Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits

Hassan Eldib, Meng Wu, and Chao Wang(✉)

Department of ECE, Virginia Tech,
Blacksburg, VA 24061, USA
chaowang@vt.edu

Abstract. Fault sensitivity analysis (FSA) is a side-channel attack method that injects faults to cryptographic circuits through clock glitching and applies statistical analysis to deduce sensitive data such as the cryptographic key. It exploits the correlation between the circuit's signal path delays and sensitive data. A countermeasure, in this case, is an alternative implementation of the circuit where signal path delays are made independent of the sensitive data. However, manually developing such countermeasure is tedious and error prone. In this paper, we propose a method for synthesizing the countermeasure automatically to defend against FSA attacks. Our method uses a syntax-guided inductive synthesis procedure combined with a light-weight static analysis. Given a circuit and a set of sensitive signals as input, it returns a functionally-equivalent and FSA-resistant circuit as output, where all path delays are made independent of the sensitive signals. We have implemented our method and evaluated it on a set of cryptographic circuits. Our experiments show that the method is both scalable and effective in eliminating FSA vulnerabilities.

1 Introduction

The rising security risks in embedded computing devices in cyber physical systems (CPS) and the Internet of Things (IoT) have led to the pervasive use of cryptographic modules, often implemented in hardware, to guarantee secure authentication, privacy, and integrity [3]. In particular, various light-weight cryptographic primitives have been recommended for securing resource-constrained devices such as Smartcards and RFID tags [10,27]. Although these cryptographic algorithms are designed to be secure against brute-force attacks, their actual implementations may not be as secure. Indeed, there have been many reported cases of attacks on cryptographic modules in embedded systems, the majority of which were through side-channel attacks [5,33,36].

Fault sensitivity analysis (FSA) is a side-channel attack [20,32,37] that exploits the correlation between secret data and the time needed to propagate these data through a cryptographic circuit. In particular, there is a large number of reported cases of such attacks on lightweight block ciphers [4,21,23,24,29,30,38,39,42]. With physical access to the circuit, an attacker can introduce clock

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 343–363, 2016.
DOI: 10.1007/978-3-319-41540-6_19

glitches until logical errors occur in the output. The attacker measures the fault intensity *critical level* [24], which is the lowest fault intensity level where a faulty output first occurs. This critical level can be compared, via a statistical analysis [9], with a set of simulated critical levels computed *a priori*, to determine the most likely values of the secret signals [41].

A countermeasure is an alternative implementation of the circuit where all signal path delays are made independent of the sensitive data. However, manually developing such countermeasure is tedious and error prone. Therefore, we propose a new method for constructing the countermeasure automatically. Given a circuit C and a set S of sensitive signals as input, our method relies on inductive synthesis [2,25,40] to compute a functionally equivalent circuit that is guaranteed to be resistant to FSA attacks. More specifically, it first generates a candidate circuit C' that, at least for some input values, produces the same output as C, and is likely to have balanced delay along the sensitive paths. Then, it invokes a verification subroutine to check that C' and C are functionally equivalent for all input values and C' is FSA-resistant. If C' passes this verification step, then a countermeasure has been synthesized. Otherwise, we block this bad countermeasure and generate another candidate circuit. The iterative *guess-and-check* procedure continues until a valid solution is found, or it runs out of time or memory.

Although inductive synthesis has been successfully applied in main domains [2,13,25,26,31,40], this is the first time that it is used to mitigate fault attacks on cryptographic circuits. In practice, however, the bottleneck of applying inductive synthesis to practical applications is the limited scalability of the synthesis tool. Since the design space is enormous, directly applying inductive synthesis to large circuits often does not work. Fortunately, in this application, FSA-resistant circuits are amenable to compositional analysis. That is, the delay of a path in a circuit is the summation of the delays of its individual path segments. Based on this observation, we have developed a *divide-and-conquer* approach, which first divides the circuit into pieces, then synthesizes a countermeasure for each piece, and finally composes them to form the final solution. In this context, our verification subroutine is implemented as an equivalence checker for C' and C augmented with a static analysis procedure for computing the delays along their sensitive paths.

We have implemented our method and evaluated it on a set of realistic cryptographic circuits, including a set of nonlinear components of AES and MAC-Keccak. Our experimental results show that the new method is both scalable and effective in eliminating FSA vulnerabilities. Furthermore, the resulting circuits are consistently smaller than the countermeasures obtained by competing techniques. To summarize, this paper makes the following contributions:

- We propose the first fully automated method for synthesizing FSA-resistant cryptographic circuits.
- We develop a new partitioned synthesis procedure to improve the scalability of our method.

– We demonstrate the effectiveness of our new method on realistic cryptographic benchmarks.

The remainder of this paper is organized as follows. First, we illustrate our main ideas using examples in Sect. 2. Then, we establish the notation in Sect. 3 and present our baseline inductive synthesis algorithm in Sect. 4. We present our partitioned synthesis procedure in Sects. 5 and 6 and our experimental results in Sect. 7. We review related work in Sect. 8 and finally give our conclusions in Sect. 9.

2 Motivation

In this section, we illustrate the main ideas behind our countermeasure synthesis method using examples. Specifically, we use the PPRM1 AES S-box implementation proposed by Morioka and Satoh [34] as the original circuit, shown partially in Fig. 1. The standard Advanced Encryption Standard (AES) algorithm has four main functions that are repeated for a number of rounds depending on the required length of the secret key. Among the four functions, S-box is the only nonlinear function. In cryptographic engineering, nonlinear functions are often the hardest to implement and protect against side-channel attacks. In particular, the S-box implementation scheme in Fig. 1 is a widely used benchmark in the cryptography field. The entire circuit is constructed from two parts: a network of XOR gates and a network of AND gates. For simplicity, we only use the network of AND gates to illustrate our synthesis algorithm. Later in this paper, we will explain how our method can be applied to larger circuits, by first partitioning a circuit into smaller regions, then synthesizing a countermeasure for each region, and finally composing the partial solutions to form the countermeasure for the whole circuit.

The circuit in Fig. 1 is vulnerable to FSA attacks because the time taken for computing the output signals depends not only on the structure of the circuit but also on the values of the sensitive input signals (e.g., bits in the cryptographic key). Consider the output signal O_0 of the AND network and the two input signals In_2 and I_{chain}. Let $\tau(I_{chain})$ and $\tau(In_2)$ be the signal arrival times of I_{chain} and In_2, respectively. If we assume that all input signals In_0-In_7 have the same arrival time, we have $\tau(I_{chain}) > \tau(In_2)$. Furthermore, the value of $\tau(I_{chain})$ depends on the value of the input signals $In_0, In_1, In_3, In_4, In_5, In_6$ as well as the number of gates along the path.

