CAD-Base: An Attack Vector into the Electronics Supply Chain

KANAD BASU, New York University, USA
SAMAH MOHAMED SAEED, City University of New York, USA
CHRISTIAN PILATO, Politecnico di Milano, Italy
MOHAMMED ASHRAF, New York University, Abu Dhabi, UAE
MOHAMMED THARI NABEEL, New York University, Abu Dhabi, UAE
KRISHNENDU CHAKRABARTY, Duke University, USA
RAMESH KARRI, New York University, USA

Fabless semiconductor companies design system-on-chips (SoC) by using third-party intellectual property (IP) cores and fabricate them in off-shore, potentially untrustworthy foundries. Owing to the globally distributed electronics supply chain, security has emerged as a serious concern. In this paper, we explore electronics computer-aided design (CAD) software as a threat vector that can be exploited to introduce vulnerabilities into the SoC. We show that all electronics CAD tools — high-level synthesis, logic synthesis, physical design, verification, test and post-silicon validation– are potential threat vectors to different degrees. We have demonstrated CAD-based attacks on several benchmarks, including the commercial ARM Cortex M0 processor [1].

CCS Concepts: • **Hardware → Electronic design automation**; • **Methodologies for EDA → Best practices for EDA**;

Additional Key Words and Phrases: Electronic Design Automation, Hardware Security, Computer-aided Design

ACM Reference Format:
Kanad Basu, Samah Mohamed Saeed, Christian Pilato, Mohammed Ashraf, Mohammed Thari Nabeel, Krishnendu Chakrabarty, and Ramesh Karri. 2019. CAD-Base: An Attack Vector into the Electronics Supply Chain. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1, Article 4 (May 2019), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Reduction in feature size has resulted in an increase of chip fabrication costs. The cost to set up a fabrication facility for a new generation of transistors is 15% higher than the preceding one [2]. In order to improve cost, semiconductor design companies have relinquished control of fabrication facilities [3]. These fabless design houses ship system-on-chip (SoC) projects to a third-party

This work is partially supported by NSF award number 1526405.

Authors' addresses: Kanad Basu, New York University, 2, MetroTech Center, Brooklyn, NY, 11201, USA, kb150@nyu.edu; Samah Mohamed Saeed, City University of New York, USA, samahsaeed88@hotmail.com; Christian Pilato, Politecnico di Milano, Italy, christian.pilato@polimi.it; Mohammed Ashraf, New York University, Abu Dhabi, UAE, mohammed.ashraf@nyu.edu; Mohammed Thari Nabeel, New York University, Abu Dhabi, UAE, mohammed.nabeel@nyu.edu; Krishnendu Chakrabarty, Duke University, USA, krish@duke.edu; Ramesh Karri, New York University, USA, rkarri@nyu.edu.

1084-4309/2019/5-ART4 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

foundry for fabrication[1]. Along with fabrication, many design houses have started to lease some generic designs from third parties known as third-party IPs [4]. Although this reusable IP paradigm is economically viable, it introduces security risks in the SoC, since the design house loses its control over some critical steps in the process.

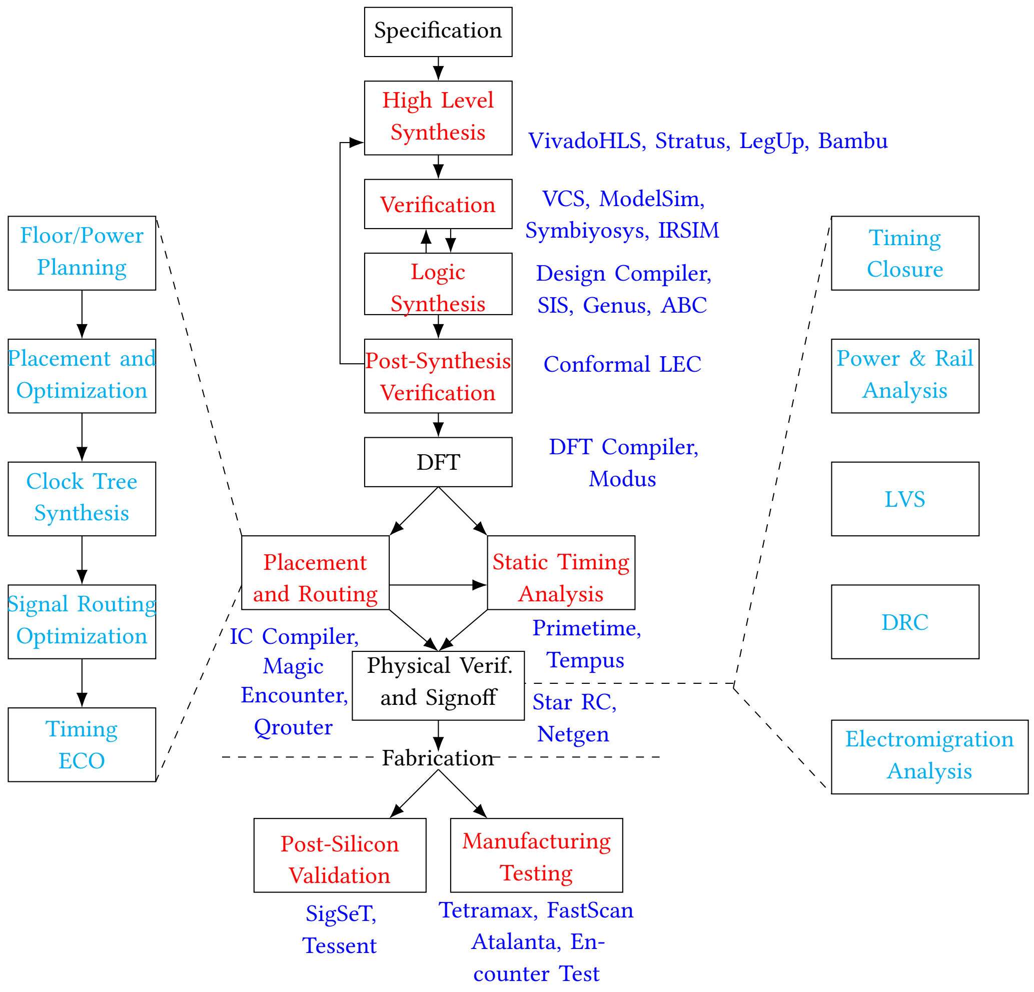


Fig. 1. Industrial-scale SoC design flow. The CAD tools used at the various stages of the design flow are in dark blue. The post-fabrication test and validation tools like Tetramax and SigSeT are used at pre-fabrication stage as well to determine the test patterns/trace signals to use. Since these patterns/signals are used after manufacturing, they have been noted post-fabrication. The design stages where we proposed attacks are in red. The sub-processes of a design stage are in cyan.

[1]Many popular semiconductor companies including Qualcomm, Advanced Micro Devices, Broadcom, NVidia are fabless.

1.1 Computer Aided Design of Complex SoCs

Computer-aided Design (CAD) tools are used in the SoC design process of translating an idea to silicon. The steps in a SoC design process are shown in Figure 1. Commercial and open-source CAD tools are shown in blue beside each major step.

The CAD tools span the full spectrum: High-Level Synthesis → Verification → Logic Synthesis → Placement and Routing → Static Timing Analysis → Post-Silicon Validation → Manufacturing Testing shown in red in the representative commercial SoC design flow in Figure 1. Besides the CAD software, the CAD tools use a large number of libraries. A logic synthesis tool uses libraries to map a generic gate-level design to a transistor-technology-dependent gate-level design. These libraries have technology-specific information regarding resistance, capacitance, timing and other characteristics of the gates used. The "lsi_10k" library used by the Synopsys Design Compiler for logic synthesis has 177 gates, each of which has multiple input/output pins and physical and timing parameters.

CAD tools are large software artifacts. Table 1 shows the sizes of the binaries of popular commercial CAD tools as well as the lines of code for open-source CAD tools. CAD tools are complex and incorporate a wide range of features.

1.2 Risks of Modern Electronics Supply Chain

A third-party foundry can introduce vulnerabilities in a design in the form of Hardware Trojans. A hardware Trojan is a stealthy, malicious modification in a design that causes the design to perform unexpectedly and may create security breaches like loss of information [5]. The risk of hardware Trojans was first published in a report by the Department of Defense [6]. Since then, extensive research has been performed in both academia and the semiconductor industry in order to secure a circuit against Hardware Trojans.

Hardware Trojans are crafty. The simplest form of Hardware Trojan can be a dopant-level Trojan [7]. A dopant-level Trojan modifies the dopant ratio on the input pin of a victim transistor, thus effectively tying it to a logic-0 or logic-1 value, modifying the design functionality. This corresponds to a stuck-at-1 or stuck-at-0 condition, which can be conveniently detected using traditional manufacturing testing methodologies. An ideal Trojan should not manipulate the normal functionality of the SoC and hence remain undetected even during post-silicon validation and manufacturing test. Trojans can be inserted at various steps of a design flow, depending on which phases are outsourced to third parties. Existing research mainly investigates Trojan insertion in a synthesized netlist or during fabrication.

Other complications with outsourcing fabrication are IP theft by stealing the optical IC masks [10], over-production of ICs [11] and counterfeiting [12]. Third-party professionals specialized IPs e.g., crypto IPs can introduce vulnerabilities [4]. Side-channel can help steal sensitive information like private key from crypto-cores [13].

There is limited research exploring the role of CAD tools in introducing vulnerabilities into an SoC. This paper will explore how malicious CAD tools can introduce them. For example, library parameters can be altered by a malicious CAD tool. Intentionally changing the timing information of a logic gate can incorrectly identify the slowest path in the design resulting in timing violations. Not all vulnerabilities are effective. Some of them will affect the functionality and hence, can be detected using pre-silicon verification, manufacturing test, and post-silicon validation methods.

1.3 Contributions of This Study

This is the first comprehensive study of CAD-based attacks. Contributions of this study are four-fold:

(1) CAD-based attacks that employ tools spanning all steps in an SoC design.
(2) Proof-of-concept demonstration of attacks on practical designs (e.g. ARM Cortex processor).

Tool Function	Synopsys		Cadence		Mentor		Open Source	
	Tool	Binary (Kb)	Tool	Binary (Kb)	Tool	Binary (Kb)	Tool	Lines of code
Verification	VCS	961	NCSim	25853	ModelSim	5847	SymbioSys	1492
High-Level Synthesis	Synphony		Stratus	297442	Catapult		Bambu	315642
Logic Synthesis	Design Compiler	292562	Genus	102562	Precision		Yosys	821466
Waveform Viewer	Custom Explorer	38429	VIVA		Tanner		GTKwave	236349
Circuit Simulator	HSpice	42454	Spectre	1991	Eldo		NGSPICE	577441
Place and Route	IC Compiler	400798	Encounter	349528	Xpedition		Qrouter	20380
							Magic	291173
Timing Analysis	Primetime	109749	Tempus		Time-it		OpenTimer	120982
Parasitic Extraction	Raphael	118234	Quantus		Calibre		FasterCap	21514
Analog Simulator	Taurus	7961	Virtuoso	302523	Xpedition		IRSIM	32304
Testing	Tetramax	36648	Encounter Test	68838	FastScan	174550	Atalanta	30294
Post-Silicon Validation							SigSET	4000

Table 1. Codebase of major commercial and open-source electronic CAD tools. The sizes of the compiled binaries and the lines of code are reported for commercial tools and open-source tools respectively. We did not have access to some of the commercial tools and hence, their code size is not reported. As one can glean, Synopsys, Mentor and Cadence are the three major commercial CAD tool vendors. CAD ecosystem is vast and vibrant. Please check the DAC exhibits list [8]. These small CAD tool companies develop point solutions and are occasionally acquired by the big three. Post-silicon validation tools are missing from this table because the only commercial post-silicon validation tool available is Tessent by Mentor Graphics. SigSeT [9] is a free post-silicon validation tool whose binary has a size of 188551 bytes. Since it is not open-source, we couldn't report the lines of code. The tools we modified in this paper are marked in blue.

