A Methodology for the Characterisation of Leakages in Combinatorial Logic

Guido Bertoni[1] and Marco Martinoli[2]

[1] STMicroelectronics, Agrate Brianza (MB), Italy
guido.bertoni@st.com
[2] University of Bristol, Department of Computer Science, Bristol, UK
marco.martinoli@bristol.ac.uk

Abstract. Glitches represent a great danger for hardware implementations of cryptographic schemes. Their intrinsic random nature makes them difficult to tackle and their occurrence threatens side-channel protections. Although countermeasures aiming at structurally solving the problem already exist, they usually require some effort to be applied or introduce non-negligible overhead in the design. Our work addresses the gap between such countermeasures and the naïve implementation of schemes being vulnerable in the presence of glitches. Our contribution is twofold: (1) we expand the mathematical framework proposed by Brzozowski and Ésik [5] by meaningfully adding the notion of information leakage, (2) thanks to which we define a formal methodology for the analysis of vulnerabilities in combinatorial circuits when glitches are taken into account.

Keywords: side-channel analysis · hardware countermeasures · glitches · formal method

1 Introduction

Side-channel attacks were first introduced by Kocher *et al.* [6] as a way to attack implementations of cryptosystems. They exploit the relation between data being processed and several physical emanations, for instance time taken or power consumed to perform computations [7]. Since its first appearance, side-channel analysis has grown quickly with newly developed attacks as well as countermeasures, which try to prevent any sensitive information from being leaked. For instance, sharing schemes randomise intermediate values in such a way that the leaked information no longer depends on any sensitive data [8]. However the efficiency of countermeasures is deeply linked to physical characteristics of the device on which they are implemented: in 2005 Mangard *et al.* [9] predicted the criticality of glitches for hardware implementations, which was then demonstrated in the same year [10]. They showed how the propagation of signals in combinatorial logic implementing an apparently secured SBox might result in critical leakages, leading to an ineffective protection. To solve the problem, Nikova *et al.* [13,14]

suggested the use of threshold implementations, which allow to tackle glitches at root by developing maps that do not handle all the shares in the same combinatorial circuit. Such maps obviously come at the cost of a significant overhead compared to the unprotected version. Implementations and practical discussions can be found in the work of Moradi *et al.* [12] and of Bilgin *et al.* [4]. As for higher-order security, the issue of glitches has been faced with a generalisation of threshold implementations [3,17], and independently by Prouff and Roche [15]. Specifically on the effects of glitches on the AES SBox Mangard and Schramm [11] have reported a deep and complete analysis.

Overall there is a gap in the capabilities of quantifying the criticality of glitches in a hardware implementation. This gap is not trivial to close, as glitches in combinatorial logic are functions of the final layout of the circuit and the environmental conditions, and might change during the life of the device. In practice two equal devices might exhibit a different behaviour in terms of glitches.

Our aim is to provide a framework for evaluating the presence of glitches under worst-case conditions without the need of detailed characterisation of the combinatorial logic, *i.e.* remaining at gate level description. In order to achieve this result, we start from the mathematical structure created by Brzozowski and Ésik [5], which simulates the propagation of electric signals inside a circuit, and we build a method to relate a modelled power consumption with the sensitive variables that have caused it. Our analysis is done in a worst-case scenario where all possible glitches are taken into account as to achieve the maximum possible generality. Our main result is an assessing tool which is able to formally describe what kind of information could be leaked, and to give an heuristic estimate about the security of sharing schemes implemented in hardware.

Organisation of the paper. Section 2 provides the abstract framework underlying our tool, with a particular emphasis on how circuits and signals propagating inside them are modelled. Section 3 describes parts of the work of Brzozowski and Ésik [5] which are also used by our construction. In Sec. 4 we present our main contribution: we expand the functionalities of the previously discussed mathematical model with the notion of leakage and we show how such an improved framework can be used to analyse cryptographic circuits. In Sec. 5 some examples of usage of our model are reported, with particular reference to the sponge function Keccak. We discuss the soundness of our approach and several practical aspects in Sec. 6, and we conclude our work in Sec. 7.

2 Preliminaries

Our work targets hardware implementations of cryptographic schemes. Since the meaning of such can be quite broad, the present section aims at specifying our environment, as well as at setting the notation we adopt. In fact, our mathematical model applies only to an abstraction of real-world circuits: we just refer to logic netlists, hence circuits formed only of logic gates and connections among them. Our tool therefore achieves a good level of generality, since it does not

require any knowledge of implementation details apart from the circuit scheme itself, which means that it is general enough to include all the above mentioned source of glitches (final layout, environmental conditions...). In particular we focus on asynchronous feedback-free circuits. We claim this is not too restrictive, because of the following argument. Circuits can be divided into two parts: the combinatorial logic and the state storing part. The combinatorial logic is indeed asynchronous, it is the part in charge of implementing the logic functionality and where glitches might propagate. The state storing part, implemented via registers or memory cells, is clocked and provides the synchronization between different sections of the circuits. Since we apply our model to logic circuits performing sensitive computations, the most natural choice is to focus on the asynchronous part only. We do not consider the presence of feedbacks in the combinatorial part for the sake of simplicity and because they are not a common construction in this field anyway.

We adopt a high-level abstraction of signals too. Since we are only interested in the Boolean value they represent, it is convenient to think of them as square waveforms which can assume the values 0 or 1. To push the abstraction further we assume the existence of a sampling frequency being fine enough to detect all the changes occurring in a signal, in such a way that a signal is represented as a finite-length bit-string. We denote by

$$\mathbb{Z}_2^* = \bigcup_{r=1}^{\infty} \mathbb{Z}_2^r$$

the set of finite-length bit-strings. We denote bit-strings by concatenation of bits, *i.e.* a bit-string $s \in \mathbb{Z}_2^*$ is of the form $s = a_1 \dots a_r$ for some bits $a_i \in \mathbb{Z}_2$ and for a certain $r \geq 1$. Grouping bits of a bit-string is also a useful notation and is denoted by $s = b_1^{p_1} \dots b_{r'}^{p_{r'}}$, where this time we assume $b_j \neq b_{j+1}$ and $p_i > 0$ for every $1 \leq i \leq r'$, $1 \leq j \leq r' - 1$. Essentially, the latter notation highlights the number of times p_i the corresponding bit b_i is repeated in a bit-string. Finally, we denote by $\ell(s) = r$ its length, by $\alpha(s) = a_1 = b_1$ its first bit and by $\omega(s) = a_r = b_{r'}$ its last one.

Further notation. We denote the power set (*i.e.* the set of all subsets) of a set S by $\mathcal{P}(S)$. Vectors are denoted by underlined letters while boldface is reserved for signals seen as transients (cf. Definition 1 and Example 1).

