# EISec: Exhaustive Information Flow Security of Hardware Intellectual Property Utilizing Symbolic Execution

Farhaan Fowze
University of Florida
Email:muhtadichoudhury@ufl.edu

Muhtadi Choudhury
University of Florida
Email:muhtadichoudhury@ufl.edu

Domenic Forte
University of Florida
Email:dforte@ece.ufl.edu

***Abstract*—Hardware IPs are assumed to be roots-of-trust in complex SoCs. However, their design and security verification are still heavily dependent on manual expertise. Extensive research in this domain has shown that even cryptographic modules may lack information flow security, making them susceptible to remote attacks. Further, when an SoC is in the hands of the attacker, physical attacks such as fault injection are possible. This paper introduces EISec, a novel tool utilizing symbolic execution for exhaustive analysis of hardware IPs. EISec operates at the pre-silicon stage on the gate level netlist of a design. It detects information flow security violations and generates the exhaustive set of control sequences that reproduces them. We further expand its capabilities to quantify the confusion and diffusion present in cryptographic modules and to analyze an FSM's susceptibility to fault injection attacks. The proposed methodology efficiently explores the complete input space of designs utilizing symbolic execution. In short, EISec is a holistic security analysis tool to help hardware designers capture security violations early on and mitigate them by reporting their triggers.**

## I. Introduction

The complexity of integrated circuits (ICs) and subsequently the computational performance of a system-on-chip (SoC) have skyrocketed in the last two decades, mainly due to breakthroughs in technology scaling. These SoCs are at the heart of modern consumer electronic devices, e.g., smartphones, smart TVs, laptops, etc. This consumer trend now shares a trillion dollar global market and is expected to continue rising [1]. On the other hand, as time-to-market becomes more stringent, IC manufacturing companies need to shorten the product cycle and review their process flows with the goal of maintaining best quality of products. Another contributing factor is the progressive refinement of the electronic design automation tools that warrants a substantial reduction of circuit design time. However, the intricate nature of modern IC functionalities along with the strict time-to-market may lead to vulnerabilities overlooked in chip designs.

From the attackers' perspective, the diversity of known attacks is high and continues to rise. An ingenious attacker can always utilize the design vulnerabilities to compromise the security of complex SoCs, particularly if there is no resiliency in the design against these attack types. As evident by recent vulnerabilities in Intel, ARM, and AMD chips (Meltdown, Spectre, etc.[2], [3]), commercial SoCs can be compromised by exploiting architectural flaws (speculative execution, ineffective isolation of concurrent processes, etc.), design-for-test (DFT) structures [4], and fault injection (CLKSCREW, One Glitch to Rule Them All, etc. [5], [6]). Such attacks can effectively leak on-chip security assets and allow remote bypass of security/assurance mechanisms, thereby putting the integrity and confidentiality of consumer devices and even national defense systems at risk. An additional concern is that SoC design houses typically license IP cores from third-party (3P) vendors, which may include malicious backdoors and hardware Trojans that lead to denial of service, information leakage, etc [7]. These design vulnerabilities may remain undetected due to the rarity of their activation phase and the exponential input space complexity. Post-silicon vulnerabilities can impact company expenditures and reputation [8], and can be difficult to address with patches. Hence, considering all the contrasting design requirements of low power usage, diminished area, high performance, while maintaining security, *pre-silicon verification* of IPs/SoCs becomes vital.

A comprehensive and adaptable policy based framework that can track *assets* in the pre-silicon stage and verify the security of IPs is missing in the literature. The policies should be tailored to the assets considered and molded according to the specific IP design and objective. In this paper, we propose EISec, a symbolic execution-based tool that can detect information flow based policy violations and exhaustively report all possible patterns leading to a specific violation. For instance, EISec could detect a planted Trojan that causes a policy violation and then produce all possible patterns that triggers it. Further, EISec is flexible and can be adapted to applications beyond typical information flow tracking and security policies, such as cipher security quantification and analysis of IP vulnerability to physical attacks.

