SOLOMON: An Automated Framework for Detecting Fault Attack Vulnerabilities in Hardware

Milind Srivastava*, Patanjali SLPSK*, Indrani Roy*, Chester Rebeiro*, Aritra Hazra† and Swarup Bhunia‡
*Indian Institute of Technology Madras
†Indian Institute of Technology Kharagpur
‡University of Florida
{milind, slpskp, indrroy, chester}@cse.iitm.ac.in, aritrah@cse.iitkgp.ac.in, swarup@ece.ufl.edu

***Abstract*—Fault attacks are potent physical attacks on crypto-devices. A single fault injected during encryption can reveal the cipher's secret key. In a hardware realization of an encryption algorithm, only a tiny fraction of the gates is exploitable by such an attack. Finding these vulnerable gates has been a manual and tedious task requiring considerable expertise. In this paper, we propose SOLOMON, the first automatic fault attack vulnerability detection framework for hardware designs. Given a cipher implementation, either at RTL or gate-level, SOLOMON uses formal methods to map vulnerable regions in the cipher algorithm to specific locations in the hardware thus enabling targeted countermeasures to be deployed with much lesser overheads. We demonstrate the efficacy of the SOLOMON framework using three ciphers: AES, CLEFIA, and Simon.**

***Index Terms*—fault attack, fault evaluation tools, formal verification**

I. Introduction

Implementations of encryption algorithms are highly vulnerable to a class of physical attacks known as fault injection attacks, where the attacker induces faults by a glitch in the clock or power supply, temperature changes, or optical beams. The faulty output, known as faulty ciphertext, is compared with the fault-free ciphertext to glean bits of the secret key. Such attacks are extremely dangerous and can break cryptographically strong ciphers within a few minutes.

However, not all faults are exploitable. The injected faults need to be precisely placed and timed for a successful attack. Recent research in fault attack protection has focused in two directions. Works such as XFC [1] and ExpFault [2] automatically identify exploitable fault locations in an abstract high level description of the cipher, while others introduce countermeasures in the hardware implementations. However, in all these works, manual identification of exploitable locations in the hardware is required. This task is tedious and requires considerable expertise. It is highly dependent on the cipher algorithm and its implementation.

In this paper, we propose SOLOMON, a framework that fills the gap between high-level fault evaluation and pinpointing fault vulnerable locations in the hardware implementation. Given a high level description of a block cipher and its hardware implementation, SOLOMON can precisely identify exploitable fault locations in the design automatically. SOLOMON can work across different abstractions of the hardware design such as Register Transfer Level (RTL) or gate-level netlist,

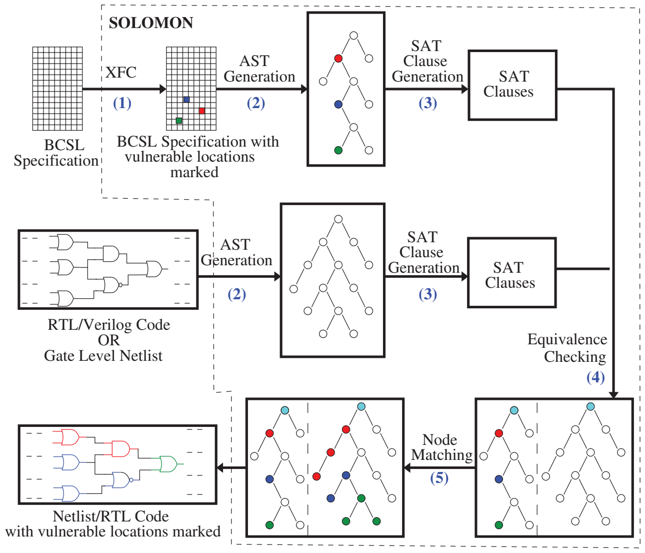

Fig. 1. A high level view of the SOLOMON flow. The regions highlighted in blue, green, and red are the vulnerable locations.

thus enabling hardware designers to deploy targeted countermeasures at the abstraction of their choice.

Figure 1 provides an overview of SOLOMON. It uses a Block Cipher Specification Language (BCSL) [3] to describe the operations of the cipher at a high-level. SOLOMON first invokes XFC [1] which performs a high-level exploitability evaluation on the block cipher specification. The output of XFC is a similar BCSL specification with the exploitable locations marked. SOLOMON then uses a Satisfiability Modulo Theory (SMT) based equivalence checking algorithm to map these exploitable locations from the specification onto the hardware design. To perform this equivalence checking, SOLOMON first represents the specification and the hardware implementation as Abstract Syntax Trees (AST). ASTs capture the syntax and structure of the specification and implementation. It then extracts SAT clauses from these ASTs and feeds them to the equivalence checking algorithm. The algorithm outputs precise locations in the implementation which correspond to the exploitable locations marked in the specification.