2.1 Power consumption model

The power consumed by a circuit is a crucial information as it is one of a number of side-channels through which a loss of sensitive information can occur [7] and is the only one we focus on throughout this work. If we consider global synchronous circuits, the power consumption can be divided in three components: the static leakage, the switching of registers and the switching of combinatorial logic. The static leakage is the amount of power needed by the circuit to maintain the current state when no switch is present. The switching of registers is the

consumption taken by the circuit for updating the state and is easily approximated by the Hamming distance of the state in two consecutive clock cycles. The value of the registers can be easily protected by masking schemes. The last contribution is the most interesting for us and is related to the consumption of the combinatorial logic. From a temporal point of view, the switching of registers usually happens at the rising edge of the clock cycle while the static leakage happens in its last part. By contrast, the consumption of combinatorial logic spans, in most cases, the entire duration of the clock cycle [16]. Consistently with the choice of addressing only the asynchronous part of a circuit, our power consumption model includes only the contribution of the combinatorial logic.

Since we deal only with circuits formed by logic gates, we assume that the power consumed by the whole circuit is the sum of the power consumed by each gate, which reduces the problem to modelling the power consumed by a gate. The simplest way is to consider the signal a gate outputs or, equivalently, the corresponding bit-string. If the output signal changes, equivalently the corresponding output bit-string switches, the gate consumes. The power consumption model we assume in the present work is then described by the following three assumptions:

1. a gate consumes power if and only if its output bit-string switches;
2. a zero-to-one switch consumes the same amount of power as a one-to-zero switch;
3. every time some power is consumed, an attacker can measure and exploit it. Hence we assume that a potential leakage exists as far as a switch occurs.

As already stated, we neglect static leakage by means of the first assumption. The second assumption is made for the sake of simplicity and it can be dropped in favour of a more realistic model built on top of a specific technology library. The third assumption assures the best possible generality: we consider as leaked every variable that has a chance to be leaked.

3 Simulation of signal propagation

Brzozowski and Ésik [5] have developed a mathematical structure which aims at simulating worst-case glitches propagation in a circuit. In essence, the model analyses how a change in the inputs propagates and which kind of response is triggered in the gates. In their work, Brzozowski and Ésik use a higher level of abstraction than bit-strings to simulate signals.

Definition 1. *A* transient *is a bit-string with no repetitions. More formally, a bit-string $t = a_1 \dots a_r \in \mathbb{Z}_2^*$ is a transient if $a_i \neq a_{i+1}$ for all $1 \leq i \leq r-1$. Equivalently, $t = b_1^{p_1} \dots b_{r'}^{p_{r'}} \in \mathbb{Z}_2^*$ is a transient if $p_i = 1$ for every $1 \leq i \leq r'$.*

Informally, transients can only be of the form $1010\dots$ or $0101\dots$ for an arbitrary finite length $r \geq 1$ (note that bits 0 and 1 can be considered as transients for which $r = 1$). We define a map from bit-strings to transients called the *contraction map* and denoted by $\gamma : \mathbb{Z}_2^* \to \mathbb{Z}_2^*$ such that:

$$\gamma(b_1^{p_1} \dots b_n^{p_n}) = b_1 \dots b_n.$$

We denote by $T = \gamma(\mathbb{Z}_2^*) \subseteq \mathbb{Z}_2^*$ the set of all possible transients.

Definition 2. *Let $t, t' \in T$ be two transients such that $\ell(t) \leq \ell(t')$. We say that t is a* prefix *of order $o = \ell(t') - \ell(t)$ of t' if $\alpha(t) = \alpha(t')$. We adopt the notation $t \preceq_o t'$. Note that if $o = 0$, $\preceq_0$ is equivalent to equality between transients.*

The rationale behind transients is the following. Contracting bit-strings is equivalent to neglecting time periods during which a signal assumes constant values 1 or 0. This results in transients being exclusively designed to represent which changes occur, but not when: the order of switches can then be freely tuned, in such a way that the worst glitchy behaviour is always shown at the output of a gate. That is to say if two transients modelling two changing signals are given as inputs to a gate, then the output will be a transient modelling the signal showing the highest possible number of changes. Next subsection specifies how to combine transients so to emulate gates' logic.

3.1 Operations among transients

As the previous discussion has suggested, the choice of transients rather than general bit-strings as a formalisation of signals relies on the operations that it is possible to define among them. Since the circuits we study are only formed of logic gates, we want those operations to preserve gates' functionalities. Therefore we aim at building a function $\hat{f} : T^n \to T$ associated to a Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ whose inputs are n transients, namely $\underline{t} = (t_1, \ldots, t_n) \in T^n$.

Example 1. Let us suppose that two signals s_1 and s_2 are given as input to a gate implementing a Boolean function $f : \mathbb{Z}_2^2 \to \mathbb{Z}_2$. Firstly, they are fixed at constant values $b_1 \in \mathbb{Z}_2$ and $b_2 \in \mathbb{Z}_2$ respectively. Suddenly, s_1 changes from b_1 to $c \in \mathbb{Z}_2$, with $c \neq b_1$. This is represented by the transient $\mathbf{s}_1 = b_1 c$ which can be either 01 or 10. Then, the idea behind the function $\hat{f}$ is to emulate the behaviour of the function f, but taking as inputs the two transients $\mathbf{s}_1 = b_1 c$ and $\mathbf{s}_2 = b_2$ (seen as a length-one transient) and producing a transient with the highest number of switches, *i.e.* as if the highest number of glitches occurred. Note that we write a variable in boldface if it is seen as a transient and that bit concatenation is denoted by simply writing one bit after the other.

The remainder of this subsection describes how to achieve the functionality discussed in Example 1. The idea is that, given two input transients $t_1 = a_1 \ldots a_n$ and $t_2 = b_1 \ldots b_m$, the first bit the gate computes is $f(a_1, b_1)$. This will be also called the initial stable state. Then the two inputs change to a_2 and b_2 respectively, and we have the freedom to decide which is the first one to affect the gate such that another change in the output (if any) is triggered. This process is built thanks to two graphs which look at all possible combinations of propagation times. Firstly, we define the directed graph $D(\underline{t})$ as follows.

Definition 3. *Given* $\underline{t} = (t_1, \dots, t_n) \in T^n$*, we define the directed graph* $D(\underline{t}) = (V, E)$ *such that:*

$$V = \{\underline{v} \in T^n \mid v_i \preceq_{o_i} t_i \text{ for every } 0 \le o_i < \ell(t_i) \text{ and } 1 \le i \le n\}$$
$$E = \{(\underline{v}, \underline{w}) \in V \times V \mid \exists! i \text{ such that } v_i \preceq_1 w_i \text{ and } v_j = w_j \text{ for every } j \neq i\}$$

Note that $D(\underline{t})$ is the graph whose nodes are all the prefixes of the components of $\underline{t}$, the simplest being $(\alpha(t_1), \dots, \alpha(t_n))$ and the longest being $\underline{t}$ itself. Edges are drawn if there exists only one change in exactly one of two nodes' components. At this point we label each vertex $\underline{v} = (v_1, \dots, v_n) \in V$ with the bit $f_\omega(\underline{v}) = f(\omega(v_1), \dots, \omega(v_n)) \in \mathbb{Z}_2$ and we construct the following graph.

