Practical Evaluation of Masking Software Countermeasures on an IoT processor

David McCann and Elisabeth Oswald
firstname.lastname@bristol.ac.uk

University of Bristol

Abstract. Implementing cryptography on Internet-of-Things (IoT) devices, that is resilient against side channel analysis, has so far been a task only suitable for specialist software designers in interaction with access to a sophisticated testing facility. Recently a novel tool has been developed, ELMO, which offers the potential to enable non-specialist software developers to evaluate their code w.r.t. power analysis for a popular IoT processor. We explain a crucial extension of ELMO, which enables a user to test higher-order masking schemes much more efficiently than so far possible as well as improve the ease and speed of diagnosing masking errors.

1 Introduction

Since the discovery by Kocher et al. [6] that the power consumption or EM field of a device can be used as a side channel to determine secret information being processed by a device, understanding and securing cryptographic devices against power analysis attacks has become an important design consideration. This is particularly true where a potential adversary has physical access to the device, such as with the emerging Internet of Things (IoT) devices.

Despite the threat and significance of power analysis attacks, the number of easy to use tools available to developers to aid them in understanding the vulnerability of their implementations to these attacks remains limited. Whilst there has been some progress towards automating the evaluation of leakage of implementations by analysing certain properties of the code for leakage vulnerabilities ([9] [1] [3] [4] [2] [11]), these methods are limited by the fact that they do not consider the leakage of the actual device itself, but rather work on an assumption about the leakage of it (such as that it will leak via a given leakage model) which may or may not reflect the real world leakage of the device.

Hence, specialist evaluation labs need to be consulted by developers who are concerned about the risks of power analysis attacks. These labs take real world traces from devices and perform their own evaluation methods on the measured traces [5] [13], which incurs a significant time and cost overhead. This demonstrates a clear need for a set of techniques (tools, methods) that Internet of Things developers can use to test their own code (at least for some attacks), without having to consult a specialist lab.

Recent work by McCann et al. [8], has provided a novel methodology for profiling a microprocessor at the instruction level. The resulting leakage profiles allow a user to generate accurate real world power traces of a given device statically (see Section 2.1 and [8]). The authors implement their approach for the ARM Cortex-M0 processor (this results in a tool called ELMO), and demonstrate the accuracy and usefulness of their tool. One of the examples that they include in their work is that of testing a masked implementation of the Advanced Encryption Standard (AES).

We believe that ELMO offers a wealth of potential applications for automating the leakage detection of code, in particular the testing of the popular masking countermeasure. Recent work makes progress along these lines: [11] point out that rather than utilising complex formal verification techniques to validate masking countermeasures, one could 'simply' apply a leakage detection technique to a masked implementation. Their approach then utilises an ad-hoc power model, which may or may not accurately reflect the reality of an implementation on a real device. This is clearly a shortcoming of their proposal and can be addressed (in the case of the M0) by utilising ELMO. However, a more important problem remains unsolved: testing masked implementations of order d requires (before statistical leakage detection can take place) to compute the product of the power consumption of all d mask-related intermediate values. However these mask related intermediate values are not necessarily known and thus strictly speaking one needs to compute the product of all combinations of d points across an entire trace. This effort grows exponentially with d, and this means leakage detection even for low leakage orders is computationally very expensive and thus seemingly unsuitable for any 'nice' push button solution.

Our Contributions

In this paper, we propose to mine some of the potential of ELMO by implementing our own mask flow methodology that automatically detects the number of masks being used in an implementation and the specific instructions (which correspond to power points) that use the individual masks. This allows us to only **choose the power leakages** related to the **same (set of) masks**, thus significantly reducing the complexity and time taken to perform the higher order leakage detection test, as well as provide mask information of an implementation that could be used be used by a developer to debug masked implementations. As a demonstrator we provide an automated first order and a highly efficient second order leakage detection tool for the ARM Cortex-M0.

1.1 Previous Work

As well as our work building on [8], previous work on automating leakage detection includes [1], where the flow of key bits covering each bit of the state is traced through an implementation and the number of key bits used in the computation of each state bit used as a measure of security. In a similar way, [9] also provide

an information flow method of tracing secret information through the state of an implementation, albeit for the purposes of identifying instructions to mask.

