An Adaptable, Modular, and Autonomous Side-Channel Vulnerability Evaluator

Michael Zohner*, Marc Stöttinger*, Sorin A. Huss*, and Oliver Stein†

*Integrated Circuit and Systems Lab (ISS), Technische Universität Darmstadt,
Email: {zohner, stoettinger, huss}@iss.tu-darmstadt.de
†Bundesamt für Sicherheit in der Informationstechnik (BSI),
Email: oliver.stein@bsi.bund.de

***Abstract*—Computer aided design (CAD) tools are fundamental for ensuring a short time-to-market in nowadays chip design. However, while CAD tools support the development of efficient designs, they fail to support the designer with regard to side-channel security. In order to better assist the designer, we propose the AMASIVE (*Adaptable Modular Autonomous SIde-Channel Vulnerability Evaluator*) framework that autonomously identifies side-channel weaknesses in a design. Instead of implementing some countermeasures straight forward, AMASIVE highlights several design-specific vulnerabilities by exploiting an adaptable attacker model. Thus, we aim at supporting the designer in identifying countermeasures that are appropriate for the device's application scenario. In this contribution we introduce the general concept of this novel framework and demonstrate its application to a hardware implementation of the block cipher PRESENT.**

I. Introduction

In the current design process the resistance of devices to side-channel attacks is still rarely analyzed. Countermeasures against side-channel attacks are selected after design completition, which makes this important step more dependent on the available resources than on the actual security requirements, cf. [1]. The reasons for a rather late consideration of side-channel vulnerabilities are manifold such as the short time to market, unpredictability of side-channel properties during the design process, or an unidentified side-channel information leakage of the algorithm. Thus, in many design processes, side-channel vulnerabilities are detected too late to be taken into account when deciding on the design architecture. This situation is rather unfortunate, because the designer has access to an enormous amount of knowledge about the device. So she or he is in the position to perform a more accurate side-channel analysis than an outside attacker ever could. However, identifying, combining, and utilizing this knowledge is a difficult task and requires experience in side-channel analysis next to hardware design skills. Recently, there have been a few contributions aimed to support the designer in securing a device against side-channel attacks. In [2] Standaert et. al propose a unified framework to cope with side-channel issues of implementations. This work introduces various formalisms and methods to identify side-channel leakage in order to compare the vulnerability of different designs. The first step towards an intelligent insertion of side-channel related countermeasures during the design phase of a hardware implementation is proposed in [3]. Here, Regazzoni et al. introduce a tool set, which may automatically transform each element of the design netlist into the side-channel resistant logic style MCML. The exchange of netlist elements is based on an evaluation to identify security sensitive parts, which need to be implemented in this logic style. Countermeasure implementation for side-channel attacks in software implementations is tackled by two approaches, which provide automated code analysis and countermeasure integration, cf. [1], [4]. The work in [1] applies a side-channel attack to the assembled code and utilizes an estimation of the mutual information [2] to evaluate and highlight vulnerabilities of a code section. The second approach, proposed by Moss et al. [4], on the other hand, features an automatic insertion of masks by evaluating the secrecy requirements of an intermediate value in the program code.

In two of these three approaches the designer has to formulate the leakage assumption for the side-channel vulnerabilities, which presumes a designer with considerable expertise in side-channel attacks. The proposal in [1] overcomes this problem by utilizing an estimation of the mutual information as side-channel distinguisher. Since the selected leakage model has the greatest impact on the success of the attack, cf. [5], [6], it would be reasonable to select the best model for each attack scenario. In addition, the leakage model relies on an adequate hypothesis function in order to identify the vulnerabilities in the design.

In this contribution we introduce the AMASIVE framework, which supports the designer in developing side-channel resistant devices. In contrast to existing contributions, we do not restrict ourselves to the automatic implementation of a specific countermeasure [7], [8]. We rather focus on a supportive autonomous identification of side-channel vulnerabilities and outline them to the designer so that the most adequate countermeasure can be chosen. The strength of the framework is the adaptability to various implementations by utilizing several attacking procedures, leakage models, and side-channel distinguishers. Similar to [2], we exploit an attacker model for our analysis in order to secure an implementation specifically to its threat scenario.

