# EXERT: EXhaustive IntEgRiTy Analysis for Information Flow Security

Jiaming Wu
University of Florida
Email:jiaming.wu@ufl.edu

Farhaan Fowze
University of Florida
Email:farhaan104@ufl.edu

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

***Abstract*—Hardware information flow analysis detects security vulnerabilities resulting from microarchitectural design flaws, design-for-test/debug (DfT/D) backdoors, and hardware Trojans. Though information flow violations can be manifested through a multitude of possible ways, prior research has only focused on detecting the existence of such vulnerabilities and no approach has been proposed to exhaustively activate all vulnerable points and reduce false positives. In this paper, we propose EXERT, a novel analysis framework that combines ATPG, SAT, and FSM analysis to detect information flow violations and perform exhaustive analysis that reports the complete set of violating input patterns for vulnerable control points. The FSM analysis, in particular, can be performed offline and helps resolve scalability limitations in prior approaches while remaining exhaustive. As proof-of-concept, EXERT is evaluated on multiple Trojan benchmarks from Trust-Hub. It detects rare Trojan triggers (activation probability $\approx$ 1.4243e-70), generates all activation patterns within minutes, and shows a 15$\times$ to 110$\times$ faster run time compared with Cadence Jasper Security Path Verification (SPV). EXERT is also applied to a larger RISC-V benchmark to identify instruction sequences that result in privilege escalation.**

## I. Introduction

As the semiconductor industry moves from vertical to horizontal supply chains,hardware security threats might occur at any point, from functional specifications to manufacturing and finally in-field usage [1]. Untrusted third parties involved in a horizontal supply chain have full access to hardware IPs and can possibly modify them during design or fabrication. Such intentional malicious modifications are often referred to as hardware Trojans [2]. Even though some parties are trustworthy. such as CAD tool providers, synthesis and optimization processes can also introduce unintentional design flaws that make hardware assets vulnerable.

Recent research has discovered multiple forms of attacks that utilize the vulnerabilities of modern processors and SoCs to perform malicious operations, steal valuable assets, and hijack control flows. For example, Spectre [3] exploits timing side-channels to speculatively perform actions that leaks private data from memory. Meltdown [4] exploits architectural design vulnerabilities to break the memory isolation and allow unauthorized access to memory data. Another concern is that intentional vulnerabilities can be added to SoCs through hardware Trojans from third-party IP (3PIP) vendors as they are not sharing the same level of trust. These hardware Trojans are designed by adversaries to be difficult to detect which results in stealthy backdoors that leak important information or act as a killswitch. These backdoors can remain undetected for years, creating huge risks for technology companies, banks, and even military defense systems.

To mitigate such issues, there is a significant need to develop scalable frameworks that detect micro-architectural flaws as well as hardware Trojans for application in pre-silicon and post-silicon stages. Further, we emphasize that such a framework should be capable of performing *exhaustive analysis*. First, exhaustiveness is the only way to *completely eliminate all flaws* and achieve a secure system. For example, exhaustive patterns search the entire state of activation vectors that possibly trigger hardware Trojans which makes a great effort in revealing stealthy Trojans. By exhaustively checking all paths and patterns, design and verification engineers can even remove false positives. Second, exhaustion can help to *better quantify how vulnerable a system is*. For example, existing tools often find a single counterexample that demonstrates a vulnerability. If an engineer knows that this pattern will never appear in real-time, it might provide the designer with a false sense of security because other patterns were not identified. Moreover, through exhaustion, one can compare the security of two systems/designs and identify the better one based on the number and/or complexity of triggering patterns. Third, recent research is exploiting AI to find vulnerabilities, generate test patterns, perform fuzzing, etc [5]. A well-known bottleneck in AI, particularly deep learning, is data. That is, a neural network may need hundreds of thousands of samples to converge. Exhaustive pattern generation can *obtain the sufficient amount of data needed to train AI-based approaches*. Last but not least, the exhaustive patterns can be used post-silicon by *real-time monitors that filter illegal input patterns/sequences* during execution. When such patterns are generated, the system can respond to avoid a security issue, e.g., through zeroization of sensitive data, system reset, etc.