Definition 4. *Let* $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ *be a Boolean function. Given* $\underline{t} = (t_1, \dots, t_n) \in T^n$ *and* $D(\underline{t}) = (V, E)$*, we define its labelled directed graph* $D_f(\underline{t}) = (V_f, E_f)$ *such that:*

$$V_f = \{f_\omega(\underline{v}) \in \mathbb{Z}_2 \mid \underline{v} \in V\}$$
$$E_f = \{(f_\omega(\underline{v}), f_\omega(\underline{w})) \in V_f \times V_f \mid (\underline{v}, \underline{w}) \in E\}$$

It is straightforward that the graph $D_f(\underline{t})$ has the same shape as $D(\underline{t})$. In particular there is a bijection between E and E_f, hence every path in $D(\underline{t})$ can be reconstructed in $D_f(\underline{t})$. Thanks to this, the output of $\hat{f}$ is computed by first considering all the paths in $D(\underline{t})$ from $(\alpha(t_1), \dots, \alpha(t_n))$ to $(t_1, \dots, t_n)$ and then reconstructing them in $D_f(\underline{t})$. Since elements of V_f are bits, each path in the latter graph uniquely defines a bit-string by concatenating its successive vertices. The contraction map γ is then applied to every such bit-strings and the output of $\hat{f}(t_1, \dots, t_n)$ is defined as the longest contraction.

Theorem 1. *Let* $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ *be a Boolean function. The function* $\hat{f} : T^n \to T$ *is well defined for any given input* $\underline{t} = (t_1, \dots, t_n) \in T^n$.

Proof. We only need to prove that if two paths in $D(\underline{t})$ lead to bit-strings whose contractions have the same length and are the longest, then such contractions are equal. In other words, let $s_1 = a_1 \dots a_k$ and $s_2 = b_1 \dots b_m$ be two bit-strings computed from the two paths such that $\ell(\gamma(s_1)) = \ell(\gamma(s_2))$, where $a_i, b_j \in \mathbb{Z}_2$. Since we are only considering paths in $D(\underline{t})$ from $(\alpha(t_1), \dots, \alpha(t_n))$ to $(t_1, \dots, t_n)$, it is true that $\alpha(s_1) = \alpha(s_2) = f(\alpha(t_1), \dots, \alpha(t_n))$ and $\omega(s_1) = \omega(s_2) = f(\omega(t_1), \dots, \omega(t_n))$. It follows that $\alpha(\gamma(s_1)) = \alpha(\gamma(s_2))$ and $\omega(\gamma(s_1)) = \omega(\gamma(s_2))$, because the contraction map cannot change the first and last bits. Since $\ell(\gamma(s_1)) = \ell(\gamma(s_2))$ holds too, $\gamma(s_1)$ and $\gamma(s_2)$ are two transients with same first and last bits and same length, hence $\gamma(s_1) = \gamma(s_2)$. □

Example 2. We report how to construct $\widehat{\texttt{AND}} : T^2 \to T$ between the two transients $010, 01 \in T$.

In Fig. 1 (left), $D(010, 01)$ is built according to Definition 3 while in Fig. 1 (right) $D_{\texttt{AND}}(010, 01)$ is computed with the function $\texttt{AND} : \mathbb{Z}_2^2 \to \mathbb{Z}_2$ as in Definition 4. In the graph $D(010, 01)$ there are only three possible paths from

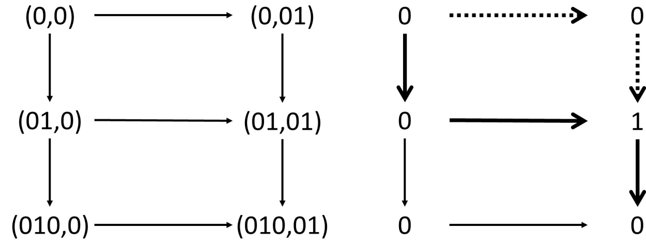

Fig. 1. $D(010, 01)$ and $D_{\texttt{AND}}(010, 01)$

$(0, 0)$ to $(010, 01)$, whose corresponding bit-strings are 0000, 0010 and 0010. By applying the contraction map to each of them, we obtain that the possible outputs of $\widehat{\texttt{AND}}$ are $\gamma(0000) = 0$, $\gamma(0010) = 010$ and $\gamma(0010) = 010$. Hence, by taking the longest possible transient we obtain that $\widehat{\texttt{AND}}(010, 01) = 010$. In Fig. 1 (right), the chosen path is highlighted with thicker arrows and another possible one leading to the same result is highlighted with thicker dash arrows.

We want to highlight the rationale behind those graphs. Each edge corresponds to a change in exactly one of the inputs. Deciding a path in those graphs is then equivalent to assuming an "order of arrival" of the inputs' changes to the gate. Such an order is chosen according to our previously discussed "longest possible output" rule.

Remark 1. The above construction is only a formal procedure to build $\hat{f}$ from a generic Boolean function f. In practice, once f is fixed, a simple rule to compute $\hat{f}$ can be derived from the graph. For instance, it is possible to prove (as it is done in [5]) that $\widehat{\texttt{AND}} : T^2 \to T$ can be defined for any two transients $t, t' \in T$ as follows:

- $\widehat{\texttt{AND}}(t, 1) = \widehat{\texttt{AND}}(1, t) = t$;
- $\widehat{\texttt{AND}}(t, 0) = \widehat{\texttt{AND}}(0, t) = 0$;
- if $\ell(t), \ell(t') > 1$, $\widehat{\texttt{AND}}(t, t')$ is the transient w such that:
  - $\alpha(w) = \alpha(t) \wedge \alpha(t')$;
  - $\omega(w) = \omega(t) \wedge \omega(t')$;
  - $u(w) = u(t) + u(t') - 1$;

where $u : T \to \mathbb{N}$ denotes the number of ones of a transient. Such a simplification also has an impact on the performance of $\hat{f}$: since we no longer need any graphs, $\hat{f}$ can be considered linear in the number of inputs (their lengths do not matter). We refer to the work of Brzozowski and Ésik [5] for more examples.

3.2 Glitch-counting algorithm

We are finally ready to state the glitch-counting algorithm, which simulates the propagation of signals inside a circuit in terms of transients. First of all, a change

in one or more inputs is assumed and represented as a transient. The glitch-counting algorithm assigns a transient to each gate as soon as the change reaches it. If the gate implements a Boolean function f, then the result is computed according to $\hat{f}$.