The remainder of the paper is organized as follows. Section 2 presents the basic ideas and concepts behind AMASIVE.

978-1-4673-2340-6/12/$31.00

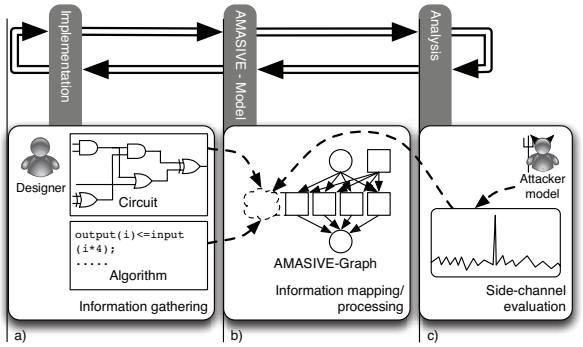

Fig. 1: Architecture of the AMASIVE Framework

In Section 3 we introduce a realization of AMASIVE, which is able to analyze VHDL designs. Section 4 demonstrates the concept of AMASIVE by performing a side-channel evaluation on PRESENT. Finally, Section 5 concludes the paper.

II. Diagnostic Leakage Analysis

AMASIVE identifies side-channel vulnerabilities from three steps as depicted in Figure 1. First, various information is collected from an arbitrary set of sources, such as the implementation of a cryptosystem or its designer (cf. Figure 1 a). Secondly, this information is combined into a graph, which represents both the design, and an attacker model, which specifies the attacker's capabilities (cf. Figure 1 b). Lastly, both the graph and the attacker model are used to identify and to highlight side-channel vulnerabilities directly in the design (cf. Figure 1 c). Note that due to the modular concept of the framework, these steps can be extended by a fourth one, for instance, by a suggestion of adequate countermeasures.

The graph and the attacker model are the basic parts of the security analysis and represent the relation between the device and the attacker in the real world. In the sequel, we first detail the graph and the attacker model and then we specify the general concept of the security analysis.

A. Graph Representation

The AMASIVE graph as depicted in Figure 1 b) models the device in terms of its architectural and algorithmic features and is introduced as a base for both the security analysis and the communication with the designer. We define the graph G as a relation between a set of nodes V, which represent execution modules, and a set of edges E, which connect nodes. Nodes have an input, which specifies the processed values, and an output, which specifies the resulting values. We distinguish three kinds of nodes: *operations*, *entropy sources*, and *registers*. Operations describe modules, which modify the processed data, such as a S-box module or an XOR gate. Entropy sources model secret values, inserted during the execution of the algorithm, i.e. keys or random numbers. Lastly, registers are elements in an architecture that store data. Additionally, the start nodes and the end nodes of the graph are marked in order to identify the input to and the output of an algorithm.

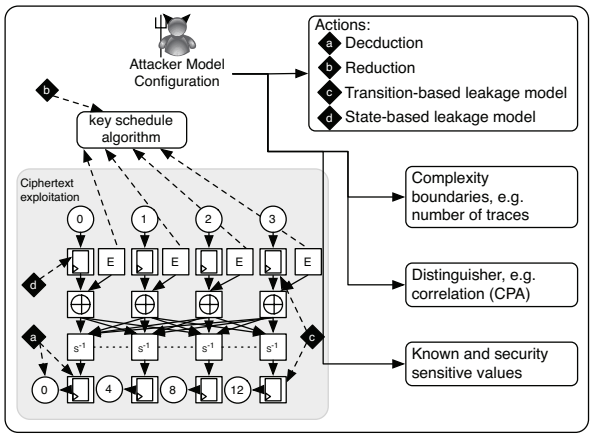

Fig. 2: Scheme of the Attacker Model Configuration

B. Attacker Model

To evaluate the security of a device an attacker model is introduced. This model represents an attacker, who aims at recovering security sensitive information. Various features can be assigned to the attacker model. Figure 2 depicts an example for the attacker model as well as its assignable features.

