ExploreFault: Identifying Exploitable Fault Models in Block Ciphers with Reinforcement Learning

Hao Guo*, Sayandeep Saha†, Vasudev Gohil*, Satwik Patnaik*, Debdeep Mukhopadhyay‡, Jeyavijayan (JV) Rajendran*
{guohao2019, gohil.vasudev, satwik.patnaik, jv.rajendran}@tamu.edu, sayandeep.saha@uclouvain.be, debdeep@cse.iitkgp.ac.in
*Electrical & Computer Engineering, Texas A&M University, College Station, Texas, USA
†UCL Crypto Group, ICTEAM/ELEN, Université catholique de Louvain, Belgium
‡Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India

***Abstract*—Exploitable fault models for block ciphers are typically cipher-specific, and their identification is essential for evaluating and certifying fault attack-protected implementations. However, identifying exploitable fault models has been a complex manual process. In this work, we utilize reinforcement learning (RL) to identify exploitable fault models generically and automatically. In contrast to the several weeks/months of tedious analyses required from experts, our RL-based approach identifies exploitable fault models for protected/unprotected AES and GIFT ciphers within 12 hours. Notably, in addition to all existing fault models, we identify/discover a novel fault model for GIFT, illustrating the power and promise of our approach in exploring new attack avenues.**

***Index Terms*—Reinforcement Learning, Fault Attack**

I. Introduction

A fault attack (FA) is an implementation-based attack in which an adversary perturbs the computation to extract secrets through faulty system responses. Researchers have demonstrated FAs over different computing platforms, from embedded systems to cloud servers [1], [2], and attackers have launched FAs on several industrial designs, including Sony Playstation [3] and Intel SGX processor [2]. While applicable to symmetric and public-key cryptosystems, FAs have seen significant progress for symmetric-key primitives (e.g., block ciphers). Therefore, protecting and testing symmetric-key cryptosystems for potential FAs remains one of the most active research areas.

A. Fault Models and Fault Space Exploration

Physical faults corrupt one or multiple bit/byte(s) in the cipher state resulting in faulty responses, which adversaries exploit for key recovery. Targeting a block cipher with FA requires a logical abstraction of physical faults, referred to as *fault models*. Adversaries mathematically analyze the underlying cipher (protected/unprotected) concerning the fault models and devise a key recovery algorithm. Typically fault models depend on the (i) nature of the injection, (ii) target implementation, and (iii) target cipher algorithm. For example, the diagonal fault model [4] discovered for AES [5] does not apply to ciphers such as PRESENT [6], GIFT [7], or SIMON [8]. This is primarily due to the structural diversities among ciphers (*i.e.,* different linear and nonlinear functions). A common trend in utilizing FAs on block cipher are to consider some state-of-the-art fault models (e.g., bit/byte/nibble fault) and target a given cipher accordingly.

While finding an attack with a pre-determined fault model works from an adversary's perspective, it is sub-optimal for a defender (designing or verifying a protected block cipher). The defender needs to understand the exploitable fault space of a block cipher, described in terms of fault models. In addition, the defender (ideally) should test the cipher concerning each exploitable fault model to reach a meaningful fault coverage.[1] However, such a task is challenging when only considering pre-determined fault models. It has been observed from the research literature that implementations often fail to provide security due to such sub-optimal fault model analysis [9]. For instance, consider the discovery of the diagonal FA on AES, which appeared later than the single-byte fault models [10]. Contrary to popular belief that only a localized single-byte fault can recover the secret key, the diagonal FA showed that even less precise, multi-byte faults can be leveraged for a successful key recovery. In addition, simple duplication countermeasures for AES were rendered ineffective upon discovering a fault model that injects two equal faults in the redundant modules [11].

B. Limitations of Existing Fault Attack Approaches

As discussed in our previous section, identifying fault models is imperative for a successful FA, necessitating an effective fault space exploration approach. However, state-of-the-art FA techniques have the following limitations, as discussed next.

Diverse Cipher Structures and Human Expertise Dependent. Historically, identifying suitable fault models for a given cipher (protected/unprotected) is a non-trivial task [4], [9], [11]. Manual efforts fall short due to the diverse structures of different ciphers and the associated expertise required. For example, with manual analysis for AES, researchers discovered an optimal FA [12] 8 years after the first FA showcased in [10].

Lack of Automation for Identifying Fault Model. There have been recent efforts in automatically analyzing protected and unprotected block ciphers against FAs [13]–[15]. However, these works require a fault model as an input and perform the analysis with respect to a given fault model, and this takes anywhere between a few minutes to hours [13], [14], [16]. However, the problem of automatically identifying a fault model has been left unexplored in the literature.

Exponential Complexity in Identifying Fault Model. In practice, the number of possible fault models is exponential to the state length (usually 64/128/256 bits) of a block cipher, as every potential single/multi-bit fault pattern[2] qualifies as a candidate fault model. However, only a few fault patterns are exploitable, and testing for each fault pattern is impractical. In a nutshell, using any of these tools to discover potential fault models is difficult, as these tools can only tell the exploitability of a given fault model/pattern but cannot discover a fault model on its own.