Our contributions can be summarized as the following:

- We present a novel Information flow security (IFS) analysis tool that can detect policy violation and produce exhaustive set of patterns responsible. The tool operates on the gate-level design at the pre-silicon stage. We also show ways to report large pattern sets concisely with byte level granularity.
- We develop a quantification analysis mechanism based on diffusion and confusion metrics for cryptographic modules. These metrics quantify the core functionalities of cryptography and enable the analysis of different cryptographic implementations. This analysis at the design stage facilitates informed decision making for hardware vendors.
- We extend EISec for Finite State Machine (FSM) analysis to incorporate specific attack models. Our methodology automatically identifies sensitive states in FSM based on potential IFS violation due to specific attack. The effectiveness is demonstrated by modeling fault injection for three popular benchmarks.

The rest of this paper is organized as follows. Section II provides background and related work on symbolic execution, ciphers and metrics, and FSM fault injection. In Section III, we discuss EISec's core framework along with how it can be used to identify vulnerabilities in ciphers and FSMs. EISec is evaluated in Section IV. Section V provides summary and plans for future work.

## II. Background and Related Work

*Symbolic Execution.* Symbolic execution is a program analysis technique that explores all path in a program utilizing symbolic

values. Concrete execution traces can explore the effects of single input values at a time. By contrast, symbolic execution can explore the complete input space effectively and provide the execution paths based on relevant conditions to explore them. Additionally, the technique also provides byte level precise symbolic representation for individual states. It is widely used in hardware verification and is perfectly suited for our exhaustive analysis technique. Symbolic execution engines utilize SMT solvers to reason the feasible execution paths. We utilized the core engine functionalities to leverage the SMT solvers and generate and solve complex queries.

Symbolic execution has been used for vulnerability detection and test case generation with high code coverage in [9], [10]. Shen et al. proposes a combination of symbolic execution and metamorphic testing for detecting hardware Trojans through Control Flow Graphs (CFG) [11]. Another work combines simulation and symbolic execution together to generate directed tests to activate typically rare branches and assignments [12]. The main disadvantage in these approaches is the limited capability to handle only certain types of vulnerable expressions in RTL (through CFG or rare nodes, respectively). Symbolic execution is specifically suited for information flow tracking (IFT) to perform taint analysis and detect malicious behavior. The process involves modeling labeled data through a system and ensuring the system adhere to security policies.

Recent work has demonstrated the effectiveness of IFT in identifying and mitigating specific vulnerabilities such as leakages through hardware Trojans, information leakage to any unclassified location, and unintended interaction of IP components of mismatched trust [13], [14]. However, existing research has not focused on exhaustive analysis utilizing symbolic execution's ability to explore the system states. In this work, we have utilized symbolic execution to detect IFS vulnerabilities and efficiently generate the exhaustive set of input patterns responsible.

*Secure Cipher.* Diffusion and confusion are two main characteristics of a secure cipher as classified by Claude Shannon [15]. These properties form the basis of implementation of cryptographic module implementation. The properties dictate how the key and plaintext inputs should drive the ciphertext output. Confusion dictates that a change in any key bit should trigger change in at least half of the output bits. The diffusion property on the other hand, marks the complexity of relation between the plaintext and ciphertext. It dictates that the change in any message bit should change at least half the ciphertext bits to hide the relationship between the message and the output. In this work, we use EISec to measure diffusion and confusion, thus allowing it to cover multiple security properties of ciphers, stemming from unintentional bugs, hardware Trojans, and fault injection.

*FSM Fault Injection.* Generally, FSMs control the overall functionalities of the hardware modules. Fault injection attacks with higher degree of accuracy (e.g., LFI and EMFI) have been particularly successful to compromise the intended controller security of circuits, i.e., altering states of the *precise* individual memory elements such as FFs and SRAM cells. The attacker can maneuver to bypass certain states or gain access to states in a manner that results in a security vulnerability, e.g., key leakage, privilege escalation, etc.