Work that also aims to automate the robustness of masked implementations includes [3] which provides a tool that identifies operations dependent upon secret key information but also those which are dependent on additional random information (as would be the case with masked operations) and uses a SAT solver to determine if the masking scheme has been applied correctly. [4] also pursues a similar approach based on SMT solvers to develop a metric for the security of masking schemes and [2] provides an approach using EasyCrypt to verify the implementation of higher order masking schemes.

The closest work to that in this paper known to the authors is [11], where traces are simulated based on a given leakage model and then the fixed vs random leakage detection test used on those traces. Although this method uses simulated power traces, the traces are generated according to a pre-defined leakage model which must be chosen by the developer and which may or may not accurately reflect a given real world device.

2 Background

2.1 ELMO

ELMO [8] is a tool that provides accurate albeit simulated traces for a subset of the Thumb instruction set for the ARM Cortex-M0 processor. ELMO draws on parametrised models (derived from carefully analysing the M0) that correspond to the sequences of three instructions. These models are then used with an emulator of the Thumb instruction set, which provides the data flow information and instruction types, with which the models can be instantiated.

ELMO traces therefore offer a unique advantage in that they provide ease of use and timing accuracy, however they also display the real world hardware leakage of an actual device in the same way as measured traces.

2.2 Leakage Detection

Leakage detection methods are ways of analysing code to determine if the code will leak information. A popular leakage detection method is known as the fixed vs random test [5]. This method compares a set of traces with a fixed key and a fixed input with a set of traces of the same size with the same fixed key but with random inputs. The sets can be compared using Welch t-test which is given in Eq. 1, where X_A and X_B are the averages of the two sets of traces A and B, S_A and S_B the respective standard deviations and N_A and N_B the respective number of traces in each set.

$$t = \frac{X_A - X_B}{\sqrt{\frac{S_A^2}{N_A} + \frac{S_B^2}{N_B}}} \tag{1}$$

If the resulting test statistic t is greater than a given confidence threshold, the null hypothesis that the two sets are taken from the same sample can be said to be rejected and thus the implementation (or part of the implementation) is considered leaky. The fixed vs random test is useful in determining whether a given implementation leaks information or not, however it is worth noting that it does not provide information on how this information leaks and via which leakage model. It can only capture leakage in the first moment in the traces, and thus, when higher order masking is analysed, traces need to be preprocessed (i.e. traces need to produced that contain the product of all combinations of d trace points).

2.3 Masking

A popular countermeasure against differential power analysis attacks is known as masking. Masking provides security by making the power consumption of the state of the cryptographic algorithm independent of the sensitive intermediate values that are being processed. This is achieved through combining the state of the algorithm with random data (the masks) such that the power consumption of the sensitive state becomes masked, and thus the implementation becomes provably secure [12] [10]. The number of masks per intermediate value determines the masking order d.

Although masking provides provable security against power analysis attacks up to the masking order d, masked implementations are still vulnerable to power analysis attacks targeting an order higher than d. In these attacks, multiple power points that represent separate values of the state using the same mask are exploited to remove the random independence on the power consumption provided by the mask and thus allow the sensitive values of the state to be exploited.

3 Automating Leakage Detection

In the following two sections we provide details on the extensions we make to elmo and how these extensions can be used first for automating a simple first order fixed vs random test and then for performing higher order leakage detection tests against masking. Irrespective of which of these masking orders is being evaluated, the first stage of performing a leakage detection test is to generate the necessary traces to be tested.

3.1 Generating Traces

In our extension of ELMO, the generation of the traces and the selection of the plaintexts, keys and masks (if they are used) are left to the programmer and so need to be programmed into the Thumb binary file. In order for a developer to do this, we have developed a number of built in functions to ELMO that enable a developer to start and stop the trigger (which indicates when to start and stop

the trace) as well as call the fixed vs random test routine. In addition to these, there are also functions to read data into the Thumb program from ELMO and vice versa to enable a developer to supply data to and from the program at runtime.