The contributions of this paper are as follows.

- SOLOMON is the first work to precisely identify fault attack vulnerabilities in a hardware design of a block

Fig. 2. Specification of the `AddRoundKey` operation in AES where inp_s, out_s and k_s represent the 128-bit input, output and key respectively

```
1  ⟨ outs ⟩ ⟨ linear ⟩ ⟨ ADDKEY ⟩⟨
2          ⟨ outs [1]: {ins[1]}: F_XOR(ins[1], F_LKUP(1, ks))} ⟩
3          ⟨ outs [2]: {ins[2]}: F_XOR(ins[2], F_LKUP(2, ks))} ⟩
                ⋮
17         ⟨ outs [16]: {ins[16]}: F_XOR(ins[16], F_LKUP(16, ks))} ⟩
18 /⟩
```

cipher. Prior works, either performed this analysis at a high-level [1], [2] or for software implementations of block ciphers [4].

- SOLOMON can work across different abstractions of hardware design such as RTL and gate-level netlist.
- We evaluate SOLOMON on 3 block ciphers namely: AES, CLEFIA [5] and Simon [6]. We consider Verilog implementations and synthesized netlists of these ciphers.
- We evaluate SOLOMON on 2 different implementations of AES: one with S-Box implemented as a lookup table and the other with composite fields. Although functionally equivalent, each implementation has a different fault vulnerability footprint.

The organization of the paper is as follows: Section II presents the related work. Section III briefly describes the Block Cipher Specification language. Section IV gives an overview of the SOLOMON flow. Section V talks about the experimental setup, evaluation and results. Section VI concludes the paper.

II. Related Work

Automatic evaluation of fault attack vulnerabilities is a pressing requirement to help implement efficient countermeasures. Recent works in this space have focused on two directions. The first approach, such as [1], [2] automatically identifies operations in the cipher algorithm that are vulnerable to fault attacks, but does not evaluate fault vulnerability at the actual implementation level. The second approach, such as [4], identifies fault vulnerable locations directly from assembly implementations of the cipher by converting code to a data flow graph and analyzing dependencies between instructions. SOLOMON is the first work that evaluates fault attack vulnerability in hardware implementations of block ciphers.

III. Block Cipher Specification

The Block Cipher Specification Language (BCSL) was proposed in [3] as a means to define an abstract specification of a block cipher[1]. BCSL captures information flow from the plaintext to the ciphertext at a high level and is agnostic of the target platform. Figure 2 shows an example of a BCSL specification for the `AddRoundKey` operation in the AES cipher. Line 2 of the specification conveys that the 1^{st} output byte is dependent on the first input byte. It also describes that the output byte is computed by doing an `XOR` of the input byte and the result of applying the `F_LKUP` function on the `KEY` with index 1.

[1]Examples of block ciphers encoded in BCSL can be found at https://bitbucket.org/casl/faultanalysis/src/master/SAFARI/Specifications

Listing 1. Verilog implementation of the AddRoundKey operation in AES

```
module AddRoundKey(in_i, k_i, out_i);
  input [127:0] in_i, k_i;
  output [127:0] out_i;

  assign out_i[127:64] = ((~in_i[127:64]&k_i[127:64])
         |(in_i[127:64]&(~k_i[127:64]));
  assign out_i[63:0] = ((~in_i[63:0]&k_i[63:0])|(in_i
         [63:0]&(~k_i[63:0]));
endmodule
```

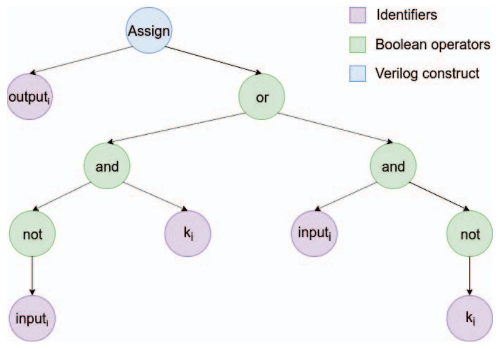

Fig. 3. AST representing the `AddRoundKey` operation of AES

IV. An Overview of SOLOMON

SOLOMON (Figure 1) takes 2 inputs – the specification of the cipher in BCSL and the cipher implementation which can be in RTL such as Verilog or a gate-level netlist. It identifies regions in the implementation that correspond to the annotated regions in the specification. This is done in five steps which we describe below.