To make the cryptographic algorithms robust against fault injection attacks, several fault detection schemes based on S-box redundancy, information redundancy and temporal redundancy have been proposed [16], [17], [18], [19]. The secret key of RSA encryption is shown to be extracted by injecting fault into the FSM of the cryptographic device even if the data path is properly protected [20]. Unlike data path of the critical components, not a lot of work have been centered on protecting the controller circuit, i.e., FSM that controls the overall circuit functionality. Recent work on set-up time based fault injection [21], [22] and LFI [23], [24] to address security in FSMs assume a distinct threat model scenario where *sensitive states* must be identified manually by the designer. EISec is complementary to such approaches by providing the sensitive states automatically and verifying it based on user-defined attack model and IFS policies. Our methodology is able to model the attack effects, generate new state transitions stemming from the attack and then verify if the malicious modification causes IFS violation.

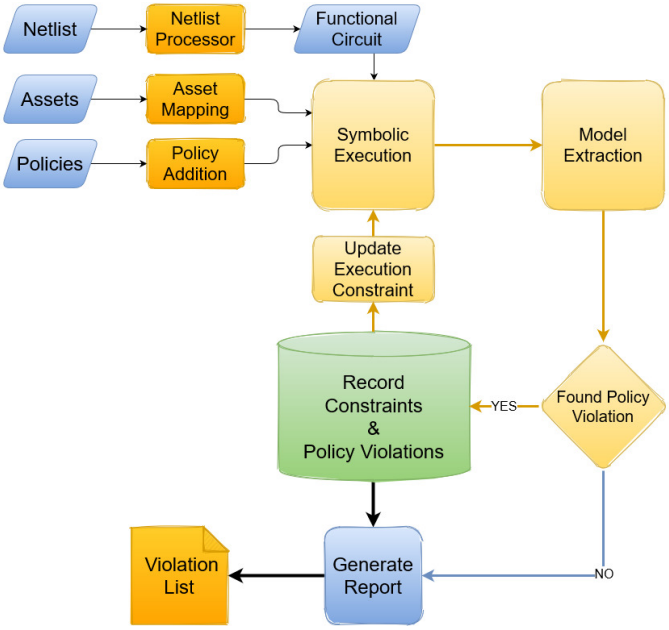


Fig. 1. High-level block diagram of EISec's core components and their interactions.

## III. Overview & Implementation

EISec provides a novel methodology of generating exhaustive set of patterns that cause information flow violations. It can also be used to quantify the strength of encryption modules and to identify vulnerable states/transitions of FSMs for specified attack models. For example, users can specify IFS policies to check whether high sensitive data is flowing to unauthorized output ports directly. Confusion and diffusion metrics along with the describing policies can check whether a threshold amount of input bits are flowing into output bits. Vulnerable FSM states can also be identified by crafting the result of the attack and then verifying the specific policies that lead to IFS violations. The core of our approach is based on symbolic execution and constraint solving through Satisfiability Modulo Theories (SMT) solver. An overview of EISec is shown in Fig. 1.

### A. IP Design Processing

EISec utilizes its own netlist processor to generate a functional equivalent circuit in C language from the design under analysis. This processing step generates the code with basic KLEE APIs and

TABLE I
SNIPPET OF NETLIST INPUT AND C CODE OUTPUT OF THE NETLIST PROCESSOR.

| | |
|---|---|
| Netlist | wire [127:0] block_reg;<br>AND2X1 U1 ( .A(a1), .B(b1), .Y(y1) );<br>DFFSR reg1 ( .D(d1), .CLK(clk), .R(r1), .S(s1), .Q(block_reg[2]) ); |
| C Code | bool block_reg[128];<br>y1 = (a1 & b1);<br>bool* block_reg_ptr2 = & block_reg[2];<br>if(s1) { block_reg[2] = 1; }<br>else if(r1) { block_reg[2] = 0; }<br>else { block_reg[2] = d1; } |

