Efficient Finite State Machine Encoding for Defending Against Laser Fault Injection Attacks

Aruna Jayasena, Khushboo Rani and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, USA

***Abstract*—Finite State Machines (FSMs) are widely used to implement complex computation sequences and communication protocols. An FSM may consist of different states with different privilege levels, such as protected and non-protected. Ideally, switching from a non-protected state to a protected state should involve an authorization transition. However, with Laser-based Fault injection (LFI), an attacker can bypass authorization by flipping bits in the FSM's state vector. In order to mitigate LFI vulnerability, one can encode the FSM states with the objective of maintaining a large Hamming Distance (HD) between each pair of states. The existing FSM encoding algorithms are either very slow, rely on the user's mathematical ability to manually generate certain state encodings, or lead to unacceptable area overhead. In this paper, we propose an automated framework for generating FSM encodings to defend against LFI attacks. The proposed technique is a fast linear code-based heuristic to produce area-efficient results. We also propose an application-specific simplification to further reduce the area overhead. Experimental results demonstrate that our proposed method is several orders-of-magnitude faster than the state-of-the-art approaches. Our approach also significantly reduces the state code length (50% on average) compared to state-of-the-art approaches.**

***Index Terms*—Hardware Security, Laser Fault Injection, Fault Tolerance, Coding Theory**

I. Introduction

Modern electronic systems utilize System-on-Chip (SoC) designs to provide the computing backbone to run diverse applications. Finite State Machines (FSM) are widely used to implement various computation sequences as well as communication protocols while designing different components of an SoC. A simple FSM can be implemented using a state register and combinational logic, as shown in Figure 1. The state register stores information about the current state. The combinational logic performs computation based on the current state and updates the state register (next state). The number of states and state security categories are determined by the specific application. The number of flip-flops used to represent the state is determined by the bit-length of the FSM state encoding, and the entire unit is called as *State Register*. In Figure 1, the state register is implemented using four flip flops to store the current state using the binary vector 0010.

A successful attack on the FSM can compromise the functionality of the system revealing secret information [1], [2]. Since FSM controls various computations inside as well across SoC components, a compromised FSM controller can lead to catastrophic consequences in safety critical systems. It is vital to secure FSMs to design trustworthy systems.

This work was partially supported by the DARPA/Synopsys AISS grant.

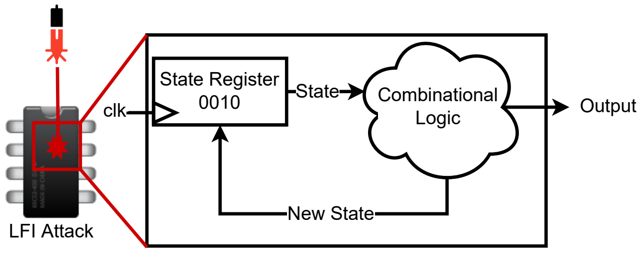


Figure 1: FSM hardware implementation with a state register and associated combinational logic.

A. Threat Model

Hardware-based attacks such as fault injection [3]–[8], side-channel [9], [10], and malicious implants [11], [12] can compromise the security and integrity of an SoC [13]. One of the fault injection attacks, Laser Fault Injection (LFI [14]–[16]) focuses on memory components such as flip-flops and SRAM cells to flip the values. As shown in Figure 1, these memory elements are essential for FSMs. An attacker can use LFI to flip bits of the state register to bypass the authorization state and gain access to a protected state. In this paper, we consider the same threat model as in the existing literature [17]. The attacker is capable of flipping a given number of state bits, and the designer can predict the maximum Bit Flipping Capability (BFC) of the attacker. Attackers have prior knowledge of the FSM state layout, so they know which bits to flip to transition from an accessible non-protected state to a desirable protected state. Attackers also have a precise laser to flip individual bits regardless of location (i.e., the attacker is not forced to flip adjacent bits). Furthermore, flipping from a logical 0 to a logical 1 is just as easy as flipping from a logical 1 to a logical 0. If the Hamming Distance (HD) between the accessible non-protected state and the desired protected state is greater than the BFC, the attacker fails to mount the attack.

B. An Illustrative Example

Figure 2 illustrates a Point-of-Sale (PoS) system FSM. The states in the FSM are categorized into protected (S_3 to S_9) and non-protected states (S_0 and S_1). The data available in the protected state is private to the user, whereas the information in the non-protected state is available to all the users. To protect the sensitive information, transitions from any non-protected state to a protected state are restricted through an *authorization* state (S_2). The PoS system starts with the default state *Landing Page* (S_0) and requires the user's input to navigate

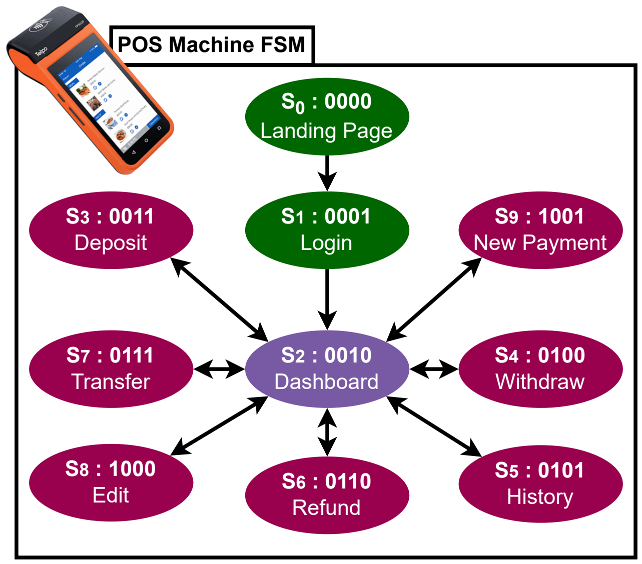


Figure 2: An example FSM for Point-of-Sale (POS) system. The state number corresponds to the binary encoding of each state. Green states represent *non-protected* states, purple represents *authorization* state, and red shows *protected* states.