In this paper, we propose EXhaustive IntEgRiTy Analysis (EXERT), a SAT and IFT (information flow tracking)-based approach that produces a complete set of I/O patterns given a netlist under test and target assets. Finite state machine (FSM) analysis and automatic test pattern generation (ATPG) method are also involved to make EXERT more scalable. Through EXERT, it is possible to generate all patterns that trigger a Trojan and cause confidentiality/integrity issues in processors. Our contributions are summarized as follows:

- We propose an exhaustive pattern generation framework based on SAT and IFT. The novelty of our approach is that we decompose sequential data-paths with fan-in cones and generate exhaustive patterns without false positives at each level during backward propagation based on IFT techniques. Also, unlike other previous techniques, our approach can work with DfT-inserted netlists.
- We utilize offline FSM extraction techniques to analyze the information flows and the input patterns driving them in FSMs. FSM extraction is performed only once based on the interaction analysis (Section III-B) of registers. This helps avoid the state explosion problem and makes EXERT's pattern generation scale.
- We experimentally run EXERT on multiple sequentially triggered Trojan benchmarks with different target designs from Trust-hub. Our proposed technique can efficiently and exhaustively detect Trojan trigger patterns for sequential Trojans with triggering probability as low as 1.4243e-70.
- We also compare the runtime of the proposed framework with the state-of-the-art commercial tool, JasperGold. Our experiments show that it is more than 15 times faster than Cadence Jasper Security Path Verification (SPV) [11].
- For further demonstration of scalability and application scope,

TABLE I
COMPARISON BETWEEN EXISTING METHODS AND PROPOSED EXERT. √ (×) DENOTES THAT A METHOD DOES (DOES NOT) POSSESS A FEATURE.

| Previous Work | GLIFT [6] | RTLIFT [7] | SecVerilog [8] | Trojan Activation [9] | IFS Verification [10] | EXERT |
|---|---|---|---|---|---|---|
| **Target** | Taint bit in netlist | Taint bit in RTL | Taint bit in RTL | Rare branch activation | Malicious observation, control point | Malicious control point activation |
| **Methods** | Taint labeling | Taint labeling | Dynamic labeling | Model checking, ATPG | ATPG-based IFT | ATPG-based IFT,SAT, FSM |
| **Scan-chain required** | N/A | N/A | N/A | √ | √ | N/A |
| **ATPG mode** | N/A | N/A | N/A | Full-sequential | Partial-scan | N/A |
| **Exhaustive** | × | × | × | × | × | √ |
| **Scalable** | Overhead | Overhead | √ | × | × | √ |

we test our framework on a 25k+ gate RISC-V processor to generate instruction sequences that lead to privilege escalation.

The rest of the paper is organized as follows. In Section II, we give the necessary background and related work. In Section III, we discuss our proposed EXERT framework and its complexity. Section IV analyzes the results from our framework on benchmarks with very complex sequential triggers. Section V provides a conclusion and directions for future work.

## II. PRELIMINARY AND RELATED WORK

*1) Information Flow Tracking (IFT)*

IFT is a well-formulated formal method for verifying security properties with confidentiality or integrity violations. Many projects have utilized IFT to build verification frameworks (e.g., GLIFT. [6], RTLIFT. [7], and SecVerilog [8], etc.). GLIFT assigns a label (tainted or not) to the target bit of a design under test and models how a single data bit propagates. Such assignments provide the foundation for modeling how a single data bit tangles with other labeled bits. RTLIFT improves upon GLIFT to provide tracking information flows in a higher level of abstraction. Another verification framework SecVerilog extends Verilog with information flow flags that support comprehensive, precise reasoning about information flows at compile time [8]. It adds a timing tag to Verilog through a type system to verify the security of the timing information flow offline. SecVerilog takes Verilog code with security labels as input. Once a target design passes verification, security labels are removed and normal Verilog code is generated. The generated Verilog code complies under information flow policies defined by SecVerilog yet without any overhead constraining code.

Existing IFT methods tend to take a qualitative approach and only enforce binary security properties. In previous work, a binary answer (yes or no) is provided regarding the flow of information between design elements. Nevertheless, GLIFT and RTLIFT also suffer from overhead in producing the shadow logic for their propagation of security tags. SecVerilog does not have such overhead as it generates information flow policy-defined RTL, but it requires the designer to hard-code information flow policies when adding labels to variables for each specific security policy. In [10], Nahiyan et al. proposed the framework of backward propagation-based IFT and utilized commercial ATPG tool Tetramax (Synopsys) [12] to generate potential patterns for propagating information flow. However, this work only generates a single pattern (which may be a false positive) and suffers from scalability issues due to sequential ATPG reliance. The summarized comparison of these works with our proposed framework is shown in Table I.

*2) Boolean Satisfiablity (SAT)*

SAT approaches are formal methods for finding an assignment of 0s and 1s to a Boolean function's variables which make it evaluate to 1. In order to utilize SAT to verify functions of a circuit, the circuit must first be transformed into formulas that can be handled by SAT solvers. SAT-based ATPGturned out to be a robust alternative to classical structural ATPG algorithms [13]. The number of unclassified faults can be significantly reduced using SAT-based ATPG. However, previous SAT-based ATPG algorithms can only be applied to combinational logic as the timing nature of sequential logic is not transformable to a Boolean representation. The robustness attributes of SAT-based ATPG algorithm is therefore not applicable to sequential pattern generation.