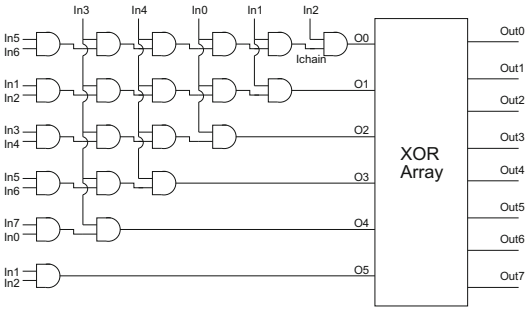

Fig. 1. PPRM1 AES S-box that is vulnerable to FSA.

If we assume that In_2 is a sensitive signal, the aforementioned mismatch in the arrival time of the input signals of the last AND gate will make signal O_0 sensitive as well.

In the context of FSA attacks, we say that the output O_0 is statistically dependent on the sensitive variable In_2 for the following reasons. When the value of In_2 is logical 1, the delay $\tau(O_0)$ is determined by $\tau(I_{chain})$. In contrast, when the value of In_2 is logical 0, the delay $\tau(O_0)$ is determined by $\tau(In_2)$. Since $\tau(I_{chain}) > \tau(In_2)$, the dependency relation and the secret value of In_2 cause a leak of the sensitive information, which is recoverable by correlation-based statistical analysis techniques [32,37].

Previously published countermeasures for FSA, typically hand-crafted by cryptographic system engineers [19,22], rely on adding buffers (delay components) to certain input-output paths to eliminate such information leaks. For example, a recently-published countermeasure in Fig. 2 was implemented by manually analyzing the input-output signal paths for each output gate, and then adding buffers accordingly to make the delay along all sensitive paths equal. However, such countermeasures often result in an unnecessarily large number of logic gates inserted into the circuit, thus leading to higher area cost and energy cost.

Our method, in contrast, can generate more efficient countermeasures. Figure 3 illustrates the circuit synthesized by our method, which is functionally equivalent to the original circuit and at the same time guarantees to be FSA-resistant. That is, the path delays are independent of the sensitive inputs. Furthermore, it is more efficient than the prior solution in Fig. 2 in terms of area cost as well as the latency of the circuit. In fact, our new solution uses only 13 logic gates as opposed to the 41 gates used by the hand-crafted circuit in Fig. 2, and the 21 gates used by the original circuit in Fig. 1.

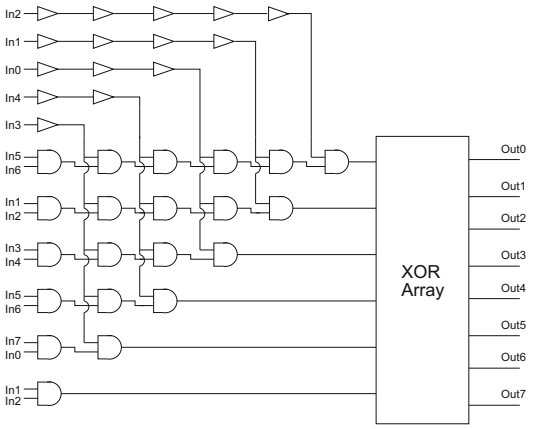

Fig. 2. S-box with buffered countermeasure [22].

It is worth pointing out that, currently, no EDA tool can be used to generate FSA-resistant circuits such as the one shown in Fig. 3. For example, traditional logic synthesis and optimization techniques, such as two- and multi-level minimizations [28], do not have the capability of identifying sensitive signal paths or ensuring that these paths exhibit the same delay. We will demonstrate this in the experiments section and explain why it is difficult to leverage state-of-the-art EDA algorithms, such as the ones implemented in the ABC tool [12], to generate FSA-resistant circuits.

Our new method leverages the idea of syntax-guided *inductive synthesis* to generate FSA countermeasures. Although inductive synthesis has been applied to many domains [2, 13, 25, 26, 31, 40], this is the first time it is used to eliminate FSA vulnerabilities in cryptographic circuits. When inductive synthesis techniques are applied to large circuits, however, scalability becomes a problem because the synthesis procedure has to search an extremely large design space for an alternative and FSA-resistant implementation of the given circuit. As mentioned earlier, we propose to solve this scalability problem using a *partitioned* synthesis procedure. After establishing the notion and present our baseline synthesis algorithm in Sects. 3 and 4, we will explain how to leverage the *divide-and-conquer* principle to scale our synthesis method to large circuits.

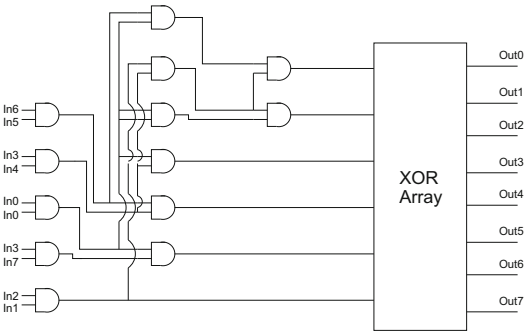

Fig. 3. S-box with our new countermeasure.

3 Preliminaries

Fault attacks are typically conducted by changing the physical environment of the circuit to introduce logical errors. Although various fault injection techniques have been used in practice, in this work, we focus on faults injected by disturbing the external clock, and more specifically, by increasing the clock frequency beyond its normal range.

Fault Sensitivity Analysis (FSA). In digital circuits, the time taken by the output to change from logical 1 to logical 0 (or vice versa) in response to changes in the inputs may depend on the circuit structure as well as values of the input/internal signals. This is important to attackers, because it means the impact of an injected fault will be significantly different depending on the internal states of the circuit. Consider the AND gate in Fig. 4, where T_A and T_B are the arrival time of input signals A and B, respectively, and T_{AND} is the gate's propagation delay. When $T_A < T_B$, i.e., signal B arrives later than signal A, the time taken for signal C to stabilize (T_c) depends on the value of signal A. Specifically,

- when A is logical 0, we have $T_c = T_A + T_{AND}$; and
- when A is logical 1, we have $T_c = T_B + T_{AND}$.

In other words, by observing the difference in T_c we can deduce the (sensitive) value of A based on our knowledge of the circuit structure. Such dependency is not unique to AND gates; other logic gates have similar properties. For a large circuit, it is not uncommon for delays along input-to-output paths to depend on the values of sensitive signals. However, to launch a successful attack, merely injecting faults is not enough; these faults must become observable.