(3) CAD-based attacks that can be spotted in commercial SoC design flows.
(4) High-impact, practical, and scalable CAD-based attacks which bypass detection.

This paper does not consider attacks by a malicious OS that can modify CAD binaries to launch attacks. Similarly, it does not consider malware that modifies CAD tools to launch attacks. The list of attacks presented in this paper are comprehensive, yet not exhaustive. A future researcher might use this work as a reference to design more attacks and introduce vulnerabilities using current and future CAD tools.

1.4 Paper Roadmap

A comprehensive picture of the commercial CAD flow is presented in Section 2. Prior works are discussed in Section 3. Exploration of malicious CAD during high-level synthesis, verification, logic synthesis, placement and routing, static timing analysis, post-silicon validation and manufacturing testing steps are taken up in Section 4. Each sub-section discusses the considered CAD tool, outlines the attack and demonstrates feasibility via experiments on benchmarks like ARM Cortex M0 processor and concludes with key-takeaways. ARM Cortex M0 is an ARMv6-M architecture-based processor, which is used in commercial devices like Infineon XMC1000, Toshiba TX00, ST Microelectronics STM32 F0, etc. Detailed specification of the processor is present in Table 2. Currently, it is available only in Verilog.

ISA	SRAM Size	Pipeline	Interrupts	AHB	UART	Combinational gates
32-bit	64KB	3-stage	32	ARM AHB-Lite	present	23260

Table 2. Specification of ARM Cortex M0 processor.

Each attack is evaluated in terms of **practicality** (*how feasible is to launch the attack standalone?*), **scalability** (*how scalable is the tool modification to launch the attack?*), **stealth** (*how difficult is* *to detect the attack?*) and **impact** (*what is the impact of the attack?*). We discuss the attacks in the context of checks and balances by upstream/downstream tools from diverse CAD companies. Section 5 concludes the study.

2 BACKGROUND

2.1 SoC Design Flow

We will look at the steps in an SoC design illustrated in Figure 1.

2.1.1 Specification. The system-level design starts with a high-level description of the SoC. For example, when designing a network router SoC, the system-level design tasks include planning the router functionality, determining the features supported, and finalizing the algorithms implemented at a high-level. If the network router design requires compression, the compression algorithm is determined. However, the implementation details, like, whether it should be realized in hardware or software is not. High-level programs like C or MATLAB are employed for this purpose.

2.1.2 High-Level Synthesis. This step translates the high-level specification (described in software-like languagees, like C, MATLAB) into a concrete hardware description (described in hardware description languages, like Verilog or VHDL). This step explores hardware-specific features like parallelism, timing, and synchronization. Manual implementation and optimization requires highly-trained designers. Hence, automated High-Level Synthesis (HLS) methods are popular [14]. HLS tools use state-of-the-art software compilers (e.g., GCC or LLVM) to extract an intermediate representation. Based on a set of input constraints and the target technology libraries, the HLS tool performs operation-to clock cycle scheduling, operation-to operator resource binding, interconnection binding and controller synthesis. Popular HLS CAD tools include Xilinx Vivado HLS, Cadence Stratus, and Mentor Catapult. Each of them starts from a different high-level language and targets specific technologies (either FPGA or ASIC) [14]. This is the first work that explores how a malicious HLS tool can be used to introduce vulnerabilities in a design.

2.1.3 Verification. Once the hardware is modeled using a high-level language, one needs to verify that the functionality is preserved during HLS. There are two approaches for verification: Simulation-based and Formal. In simulation-based verification, the design is simulated on a large number of inputs. There are an abundance of fast, accurate commercial simulators like Mentor Graphics ModelSim and Synopsys VCS. Co-simulation approaches are normally used for verification between high-level specifications and RTL descriptions. FPGA-based hardware emulators are also used to accelerate simulation [15].

Formal verification, on the other hand, formally proves the model of a system against a formal specification. Formal verification either proves absence of bugs or generates counterexamples identifying the bugs. It does not use and hence does not depend on the input vectors and can find bugs faster than simulation-based methods. Equivalence Checking, Model Checking and Theorem Proving are the popular verification techniques. Appendix A explains more details on formal verification.

Formal verification techniques result in state-space explosion and hence, are unsuitable for large designs. There is a large memory requirement to verify all states correctly. Recently, techniques like bounded model checking that perform model checking for a limited time period or on a constrained state space by unrolling the design a specific number of times have been proposed [16]. Although this allows scaling to large designs, it offers limited guarantees.

As seen in Figure 1, Verification is performed in two phases, once after HLS and once after logic synthesis. Generally, simulation-based verification is used after HLS and equivalence checking between the original RTL and the generated gate-level netlist is used to ensure equivalence after logic synthesis. Although verification CAD tools have been used to detect security vulnerabilities, malicious applications of them have not been explored before.

2.1.4 Logic Synthesis. Logic Synthesis transforms an RTL description to an optimized gate-level netlist of combinational and sequential gates from a standard cell library. A typical cell library comprises of AND, OR, D Flip-Flop and other basic gates annotated with parameters like delay and power. The four main steps of logic synthesis are: logic simplification, logic synthesis, technology mapping and optimization. Popular CAD tools used for logic synthesis include Synopsys Design Compiler and Cadence Genus. Malicious logic synthesis tools have been used by [17, 18] to create backdoor in the design. However, in this paper, we explain how those attacks are vulnerable as they can be detected by post-synthesis verification/validation tools like Logical Equivalence Checker.

2.1.5 Placement and Routing. This step determines how the various gates can be arranged in a die and the wires between them can be routed. Placement and routing (PNR) CAD tools help with these tasks while optimizing the wirelength and area of the die. A PNR tool accepts the gate-level netlist, design constraints, technology information (Library Exchange Format or lef), macro and standard cell physical and timing libraries (Timing Library File or tlf) as input and carries the design through floorplan, placement, optimization, clock tree synthesis and routing to produce a physical layout. The layout obtained is subjected to design checks and is readied for tape out to the chosen foundry for fabrication. Synopsys IC Compiler and Cadence Encounter are popular PNR CAD tools. This is the first time a malicious PNR tool is used to introduce design vulnerabilities.

2.1.6 Timing Analysis. This step identifies critical paths in a design in order to determine the clock frequency at which the circuit will operate. The input to a timing analysis tool are the synthesized gate-level netlist and the timing cell library. The timing cell library contains timing information for the various building block cells used in the design. The timing analysis CAD tool identifies possible paths with small delay defects[2], which can be used by the manufacturing test tools to detect those defects. Commercial tools include Synopsys Primetime and Cadence Tempus. Till date, there has not been any research on utilizing malicious timing analysis tools.

2.1.7 Post-Silicon Validation. This step detects functional faults after the SoC is fabricated. Due to a reduction in feature size and time-to-market pressures, a lot of functional errors escape the pre-silicon verification. Post-silicon validation catches these errors. Although post-silicon validation operates on a manufactured SoC like manufacturing test (Section 4.9), the former targets functional errors, while the latter targets electrical errors.

The primary problem with post-silicon validation is the limited observability of internal signal states [19]. Scan-chains, which are used for electrical testing can not be directly employed since post-silicon validation must be operated at-speed [20]. Some recent design-for-debug techniques like embedded trace buffers improve the circuit observability by storing internal signal states. The untraced signal states are restored. Mentor Tessent is a commercial post-silicon validation tool. The attacks we propose in this paper using malicious post-silicon validation tool are completely novel.

2.1.8 Manufacturing Testing. Once the chip is fabricated, manufacturing testing is performed to detect electrical and manufacturing defects introduced during chip fabrication. Manufacturing testing can be either static or dynamic. Static testing is used to identify faults which modify values at particular nodes in the design. These include stuck-at faults. On the other hand, dynamic testing is used to detect delay faults along paths induced during manufacturing. Automatic Test Pattern Generation (ATPG) tools are used in both cases. The ATPG tool accepts the netlist as input and identifies the possible fault locations and generates patterns to activate those faults (if they exist)

[2]Small-delay defects are timing faults called by delays which are extremely small compared to the system clock cycle, about 10-25%.

and propagate their effects to an observable output. Tetramax by Synopsys and Encounter Test by Cadence are popular commercial ATPG tools. We, for the first time, explain how malicious ATPG tools can impair a design flow.

3 PRIOR WORK

Ref.	Hardware Trojan		
	Approach	Phase	Function
[21]	Attack	H/W Description Lang.	leak information
[22]	Defense	H/W Description Lang.	modify function
[23]	Defense	H/W Description Lang.	modify function
[24]	Defense	Register Transfer Level	modify function
[25]	Defense	H/W Description Lang.	modify function
[2]	Attack	Layout	modify function
[26]	Attack	Layout	leak information
[27]	Attack	Register Transfer Level	leak information

Table 3. State-of-the-art Hardware Trojan techniques.

Hardware Trojans are a risk to ICs [28]. A Trojan is excited by a rare condition. The condition which switches on a Trojan is known as the *trigger*. Since the trigger conditions are rare events, they are not exposed during manufacturing test and generally manifest after continued in-field operation [29]. Trojans can be either digital or analog depending on the Trigger condition. While digital Trojans are activated by Boolean conditions like an input pattern, analog Trojans can be activated by physical parameters like temperature and delay [30]. A digital Trojan ç classified as combinational or sequential [23]. A combinational Trojan is activated when specific input values arrive at a particular gate. For example, if an AND gate in the design has very low probability of being turned on, a 1 at the output of the AND gate can serve as a trigger for a combinational Trojan. On the other hand, sequential Trojans are activated only when the inputs follow some sequence over a predetermined number of clock cycles. A simple counter can serve as a sequential Trojan. The effect of a Trojan, the *payload*, can be many: functional failure (either through errors or denial-of-service) and spilling confidential information through power and timing side-channels [26]. A classification of Hardware Trojans based on Trigger and Payload is presented in [31].

An SoC is secured against a Trojan by either of the two approaches: (i) using proper Design-for-Security (DfS) mechanisms that can either make Trojan insertion challenging or cause the inserted Trojan to be easily exposed to manufacturing testing or post-silicon validation tests, and (ii) using special test patterns that are crafted to unmask a Trojan. Popular DfS techniques include insertion of dummy cells in the underutilized design space and key obfuscation-based circuit design. However, DfS methodologies incur a substantial area overhead [23]. While test patterns are effective in detecting combinational Trojans, they aren't suitable when the Trojans manifest after a long time duration in the field, like, sequential Trojans. Apart from specially crafted test patterns, a Trojan inserted in a circuit can be detected by side-channels [32] or using special sensors [33].

Identifying unused or nearly unused logic for detecting stealthy hardware backdoors has been recommended by various researchers. The two most prominent techniques are UCI [22] and FANCI [34]. However, these methods are not impeccable. Struton et al. [35] have demonstrated that UCI can be defeated in building stealthy hardware. FANCI, on the other hand, flags 1-8% of the all nodes in a circuit as suspicious, which might not be feasible to analyze for large industrial circuits [36]. Also, FANCI has been observed to report suspicious nodes even when the circuit is Trojan free [36]. Moreover, both UCI and FANCI operate on a HDL or netlist representation of a design. As explained in Section 4.4.2, HDL or netlist vulnerabilities, if introduced by the Logic Synthesis CAD tool, can be easily detected in post-synthesis verification phase. These techniques are not applicable to the attacks using Verification, PNR, STA, Testing and Post-Silicon Validation tools.