trigger functions required by our symbolic execution core. The design netlist is first converted to a technology independent intermediate representation where the library specific gate names are converted to their corresponding operations. This intermediate step makes our approach applicable to any specific technology library. The functional C code generator transforms the intermediate representation to the corresponding functional circuit. The functional circuit extracts two major aspects from the netlist: operations and memory layout. Table I shows an example of the C code generated from a given netlist. The functional circuit contains all the arithmetic and logical operations converted to their equivalent C code. The code contains a memory map of the netlist as well. The memory elements like registers are first allocated and mapped as pointers in the functional circuit to maintain a distinct location. All memory operations in the netlist are converted to C pointer references. The pointer operations allow updates to the registers to be explicitly visible to all further users of the registers. The list of input and output points are read from the netlist and passed along to the core engine for analysis. The input list along with the user provided asset list is utilized to determine the symbolic memory regions. Register sizes and corresponding pointers are used by $klee\_make\_symbolic$ function to build the symbolic memory region. The code generation process works in a single pass where all the register declarations are stored in a map and the operations are transformed to C. The register map is finally used to generate pointer declarations and allocations at the beginning of the code.

### B. EISec Framework Core

The core of EISec framework is built on the popular symbolic execution engine KLEE [25]. The LLVM bitcode generated in the processing step contains the input and output points in the design. The asset mapping phase tags the user-specified critical assets in the circuit. The policy addition phase transforms the user-specified IFS policies into the code of the design under test. The code is compiled with Clang compiler to generate LLVM bitcode which is then analyzed by the framework core.

*Symbolic Execution.* Symbolic execution efficiently compresses the complete input space into logical partitions according to the information flow in the system. Fig. 1 shows the inputs that are provided to the symbolic execution engine. We explore paths in the design to examine the system states against the given policy. The core KLEE execution engine have been modified to track information flow relevant to the assets. Noninterference properties are utilized by marking assets as high (H) sensitive and everything else low (L) sensitive at the beginning. Registers receiving data directly and indirectly from the asset are marked high sensitive as long as it is not overwritten by some low sensitive data. Indirect information flow through sandboxing and lookup tables are marked high sensitive as long as they are impacted by high sensitive data. Symbolic execution engine maintains the memory state for different paths concurrently. The execution engine also maintains the conditions required to explore individual paths. Since symbolic execution faces path explosion we have utilized under-constrained symbolic execution and modeled the environment to focus on the policy relevant portions of the design. To make SMT solver operate efficiently, we taking advantage of the caching solver in KLEE to cache the solved constraints.

*Model Extraction.* Our model generation system utilizes the path conditions and memory states of the relevant registers to generate a system model. The model encodes the information flows in registers through different paths and points in execution. The generated model contains path-specific byte-level precise information flow recorded for all registers and output ports. The path specific information contains exact control points and trigger conditions for individual execution paths.

*Constraint Solving & Exhaustive Pattern Generation.* The information flow model along with the policies are passed to the SMT solver for constraint solving. The solver verifies the policies in the context of the model. Our policies are translated to C functions into the system code. Policies are verified at a per cycle basis and at the end of the full model generation. If any policy violation is detected, the counterexample generation mechanism of the symbolic execution engine is used to generate violating patterns. Our exhaustive analysis mechanism adds generated patterns as constraints to the SMT solver. The solver iteratively updates constraints and generates new patterns. The generation process terminates when no new pattern can be found that causes policy violation. To scale this approach, a range based data is produced. We utilized the SMT solver to determine the maximum and minimum values of each byte of the input that can cause policy violation. The range based data concisely reports the possible patterns.

### C. Quantitative Analysis

Our symbolic execution based methodology keeps track of the data at the byte level. For each bit of the output, we are able to track its dependence on input bits. We utilize this capability of our tool to perform quantitative analysis of cryptographic modules and report Shanon's diffusion and confusion metrics. Both of these metrics are generated using our information flow model that indicates the relation between inputs (plaintext and key) and output (ciphertext).

*Diffusion.* The diffusion metric indicates how ciphertext outputs change when single bits of the plaintext input is altered. For a standard diffusion mechanism, one bit change in the input should impact at least half of the ciphertext bits. We utilize our information flow model with SMT constraint solving to determine the effects of each input bit value. For each input bit, both values are considered and the output model is updated by constraining the input bit to a single value. The SMT solver then performs a bitwise comparison to determine which output bits differ. We report the total number of output bits that pass the 50% diffusion criterion as well as indicate exact indexes of input bits contributing to each output bit.