We assign a local view $\mathbb{V} \subseteq (V \times \mathbb{R})$ to the attacker model that sets all nodes of the graph $G = (V, E)$ in relation with an entropy level, which specifies the uncertainty of the attacker regarding the value of the node. Furthermore, we create a set $\mathbb{S} \subseteq \mathcal{P}(V)$, consisting of sets of security sensitive nodes, which lead to a key recovery when known. With the known and security sensitive nodes marked, the attacker model knows both the starting points and the target points in the graph. However, the attacker model still requires a set of actions $\mathbb{A}$ for traversing the graph. These actions represent the performable attacks of the attacker model. An action $a \in \mathbb{A}$ is defined as a function $a : \mathbb{V} \mapsto \mathbb{V}$. A requirement function $R_G : (\mathbb{V}, \mathbb{A}) \mapsto \{0, 1\}$ checks if the requirements for performing an action $a \in \mathbb{A}$ in the attacker view $\mathbb{V}$ are satisfied. R_G is defined as

$$R_G(\mathbb{V}, a) = \begin{cases} 1 & \text{, if requirements of a hold for } \mathbb{V} \\ 0 & \text{, else.} \end{cases} \quad (1)$$

The strength of the attacker can be increased or decreased by adding or removing known nodes, actions, or by changing the complexity boundaries. Note that the attacker model is independent of the constructed graph and can be varied without requiring changes in the graph.

C. Security Analysis

We express the security analysis as a game where the goal of the attacker is to recover a set of security sensitive nodes from the graph $G = (V, E)$. The attacker is given a set of initially known nodes $\mathfrak{I}$ by adding these nodes with an entropy of zero to his view $\mathbb{V}$:

$$\mathbb{V} = \bigcup_{\forall n \in \mathfrak{I}} \{(n, 0)\}. \quad (2)$$

Algorithm 1 AMASIVE Security Analysis

Require: all nodes $n \in V$ of the AMASIVE graph $G = (V, E)$ are contained in the view $\mathbb{V}$
Require: set of security sensitive nodes $\mathbb{S} \subseteq \mathcal{P}(V)$
Require: ordered list of N actions to traverse the graph $\mathbb{A} = \{a_1^G, a_2^G, ..., a_N^G\}$ with corresponding requirement function $R_G : (\mathbb{V}, \mathbb{A}) \mapsto \{0, 1\}$
Require: computational complexity boundary C

```
i = 1
repeat
    if R(V, a_i^G) = 1 then
        V = a_i^G(V)
        i = 1
    else
        i = i + 1
    end if
until i > N
```

Attacker wins if $\exists\ \{s_1, s_2, ..., s_p\} \in \mathbb{S}$ with $(s_j, \epsilon_j) \in \mathbb{V}$ such that $(\sum_{j=1}^{p} \epsilon_j) < C$

Every node n, contained in the set $(V \backslash \mathfrak{I})$, is added to $\mathbb{V}^+$ together with its respective number of bits n_b as entropy:

$$\mathbb{V}^+ = \mathbb{V} \cup \bigcup_{\forall n \in (V \backslash \mathfrak{I})} \{(n, n_b)\}. \tag{3}$$

Subsequently, the attacker tries to apply actions in order to recover information about further nodes. Algorithm 1 details the security analysis game. The attacker wins the game if the complexity for guessing a group of security sensitive nodes is lower than his pre-defined computational complexity bound. If the attacker wins the game, the designer is notified by highlighting the recovered nodes and by visualizing the corresponding sequence of actions that led to the node's recovery.

III. Diagnostic Framework

In the following we present an implementation of AMASIVE, which analyzes FPGA designs written in VHDL. First, we describe the sources from which we collect the required information for the analysis. Then we outline the construction of the graph. Lastly, we detail the attacker model and the subsequent steps performed in the security analysis.

A. Information Collection

The information we need from the hardware design is the structure of the algorithm and a functional description of the operations. Also, in order to specify the attacker, we need information about the attacker model. In order to obtain this information, we require the following sources: the VHDL source code, its simulation results, and the designer.