Existing Trojan insertion methodologies operate either at the front-end (HDL-level) or back-end (layout-level). Table 3 illustrates some of the prevailing Trojan insertion approaches suggested in the existing literature. The second column specifies whether the research suggests an attack or a defense mechanism. If the paper suggests a defense mechanism, we have reported the attack model the authors consider to defend against. The last two rows describe a special scenario called multi-level attacks, where two or more participants in the design processes collude to introduce Trojans.

CAD tools present another threat to introduce vulnerabilities into a design [27, 37]. CAD tools are essential to any design flow in order to bring up a design from a concept to silicon. Application of malicious logic synthesis tools to introduce Trojans have been reported in [17, 18, 38]. All of these approaches consider a design whose Finite-State-Machine (FSM) is not completely specified. The malicious tools introduce undefined transitions in the FSMs, which act as Trojans. These extra transitions are triggered by rare conditions. Once triggered, they may lead a circuit to a completely different state, whose functionality is radically different. However, these threat models have limitations and will be explained in Section 4.4.2. Although these attacks can survive stand-alone, they will be easily detected in a complete tool flow during post-synthesis verification. A completely specified design approach to counter the malicious synthesis tools has been presented in [39]; however, as indicated by [38], complete specification of an FSM incurs a considerable area overhead. Application of CAD tools for hardware metering in order to prevent a malicious foundry from overproduction has been examined by Koushanfar et al. [40].

4 CAD-INDUCED SECURITY VULNERABILITIES

In this section, we will show how malicious CAD can be used during design to launch attacks. Front-end and back-end tools can be evaluated with the same benchmarks since they are all based on RTL designs specified in HDL. However, evaluating HLS requires a C/C++ benchmark.

4.1 Threat Model

Malicious CAD tools can be weaponized in two ways – commercial and the nation-state. Leading CAD companies are developing their own IP portfolios in competition with fabless design companies. If a design house creates an IP competing with the one being developed by the CAD company, both parties have an incentive to sabotage the competition's IP. A malicious CAD could be used to sabotage the competition's IP. Since total function failure is detected and undermines the trust across all design companies, surreptitious performance degradation caused by inappropriate optimizations, crosstalk faults and IR-drop can stealthily hurt the fabless design house's IP and reputation.

Malicious CAD can be part of nation-state attack arsenal. CAD tools developed in one country can be used to build SoCs in another. The former can force the CAD companies to ship malicious CAD tools using national interest arguments. These malicious CAD tools can insert backdoors without the knowledge of the design houses in the second country noticing it. Once a backdoor is inserted, those SoCs are almost owned and controlled by the first country [31].

A CAD tool is generally agnostic of the design its processing. However, it can operate in either of two ways:

- **When a rogue employee is present in the design house:** In general, R&D engineers of EDA CAD tools develop a lot of backdoors for debug purposes, which are unknown to the design house. These backdoors are present in the executable sent to the design house, but can be activated only by a trigger, known to the EDA company. The malicious CAD company can create such a backdoor, which will introduce these vulnerabilities when turned on. If a rogue employee is present in the design house to whom this information is transmitted, he doesn't need to modify his scripts. He can just activate this backdoor to implement the attacks.

- **When no rogue employee is present in the design house:** In this case, the EDA CAD tool can automatically turn on the backdoor trigger by parsing the design it processes. For example, a malicious HLS tool may be set with a capability to parse the input C code and identify specific patterns to compromise, like rounds of an AES program. If yes, the HLS tool can automatically turn on the trigger that introduces vulnerabilities.

4.2 Malicious High-Level Synthesis

High-Level Synthesis (HLS) generates an RTL description starting from a high-level language (e.g., C, C++, Matlab). An important step during HLS is scheduling, which determines which operations are executed in each clock cycle. This has a twofold effect: it determines the latency of the design and imposes constraints on the resource binding step, where hardware resources are allocated to processes. Operations executing in the same clock cycle require distinct hardware resources. They can be reused by operations executing in other clock cycles. Not all hardware resources are used in every clock cycle. This gives an opportunity for a malicious HLS tool to introduce a stealthy hardware Trojan by reusing these resources to implement a back-door. Checking the equivalence between RTL designs and high-level specifications is an open problem [41]. It is hard to detect hardware Trojans that are activated only with rare input conditions using simulations.

4.2.1 Attack. We present two possible attacks. Consider the fragment of C code shown in Figure 2 representing the body of a digital filter. The corresponding scheduled Data Flow Graph (DFG), in Figure 3(a), uses two adders and two multipliers to execute operations in parallel in states S3 and S4. In the other states, only one resource is used.

```
r1 = x1 + t2;
r2 = r1 * a11 + t2;
r4 = r2 + t1;
r3 = r4 + a21 + t1;
y1 = c * (r1 + r2);
t1 = r3;
t2 = r3 + r4;
```

Fig. 2. Example of C code extracted from a digital filter specification.

A malicious HLS tool can implement a hidden function to corrupt the register storing the output value *y1*, as shown in Figure 3(b). This backdoor attack implementation reuses the functional units unused in each clock cycle to implement additional fake operations (shown in red in Figure 3(b)), without introducing any additional FSM states besides the ones already defined by the scheduled DFG. Therefore, the resulting implementation has no performance overhead since the critical path is given by the original function being executed, while a few more registers (in red) maybe needed to store intermediate values of the hidden backdoor function. When all resources are used in every clock cycle, the backdoor introduces new resources, maintaining the same delay. The FSM requires minimal changes (≈10% overhead in this small example) to produce the additional signals for multiplexers and registers. The overhead further reduces when the size of the design is large relative to the size of the backdoor. These attacks are hard to detect because they reuse existing design resources. Therefore, methods like FANCI are not able to detect *unaffecting* or *always-affecting* dependencies [42]. In fact, corrupting values can effectively affect the outputs

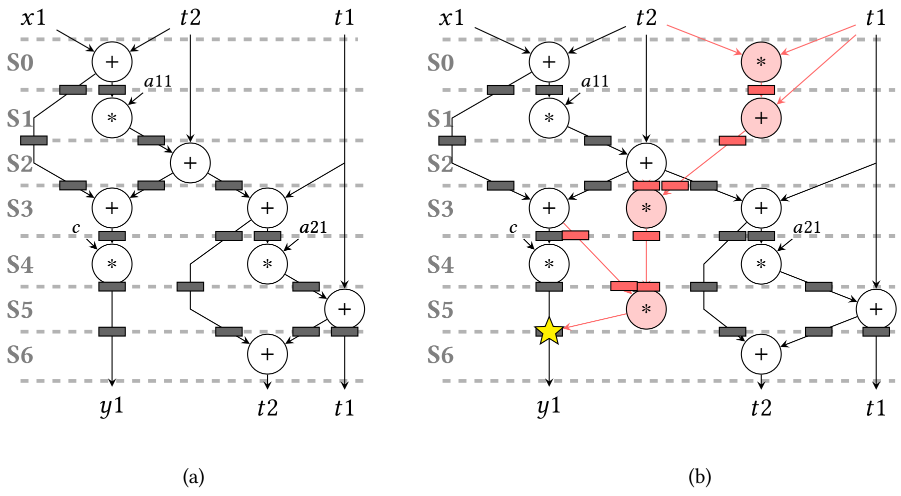


Fig. 3. Manipulating the scheduled DFG can introduce a hardware Trojan that corrupts the output $y1$. (a) Scheduled DFG. (b) Scheduled DFG with a hidden trojan.

when the Trojan is triggered through the output registers. Automatic analysis methods are not able to devise which is the logic path that generate the correct results.

The second attack entails manipulating the Finite State Machines (FSM). The malicious HLS tool modifies the interactions between the hardware resources and the FSM to change the control flow in specific cases. For example, reduced-round attacks are used to compromise the security of symmetric cryptographic algorithms [43, 44]. Consider a hardware module implementing the Advanced Encryption Standard (AES) algorithm that uses a 192 bit-key. The number of rounds for the AES is controlled by a counter, while the round microarchitecture is implemented only once and reused as many times as needed. To secure AES-192, one needs to execute a minimum of 12 rounds. However, the key can be recovered if the number of rounds used is 8 or less [43]. The HLS tool can insert a backdoor to exit from the loop earlier to execute less rounds when the Trojan is activated. There are many different ways to implement this malicious attack, including modifications to the comparator. However, all approaches are conceptually equivalent to that shown in Figure 4, where the component executes less rounds in the compromised version.

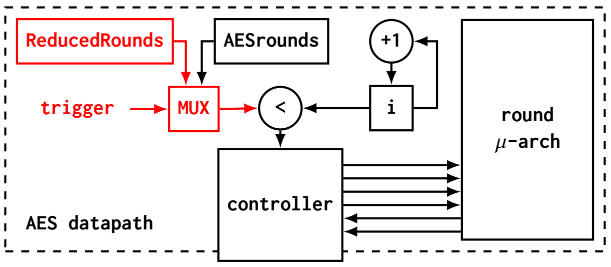


Fig. 4. Manipulation of the AES-192 FSM implementation that reduces the number of rounds.

4.2.2 Experiments. We use Bambu [45] HLS tool to insert Trojans in benchmarks from the CHStone [46] and MachSuite [47]. These benchmarks are specified in C and represent third-party

IPs that a designer may want to include in his/her design. We use these benchmarks to evaluate the backdoor attack, while the AES cryptographic benchmark is used to evaluate the FSM manipulation. Since a high-level description of the ARM Cortex processor in C, MATLAB or any software language was not available, we could not use it for our HLS experiments.

A set of 20 random inputs is applied for each design to generate the golden output values in software. Bambu is used to generate the corresponding Verilog descriptions and testbenches. These are then validated by comparing the simulation outputs with the golden ones. Next, the designs are Trojaned using the malicious Bambu to add backdoor functionality that is activated with a predefined input sequence. When activated, the Trojan provides meaningful but wrong results as output, endangering the system where the components are used.

The resource overhead has two parts: the trigger circuitry to determine the Trojan activation and the payload to provide the malicious function. While the first is fixed and it is determined by the input sequence to detect, the second must be designed in a way that it is small, but provides enough entropy in the results to avoid its detection using correlation analysis. Since we start from C functions with a single return point, we aimed at corrupting the register storing the return value of the top module by doing extra computation on the input values. On the other hand, no performance or result changes are observed when the Trojan is inactive. Figure 5 shows the area overhead. We obtained a maximum of 13% area overhead (without considering on-chip memories), while the average overhead is less than 5%.

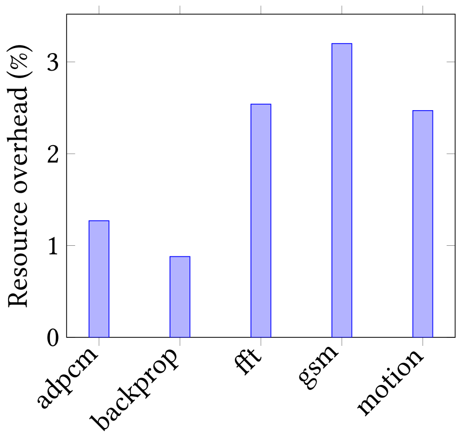


Fig. 5. Area overhead of backdoors by a malicious HLS.

AES-192 is used to evaluate FSM manipulation. The operation incrementing the round loop counter is exploited. When the trigger is turned on by a pre-defined sequence of inputs, the counter is incremented by two; so, half the rounds are performed. This attack is achieved by adjusting the control signals of the FSM and forcing the counter to a different functional unit when the Trojan is activated. The area overhead is less than 2% for the trigger circuitry and the added functional unit.