through the *Login* state (S_1). A user enters a valid username-password pair before transitioning into a secure dashboard with potential point-of-sale options. The FSM guarantees a secure PoS system by restricting any unauthorized transition from a non-protected state to a protected state. However, a single bit flip in the binary state vector can break the PoS FSM security. For example, users are expected to always enter state 0010 through the (Username, Password) transition. However, an attacker who accesses the *Landing Page* can flip the second state bit (0000 to 0010) to enter the *Dashboard* page.

C. Limitations of Existing Methods

When implementing FSM in hardware, commercial hardware synthesis tools do not take into consideration of the current laser fault injection attacks. There are several attempts to mitigate the LFI attacks in conventional FSMs. For example, linear coding techniques [18] incur a high area overhead due to the need for complex error-correcting circuitry. Zwanzger [19] utilizes probabilistic prediction of the best possible code extension to reduce the area overhead compared to linear codes. A security-aware encoding proposed in [2] uses a one-hot-based encoding where HD bits are concatenated for every new state. This can lead to unacceptable overhead. PATRON [17] and SPARSE [20] present promising directions for FSM vulnerability analysis as well as layout-aware LFI-resistant encoding techniques. These approaches assume that the user will provide suitable encoding or use of existing database optimal codes that they can check using binary decision diagrams and provide recommendations. None of these approaches present automated generation of state encoding for given FSMs to defend against LFI attacks. Moreover, these iterative refinement procedures do not guarantee fast encoding generation time or the minimality of the encoding length.

D. Research Contributions

In this paper, we propose an automated framework to generate state encoding to mitigate LFI vulnerability. The basic idea is to generate encoding, which ensures hamming distances among the states more than the bit flipping capability of the attack. If a bit flipping attack is attempted to change the binary state vector of the state register, it is guaranteed to fail since there won't be any matching state in the FSM with the new state vector. This will stop an attacker from jumping to a protected state and accessing sensitive data. Specifically, this paper makes the following major contributions.

- We propose a greedy encoding scheme, AREST, which outperforms state-of-the-art FSM encoding techniques in terms of encoding generation time as well as encoding length (area overhead) to defend against LFI attacks.
- We propose an efficient heuristic that specializes in the state encoding problem based on application diversity to further reduce the area overhead.
- Extensive experimental evaluation demonstrates that AREST provides several orders-of-magnitude speedups in encoding generation time as well as a significant reduction in encoding length (50% on average) compared to state-of-the-art approaches [19].

The remainder of the paper is organized as follows. Section II provides relevant background and surveys related efforts. Section III describes our proposed contributions. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

We first provide relevant background on linear coding and a database of optimal codes since they have been used in the literature for encoding FSM states. Next, we survey related efforts to highlight the novelty of our proposed work.

A. Linear Coding

Linear codes are sets of codewords where each codeword is a linear combination of other codewords. For example, in the case of FSM encoding, it represents binary vectors. Linear codes can be represented as generator matrices where rows are codewords that form a basis for the given linear code. Linear code construction has been studied extensively in information theory to produce error-correcting codes. Similar to FSM encoding, the codewords in error-correcting codes must be at least some HD away from all other codewords to detect errors. There are several existing heuristics that efficiently generate generalized linear codes [19], [21]. Exhaustive search [22] traverses through all possible scenarios and selects the codewords that satisfy the HD constraint. The exhaustive search does not scale for FSMs with a large number of states.

B. Databases of Optimal Codes

There are tables showing maximum code sizes for certain HDs and encoding lengths [23]. These tables can be extended to include known optimal codes. Designers can search through databases or files containing these codes to find appropriate

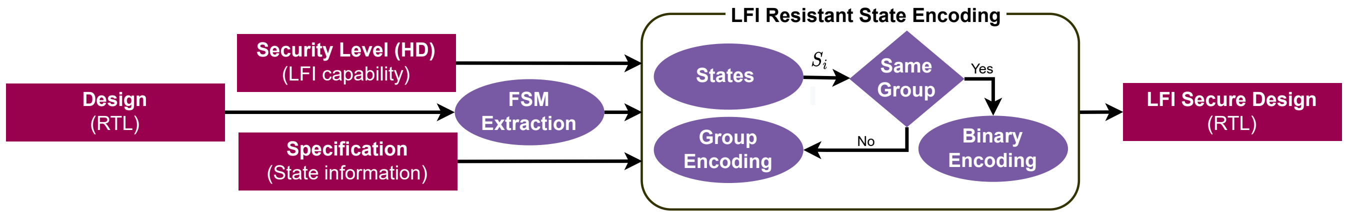


Figure 3: An overview of our proposed framework to automatically generate LFI-resilient FSM encoding for RTL models.

encodings for their FSMs. However, This is an impractical solution for the following reasons. The encoding selection time can be high due to searching in a very large (online) database of all possible combinations (see the exhaustive search times in Figure 6). Most importantly, the database may be corrupted or hacked. In fact, it may be easier for the attacker to hack an online database than to perform an LFI attack.