In practice, the chance of producing a faulty output depends on the intensity of the faults injected to the circuit, for example, through over-clocking [20,23], as shown in Fig. 5. Note that the information leak is specific to the *faults* as opposed to generic *timing* attacks. Without fault injection, the tiny delay variation in the combinational logic part of this sequential circuit ($C(i, o)$) would not be visible to attackers. This is because the output signals (o) are always synchronized by the flip-flops before they are propagated to the next clock cycle. However, faults injected via clock glitching may destabilize the flip-flop based synchronization scheme, causing the information leak.

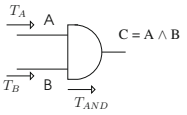

Fig. 4. Fault sensitivity of an AND gate.

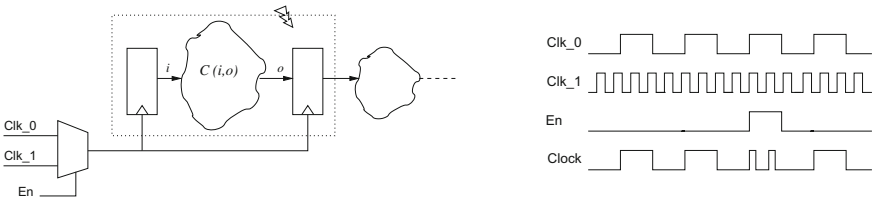

Fig. 5. Injecting faults via clock glitching [20,23]: (a) the circuit and (b) the timing diagram.

Following Ghalaty et al. [23], we define the *fault intensity* and *fault sensitivity* of a circuit as follows. The fault intensity is the strength of the faults by which a circuit is pushed outside of its normal operating condition. Since faults are introduced through clock-glitching, the fault intensity corresponds to the shortened clock cycle. The fault sensitivity is defined as the fault intensity where the circuit starts to generate faulty output. In our work, the fault intensity corresponds to the critical paths in the circuit. FSA, in particular, relies on exploiting the dependency between the values of sensitive input signals and the fault sensitivity critical level, or simply the *critical level*, under which injected faults become observable in the circuit's output.

Attacks and Countermeasures. We assume the attacker has knowledge of the circuit under attack. In this case, an FSA attack often consists of three steps:

1. The attacker injects faults through clock-glitching and measures the critical level of the circuit for a set of N randomly generated plaintexts (inputs);
2. The attacker computes, using computer simulation, the critical level for each of the N selected plaintexts and combinations of the sensitive data values;
3. The attacker performs a correlation analysis between the *measured* critical level and the *simulated* critical level for each combination of sensitive data values.

In the third step above, the sensitive data value combination that results in the highest correlation coefficient will be identified and used to deduce the sensitive data value.

Since the necessary condition for FSA attacks is having easily distinguishable fault sensitivity *critical levels* for various sensitive data value combinations, the goal of a countermeasure is to disable this condition. Generally speaking, among output signals whose arrival time depend on the sensitive data, the greater the difference in their arrival times, the more distinguishable the critical levels, and consequently, the higher the chance that attackers can successfully deduce the sensitive data. Therefore, the ideal countermeasure is an alternative and functionally-equivalent implementation of the original circuit that has the same delay for all its sensitive input-output signal paths.

Previously published FSA countermeasures [19,22] mainly rely on adding delay elements to certain parts of the circuit to make the arrival time of all output signals independent of the sensitive data. However, this approach may add an unnecessarily large number of delay elements (buffers), which results in higher area cost and power cost (Fig. 2). In contrast, our method can generate a potentially more efficient countermeasure (Fig. 3) using the new inductive synthesis technique.

4 Synthesis of FSA Countermeasures

Our method takes a circuit C and a set S of sensitive signals as input and returns an FSA-resistant circuit C' as output. It consists of a synthesis subroutine and a verification subroutine, where the synthesis subroutine guesses a candidate solution and the verification subroutine checks whether it is a valid solution. In this work, the verification subroutine has to check two properties: (1) the new circuit C' is functionally equivalent to C; and (2) the new circuit C' is FSA-resistant.

More formally, we say that two circuits $C(i, o)$ and $C'(i', o')$ are functionally equivalent if $(i = i') \rightarrow (o = o')$. Let $\pi_A(i'_A, o'_C)$ and $\pi_B(i'_B, o'_C)$ be two sensitive paths in C', where $i'_A, i'_B \in S$ are the sensitive inputs and $i'_A \neq i'_B$. Let $\tau_v(\pi)$ be the delay of the path π under the input valuation v – different input values can lead to different delays of the same path. We say that C' is FSA-resistant if $\tau_v(\pi_A(i'_A, o'_C)) = \tau_v(\pi_B(i'_B, o'_C))$ for *any two such* paths π_A and π_B and *any valuation* v of the input signals.

To reduce the computational cost, we choose to formulate the synthesis subproblem in a way that every solution C' is guaranteed to be FSA-resistant. Therefore, the verification subroutine only needs to check the functional equivalence of C and C'. The main idea behind our synthesis subroutine is to construct a *template circuit*, whose instantiations are guaranteed to be FSA-resistant. Without loss of generality, we assume all logic gates have *unit propagation delay*, and being FSA-resistant means that all paths from sensitive signals to the output have an equal number of logic gates. Consider the example in Fig. 1 again, whose template circuit is shown in Fig. 6. Gates and input/output signals in this diagram are distributed to five different levels, where Level 0 consists of only output

signals, Level 4 consists of only input signals, and in between the two levels are the logic gates of various types. This is a template because neither the types of internal logic gates nor the connections between the gates have been fixed.

To make sure all instantiations of this template circuit are FSA-resistant, we require that (1) all sensitive input nodes are placed on the same level–although they do not have to be at the bottom level–and (2) the output and input of each node are constrained to be connected, respectively, to either a node of one level higher or a node of one level lower, to ensure that the level assigned to this node remains valid. Implicitly, the above constraints guarantee an equal number of gates between each output signal and the corresponding sensitive inputs. Note that the circuit only needs to have equal-delay paths for each output; it does not need to have the same delay for all outputs.