To summarize, the limitations mentioned above result in a gap in the automation efforts for FA, especially in fault model identification, which we focus on and address in this work.

C. Our Research Contributions

In this work, we address the aforementioned limitations and propose a generalized and simulation-based fault model identification framework, `ExploreFault`. `ExploreFault` identifies exploitable fault models utilizing a combination of reinforcement learning (RL) and *t*-test and applies to unprotected and protected implementation of block ciphers. Broadly, our approach explores the state of a cipher in a bit-wise manner and identifies exploitable multi-bit fault patterns. Such patterns are later abstracted as fault models

[1]Fault coverage is the percentage of faults for which we can obtain the exploitability status in a block cipher.

[2]Fault pattern is a set of bits where to inject faults. The fault model is a generalization of fault patterns.

considering some physical constraints of fault injection.[3] One crucial step in this process is to identify the exploitability of fault patterns explored by the RL agent. To that end, we utilize the *t*-test for its ease of computation. The results of the *t*-test provide feedback to the RL agent during training. Researchers have used the *t*-test to identify FA-related information leakage in protected implementations [13]. However, we utilize higher-order calculation and a collection of intermediate block cipher states and ciphertext to aid the RL agent.

However, simply using *t*-test to train the RL agent is not enough. To discover successful fault models, we face challenges such as: (i) long evaluation time for calculating *t*-test during training and (ii) sub-optimal convergence of the learning. To overcome these challenges, we utilize strategies that (i) perform reward calculation when the final fault pattern is created and (ii) employ an exponential reward function for returning more reward to the RL agent on each bit increase in the fault pattern selected. The runtime of `ExploreFault` is between 3–12 hours, replacing expert knowledge and tedious analysis, which might take up to a few weeks or even months. We provide further details regarding how we address the challenges in Section III. The primary contributions of our work are enumerated as follows.

1) We close the gap in state-of-the-art FA automation by conceptualizing and implementing an automated fault model identification/discovery methodology, `ExploreFault` using RL.
2) To the best of our knowledge, this is the first work to utilize RL in the context of FAs. We argue that RL is an appropriate choice given its exploration power across complex search spaces. One major challenge in FA testing is the size and complexity of the search space, which is efficiently explored with `ExploreFault`.
3) We demonstrate that `ExploreFault` identifies all known exploitable fault models for AES, thereby replicating 4 prior works.
4) `ExploreFault` identifies exploitable fault models even in the presence of countermeasures. To this end, we performed a case study on AES with a duplication-based countermeasure.
5) To showcase the generality of `ExploreFault`, we present results on another block cipher, GIFT, which is a widely-used lightweight block cipher for its smaller implementation size and high runtime efficiency. It is a part of GIFT-COFB mode [18], which is a NIST LWC finalist.
6) For GIFT, `ExploreFault` identifies a new fault model (in addition to previously discovered fault models), which leads to key recovery, which we verify using the ExpFault tool [19]. Since 2017 (when GIFT was proposed), researchers discovered only two fault models using human analysis. However, using `ExploreFault`, we discovered/identified a new fault model within 12 hours in an automated manner. This significant reduction in time attests to the power and promise of `ExploreFault`.

II. Background and Preliminaries

In this section, we provide a brief overview of fault attack (FA), related automation approaches in FAs, and reinforcement learning.

A. Fault Attack

Block ciphers are iterative structures where one mathematical transformation called *round* operates on a secret state (of block size 64/128/256 bits) several times to generate the ciphertext. Each round consists of several linear and nonlinear Boolean functions (often some special nonlinear functions called S-boxes) and XORs a round key generated from a key schedule. For example, the AES block cipher consists of 16 8-bit S-box operations in a round function, followed by a byte shifting, multiplication by a constant matrix, and XORing of 128-bit round keys. GIFT [18] comprises 4-bit S-Boxes, followed by a bit-permutation layer, and XORing of round key bits. FAs aim to recover a few of such round key bits from a block cipher and thereby compute the master key. For most cases, especially for unprotected implementations, the faulty system responses are the faulty ciphertexts. The key recovery strategy in any FA is to utilize some statistical bias in the intermediate state of a block cipher due to fault injection. Such statistical bias (*i.e.,* deviation of the intermediate state from a uniformly random state) is either expressed as a system of equations (in differential fault analysis, a.k.a. DFA attacks) or measured through some statistical test (for statistical fault attacks, a.k.a. SFA, SIFA, FTA). Another important property of such statistical bias is that it only becomes visible with the correct key. Therefore, the adversary obtains the faulty system responses, guesses a few key bits, and partially decrypts till the biased state. If the bias is observed, the corresponding key guess is considered correct.