The VHDL source code is parsed in order to extract the structure of the algorithm. Furthermore, we define constraints for the VHDL source code, in order to increase the readability of the graph. Such constraints are, for instance, that each operation node in the graph is described in a separate VHDL entity. We also introduce flags aimed for a direct communication of the designer with the VHDL parser. By using such a flag the designer can, e.g., enter names for nodes or define the type of a VHDL entity.

The AMASIVE framework is able to map the complete functionality of the investigated algorithm to the graph model. Therefore, simulation procedures and the tools from the Xilinx, Inc. design suite ISE are utilized to generate a lookup table, which describes the functionality of each entity. After the functionality of the VHDL entity is provided for each node in the graph, the graph is able to mimic the complete implemented algorithm. Thus, the framework is able to simulate the complete algorithm at a functional level even without detailed information about the implementation.

We only query the designer if information can not be obtained autonomously. In the current version, we ask the designer to specify the attacker model by stating the public and security sensitive values, specifying the leakage model, and by defining the security constraints.

After collecting the information we compose it using an XML scheme. In this XML scheme we define four different elements: a register, an operation, an entropy source, and a channel. Each element is attributed by general features such as a name, an identifier, a bit size, and an entropy level.

B. Constructing the Graph

In order to construct the graph, we first need to instantiate all elements of the XML description as node. The instantiation ensures that each implemented module is also modeled in the graph. When each entity is instantiated, loops in the VHDL code are unrolled. From the instantiation of the entities we obtain a node for every execution of a module on the FPGA.

In the next step the nodes are connected using the channel elements. The connected nodes result in a graph that represents the steps in the execution of the algorithm. Subsequently, permutations are dissolved, since on a FPGA they are represented as a permutation of wires. The dissolving is done by using the permutation lookup table, gained from the simulation of the VHDL design, and connecting the outputs of the previous and the inputs of the next elements of the permutation.

The last step in the process of constructing the graph is the assignment of non-trivial features to the nodes. These attributes describe features that have to be computed from a combination of directly readable features. Non-trivial features are for instance the invertibility of functions, the bijectivity of functions, and the vulnerability of a function with respect to side-channel attacks.

C. Graph Analysis

After the construction of the graph, the security analysis is performed. As a first example distinguisher for the security analysis we implemented the CPA. An essential step of the CPA is to make appropriate leakage predictions by formulating hypotheses about exploitable intermediate values or register transitions in the algorithm cf. [6]. The identification of a suitable hypothesis function can currently be performed for

Algorithm 2 Hypothesis function identification for HW model

Require: view $\mathbb{V}$ for the graph $G = (V, E)$
Require: computational complexity boundary C
1: Identify operation node op, where node i and entropy node e are input nodes, o is the output node, $\{(i, \epsilon_i), (e, \epsilon_e), (o, \epsilon_e)\} \subseteq \mathbb{V}$ holds, and $\epsilon_e > 0$
2: **if** $\epsilon_i = 0$ **and** $\epsilon_o > 0$ **then**
3: hypothesis function $\mathfrak{h} = HW(op\ (i, e))$
4: **else if** $\epsilon_i > 0$ **and** $\epsilon_o = 0$ **and** op is invertible **then**
5: hypothesis function $\mathfrak{h} = HW(op^{-1}(o, e))$
6: **else**
7: **return** error: no suitable hypothesis found
8: **end if**
9: **if** $\epsilon_e < C$ **then**
10: **return** hypothesis function $\mathfrak{h}$
11: **else**
12: **return** error: no suitable hypothesis found
13: **end if**

Algorithm 3 Hypothesis function identification for HD model

Require: view $\mathbb{V}$ for the graph $G = (V, E)$
Require: computational complexity boundary C
1: Identify register transition from register node $(p, 0) \in \mathbb{V}$ to register node $(r, \epsilon_r) \in \mathbb{V}$ with entropy source $(e, \epsilon_e) \in \mathbb{V}$ along the path and $\epsilon_r, \epsilon_e > 0$
2: **if** a suitable register transition is identified **then**
3: **for all** edges i from/to r **do**
4: traverse graph along i until node $(n, 0) \in \mathbb{V}$ is reached
5: add path from i to n to hypothesis function $\mathfrak{h}_t$
6: **end for**
7: hypothesis function $\mathfrak{h} = HD(p, \mathfrak{h}_t)$
8: compute total entropy E of $\mathfrak{h}$ by summing up all entropy values for all entropy source nodes in $\mathfrak{h}$
9: **if** $E < C$ **then**
10: **return** $\mathfrak{h}$
11: **end if**
12: **end if**
13: **return** error: no suitable hypothesis found