In order to perform a fixed vs random test, traces with a fixed key and fixed plaintext must be generated as well as an equal number of traces with a fixed key and random plain texts. For ELMO to recognise which traces are which, the traces must be generated with the first half of the acquisition being the traces with a fixed key and fixed plaintext (as specified by the programmer) and the second half the traces with a fixed key and random plaintext, where the start and end of the trace is determined by calling the start trigger and end trigger functions respectively.

As well as being responsible for generating the order of the traces, the developer also has the responsibility of specifying how many traces are to be used in the analysis by starting and stopping the trigger the correct number of times. ELMO automatically takes this number to be half of the total number of times the trigger was started and stopped (with the first half being the fixed traces and the second the random traces). Following the generation of the two blocks of traces, the function to call the fixed vs random test routine should be called.

3.2 Automated Fixed vs Random

When ELMO encounters the function to call the fixed vs random test routine, it automatically performs a t-test on the fixed and random traces. The results of the fixed vs random test are generated and stored in a file which has each output value of the t-test (corresponding to each instruction) as a line in the file. As ELMO also outputs the instructions that were executed, instructions which are identified as being leaky (by having a t-test value of either > 4.5 or < -4.5, which for large N (> 5000) means the null hypothosis is rejected at the 99.999% confidence interval) can be easily traced to the exact instruction.

3.3 Spotting Flaws in Countermeasures

Carrying out the fixed vs random test in this way easily enables a developer to see how leaky the code he or she is trying to develop is and which instruction sequence the leakage is coming from. Another advantage of this method however, as highlighted in [11], is that it allows the developer to assess the implementation of side channel countermeasures and whether these have been applied properly. If a first order countermeasure (such as first order boolean masking) has been implemented and leakage is detected in the (first order) fixed vs random test, this suggests that the countermeasure has not been implemented correctly. Using the leakage information produced by ELMO in this way developers can also easily assess the robustness of their countermeasures.

4 Evaluating Higher Order Leakages Against Masking

Where masking countermeasures are used, evaluating second order leakages against the masks requires preprocessing of the points of the trace which use the same mask. One method of doing this is to multiply the points of the trace which use the mask together [7].

Because normally one does not know which trace points correspond to which masks, one has to exhaustively compute all combinations of d trace points (excluding symmetries). Thus already in the case of a simple first order masking scheme, where only combinations of two points need to be considered, the length of preprocessed traces grows to $(n^2 + n)/2$ (where n is the length of the original traces).

ELMO traces however are different to real world traces in this respect as, although containing the leakage of multiple points in the traces, each leakage point is specifically for a triplet of instructions. This means that we can be certain of the time point of the data dependant power consumption of a single instruction and thus if that instruction is masked, the time point at which any mask dependent power consumption occurs. Another feature is that we are emulating the functionality of the program being evaluated and so we can easily create a data flow model to map the flow of masks through the program and so be able to automatically detect which instructions (and thus corresponding power points) are masked with the same mask.

This means that we can significantly reduce the complexity of carrying out higher order leakage evaluations by using our own mask flow method to map the flow of masks through the program. By automatically detecting how many masks are used in the program and which instructions they are used with we can efficiently perform a fixed vs random test on only the relevant points for all masks that are present, greatly reducing the value of n. This provides a robust and efficient method of higher order leakage detection.

4.1 Mask Flow

The mask flow method works by modelling each mask as a boolean matrix of $n \times 32$ bits, where 32 is the word size of the ARM Cortex-M0 and n is the number of possible independent mask bits that could mask each bit of the word size. A 1 in the matrix indicates that a specific random bit (as determined by the location on the y-axis) masks the corresponding bit of the word on the x-axis. A 0 means that it does not. One matrix is generated for each operand of an instruction and as each matrix contains all the masking information of each bit of the operand, each unique mask, and it's level of security (as understood through the number of independent random bits masking each bit of the word), can be deduced for each instruction.

To model the flow of masks through the program, a $n \times 32$ bit matrix needs to be generated and stored for each operand of each instruction. This is easily done by including two matrices (one for each of the operands) in ELMO's linked list structure of the data flow model that stores the information of the data being

processed by each of the operands and the instruction type of each instruction that is used in simulating the power consumption of the instructions.