B. Automation in Fault Attacks

Researchers have developed several automated tools for analyzing protected/unprotected block ciphers concerning FAs. Tools such as ExpFault [14] or XFC [15] analyze unprotected implementations and generate the attack algorithm for a given fault according to some fault model. On the other hand, tools such as ALAFA [13] or FIVER [16] analyze the protected block ciphers. FIVER considers the bit fault model and formally establishes that no fault reaches the output (or some observation point) based on ideas from integrated circuit testing. ALAFA analyzes the faulty system responses for potential information leakage. It is a fault simulation-based approach that applies *t*-test between two ciphertext distributions to detect potential information leakage. However, ALAFA also takes predefined fault models as input. This work aims to fill the gap in FA automation flow by automatically identifying fault models that an adversary can provide to the aforementioned tools. To that end, we utilize reinforcement learning, which we describe briefly next.

C. Reinforcement Learning

RL entails an agent learning to solve problems requiring a sequential decision-making process. The agent learns using a trial-and-error manner by interacting with the environment. Through interactions, the agent understands the environment and learns actions that maximize reward or minimize penalties from the environment. The maximum number of interactions the agent can have with the environment before reset is called an episode. To formulate RL problems using a Markov decision process, an RL agent operates on state space $\mathcal{S}$ through a set of actions $\mathcal{A}$. The state transition in this state space happens through the actions in $\mathcal{A}$, with each transition having some probability P. The reward $\mathcal{R}$ measures the fitness to take action in a given state. Through interactions with the environment, an optimal control policy π_θ is found to achieve the maximum cumulative expected reward. The policy π_θ parameterized by θ is defined as a mapping from the set of states $\mathcal{S}$ to the set of probability measures on $\mathcal{A}$ in each state.

RL agents have demonstrated tremendous efficiency in navigating high-dimensional search space and finding optimal policies. As a result, researchers have used RL in security problems, such as fuzzing [20], internet-of-things security [21], and cybersecurity [22]. However, utilizing RL for fault space exploration is unexplored, and, hence is the focus of this work.

III. ExploreFault : Exploring Fault Space with Reinforcement Learning

Faults in block ciphers can corrupt one or multiple bits in a state. For example, nibble/byte/multi-byte faults are multi-bit faults that often occur in practical systems. Therefore, identifying exploitable

[3]Practical fault injection setups can corrupt single-bits [17] or one/multiple bytes/nibbles in the cipher state [4], [12]. Even if `ExploreFault` identifies a multi-bit fault pattern, it might not be possible to individually corrupt each bit in the pattern. Therefore, abstracting multi-bit patterns identified by `ExploreFault` as single/multi-byte faults is important.

multi-bit fault patterns is a reasonable approach to discovering fault models. This approach, however, comes with two major challenges: (i) generating interesting fault patterns, (ii) efficiently deciding the exploitability of each fault pattern.

We address the first challenge using RL. The RL agent utilizes the exploitability information of each discovered fault pattern and figures out new exploitable multi-bit fault patterns with a higher bit count. Such fault patterns are later abstracted to nibble/byte or multi-byte fault patterns during an abstraction phase. For the second challenge, we utilize Welch's *t*-test to calculate the exploitability of the fault patterns in a generic manner. The overall flow culminates into a fault model identification framework–`ExploreFault`.

A. Why Reinforcement Learning?

There are two main features that make RL a good fit for fault pattern exploration: (i) **Sequential decision-making.** Automatically finding optimal multi-bit fault patterns is inherently a sequential process, as one needs to augment bits one at a time in the pattern and then check the exploitability. The choice of the bit to be added is also critical. RL is a good fit for such sequential processes. and (ii) **Huge search space.** The search space associated in this case is huge. For an 128-bit cipher state, there are 2^{128} possible patterns, and it is impractical to explore all of them. Even considering the inherent regularities (e.g., same S-Boxes operating on each byte/nibble) in the block cipher structures, the search space remains huge and complex to model. Smartly navigating this search space, therefore, requires a generalization technique that can learn from prior experiences and navigate the search space better. RL is a perfect fit for such problems, which has been demonstrated by other security domains [20]–[22].

B. Preliminary Formulation

Now we design the preliminary RL agent by formulating a fault model identification problem as a Markov decision process.

- **State** at time t, s_t, is a binary vector, whose i^{th} entry indicates if a fault is injected into the i^{th} bit of the block cipher state or not.
- **Action** at time t, a_t, is the bit location that the RL agent selects to inject a fault. For each action, the RL agent can choose any bit location from all the bit locations of the block cipher state.
- **State transition** $P(s_{t+1}|a_t, s_t)$ denotes how the state evolves. In our case, the next state s_{t+1} is obtained in a deterministic manner by updating the i^{th} entry in the current state s_t as 1, where $i = a_t$.
- **Discount rate** γ $(0 \leq \gamma \leq 1)$ scales down rewards over time.
- **Reward** at time t, $\mathcal{R}(s_t, a_t)$, depends on the information leakage denoted by l. As shown in Equation (1), if l is less than a threshold θ, *i.e.,* information leakage can not be observed, reward is β (< 0). Otherwise, the reward is equal to n, the number of bits chosen by the RL agent so far. This linear reward targets maximizing the number of bits while maintaining a high information leakage. All the subsets of that fault model are classified as fault models as well, leading to more fault models being discovered.