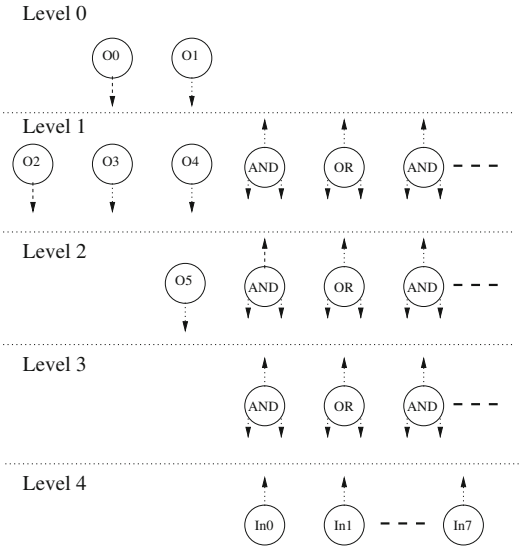

Fig. 6. FSA-resistant template circuit structure.

To reduce the computational overhead within the SyGuS tools, we also statically estimate the level where the output signals should be placed in the template circuit, based on the number of inputs they are connected to and the level required for each input. Since this is only an estimation, initially we assign the minimal depth needed to separate an output node from the sensitive input nodes to fit all nodes in between. If the depth turns out to be insufficient, the synthesis subroutine would fail to return a solution, in which case we shift the output nodes one level up to enlarge the design space, and invoke the synthesis subroutine again.

In principle, this synthesis subproblem can be specified using the SyGuS specification language and solved using the associated tools developed by Alur et al. [2] (or the Sketch tool by Solar-Lezama [40]). In practice, however, there are two significant challenges. The first challenge is due to a limitation in the implementation of SyGuS tools. Specifically, they are designed for synthesizing a function with a single output, whereas we need to synthesize a circuit with multiple output signals, and these output signals must share logic gates that fall in their cone-of-influence as much as possible. Although SyGus allows the use of multiple functions (each with a single output) to mimic a circuit with multiple output signals, in such case, the SyGuS tools would not return a solution where

the internal nodes are shared among these functions. Therefore, we need to modify the SyGuS tools, so that internal nodes can be shared among multiple functions.

Figure 7 shows an example SyGuS specification. The original circuit is given by the function `Spec`, which defines the output signals O_o and O_1 of the circuit in Fig. 1. Note that **ite** is a special operator that we use as a *work-around* since SyGuS does not allow the output to be a concatenation of bits. Inside the SyGuS tool, we made some modification to permit the solver to return a circuit where different output signals share the same set of intermediate logic gates – this is crucial for us to generate a compact circuit. The template of the output circuit is given by the function Impl, which specifies the pool of components that can be used by the synthesizer, including the output node `Start`, and nodes on the remaining three levels: `d0`, `d1` and `d2`. On each level, both AND and OR gates may be used. The primary inputs are `i0` – `i6`. The constraint at the bottom of the file states that the two circuits are functionally equivalent. Given this specification as input, the SyGuS tool will generate the desired countermeasure.

```
(define-fun Spec ((i0 Bool)(i1 Bool)(i2 Bool)(i3 Bool)(i4 Bool)(i5 Bool)(i6
    Bool)) Int
  (+ (ite (and i2 (and i1 (and i0 (and i4 (and i3 (and i5 i6)))))) 1 0 )
     (ite (and i1 (and i0 (and i4 (and i3 (and i1 i2))))) 2 0 ) )
)
(synth-fun Impl ((i0 Bool)(i1 Bool)(i2 Bool)(i3 Bool)(i4 Bool)(i5 Bool)(i6
    Bool)) Int
  ((Start Bool ( (+ (ite d0 1 0)
                    (ite d0 2 0)) ) )
   (d0 Bool ( (and d1 d1)
              (or d1 d1) ) )
   (d1 Bool ( (and d2 d2)
              (or d2 d2) ) )
   (d2 Bool ( (and d3 d3)
              (or d3 d3) ) )
   (d3 Bool ( i0 i1 i2 i3 i4 i5 i6 ) ) )
)
(constraint (= (Spec i0 i1 i2 i3 i4 i5 i6) (Impl i0 i1 i2 i3 i4 i5 i6) ) )
```

Fig. 7. Automatically generated synthesis subproblem for O_0 and O_1 (Fig. 1) in SyGuS language.

The second challenge is to scale up this new synthesis method to large circuits. Even with the optimizations mentioned above, state-of-the-art SyGuS tools can only handle small circuits, since as the circuit size increases, the design space that SyGuS has to search through increase dramatically. Although we believe the performance of SyGuS tools will continue to improve in the coming years, such improvement alone is unlikely to be sufficient for handling realistic circuits. Therefore, we propose a new method based on the idea of *divide-and-conquer*. It leverages a nice *compositionality* property of the FSA-resistant circuit: If each partition of a circuit is FSA-resistant, then the whole circuit is guaranteed to be FSA-resistant as well.

5 The Partitioned Synthesis Approach

To partition the given circuit, we first represent the combinational part as a directed acyclic graph (DAG), whose input nodes are either *primary inputs* or *pseudo primary inputs* (outputs of latches from the previous clock cycle). Then, we traverse the DAG in a topological order to identify the vulnerable (sensitive) output signals. Specifically, if there are discrepancies between the delays along different paths to the output from different sensitive inputs, we consider it to be vulnerable. For each vulnerable output signal, we build a circuit region by iteratively including logic gates in its fanin and fanout cones until the region size reaches a predefined limit. We invoke the SyGuS tool on each circuit region to synthesize the replacement circuit. By replacing the old circuit region with the new circuit, we can eliminate the vulnerability. This process of extracting, synthesizing, and replacing vulnerable circuit regions is repeated until no vulnerable circuit region exists any more.

5.1 The Overall Algorithm

The pseudocode of our partitioned synthesis procedure is shown in Algorithm 1, where P denotes the original circuit, *InputSort* denotes a map from each input of P to a type (sensitive or non-sensitive), *GatesPD* denotes a map from each gate in P to its propagation delay, and *GatesSyn* denotes a set of logic gates (components) to be used by SyGuS for synthesizing the new circuit. The parameter *lev* is a bound on the maximum number of levels of the new circuit region to be synthesized.