1) **Exploitable fault characterization:** SOLOMON leverages previous work, called XFC [1], to characterize exploitable faults at a high level. XFC uses a coloring scheme to formally capture differential fault attack vulnerabilities in block ciphers. The output of XFC is a BCSL specification marked with the operations in the cipher which are vulnerable to fault attacks.
2) **Abstract Syntax Tree generation:** Mapping lines of specification to the implementation is not trivial since the same operation may be described differently in each file. For example, the specification may describe an operation at a byte level while in the implementation the same operation may be done using quad-words (128 bit). Hence to compare the two, SOLOMON automatically generates Abstract Syntax Trees (ASTs) from the specification and implementation. ASTs capture the structural details in each input. Each node in the AST represents a construct occurring in the input. We denote the specification and implementation ASTs as AST_{spec} and AST_{imple} respectively. Listing 1 shows a Verilog module which performs the `AddRoundKey` operation of AES. Figure 3 shows a part of the AST that represents the same functionality (*i.e.* $out_i = in_i \oplus k_i$, where in_i, out_i and k_i are of 128 bits). The corresponding specification is described in Figure 2.
3) **SAT clause generation:** At the end of the previous step, there are two ASTs – one for the cipher specification and another for the implementation. For each AST, SOLOMON

Algorithm 1: Generating SAT clauses

```
   Input: AST $A$
   Output: map of variable names to SAT clauses $E$
1  $E$ = empty map
   /* dependency resolution phase */
2  $G_f$ = empty graph // dependency resolving graph for functions
3  for each function definition $F$ in $A$ do
4      for each function call $C$ in $F$ do
5          add directed edge $(C, F)$ to $G_f$
6      end
7  end
8  $G_a$ = empty graph // dependency resolving graph for assign statements and module instantiations
9  for each assign statement or module instantiation $S$ in $A$ do
10     $L$ = set of identifiers on LHS of $S$
11     $R$ = set of identifiers on RHS of $S$
12     for each $l$ in $L$ do
13         for each $r$ in $R$ do
14             add directed edge $(r, l)$ to $G_a$
15         end
16     end
17 end
   /* expression generation phase */
18 $T_f$ = topologicalSort($G_f$)
19 for each function definition $F$ in $T_f$ do
20     for each blocking substitution in $F$ do
21         generate SAT clause for RHS
22         update entry corresponding to LHS in $E$
23     end
24 end
25 $T_a$ = topologicalSort($G_a$)
26 for each node $N$ in $T_a$ do
27     for each incoming edge $I$ do
28         if $I$ is assign statement then
29             generate SAT clause for RHS
30             update entry corresponding to LHS in $E$
31         end
32         else
               /* Module instantiation */
33             generate SAT clause for instantiated module // recursive call
34             convert module instantiation to an equivalent assign statement
35             generate SAT clause for RHS
36             update entry corresponding to LHS in $E$
37         end
38     end
39 end
```

automatically generates SAT clauses by parsing the AST in a bottom to top manner and incrementally generating clauses for each node.

We assume that each encryption round is represented as a separate module in both the specification and implementation. We represent these modules as $round_{spec}$ and $round_{imple}$ respectively. The SAT clause generation works on a round-by-round basis. Algorithm 1 generates SAT clauses for one round. This algorithm works in two phases – a dependency resolution phase and an expression generation phase.

Dependency Resolution Phase: Lines 1-17 in Algorithm 1 correspond to the dependency resolution phase. The ASTs can have inter-dependencies such as nested `function` calls, `module` instantiations and blocking `assign` statements. To resolve these dependencies, we maintain two data structures: a `function` graph G_f and an `assign` graph G_a. Lines 3-7 describe the process for resolving function dependencies. We parse the round modules looking for `function` definitions. For each `function` definition F, we parse its AST to check for `function` calls within F. For each such `function` call, we update the dependency graph with the appropriate edge. We adopt a similar approach for resolving dependencies between blocking `assign` statements which is described in Lines 9-17. Dependencies for `module` instantiations can be resolved by parsing the module instantiation AST to find input and output pins and adding appropriate edges to G_a Consider Listing 1 as an example. We create nodes in the G_a graph according to the declarations in Lines 2 and 3. Based on the `assign` statements in Lines 5 and 6, we add edges from in_i to out_i and from k_i to out_i. At the end of the dependency resolution phase, we have two graphs G_a and G_f which contain the dependencies between components of the Verilog code.