$$\mathcal{R}(s_t, a_t) = \begin{cases} \beta, & \text{if } l < \theta \\ n, & \text{otherwise} \end{cases} \tag{1}$$

C. Determining Fault Exploitability

The core of the reward calculation is to determine the exploitability of a given fault pattern. However, determining the exploitability of a fault pattern in a generic manner is challenging, stemming from the diversity (*i.e.*, different mathematical structure) of the block cipher and associated countermeasures (if any). In addition, the fault pattern determines the nature of the attack path. Although an adversary can utilize tools such as ExpFault or XFC to determine the fault exploitability, they require complex modeling and analysis steps. Furthermore, such tools only apply to unprotected ciphers, and we require an entirely different strategy for protected ciphers [13], [16]. Therefore, we resort to a simple albeit generic strategy suitable for calculating rewards during RL training and applicable for protected and unprotected implementations with minor modifications.

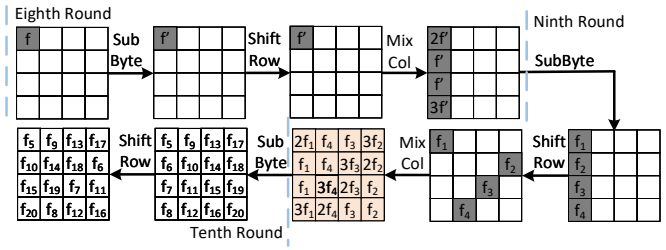

Fig. 1. Fault propagation for fault injected in 8^{th} round of AES. Linear pattern can be observed in the input of 10^{th} round (shaded orange) [14].

We utilize the popular Welch's *t*-test to determine fault exploitability by measuring information leakage for FA from protected implementations based on ciphertexts [13]. For unprotected implementations, we use *t*-test on faulty intermediate states of a cipher. The main intuition is that *if an intermediate state is distinguishable from a uniformly random state, it may have some exploitable statistical patterns.* We utilize the *t*-test here to distinguish between a uniformly random state from a (faulty) intermediate state differential (*i.e.,* XOR differential between a correct and corresponding faulty state originated from a fault pattern). This state differential is chosen from the input/output or intermediate computations of the last few rounds.[4] To generate the faulty state data, we perform fault simulation with a given fault pattern with several random plaintexts. The null hypothesis for the *t*-test is that the two datasets are from the same population. We set the parameter θ as 4.5. If the *t*-test statistic l is larger than 4.5, we can reject the null-hypothesis with confidence $> 99.999\%$. Therefore, if *t*-test shows that the state differentials are different from a random distribution, it indicates that some distinguishable patterns can be observed, which can be exploitable for an attack.

The *t*-test for evaluating the state differential is performed separately for each bit/nibble/byte of a (64/128) bit state differential. Considering the state-differential as a single variable will involve a probability distribution with support of 2^{64} or 2^{128}, which is impractical to handle. However, treating each bit/nibble/byte separately can be sub-optimal in many cases. For illustration, we consider AES with a single-byte fault injected at the 8^{th} round. The *t*-test checks the input of the 10^{th} round, where an adversary can observe some linear patterns in the state differentials (Fig. 1). Now such linear state-differential patterns can only be sensed by *t*-test if at least two bytes are considered together. To handle such multi-bit/nibble/byte state patterns in a generic manner, we use higher-order *t*-test, as described in [13]. It computes the higher-order statistical moments from the data and applies the *t*-test on that data. As indicated in Table I, injecting byte faults or diagonal faults in AES do not show any leakage with the first-order t-test. Conversely, the second-order t-test can easily capture this information leakage. Our strategy is to first apply a byte-wise *t*-test (order $d = 1$) and then gradually increase the order ($d = 2, 3, \ldots, G$). The highest order G is an evaluator choice. We use $G = 2$ for our test cases, as no new fault patterns were discovered beyond this.

D. Improving Preliminary RL Formulation

The preliminary formulation of the RL agent results in slow convergence. The agent finishes only 72 episodes in 24 hours and only

[4]Checking the last few rounds for statistical patterns remains sufficient as most FAs either target the last few rounds or are round oblivious.

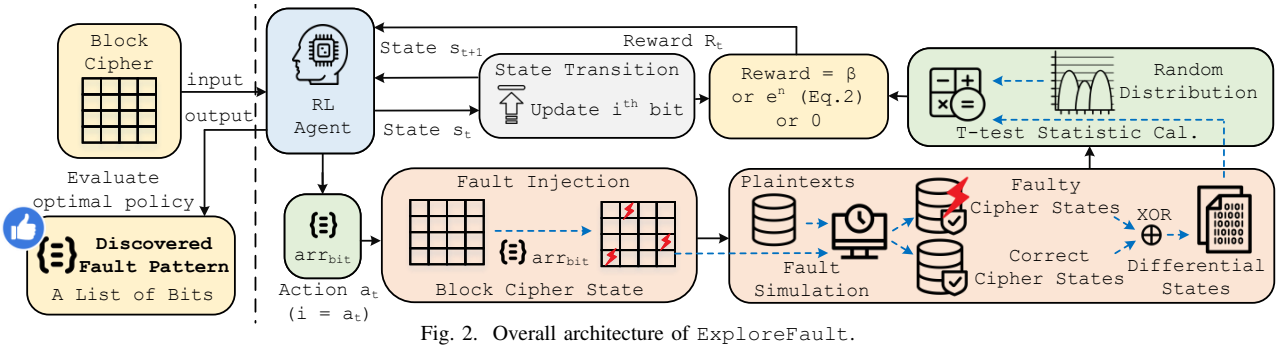

Fig. 2. Overall architecture of ExploreFault.