Algorithm 1. Partitioned FSA-countermeasure synthesis procedure.

```
1:  GEN-COUNTERMEASURE (P, InputSort, GatesPD, GatesSyn, lev) {
2:    while ( true ) {
3:      for each ( gate g ∈ P) {
4:        MaxPD[g] ← GETMAXPD (g, GatesPD, P)
5:        MinAr[g] ← GETMINAR (g, GatesPD, P)
6:        MaxAr[g] ← GETMAXAR (g, GatesPD, P)
7:      }
8:      sGate ← GETSENSITIVE (MaxPD, MinAr, MaxAr, P)
9:      if (sGate = ∅)
10:       return P;
11:     n ← 2^lev − 1
12:     newReg ← ∅
13:     while (newReg = ∅) {
14:       reg ← GETREG (sGate, MinAr, MaxAr, P, n)
15:       newReg ← SYNTHESIZE (reg, MinAr, GatesSyn, lev)
16:       n ← n − 1
17:     }
18:     P ← UPDATEREGION (P, reg, newReg)
19:   }
20: }
```

Our method first identifies a sensitive gate $sGate \in P$ (Lines 3–8), based on which it generates small circuit regions (Line 14). It starts by analyzing each gate $g \in P$ while creating three auxiliary tables:

– $MaxPD[g]$ denotes the maximum path delay from g to the output of P,
– $MinAr[g]$ denotes the minimum arrival time of any sensitive input to g, and
– $MaxAr[g]$ denotes the maximum arrival time of any sensitive input to g.

The subroutine GetSensitive returns the next sensitive gate *sGate*, which is a gate $g \in P$ such that the maximum arrival time $MaxAr[g]$ differs from the minimum arrival time $MinAr[g]$. In the presence of multiple choices, this subroutine returns a gate with the smallest propagation delay from the sensitive inputs. In the case of a tie, the gate with the maximum propagation delay $MaxPD[g]$ to the primary output is selected. This heuristic helps our method find a small countermeasure circuit.

Next, it invokes the subroutine GetReg to extract a circuit region *reg*, consisting of logic gates in the fanin and fanout cones of *sGate*. From *reg*, we synthesize a new circuit *newReg*, which is functionally equivalent to *reg* and, at the same time, FSA-resistant. If the synthesis subroutine fails to find *newReg* for *reg*, it will be invoked again for a circuit region *reg* with a smaller number of gates. There may not always exist an FSA-resistant *newReg*, for example, when the mismatch between the maximum and minimum arrival times of the inputs of *reg* exceeds the maximum depth of *newReg* defined by *lev*. In such case, *newReg* is synthesized with the goal of reducing the mismatch between the arrival times, and the residual mismatch will be eliminated in a later iteration. After finding the new region, we replace it with the old region in P. We keep updating P until no more sensitive gates remain in the circuit. At this point, the new circuit P is returned.

5.2 Region Selection

Inside the subroutine GetReg, the sensitive gate *sGate* is added to *reg* first. Then, we expand *reg* by adding the sensitive fanout gates transitively. When no sensitive fanout gate exists, we add the sensitive fanin gates of *sGate* transitively. When there are multiple sensitive fanin gates, we always add the gate with the minimum arrival time first, until *reg* reaches a predefined size limit n. This heuristic ensures that we follow a topological order and therefore avoids the need to re-synthesize countermeasures for the same gate. It also reduces the maximum mismatch in the arrival time by decreasing the circuit's maximum depth.

Given *lev*, which is controlled by the user, each new region would have a maximum size of $2^{lev} - 1$ gates. The maximum size occurs when all region inputs have equal arrival time from the inputs of P. If the region inputs have different arrival time, however, they will be assigned to different levels in the template circuit, which means the total number of gates would be less than $2^{lev} - 1$.

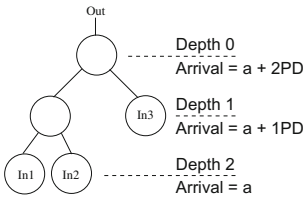

Fig. 8. The size of the new region.

For example, the region in Fig. 8 has $lev = 2$, but since the three inputs have arrival time of a, a and $a + 1$, respectively, the template circuit would have two nodes as opposed to the maximum $(2^2 - 1) = 3$ nodes.

In Algorithm 1, both *GatesSyn* and *lev* are parameters that may be controlled by the user. They are used to identify a sweet spot for the application with respect to several optimization factors. For example, by including more types of gates in *GatesSyn*, the number of solutions to be examined by the SyGuS tool will increase. It may lead to a more compact solution, but may also increase the search time. Similarly, having a larger *lev* will improve the quality of the synthesized circuit, since gate sharing is more likely in a larger circuit than in a smaller circuit. On the other hand, having a larger *lev* will significantly slow down the synthesis procedure.

6 The Synthesis Subroutine

Given a circuit region *reg*, the subroutine SYNTHESIZE searches for a functionally equivalent new circuit *newReg* that is also FSA-resistant. The pseudocode of this subroutine is shown in Algorithm 2, where the input consists of *reg*, the map *MinAr*, the set *GatesSyn* of logic gates (components) to used in creating *newReg*, and *lev*.

Algorithm 2. The synthesis subroutine based on SyGuS.

```
1:  SYNTHESIZE (reg, MinAr, GatesSyn, lev) {
2:    testEx ← ∅
3:    Depth ← GETINPUTDEPTH (reg, lev, MinAr)
4:    while (true) {
5:      newReg ← GENNEWREGION (reg, testEx, Depth, GatesSyn, lev)
6:      if (newReg exists) {
7:          test ← CHECKEQUIVALENCE (reg, newReg)
8:          if (test = ∅)
9:              return newReg;
10:         testEx ← testEx ∪ {test}
11:     }
12:     else
13:         return ∅;
14:   }
15: }
```

The subroutine starts by initializing the set *testEx* to an empty set. This is a set of input values used by SyGuS to generate a partially equivalent candidate circuit. That is, at least for these test input values, *newReg* and *reg* guaranteed to produce the same output. Later, we will invoke the verification subroutine to check if *newReg* and *reg* produce the same output for all possible input values.

Subroutine GETINPUTDEPTH computes the appropriate depth for each of the input signals in order to reduce the discrepancies among their arrival time at the outputs. At Line 4, the subroutine enters a while-loop that contains two main steps. In the first step, it calls SYNNEWREGION to search for *newReg*. In the second step, it calls CHECKEQUIVALENCE to prove the functional equivalence

of *reg* and *newReg*. If they are not equivalent, a counterexample, denoted *test*, will be returned. This new input value will be added to *testEx* before the while-loop enters the next iteration. The larger the set *testEx*, the more likely that the next *newReg* is functionally equivalent to *reg*.

Computing the Input Depth. The subroutine GetInputDepth computes, for each input signal in *reg*, the allowed depth in *newReg* (the *level* as described in Fig. 6). Recall that each input signal in *reg* may have a different arrival time. Therefore, inside *newReg*, they need to be placed at different levels (or have different depths) in order to eliminate the mismatch in the time taken for them to arrive at the output. Consider, for example, the circuit in Fig. 9, which has different delay along different input-to-output paths in *reg* (boxed region). To eliminate the mismatch in *newReg*, node X should be placed one level closer to the output O than nodes A and B. The pseudocode for computing the depths of all input signals (for creating *newReg*) is shown in Algorithm 3.