Expression Generation Phase: Lines 18-39 correspond to the expression generation phase. We perform a topological sort on the graphs G_a and G_f to obtain the nodes of the graphs in the order of their dependencies. We then traverse these graphs in this order and incrementally generate an expression for each node N present in the graph. At the end of Algorithm 1 we have an expression map E that maps variable names to corresponding SAT clauses. We get 2 such expression maps, E_{spec} and E_{imple} for the specification and implementation, respectively.

Consider Listing 1 as an example. After parsing the two `assign` statements on Lines 5 and 6, we obtain the dependency graph G_a where out_i depends on in_i and k_i. We initialize $E[out_i]$ (the SAT clause corresponding to out_i) to 0^{128} – a 128 bit vector of zeros. On parsing Line 5 and Line 6, we generate the expressions for the RHS and update the map entry $E[out_i]$ which now becomes,

$$((\neg in_i[127:64] \& k_i[127:64]) \mid (\neg k_i[127:64] \& in_i[127:64])) \mid\mid 0^{64},$$

$$\text{and } ((\neg in_i[127:64] \& k_i[127:64]) | (\neg k_i[127:64] \& in_i[127:64]))$$
$$\mid\mid ((\neg in_i[63:0] \& k_i[63:0]) | (\neg k_i[63:0] \& in_i[63:0])),$$

respectively, where $||$ is the concatenation operation.

Graphs G_a and G_f are Directed Acyclic Graphs (DAGs) as they represent the dependency between Verilog constructs. Thus, the topological sorts of these graphs converge in finite number of steps. The `for` loops in the Algorithm 1 run for finite number of iterations. The time complexity of algorithm 1 depends linearly on the number of `function` definitions, `assign` statements, module instantiations and wires. At the end of SAT clause extraction, we have expressions for each variable in $round_{spec}$ and $round_{imple}$. Next, we check the outputs of these modules for equivalence.

4) **Equivalence checking:** To compare SAT clauses for equivalence, SOLOMON generates appropriate constraints and feeds them to an SMT solver. Two constraints are used to represent the functionality of the specification and implementation. Another constraint enforces the equality of the inputs being fed into the first two SAT clauses. One more constraint enforces that the specification and implementation outputs must differ. If the SMT solver gives an UNSAT (unsatisfiable) for these constraints, then

Algorithm 2: Node matching

```
   Input: Expressions maps: $E_{spec}$, $E_{imple}$, topologically sorted nodes of
          assign graph ($G_a$): $nodes_{spec}$, $nodes_{imple}$
   Output: Maximal set of pairs of nodes $(S, I)$ where $S \in AST_{spec}$ and
          $I \in AST_{imple}$ and $E_{spec}[S] == E_{imple}[I]$
 1 node_pairs = empty list
 2 for $node_{spec}$ in $nodes_{spec}$ do
 3     for $node_{imple}$ in $nodes_{imple}$ do
 4         $expr_{spec}$ = $E_{spec}[node_{spec}]$
 5         $expr_{imple}$ = $E_{imple}[node_{imple}$
 6         if $expr_{spec} == expr_{imple}$ then
 7             append $(node_{spec}, node_{imple})$ to node_pairs
 8         end
 9     end
10 end
11 return node_pairs
```

it implies that the two SAT clauses are equivalent i.e. AST_{spec} and AST_{imple} represent the same functionality. If the outputs of $round_{spec}$ and $round_{imple}$ match, we next attempt to match intermediate variables defined in the round modules.

5) **Node matching:** We now attempt to obtain a finer mapping by matching subgraphs in AST_{spec} and AST_{imple} which correspond to intermediate variables in the specification and implementation expressions. As described in Algorithm 2, we iterate over the AST_{spec} and AST_{imple} nodes in topological order We then compare nodes of the ASTs using the SMT solver ($==$ in Line 6 represents this operation). We store all the pairs of matching nodes (S, I) where nodes S and I belong to AST_{spec} and AST_{imple} respectively. Using these pairs of nodes we can identify specific regions in the cipher implementation which correspond to the vulnerable locations in the specification. The time complexity of Algorithm 2 is linearly dependent on the number of nodes in AST_{spec} and AST_{imple}.