4.2.3 Possible Defense. C-to-RTL formal verification is a potential defense method against HLS-level attacks, but *sequential equivalence checking* is not well established [41]. Other methods like *discrepancy analysis* [48, 49] is based on the comparison of hardware and software traces, but require exhaustive simulations and it is part of the HLS flow, so untrustable in this context. Hardware monitors can also be used to detect anomalies during the execution of the hardware IP cores [48, 50]. However, a compromised HLS tool might generate compromised monitors. Hence, the generation process of the on-chip monitors must be separated from the HLS of the components.

Practicality: HLS attacks are standalone and hence, practical. When the tool is not able to automatically identify specific patterns to compromise, it requires collusion with the IP designer. **Scalability:** HLS attacks are scalable. The impact of the modifications reduces as the size of the circuit grows. **Stealth:** HLS attacks are stealthy and FSM manipulation during HLS is harder to detect. HLS starts from a C/C++ specification and a golden reference FSM is not available in the software specification. Logic equivalence check applied after logic synthesis do not detect HLS-based attacks because it assumes the RTL generated by HLS as the golden model. **Impact: HIGH** Correlating the C/C++ specification and the RTL is difficult, if not impossible. One solution is to use formal methods for equivalence checking after HLS, but they are well established and simulation methods are preferred [41]. However, the trigger conditions are difficult to be identified. Hence, an attacker can get away with the attacks.

4.3 Malicious Verification Tools

SoC designers use formal verification tools such as bounded model checkers (BMC) to verify not only the functional properties and specifications of a design but also its security properties. The security properties may target presence of design bugs or Trojans. Different types of security bugs are encountered in an SoC [51]. The security properties that are proved may include preventing host software from modifying the internal elements of security controller IP, disabling the firmware from reading the encryption/decryption key from the hardware, and accessing privileged memory or I/O directly, while running in the user mode. Security properties can verify presence of Trojans that either modify the function or leak sensitive information [52, 53].

4.3.1 Attack. We consider bounded model checking (BMC) as the example verification framework. An RTL description of the design and select design properties are inputs to the BMC tool. The design is unrolled for n cycles, either specified by the user or limited by the BMC, to check properties that are activated within n clock cycles. The tool should verify each property and produce counterexamples for failed properties. A malicious BMC can reduce the number of cycles for which the design is unrolled. It can skip certain properties. In both scenarios, the BMC can fail to detect Trojans or bugs in the design.

Figure 6 shows the Verilog description of a circuit that triggers a Trojan in AES design provided by Trust-hub. The Trojan targets leaking the least significant 8 bits of the secret key through the load output. The Trojan is activated when a sequence of four input patterns is applied. Consider in this example, if the Trojan is triggered, load leaks the secret key. The BMC should verify the security properties ($load_i$!= key_j, $load_i$!= key_j) to detect the security leakage, which requires unrolling the design for at least 4 cycles to verify these properties. However, a malicious BMC tool can alter this number to any value that is less than 4.

4.3.2 Experimental Results. We use SymbiYosys BMC [54], an open source formal verification tool, to check properties of benchmarks from trust-hub and SymbiYosys website [54, 55] as well as the ARM Cortex M0 processor. We select Trojans which either leak the secret key or change the function. Furthermore, we select designs which include bugs that violate the design specifications. We embed security and safety properties in these benchmarks to detect malicious activities. We unroll each design for 5 clock cycles, which is large enough to detect any malicious activity in the given benchmarks, and run the BMC.

```
module Trojan_Trigger(input rst, input [127:0] state, output Tj_Trig);
reg Tj_Trig;
reg State0, State1, State2, State3;
always @(rst, state)
begin
    if (rst == 1) begin
        State0 <= 0; State1 <= 0; State2 <= 0; State3 <= 0;
   end else if (state == 128'h3243f6a8_885a308d_313198a2_e0370734) begin
        State0 <= 1;
   end else if ((state == 128'h00112233_44556677_8899aabb_ccddeeff) &&
   (State0 == 1)) begin
        State1 <= 1;
   end else if ((state == 128'h0) && (State1 == 1)) begin
        State2 <= 1;
   end else if ((state == 128'h1) && (State2 == 1)) begin
        State3 <= 1;
   end
end
always @(State0, State1, State2, State3)
begin
   Tj_Trig <= State0 & State1 & State2 & State3;
end
endmodule
```

Fig. 6. Example of a sub-circuit that triggers a Trojan in AES design.

The results in Table 4 shows that unrolling the design for 5 clock cycles can detect Trojans/bugs of the given benchmarks except the RISC processor. The verification team should unroll the design for > 100 cycles to activate the Trojans embedded in it.

	Trojan/bug	n = 5	n=4	n=3	n=2	n=1
Benchmark	effect	Detected/ Execution time (s)				
Memory	Modify Fun.	Yes / <1	Yes / <1	Yes / <1	No / <1	No / <1
AES-T800	Leak Info.	Yes / 28	Yes / 27	No / 19	No / 18	No / 16
AES-T1100	Leak Info.	Yes / 22	Yes / 21	No / 15	No / 12	No / 11
RISC-T300	Modify fun.	No / 2	No / 2	No / 2	No / 1	No / 1
RISC-T400	Modify fun.	No / 2	No / 2	No / 2	No / 1	No / 1
ARM_Cortex	Modify Fun.	Yes / 47	Yes / 47	No / 27	No / 24	No / 22

Table 4. Ability of a Bounded Model Checker to detect Trojans for different number of unrollings (n).

4.3.3 Possible Defense. A potential defense mechanism can be conducted by the design/verification teams by intentionally modifying or adding a sub-circuit into the design to violate its specifications after a number of clock cycles, equivalent to the number of clock cycles for which the design is unrolled. If the verification tool reduces the user-defined number of clock cycles for verification, the design with intentionally introduced bugs will be proven functional. The malicious verification tool can be discarded by the design house.

Practicality: These attacks can be launched by modifying the BMC tool and hence, are practical. **Scalability:** The attack is scalable. **Stealth:** BMC CAD attack modifies the number of clock cycles to unroll the design. The verification team/designer can detect the malicious behavior by monitoring the execution time of the BMC. A small reduction in the number of unrollings may be undetectable. The attacker can also design a malicious BMC, which reduces the number of cycles to unroll the design for a few security properties. **Impact: HIGH** This attack can not be detected by downstream CAD tools.

4.4 Malicious Logic Synthesis

Logic Synthesis converts an RTL description to gate-level netlist. An important step in this conversion is extraction of Finite State Machines (FSMs) and generation of gate-level models from the FSM [56]. Often, there are undefined transitions and states in an FSM, as explained in Section 4.4.1. A malicious synthesis tool can use them to control the FSM and insert undesired transitions in the circuits. These additional transitions provide a back-door into the circuit and thereby, introduce vulnerabilties. An approach to mitigate such problems have been proposed by Dunbar et al. [38]. However, the hardware overhead of such countermeasures is around 90%, which is unrealistic for any practical design.

4.4.1 Attack. A malicious logic synthesis tool can use different approaches to create backdoors in the circuit. Let us consider the FSM shown in Figure 7(a). All transitions are not defined in this FSM. For example, the next state of the system when the current state is B and the input is 0 has not been defined. A malicious synthesis tool can take advantage of undefined transitions and create an FSM with an extra transition, as shown in Figure 7(b). We call this type of attack as Transition Attack. There is a 14% area overhead to add the undesired transition. A malicious synthesis tool can incorporate transitions to create vulnerabilities in the design without changing the original functionality. In the FSM described in Figure 7(a), if one wants to travel from state B to state A, she has to travel via state C. In the modified FSM in Figure 7(b), the malicious CAD tool has created a back-door for direct transition from state B to A. The attack model in [17] is a variation of the Transition Attack.

The second attack that we present entails adding an Extra State. This was also proposed in [18]. The malicious synthesis tool finds whether extra states can be added. Consider the FSM in Figure 7(a). Although 2 flip-flops can completely represent 4 states, this FSM comprises of only 3 states. An adversary can add an additional state to this FSM. Along with the extra state and using the undefined transitions like Transition Attack, a malicious tool creates vulnerabilities as described in Figure 7(c). Although the number of flip-flops remain the same, the new state D, incurs 14% area overhead. The additional state adds a back-door to the design.

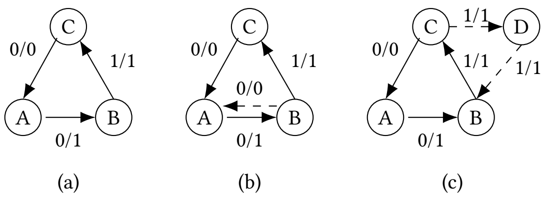


Fig. 7. Manipulating FSM to introduce vulnerabilities. All FSMs use only 2 flip-flops.(a) Original FSM to detect pattern 001. (b) Transition attack. (c) Extra state attack

4.4.2 **Limitations:** The extra transitions or states proposed in [18, 38] can be detected by Post-Synthesis Verification tools like Logical Equivalence Checker (LEC) in commercial flows. Consider the extra transition B to A, when the input is 0 in Figure 7(b). Since this transition is not in the original FSM in Figure 7(a), the LEC tool deals with this as a don't care. When the system is at state B and the input is 0, there are 3 possible outcomes in the original FSM, transit to A or C or remain in B. However, in case of Figure 7(b), the LEC tool observes that there is no valid transition from B to C, when input is 0, which was present in Figure 7(a). This counterexample detects the extra, unauthorized transition.

Although [17] mentioned that the Trojan escaped LEC check, their example had the default value in a case statement using don't cares. RTLs containing don't care assignments are not a standard commercial coding style for synthesizable RTL, since such RTL can not be simulated [57]. These RTLs create X propagation during simulation which results in an RTL versus gate simulation mismatch. These types of designs can escape LEC checks only when "SET X Conversion" is set to don't care. Commercial lint tools can identify the 'X's [17] and detect the attacks.

4.4.3 Possible Defense. As mentioned in Section 4.4.2, a post-synthesis verification tool like LEC checker can easily detect these attacks. The only way the logic synthesis attack can be avoided is by a collusion between the synthesis tool and the validation tool. Even in that case, the attack might be detected if the newly introduced transitions are rare. Existing malicious hardware detection techniques like UCI [22] and FANCI [34] observe unused and nearly-unused elements in a circuit and identify them as potential malicious parts. A rare transition introduced by the Logic Synthesis tool using the transition attack can be detected using these techniques.

Practicality: The attacks are practical since other tools are not needed. **Scalable:** The attacks scale independent of the FSM size. **Stealth:** As explained in Section 4.4.2, the attacks are not stealthy. **Impact: LOW** The only scenario when these attacks are possible is when the post-synthesis verification tool colludes with the logic synthesis tool. The LEC tool will not report a mismatch in this case by setting X-conversion to don't cares.

4.5 Malicious Placement and Routing

PNR tools arrange the components of the design on a die and route the wires between them. Once the design passes through all stages of PNR, Sign-off is performed to establish the design is ready for tape out. Sign-off checks include:

- Parasitic Extraction (SPEF generation).
- Physical Design Rule Check (DRC).
- Layout Versus Schematic (LVS).
- Signal Integrity Check.
- Rail Analysis (heat maps, effective resistances and IR drops).
- Electromigration Analysis.

We show how a malicious PNR tool can introduce vulnerabilities by focusing on crosstalk fault attacks and IR-drop attacks.

4.5.1 Crosstalk Fault Attacks. Crosstalk faults are induced when parasitic coupling capacitance between two wires exceeds a threshold [58]. When a signal transition takes place on a wire, it influences the signal value on an adjoining wire. The first wire is the aggressor while the second wire is the victim. A signal transition on the aggressor creates an unnecessary voltage fluctuation on the victim. Details on crosstalk faults are illustrated in Appendix B. We describe a two-step crosstalk fault attack. In the first step, the attack is initiated by increasing the coupling capacitances between wires in a chip. In the second step, a false netlist is generated to evade detection by the Sign-off appliances. Both these netlists are stored in the design database.