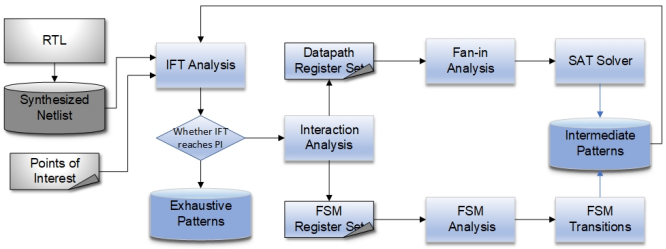



Fig. 1. Block diagram of EXERT framework.

*3) Jasper Gold Security Path Verification (SPV)*

Commercial CAD/EDA tools are also equipped with information flow tracking capabilities. The Cadence Jasper Security Path Verification (SPV) App [11] is a formal verification tool that takes the design (RTL/gate-level) to prove whether a secured area can be accessed through paths or not. It proves the leakage with mathematical certainty and a detailed waveform of how security assets are leaked if a path is detected. Based on its own path-sensitization technology, it finds paths that propagate data from sources and destinations provided by the user. However, the path-sensitization technology is not pattern-driven and requires hard-coding the policies to use it. In Section IV, we show that this is more than one to two orders of magnitude slower than EXERT.

*4) Finite State Machines (FSMs)*

For convenience, an FSM is typically represented as a directed graph where each vertex represents a state and each edge represents the transition from the current state to the next state.Transition conditions consist of an input pattern for creating a valid transition in an FSM from its initial state to final state. While FSMs can be modeled as graphs in high-level description, their physical implementation in low-level netlists is a set of registers with combinational logic and feedback nets connecting them. Tools have been developed in the prior work to convert gate-level netlist descriptions into FSM graphs [14]. We make use of these open-source tools to aid EXERT.

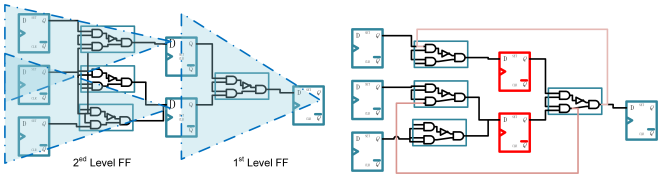
## III. EXERT FRAMEWORK

The objective of this paper is to develop a framework called EXERT, which analyzes the integrity violations of any malicious points and generates the exhaustive patterns to activate such violations. Our proposed framework is the first approach that could do so exhaustively while maintaining scalability. EXERT takes the target netlist[1] of the design and points of interest as input, then outputs the malicious points and their propagation patterns. Our target points of interest can be either Trojan triggering nets or any potential malicious

[1]Our framework analyzes designs at the gate level instead of RTL because the synthesis process may introduce vulnerabilities due to optimization of don't cares [15], DfT insertion [16], etc. Nevertheless, EXERT can analyze RTL after it is synthesized into a netlist.

**Algorithm 1** IFT Backward Propagation Algorithm

**Input:** Nets of interest, gate-level netlist;
**Output:** Register control points, Primary input control points;
1: start point FF level: $FFlevel \longrightarrow 1$
2: **for** all $Asset$ in Nets of interest **do**
3: $FaninReg \longleftarrow \{\}$
4: $FaninReg$ at $FFlevel \longleftarrow fanin(Asset, FFlevel)$
5: $FaninPI$ at $FFlevel \longleftarrow fanin(Asset, FFlevel)$
6: **while** $FaninReg$ exists **do**
7: invoke **Interaction Analysis**
8: Update Asset: $Asset$ = $FaninReg$
9: $FFlevel$ += 1
10: $FaninReg$ at $FFlevel \longleftarrow fanin(Asset, FFlevel)$
11: $FaninPI$ at $FFlevel \longleftarrow fanin(Asset, FFlevel)$
12: append $RegisterControlpoints \leftarrow FaninReg$
13: append $PrimaryInputControlpoints \leftarrow FaninPI$
14: **return** $RegisterControlpoints$, $PrimaryInputControlpoints$




(a) (b)

Fig. 2. Interaction analysis showing how IFT propagates backward through levels of registers: (a) without feedback nets, registers are classified as data-path registers; and (b) with feedback nets, the registers connected by feedback nets are classified as FSM state registers.

point that violates an integrity policy (e.g., modify an asset through an unauthorized input port). The overall EXERT framework is shown in Figure 1. It primarily consists of four modules: IFT analysis, interaction analysis, fan-in analysis and FSM analysis. Each module is explained in the subsections below.