the Hamming weight (HW) model and for distance based leakage models such as the Hamming distance (HD) model. However, due to the modularized design of AMASIVE, further leakage models (such as the glitch model or bit models) will be integrated soon. Also note that further side-channel attacks, such as the template attack, can be integrated into the analysis.

In order to start the security analysis of the graph, the attacker model has to be defined using the specifications of the information collection phase. The view of the attacker is initialized by assigning all public nodes the entropy zero and all non-public nodes their corresponding bit-size as entropy. The security sensitive nodes assembled into groups and then added to the set of security sensitive nodes. Subsequently, the actions for the attacker model are assigned. Currently, we have defined four actions: *deduction*, *reduction*, leakage exploitation using the Hamming weight model, and leakage exploitation using the Hamming distance model. The framework uses these actions in order to automatically analyze the resulting graph.

The simplest action, the deduction, evaluates an operation or propagates a value and sets the entropy of the corresponding node in the attacker view to zero. The deduction requires either the input of a node or the output to be known and the node to be invertible.

The reduction infers information about a node and reduces it's entropy. A reduction currently requires an associated node to have a reduced complexity or an operation to have certain features such as non-surjectivity.

A power attack based on the Hamming weight model sets the entropy of the corresponding node in the attacker view to zero. The requirement of a power attack based on the Hamming weight model is a suitable hypothesis function. The process of finding such a function is denoted in Algorithm 2.

A power attack based on the Hamming distance model sets the entropy of all nodes, included in the hypothesis function, to zero. However, finding the required hypothesis function for the Hamming distance model is somewhat more complex due to the requirement of a register transition. Algorithm 3 depicts the procedure of finding a hypothesis function for the Hamming distance model.

The security analysis of the graph is then performed as stated in Algorithm 1 using the defined actions. If the attacker wins the security analysis, then the actions and nodes, which led to the recovery of the secret, are displayed. Furthermore, in order to determine the practical feasibility of the attack, a CPA is performed for each identified hypothesis function. Every hypothesis function is output into a C-source-file and executed by a MATLAB script that implements a CPA. The feasibility of each hypothesis function is verified by checking whether the required number of measurements is within the complexity boundary. If the CPA is succeeds within the predefined complexity boundary, the node is deemed recoverable and its entropy is updated according to the results of the CPA. Otherwise, the node is deemed not recoverable, and all consecutive actions have to be re-evaluated.

When all side-channel attacks have been performed and their feasibility has been evaluated, the designer is notified whether the mounted attack was successful and how high the complexity for recovering each node was. The designer can then evaluate for which nodes he should implement countermeasures. Afterwards he can then rerun the security analysis in order to quantify the achieved security improvement.

IV. Evaluation on the Block Cipher PRESENT

In this section we demonstrate an analysis of the block cipher PRESENT [9] by exercising the AMASIVE framework. In order to increase the understandability, we focus on the analysis of the first and the last round. Figure 3 a) depicts the block diagram of a PRESENT implementation, which