*Confusion.* The confusion metric reports the complexity of relation between input key bits and output ciphertext bits. Each output bit ideally should depend on all the key bits. We utilize information model of the ciphertext to do a bitwise check for confusion characteristics. We analyze the model for each bit to determine the key bits involved in producing the ciphertext bits. Our analysis is able to identify the

total number and exact indices of the key bits involved in each output bit. We report the number of ciphertext bits passing the threshold. The threshold here indicates the number of key bits contributing to each ciphertext bit. In this paper it is set to half the number of key bits by default.

### D. FSM Analysis

Another major feature of EISec is its flexibility to handle special scenarios such as physical attacks. Specifically, we describe how it can be used to analyze FSMs in the context of fault injection. The FSM analysis engine consists of two components. First, attack model based exploration deals with the context building and secondly, the security property violation analysis identifies vulnerabilities within the context.

*Attack Model Based State Exploration.* Our approach provides an efficient way of specifying attack models such as fault injection on FSMs. The model represents FSM states and transitions symbolically as well as detailing how these can be manipulated. We utilize the symbolic execution engine core to explore the effects of the attacks and propagate their impacts to successor states. Our modeling approach is capable of triggering the faults at user-specified points and frequency. Models can be specified in C programming language. For example, we can specify Laser Fault Injection (LFI) with the following code snippet:

```
make_symbolic(&fault_next_state);
int distance = (fault_next_state ^ next_state);
int x = count1s(distance);
assume(x <= numlaser);
next_state = fault_next_state;
```

Fig. 2. Simplified code snippet of LFI model.

As shown in Fig. 2, our LFI specification starts with an unconstrained symbolic value contained in $fault\_next\_state$. Next, this value is constrained to produce the faulty $next\_state$. The constraints first establish a relation with the current $next\_state$ value through the $xor$ operation. The next two lines put additional constraints on the $fault\_next\_state$ by constraining the number of bit flips caused by the fault to be less than or equal to the number of lasers ($numlaser$). The $numlaser$ parameter is provided by the user. Finally, the concrete $next\_state$ value is overwritten with a constrained symbolic $fault\_next\_state$ value. The execution core will now spawn multiple state transitions out of the $next\_state$ based on the fault injection constraints provided. There will be state transitions for different modeled laser configurations. For example, if $numlaser$ is 2, then $next\_state$ will spawn for 1 & 2 laser scenarios independently.

*Security Property Violation.* To reduce false positives, state transitions spawned from the model based exploration are tested for security property violation. Our core engine triggers faulty state transitions alongside specified state transitions. The SMT solver determines feasible transitions in the context of the model. Once these transitions are executed by the core engine, the information flow policy violations are checked, e.g., confusion or diffusion below their thresholds. It is important to note that the policy violation found at a faulty state entails that the preceding state (specified state) is vulnerable to the modeled attack policy (fault injection).

TABLE II
IFS POLICIES AND EVALUATION RESULTS. THE DETECTED POLICY VIOLATIONS (HOLDS) ARE MARKED BY "⋆" ("✓").

| Policy | SAEAES | Subterranean |
|---|---|---|
| **P1:** { $s_i \in S$ — $s_i \models$ !(Start & Done) } | ⋆ | ✓ |
| **P2:** {$s_i \in S$, $\forall m, n, j, k$ — $0 \leq j < k < N$, $0 \leq m < n < KS$, $s_i \models ((out[j]! = key[m]) \wedge \ldots \wedge (out[k]! = key[n]))$ } | ✓ | ✓ |
| **P3:** {$s_i \in S$, $\forall m, n, j, k$ — $0 \leq j < k < N$, $0 \leq m < n < BB$, $s_i \models ((out[j]! = M[m]) \wedge \ldots \wedge (out[k]! = M[n]))$ } | ✓ | ✓ |
| **P4:** { $s_i \in S$, $\forall i$ — $(s_i = final) \models \forall (i - R < j < i)$ $s_j \models authorized$ } | ✓ | ✓ |

## IV. EVALUATION

We present the evaluation of our three proposed methodologies on popular modules. Our experimental results and findings are listed in the next three subsections. We demonstrate our results on variety of benchmarks: both block and stream cipher based encryption modules, RSA encryption, and RISC processor controller. Note that, the quantitative comparison of exhaustive analysis is not feasible due to limited existing tools ( e.g., SAT, formality based ) capability, i.e., they stop execution upon finding only one satisfying pattern unlike EIsec.