Given a circuit with m inputs and k gates, we denote by $\underline{X} = (X_1, \ldots, X_m)$ the vector of input variables and by $\underline{s} = (s_1, \ldots, s_k)$ the vector of state variables, which are the gates' outputs. We use boldface to distinguish when variables are used as transients, as in Example 1.

Definition 5. *We call* excitation *the Boolean function $S_j : \mathbb{Z}_2^m \times \mathbb{Z}_2^{k-1} \to \mathbb{Z}_2$ by which the state variable s_j is computed.*

$$s_j = S_j(\underline{X}, \underline{s}) = S_j(X_1, \ldots, X_m, s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_k)$$

The above definition simply establishes a notation for the Boolean function each gate implements. It can be further extended so as to take into account all the excitations in a given circuit.

Definition 6. *Given a circuit with m inputs and k gates, the function $S : \mathbb{Z}_2^m \times \mathbb{Z}_2^k \to \mathbb{Z}_2^k$ defined by $S(\underline{X}, \underline{s}) = (S_1(\underline{X}, \underline{s}), \ldots, S_k(\underline{X}, \underline{s}))$ is called the* vector of excitations of the circuit*.*

Note that in Definition 6, the j$^{\text{th}}$ component of the vector $\underline{s}$ is dropped when given as input to each S_j for every $1 \leq j \leq k$, according to Definition 5. This is because we only deal with feedback-free circuits.

Example 3. Let us consider the circuit in Fig. 2. It has input vector $\underline{X} = (X_1, X_2, X_3)$, state vector $\underline{s} = (s_1, s_2, s_3)$ and excitation functions given by the following Boolean expressions.

$$\begin{aligned} s_1 =& S_1(X_1, X_2, X_3, s_2, s_3) = X_1 \wedge X_2 \\ s_2 =& S_2(X_1, X_2, X_3, s_1, s_3) = X_2 \vee X_3 \\ s_3 =& S_3(X_1, X_2, X_3, s_1, s_2) = s_1 \oplus s_2 \end{aligned}$$

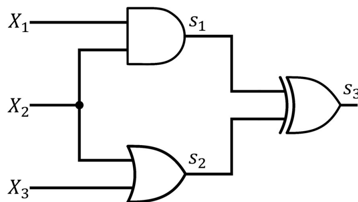

Fig. 2. Example of a circuit with input and state variables

Initially, suppose that the input $\underline{X}$ assumes the value $\underline{X} = \underline{a}' = (a'_1, \ldots, a'_m) \in \mathbb{Z}_2^m$, and that the state has the value $\underline{s} = \underline{b} = (b_1, \ldots, b_k) \in \mathbb{Z}_2^k$. We assume that the circuit is stable, *i.e.* $S(\underline{a}', \underline{b}) = \underline{b}$, and that the input changes to $\underline{a} = (a_1, \ldots, a_m) \in \mathbb{Z}_2^m$. We call this a *transition* and we denote it by $a'_1 \ldots a'_m \to a_1 \ldots a_m$.

Definition 7. *The* transition function $\circ : \mathbb{Z}_2 \times \mathbb{Z}_2 \to T$*, given* $a, b \in \mathbb{Z}_2$*, returns:*

$$a \circ b = \begin{cases} a & \textit{if } a = b \\ ab & \textit{if } a \neq b \end{cases}$$

where ab *denotes the concatenation of* a *and* b*, which is a transient. This notation is extended to vectors. If* $\underline{a}' = (a'_1, \ldots, a'_m)$ *and* $\underline{a} = (a_1, \ldots, a_m)$*, then:*

$$\underline{a}' \circ \underline{a} = (a'_1 \circ a_1, \ldots, a'_m \circ a_m)$$

The glitch-counting algorithm starts with the circuit in the initial stable state $(\underline{a}', \underline{b})$. The input is then set to $\underline{\mathbf{a}} = \underline{a}' \circ \underline{a}$ and is kept constant at that value for the duration of the algorithm. After the input changes, some state variables become unstable in the sense that they no longer represent the correct logic output of their gate. We set all unstable variables at the same time to their excitations as soon as the input change propagates till their gate. We then obtain a new internal state, which is a vector of transients, and the process is repeated until all the state variables become stable again, *i.e.* their value is the correct Boolean output of their gate. Formally, the glitch-counting algorithm is specified below.

Algorithm 1 Glitch-counting algorithm

Input: The initial stable state $(\underline{a}', \underline{b})$, the new input $\underline{a}$ and the vector of excitations among transients $\hat{S}(\underline{\mathbf{X}}, \underline{\mathbf{s}})$ of a circuit.
Output: A list of k transients, one per each gate's output, describing the worst possible switching activity during the transition $\underline{\mathbf{a}} = \underline{a}' \circ \underline{a}$.

1: $h \leftarrow 0$;
2: $\underline{\mathbf{a}} \leftarrow \underline{a}' \circ \underline{a}$;
3: $\underline{\mathbf{s}}^0 \leftarrow \underline{b}$;
4: **repeat**
5:     $h \leftarrow h + 1$;
6:     $\underline{\mathbf{s}}^h \leftarrow \hat{S}(\underline{\mathbf{a}}, \underline{\mathbf{s}}^{h-1})$;
7: **until** $\underline{\mathbf{s}}^h = \underline{\mathbf{s}}^{h-1}$;
8: **return** $\underline{\mathbf{s}}^h$;

Example 4. Suppose that, in the situation of Example 3, the input changes from $\underline{a}' = (1, 0, 0)$ to $\underline{a} = (0, 1, 0)$, hence the transition $100 \to 010$ occurs. The execution of the algorithm is summarised in Table 1, where each row represents one iteration of the cycle and each column refers to one variable (both input and state) of the circuit. The last two rows are identical, meaning that we have reached a stable state again and the algorithm terminates. At each step, the algorithm computes the whole vector of excitations of the circuit, hence considering

all gates. However, it follows the behaviour of real-world signal propagation, hence earlier gates (*i.e.* closer to circuit inputs) are affected first. Indeed the first row just represents the initial state (when only inputs have changed), the second one depicts a change in the first line of gates while in the third row signals propagate till the last `XOR`. Figure 3 is a graphical representation of the final situation, which is the output of the algorithm without intermediate steps. Note that the final logic situation can be retrieved from Table 1 by computing $\omega(\mathbf{s}_1)$, $\omega(\mathbf{s}_2)$ and $\omega(\mathbf{s}_3)$ which are the correct (*i.e.* stable) Boolean outputs for inputs $\omega(\mathbf{X}_1)$, $\omega(\mathbf{X}_2)$ and $\omega(\mathbf{X}_3)$.