C. Related Work

There are many promising efforts for designing secure FSMs. Nahiyan et al. [2] propose an encoding scheme for fault-tolerant FSMs which encodes protected states with one-hot encoding and non-protected states with binary encoding. While it automates all but the initial state encoding, the one-hot encoding scheme assumes that an LFI attacker can only flip one bit. Extending the one-hot encoding scheme to HD-hot by concatenating HD bits for every new state is unsuitable because it can lead to unacceptable area overhead. While robust nonlinear codes [18] also incur area overhead in the form of error-correcting circuitry, this overhead improves security. Robust linear codes have limitations on the types of possible codes (Punctured Cubic Code and Quadratic Sum Code), which can be used in encoding generation.

PATRON [17] also categorizes states as protected and non-protected but requires pre-encoded state values such that $HD > BFC$. They suggest using Naïve way or a database of optimal codes. Then, they proceed to generate non-secure encodings which are HD apart from all protected states but not each other. This significantly reduces area overhead. However, the applicability of PATRON is also limited because it does not automate protected state encoding or consider the required code size when generating non-protected states (the number of generated states appears to depend on the vector length of the input protected states). PATRON also assumes that the protected states should remain the minimum HD apart when this may not be the case due to design mistakes. SPARSE [20] extends [17] to improve the area overhead using integer linear programming. Similar to PATRON [17], it also assumes the availability of pre-encoded state values such that $HD > BFC$. As a result, it inherits all the inherent disadvantages of PATRON [17].

Zwanzger [19] extends linear codes with an enforced minimum distance by probabilistically predicting the best possible code extensions. While the author does not explicitly identify this as an FSM encoding scheme, the produced generator matrix's rows can be used as encodings. The minimum-length encodings can be found by iterating through potential code lengths and seeing which lengths generate results, and the minimum HD can be directly given as the enforced minimum distance. We consider this to be the state-of-the-art for comparison with our proposed algorithm for the following three reasons. (1) Unlike the scheme in [18], it can be manipulated to give minimized area. (2) Unlike the scheme in [2], it can be extended to multiple HD values. (3) Unlike the scheme in [17], [20], it does not require all protected state encodings to be given as input. While this method is promising, it also faces practical limitations due to its probabilistic nature. We experimentally found that certain starting inputs yield no results, sub-optimal results, or take an impractically long time to finish, while others take significantly shorter times and yield better results. In other words, this approach is not reliable for generating efficient state encodings.

III. AREST: LFI ATTACK RESISTANT FSM ENCODING

Figure 3 shows an overview of our proposed approach, AREST. AREST consists of three major tasks: (i) FSM state extraction, (ii) FSM state encoding, and (iii) grouping of states. The first task needs to extract an FSM from Register-Transfer Level (RTL) models to determine the number of states and the security requirements of each state. The second task performs a greedy encoding of the FSM states. The third task will generate potential state encodings while identifying security simplifications to reduce area overhead. The input to an LFI-secure FSM generation framework would be an RTL model with labeled protected states. The output would be LFI-attack resistant RTL design. The remainder of this section describes these tasks in detail.

A. FSM State Extraction

FSM state extraction involves exploiting coding guidelines required by synthesis tool vendors (shown in many released synthesis tool guides [24]). These guidelines generally recommend defining a state signal (usually a binary vector) and switching between FSM states using a case statement or if-else block with the encodings of each state as the condition of each statement. The state encodings can be changed by modifying the conditional values in the FSM case statement. If an FSM does not follow vendor guidelines, we do not consider it as vulnerable because it is likely unsynthesizable and cannot be deployed in the field. We utilize the Yosys synthesis tool that allows automated FSM extraction [25]. These synthesis tools also provide encoding options such as one-hot or binary encoding. However, they do not take individual states' security requirements into account when

Algorithm 1 GenerateEncodings

```
$n$: Number of states
$b$: Bit flipping capability
$C$: Generated encoding set
$codeLength$: Intermediate binary state vector length
$A$: Set of all binary encoding for $codeLength$

 1: $C \leftarrow 0$
 2: if $n > 1$ then
 3:     $codeLength \leftarrow max(b + 1, \lceil \log(n) \rceil)$
 4:     while $n > |C|$ do
 5:         $A \leftarrow$ all binary encoding for $codeLength$
 6:                                                  ▷ $|A| = 2^{codeLength}$
 7:         for $\forall a \in A$ do
 8:             for $\forall c \in C$ do
 9:                 if $HammingDistance(a, c) \le b$ then
10:                     Reject a
11:                 end if
12:             end for
13:             if a is not rejected then
14:                 $C \leftarrow C \cup a$
15:             end if
16:             if $|C| \ge n$ then        ▷ Enough encodings found.
17:                 Return $C$
18:             end if
19:         end for
20:         $codeLength \leftarrow codeLength + 1$
21:     end while
22: end if
```

encoding states. Our proposed solution can be incorporated into these frameworks if a protected state labeling system is added.

While our proposed approach makes FSM secure, it may increase the size of the encoding. As a result, it implicitly increases the number of undefined states, which in turn can create security vulnerabilities. A simple and straightforward way to address this problem is to ask the designer to always use a *default* case statement. In fact, it is a common coding guideline in the industry to add a *default* scenario since designers typically do not enumerate all possible states.