TABLE I
COMPARISON OF FIRST-ORDER AND SECOND-ORDER t-TESTS FOR AES

Fault Model	Faulty Bits	First-order	Second-order
Byte	0, 1, 2, 3, 4, 5, 6, 7	2.36 (< 4.5)	207.56
Diagonal	29, 34, 35, 38, 77, 118	1.41 (< 4.5)	207.14

TABLE II
TRAINING RATE COMPARISON FOR AES

Method	Training Rate	
	Episodes/Min.	Steps/Min.
Reward at each step	0.054	6.802
Reward at end of episode	6.221	787.667
Improvement	115.2×	115.8×

explores up to 3-bit fault patterns for AES. We adopt the following strategies to improve efficiency, as discussed below.

Long Training Time. The reward is computed at each step during training in the preliminary formulation. Each reward computation involves performing fault injection with the current selection bits, fault simulation, and t-test evaluation. All the aforementioned steps take about 1 second, and since an RL agent requires thousands of episodes to learn, the reward computation becomes a bottleneck preventing the agent from learning quickly. To reduce the training time, we calculate the reward only at the end of an episode. Note that this reduction in reward computation time may lead to a degradation in accuracy since the agent receives feedback less frequently. However, empirically, we observe that the hit to accuracy is insignificant since the agent can still identify exploitable fault models (Section IV). On the other hand, the solution provides $> 115\times$ speedup in training, as evidenced by results in Table II.

Sub-optimal Convergence. Although the preliminary formulation detailed above results in a bit fault model, the agent can only converge to a sub-optimal policy till the multi-bit fault model, where the number of bits is less than or equal to 3. Therefore, the fault space for the block cipher needs to be well explored. We replace the linear reward function with an exponential reward function (Equation (2)) so that for each additional bit (successfully) selected by the agent, it gets more reward. In other words, the marginal number of bits increases, returning more reward to the agent, and facilitating the RL agent to find more bits to inject faults.

$$\mathcal{R}(s_t, a_t) = \begin{cases} \beta, & \text{if } l < \theta \\ e^n, & \text{otherwise} \end{cases} \quad (2)$$

Figure 3 illustrates the impact of the two reward functions on the convergence of the agent. In the case of the linear reward function, the agent converges to a reward of 3. In contrast, with the exponential reward, the agent converges well; in fact, the logarithm of the converged reward is 17.

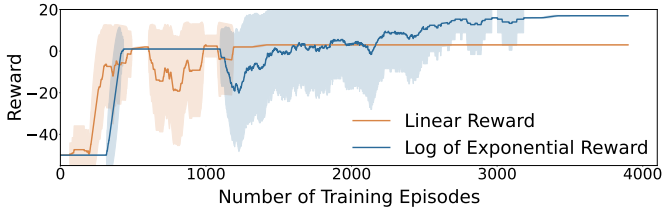

Fig. 3. Agent's inability/ability to learn with linear/exponential reward (AES).

E. Final Agent Architecture

Figure 2 illustrates the flow of ExploreFault that includes all the solutions we described above. In the beginning, the state vector s_0 is initialized to zero, and an empty array arr_{bit} is created to record all actions chosen by the agent. The agent takes its first action a_0, and the selected bit location a_0 is appended to arr_{bit}. Subsequently, the environment determines the next state s_1, where the a_0^{th} entry in the state vector is updated as 1. At each step t, the RL agent takes an action a_t according to the current state s_t, and the next state s_{t+1} is defined as a vector where the i^{th} entry in s_t is updated as 1, where $i = a_t$. In the meantime, the action a_t is added to arr_{bit} if a_t is not in arr_{bit}, and the reward is given as 0 for all intermediate steps. The cycle goes on for T steps, where T is the total number of bits in the block cipher state. In the last step, final arr_{bit} is created, which is the fault model chosen by the RL agent. We perform fault injection on the state of block cipher according to this fault pattern. Then, we collect faulty ciphertexts and intermediate block cipher states and measure the t-test statistic based on the differential state distribution. Finally, the reward is calculated following Equation (2).