### A. Information Flow Tracking Analysis

As shown in Algorithm 1 and illustrated in Figure 1, the first step of EXERT is IFT analysis. IFT analysis takes the nets of interest and target gate-level netlist as input, and reports the register control points and primary input control points which potentially represent the existence of information flows that violate an integrity policy. Our proposed analysis first takes the net of interest as the starting point and records all fan-in registers and fan-in primary inputs (lines 3 to 5 in Algorithm 1). When fan-in registers exist, the interaction analysis module is invoked to determine which set of registers these fan-in points belong to. Next, the nets of interest are updated with fan-in registers (lines 8 to 9) and keep looking for fan-ins in the prior level of flip-flops (FFs) (lines 9 to 11) until the complete set of control points with all their corresponding FF levels are recorded.

### B. Interaction Analysis

This module, which is invoked in IFT analysis, takes the whole netlist as input, targets the feedback nets in a design for classification. In Section II-4, we discussed that an FSM's physical implementation is a set of state registers with feedback nets connecting them (see Figure 2). The objective of interaction analysis is to classify registers into datapath and FSM register sets to better analyze them with fan-in analysis and/or FSM analysis described in Sections III-C and III-D. This allows for tracking information flows with better performance. Simply tracking information flows in FSMs inevitably results in a loop or reverse flow. The registers in Figure 2(a) are classified as data-path registers since there are no feedback nets connected. Figure 2(b) shows the feedback nets and associated FSM/state registers in red. Note the interaction analysis will only be performed once and is conducted offline. This further promotes efficiency in EXERT.

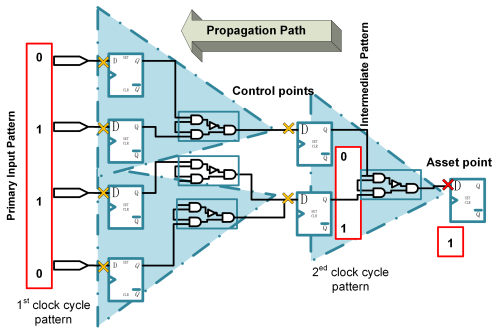



Fig. 3. Illustration of fan-in analysis. The red cross denotes the target asset point of interest. Yellow crosses denote the detected control points. The intermediate patterns match every control point detected.

### C. Fan-in Analysis

In this section, we describe the fan-in analysis module applied on the data-path registers that were identified by interaction analysis in Section III-B. It aims to generate exhaustive patterns on data-path asset points (classified by interaction analysis) with scalable effort. Our fan-in analysis is capable of tracking information flows through integrity verification and generating complete sets of patterns that propagate them. The fan-in analysis engine is based on the concept of decomposing a sequential circuit into cascading combinational circuits by tracking the fan-in cone of every asset point. As illustrated in Figure 3, the two-stage data path is decomposed into three cascading combinational cones. The inputs of the decomposed combinational circuit are the *control points* set of the output, which is the target asset point. The control point set will give us a new asset point set in the next level of registers and keep looking for fan-in cones until no more control points are detected. During the fan-in analysis, SAT engines [17] and ABC synthesis [18] are utilized to generate intermediate patterns while AIGer [19] is compresses and constrains them. AIGer is fast and scales well for the creation and manipulation of circuits between SAT solver calls [20] at every cycle.

#### 1) Pattern-Driven Information Flow Tracking

The proposed fan-in analysis module is presented in Algorithm 2 and illustrated in Figure 3. The algorithm first takes the netlist under test and target asset point (e.g., red cross in Figure 3) as inputs. Then, it looks for fan-in registers of the asset point and extracts the combinational block with fan-in registers as inputs and asset point as output using Design Compiler(Procedure I of Algorithm 2). Given the fan-in registers from IFT analysis (see Section III-B), it shows if there are information flows from fan-ins to target asset; hence, there may exist a set of patterns to propagate them (see *Intermediate Pattern* in Figure 3). Next, the extracted combinational block is transformed into CNF and fed into a SAT engine to generate the complete set of patterns that propagate the information flow (Procedure II in Algorithm 2). At this point, the list of control points that present information flows and the intermediate patterns driving them are reported. With the IFT analysis (see Section III-B) detecting control points at the next level, this procedure will keep being invoked until no more fan-in registers are detected, i.e., where our analysis reaches primary inputs. The patterns generated at every cycle make up the exhaustive pattern sequence as the IFT analysis aborts. The patterns generated here expand our IFT analysis with propagating patterns which makes our analysis a pattern-driven information flow framework. In order to keep our algorithm exhaustive but still scalable, the intermediate patterns generated at each level need compression and constraining so that invalid/reproduced patterns can be discarded (see Procedure III of Algorithm 2). This procedure is necessary to discard the false positive information flows without their