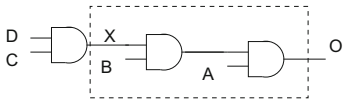

Fig. 9. Example of circuit *reg*.

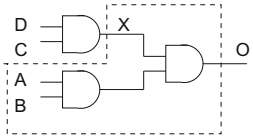

Fig. 10. Example of the *newReg*.

Algorithm 3. Computing the depths of the input nodes in *newReg*.

```
1: getInputDepth (reg, lev, MinAr) {
2:    minMinAr ← minimum of MinAr[in] for all input in
3:    for each (input signal in ∈ reg) {
4:       Δ_Ar ← MinAr[in] − minMinAr
5:       newRegDepth[in] ← max(2, (lev − Δ_Ar))
6:    }
7:    return newRegDepth;
8: }
```

Generating the Candidate Circuit. Subroutine SynNewRegion computes a candidate circuit that behaves the same as *reg* at least for the test cases in *testEx*. It follows our description in Sect. 4, where the template circuit is constructed using the SyGuS specification language. Then, it invokes the solvers in the SyGuS tool [2] to compute a solution. For example, a solution returned by SyGuS for the example in Fig. 9 is shown in the boxed region in Fig. 10. The resulting circuit, denoted *newReg*, is checked by the verification subroutine. If no candidate circuit exists and the subroutine SynNewRegion returns an empty set, Algorithm 1 will invoke it again on a smaller region.

More formally, using the SyGuS specification language, we construct a logical formula Φ for *reg*, whose satisfying assignment directly corresponds to a candidate solution for *newReg*. The logical formula Φ is defined as follows:

$$\Phi = \Phi_{reg} \wedge \Phi_{template} \wedge \Phi_{EqI} \wedge \Phi_{EqO} \wedge \Phi_{testEx},$$

where Φ_{reg} encodes the input-output relation of the original circuit *reg*, $\Phi_{template}$ encodes the input-output relation of the template circuit (as in Fig. 6), Φ_{EqI} asserts that *reg* and the template circuit share the same input values, Φ_{EqO} asserts that *reg* and the template circuit have the same output, and Φ_{testEx} restricts the input values to the examples in *testEx*.

Verifying the Equivalence. For each candidate circuit *newReg*, we also need to verify that it is equivalent to *reg* for all input values (not just the input values in *testEx*). This is a standard equivalence checking problem, for which we construct a logical formula Ψ such that Ψ is satisfiable if and only if *newReg* and *reg* are not equivalent. The new formula Ψ is defined as follows:

$$\Psi = \Psi_{reg} \wedge \Psi_{newReg} \wedge \Psi_{EqI} \wedge \Psi_{UneqO},$$

where Ψ_{reg} encodes the input-output relation of *reg*, Ψ_{newReg} encodes the input-output relation of *newReg*, Ψ_{EqI} asserts that *reg* and *newReg* share the same input values, and Ψ_{UneqO} asserts that *reg* and *newReg* have different output values.

If Ψ is satisfiable, a test case (input value) will be generated to show why two regions are not equivalent. In such case, we add the new test to *testEx* so that the bad solution will not be computed in the future. Then, we invoke SynNewRegion again.

7 Experiments

We have implemented our method using the SyGuS solvers [2] and conducted experiments on a set of circuits that implement various parts of the Advanced Encryption Standard (AES) and MAC-Keccak, which is the SHA-3 crypto-hashing algorithm recently standardized by NIST. Table 1 shows the statistics of these benchmarks, including the name, a brief description, the circuit size, as well as the number of input and output signals (bits). The source code of our synthesis tool as well as the input files and instructions to reproduce our experiments are available for artifact evaluation.

During the experiments, we used the AND, XOR, OR and NOT gates as components in *GatesSyn* for synthesizing new regions. We set the depth *lev* to 3. To compare with state-of-the-art techniques, we implemented the buffer insertion method as described in [19,22]. We also applied the logic optimization algorithms in the ABC tool [12], to check if standard algorithms in EDA tools can be used to generate FSA-resistant circuits (the answer is no). All our experiments were conducted on a computer with a 3.4 GHz Intel i7-2600 processor and 4 GB RAM.

Table 1. Statistics of the set of benchmark circuits used in our experiments.

Name	Circuit description	Nodes	Inputs	Outputs
C1	MAC-Keccak nonlinear masked Chi function 1 [8]	35	10	1
C2	MAC-Keccak nonlinear masked Chi function 2 [8]	35	10	1
C3	Generated MAC-Keccak nonlinear masked Chi function 1 [14]	44	10	1
C4	Generated MAC-Keccak nonlinear masked Chi function 2 [14]	44	10	1
C5	Unmasked MAC-Keccak nonlinear Chi function [8]	6	3	1
C6	AES S-Box design of nonlinear invg4 function [11]	83	4	4
C7	AES S-Box design of nonlinear mul4 function [11]	63	8	4
C8	AES S-Box single round nonlinear functions [11]	209	8	8
C9	Complete AES PPRM1 S-box design [34]	8,054	8	8
C10	Complete AES Boyar-Peralta S-box design [11]	156	8	8

Table 2 shows our experimental results. Columns 1-2 show the benchmark name and the number of nodes in the original circuit. Columns 3-4 show the number of nodes in the new circuit obtained by buffer insertion, and the node increase in percentage. Columns 5-6 show the number of nodes in the new circuit obtained by our method, and the node increase in percentage.

Table 2. Comparing our synthesis method with buffer insertion [19,22].

Name	Nodes	Buffer insertion		New method	
		Nodes	Increase	Nodes	Increase
C1	35	51	45 %	42	20 %
C2	35	48	37 %	40	14 %
C3	44	54	22 %	48	9 %
C4	44	59	34 %	45	2 %
C5	6	9	50 %	9	50 %
C6	83	134	61 %	98	18 %
C7	63	79	25 %	73	15 %
C8	209	292	39 %	244	16 %
C9	8,054	77,717	864 %	8,943	11 %
C10	156	9,585	6044 %	370	137 %

The results in Table 2 demonstrate the effectiveness of our method in synthesizing more compact countermeasures against FSA attacks. Compared to the buffer insertion method, the circuits produced by our method are consistently smaller. For example, our new circuit for C9 has only 11 % more nodes than the

original circuit, whereas the circuit produced by the buffer insertion method has 864 % more nodes.