F. Constructing the Fault Models

ExploreFault returns a multi-bit fault pattern in the end. However, in most practical cases, faults are single-bit or multi-nibble/byte. In addition, the nibble/byte boundaries are defined concerning the structure of the round function (e.g., as inputs of S-boxes). Therefore, we abstract these multi-bit fault patterns to nibble/byte-wise patterns while generating the final fault models. To construct the corresponding fault models, we examine each of the bits in the fault model and identify the bytes to which those bits belong. Finally, to ensure this more abstract byte-wise fault model is also exploitable, we evaluate the new byte-wise fault model offline using t-test. If such byte-wise models also result in high t-test values, we report them as fault models. Otherwise, we report the specific multi-bit pattern observed by RL. Furthermore, we also extract fault patterns showing high leakage from the RL training log. These patterns are also considered exploitable and abstracted to bytes/nibbles. In general,

TABLE III
COMPARISON OF EXPLOREFAULT WITH EXISTING FAULT MODEL IDENTIFICATION RESEARCH ON UNPROTECTED AES AND GIFT BLOCK CIPHERS

Block Cipher	Technique	Year	Fault Model				Automated/Manual Analysis	Time
			Bit	Nibble	Byte	Diagonal		
AES	[10]	2003	✓		✓		Manual	N/A
AES	[23]	2003			✓		Manual	N/A
AES	[4]	2009				✓	Manual	N/A
AES, GIFT	[14]	2018		✓	✓		Manual	N/A
GIFT	[24]	2021	✓	✓			Manual	N/A
GIFT	[25]	2022	✓	✓			Manual	N/A
AES, GIFT	ExploreFault (This work)	2022	✓	✓	✓	✓	Automated	$<$ 12 hours

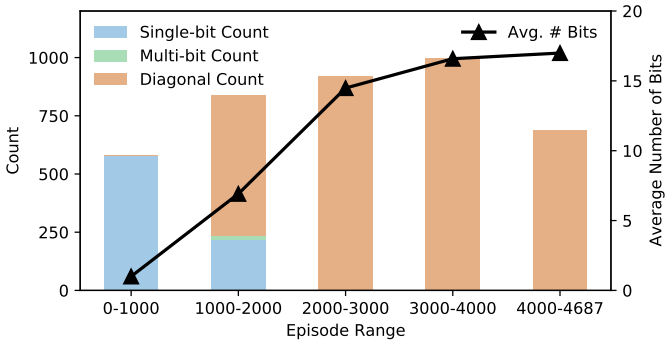

Fig. 4. Fault models discovered by ExploreFault for unprotected AES.

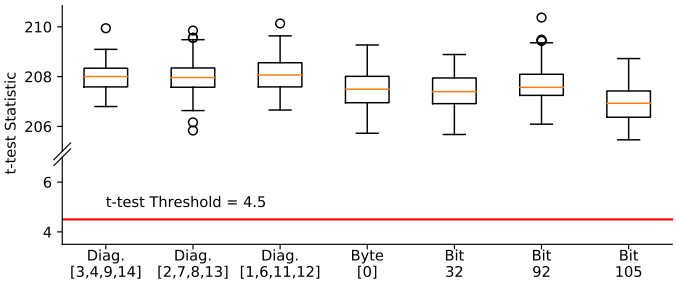

Fig. 5. Random fault simulation for discovered fault model for AES.

we see most proper subsets of the final multi-bit fault pattern as exploitable. Finally, RL only returns a set of representatives from different fault models but not all. However, exploiting the structural similarities among different parts of a block cipher, we extend them to other undiscovered instances. For instance, if RL returns a few representatives of a byte fault model, we also generate and check the other representatives using *t*-test.

G. Evaluating Protected Implementations

We extend the capability of ExploreFault to analyze a protected block cipher by evaluating the *t*-test on the ciphertext instead of the intermediate differential block cipher states. In these cases, we attempt to observe if some information leakage exists at the ciphertext output due to faults, and the RL agent learns based on that information. For countermeasures which mute the ciphertexts upon detecting a fault, we return a random string having the same length as the ciphertext upon each muting. The exploration of RL happens on the state-space of the protected cipher in a similar manner as the unprotected one. Having detailed our fault space exploration technique, we present our results on widely-used block ciphers.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

We implement ExploreFault using Python 3.8.10 with PyTorch1.6 package. We employ Proximal Policy Optimization [26] as our RL algorithm and use the vectorized environment in Stable-Baselines3 to significantly reduce the training time. The experiments are performed on an Ubuntu 20.04 machine with a 32-core 280w AMD processor and an NVIDIA A5000 GDDR6 GPU. We set the parameter β as -50 so that the reward is negative iff the fault injection locations selected by the RL agent do not lead to a successful fault attack.

B. Evaluation of AES Block Cipher Without Countermeasure

Discovered Fault Model According to Learned Policy. We first evaluate ExploreFault on AES without any countermeasure. We select the last three rounds of AES and run ExploreFault on each round independently and note that ExploreFault converges to an optimal policy. The fault model derived from the converged optimal policy is to inject faults in bytes $\{2,\ 7,\ 8,\ 13\}$ in the 8^{th} round of AES. This is the diagonal fault model, reported by [4]. The 9^{th} round injections also report exploitable fault models. However, we found the most interesting fault models for the 8^{th} round, so we mainly discuss them here.