The malicious PNR tool can introduce crosstalk by placing rapidly switching aggressor nets near victim nets introducing timing failures in the victim nets. For example, the malicious tool can include artificial blockages in the design as shown in Figure 8 to achieve this goal. In Figure 8, there are 4 units, A, B, C, and D, which need to be connected. When there is no blockage, the wires are routed as shown in Figure 8(a). However, in Figure 8(b), a blockage is included in the design, causing the wire from C to D to reroute. The two wires share 2 blocks, reducing the gap between them and increasing the coupling capacitance. Hence, they are prone to crosstalk faults.

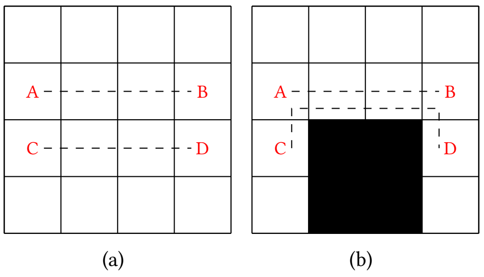


Fig. 8. Malicious blockage alters routes. (a) Routing without blockage and (b) Routing due to blockage.

Crosstalk faults can be spotted by signal integrity check during sign-off. To fend off detection, an attacker can follow these steps: The PNR tool generates two netlists (Figure 9), one for Sign-off check and the other for logic versus schematic (LVS) check. However the two netlists are not equivalence checked; both are assumed functionally identical since they are picked up from a common design database. The malicious PNR tool can generate malicious netlist for the sign-off checks while retaining the original netlist with the crosstalk faults in the design database. Since the sign-off tool analyzes $Netlist_1$, it doesn't find the crosstalk faults and signs-off the altered SoC for tape out. The foundry does design rule checks and is unable to detect the attack.

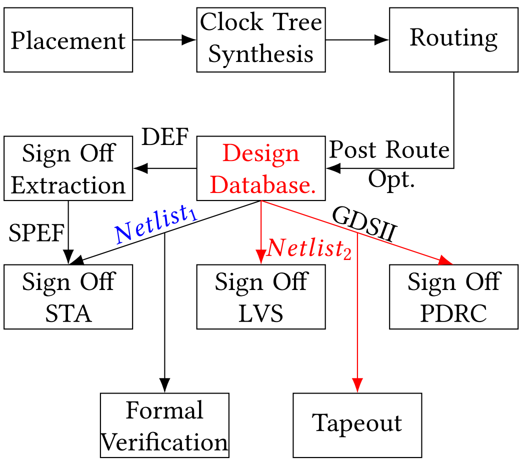


Fig. 9. Malicious PNR tools introduce crosstalk faults.

4.5.2 IR-Drop Attack. Power structures are generated by a PNR tool and the strength of the structure is analyzed in terms of IR-drop and effective resistance using a Sign-off rail analysis tool. Sign-off rail analysis tools (Figure 9) extract the RC information of the power structure and take in the power grid view of all standard cells and macros used in the design. An increase in resistance causes an IR-drop. Excessive IR-drop reduces switching speed and noise margins in circuits, which might lead to functional failures [59].

A malicious PNR tool may introduce a high resistivity path or even disconnect power for selected gates of the design. This causes IR-drops in those gates, as a result of which, they receive lower voltages. The gates with the incorrect power structure will be present in $Netlist_2$ and the GDSII files, which are signed-off for tape out. However, the BAD PNR tool dumps proper power connectivity without any additional IR drop to the cells in $Netlist_1$, which will not report any additional IR-drop. The rail analysis cannot capture the IR-drop/effective resistance/power disconnect violation.

4.5.3 Experimental Results. In order to illustrate our crosstalk-attack, we used circuits from the Opencore benchmark suite and ARM Cortex M0 processor. We use Synopsys Design Compiler for Synthesis, Cadence Encounter for PNR and the the gscl45nm library [60]. For each benchmark, two crosstalk-suspected lists are generated, one without blockage and the other with blockage. Each element of the list is a pair of wires whose coupling capacitance exceeds the crosstalk threshold. Since we do not have access to the Encounter source, we created a script to accomplish this.

Comparison of the two crosstalk-suspect lists is presented in Figure 10. Cuviello et al. [61] has demonstrated that 0.2 pF is the threshold for crosstalk glitch. We have used the same value in our experiments. All results indicate increase in number of crosstalk-suspected wires with the blockage.

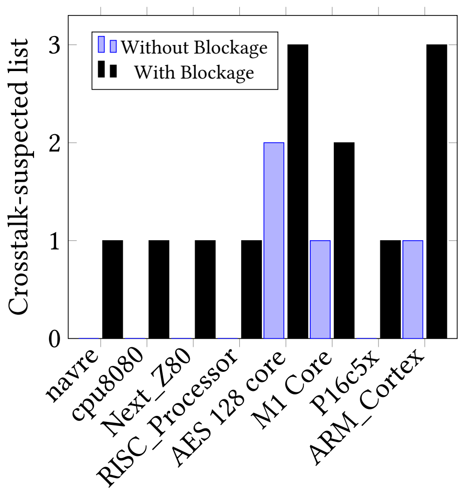
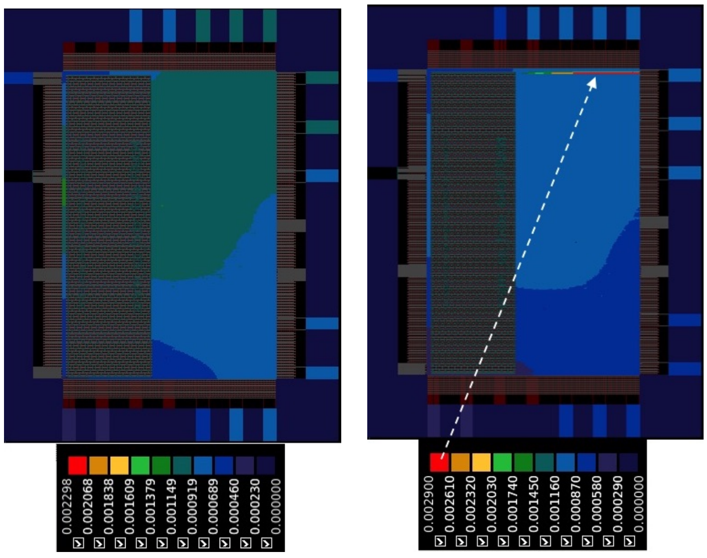


Fig. 10. Comparison of Crosstalk-suspect list. The number of crosstalk-suspected pairs increase with artificial blockage introduced by the malicious PnR tool. These faults will not be reported and the design will be sent for sign-off with these faults; thus, affecting the overall yield.

In order to demonstrate the IR-drop attack, we used the ARM Cortex-M0 processor design. We used Synopsys PrimeRail as the IR-drop measurement tool. The baseline IR-drop image is shown in Figure 11(a). The processor layout is color-coded. The color legend is shown at the bottom. Most of the design is blue and green. The PrimeRail image after removing 38 pairs of power vias is shown in Figure 11(b). Yellow and red regions indicate high IR-drop - and is pointed to by an arrow.



(a) (b)

Fig. 11. PrimeRail screenshots of IR-drop regions across the ARM Cortex-M0 layout.(a) Original IR-drop image. (b) IR-drop image after IR-drop attack. As can be seen, the malicious PnR tool increases the IR-drop, indicated by the red lines in Figure (b). The original design has a low IR drop, and this affects the overall yield.

4.5.4 Possible Defense. In this section, we have proposed two attacks: crosstalk attack and IR-drop attack. Both faults (Crosstalk and IR-drop) are generally detected using sign-off tools, used in the next level of the design flow. The sign-off detection can be avoided by generating different netlists as explained in Section 4.5.1. A possible detection mechanism will involve generating the $Netlist_2$ from the GDSII file and then performing an equivalence checking with $Netlist_1$. Any discrepancy would identify the PNR tool as malicious. However, currently, there exist no GDSII to Netlist conversion software. It is highly unlikely that two different vendors will insert faults of the same type. Therefore, if the results from two different vendors match, we can be confident that there are no design vulnerabilities. However, if they don't match, we can not be sure which tool to trust. Since, currently, no verification is performed on the two netlists, there is no way to know which one is correct. Therefore, we suggest to modify the design flow in order to allow an equivalence check between the two netlists.

Practicality: Both the attacks are standalone, and hence, practical. **Scalable:** The attacks are scalable for large designs. **Stealth:** Detection during Sign-Off stage can be avoided. **Impact: MEDIUM** These failures should not be limited to one process corner. Otherwise, the design house and foundry can work together to fine tune the process and make sure very few chips are manufactured with the worst process. This will cause a small yield loss which the design house can bear.

4.6 PNR-based Synthesis Attacks

In this section, we describe an approach to apply the Logic Synthesis-style attacks using PNR tools. As seen in Figure 9, the PNR tool generates two netlists – $Netlist_1$ is used for timing and formal verification and $Netlist_2$ is used for tapeout. The equivalence of these two netlists is never checked.

The designer assumes that these two netlists are functionally equivalent. A malicious PNR tool can introduce Trojans in $Netlist_2$, while $Netlist_1$ remains intact.

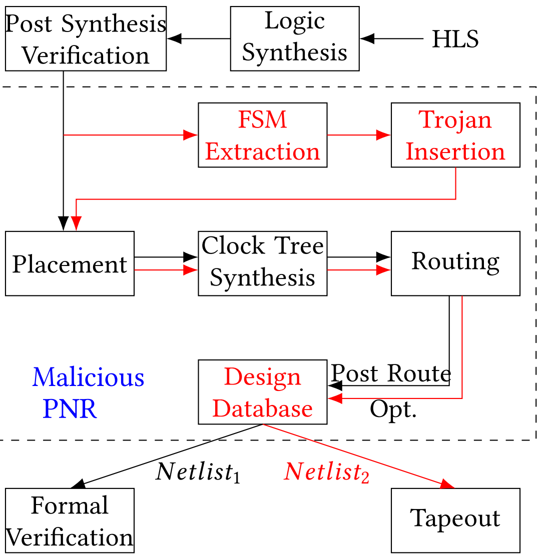


Fig. 12. Synthesis attacks using Malicious PNR tools.

At first glance, the PNR tool does not have access to the FSM from the netlist and the synthesis tool to modify the FSM. This can be remedied by altering the PNR tool such that it extracts an FSM from the netlist. The post-synthesis Logic equivalence checkers (LEC) do that. The portion of the code that extracts the FSM can be embedded into the PNR tool or the PNR can call those functions. The designer will be unaware of this if the PNR tool doesn't log these steps. Once the FSM is extracted, the malicious PNR tool can launch the Transition or Extra State attack described in Section 4.4.1. The FSM modifications code can be embedded in the PNR tool or it can call the malicious logic synthesis tool for this purpose. The final $Netlist_2$, contains the Trojan, along with the GDSII file, which is sent to tapeout. The process is shown in Figure 12. The red path shows the generation of the Trojan-inserted netlist, while the black path represents the benign netlist generation.

4.6.1 Possible Defense. Although modifications in logic synthesis can be detected using LEC checks or tools like UCI [22] and FANCI [34], the modifications in this attack are manifested as a GDSII file and not HDL/RTL. Hence, the existing tools will fail to detect the extra transitions or states. A possible defense can be similar to the one described in Section 4.5.4, by equivalence checking both the netlists. However, this will require a GDSII to netlist conversion software, which, till date, doesn't exist.