### A. IFS Violation

We have evaluated our IFS violation engine on popular cryptographic benchmarks from NIST. Table II summarizes our experimental results for the popular lightweight cryptography modules, SAEAES and Subterranean . We have crafted the general policies (**P1**-**P4**) and evaluated them for both modules. EISec is able to analyze and map the general policies in the context of these diverse modules. We describe the policies in the context of the modules below and discuss how effective our methodology is in analyzing general policies in varying contexts.

- The policy **P1** specifies for all state $s_i$ in the set of states $S$, the "Start" and "Done" signal must not be asserted to true at the same time. The signals indicate the beginning and completion of encryption operation. Any design flaw or Trojan that can assert the "Done" signal prior to encryption can potentially leak input information. We have detected violation for this policy in SAEAES. Once detected, we ran our exhaustive analysis framework to identify the input patterns that may lead to this violation; EISec has accurately reported all such input patterns responsible for the policy violation, i.e., the policy is violated for all values of the key input. A snippet of the report can be seen in Fig. 3. EISec first detected violation and started reporting individual 128 bit input patterns shown as pattern 1 to pattern 3 in the figure. Since there were too many patterns to be reported, EISec compressed the output by generating ranges of data for each byte of key. For key[0] to key[7] the range was found to

```
============ Pattern 1 ==============
key : 00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000
============ Pattern End ==============  ...
============ Pattern 3 ==============
 key : 11000000000000000000000000000000000000000000000
 00000000000000000000000000000000000000000000000000000
 000000000000000000000000000000000000
============ Pattern End ==============  ...
============== Range data ==============
Size : 128
Asset: key [0-7]  Range: [0, 255]
Asset: key [8-15]  Range: [0, 255]  ...
```

Fig. 3. Snapshot of pattern generation output of EISec.

TABLE III
QUANTIFICATION ANALYSIS STATISTICS PRESENTED AS NUMBER OF OUTPUT BITS PASSING THE CRITERIA.

| Benchmark | Diffusion Pass/# bits | Confusion Pass/# bits |
|---|---|---|
| SAEAES | 64/64 | 64/64 |
| Subterranean | 0/32 | 32/32 |

be 0 to 255, encompassing all possible values of the input for those bits. The tool generates data range for all 128 bits of the key input. In the case of the Subterranean module, no violation occured.

- **P2** and **P3** are both noninterference information flow policies. Both specify that full and partial elements of key (**P2**) and plaintext (**P3**) should not be flowing to any output port directly. The EISec generated model contains the symbolic output model for individual output bits. The model is evaluated against all asset input bits. The evaluation is done for all states in every path. EISec validated the security of these policies for both modules.
- **P4** is a path based policy. It specifies that the final state $s_i$, should only be preceded by a number ($R$) of authorized states when output becomes visible. In the case of SAEAES the policy dictates that ten ($R = 10$) $Do\_round$ states must precede the $Final\_Round$ state. Subterranean final state is preceded by round states that produce non-linearity, asymmetry, mixing and dispersion.

### B. Information Leakage Quantification

We have applied our analysis on multiple encryption modules to quantify strength in preventing information leakage. The key and plaintext inputs are marked as high sensitive for this analysis. The symbolic execution engine propagates the values and forms operation states according to the module implementation. Table III summarizes our results. The diffusion and confusion data is presented in terms of the number of ciphertext bits passing the criteria compared to the number of total output bits. The evaluation criteria for our tests was that half the key bits and message should affect each output ciphertext bit. These results can quantitatively reveal the diversity of these two benchmarks. As evident from the result, SAEAES provides good confusion and diffusion. The threshold number of key and message bits flowing into each output bit is consistent with the behavior of the block cipher based implementation of SAEAES. It takes a chunk of the message bits and produces ciphertext utilizing