To demonstrate the efficacy of ExploreFault, we perform the same experiments with different hyperparameter settings and observe that it finds different fault models, including the byte fault model, bit fault model, and other diagonal fault models. However, to speed up the process and exploit the inherent regularities present in such cipher structures, we also directly check the other diagonal faults using *t*-test. As listed in Table III, ExploreFault automatically finds all the fault models reported by prior works (which only find one or two fault models with manual analysis or through previous studies).

Discovered Fault Model During Training. Figure 4 depicts the different fault models discovered by ExploreFault during training and the average number of bits selected by ExploreFault per 1K episodes. During the first 1K episodes, $\approx$ 600 episodes report a single-bit fault model whose *t*-test value is larger than 4.5. Then, ExploreFault navigates the AES state's fault space and discovers multiple bits and diagonal fault models in the second 1K episodes. Finally, for the last 3K episodes, ExploreFault sticks to the diagonal fault model while maximizing the number of bits to inject faults in. We also observe that the subsets of the fault model ExploreFault converges to are valid fault models too.

Simulations on the Discovered Fault Model. Next, we verify that the fault models discovered by ExploreFault indeed lead to successful information leakage. To that end, we inject 100 random faults for each fault model discovered by ExploreFault and measure the information leakage through *t*-test. Figure 5 illustrates the distributions of the *t*-test statistics. Note that for all fault models, the *t*-test statistics are well above the leakage classification threshold of 4.5, validating that our fault models cause information leakage.

TABLE IV
RESULTS ON PROTECTED AES

Bit Selected by ExploreFault		Episode Length	# Episodes	Runtime
Branch #1	Branch #2			
76	76	256	≈ 3000	$<$ 3 hours

TABLE V
DISCOVERED FAULT MODEL DURING THE FIRST 1K TRAINING EPISODES

Fault Model	Nibble Location	# Times
2 nibbles	$\{\mathbf{10},\ \mathbf{11}\}$	1
3 nibbles	$\{1, 10, 11\}$, $\{5, 10, 11, \}$, $\{\mathbf{9}, \mathbf{10}, \mathbf{11}\}$, $\{\mathbf{10}, \mathbf{11}, \mathbf{14}\}$	4
4 nibbles	$\{\mathbf{8}, \mathbf{10}, \mathbf{11}, \mathbf{12}\}$, $\{\mathbf{8}, \mathbf{9}, \mathbf{10}, \mathbf{11}\}$, $\{\mathbf{8}, \mathbf{9}, \mathbf{11}, \mathbf{14}\}$, ...	76
5 nibbles	$\{\mathbf{8}, \mathbf{9}, \mathbf{10}, \mathbf{11}, \mathbf{12}\}$, $\{\mathbf{9}, \mathbf{10}, \mathbf{11}, \mathbf{12}, \mathbf{14}\}$, ..	193
6 nibbles	$\{\mathbf{8},\ \mathbf{9},\ \mathbf{10},\ \mathbf{11},\ \mathbf{12},\ \mathbf{14}\}$, ...	255

C. Evaluation of AES Block Cipher With Countermeasure

We now investigate the capability of ExploreFault towards fault model identification for AES with a countermeasure, which employs redundant modules of the cipher. After collecting ciphertexts from both modules, the distributions of those ciphertexts are examined. If they are different, the fault gets captured. To evade this countermeasure, ExploreFault learns to automatically select the same set of bits to inject faults at bit 76 in the computational branches. Table IV demonstrates that ExploreFault successfully chooses the fault model for protected AES.

D. Evaluation of GIFT Block Cipher Without Countermeasure

To further demonstrate the generality of our proposed approach, we evaluate ExploreFault on another block cipher, GIFT. ExploreFault is designed to select bits to inject faults in the 25^{th} round of GIFT-64, and *t*-test calculation is performed on differential state after the S-box function in the 27^{th} round and later. Table V exhibits the discovered fault models and their corresponding occurrence frequencies during the first 1K training episodes. The most significant result is that ExploreFault finds a new fault model (in addition to those found by previous works), which is to inject a fault at nibbles $\{8, 9, 10, 11, 12, 14\}$. **It is worth mentioning that no previous work has reported this fault model.**

Previous works on GIFT report attacks with single nibble/byte or single-bit faults [14], [24], [25]. To evaluate the new multi-nibble fault model discovered by ExploreFault, we utilize the ExpFault tool [19]. Given a fault model, ExpFault reports a potential key recovery strategy. ExpFault can recover 80 of 128 key bits for the new multi-nibble fault. The offline complexity for recovering these 80 key bits is $2^{33.15}$, which is small enough to search exhaustively. Due to the key-schedule structure of GIFT, the remaining 48 key bits cannot be recovered with this fault injection and requires a similar fault to be injected at round 23. Overall, ExploreFault provides us with meaningful insights into the fault models. Note that *t*-test-based exploitability identification may lead to fault models that are exploitable only with very high key-recovery complexity (which can be calculated using tools such as ExpFault). However, as our experiments indicate, ExploreFault successfully identifies/discovers fault models that lead to practical key-recovery complexity.