Practicality: If the PNR tool is designed to include the malicious functions, the attack is standalone. **Scalability:** The attacks are highly scalable. **Stealth:** Current design flows do not support detection of Trojans inserted during PNR, since no equivalence check is performed between $Netlist_1$ and $Netlist_2$ or between the GDS and DEF files. Sign-off checks and Physical verification checks involving design rules and LVS will pass through without detecting the Trojan. **Impact: HIGH** It can insert an extremely strong Trojan in the design, without being detected.

4.7 Malicious Static Timing Analysis

Delay faults are produced when a defect in any gate increases its delay so that the overall input-output path delay exceeds the system clock cycle. The value along this path arrives at a storage element later than scheduled. Hence, an incorrect value is stored into it, which causes an wrong output. At the nanometer scale, not all delay faults are detected using the prevailing methods. Small-delay defects (SDD) model these faults [62]. SDDs are faults created by extremely small delays compared to the clock cycle [63].

In order to establish how a malicious CAD tool can not examine all the SDD, we first present how Automatic Test Pattern Generation (ATPG) tools detect SDDs. Let us look at the timing diagram of a sample circuit with 3 paths in Figure 13.

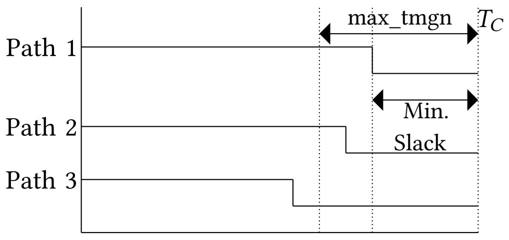


Fig. 13. Timing diagram of a sample circuit.

ATPG tools focus on paths with maximum slack, such as Path 3. On the other hand, SDD detection algorithms target paths with minimum slack. In Figure 13, this corresponds to Path 1. Synopsys Tetramax tune the *max_tmgn* parameter for this purpose. Faults on paths with slack less than *max_tmgn* are set as SDDs, while those with slacks more than *max_tmgn* are set as normal delay faults. *max_tmgn* is expressed as a fraction of the total clock cycle. For example, if the system clock cycle is 1 ns and the value of *max_tmgn* is 0.1, paths whose slack is less than 0.1 ns are tested for SDDs and the rest are tested for normal path delay violations. A Static Timing Analysis (STA) tool reports the slack along each path.

4.7.1 Attack. A malicious STA tool can give false timing information to the ATPG tool, which in turn, generates an incorrect set of timing violation detection patterns. Hence, critical large and small delay faults may go undetected, thus affecting the yield. In order to report inaccurate information, the malicious STA tool develops a false library and uses it. For example, consider a designer using the lsi_10k library. The STA tool develops a fake library internally, lsi_10k_fake, with some physical parameters different from the lsi_10k. Examples include a change in the rise time or fall time for one or more gates. Whenever the designer does timing violation tests, the STA tool uses the lsi_10k_fake library and reports timing information based on the altered parameters, leaving several faults undetected. As we will see shortly, this leads to reduction in yield.

4.7.2 Experimental Results. We use Synopsys Primetime STA tool [64]. Since we cannot modify Primetime code, we force it to use an altered library. We modify physical parameters to generate an altered library. We changed the timing information of the 4-input And-Or cell (AO4) and 2-input And-Or cell (AO2) gates in the fake library. We use designs from Opencores and the ARM Cortex M0 [65]. Details about modification in the AO2 and AO4 gates are presented in Table 5. We synthesized the designs using the original and fake libraries, perform STA and generate test patterns for SDD.

Statistical delay quality level (SDQL) measures the number of SDDs that remain undetected by an SDD test [66]. Details on SDQL are explained in Appendix C. A higher SDQL indicates higher delay faults. Modifications in Table 5 affect the SDQL. The SDQL information is obtained from

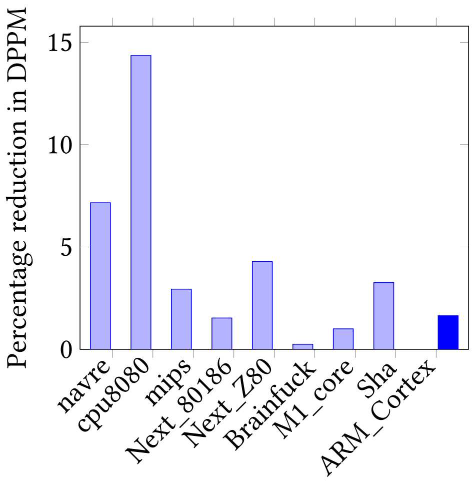


Fig. 14. Change in test escape rates expressed as DPPM when the original library is used instead of the fake one.

	AO2		AO4	
	original	malicious	original	malicious
intrinsic rise time	0.82	0.12	0.92	0.22
intrinsic fall time	0.57	0.07	0.37	0.07

Table 5. Malicious modifications in *lsi*_10*k* library.

Primetime STA. According to [67], the test escape rates of chips, expressed as Defective Parts Per Million (DPPM) is calculated as: $DPPM = \frac{SDQL}{N}$, where N is the number of delay faults. DPPM is a reliable measure of the probability that a chip being declared defective. Higher SDQL indicates higher DPPM, that is, higher probability of failure. Thus, a modification in the library results in change of SDQL value, which in turn, changes the DPPM. Figure 14 highlights the shift in DPPM results when the original library is used instead of the fake library. The value of *max_tmgn* is held constant at 0.25. Figure 14 demonstrates that a malicious STA tool can worsen the yield.

4.7.3 Possible Defense. Since the results of STA are not checked downstream, this attack is resistant against any further validations. However, care should be taken to affect all process corners approximately equally. If only one corner is affected, it can be easily fixed by the design house.

Practicality: This attack depends on the foundry to insert faults. **Scalability:** The attack is scalable. **Stealth:** This attack is immune against downstream checks. **Impact: MEDIUM** This attack reduces yield and incurring losses to the design house. The malicious STA tool should affect all process corners. If one corner is affected, the design house can collaborate with the foundry to address this.

4.8 Malicious Post-Silicon Validation

Trace buffers are used during post-silicon validation to store some of the internal signal states of a circuit. Since the trace buffer size is limited, only a few signal states can be stored. Therefore, the signals to be stored in the trace buffer have to be chosen carefully. Trace signals are chosen according to their restoration ability. Signal restoration refers to reconstructing an untraced signal state. Signal restoration is explained in Appendix D. Restoration ratio is a parameter to measure the restoration performance of a set of trace signals. It is the ratio of the total number of states restored and the total number of states traced. A higher restoration ratio indicates a stronger restoration performance. Different signal selection approaches like total restorability-based [19], simulation-based [68], ILP-based [9] have been proposed to maximize the restoration ratio. We used SigSET [9] to demonstrate how a malicious signal selection might impair performance.

4.8.1 Attack. The trace buffer width denotes the number of signal states that can be stored each cycle during post-silicon validation. This varies from 8, 16 to 32. The signal selection tools accept the circuit netlist and trace buffer width as inputs and returns the set of trace signals. A malicious signal selection tool can drop some of the correctly selected signals and add others to the list of selected signals. However, care should be taken so that the final restoration ratio does not differ by much in order to avoid suspicion. Hence, signals, whose restoration power are minimal are excluded from the set of trace signals. If w is the trace buffer width, the malicious signal selection tool will generate a list S and S' of w and $w + k$ signals respectively, where the parameter k denotes the number of trace signals to be modified. The last k signals in S are replaced with the last k signals in S'. The incorrect set of trace signals S is then provided to the user. Signals are ordered in descending order of restoration performance, so the k signals replaced will have less restoration ability than the previous $w - k$ signals in the original list S. An incorrect set of trace signals with reduced abilities can lead to degradation in post-silicon validation performance.

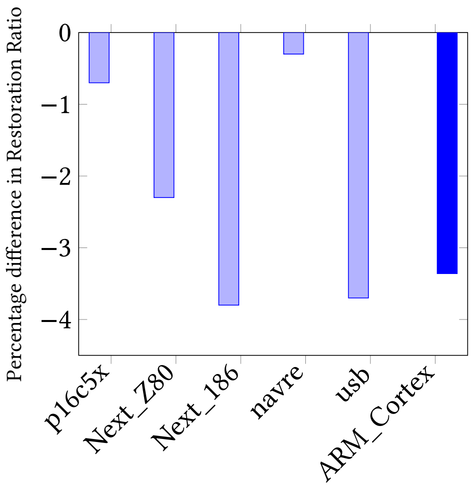


Fig. 15. Comparison of Restoration Ratio.

4.8.2 Experimental Results. We used Opencores benchmarks and the ARM Cortex M0 processor in the experiments. A 32-bit trace buffer is chosen. The modified set of signals are generated by replacing 2 signals in each list. Care is taken so that signals, which help restoring a lot of untraced states, are not touched. Change in restoration ratio due to modified trace signals is presented in Figure 15. The signals generated by the malicious CAD tool always have less restoration capability compared to the original set. This will affect post-silicon validation. The maximum reduction in restoration ratio is 3.8%. The minimal reduction in restoration ratio does not raise an alarm.

4.8.3 Possible Defense. The results of post-silicon validation are never verified at a later stage. Moreover, there is no "later stage", since post-silicon validation is performed after the chip is fabricated. Therefore, these attacks are robust against any downstream checks.

Practicality: These attacks depend on a malicious player in the design flow to insert vulnerability, which can be left undetected. **Scalable:** These attacks are scalable. **Stealth:** Post-silicon validation results are never validated in downstream design stages and hence, this attack can never be detected. The only way to detect this attack will be to cross-check the restoration ratio using a different signal selection tool, which would lead to a dilemma – which tool to trust? **Impact: LOW** This attack can only help in retaining existing faults in the design.

4.9 Malicious Manufacturing Testing Tools

Automatic Test Pattern Generation (ATPG) algorithms have two objectives - activate an electrical fault and propagate it to an observable output, which might be a primary output for the circuit or a flip-flop in a scan chain. The ultimate goal of the ATPG algorithm is to reduce test pattern count, thus reducing test time and memory. A malicious ATPG algorithm can undermine either of the two primary objectives, by dropping or modifying patterns.

Let us review how a conventional ATPG algorithm operates using the open-source ATALANTA ATPG tool [69]. The input to the tool are the design and the Fault list F, which gives a list of potential faults. ATALANTA chooses a single fault f from the Fault list F and generates patterns for it. The don't care bits are filled with 0. This pattern is applied on the design to identify additional faults it can detect. All these faults are added to another set F'. The members of F' are excluded from F. The pattern is stored in a pattern set P. This process continues until no fault remains in the fault-list F[3]. We will present two attacks targeting general ATPG and test generation for Trojans respectively.

4.9.1 Malicious Fault Dropping. In the first attack, some faults are deliberately left undetected by the ATPG tool. The number of faults dropped is kept small, since an unexpected rapid drop in fault coverage will lead to suspicions. If this knowledge is passed on to a malicious foundry, it can exploit them to introduce defects or Trojans in the design. This is a multi-level attack as proposed by [27] and is unlikely to succeed. Of course, dropping faults will reduce the yield even if the foundry is not in collusion. Malicious fault-dropping by an ATPG is explained in Algorithm 1 in Appendix E.

4.9.2 Malicious Trojan Site Dropping. Instead of dropping all faults, the fault list can be pruned to drop faults from potential Trojan locations. This will provide the foundry places to introduce Trojans. In order to identify the Trojan sites, one can adopt MERO [70]. Gates which have the least probability of activation are identified as potential Trojan locations by MERO.

4.9.3 Experimental Results. In order to demonstrate fault dropping attack, we used the open source ATALANTA tool and *ISCAS*'85 benchmarks. ATALANTA can only run on such combinational benchmarks[4]. However, any commercial ATPG tool operating on a combinational or sequential circuit can utilize the same technique to drop faults by modifying their internal code. Figure 16(a) illustrates the difference in coverage when original ATALANTA is applied, and when malicious ATALANTA drops faults. We assumed $k = 1$ and $n = 5$. For all benchmarks, the coverage reduces when faults are dropped, the maximum reduction being 1.17%.