Table 1. Example of a glitch-counting algorithm's execution

h	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$
0	10	01	0	0	0	0
1	10	01	0	010	01	0
2	10	01	0	010	01	0101
3	10	01	0	010	01	0101

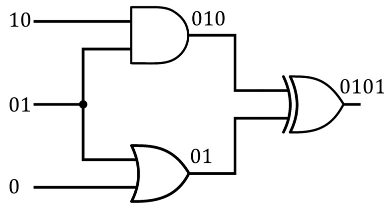

Fig. 3. Example of a glitch-counting algorithm's execution

We conclude the present section with a theorem stating the asymptotic running-time of the glitch-counting algorithm. The proof is extensively discussed by Brzozowski and Ésik [5] and is then omitted here.

Theorem 2 (Section 8 of [5]). *Given a feedback-free circuit and a transition of its inputs, the glitch-counting algorithm always terminates. Moreover, it runs in $O(m + k^2)$ time where m is the number of inputs and k the number of gates.*

4 LP model

The glitch-counting algorithm was developed in the first place to prevent unnecessary power consumption by discarding netlists being particularly exposed

to glitches propagation [5]. Our main contribution is the *LP (Leakage Path) model*, which is a mathematical abstraction that expands the functionalities of the glitch-counting algorithm and relates its simulations to the notion of leakage. Our result leads to a tool that allows to evaluate if a circuit has a critical leakage from the security point of view. The remaining of this section explains the structure of the LP model, which is formed of the following mathematical entities:

input variables are the only part of a circuit that can trigger a signal propagation. If no input variable changes, no signal propagates and no power is consumed, therefore no leakage exists according to our power model;
literal transients are sets of input variables. For each gate reached by a signal's change, a literal transient contains which variables have caused the change and could then be leaked;
literifiers are the link between transients and leakage. Essentially, they relate the input and output transients of a gate to the appropriate literal transient.

The general idea behind the above three objects is the following. The process begins with a change in the input variables, which generates a signal propagation inside the circuit and affects some gates. The gates are then supposed to produce a new output based on the new inputs and their final result depends on which variables have changed and how. In this framework, literifiers are responsible to retrieve the variables involved and represent them via literal transients.

4.1 Structure of LP model

We now describe in details each part of the LP model with respect to a single gate. This means that when we talk of input variables, we mean the variables that are directly given as inputs to it. Next subsection will proved a broader view, showing how to apply notions for single gates to a whole circuit. Following the same notation as the input variables of a circuit, we denote such variables by X_j and by $\mathbf{X}_j$ if they are seen as transients; we assume that $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ is the Boolean function implemented by the gate and we denote by $\hat{f} : T^n \to T$ the corresponding function among transients.

As stated in the introduction of this section, input variables are of great importance for both the glitch-counting algorithm, since nothing could be simulated without a change of theirs, and the LP model. In essence, they are the objects our study targets as we aim at following their propagation along the circuit.

Definition 8. *Given a gate with n inputs, namely $X_1, \ldots, X_n$, we call* literal transient *any subset of $\{1, \ldots, n\}$. The set of literal transients is denoted by $I = \mathcal{P}(\{1, \ldots, n\})$.*

Literal transients are a generalisation of transients: instead of being finite alternated sequences of zeros and ones, they are finite sets of input variables. In a sense, they are the result we are looking for: the analysis of a circuit by means

of the LP model consists in assigning a literal transient to each gate. Their utility stems from the fact that they list which input variables are responsible for the power consumption and could then be leaked according to our power model. This is strictly connected with the rationale behind transients. In both cases we assume the worst possible scenario: transients are supposed to switch as if the worst possible combination of glitches occurred in the same way as literal transients list all variables being leaked in the worst possible case. It is clear from the above discussion that the core of the LP model is the way we assign literal transients to gates.

Literifiers are functions establishing which input variables are leaked by a gate, *i.e.* the ones having caused a change in its output. They depend on how the gate's inputs change, *i.e.* which transients enter in it, and on the implemented logic.

First of all, we represent the input of a gate as the following vector of couples:

$$((t_1, l_1), \ldots, (t_n, l_n)) \in (T \times I)^n.$$

We call it *transient-variable representation*: the first component of each couple is a transient modelling how that input signal changes, while the second one is a literal transient listing the input variables responsible for that change.

Example 5. Recalling Figs. 1 and 3, the gate computing $\mathbf{s}_1 = 010$ has the following input according to the transient-variable representation.

$$((10, \{1\}), (01, \{2\}))$$

In Example 5 we have assumed that the literal transient of a circuit's input is just the singleton containing its index. As for now, the transient-variable representation is directly possible only for gates at height 1, *i.e.* whose inputs are inputs of the circuit itself. In that case each literal transient is simply the singleton of a variable. In the next subsection we will show a procedure similar to the glitch-counting algorithm to meaningfully apply literifiers also to gates whose inputs have already been processed. Such gates are said to have height grater than 1. Informally speaking, the height of a gate is inductively defined to be 1 if all its inputs are circuit inputs, and to be the maximum height of its inputs plus one otherwise. We intentionally omit any further formalisation to avoid heavy notations. As an example, in the circuit in Fig. 2 the `AND` and `OR` gates are at height 1 and the `XOR` is at height 2.

When building the output of the function $\hat{f}$, Theorem 1 guarantees that the described procedure yields a unique result. This means that, without loss of generality, we can always assume a unique path in $D(\underline{t})$ producing the output of $\hat{f}$ exists. Since a path is nothing more than a collection of edges, we denote it by $P \subseteq E$. Note that considering P as a subset of E results in neglecting the order of the vertices. Although this could be an issue with generic graphs, the particular structure of E makes such a set representation unambiguous.

Definition 9. *Let $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ be the Boolean function implemented by a gate and let $D(\underline{t})$ be the graph used to compute $\hat{f}$ on input $((t_1, l_1), \ldots, (t_n, l_n)) \in (T \times$*

$I)^n$. For every edge $(\underline{v}, \underline{w}) \in E$, we define the edge-label function $e_{\text{lab}} : E \to I$ *as follows:*

$$e_{\text{lab}}(\underline{v}, \underline{w}) = l_i$$

where i is such that $v_i \preceq_1 w_i$ and $v_j = w_j$ for every $j \neq i$.

Note that e_{lab} is well defined by definition of E. The definition of literifier for a single gate assuming its input is in transient-variable representation follows.

Definition 10. *The* literifier *of a gate implementing a Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ is the function $L_f : (T \times I)^n \to I$ such that:*

$$L_f((t_1, l_1), \ldots, (t_n, l_n)) = \bigcup_{(\underline{v}, \underline{w}) \in P} e_{\text{lab}}(\underline{v}, \underline{w})$$

Recall that each edge of $D(\underline{t})$ represents a change in one component of the vector $\underline{t} = (t_1, \ldots, t_n)$. For instance, let us assume that $e \in P$ links two vertices which differ in the j$^{\text{th}}$ component. The edge-label function e_{lab} is firstly used to label e with the literal transient corresponding to the j$^{\text{th}}$ component, hence l_j. Once this is done for every edge in the path P, the literifier returns the union of the labels.