**Algorithm 2** Fan-in Cone Extraction and Analysis

**Procedure I:** Fan-in Cone Extraction
**Input:** Gate-level netlist, Asset point, fan-in register set from Interaction analysis;
**Output:** Combinational block in RTL, Asset Control Points
1: **for** all $register$ in fan-in Register set **do**
2: Create Primary Input at $register$
3: Create Primary Output at $TargetAsset$
4: Re-synthesis $\longrightarrow$ Combinational block in RTL
5: **return** Combinational block in RTL

**Procedure II:** SAT-based Pattern Generation
**Input:** RTL file, Asset point
**Output:** Intermediate Patterns;
1: Transform RTL into CNF format $\longrightarrow$ CNF expression
2: Feed to SAT solver $\longrightarrow SatisfyingPattern$
3: **repeat**
4: CNF expression $\longleftarrow$ Addclause($SatisfyingPattern$)
5: SAT($CNFexpression$) $\longrightarrow SatisfyingPattern$
6: append $IntermediatePatterns \leftarrow SatisfyingPattern$
7: **until** $SatisfyingPattern = \emptyset$
8: **return** Intermediate Patterns

**Procedure III:** Constrain and Compress
**Input:** Intermediate Patterns
**Output:** Exhaustive Primary Input Patterns, Compressed Block in AIGer, Asset Control Points
1: Classify patterns into Primary Input Control Point Patterns and Register Control Point Patterns
2: **for** Primary Input Control Point Patterns **do**
3: Stored as exhaustive Patterns at current $FFlevel$
4: **for** Register Control Point Patterns **do**
5: Construct in AIGer formate as constraints for next $FFlevel$
6: $AIGer(IntermediatePattern, Constraints) \longrightarrow$ Delete invalid intermediate patterns
7: $AIGer \longrightarrow$ Exhaustive Patterns
8: **return** Exhaustive Patterns at current $FFlevel$, $AIGer$ at current $FFlevel$

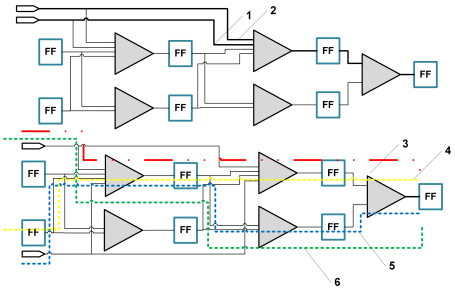



Fig. 4. State explosion with the growth of register level. At the first level, only 2 paths are detected. At the second level, 4 paths are detected. At the $n$-th level, $2^n$ paths are detected.

associated propagating patterns.

*2) Preventing State Explosion*

As the number of state variables in the system increases, the size of the system state space grows exponentially. This is called the "state explosion problem" [21]. As illustrated in Figure 4, the number of paths that represents the information flow grows exponentially with register depth, with 2 paths in the first level and 4 more in the second level. This makes our pattern-driven information flow tracking a state explosion problem as tracking information flows performs is similar in complexity as tracking paths. In order to make EXERT more scalable, we compress the intermediate patterns and paths at every level to analyze them as a whole in AIGer format. As illustrated in Figure 5, after generating the intermediate patterns, the patterns recognized in fan-in cones are compressed at every level and sequentially composed for the next level. As a pattern-driven information flow analysis framework, paths without valid propagating patterns are discarded.

With our compression and constraining in AIGer, our analysis is more scalable compared to an uncompressed one. As shown in Figure 6, since paths in the former levels are analyzed as a whole, the newly detected paths in every level of register grow linearly instead

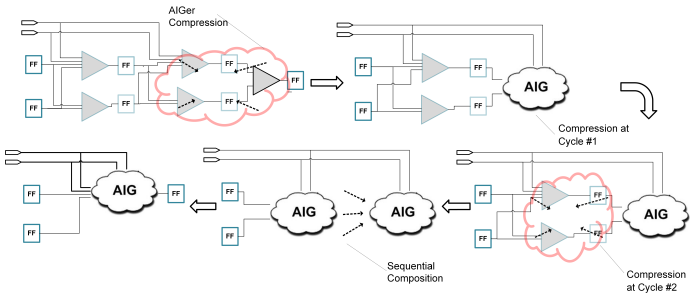



Fig. 5. AIGer Compression and constraints. At every cycle, the intermediate patterns at the control points are compressed in AIGer format. By sequentially composing in the next cycle, invalid intermediate patterns are discarded.