[3]In this section, we will analyze only stuck-at faults. Delay faults attacks has been explained in Section 4.7.
[4]Opencore benchmarks and ARM Cortex have sequential components.

The results for malicious Trojan site dropping are presented in Figure 16(b). The coverage decreases when faults are dropped. However, the drop in coverage is less than Figure 16(a). This is because the search space is pruned when Trojan locations are dropped, compared to when any fault can be dropped. Hence, the number of faults dropped reduces and the coverage curtailment is negligible.

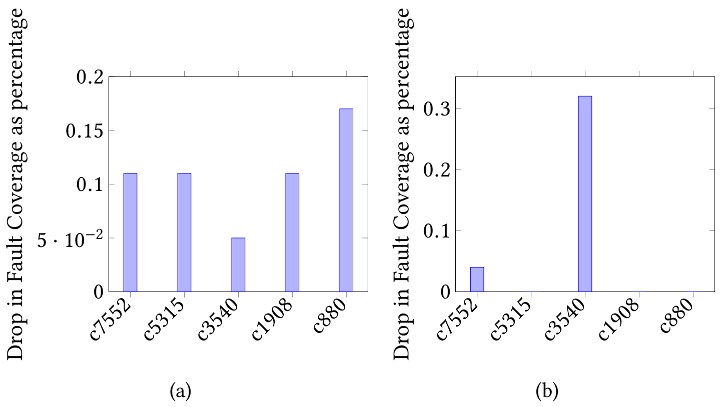


Fig. 16. Fault Coverage drops (a) slightly after fault dropping, (b) negligibly after malicious Trojan site dropping.

4.9.4 Possible Defense. Similar to post-silicon validation, the manufacturing testing results are never verified. Hence, this attack is also robust against downstream checks.

Practicality: This attack depends on the foundry to introduce the rare faults, which will, then remain undetected. An ATPG tool can't introduce a fault by itself. **Scalability:** This attack is scalable. **Stealth:** Since test patterns are never verified, it is extremely difficult to catch this attack. **Impact: LOW** Any manufacturing fault undetected by the ATE remains in the chip when shipped to the customer, thereby incurring losses. Collusion with foundry is necessary to insert Trojans.

5 TO CONCLUDE, CAD COULD BE BAD

Table 6 summarizes key takeaways of this study that reviewed an extensive set of attacks using CAD tools. All attacks are scalable. An attack is practical if it is easy to launch without collusion. An attack is stealthy if it is difficult to detect. HLS, verification, and PNR tools can introduce attacks that have high impact as they can go undetected even within a commercial CAD flow. STA, test and post silicon validation tools can introduce vulnerabilities but require collusion with a malicious foundry. This minimizes impact. Logic synthesis attacks can be similarly detected by checks and balances in the design flow. One way to deal with malicious CAD tools is to use CAD tools from multiple vendors; design tools from one vendor and verification/validation tools from another.

Design stage	Practical	Stealthy	Impact
HLS	High	Yes	High
Verification	High	Yes	High
Logic Synthesis	High	No	Low
PNR	High	Yes	High
Static Timing Analysis	Low	Yes	Medium
Post-Silicon Validation	Low	Yes	Low
Manufacturing Testing	Low	Yes	Low

Table 6. CAD-based attacks: an assessment summary.

REFERENCES

[1] J. Yiu, *The Definitive Guide to Arm® Cortex®-m0 and Cortex-m0+ Processors.* Academic Press, 2015.

[2] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 18–37, IEEE, 2016.

[3] R. Kumar, *Fabless semiconductor implementation.* McGraw-Hill, Inc., 2008.

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

[5] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *Proceedings of High Level Design Validation and Test Workshop*, pp. 166–171, IEEE, 2009.

[6] D. R. Collins, "Trust, a proposed plan for trusted integrated circuits," tech. rep., DEFENSE ADVANCED RESEARCH PROJECTS AGENCY ARLINGTON VA MICROSYSTEMS TECHNOLOGY OFFICE, 2006.

[7] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 197–214, Springer, 2013.

[8] "https://dac.com/content/exhibits."

[9] K. Rahmani, P. Mishra, and S. Ray, "Efficient trace signal selection using augmentation and ilp techniques," in *Proceedings of International Symposium on Quality Electronic Design*, pp. 148–155, IEEE, 2014.

[10] T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. D. Nguyen, and C. Irvine, "Managing security in fpga-based embedded systems," *Proceedings of IEEE Design & Test of Computers*, vol. 25, no. 6, 2008.

[11] J. A. Roy, F. Koushanfar, and I. L. Markov, "Ending piracy of integrated circuits," *Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[12] M. M. Tehranipoor, U. Guin, and D. Forte, "Counterfeit integrated circuits," in *Counterfeit Integrated Circuits*, pp. 15–36, Springer, 2015.

[13] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proceedings of Computer Security Applications Conference*, pp. 473–482, IEEE, 2006.

[14] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1591–1604, Oct. 2016.

[15] A. H.-W. Tseng, H. Van Tang, and V. C. Ta, "Logic emulator using a disposable wire-wrap interconnect board with an fpga emulation board," May 11 1999. US Patent 5,903,744.

[16] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, *et al.*, "Bounded model checking.," *Advances in computers*, vol. 58, no. 11, pp. 117–148, 2003.

[17] N. Fern, S. Kulkarni, and K.-T. T. Cheng, "Hardware trojans hidden in rtl don't cares—automated insertion and prevention methodologies," in *Proceedings of IEEE International Test Conference*, pp. 1–8, IEEE, 2015.

[18] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "Avfsm: a framework for identifying and mitigating vulnerabilities in fsms," in *Proceedings of ACM/EDAC/IEEE Design Automation Conference*, pp. 1–6, IEEE, 2016.

[19] K. Basu and P. Mishra, "Rats: Restoration-aware trace signal selection for post-silicon validation," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 21, no. 4, pp. 605–613, 2013.

[20] K. Basu, P. Mishra, and P. Patra, "Efficient combination of trace and scan signals for post silicon validation and debug," in *Proceedings of IEEE International Test Conference*, pp. 1–8, IEEE, 2011.

[21] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware.," *Leet*, vol. 8, pp. 1–8, 2008.

[22] M. Hicks, M. Finnicum, S. T. King, M. M. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proceedings of IEEE Symposium on Security and Privacy (S&P*, pp. 159–172, IEEE, 2010.

[23] Y. Huang, S. Bhunia, and P. Mishra, “Mers: statistical test generation for side-channel analysis based trojan detection,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 130–141, ACM, 2016.

[24] A. Waksman and S. Sethumadhavan, “Silencing hardware backdoors,” in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 49–63, IEEE, 2011.

[25] X. Wang, S. Narasimhan, A. Krishna, T. Mal-Sarkar, and S. Bhunia, “Sequential hardware trojan: Side-channel aware design and placement,” in *Proceedings of IEEE International Conference on Computer Design*, pp. 297–300, IEEE, 2011.

[26] L. Lin, W. Burleson, and C. Paar, “Moles: malicious off-chip leakage enabled by side-channels,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 117–122, ACM, 2009.

[27] S. S. Ali, R. S. Chakraborty, D. Mukhopadhyay, and S. Bhunia, “Multi-level attacks: An emerging security concern for cryptographic hardware,” in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–4, IEEE, 2011.

[28] D. Colins, “Trust in integrated circuits (tic),” *DARPA Solicitation BAA07-24*, 2007.

[29] G. Bloom, B. Narahari, and R. Simha, “Os support for detecting trojan circuit attacks,” in *Proceedings of IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 100–103, IEEE, 2009.

[30] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, “Hardware trojan attacks: threat analysis and countermeasures,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[31] M. Tehranipoor and F. Koushanfar, “A survey of hardware trojan taxonomy and detection,” *IEEE design & test of computers*, vol. 27, no. 1, 2010.

[32] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, “Trojan detection using ic fingerprinting,” in *Proceedings of IEEE Symposium Security and Privacy*, pp. 296–310, IEEE, 2007.

[33] J. Li and J. Lach, “At-speed delay characterization for ic authentication and trojan horse detection,” in *Proceedings of IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 8–14, IEEE, 2008.

[34] A. Waksman, M. Suozzo, and S. Sethumadhavan, “Fanci: identification of stealthy malicious logic using boolean functional analysis,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 697–708, ACM, 2013.

[35] C. Sturton, M. Hicks, D. Wagner, and S. T. King, “Defeating uci: Building stealthy and malicious hardware,” in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 64–77, IEEE, 2011.

[36] F. Farahmandi, Y. Huang, and P. Mishra, “Trojan localization using symbolic algebra,” in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pp. 591–597, IEEE, 2017.

[37] J. A. Roy, F. Koushanfar, and I. L. Markov, “Circuit cad tools as a security threat,” in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 65–66, IEEE, 2008.

[38] C. Dunbar and G. Qu, “Designing trusted embedded systems from finite state machines,” *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 5s, p. 153, 2014.

[39] M. Potkonjak, “Synthesis of trustable ics using untrusted cad tools,” in *Proceedings of ACM/IEEE Design Automation Conference*, pp. 633–634, IEEE, 2010.

[40] F. Koushanfar and M. Potkonjak, “Cad-based security, cryptography, and digital rights management,” in *Proceedings of ACM/EDAC/IEEE Design Automation Conference*, pp. 268–269, ACM, 2007.

[41] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang, “Challenges and trends in modern SoC design verification,” *IEEE Design & Test*, vol. 34, pp. 7–22, Oct. 2017.

[42] A. Waksman, M. Suozzo, and S. Sethumadhavan, “FANCI: Identification of stealthy malicious logic using boolean functional analysis,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS ’13, pp. 697–708, 2013.

[43] P. Derbez, P.-A. Fouque, and J. Jean, “Improved key recovery attacks on reduced-round AES in the single-key setting,” in *Advances in Cryptology – EUROCRYPT 2013* (T. Johansson and P. Nguyen, eds.), pp. 371–387, 2013.

[44] S. Sanadhya and P. Sarkar, “Attacking reduced round SHA-256,” in *Applied Cryptography and Network Security* (S. Bellovin, R. Gennaro, A. Keromytis, and M. Yung, eds.), pp. 130–143, 2008.

[45] C. Pilato and F. Ferrandi, “Bambu: A modular framework for the high level synthesis of memory-intensive applications,” in *Procedings of the International Conference on Field-Programmable Logic and Applications*, pp. 1–4, Sept. 2013.

[46] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, “Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis,” *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[47] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, “MachSuite: Benchmarks for accelerator design and customized architectures,” in *Proceedings of the IEEE International Symposium on Workload Characterization*, (Raleigh, North Carolina), October 2014.

[48] P. Fezzardi, M. Lattuada, and F. Ferrandi, “Using efficient path profiling to optimize memory consumption of on-chip debugging for high-level synthesis,” *ACM Transactions on Embedded Computing Systems*, vol. 16, pp. 149:1–149:19, Sept. 2017.

[49] P. Fezzardi, C. Pilato, and F. Ferrandi, "Enabling automated bug detection for IP-based designs using high-level synthesis," *IEEE Design & Test*, 2018.

[50] M. B. Hammouda, P. Coussy, and L. Lagadec, "A unified design flow to automatically generate on-chip monitors during high-level synthesis of hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, pp. 384–397, Mar. 2017.

[51] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial soc design," in *Proceedings of Design, Automation Test in Europe Conference Exhibition*, pp. 1–2, March 2014.