Table 3 shows the statistics of our iterative synthesis method, where Column 2 is the number of calls that we made to SyGuS to generate the new circuit regions. Among them, Column 3 shows the number of successful calls and Column 4 shows the number of failed calls. The results shows that calls to SyGuS almost always succeed – recall that when the SyGuS solver fails, the size of *reg* has to be reduced before we try again. Column 5 is the total time taken by our synthesis method to generate the final result.

Table 3. Statistics of our new synthesis method.

Name	Synthesis iterations	Successful iterations	Failed iterations	Total time [s]
C1	7	7	0	1.22
C2	5	5	0	0.10
C3	4	4	0	0.09
C4	2	2	0	0.06
C5	4	3	1	0.13
C6	23	23	0	0.48
C7	12	12	0	0.26
C8	47	47	0	1.11
C9	2,627	2,627	0	412.3
C10	219	217	2	13.7

For most benchmark circuits, the time taken by our method to synthesize the countermeasure is negligible. Furthermore, the synthesis time only increases moderately as the countermeasure circuit size increases. Finally, compared to prior techniques such as the buffer insertion method, our new method is more effective in reducing the area cost: as the circuit size increases, the saving also increases.

To confirm that standard EDA algorithms cannot generate FSA-resistant circuits, we also applied the ABC tool [12] to all the benchmark circuits. Specifically, ABC has a command called *balance*, which is designed to balance the delay along input-output paths in a circuit. To preform balancing, ABC starts by converting the circuit into an And-Inverter-Graph (AIG). This results in a circuit containing only AND gates and Inverters. Then ABC heuristically optimizes the new circuit by balancing the number of two-input AND gates between the circuit output and the primary inputs.

Unfortunately, ABC does not distinguish sensitive input signals from insensitive ones. As such, it cannot be used to target only the sensitive input-output paths. For the sake of comparison, we conducted the experiments using a variant of our method (a weakened version) that does not differentiate between the type of the primary inputs either. That is, as in ABC, we pretend that all input

signals of the circuit are sensitive. Table 4 shows the results of our experiments. Here, the focus is on comparing the size of the new circuits and the depth of the circuits (longest path).

Table 4. Comparing our method with the *balance* command of ABC [12].

Name	Depth	ABC		Our new method	
		Node increase	Depth	Node increase	Depth
C1	8	300 %	27	20 %	5
C2	7	300 %	27	31 %	6
C3	7	273 %	25	20 %	6
C4	8	273 %	28	18 %	6
C5	3	233 %	7	50 %	3
C6	9	285 %	31	18 %	7
C7	7	322 %	21	16 %	7
C8	17	308 %	33	17 %	15
C9	156	80 %	586	11 %	17
C10	24	476 %	64	137 %	23

The results in Table 4 show a noticeable difference in the quality of the synthesized circuits. First, in all cases, our new circuits are significantly smaller than those obtained by ABC. Indeed, the node increase percentage by ABC ranges from 80 % to 476 % (the average is 285 %), whereas in our method, it only ranges from 2 % to 137 % (the average is 33 %). In addition, the longest path (Depth), measured by largest number of gate levels between any primary input and the circuit output, is also significantly smaller in the our new circuits.

8 Related Work

As we have mentioned earlier, our method is the first inductive synthesis based method for synthesizing FSA-resistant circuits. Although there is a large body of work on logic synthesis and optimization, traditional EDA algorithms cannot be used to solve this problem. Since our method relies on inductive synthesis, as opposed to matching some known patterns and then applying predefined transformations, it can search through a larger design space and therefore generate solutions that are better than hand-crafted countermeasures. In addition, our solutions are provably secure.

Ghalaty et al. [22] proposed a method for implementing FSA countermeasures based on the addition of delay elements at the inputs of certain gates in the circuit, to equalize the path delays from sensitive inputs. As we have demonstrated through experiments, their method can lead to countermeasures with significantly more logic gates. Furthermore, it does not guarantee to eliminate

the mismatch in the arrival time of the input signals for all gate types; in particular, it ignores the XOR gates. Due to this reason, their countermeasure may still be vulnerable to FSA attacks.

Endo et al. [19] proposed another countermeasure to defend against FSA attacks based on adding a configurable buffer circuitry to delay the propagation of the output signals from the cryptographic module. However, their method is a post-silicon solution, which means it does not seek to modify the implementation of the original circuit as in our case. In general, a post-silicon solution is more expensive to implement, since the delay period needs to be configured after the chip is manufactured. To configure the delay, they first measure the delay needed for securing the manufactured cryptography module and then store the delays in an on-chip memory. As for the experimental evaluation, they implemented the countermeasure only for the benchmark C9, and reported a gate overhead of 10 % to 16 %, which is similar to our solution. However, their countermeasure was designed manually, whereas ours is generated automatically.

There is also a large body of work on verifying and synthesizing countermeasures against other types of side-channel attacks. They include, for example, the verification tools developed by Bayrak et al. [7], the *SC Sniffer* tool developed by Eldib et al. [15–18], the compiler assisted masking tool developed by Moss et al. [35], the code morphing method proposed by Agosta et al. [1], and the tool developed by Eldib et al. [14] for synthesizing masking countermeasures for cryptographic software. However, none of these existing tools can handle fault injection based attacks on cryptographic circuits. Although Barthe et al. [6] developed a method for systematic analysis of the security of cryptographic implementations against fault attacks, their focus was on finding fault attacks against cryptographic implementations, as opposed to synthesizing the countermeasures.

9 Conclusions

We have presented a new method for synthesizing cryptographic circuits to defend against fault sensitivity analysis based attacks. Our method relies on syntax-guided inductive synthesis to search for a new circuit that is functionally equivalent to the original circuit and at the same time FSA-resistant. It has the potential to discover more compact and efficient implementations than existing techniques. We have implemented the method and evaluated it on a set of cryptographic circuits. Our experiments show that the method is both scalable and effective in eliminating FSA vulnerabilities. For future work, we plan to evaluate the countermeasures synthesized by our new method on real hardware to assess its resistance against FSA attacks.