Example 6. Following Example 4, let us compute the literifier $L_{\texttt{AND}}((10, \{1\}), (01, \{2\}))$ associated to the gate computing s_1. Figure 4 depicts a similar situation as in Fig. 1 and the same discussion follows. In addition, we apply the edge-label function e_{lab} to the edges in path P and compute the following literifier.

$$L_{\texttt{AND}}((10, \{1\}), (01, \{2\})) = \{2\} \cup \{1\} = \{1, 2\}$$

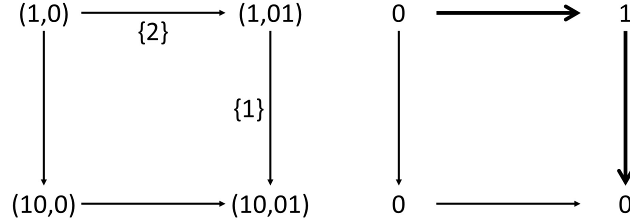

Fig. 4. $D(10, 01)$ and $D_{\texttt{AND}}(10, 01)$

Remark 2. Similarly to Remark 1, the above is just a formal procedure to compute literifiers. Once f is known and fixed, more straightforward approaches are possible. For instance, the following is the literifier associated to a gate implementing the Boolean function $\texttt{AND} : \mathbb{Z}_2^n \to \mathbb{Z}_2$:

$$L_{\texttt{AND}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \exists j \in \{1, \ldots, n\} \text{ such that } t_j = 0 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

where $J = \{j \in \{1, \ldots, n\} \mid \ell(t_j) > 1\}$. Intuitively, the upper branch states that if there exists one input which is the fixed 0, then the output will be the fixed 0 no matter how other inputs change. Since the output is fixed, no power is consumed and the set of leaked variables is empty. Otherwise, the union of all literal transients corresponding to non-constant transients is returned. Since we are in the second branch there is no constant 0 transient, which results in the rule excluding only literal transients equal to the constant 1, as they do not contribute to the switch activity of an AND gate. We refer to Appendix A for a list of other compact definitions of literifiers. As before, we stress that such a simplification has a positive impact on performance.

4.2 Application to circuits

We conclude this section by showing how to apply the LP model to a given circuit with m inputs and k gates. For instance in Figs. 1 and 3, on one hand it is immediate that the transient-variable representation of gate computing s_1 is the one shown in Example 5, but on the other it is less clear what it should be for gates whose inputs are not the inputs of the circuit, *e.g.* for the one computing s_3.

We recall that we denote by $\underline{X} = (X_1, \ldots, X_m)$ the input variables and by $\underline{s} = (s_1, \ldots, s_k)$ the state variables of a circuit. Moreover we denote by $S_j(\underline{X}, \underline{s})$ the Boolean function s_j is computed by, which can depend on all input and state variables except s_j itself.

The idea is simply proceeding by height: the only gates we can directly compute literifiers for are those at height 1, since the input literal transients are just singletons of input variables. Once all literifiers at height 1 have been computed, we can apply those at height 2: their input literal transients can be either singleton of input variables or outputs of gates at height 1. This procedure always terminates as there are finitely many gates and is well-defined as there are no feedbacks.

Example 7. We conclude what Example 6 has begun by computing all literifiers of Example 4. The only other gate at height 1 is the one computing s_2, for which we have the following.

$$L_{\texttt{OR}}((01, \{2\}), (0, \{3\})) = \{2\}$$

We now have all the information to compute the literifier for the last gate.

$$L_{\texttt{XOR}}((010, \{1, 2\}), (01, \{2\})) = \{1, 2\} \cup \{2\} = \{1, 2\}$$

Compact definitions of $L_{\texttt{OR}}$ and $L_{\texttt{XOR}}$, in the same fashion as in Remark 2, can be found in Appendix A.

Figure 5 depicts the final outcome of the LP model applied to the circuit in Fig. 2 during transition $100 \to 010$. Essentially, the LP model adds one literal transient per gate to the output of the glitch-counting algorithm. They describe

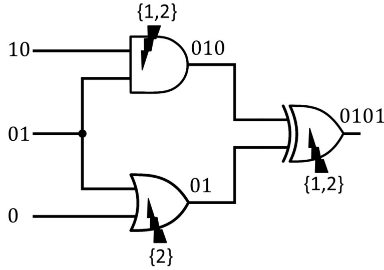

Fig. 5. Application of literifiers to a circuit

which input variables cause a particular gate to switch and whose values could then be leaked through the power consumption. Collecting such an information for all transitions gives the designer a powerful tool to predict possible flaws. In the next section we deepen this discussion while providing a real-world use case.

Final remarks. In the present subsection, we have shown how to practically apply the LP model to the netlist of a circuit. Although the example we have considered was trivial, the LP model is a formal tool to analyse netlists with an arbitrary number of inputs and gates, where an ad hoc analysis would require much more effort. Once a netlist and an input transition are fixed, the LP model provides a list of variables based on which a risk assessment in the context of side-channel analysis is facilitate. As the next section will suggest, a full analysis would require the LP model to run over every non-trivial input transition, hence $2^{2m} - 2^m$ times where m is the number of inputs and where we have subtracted transitions from an input to itself as they clearly do not produce any consumption in our power model. Such exponential requirement is a drawback of our approach: a deeper insight will be given in Sec. 6. Also, it is possible to reduce the number of transitions over which the LP model needs to be run by developing heuristics specifically designed for a circuit. Finally, for a fixed transition the overall complexity is asymptotically bounded by the running time of the glitch-counting algorithm, described in Theorem 2.

5 Case of study: Keccak

The present section provides an application of the LP model to Keccak. We show how to face the following issue thanks to our tool: an unprotected implementation of Keccak's non-linear layer is obviously susceptible to side-channel attacks, but a possible 2-shares scheme is still weak in the first order because of glitches. We formally show the validity of the latter statement while suggesting a deeper insight on how to circumvent the issue without adopting more costly countermeasures. The reason why we chose to adopt Keccak as our case of study mainly relies on it being deployed in real-world applications while still having a not too complex structure. It is then the ideal candidate for being a test bench.