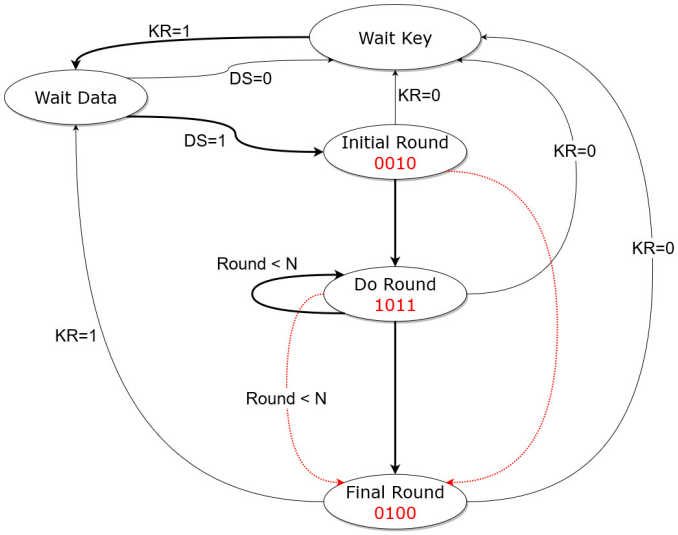


Fig. 4. Attack model based state exploration for AES. The dotted lines indicate the state transitions possible due to LFI.

TABLE IV
FSM ANALYSIS STATS

| Benchmark | Vulnerable States | Modeled | Violations |
|---|---|---|---|
| AES | 2 | 20 | 10 |
| RSA | 2 | 4 | 2 |
| RISC | 6 | 7 | 6 |

the bits and rounds of keys produced. On the other hand, we can see that the Subterranean benchmark provides good confusion, i.e., all 32 bits of output ciphertext pass the confusion criteria. However, the benchmark lacks diffusion. This behavior is consistent with the specification because it works as a stream cipher. The message bits are absorbed individually and operated on independently to produce the ciphertext. As evident from our analysis, the methodology provides parallel processing capability at the expense of diffusion complexity.

*Note that the confusion and diffusion metrics and associated policies (exceeding thresholds) can serve as more general policies that encapsulate multiple vulnerabilities, e.g., IFS violations of **P1**-**P4** in Section IV-B would all be naturally captured by these two metrics.*

### C. Automated Vulnerable State Detection

We evaluated our FSM analysis methodology by modeling laser fault injection (LFI) for AES, RSA, and RISC controller. The LFI model snippet is given in Fig. 2. The model for both AES and RSA manipulate the next state values of the registers. The concrete next state data are overwritten by symbolic values constrained by the LFI model. The SMT solver determines the feasibility of the states that can be spawned because of the model and the symbolic execution engine spawning these states. We use the number of laser faults ($x$ in Fig. 2) as input parameter [23], [24]. Essentially, we increased $x$ until every state becomes a possible next state from the current state. With this configuration, we applied our IFS policies to identify the modeled states that lead to policy violation. On AES, for 20 arbitrary models, two vulnerable states were detected: $Do\_round$ and $Initial\_round$. 10 instances of policy violations were subsequently detected. Out of

these, 9 spawned from different round index values of $Do\_round$. One new violation was detected that showed that $Initial\_round$ data is also similarly vulnerable to LFI as $Do\_round$. Similarly, for RSA there were 2 vulnerable states detected from 4 modeled transitions that led to 2 instances of policy violation. The LFI model for the RISC controller was targeted branch operations. There are 6 branch operations in RISC controller. These are branch if equal, not equal, greater than, less than, unsigned greater than, and unsigned less than. The LFI model essentially could trigger the opposite branch being taken for the operations rendering all 6 of the states vulnerable. The branch codes are all serial numbers and do not have LFI protective mechanisms.

With such FSM vulnerabilities identified in pre-silicon, targeted countermeasures can be considered by the designer to address them. For example, CAD tools can automatically incorporate logical countermeasures through duplication, partial duplication, time shifted output, etc. Alternatively, security-aware FSM encoding can be used to protect sensitive states in the FSM [21], [23], [22], [24].