This matrix is able to store the mask information for each instruction's operands, however if we are to map the flow of the masks through the program, we also need to have a method of mapping the flow of masks through the state of the program. This includes the registers in which the output of operations are stored as well as RAM where data can be written or stored to. This is achieved by generating $n \times 32$ bit matrices for each register in which we store the output mask information. We also generate an n by m matrix, where m is the size of RAM, in which we store the mask information for each bit of the state which is stored to memory. In this way, when data is read from or written to memory, the corresponding mask information for each bit of memory being read from or written to is also operated on in the same way as the data.

Finally, we need to adopt a set of rules which describe how an operation on two masked operands affects the mask of the output. These rules need to allow the masks to be tracked properly through the program in such a way that the output model of the mask, in the form of its matrix, correctly represents mask of the output.

We implement our mask flow analysis for use with boolean masking, where the random masks are added and removed from the state using the exclusive-or operation. We therefore adopt the rules shown in Table 1 to model the different instructions according to their types.

Operation	**Matrix Returned**
Load	Load mask matrix from memory into register.
Store	Store mask matrix in register to memory matrix.
Shift Left	Shift matrix left by value of data shift.
Shift Right	Shift matrix right by value of data shift.
Rotate Right	Rotate matrix right by value of data rotation.
Exclusive-Or	Exclusive-or operation of all corresponding bits of the two operand matricies.
Other Arithmetic Operations	Zero matrix containing no mask information.

Table 1: Rules for mask matrix output for operations on mask matricies.

For memory operations the mask information is simply loaded or stored to the memory matrix, ensuring that mask information is not lost during memory operations. For shift and rotate instructions the mask simply shifts with the data to ensure that the mask reflects the correct bits of the data which are masked. As we are analysing boolean masking, the exclusive-or operation exclusively-ors all bits of the two mask matricies of the operands. This insures that all operations that would lead to the addition, subtraction or changing of a mask are taken into consideration. As we are only considering boolean masking, for simplicity all other arithmetic instructions return a zero mask matrix.

4.2 How to use the mask flow

The first stage of using the mask flow analysis is to initialise the matrix of the memory location of the mask in RAM. This is essential as it introduces the mask into the program. In order to do this, we developed an inbuilt function in ELMO that initialises the mask flow (the initialise mask flow function). This function assumes an 8 bit mask where each bit in the mask is random and independent of all other bits in the mask. This assumption is important for the mask flow analysis as it needs to know whether each bit of the mask applied to each bit of the state is the same random bit used elsewhere or a new random mask bit as this will affect the security level of the mask.

The initialise mask flow function therefore effectively creates a diagonal line of ones in the matrix which is eight bits by eight bits. This indicates that each of the eight bits of the memory location are masked by one bit of an independent mask. In order to specify different random masks, we developed a related ELMO function that specifies the bit number to start from on the y axis of the matrix (the mask flow start function). This allows multiple independent mask bits to mask a single bit of the state by recalling the function but specifying a different start point.

This is shown in Figure 1 and Figure 2. Both of these show the mask matrix of a 32 bit operand with n (the size of the number of possible mask bits) equal to 16. Figure 1 shows 32 bits of memory or a register or operand that is masked with a single 16 bit mask that is used twice so that a single bit of the mask is used on two bits of the 32 bit state. The overall effect here is that each bit is masked with a single bit however the same mask bits are only reused once. To initialise this mask configuration, you could run the initialise mask flow function four times for the memory address of each byte, using the set mask flow start function to change the start point for the initialisation to 8 from 0 for bytes 2 and 4.

Figure 2 shows the state when the 16 bit mask shown in Figure 1 is split into two eight bit masks. Here there are still 16 bits of random mask but, unlike in the other case, the same masks are used four times so that four bits of the state are covered by the same mask bits. The advantage of this method however is that each bit of the state is now covered by two mask bits rather than 1. The result of this is that the secret is divided into three shares, which can provide higher levels of security. This configuration could be created by calling the initialise mask flow function eight times, twice for each byte of the 32 bits with each time having a start point of 0 and 8 respectively.