KECCAK is a family of sponge functions that uses a permutation from a set of seven possible ones as a building block [2]. The permutations are defined over a state $s \in \mathbb{Z}_2^b$ where $b = 25 \times 2^\ell$ is called *width* of the permutation and $\ell \in \{0, \ldots, 6\}$. Each round is formed of five maps: three linear maps aiming at diffusion and dispersion, one non-linear map aiming at confusion and one addition with round constants. When it comes to implement sharing schemes, linear maps can be directly applied to each share separately. By contrast, non-linear maps need to handle every share to preserve correctness. Therefore we focus on the only non-linear map of KECCAK, namely $\chi : \mathbb{Z}_2^5 \to \mathbb{Z}_2^5$ acting on groups of five bits of the state called *rows*. For a complete description of KECCAK we invite the reader to refer to the work of Bertoni *et al.* [2].

The map χ can be seen as the parallel application of five identical maps each defined on three consecutive bits (modulo 5) of a row. Formally:

$$\chi_i : r_i \leftarrow r_i \oplus \overline{r}_{i+1} r_{i+2} \tag{1}$$

where $r \in \mathbb{Z}_2^5$ denotes a row of the KECCAK state and the index i is computed modulo 5. For our analysis, it is important to note that the five instances of the map $\chi_i : \mathbb{Z}_2^3 \to \mathbb{Z}_2$ are completely independent, they do not share gates in their computation. As a result, we can focus on a specific χ_i without loss of generality. The sharing scheme we adopt in our analysis is a 2-shares Boolean scheme, *i.e.* each row is split in two shares $a, b \in \mathbb{Z}_2^5$ such that $r = a \oplus b$ [1]. Our results can be easily generalised to many shares. In this setting, (1) can be masked as follows:

$$\begin{aligned} a_i &\leftarrow a_i \oplus \overline{a}_{i+1} a_{i+2} \oplus a_{i+1} b_{i+2} \\ b_i &\leftarrow b_i \oplus \overline{b}_{i+1} b_{i+2} \oplus b_{i+1} a_{i+2} \end{aligned} \tag{2}$$

where a straightforward computation shows that (2) are correct as (1) is simply retrieved by `XOR`ing them. If the order of operations was kept fixed from left to right, *e.g.* using software constraints, then the above sharing scheme would be secure in the first order. However if (2) were implemented in hardware, such condition could not be guaranteed, for instance because of glitches. This results in possible vulnerabilities when the values a_{i+2} and b_{i+2} are involved in the computation of the 3-inputs `XOR` at the same time, in one of the two branches.

As both the glitch-counting algorithm and the LP model work with netlists, the first step in the analysis of (2) is to produce one. It can be easily seen that the two branches are symmetric, hence we can focus only on the first without loss of generality, *i.e.* the one computing a_i. Figure 6 depicts its representation as an hardware netlist, where the naming conventions presented at the beginning of Subsec. 3.2 have been used. In particular, the input vector $X = (X_1, X_2, X_3, X_4)$ corresponds to $(a_i, b_{i+2}, a_{i+1}, a_{i+2})$.

Our analysis targets the netlist in Fig. 6 and proceeds as follows. First of all an input transition is fixed among all the $2^8 - 2^4 = 240$ non-trivial possible ones. Then, the glitch-counting algorithm is applied as shown in Subsec. 3.2 and all the transients are computed, one per gate. Table 2 reports the execution of the glitch-counting algorithm for the input transition $0110 \to 0001$.

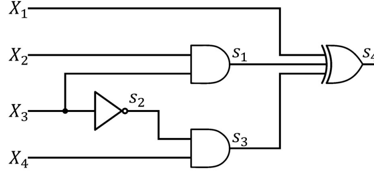

Fig. 6. Netlist of χ_i for one share

Table 2. Glitch-counting algorithm's execution for the shared χ_i circuit

h	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_3$	$\mathbf{s}_1$	$\mathbf{s}_2$	$\mathbf{s}_3$	$\mathbf{s}_4$
0	0	10	10	01	1	0	0	1
1	0	10	10	01	10	01	0	1
2	0	10	10	01	10	01	01	10
3	0	10	10	01	10	01	01	101
4	0	10	10	01	10	01	01	101

At this point, suitable literifiers can be applied as described in Subsec. 4.2, hence starting from gates at height 1. In our example, this means computing the literifiers corresponding to s_1 and s_2 first, respectively an AND and NOT literifiers.

$$L_{\texttt{AND}}((10,\{2\}),(10,\{3\})) = \{2\} \cup \{3\} = \{2,3\}$$
$$L_{\texttt{NOT}}(10,\{3\}) = \{3\}$$

There are two gates at height higher than 1: first we compute $L_{\texttt{AND}}$ for the gate computing s_3 and finally $L_{\texttt{XOR}}$ is applied.

$$\begin{aligned} L_{\texttt{AND}}((01,\{3\}),(01,\{4\})) &= \{3\} \cup \{4\} = \{3,4\} \\ L_{\texttt{XOR}}((0,\{1\}),(10,\{2,3\}),(01,\{3,4\})) &= \{2,3\} \cup \{3,4\} = \\ &= \{2,3,4\} \end{aligned}$$

We refer the reader to Appendix A for the definitions of $L_{\texttt{NOT}}$ and $L_{\texttt{XOR}}$. Figure 7 summarises the execution of both the glitch-counting algorithm and of the LP model for the transition $0110 \rightarrow 0001$.

To take the most out of the proposed method, a vulnerability definition based on critical combinations of variables needs to be formulated. This is checked among all the literal transients produced by the model, which has been run over all possible non-trivial input transition.

In the specific case of Keccak, a natural vulnerability of the circuit in Fig. 6 arises when the two variables a_{i+2} and b_{i+2} are processed in the same moment by the last XOR gate, as this could leak the value $a_{i+2} \oplus b_{i+2} = r_{i+2}$ which is unshared. In our model, this translates to the existence of $\{2\}$ and $\{4\}$ in the same literal transient corresponding to the XOR gate, since X_2 and X_4 are

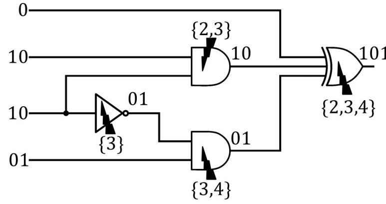

Fig. 7. Shared χ_i circuit after LP model

the input variables corresponding to a_{i+2} and b_{i+2}. By running the model for all the $2^8 - 2^4$ non-trivial possible input transitions, we have found that 32 out of 240 match our vulnerability definition and could then lead to a critical first order leakage. At this point, the designer possesses valuable information to base security improvements on. In particular, leaving our gate-level abstraction, the designer can carefully tune place-and-route paths in order to minimise the occurrence and impact of those critical transitions. If such an operation is not feasible, the designer still has a valid and sound criterion why to switch to an higher number of shares (3 in the case of KECCAK, since χ has degree 2). It is important to note that further analyses, possibly by means of different and finer vulnerability definitions, can be carried out without rerunning the whole model.