B. Automated Encoding of FSM States

Our proposed method for generating FSM encodings (AREST) is presented in Algorithm 1. We always start with 0 as the initial encoding because 0 requires the minimum number of flip-flops to represent. We then iterate through possible encoding lengths, search the space within that length, and check each new value against all previous encodings. If the value does not violate the distance requirement for any encoding in the set, we add the new value to the set.

Definition 1: Hamming bound [26]
Let $n_{(l,d)}$ be the maximum number of binary encodings of codelength l that is separated by the Hamming distance of at least b. Then,

$$n_{(l,b)} \le \frac{2^l}{\sum_{k=0}^{t} \binom{l}{k}} \tag{1}$$

where the upperbound t is,

$$t = \lfloor \frac{b-1}{2} \rfloor \tag{2}$$

Definition 2: Perfect Code
An encoding that attains the Hamming bound is known as perfect code.

Lemma 1: When we increase the $codeLength$ by 1, we can add at least one more encoding to the set.

Proof: From the Hamming bound we can prove that as long as $b < l - 2$, incrementing $codeLength$ l by 1 will always increase the Hamming bound with a value greater than 1 ($\Delta n_{(l,b)} > 1$). Since we start the $codeLength$ l from $max(b + 1, \lceil \log(n) \rceil)$, if we have already attained the Hamming bound (which means a perfect code) and still want to add another state to the encoding set, we should find the encoding with $codeLength + 1$ ($l + 1$).

Proof of Correctness for Algorithm 1: Suppose we have to find the best state encoding set for the n number of states with a BFC of b.
Let $A_2(n, b) = \{(0,1)^c_1, ..., (0,1)^c_n\}$ be the solution given by the AREST algorithm and let $O = \{(0,1)^q_1, ..., (0,1)^q_n\}$ be the optimal solution. Here $(0,1)^i_j$ represent the j^{th} state encoding with the code length of i. We have to prove that $c \le q$.

- For the base case when $n = 2$,
  $A_2(2, b) = \{(0,1)^b_1, (0,1)^b_2\}$ and $q = b + 1$.
  Therefore, $c = q = b + 1$.
- For $p > 2$, assume that the above statement is true for $n = p - 1$ where $c \le q$ for $n = p - 1$ such that $A_2(p-1, b) = \{(0,1)^c_1, ..., (0,1)^c_{p-1}\}$.
- When $n = p$, we have to add another state to the solution from $n = p - 1$.
  - Case 1: If we can find another encoding that satisfies the Hamming distance b with other encodings in same $codeLength$, which means $A_2(p, b) = \{(0,1)^c_1, ..., (0,1)^c_p\}$.
    Therefore , $c \le q$ for $n = p$.
  - Case 2: When $codeLength$ is not enough to add another encoding which satisfies the Hamming distance b with other encodings, which means $A_2(p, b) = \{(0,1)^c_1, ..., (0,1)^{c+1}_p\}$. Since for $n = p - 1$ we had q as the optimum value and $c \le q$, we cannot add one more encoding to the the $codeLength = q$ without violating the Hamming distance b due to it being a perfect code. Therefore, by using **Lemma 1**, we will find the encoding set with $codeLength + 1$.
    Therefore, $c + 1 \le q + 1$ for $n = p$

Example 1: If a designer wants to encode the FSM described in Figure 2 using Algorithm 1, they will take the number of states (assume 10 states) and the minimum required HD (assume a BFC of 3 bits) as inputs. The encodings would be produced as follows. The first encoding would be 0 by default. The algorithm would then count through all numbers, which can be represented using 4 bits (1 through $2^4 - 1$). In each iteration, the algorithm would check whether the current value fulfills the minimum HD requirement for each encoding in the set (containing only 0 in the first iteration). If the requirement is fulfilled for all encodings, then the new value would be added to the set of encodings. Once the size of the encoding set is equal to the required number of encodings, the algorithm returns the set. If no set of 4-bit encodings could be found, then the algorithm moves on to trying 5-bit encodings. In this case, the final result would be the following set where each pattern represents the encoding of one of the 10 FSM states (0000000, 0000111, 0011001, 0011110, 0101010, 0101101, 0110011, 0110100, 1001011, 1001100). ■

C. Efficient Encoding with Grouping of FSM States

To the best of our knowledge, previous works split their state encoding schemes into protected and non-protected groups and maintained a minimum distance between each protected state and all other states. Real-world applications may have more complex privilege-level structures. Designers may decide to ignore potential LFI jumps between two protected states because entering one protected state implies authorization to enter another (e.g., once a user logs in, they can access any page associated with their account). Designers may also identify an indefinite number of state groups that should be protected from each other but not from members of the same group (e.g., user A can access any of its own pages, and user B can access any of its own pages, but neither can access the other's pages). Previous works assume that all protected states belong to different groups when this may not be the case. This assumption leads to unnecessary area overhead because members of each distinct group must fulfill the minimum distance requirement from the members of all other groups.

To generate minimum-length encodings with state groups taken into consideration, we propose Algorithm 2. This heuristic generates a minimum distance encoding for each protected state group and appends binary encodings to represent the states within the group. First, we encode all the groups considering each group as one state [Algorithm 2, line 2] using Algorithm 1. Then, based on the most extended group's size, we create binary encoding for each state inside each group [Algorithm 2, line 4 - 6]. The algorithm concatenates $\lceil log_2(largest\ group\ size) \rceil$ bits encoding to the states of all the groups to create the final state encoding [Algorithm 2, line 6]. This process repeats till all the states in the FSM are covered.