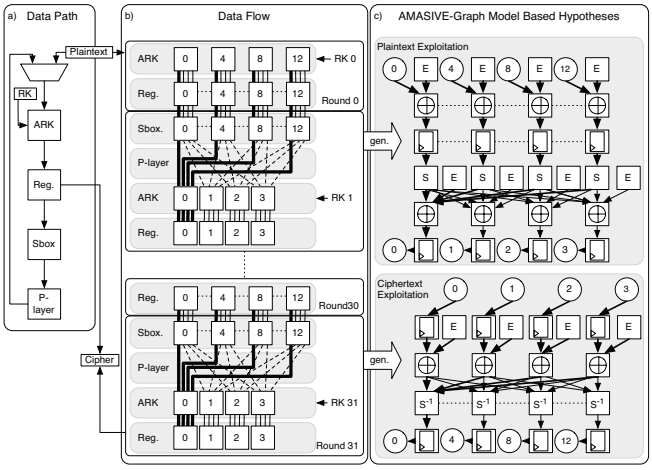

Fig. 3: PRESENT represented as an AMASIVE graph

processes a 64 bit input message and a 80 bit key. In each round this encryption algorithm separates the 64 bit wide input into 4 bit blocks and XORes them with 4 bit blocks of the 64 bit round key. The result of this operation is then stored in a register, from where it is substituted and permuted. In total, PRESENT performs 31 rounds as well as an additional last round, where the last round key is added and the result is stored in a register.

1) Information Collection: The workflow starts with the analysis of PRESENT by parsing its VHDL representation and by producing an XML description, which contains the instantiated modules AddRoundKey, S-box, P-layer, and the channels. We obtain the following elements in the XML description: two operations (AddRoundKey (ARK) and S-box), a permutation (P-layer), a register (Reg), an entropy source (Entropy), and the channels, which connect these elements.

Next, we simulate the design in order to obtain a functional description of the operations. Thus we obtain a 16×16 lookup-table for the S-box and a 256×16 lookup-table for the ARK operation. The permutation P-layer is stored in an 64×2 lookup-table.

Lastly, we define our attacker model. We assume the input and output of the algorithm to be known, since this is a common scenario in side-channel analysis. Subsequently, we mark all Entropy elements as security sensitive and define that the knowledge of all Entropy elements of each round leads to the recovery of the secret key. We allow our attacker to be able to run a brute-force attack up to a complexity of 2^{32} and perform up to 10.000 measurements. Finally, we grant the attacker the ability to execute a deduction, a reduction, and a CPA using the Hamming distance model.

2) Model Building: In order to construct the graph, we instantiate each XML element the number of times it will be executed. The numbers of instantiations of each element for the PRESENT example are depicted in Table I. Subsequently, we connect the channels to the nodes and dissolve the permutations using the P-layer lookup-table. A fragment of the resulting graph is visualized in Figure 3. This figure details the processing of four 4 bit blocks for the first and the last round. Note that the total graph for the first and the last round consists of four times the graph in Figure 3 b).

Next, we evaluate non-trivial features of the operations. The S-box is determined to be bijective and invertible. The ARK operation, which is an XOR between two nodes, is determined to be non-injective and invertible if at least one of both input nodes and the output node are known.

3) Security Analysis: After generating the graph, we can perform the security analysis utilizing the pre-defined attacker model. Figure 3 b) depicts the first and last round of the resulting graph and can be referred to for the visualization of the security analysis. Since we have multiple nodes of one type in each round, we denote the i-th node n of round d as n_i^d. In the following we denote the ARK nodes as x, the S-box nodes as s, the Reg nodes as r, the Entropy nodes as e, the Input nodes as i, and the Output nodes as o. Also, we use their corresponding capital letters, together with a round index, to denote the set of all associated nodes of a round.

We begin the analysis by initializing the view $\mathbb{V}$ of the attacker with the input nodes $Input$ and output nodes $Output$:

$$\mathbb{V} = \bigcup_{\forall n \in (Input \cup Output)} \{(n, 0)\}.$$

The remaining nodes are added to $\mathbb{V}$ with their corresponding bit-size as entropy. Then we add the Entropy nodes to the security sensitive nodes by grouping all nodes $e^d \in V$ of one round into:

$$\mathbb{S} = \bigcup_{0 \leq d < 32} \{\{e_0^d, e_1^d, ..., e_{15}^d\}\}.$$

Finally, we determine and prioritize the available actions. The priority of the actions is: deduction ded_G, reduction red_G, and the CPA using the Hamming distance leakage model HD_G.