The sharing scheme we have analysed [1] has not gained much popularity due to its weakness in the presence of glitches. However, our analysis is able to capture more details: we can quantify and list all those transitions threatening the security of unshared values. In this case a designer could just patch them while being sure that all the others will never show a critical leakage of the first order even in the presence of glitches. Since our aim was just to exemplify the potentiality of our model, we consider the latter modification as being out of scope for the present work, but an interesting future direction towards sound and lightweight countermeasures.

6 Computational effort and multi-output circuits

As we briefly mentioned while justifying the choice of KECCAK, its combinatorial circuits are relatively simple and allow to verify the correctness of the proposed method easily. Since our aim is not to find a specific method for KECCAK but a rather generic methodology, there are two further topics that need to be addressed: the computational complexity for a generic circuit and the applicability of the method to multi-output combinatorial circuits.

The former topic has been partially addressed in Subsec. 3.2 for the glitch-counting algorithm (Theorem 2) and in Subsec. 4.2 for the LP model. If we refer to KECCAK as a practical example and we think at an implementation performing one round in one clock cycle, the target combinatorial circuit is the concatenation of θ, one of the linear maps, and χ [2]. This combinatorial circuit

can be seen as a circuit with 33 input bits and 1 output bit in the unprotected version, while the protected version using two shares is a 44-inputs circuit [1]. As described in Subsec. 4.2 this would turn in computing the propagation of glitches through k gates for each of the $2^{2m} - 2^{m}$ non-trivial input transitions. Considering that the computation can be parallelised and the evaluation of the glitch-counting algorithm is not a very complex computation, we claim that the method could be applicable for a circuit with 44 inputs but would require a well optimized implementation.

Multi-output circuits are also a very interesting target. In such circuits there are gates contributing to the computation of different output bits. One approach for tackling these circuits is to divide the circuit in n independent circuits with single output, where n is the number of outputs of the initial combinatorial logic, and apply the proposed method to each of them separately. Such approach could however introduce an overhead as a single input might be used by more than one sub-circuits. A further, more advanced solution for approaching multi-output circuits and the computational effort when the number of inputs is large would be the development of heuristic approaches as adopted by silicon compilers. We see this as a future development. Finally note that there is nothing preventing the model to be applied to multi-output circuits as it is, but it would be required to develop meaningful vulnerability definitions based also on the cryptographic algorithm. A similar discussion applies to high-order analysis. The LP model can still be used but more sophisticated vulnerability definitions are needed to interpret its results.

7 Conclusions

In their work, Brzozowski and Ésik [5] have developed a mathematical structure to estimate the potential waste of power of a circuit due to glitches. Our first contribution is the expansion of such framework to include a formal definition of leakage. We have then defined a formal procedure to analyse circuits in the context of side-channel analysis which take into account the effect of glitches on the order of operations. Our work analyses only the combinatorial logic, hence achieves a good level of generality since it is not touched by real-world constraints. As a consequence, the LP model allows to retrieve how much a given protection scheme can be affected by glitches, thus enabling a deep analysis. Using the proposed methodology, a designer might explore alternative designs for solving local problems of glitches instead of adopting more costly solutions.

Acknowledgements

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR).

Furthermore, Marco Martinoli has been supported in part by the Marie Skłodowska-Curie ITN ECRYPT-NET (Project Reference 643161). Finally, we thank Maria Chiara Molteni for corrections and useful comments.

References

1. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Building power analysis resistant implementations of Keccak. In: Second SHA-3 candidate conference (2010)
2. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 313–314. Springer, Heidelberg (May 2013)
3. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 326–343. Springer, Heidelberg (Dec 2014)
4. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: A more efficient AES threshold implementation. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT 14. LNCS, vol. 8469, pp. 267–284. Springer, Heidelberg (May 2014)
5. Brzozowski, J., Ésik, Z.: Hazard algebras. Formal Methods in System Design 23(3), 223–256 (2003)
6. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (Aug 1999)
7. Kocher, P.C., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. Journal of Cryptographic Engineering 1(1), 5–27 (2011)
8. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: Revealing the secrets of smart cards, vol. 31. Springer Science & Business Media (2008)
9. Mangard, S., Popp, T., Gammel, B.M.: Side-channel leakage of masked CMOS gates. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 351–365. Springer, Heidelberg (Feb 2005)
10. Mangard, S., Pramstaller, N., Oswald, E.: Successfully attacking masked AES hardware implementations. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 157–171. Springer, Heidelberg (Aug / Sep 2005)
11. Mangard, S., Schramm, K.: Pinpointing the side-channel leakage of masked AES hardware implementations. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 76–90. Springer, Heidelberg (Oct 2006)
12. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of AES. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (May 2011)
13. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 06. LNCS, vol. 4307, pp. 529–545. Springer, Heidelberg (Dec 2006)
14. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of nonlinear functions in the presence of glitches. In: Lee, P.J., Cheon, J.H. (eds.) ICISC 08. LNCS, vol. 5461, pp. 218–234. Springer, Heidelberg (Dec 2009)
15. Prouff, E., Roche, T.: Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 63–78. Springer, Heidelberg (Sep / Oct 2011)
16. Rabaey, J.M., Chandrakasan, A.P., Nikolic, B.: Digital integrated circuits, vol. 2. Prentice hall Englewood Cliffs (2002)
17. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 764–783. Springer, Heidelberg (Aug 2015)

A Appendix

We report the compact definition of literifiers for the most basic gates, namely `NOT`, `AND`, `OR` and `XOR`, in the same fashion of Remark 2 of the main paper.

A.1 `NOT`

The idea is simple: if the input transient does not switch, so does the output and then no power is consumed. Otherwise, the only possible literal transient is returned.

$$L_{\texttt{NOT}}(t, l) = \begin{cases} \emptyset & \text{if } \ell(t) = 1 \\ l & \text{otherwise} \end{cases}$$

A.2 `AND`, `OR`

The `AND` literifier has already been presented in the main paper in Remark 2 and a completely similar discussion applies to the `OR`.

$$L_{\texttt{AND}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \exists j \in \{1, \ldots, n\} \text{ such that } t_j = 0 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

$$L_{\texttt{OR}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \exists j \in \{1, \ldots, n\} \text{ such that } t_j = 1 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

where $J = \{j \in \{1, \ldots, n\} \mid \ell(t_j) > 1\}$.

A.3 `XOR`

This case is slightly different than the previous two, since a `XOR` gate switches whenever at least one input switches. This restricts the cases in which $L_{\texttt{XOR}}$ returns the empty set.

$$L_{\texttt{XOR}}((t_1, l_1), \ldots, (t_n, l_n)) = \begin{cases} \emptyset & \text{if } \forall j \in \{1, \ldots, n\},\ \ell(t_j) = 1 \\ \bigcup_{j \in J} l_j & \text{otherwise} \end{cases}$$

where $J = \{j \in \{1, \ldots, n\} \mid t_j \neq 0\}$.