Example 2: Assuming the attacker has a BFC of 2, if a designer wanted to split the non-protected states into a group and all protected states into distinct groups, they would run Algorithm 2 with two groups (one protected group for each protected state and one non-protected group), an HD of three, and the largest group size of two (the number of non-protected states). The algorithm would then generate nine group vectors HD apart and concatenate the ceiling of $log_2(largest\ group\ size)$ bits to these vectors to represent all members of the largest group (one bit in this case). The end result would be the set (00000000, 00000001, 00001111, 00110011, 00111101, 01010101, 01011011, 01100111, 01101001, 10010111) in which the first two encodings correspond to the non-protected states. ■

Algorithm 2 State Grouping

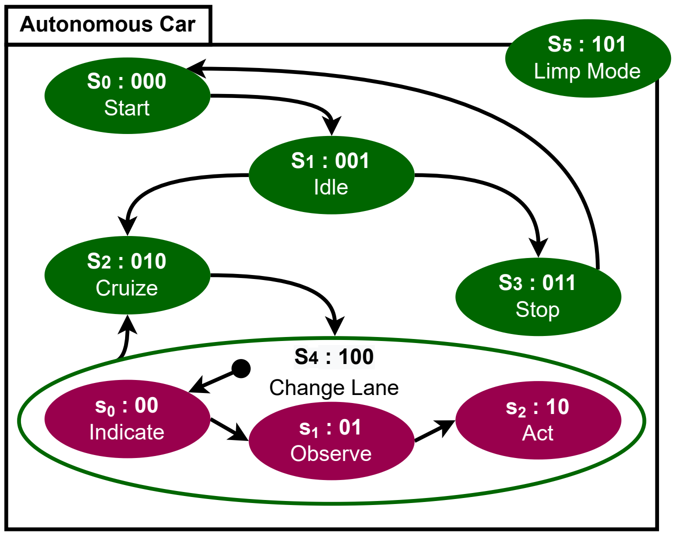
```
$b$: Bit flipping capability
$G$: Set of Groups
$C$: Generated encoding set
1:  $n = max(|g_i|), \forall g_i \in G$
2:  $D \leftarrow GenerateEncodings(|G|, b)$
3:  if $n > 1$ then
4:      for $\forall j$ in $\lceil \log(n) \rceil$ do
5:          for $\forall d_i \in D$ do
6:              $C \leftarrow d_i.[j^{\lceil \log(n) \rceil}]$   ▷ Concatenate with binary encoding.
7:          end for
8:      end for
9:  end if
10: Return $C$
```

Example 3: If, however, the same designer wanted to split the protected states into groups by varying security levels (Edit Account, Deposit Funds, Withdraw Funds, and Transfer Funds in one high-security group and all other protected states in a normal-security group), then they would run Algorithm 2 with three groups (non-protected, normal-security, high-security), an HD of three, and the largest group size of four (the normal-security group's size). In line 3 of Algorithm 2, this would essentially run Algorithm 1 with three states (one for each group) and an HD of three. The rest of the members of each group would be represented by concatenating binary counts to the results of 1. Binary counting ensures that members of a group are as close together as possible. The resultant encodings would be (0000000, 0000001, 0011100, 0011101, 0011110, 0011111, 1100100, 1100101, 1100110, 1100111). Note that the encoding lengths dropped from eight to seven when making this change. ■

D. Handling Nested FSMs

Nested FSMs are designed to simplify the modeling process of FSMs based on the hierarchy of the abstraction level. Figure 4 represents an autonomous car FSM with five main states on the parent FSM. Inside state S_4 (*Change Lane*), there is a child FSM which consists of another three states. Designers can use two methods when implementing this on behavioral models on hardware designs.

1) Implement flag signals inside parent states and trigger the child FSMs using the flag signal. Figure 5 shows the corresponding hardware implementation. In this case, the proposed technique AREST can be directly applied to child and parent FSMs separately, and LFI-resistant hardware implementation can be obtained since the child FSM is isolated from the parent FSM.



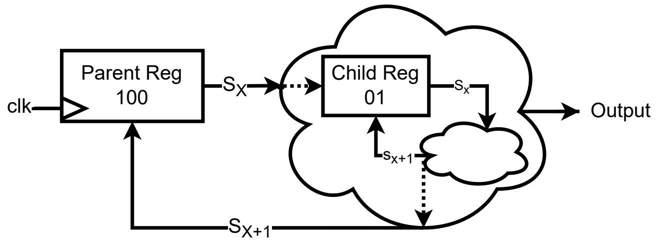


Figure 5: Hardware implementation of an external flag triggered child FSM. Here, the child FSM resides inside the parent FSM's logic circuit.

2) Considering child FSM as a part of the parent FSM and consider it as a separate group. Here the proposed algorithm can be applied with state grouping, and LFI-resistant implementation can be obtained.

IV. Experiments

This section demonstrates the effectiveness of our proposed framework. We first describe the experimental setup. Next, we present the experimental results.