V. Experimental Setup, Evaluation and Results

Experimental Setup: SOLOMON is implemented in Python 3.6.7. ASTs are created using the pyverilog [7] package. For equivalence checking, the Z3 Theorem Prover [8] is used. We evaluate SOLOMON on 3 cipher implementations: AES, CLEFIA[2] [5] and an inhouse developed Simon [6] implementation in Verilog. Two AES implementations were used – AES_C[3] (with composite field S-Boxes) and AES_L (with lookup table S-Boxes). To obtain gate level netlists, we synthesize the Verilog codes using Synopsys DC compiler M-2016.12-SP5-4.

Evaluation and Results: For evaluation, we use an Ubuntu 16.04 machine with Intel Core i7-3770 CPU @ 3.4 GHz. For each cipher, we consider a fault location annotated by XFC in the BCSL specification. Using SOLOMON, we map that fault to wires/registers in Verilog code and gates in the netlist. Table I shows how the footprint of a fault increases from the specification to the Verilog code and then to gate-level netlist. Simon is a curious exception to this observation because its Verilog implementation is highly optimized, resulting in much lesser lines of code as well as the fact that many of the faults map onto wires, rather than gates in the netlist. We also compare the two different implementations of AES - AES_C and AES_L as shown in Table I. The composite field implementation is more optimized than the lookup implementation in terms of area and hence consumes lesser lines of code as well as gates. The table also shows the execution time of the 5 steps of SOLOMON (refer Figure 1). Steps (4) and (5), which involve the SMT solver, are the slowest. The overall execution time is under 10 minutes.

TABLE I
RESULTS OF SOLOMON ON DIFFERENT CIPHER IMPLEMENTATIONS

		$\mathbf{AES_C}$	$\mathbf{AES_L}$	**CLEFIA**	**Simon**
No. of AST nodes	$spec$	50379		28286	3961
	$imple$	2046	46041	3832	141
Execution time of each step (in sec)	**(1)**	0.02	0.02	0.02	0.06
	(2) $spec$	1.92		1.08	0.17
	(2) $imple$	0.12	1.75	0.17	0.03
	(3) $spec$	5.72		3.31	0.44
	(3) $imple$	0.23	5.54	0.56	0.36
	(4)	437.06	2.64	46.40	0.02
	(5)	13.8	0.92	136.7	0.24
	Total	458.87	18.51	188.24	1.32
Fault Vulner--ability analysis	**Fault location**	7 MixColumns to 8 ShiftRows		13 DXor to 14 SubByte	30 Rot_2 to 30 Concat
	Verilog lines	365	4173	609	24
	Gates	4590 (11.56%)	10477 (9.85%)	5324 (5.53%)	52 (2.83%)

VI. Conclusion and Future Work

Given a hardware implementation of a block cipher in RTL or netlist, SOLOMON uses formal techniques to automatically identify fault attack vulnerable components in the design within a few minutes. We demonstrate the efficacy of SOLOMON on AES, CLEFIA and Simon and investigate the vulnerabilities of two different implementations of AES.

SOLOMON suffers from two limitations. First, it does not handle sequential logic, such as `always` blocks in Verilog. This can be handled by introducing time as a variable in the SAT clauses and accounting for values of variables at different timesteps. Second, since SOLOMON uses the output from XFC, limitations of XFC [1] also manifest in SOLOMON. However SOLOMON can potentially use other compatible high-level fault evaluation tools, such as ExpFault [2]. These limitations would be addressed in the future work.

References

[1] P. Khanna, C. Rebeiro, and A. Hazra, "XFC: a framework for exploitable fault characterization in block ciphers," in *DAC*. IEEE, 2017, pp. 1–6.

[2] S. Saha, D. Mukhopadhyay, and P. Dasgupta, "ExpFault: an automated framework for exploitable fault characterization in block ciphers," *IACR TCHES*, pp. 242–276, 2018.

[3] I. Roy, C. Rebeiro, A. Hazra, and S. Bhunia, "SAFARI: Automatic synthesis of fault-attack resistant block cipher implementations," *IEEE TCAD*, 2019.

[4] X. Hou, J. Breier, F. Zhang, and Y. Liu, "Fully automated differential fault analysis on software implementations of block ciphers," *IACR TCHES*, pp. 1–29, 2019.

[5] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit blockcipher CLEFIA," in *IACR FSE*. Springer, 2007, pp. 181–195.

[6] R. Beaulieu *et al.*, "The SIMON and SPECK lightweight block ciphers," in *DAC*. IEEE, 2015, pp. 1–6.

[7] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog HDL," in *ARC*, 2015.

[8] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340.

[2]https://www.sony.net/Products/cryptography/clefia/download/index.html
[3]http://www.risec.aist.go.jp/project/sasebo/