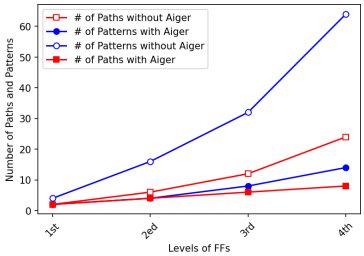



Fig. 6. Plots of path/pattern growth with FF level.

of exponentially. Let $m$ be the number of paths in a single level of registers, and $N$ as the level of registers. To search all flows, the timing complexity without compression is $O(m^N)$ while the timing complexity with the proposed AIGer compression is $O(N)$, which shows a significant reduction in complexity. The number of patterns detected is also reduced. That is, AIGer compression and composition reduces patterns which were valid in the previous level of control points but actually become invalid in the next level of registers. For example, this can be seen in Figure 3. We may detect *multiple* intermediate patterns at the first level of FFs, which represent the states of current control points. However, some of the intermediate patterns may not be valid as these states of control points may not be reachable in the next level of fan-in analysis.

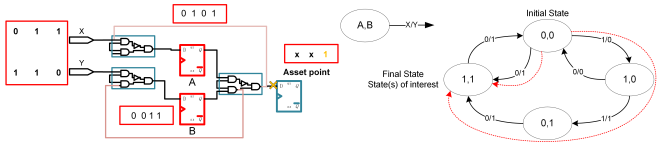
### *D. FSM Analysis*

In Section III-B, our interaction analysis module classifies registers into datapath set and FSM set. In order to modulate the information flows in feedback nets (also referred to as the FSM registers set), we model the feedback nets along with its connected registers together as FSMs. The FSM analysis aims to generate propagating patterns by analyzing a group of register as FSMs. Our FSM analysis module takes the netlist and the set of registers (see Figure 7(a)) classified by our interaction analysis (see Section III-C) as inputs, and constructs an FSM (see Figure 7(b)) that represents the behavior of the register set over the course of the operation of the netlist in a graph [14].

When an FSM register is detected during our information flow analysis, EXERT will discard some information flows by tracking its fan-ins. This is because if the register belongs to a FSM, tracking its information flows through fan-ins will inevitably result in a looping flow or path since every register in an FSM is capable of reaching the others. Thus, we analyze all the registers in the FSM to generate patterns as a whole. By modeling FSM with a state transition graph, we can find the sequences of patterns that will force an FSM to transition from an initial state to a final desired control state at a higher level of abstraction than the netlist itself . When combined with our fan-in analysis, the "initial state" is defined from the register values after reset and the "final state" is defined for

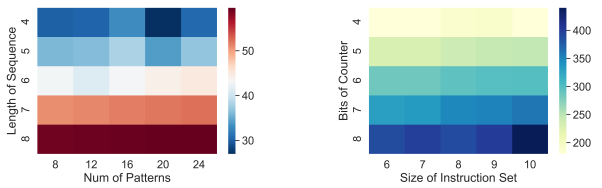
TABLE II
RESULTS OF EXERT ANALYSIS ON SEQUENTIAL TROJAN BENCHMARKS.

| Benchmark | # Gates | Trojan Trigger | Activation Probability | Trojan Payload | # Patterns | Length in Clock Cycles | Time(sec) |
|---|---|---|---|---|---|---|---|
| AES-T1400(M) | 513 | Predefined sequence | 8.8162e-39 | Leaks keys | 12 | 4 | 46.78 |
| AES-T1400(Max) | 1336 | Predefined sequence | $\leq$4.4750e-142 | Leaks keys | 600 | 400 | 4743.49 |
| PIC16-T200(M) | 2015 | Counting predefined instructions | 2.8231e-118 | Manipulates instruction register | 729 | 100 | 453.90 |
| s38417-T100 | 5431 | Comparator | 1.4243e-70 | Manipulates scan mode | 16 | 64 | 339.34 |
| MC8051-T400 | 2040 | Predefined instruction sequence | 2.7105e-20 | Manipulates flag registers | 1 | 5 | 187.68 |




(a) (b)

Fig. 7. Example of FSM analysis where two registers A,B denote the state variables and X/Y denote the two inputs. (a) Netlist level abstract of the FSM with a valid sequence of patterns; and (b) Extracted graph of the FSM with 4 states made of A,B and 6 state transitions made of X/Y .




(a) (b)

Fig. 8. Heatmap of runtime result (seconds): (a) AES-T1400(M); and (b) PIC16-T200(M).

the "control state(s) of interest" which enables the information flow in the paths/patterns. In order to make the patterns exhaustive and scalable, we always look for the shortest paths in the state transition. Any looping paths or repeated states are discarded. As illustrated in Figure 7, two paths are detected from the initial state 0,0 to the final state 1,1; thus two pattern sequences consisting of these state transitions are generated (Figure 7 shows one of the two sequences in red). The two sequences of patterns will force the asset point to be 1 in the last clock cycle while the state values in the other two clock cycles don't matter in our FSM modeling. The patterns in FSM state transitions are also constrained along with intermediate patterns from fan-in analysis as shown in Figure 1. The combined patterns are compressed and moved to the next level of IFT analysis.