A. Experimental Setup

We ran all our experiments on a host with an Intel Core i7 2.7 GHz CPU and 16 GB of RAM. We developed our proposed work ARest and *Exhaustive* algorithm using Python 3.8.10 code. The author provided the code for the *Probabilistic* [19] approach. We compared our proposed approach ARest with existing methods: *Probabilistic* [21] and *Exhaustive* [22]. We evaluate ARest with different performance parameters such as encoding time with a different number of states and BFC values, length of the generated binary state vector, and hardware area overhead for FSMs. We have used both *Yosys* [25], an open-source tool, and *Pyverilog* [27] python library kit for extracting FSM. For hardware overhead analysis, we have selected several cryptographic hardware designs from OpenCores [28] and synthesized two implementations (with and without our technique) using *Yosys*.

B. Comparison of Encoding Time

Figure 6 represents the runtime comparison of ARest with *Probabilistic* and *Exhaustive* with different BCF values. In [19], the author explains that their method is more suitable for large HDs because the large distance allows additional pruning of possible matrix extensions. In case of LFI-resilient encoding, the attackers can accurately flip only few bits (up to 3 bits due to limited access to precise lasers [17], [20]). Therefore [19] experimentally appears to be unsuitable for this heuristic. In Figure 6a and 6b, we see that the *Probabilistic* heuristic is slower than an *Exhaustive* algorithm for BFC of 1 and 2. As BFC grows, the *Probabilistic* approach starts outperforming *Exhaustive* searches. ARest outperforms both the *Exhaustive* (3076 times faster on average) and *Probabilistic* (2114 times faster on average) approaches in terms of encoding generation time. To evaluate the scalability, we have explored the runtime behavior of ARest in Figure 8 for large and complex FSMs. The increment of encoding vector generation time compared to the number of the states is negligible (in milliseconds).

C. Comparison of Encoding Length

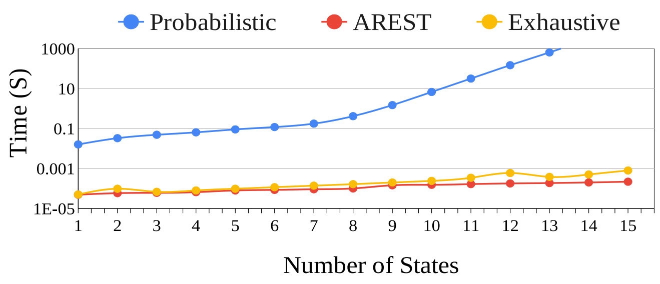
Figure 7 compares the code length in terms of binary state vector size. The size of the vector in state encoding directly impacts the number of flip-flops in the FSM system. Hence, lesser flip-flops will result in a system design with a smaller area on the chip. As it is evident from the graphs, our proposed work ARest significantly outperforms *Probabilistic* encoding in reducing the vector size by more than 50% on average. As expected, our approach provides the same (smallest) code length as *Exhaustive*. Note that *Exhaustive* is not a scalable solution since it requires exponential time as demonstrated in Figure 6.

D. Encoding under State Grouping

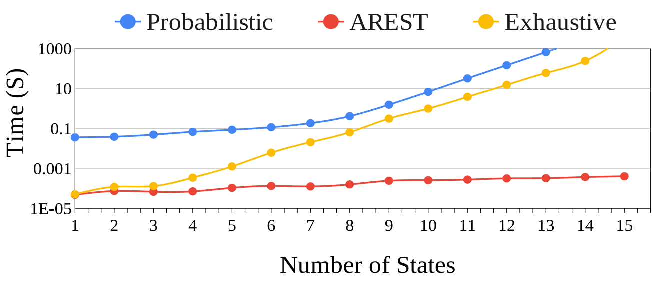
In Figure 9, we demonstrate how ARest affects binary state vector length and area with state-grouping. To show the maximum difference, we considered two extreme scenarios: (Case 1) where all states are considered to be in the same group, and (Case 2) where all states are considered distinct as groups. We can observe that the generated encoding length increases when the number of groups increases for the same number of states in an FSM. The red line in the graphs shows the maximum possible generated encoding length when we split states into individual groups. The area overhead increases with the number of states since increasing the number of groups increase the number of states from which every other state must maintain a minimum required HD for a secure FSM. For example, ARest requires 50% longer encoding for Case 2 over Case 1 when the number of states for the FSM is 20. The difference in the generated encoding grows as the number of states grows in the FSM.

E. Area Overhead

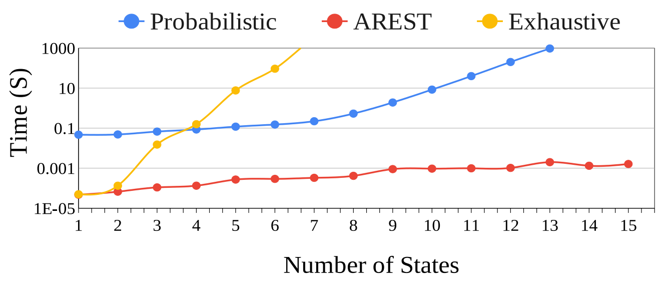
For overhead analysis, we have selected several cryptography designs from OpenCores [28] to find the area overhead