After the initialization we start the security analysis of the AMASIVE graph. The first deduction ded_G on $\mathbb{V}$ yields

$$\mathbb{V}^+ = ded_G(\mathbb{V}) = \{(n, \epsilon) \in \mathbb{V} | n \notin R^{31}\} \cup (R^{31} \times 0).$$

In other words, we can simply recover the value of the last round's Reg nodes $R^{31} = \{r_0^{31}, r_1^{31}, ..., r_{15}^{31}\}$ by propagating the value of the output. We set $\mathbb{V} = \mathbb{V}^+$ and since ded_G successfully recovered some nodes, we can again perform ded_G on the updated $\mathbb{V}$, yielding:

$$\mathbb{V}^+ = ded_G(\mathbb{V}) = \{(n, \epsilon) \in \mathbb{V} | n \notin X^{31}\} \cup (X^{31} \times 0).$$

Thus, the second ded_G yields the value of the ARK nodes of the last round $X^{31} = \{x_0^{31}, x_1^{31}, ..., x_{15}^{31}\}$ by simply propagating the value from the recovered Reg nodes R^{31}.

We again set $\mathbb{V} = \mathbb{V}^+$ and observe that we can not apply ded_G to $\mathbb{V}$, since no value can be propagated and no operation evaluated. Thus, we verify the requirements for the reduction

XML elements	ARK	S-box	P-layer	Reg	Entropy	Channel
initially	-	-	-	16	-	16
per round	16	16	1	16	16	64
last round	16	-	-	16	16	48
Total	**512**	**496**	**31**	**528**	**512**	**2048**

TABLE I: Numbers of resulting nodes in the AMASIVE graph

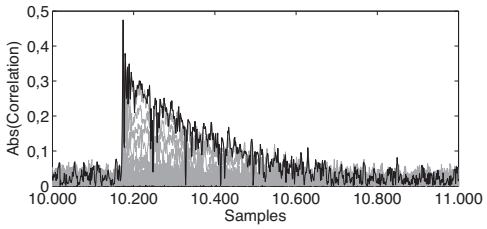

Fig. 4: CPA on PRESENT with hypothesis $\mathfrak{h}_{e_0^{31}}$

and also observe that it can not be applied to $\mathbb{V}$. Thus, we verify the requirement for the CPA using the Hamming distance model HD_G. Indeed, by applying Algorithm 3 to $\mathbb{V}$ we obtain hypotheses for all nodes in E^0 and E^{31}, which are visualized in Figure 3 c). In the following we use the short notation $n_{j,b}$ to refer to the b-th most significant bit of n_j.

The hypothesis function $\mathfrak{h}_{e_0^0}$ for element $e_0^0 \in E^0$ is

$$\begin{aligned} \mathfrak{h}_{e_0^0} &= HD(r_0^0, r_0^1) \\ &= HD(S(i_0 \oplus e_0^0), (x_{0,0}^1 || x_{1,0}^1 || x_{2,0}^1 || x_{3,0}^1)), \end{aligned}$$

where $x_{j,b}^1 = S(i_j \oplus e_j^0)_b \oplus e_{j,b}^1$ for $0 \leq j < 16$. Analyzing the hypothesis function of e_0^0 shows that we also have to make a hypotheses for e_1^0, e_2^0, e_3^0, and e_0^1. This raises the complexity of a CPA to $(2^4)^5 = 2^{20}$, which still lies in the predefined complexity boundary of 2^{32}. Thus, we obtain a successful Hamming distance hypothesis and are able to set the entropy for the nodes e_0^0, e_1^0, e_2^0, e_3^0, and e_0^1 to zero.

The hypothesis function $\mathfrak{h}_{e_0^{31}}$ for $e_0^{31} \in E^{31}$ is:

$$\begin{aligned} \mathfrak{h}_{e_0^{31}} &= HD(r_0^{30}, r_0^{31}) \\ &= HD(S^{-1}(s_{0,0}^{31} || s_{4,0}^{31} || s_{8,0}^{31} || s_{12,0}^{31}), r_0^{31})), \end{aligned}$$

where $s_{j,b}^{31} = o_{j,b} \oplus e_{j,b}^{31}$. In contrast to the hypothesis $\mathfrak{h}_{e_0^0}$, $\mathfrak{h}_{e_0^{31}}$ only requires a complexity of 4 bit, making an attack more likely to succeed with a smaller number of measurements.