Note that alternative methods for generating patterns on sequential circuits without DfT, such as sequential ATPG, search for a sequence of test vectors through the huge space of all test vector sequences. Such methods are not scalable and our evaluation with Synopysis Tetramax (not shown) terminates as sequential depth increases due to an abort limit.

## IV. EVALUATION

We applied our EXERT framework to generate exhaustive patterns that trigger Trojans on multiple Trust-Hub benchmarks [22] and that exploit escalation in a RISC-V processor. We used an Intel(R) Xeon E5-2450L 32 cores CPU with 128GB memory operating at 1.80GHz for synthesis and running EXERT. Table II summarizes theTrojan benchmarks, their sizes, and our results.

### A. AES-T1400

The first Trojan benchmark is AES-T1400. AES-T1400 Trojan's trigger circuit is composed of a finite state machine (FSM) and it triggers when a sequence of plaintexts is observed. After synthesis, the Trojan triggering module is formulated in the netlist and the triggering condition is hard to extract. By applying our framework, not only are the four plaintexts detected but also the right sequence from the netlist level of design.

In order to test the exhaustiveness and scalability of our framework, we modify the triggering circuit of AES-T1400 and increase the predefined plaintext sequences. We also incorporate more states into the triggering circuit up to eight states and create 25 modified benchmarks (denoted with '(M)' in Table II) by increasing the sequence length of predefined plaintext. Nevertheless, we modified the sequence length to 400 cycles with 600 predefined patterns to test the worst-case run time (denoted with Max), as a maximum of unrolling on AES benchmark in prior work with commercial tools is 400 clock cycles [23]. Then we apply EXERT on all these AES-T1400(M) Trojan benchmarks as well as AES-T1400(Max) and record the run time. As illustrated in Figure 8(a) and Table II, the run time shows a linear growth with the increase of sequence length, along with a minor growth trend when we increase the number predefined patterns and keep the length same. Such linear increase validates that EXERT achieves exhaustiveness while being scalable.

### B. PIC16-T200

PIC-T200 manipulates the instruction register of the PIC microprocessor and is triggered by counting the number of instructions executed. The Trojan payload of PIC-T200 is triggered by a predefined instruction set read from the microprocessor's RAM [24], which requires a RAM initialization. Therefore, in order to apply our EXERT framework on PIC-T200 benchmark, we create a 4-bit input bus where the microprocessor reads the instruction bits to bypass the RAM initialization. Note that our current framework is not able to read predefined RAM instructions and reading instructions from primary input will not affect the correctness of pattern generation.

Note that our framework only detects the shortest activating paths of the Trojan. In normal operation, the majority of instructions will not trigger the counter and create don't care states in the Trojan's activation paths. EXERT automatically discards don't care states as there is no information flow to the malicious counter. Similar to AES-T1400, we modify the Trojan triggering circuit size by changing the counter overflow threshold and length of instruction set. We created 25 benchmarks (denoted by PIC16-T200(M)) and the run time results are shown in Figure 8(b) and Table II. Once again, linear growth is observed.

### C. s38417-T100

Unlike previous benchmarks from Trust-Hub, s38417-T100 is scan-chain inserted netlist and the Trojan is triggered by a sequential counter in functional mode. The s38417-T100 Trojan payload enables the scan enable signal of a part of one scan chain in the functional mode which allows an adversary to leak internal signal values. Scan chain testing utilizes scan flip-flop to access any registers by shifting in any input vector. However, such scan chain access creates additional unauthorized information flows through a whole scan chain, resulting in false positive control points. In order to apply EXERT on designs with scan chain access while reducing false positives from scan flip-flops, we discard the information flows from the scan chains. In our analysis, we identify the scan-in port of the scan flip-flop and discard its fan-in register as a control point. Table II shows that all 16 triggering patterns are detected in less than 6 minutes.

### D. MC8051-T400

We apply EXERT on another Trojan benchmark, MC8051-T400, to further test its scalability on processor designs. The Trojan in MC8051 is triggered when a specific sequence of commands is executed and the Trojan payload disables interrupt handling after activation. Similar to PIC16-T200, MC8051-T400 also requires a RAM initialization to execute instructions. EXERT successfully detects the predefined sequence of instructions. Note that MC8051-T400 is the largest microprocessor benchmark on Trust-hub with more than 2000 gates. EXERT maintains scalability on two benchmarks with similar triggering condition while having huge differences in benchmark size.