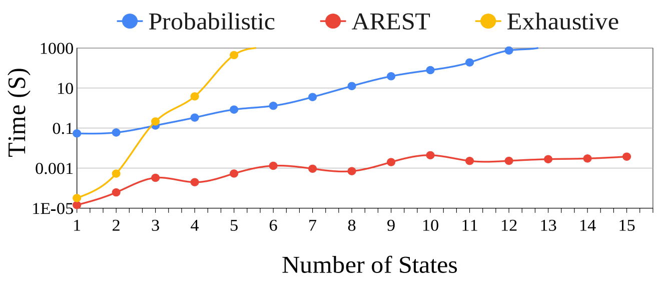
(a) BFC = 1



(b) BFC = 2

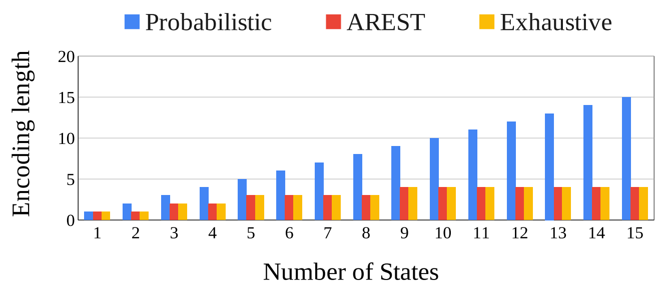


(c) BFC = 3

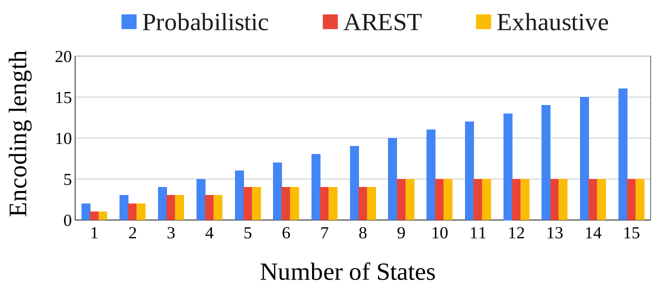


(d) BFC = 4

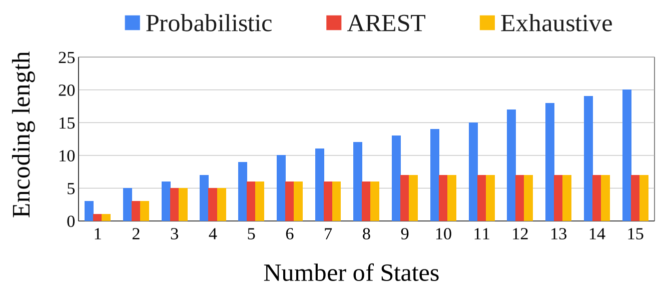
Figure 6: Runtime comparison of AREST with *Exhaustive* [22] and *Probabilistic* [21] techniques for four different BFC values



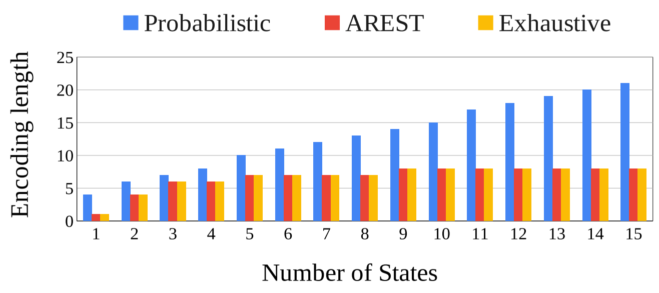
(a) BFC = 1



(b) BFC = 2



(c) BFC = 3



(d) BFC = 4

Figure 7: Generated encoding length comparison of proposed work AREST with *Exhaustive* [22] and *Probabilistic* [21] techniques for four different BFC values

of the proposed work AREST. We have extracted the state machines, which were encoded using *binary encoding* from the selected designs and counted the number of protected and non-protected states. Table I shows the result for the extra overhead added with our proposed solution on synthesized design (column 5) compared to binary encoding. The column 4 and 5 show the code length values produced by Algorithm 1 and 2, respectively. The results highlight the fact that the proposed encoding mechanism adds negligible hardware overheads compared with the original size of the design with respect to the number of gates. Note that the number of gates for the overhead calculations was calculated by removing the FSM optimization pass during synthesis.

V. Conclusion

Finite State Machines (FSMs) are widely used to implement various computation sequences and communication protocols while designing different components of a System-on-Chip

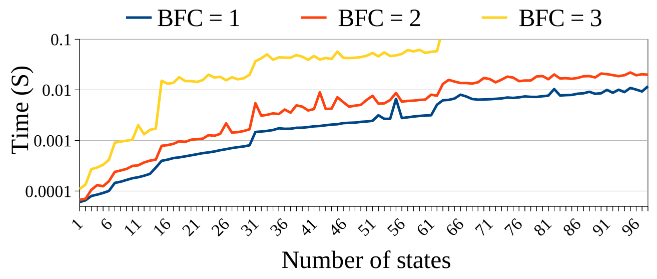


Figure 8: Runtime graph of the proposed policy AREST with different FSM size and BFC values

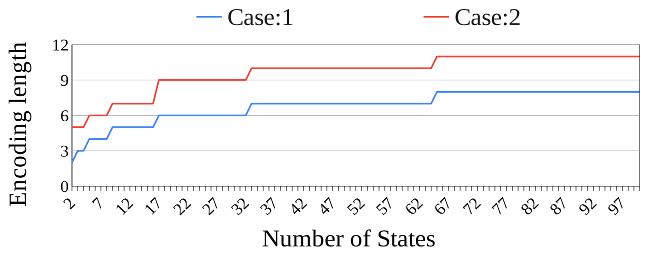


Figure 9: Comparing the vector lengths for different protected state grouping schemes.