Once the mask flow has been initialised, traces can start to be taken for the program to be evaluated. For the first trace generated, each instruction stores the two matrices associated with each operand in the linked list structure that stores the data flow and instruction type information. After the first trace has ended (as signalled by the end trigger function), the mask information is then analysed to detect if and where any of the same masks are used in the program, where an instruction is deemed to use the mask if at least one of its operands does. As our trace generating method uses the data of the previous instruction operands to

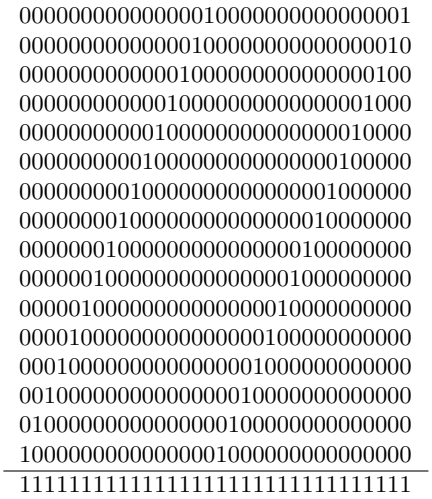

Fig. 1: Single 16 bit mask

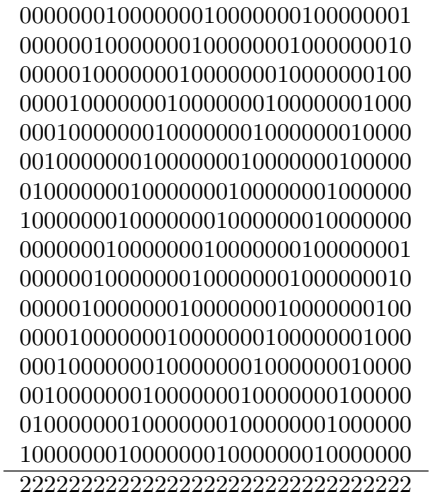

Fig. 2: Two 8 bit mask

determine its power consumption, the instruction following a masked instruction will also be influenced by its masked data. For this reason we also include the masked instruction's subsequent instruction in the masked instruction index list.

Once this list is compiled and we have the indexes of the masked instructions and their respective masks, we can then use this information to ensure that only instructions affected by the same masks are included in the preprocessed traces for higher order analysis. If masks are detected, this happens automatically after the first order fixed vs random where a fixed vs random test is carried out on the preprocessed traces for each of the masks. If no masks are found, then the analysis ends after the initial first order fixed vs random test.

4.3 Other uses of mask flow output

As well as being a useful tool in making the preprocessing stage less computationally intensive for higher order leakage evaluation, the output of the mask flow analysis also provides useful information for debugging a masked implementation by providing a list of the mask numbers used in each operand of each instruction. Using this information along with the assembly instructions of the program, a developer is able to see exactly which instruction has used which mask. If first order leakages are detected in a masked implementation, a developer can assess which instruction is leaking and which masks (if any) are being used for the instruction.

In addition to this, ELMO can print the matrix output of each mask found during the mask flow analysis to a file. This allows a developer to see the exact nature of the masks and whether implementation errors have inadvertently changed their configuration to produce errors in the code or render it less secure.

5 Examples of use

We here provide an example of how ELMO works by using it to analyse one round of AES masked with two 8 bit masks (one for the key byte, m_k, and one for the state byte, m_p) that is implemented in Thumb assembly. The masking method works by first recomputing the SBox to ensure that when the SBox value is loaded, the correct SBox value and mask is returned for the masked statebyte, before calling the AES round function a single time. The trigger is started and stopped before and after calling the AES round function.

The traces are taken in two sections, the first with the fixed key and plaintext and the second for the fixed key and random plaintext. The masks and random data that is used is stored in the file which is accessed using the read data function which reads data into the program from a given file. The file is reset using the reset datafile function after taking the first section of traces (the fixed traces) to ensure that the masks used for both sets of traces are the same.

The mask information is set for each of the masks using the initialise mask flow function with the addresses of the two masks. The set mask flow start function is also used to identify these two different masks as independent masks that provide an independent random mask bit for each state bit that is masked. In this implementation the start bit of the initialisation was set to 0 for m_p and 8 for m_k.

Once the program has been compiled into a Thumb binary, it can be run by ELMO which immediately begins the process of generating the traces. The mask flow analysis is only carried out after the first trace where the number of masks used and their respective instructions used with is determined and then stored for the later analysis. After the traces have been generated the fixed vs random analysis is carried out as indicated by the call to the fixed vs random function.