## V. Conclusion and Future Work

In this paper, we have presented EISec for holistic and exhaustive information flow analysis of hardware IPs and demonstrated its ability to detect IFS policy violations. Compared to concrete execution methodology, our symbolic execution based approach can efficiently explore all $2^n$ input space for test modules and automatically extract a model of the system. Our FSM analysis engine can incorporate specific attack models to enrich analysis context, such as automated identification of sensitive FSM states and quantification of cipher security characteristics. Thus, EISec can aid hardware designers so that they make informed decisions early in the design process and avoid unwanted bugs and security vulnerabilities.

Currently, EISec works exhaustively on individual modules and effectively detects IFS violations. However, we plan to extend it for inter-module hierarchical analysis. The module interactions will reduce our environment constraining load and the tool will systematically explore relevant scenarios. Also, EISec supports a variety of state and path property specifications with IFS policies that can be specified by the user for a variety of contexts. In future, we want to incorporate automatic policy generation engine that can fully automate the policy interpretation. This process would allow EISec to be readily plugged into variety of analysis tools.

## VI. Acknowledgements

This research was supported in part by AFOSR under award ID FA8650- 20-C-1719 and Intel.

## References

[1] "Automotive and iot will drive ic growth through 2021," Nov. 2017.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, IEEE, 2019.

[4] G. K. Contreras, A. Nahiyan, S. Bhunia, D. Forte, and M. Tehranipoor, "Security vulnerability analysis of design-for-test exploits for asset protection in socs," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 617–622, IEEE, 2017.

[5] A. Tang, S. Sethumadhavan, and S. Stolfo, "{CLKSCREW}: exposing the perils of security-oblivious energy management," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1057–1074, 2017.

[6] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, (New York, NY, USA), p. 2875–2889, Association for Computing Machinery, 2021.

[7] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 1–23, 2016.

[8] "Chip problem limits supply of quad-core opterons," Nov. 2007.

[9] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, pp. 65–88, Springer, 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[11] L. Shen, D. Mu, G. Cao, M. Qin, J. Blackstone, and R. Kastner, "Symbolic execution based test-patterns generation algorithm for hardware trojan detection," *computers & security*, vol. 78, pp. 267–280, 2018.

[12] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, "Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution," in *2018 IEEE International Test Conference (ITC)*, pp. 1–10, IEEE, 2018.

[13] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1691–1696, IEEE, 2017.

[14] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.

[15] C. E. Shannon, "Communication theory of secrecy systems," *The Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[16] J. Chu and M. Benaissa, "Error detecting aes using polynomial residue number systems," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 228–234, 2013.

[17] H. Mestiri, N. Benhadjyoussef, M. Machhout, and R. Tourki, "High performance and reliable fault detection scheme for the advanced encryption standard," *International Review on Computers & Software (IRECOS)*, vol. 8, no. 3, pp. 730–746, 2013.

[18] P. Maistri and R. Leveugle, "Double-data-rate computation as a countermeasure against fault analysis," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1528–1539, 2008.

[19] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual international cryptology conference*, pp. 513–525, Springer, 1997.

[20] B. Sunar, G. Gaubatz, and E. Savas, "Sequential circuit design for embedded cryptographic applications resilient to adversarial faults," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 126–138, 2007.

[21] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "Avfsm: A framework for identifying and mitigating vulnerabilities in fsms," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

[22] A. Nahiyan, F. Farahmandi, P. Mishra, D. Forte, and M. Tehranipoor, "Security-aware fsm design flow for identifying and mitigating vulnerabilities to fault attacks," *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 38, no. 6, pp. 1003–1016, 2018.

[23] M. Choudhury, D. Forte, and S. Tajik, "Patron: A pragmatic approach for encoding laser fault injection resistant fsms," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 569–574, IEEE, 2021.

[24] M. Choudhury, S. Tajik, and D. Forte, "Sparse: Spatially aware lfi resilient state machine encoding," in *Proceedings of the 10th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–8, 2021.

[25] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.