(SoC). The security and integrity of an SoC can be compromised if the security of the FSM is bypassed. Laser Fault Injection (LFI) attacks are used to target FSMs by flipping bits of the flip-flops to transition to a protected state. In this paper, we proposed a method for automatically encoding LFI-Attack-Resistant FSMs, AREST. AREST performs state encoding, ensuring that the hamming distance is more than the bit-flipping capability of the LFI attack. Our proposed work demonstrated two major advantages compared to state-of-the-art approaches: significant reduction in code length (50% on average) and drastic improvement (several orders-of-magnitude) in encoding generation time. We have also explored optimization opportunities in real-world applications by grouping states with compatible privilege levels.

Table I: Synthesized area overhead against binary encoding comparison for selected hardware designs with and without state grouping for BFC of 2 (HD = 3). Here $|PS|$ represents the number of protected states, and $|NPS|$ represents the number of non-protected states.

Design	$\|PS\|$	$\|NPS\|$	**CodeLength State Grouping Without**	**CodeLength State Grouping With**	**Synthesized Area overhead**
ECC	4	9	9	7	≈ 0.001%
AES	2	3	7	6	≈ 0.001%
SHA256	3	4	7	6	≈ 0.002%
RSA	4	3	7	6	≈ 0.002%
MIPS	5	14	9	9	≈ 0.01%
MEMORY	8	58	11	11	≈ 0.03%

REFERENCES

[1] Farimah Farahmandi and Prabhat Mishra. FSM anomaly detection using formal analysis. In *IEEE International Conference on Computer Design (ICCD)*, pages 313–320, 2017.
[2] Adib Nahiyan et al. Security-aware fsm design flow for identifying and mitigating vulnerabilities to fault attacks. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 38(6), 2019.
[3] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: optimizing voltage fault injection attacks. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pages 199–224, 2019.
[4] Philippe Maurine. Techniques for em fault injection: equipments and experimental results. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–4. IEEE, 2012.
[5] Mahmoud A Elmohr, Haohao Liao, and Catherine H Gebotys. Em fault injection on arm and risc-v. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 206–212. IEEE, 2020.
[6] Haohao Liao and Catherine Gebotys. Methodology for em fault injection: Charge-based fault model. In *Design, Automation & Test in Europe (DATE)*, pages 256–259. IEEE, 2019.
[7] Jörn-Marc Schmidt and Michael Hutter. *Optical and em fault-attacks on crt-based rsa: Concrete results.* na, 2007.
[8] Mohammad Eslami, Behnam Ghavami, Mohsen Raji, and Ali Mahani. A survey on fault injection methods of digital integrated circuits. *Integration*, 71:154–163, 2020.
[9] Hasini Witharana and Prabhat Mishra. Speculative load forwarding attack on modern processors. In *International Conference on Computer-Aided Design (ICCAD)*, 2022.
[10] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. Scalable test generation for trojan detection using side channel analysis. *IEEE Trans. on Information Forensics and Security*, 13(11):2746–2760, 2018.
[11] Yangdi Lyu and Prabhat Mishra. Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(7):1287–1300, 2020.
[12] Yangdi Lyu and Prabhat Mishra. Maxsense: Side-channel sensitivity maximization for trojan detection using statistical test patterns. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(3):1–21, 2021.
[13] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. *System-on-Chip Security*. Springer, 2020.
[14] Joaquin Rodriguez, Alex Baldomero, Victor Montilla, and Jordi Mujal. Llfi: Lateral laser fault injection attack. In *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–47, 2019.
[15] Shahin Tajik et al. Laser fault attack on physically unclonable functions. In *Fault diagnosis and tolerance in cryptography*, pages 85–96, 2015.
[16] Brice Colombier et al. Multi-spot laser fault injection setup: new possibilities for fault injection attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 151–166, 2021.
[17] Muhtadi Choudhury, Domenic Forte, and Shahin Tajik. Patron: A pragmatic approach for encoding laser fault injection resistant fsms. In *Design, Automation Test in Europe Conference*, pages 569–574, 2021.
[18] Victor Tomashevich, Yaara Neumeier, Raghavan Kumar, Osnat Keren, and Ilia Polian. Protecting cryptographic hardware against malicious attacks by nonlinear robust codes. In *IEEE DFT*, pages 40–45, 2014.
[19] Johannes Zwanzger. *Computergestützte Suche nach optimalen linearen Codes über endlichen Kettenringen unter Verwendung heuristischer Methoden*. PhD thesis, Bayreuth, 2011. msc: 51C05; msc: 94B05.
[20] Muhtadi Choudhury, Shahin Tajik, and Domenic Forte. Sparse: Spatially aware lfi resilient state machine encoding. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2021.
[21] Sunghyu Han. Finding good linear codes using a simple extension algorithm. *IEEE Transactions on Information Theory*, 57(10), 2011.
[22] Stephan Mertens. Exhaustive search for low-autocorrelation binary sequences. *Journal of Physics A*, 29(18), 1996.
[23] M. Best, A. Brouwer, F. MacWilliams, A. Odlyzko, and N. Sloane. Bounds for binary codes of length less than 25. *IEEE Transactions on Information Theory*, 24(1):81–93, 1978.
[24] Intel® quartus® prime pro edition user guide, Jun 2021.
[25] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.
[26] Thi Ngoc Giau Le and Thanh Toan Phan. A simple proof of the improved johnson bound for binary codes. *Bulletin of the Korean Mathematical Society*, 56(2):391–397, 2019.
[27] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, pages 451–460, Apr 2015.
[28] Opencores. https://opencores.org/.