Rosita++: Automatic Second-Order Leakage Elimination from Cryptographic Code

Madura A. Shelton[1], Niels Samwel[3], Łukasz Chmielewski[3,4], Lejla Batina[3], Markus Wagner[1], Yuval Yarom[1,2]

1. University of Adelaide, AU. 2. Data61, CSIRO, AU. 3. Radboud University, NL. 4. Riscure, NL.

Introduction

As devices interact with their operating environment, they can emit signals that are correlated to the work it is performing at a given time. This process might leak information that are required to be kept secret even-though the mathematical primitives are proven to be secure. These emissions can happen in any measurable quantity and are referred to as 'side channels'. Past research show that power analysis, electro-magnetism, acoustic, and photonic based side channels can be used to recover secrets from cryptographic devices.

Countermeasures that were proposed against such attacks included many techniques including hardware designs to reduce emissions, adding noise to hide the signal, employing polymorphic code, and information masking techniques. The most common among these is masking, where secret values are split in to two more shares by mathematically combining them with random values. Masking schemes ensure the theoretical security of a cryptographic implementation. Yet, despite that masking schemes often fail in practise to provide the theoretical guarantees.

The main cause for such failures is that an assumption called the **Independent Leakage Assumption** (**ILA**, proposed by Renauld et al.) is breached in practise. This happens due to unintended interactions that happen between the intermediate values of an implementation. The causes for such effects are micro-architectural details that are specific to the devices, and mainly stem from the reuse of internal registers without clearing them.

Fixing such leakage manually is a tedious job and requires expertise in many disciplines. In Rosita, Shelton et al. demonstrated a code rewriting engine that could apply code fixes to such implementations by employing the power traces generated by an emulator. This method avoids the time that is spent on physical experiments therefore is much faster than the physical process.

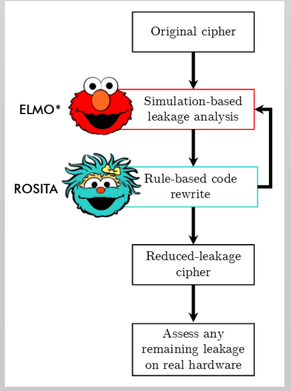

Figure 1. Rosita(++) workflow

Security of d-order implementations

In Boolean masking, a d-order implementation is an implementation that splits a sensitive value in to d+1 shares (v_0,v_1,v_2,…v_d). This is achieved by combining the sensitive value (v) and d random values (v_1,v_2,…v_d) such that the first share (v_0) is,

$$v_0 = v \oplus v_1 \oplus v_2 \oplus \cdots \oplus v_d$$

- To break a d-order implementation an eavesdropper needs to reveal all d+1 shares.
- By using a single probe we can sample power values when the CPU processes different intermediate values.
- Theoretically, a first-order masked implementation should not show leakage with a single probe. But due to ILA breaches information from more than a single intermediate value is available through a single sample.
- Using a single probe is the same as a **univariate** acquisition were we acquire samples of a single variable at a set interval,
- The samples acquired in a univariate setting can be used for **bivariate** analysis by considering the combinations between pairs of samples (i.e.${}^{n}C_{r}$). This grows quadratically with respect to the sample count.

Q Can we observe second-order leakage on a second-order masked implementation? Yes!

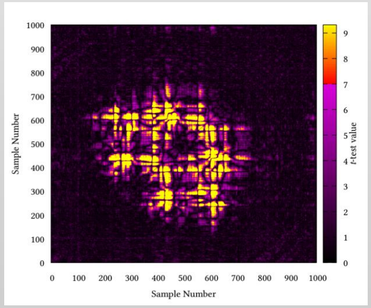

Figure 2. 3 share Xoodoo (chi function) second-order leakage (TVLA)

As shown in **Figure 2** and **Figure 3** we evaluated two second-order masked implementations by using bivariate Test Vector Leakage Assessment (TVLA). Before running the second order evaluation we made sure that the implementations do not show leakage in the first order by running first-order TVLA on the same traces. The number of traces that were used was 2 million for each implementation. The traces were collected from an ARM Cortex-M0 MCU on STM32F030 Discovery board.

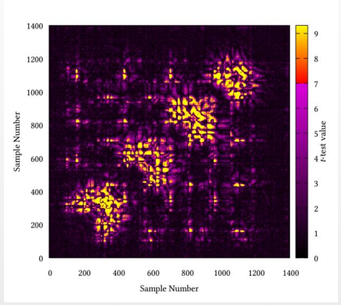

Figure 3. 3 share Present (S-Box lookup) second-order leakage (TVLA)

Results

Q Can we fix them? Yes*

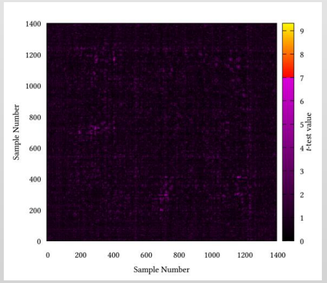

Figure 4. 3 share Xoodoo second-order leakage (TVLA)

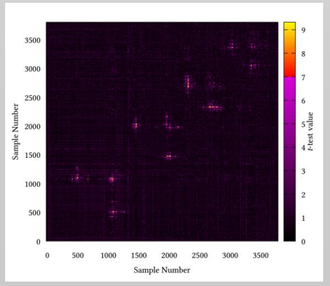

Figure 5. 3 share Present second-order leakage (TVLA) *

Methodology

The root cause detection for Rosita++ employs two different methods. The first, eliminates each component of the power model and re-evaluates the leakage to find offending components, when found the previously removed components are used as root causes to drive the code fixed. The second method uses a Monte Carlo simulation to find the sources of leakage.

Component elimination We the power model proposed by McCann et al in ELMO with some modificatoins. It comprises of a number of components that depend on the values from each execution trace of a virtual machine. These values can depend on a single share or multiple shares in combination. This combined with another share makes all three shares complete and thus ends up leaky. A scenario of the power values leading to a leak is shown in **Figure 6**.

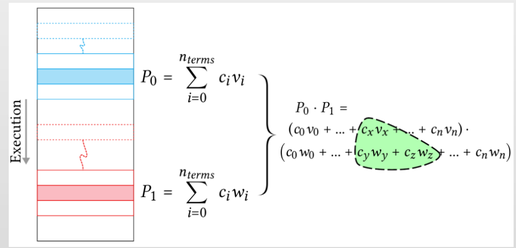

Figure 6. Components contributing to a leak

Monte Carlo We use the slower Monte Carlo method to find leakage which runs for a set number of iterations when the first method fails. In each iteration a random set of components are selected and the leakage is evaluated using TVLA. If the power values from a set of components becomes leaky, a score of 1 is awarded to each component that was in the set. This continues for a set number of iterations and finally all outliers are selected from the list of scores for each component. Each component that ended with a high score has a higher probability of contributing a share or a combination of shares to the power model.

Conclusion

We demonstrated that by using new methods for root cause detection that it is possible to apply code fixes to second-order masked implementations. These methods are not tied to the model that we currently use in Rosita++ and can be used with any other model. Our target was to fix all detected leakage and as such the improvement of performance was left as future work. The performance can be improved by providing Rosita++ with the global view of code fixes where avoidable code fixes could be removed by optimizing the code fixes application process.