniques that ensure constant time as well as cache safe implementations of symmetric primitives on embedded devices, with the goal to upstream the results
  - In the future gcc-arm should include options that automatically improve the security of code
  - We hope to learn from this process and thus scout out the appetite for more leakage-aware compilation options

Wrap up

- Cryptography is not only for Cryptographers
- Making cryptography work in practice is a huge challenge
- Compilers are integral to software development and they **should** be leakage aware
- **ELMO** is open source and we are restarting work on it:

github.com/bristol-sca/elmo

"Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages", Usenix 2017, McCann, Oswald, Whitnall