V. CONCLUSION

To the best of our knowledge, we report the first-ever work in utilizing RL for fault space exploration in the context of FA. By conceptualizing and developing an automated methodology, without relying on human expertise, ExploreFault closes the gap in state-of-the-art FA techniques by automatically identifying all fault models discovered by human experts across 19 years (2003–2022) for AES and GIFT. We demonstrate the success of ExploreFault in 3 scenarios: (i) unprotected AES, (ii) AES with a countermeasure, and (iii) unprotected GIFT. Notably, for GIFT, ExploreFault discovers a new exploitable fault model (within 12 hours) leading to key recovery, which is interesting given that in the last 5 years since GIFT was proposed, researchers have reported only two fault models. This significant reduction in time to identify a new fault model attests to the power and promise behind utilizing RL to explore large search spaces. Finally, ExploreFault provides defenders with a quick and automated way to evaluate the susceptibility of ciphers to FAs, as well as to test resilience of fault attack-protected implementations.

ACKNOWLEDGMENTS

The work was partially supported by the National Science Foundation (NSF CNS–1822848 and NSF DGE–2039610). The work also receives partial support from the project entitled "Development of Secured Hardware And Automotive Systems", funded by C3iHub, IIT Kanpur, under the National Mission on Interdisciplinary Cyber-Physical Systems, Department of Science and Technology (DST), Govt. of India.

REFERENCES

[1] A. Barenghi *et al.*, "Low voltage fault attacks to AES," in *IEEE HOST*, 2010, pp. 7–12.
[2] K. Murdock *et al.*, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *IEEE S&P*, 2020, pp. 1466–1482.
[3] Y. Lu, "Attacking hardware AES with DFA," *arXiv preprint arXiv:1902.08693*, 2019.
[4] D. Saha *et al.*, "A diagonal fault attack on the advanced encryption standard." IACR Cryptology ePrint Archive, 2009.
[5] J. Daemen *et al.*, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
[6] A. Bogdanov *et al.*, "PRESENT: An ultra-lightweight block cipher," in *CHES*. Springer, 2007, pp. 450–466.
[7] S. Banik *et al.*, "GIFT: a small PRESENT," in *CHES*. Springer, 2017, pp. 321–345.
[8] R. Beaulieu *et al.*, "The SIMON and SPECK lightweight block ciphers," in *ACM/IEEE DAC*, 2015, pp. 1–6.
[9] S. Patranabis *et al.*, "Fault tolerant infective countermeasure for AES," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 3–17, 2017.
[10] C. Giraud, "DFA on AES," in *Proc. of 4th Int. Conf. on Advanced Encryption Standard*. Springer, 2004, pp. 27–41.
[11] S. Patranabis *et al.*, "A biased fault attack on the time redundancy countermeasure for AES," in *COSADE*. Springer, 2015, pp. 189–203.
[12] M. Tunstall *et al.*, "Differential fault analysis of the advanced encryption standard using a single fault," in *WISTP*. Springer, 2011, pp. 224–233.
[13] S. Saha *et al.*, "ALAFA: Automatic leakage assessment for fault attack countermeasures," in *ACM/IEEE DAC*, 2019, pp. 136–142.
[14] ——, "ExpFault: an automated framework for exploitable fault characterization in block ciphers," *IACR TCHES*, no. 2, pp. 242–276, 2018.
[15] P. Khanna *et al.*, "XFC: A framework for eXploitable Fault Characterization in block ciphers," in *ACM/IEEE DAC*, 2017, pp. 1–6.
[16] J. Richter-Brockmann *et al.*, "Fiver–robust verification of countermeasures against fault injections," *IACR TCHES*, pp. 447–473, 2021.
[17] S. Saha *et al.*, "Fault template attacks on block ciphers exploiting fault propagation," in *EUROCRYPT*. Springer, 2020, pp. 612–643.
[18] S. Banik *et al.*, "Gift-cofb," *Cryptology ePrint Archive*, 2020.
[19] S. Saha, "ExpFault," https://github.com/sayandeep-iitkgp/ExpFault, 2022.
[20] K. Böttinger *et al.*, "Deep reinforcement fuzzing," in *IEEE S&P Workshops*, 2018, pp. 116–122.
[21] X. Liu *et al.*, "On deep reinforcement learning security for Industrial Internet of Things," in *Computer Communications*, vol. 168, 2021, pp. 20–32.
[22] T. T. Nguyen *et al.*, "Deep Reinforcement Learning for Cyber Security," in *IEEE TNNLS*, 2021, pp. 1–17.
[23] G. Piret *et al.*, "A Differential Fault Attack Technique Against SPN Structures, with Application to the AES and KHAZAD," in *CHES*. Springer, 2003, pp. 77–88.
[24] H. Luo *et al.*, "General differential fault attack on present and gift cipher with nibble," *IEEE Access*, vol. 9, pp. 37 697–37 706, 2021.
[25] S. Liu *et al.*, "Fault attacks on authenticated encryption modes for gift," *IET Information Security*, vol. 16, no. 1, pp. 51–63, 2022.
[26] J. Schulman *et al.*, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017.