[52] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal security verification of third party intellectual property cores for information leakage," in *Proceedings of International Conference on VLSI Design and International Conference on Embedded Systems*, pp. 547–552, 2016.

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

[54] C. Wolf, "Formal verification with symbiyosys and yosys-smtbmc," in *http://www.clifford.at/papers/2017/smtbmc-sby/*.

[55] M. Tehranipoor, R. Karri, F. Koushanfar, and M. Potkonjak, "Trusthub," in *http://trust-hub.org*.

[56] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[57] M. Turpin and P. V. Engineer, "The dangers of living with an x (bugs hidden in your verilog)," in *Synopsys Users Group Meeting*, 2003.

[58] A. Rubio, N. Itazaki, X. Xu, and K. Kinoshita, "An approach to the analysis and detection of crosstalk faults in digital vlsi circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 3, pp. 387–395, 1994.

[59] J. Saxena, K. M. Butler, V. B. Jayaram, S. Kundu, N. Arvind, P. Sreeprakash, and M. Hachinger, "A case study of ir-drop in structured at-speed testing," in *Proceedings of International Test Conference*, p. 1098, IEEE, 2003.

[60] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, *et al.*, "Freepdk: An open-source variation-aware design kit," in *Proceedings of IEEE International Conference on Microelectronic Systems Education*, pp. 173–174, IEEE, 2007.

[61] M. Cuviello, S. Dey, X. Bai, and Y. Zhao, "Fault modeling and simulation for crosstalk in system-on-chip interconnects," in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pp. 297–303, 1999.

[62] X. Lin, K.-H. Tsai, C. Wang, M. Kassab, J. Rajski, T. Kobayashi, R. Klingenberg, Y. Sato, S. Hamada, and T. Aikyo, "Timing-aware atpg for high quality at-speed testing of small delay defects," in *Proceedings of Asian Test Symposium*, pp. 139–146, IEEE, 2006.

[63] A. E. BRAUN, "Pattern-related defects become subtler, deadlier," *Semiconductor international*, vol. 28, no. 6, pp. 53–60, 2005.

[64] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys® Design CompilerTM Physical CompilerTM and PrimeTime®*. Springer Science & Business Media, 2007.

[65] "Opencores.org."

[66] Y. Sato, S. Hamada, T. Maeda, A. Takatori, and S. Kajihara, "Evaluation of the statistical delay quality model," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 305–310, ACM, 2005.

[67] S. Deutsch, K. Chakrabarty, S. Panth, and S. K. Lim, "Tsv stress-aware atpg for 3d stacked ics," in *Proceedings of Asian Test Symposium*, pp. 31–36, IEEE, 2012.

[68] D. Chatterjee, C. McCarter, and V. Bertacco, "Simulation-based signal selection for state restoration in silicon debug," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 595–601, IEEE Press, 2011.

[69] H. Lee and D. Ha, "Atalanta: An efficient atpg for combinational circuits," tech. rep., Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993.

[70] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Cryptographic Hardware and Embedded Systems*, pp. 396–410, Springer, 2009.

[71] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*, pp. 263–268, ACM, 1997.

[72] P. Mishra, N. Dutt, N. Krishnamurthy, and M. Ababir, "A top-down methodology for microprocessor validation," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 122–131, 2004.

[73] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*, pp. 25–60, Springer, 1993.

[74] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the ACM symposium on Theory of computing*, pp. 151–158, ACM, 1971.

[75] C. H. Gebotys and M. I. Elmasry, "Vlsi design synthesis with testability," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 16–21, IEEE Computer Society Press, 1988.

APPENDIX A FORMAL VERIFICATION TECHNIQUES

Equivalence Checking is the most popular method employed for formally verifying a circuit [71]. Equivalence of two different circuits are verified using Binary Decision Diagrams (BDDs). Equivalence Checking is fast compared to simulation-based verification. Formality, the equivalence checking tool from Synopsys, can verify million gate designs in under an hour [72].

Model Checking is another formal verification procedure in which, design models are verified against specifications [73]. Typical design models are automata-based and specifications are either temporal properties or environmental conditions. A model checker either returns a counterexample if the design is flawed, or proves the design matches the specification. VIS and SMV are two commercial Model Checkers.

Theorem Proving is the most powerful formal verification procedure [74]. Both the design and specification are expressed as logical formula and they are checked against each other by proving a theorem. However, this procedure requires human intervention. A high-level of human competence is necessary for theorem proving and hence, it is still limited to academic usage.

APPENDIX B EFFECTS OF CROSSTALK FAULTS

The effects of crosstalk are twofolds: glitch and delay. A glitch occurs when only the aggressor wire has a transition, while the victim wire retains a stable value. Figure 17 demonstrates how a glitch is formed on a victim due to signal transition on an aggressor.

Crosstalk delays are caused when both aggressor and victim wires have transitions. The transition on the aggressor wire creates a delay on the transition on the victim wire. If both the transitions are in the same direction, the delay is negative, that is, the transition on the victim wire occurs faster than expected. On the other hand, if both the transitions are in different directions, the delay is positive, that is, the transition in the victim occurs later. Figure 18 illustrates an example of positive crosstalk delay on the victim wire.

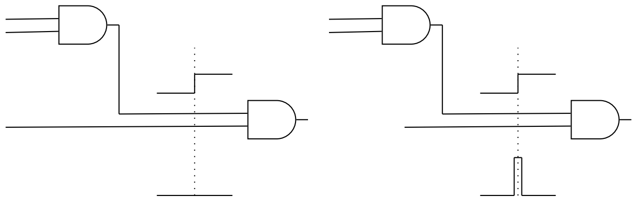

(a) Cause of Glitch. (b) Glitch due to Crosstalk.

Fig. 17. Glitch caused due to crosstalk effect.

APPENDIX C STATISTICAL DELAY QUALITY LEVEL

Statistical delay quality level (SDQL) measures the number of SDDs that remain undetected by an SDD test. Consider a circuit with two paths A and B, as shown in Figure 19. Let A be the longest path in the circuit, i.e., one with the minimal slack; while B be the longest path in the circuit which is sensitized. Slack of B is assumed to be more than A. Let T_{mc} be the system clock and T_c be the testing clock. T_{margin} is defined as the difference in time between T_{mc} and Path A, while T_{de} is defined as the difference in time between T_c and Path B.

As explained by [66], faults whose delays are less than T_{margin} are rejected, while faults with delays greater than T_{de} are detected. Faults with delays in between these two, that is, faults whose

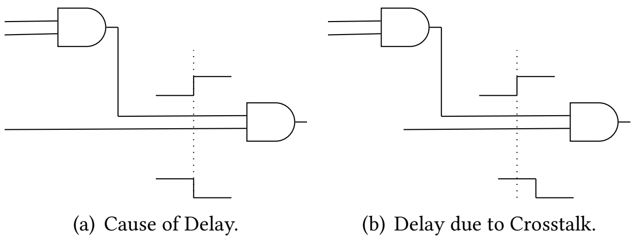

(a) Cause of Delay. (b) Delay due to Crosstalk.

Fig. 18. Delay caused due to crosstalk effect.

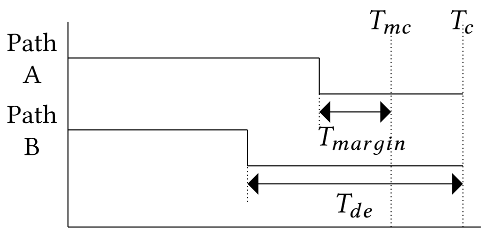


Fig. 19. Timing diagram for SDQL.

delays are greater than T_{mgn} and less than T_{de} remain undetected. SDQL measures the total number of those faults. If s represents the delay of a fault, SDQL measures faults such that $T_{mgn} \leq s \leq T_{de}$.

If N is the number of nodes in a design, and $2N$ is the number of assumed faults (each node can have a rise and a fall delay fault), the SDQL of a chip is expressed by:

$$SDQL = \sum_{K=1}^{2N} \int_{T_{mgn}}^{T_{de}} F(s)ds \tag{1}$$

A lower SDQL indicates less faults.

APPENDIX D SIGNAL RESTORATION

Trace signals are chosen according to their restoration ability. Signal restoration refers to reconstructing an untraced signal state. Consider the two input AND gate in Figure 20. Let a and b be the inputs, and c be the output. Let us examine a case when either of the two inputs is traced. For example, if a is traced and state of a is 0, the state of output signal c is inferred as 0 as well, since 0 is the dominating input for an AND gate. Thus, tracing a can restore c. This is known as forward restoration, where tracing the inputs helps to restore the output. Now, consider the case, when output c is traced. If the value of c is 1, both a and b are a 1, since 1 is the non-dominating input of an AND gate. This is backward restoration, where tracing the output can restore the inputs. Although we illustrated the example for a 2-input AND gate, similar restoration concepts apply to other boolean gates like OR, NOT, etc. with varying number of inputs.

APPENDIX E FAULT DROPPING ALGORITHM

Algorithm 1 describes the fault-dropping algorithm for malicious ATPG tool. In Algorithm 1, k indicates the maximum number of bits that are modified for each dropping pattern, and n indicates the maximum number of faults dropped. Typically, n should be less than 5, to ensure minimal reduction in coverage. Patterns which detect only a single fault are identified and up to k bits of

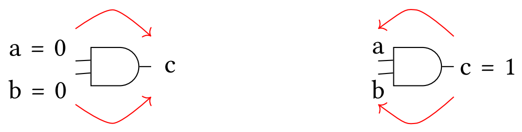


Fig. 20. Signal Restoration using a 2-input *AND* gate

ALGORITHM 1: Malicious Fault Dropping using ATPG

```
Algorithm Attack(Design, k, n [faultlist F], [suspect-list S])
If faultlist not provided, generate all faults from design and store in faultlist $F$;
Create an empty set of patterns $P$;
$count\_patterns\_modified = 0$;
while $F$ is not empty or new patterns can be generated do
    Pick a $f$ from $F$;
    Generate pattern $p$ for $f$;
    Fill the don't cares of $p$;
    Find the set of faults $F'$ detected by $p$;
    $F = F - F'$;
    if $F' = 1$ AND $count\_patterns\_modified < n$ AND $f \in S$ then
        Switch $k$ number of 1's of $p$ to 0;
        $count\_patterns\_modified = count\_patterns\_modified + 1$;
    end
    $P = P + p$
end
return $P$;
```

those patterns are altered. Thus, each pattern modification will affect only a single fault. n such patterns are altered. Both n and k are regulated by the malicious ATPG tool. After generating a pattern for a fault f, we figure out if a pattern detects only one fault, i.e, if F' has only one member. This pattern is now a potential candidate for modification. If we choose to avoid detection of n faults, we continue this step for n patterns.

APPENDIX F DESIGN FOR TEST (DFT)

In the DFT stage, scan chains are inserted to a synthesized netlist in order to make the design suitable for Manufacturing Testing [75]. Each flip-flop is converted to a scan flop using a multiplexer, which selects the test mode. During functional mode, the flip-flop will act normally. During scan mode, the flip-flop receives data from the scan chain. Scan chains improve the controllability and observability during manufacturing testing. Commercial DFT tools include Synopsys DFTMAX and Cadence Encounter Test.

A possible attack scenario using malicious DFT tools can be to skip some flip-flops in the scan chains. For example, if there are n flip-flops in the design numbered F_1, F_2 up to F_n, the malicious DFT tool can skip F_k where $k < n$. This will result in the combinational logic close to F_k as well as F_k itself remaining untested and hence, a reduction in test coverage. This attack will remain undetected as there are no methods for scan chain validation. Since this is not a functional modification of the design, equivalence checking will also not be able to detect it. Unfortunately, we could not demonstrate experimental results for this because there are no opensource DFT tools to the best of our knowledge and this attack can not be launched using modified scripts.