Since we found a hypothesis for HD_G, we can update $\mathbb{V}$:

$$\begin{aligned} \mathbb{E} &= E^0 \cup E^1 \cup E^{31} \\ \mathbb{V}^+ &= HD_G(\mathbb{V}) = \{(n, \epsilon) \in \mathbb{V} | n \notin \mathbb{E}\} \cup (\mathbb{E} \times 0). \end{aligned}$$

In theory, the next step would be to use ded_G again on the updated $\mathbb{V}$, which would result in the recovery of X^0 and X^{31}, but we will stop the analysis here, since we have collected sufficient information. The current set of $\mathbb{V}$ contains the round keys of the first, second, and last round. Due to the key-schedule of PRESENT, using one round key we can already recover the actual key by guessing the 2^{16} remaining bits.

However, in order to verify the practicability of the attack, we mounted a CPA using the hypotheses $\mathfrak{h}_{e_i^{31}}$ on a SASEBO-II FPGA, which executed the described PRESENT implementation. We performed 10.000 measurements and used the Pearson correlation coefficient as comparison method between the hypotheses and power traces. The resulting correlation of the attack on the last round of PRESENT using $\mathfrak{h}_{e_0^{31}}$ is depicted in Figure 4 and confirms the correctness of the hypothesis.

V. Conclusion

In this paper we presented an adaptable framework for side-channel security analysis, called AMASIVE. This framework represents the implemented algorithm during its workflow as a dedicated graph and exploits a sophisticated and adaptable attacker model in order to determine side-channel vulnerabilities in the implementation.

We first introduced the general concept of our framework. Subsequently, we discussed a realization, which analyzes VHDL-based hardware implementations. Finally, we demonstrated the general concept of our framework by exercising it on the block cipher PRESENT and identified hypotheses functions for a CPA, which we verified by performing a side-channel attack.

In future work we will extend the security analysis by a fourth step: the suggestion of countermeasures. We will add new models to the security analysis, such as the glitch and the bit model. Also, we will considerably extend the capabilities of the attacker by introducing new attack and evaluation methods, such as the template attack and the mutual information metric. Moreover, we will apply the framework to further algorithms to analyze its performance.

Acknowledgements

The work presented in this contribution was supported by the German Federal Ministry of Education and Research (BMBF) in the project *RESIST* through grant number 01IS10027A. We would like to thank André Seffrin for providing us with his VHDL Parser.

References

[1] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *DAC*, L. Stok, N. D. Dutt, and S. Hassoun, Eds. ACM, 2011, pp. 230–235.

[2] F.-X. Standaert, T. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 5479. Springer, 2009, pp. 443–461.

[3] F. Regazzoni, A. Cevrero, F.-X. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne, "A design flow and evaluation framework for dpa-resistant instruction set extensions," in *CHES*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds., vol. 5747. Springer, 2009, pp. 205–219.

[4] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Automatic insertion of dpa countermeasures," *IACR Cryptology ePrint Archive*, vol. 2011, p. 412, 2011.

[5] E. Oswald, L. Mather, and C. Whitnall, "Choosing distinguishers for differential power analysis attacks," in *Non-Invasive Attack Testing Workshop*, September 2011.

[6] M. A. Elaabid and S. Guilley, "Practical improvements of profiled side-channel attacks on a hardware crypto-accelerator," in *AFRICACRYPT*, ser. Lecture Notes in Computer Science, D. J. Bernstein and T. Lange, Eds., vol. 6055. Springer, 2010, pp. 243–260.

[7] W. He, E. de la Torre, and T. Riesgo, "A precharge-absorbed dpl logic for reducing early propagation effects on fpga implementations," in *ReConFig*, P. M. Athanas, J. Becker, and R. Cumplido, Eds. IEEE Computer Society, 2011, pp. 217–222.

[8] T. Popp and S. Mangard, "Implementation aspects of the dpa-resistant logic style mdpl," in *ISCAS*. IEEE, 2006.

[9] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *CHES*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds., vol. 4727. Springer, 2007, pp. 450–466.