Acknowledgments. This work was primarily supported by the NSF under grant CNS-1128903. Partial support was provided by the ONR under grant N00014-13-1-0527. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. Agosta, G., Barenghi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: ACM/IEEE Design Automation Conference, pp. 77–82 (2012)
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: International Conference on Formal Methods in Computer-Aided Design, pp. 1–8 (2013)
3. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
4. Bagheri, N., Ebrahimpour, R., Ghaedi, N.: New differential fault analysis on PRESENT. EURASIP J. Adv. Sig. Proc. **2013**, 145 (2013)
5. Balasch, J., Gierlichs, B., Verdult, R., Batina, L., Verbauwhede, I.: Power analysis of Atmel CryptoMemory–recovering keys from secure EEPROMs. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 19–34. Springer, Heidelberg (2012)
6. Barthe, G., Dupressoir, F., Fouque, P., Grégoire, B., Zapalowic, J.: Synthesis of fault attacks on cryptographic implementations. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1016–1027 (2014)
7. Bayrak, A., Regazzoni, F., Brisk, P., Standaert, F.-X., Ienne, P.: A first step towards automatic application of power analysis countermeasures. In: ACM/IEEE Design Automation Conference, pp. 230–235 (2011)
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview. URL: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf
9. Biham, E.: Differential cryptanalysis. In: Encyclopedia of Cryptography and Security, 2nd edn., pp. 332–336 (2011)
10. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
11. Boyar, J., Peralta, R.: A small depth-16 circuit for the AES S-Box. In: SEC, pp. 287–298 (2012)
12. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
13. Eldib, H., Wang, C.: An SMT based method for optimizing arithmetic computations in embedded software code. IEEE Trans. CAD Integr. Circ. Syst. **33**(11), 1611–1622 (2014)
14. Eldib, H., Wang, C.: Synthesis of masking countermeasures against side channel attacks. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 114–130. Springer, Heidelberg (2014)
15. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Trans. Softw. Eng. Methodol. **24**(2), 11:1–11:24 (2014)
16. Eldib, H., Wang, C., Schaumont, P.: SMT based verification of software countermeasures against side-channel attacks. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2014)
17. Eldib, H., Wang, C., Taha, M., Schaumont, P.: QMS: evaluating the side-channel resistance of masked software from source code, pp. 209:1–209:6 (2014)

18. Eldib, H., Wang, C., Taha, M., Schaumont, P.: Quantitative masking strength: quantifying the power side-channel resistance of software code. IEEE Trans. CAD Integr. Circ. Syst. **34**, 1558 (2015)
19. Endo, S., Li, Y., Homma, N., Sakiyama, K., Ohta, O., Fujimoto, D., Nagata, M., Katashita, T., Danger, J.-L., Aoki, T.: A silicon-level countermeasure against fault sensitivity analysis and its evaluation. IEEE Trans. Very Large Scale Integr. Syst. pp. 1–10 (2014)
20. Endo, S., Sugawara, T., Homma, N., Aoki, T., Satoh, A.: A configurable on-chip glitchy-clock generator for fault injection experiments. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **95–A**(1), 263–266 (2012)
21. Fuhr, T., Jaulmes, É., Lomné, V., Thillard, A.: Fault attacks on AES with faulty ciphertexts only. In: International Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 108–118 (2013)
22. Ghalaty, N.F., Aysu, A., Schaumont, P.: Analyzing and eliminating the causes of fault sensitivity analysis. In: Design, Automation and Test in Europe, pp. 1–6 (2014)
23. Ghalaty, N.F., Yuce, B., Schaumont, P.: Differential fault intensity analysis on PRESENT and LED block ciphers. In: Mangard, S., Poschmann, A.Y. (eds.) COSADE 2015. LNCS, vol. 9064, pp. 174–188. Springer, Heidelberg (2015)
24. Ghalaty, N.F., Yuce, B., Taha, M.M.I., Schaumont, P.: Differential fault intensity analysis. In:International Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 49–58 (2014)
25. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 62–73 (2011)
26. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In : ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 281–292 (2008)
27. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED block cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 326–341. Springer, Heidelberg (2011)
28. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Kluwer Academic Publishers, Boston (1996)
29. Järvinen, K., Blondeau, C., Page, D., Tunstall, M.: Harnessing biased faults in attacks on ECC-based signature schemes. In: International Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 72–82 (2012)
30. Jeong, K., Lee, C., Lim, J.: Improved differential fault analysis on lightweight block cipher lblock for wireless sensor networks. EURASIP J. Wireless Commun. Netw. **2013**, 151 (2013)
31. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. Commun. ACM **55**(2), 103–111 (2012)
32. Li, Y., Sakiyama, K., Gomisawa, S., Fukunaga, T., Takahashi, J., Ohta, K.: Fault sensitivity analysis. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 320–334. Springer, Heidelberg (2010)
33. Moradi, A., Barenghi, A., Kasper, T., Paar, C.: On the vulnerability of FPGA bitstream encryption against power analysis attacks-extracting keys from Xilinx Virtex-II FPGAs. In: IACR Cryptology (2011)
34. Morioka, S., Satoh, A.: An optimized S-Box circuit architecture for low power AES design. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems-CHES 2002. LNCS, vol. 2523, pp. 172–186. Springer, Heidelberg (2003)

35. Moss, A., Oswald, E., Page, D., Tunstall, M.: Compiler assisted masking. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 58–75. Springer, Heidelberg (2012)
36. Paar, C., Eisenbarth, T., Kasper, M., Kasper, T., Moradi, A.: Keeloq and side-channel analysis-evolution of an attack. In: Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 65–69 (2009)
37. Sakamoto, H., Li, Y., Ohta, K., Sakiyama, K.: Fault sensitivity analysis against elliptic curve cryptosystems. In: Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 11–20 (2011)
38. Sakiyama, K., Li, Y., Gomisawa, S., Hayashi, Y., Iwamoto, M., Homma, N., Aoki, T., Ohta, K.: Practical DFA strategy for AES under limited-access conditions. JIP **22**(2), 142–151 (2014)
39. Santis, F.D., Guillen, O.M., Sakic, E., Sigl, G.: Ciphertext-only fault attacks on PRESENT. In: International Workshop on Lightweight Cryptography for Security and Privacy, pp. 85–108 (2014)
40. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transfer **15**(5–6), 475–495 (2013)
41. Yuce, B., Ghalaty, N.F., Schaumont, P.: TVVF: estimating the vulnerability of hardware cryptosystems against timing violation attacks. In: IEEE International Symposium on Hardware Oriented Security and Trust, pp. 72–77 (2015)
42. Zhao, X., Guo, S., Zhang, F., Wang, T., Shi, Z., Ji, K.: Algebraic differential fault attacks on LED using a single fault injection. IACR Cryptology ePrint Archive 2012, p. 347 (2012)