First Order Leakage Detection Once the traces have been generated, the first order leakage detection test begins and informs us if any leaky instructions have been found. The results are also printed to a file which can be used in conjunction with the assembly program file and mask information file to debug the source code.

In the case of our example we are told that 22 instructions leak via first order leakage. This might seem surprising as we have an implementation that should be masked to protect the implementation against first order attacks, indicating that there is either a bug in the implementation of masking or there are other factors contributing to leakage that the developer did not take in to account. Table 2 shows one of these leaky instructions (shown in red) with the instruction before and after (as ELMO analyses instructions in sequences of three) along with the masks used on each of the operands of the instruction.

By examining this sequence we can see that both operands of the `eors` instruction are masked with masks 3 and 2 respectively which, if we take the instruction insolation, should eliminate all leakage from the instruction itself. If however we examine the prior instruction (the `ldrb`), we can see that the mask

Instruction	Operand 1 Mask	Operand 2 Mask
ldrb	0	2
eors	3	2
ldrb	0	3

Table 2: Leaky instruction triplet with leaky instruction in red.

for operand 2 is the same as that for operand 2 of the `eors` instruction (mask number 2). We could therefore conclude that it is likely that interactions between the hamming distance of these two operands has caused the masks to intereact in such a way that the underlying data is leaked. This could then be amended by, for example, inserting a dummy instruction that does not use masked operands between the first `ldrb` and the `eors` instructions, preventing this interaction of the masks and the resulting leakage. In this way the mask flow analysis of ELMO can be used to easily understand and help remove subtle flaws in masking implementations.

Second Order Leakage Detection If masks are identified, ELMO will proceed to preprocess and evaluate the traces mask by mask. In the case of our example a number of distinct masks are found as the implementation works with a number of representations of the two masks we defined at the beginning of the program. We select mask number 3 to evaluate which represents the single mask byte m_p in our example which masks the plaintext.

Without the mask flow analysis, the preprocessing stage would require each instruction's power profile to be multiplied by every other instruction's, leading to extremely large traces of size $(n^2+n)/2$. In our case the size of the trace is 452 and so we would end up with preprocessed traces of size 102378. However, as our mask flow analysis identifies where the masks are used in the trace, we are able to only carry out the preprocessing on only the relevant masked instructions. For mask number 3, this gives us a trace size of 6441, around 16 times smaller than preprocessing in the naive way. This much reduced size of the the trace to be evaluated makes the second order analysis significantly faster and requires significantly less memory.

Carrying out the fixed vs random test on these preprocessed traces, ELMO informs us of the number of preprocessed points which leak. In our case for mask number 3 this number is 600. The output of the t-test is stored to a file and can then be used along with the assembly output and mask information files to identify the specific instruction pairs and thus the instructions that are leaking. This can be done for all detected masks in the program.

6 Conclusion

Testing implementations of (higher-order) masking is very costly if the association between (sets of) masks and leakage points is unknown, which is normally

the case. However, we observed that using our extended version of ELMO enables a developer to determine this association and thus leakage detection testing as means to verify masking implementations can be done efficiently in this context, with a novel mask flow technique that we introduce in this paper. Furthermore, we show that by using the mask flow and leakage detection output from ELMO, subtle errors in masking schemes can be more easily identified, understood and ultimately resolved.

References

1. G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 81:1–81:6. ACM, 2013.
2. G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.
3. A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In G. Bertoni and J. Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.
4. H. Eldib, C. Wang, M. M. I. Taha, and P. Schaumont. QMS: evaluating the side-channel resistance of masked software from source code. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 209:1–209:6. ACM, 2014.
5. G. Goodwill, J. J. B. Jun, and P. Rohatgi. A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop, 2008.
6. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
7. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
8. D. McCann, C. Whitnall, and E. Oswald. ELMO: emulating leaks for the ARM cortex-m0 without access to a side channel lab. *IACR Cryptology ePrint Archive*, 2016:517, 2016.
9. A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2012.
10. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013.

11. O. Reparaz. Detecting flawed masking schemes with leakage detection tests. In T. Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2016.
12. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In S. Mangard and F. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
13. T. Schneider and A. Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.