### E. RISC-V Privilege Escalation

EXERT can also generate patterns on much larger benchmarks like RISC-V (25786 gates after synthesis, which can further test EXERT's scalability. To better establish a secure hardware platform, RISC-V is designed with four privileged levels (User, Machine, Supervisor and Reserved level) that aim to provide isolation between different components of instruction stacks. Any attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised [25] while only legal instructions could change the privileged state. As each privileged state is encoded with a Control and Status Register (CSR), we set the Machine mode CSR as our asset to apply EXERT framework since Machine mode is supposed to have the highest level of security. Our EXERT framework successfully detects more than 76 control points that potentially break the isolation policy between different levels of privileged states with 1769 patterns in 75 minutes. Note that such patterns cannot be generated by sequential ATPG as the register depth exceeds its abort threshold.

### F. Run Time Comparison with Jasper Gold SPV

In this section, we apply Jasper Gold SPV on AES-T1400 and PIC16-T200 Trojan benchmarks to compare the run-time result with EXERT. The AES-T1400 benchmarks consist of Trojan paths and Valid paths. In JasperGold SPV, we set the complete bus of key input ports as the source and one bit of the output port as the destination to verify whether a sensitization path exists for the Trojan paths. However, as a policy-driven validation tool, JasperGold requires us to hardcode the generated pattern as policy insertions so that it can keep generating different patterns. The result shows JasperGold takes 60.06 seconds to validate a path with a *single pattern*. On the other hand, EXERT can generate *all 12 patterns* in only 46.78 seconds. As for PIC16-T200, Jasper Gold requires 76.86 seconds to validate a *single pattern* on a path while our EXERT framework only takes 453.90 seconds to generate *all 729 patterns*.

## V. Conclusion and Future Work

In this paper, we proposed a novel framework called EXERT which performs exhaustive analysis to report all patterns causing information flows without the need of white-box knowledge of the IP. We have experimentally validated our framework by applying it on Trojan benchmarks from Trust-Hub as well as a large RISC-V processor benchmark. EXERT exhaustively analyzes integrity violations and provides patterns while remaining scalable.

Our initial EXERT framework utilizes FSM analysis to revolve looping information flows in feedback nets. However, there are more complex situations where multiple FSMs are detected during our offline FSM extraction. With further analysis, these FSMs can interact with each other which requires constraining them with graph transition modeling. Future work shall focus on integrating multiple FSMs, as an alternative method of constraining multiple graphs, thus reducing the complexity even further. Moreover, as mentioned earlier, our EXERT framework also has potential for real-time applications. In future work, we plan to use EXERT to build monitors that filter illegal patterns during execution.

## VI. Acknowledgements

This research was supported by AFOSR under Award ID FA8650-20-C-1719. The authors are grateful for the help provided by Dr.Radu Teodorescu and Moein Ghaniyoun on RISC-V ISA and privilege escalation.

## References

[1] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[2] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 67–70, 2011.

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

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 973–990, USENIX Association, Aug. 2018.

[5] S. Roy, S. K. Millican, and V. D. Agrawal, "Training neural network for machine intelligence in automatic test pattern generator," in *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, pp. 316–321, 2021.

[6] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner, "Gate-level information flow tracking for security lattices," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, nov 2014.

[7] 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, 2017.

[8] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *SIGARCH Comput. Archit. News*, vol. 43, p. 503–516, mar 2015.

[9] J. Cruz, F. Farahmandi, A. Ahmed, and P. Mishra, "Hardware trojan detection using atpg and model checking," in *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pp. 91–96, 2018.

[10] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *2017 IEEE International Test Conference (ITC)*, pp. 1–10, 2017.

[11] "Jasper security path verification app." https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/.

[12] "Testmax atpg:advanced pattern generation." https://www.synopsys.com/implementation-and-signoff/test-automation/testmax-atpg.html.

[13] S. Eggersglüß, R. Wille, and R. Drechsler, "Improved sat-based atpg: More constraints, better compaction," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 85–90, 2013.

[14] T. Meade, J. Portillo, S. Zhang, and Y. Jin, "Neta: When ip fails, secrets leak," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ASPDAC '19, (New York, NY, USA), p. 90–95, Association for Computing Machinery, 2019.

[15] 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, 2016.

[16] 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, 2017.

[17] "The glucose sat solver." https://www.labri.fr/perso/lsimon/glucose/.

[18] "Abc: A system for sequential synthesis and verification." http://people.eecs.berkeley.edu/~alanmi/abc/.

[19] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[20] "pyaiger: A python library for manipulating sequential and combinatorial circuits.." https://github.com/mvcisback/py-aiger.

[21] Z. Xin-feng, W. Jian-dong, L. Bin, Z. Jun-wu, and W. Jun, "Methods to tackle state explosion problem in model checking," in *2009 Third International Symposium on Intelligent Information Technology Application*, vol. 2, pp. 329–331, 2009.

[22] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 471–474, 2013.

[23] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015.

[24] "Pic16f84a data sheet." https:ww1.microchip.com/downloads/en/DeviceDoc/35007b.pdf.

[25] "The risc-v instruction set manual." https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.