Black-Hat High-Level Synthesis: Myth or Reality?

Christian Pilato, *Member, IEEE,* Kanad Basu, *Member, IEEE,*
Francesco Regazzoni, *Member, IEEE,* and Ramesh Karri, *Senior Member, IEEE*

***Abstract*—Hardware Trojans are a major concern for integrated circuits and all parts of the electronics supply chain are vulnerable to this threat. Trojans can be inserted directly by a rogue employee or through a compromised computer-aided design (CAD) tool at each step of the design cycle, including an alteration of the design files in the early stages and the fabrication process in a third-party malicious foundry. While Trojan insertion during the latter stages has been largely investigated, we focus on High-Level Synthesis (HLS) tools as a likely attack vector. HLS tools are used to generate Intellectual Property (IP) blocks from high-level specifications. To demonstrate the threat, we compromised an open-source HLS tool to inject three examples of HLS-aided hardware Trojans with functional and non-functional effects. Our results show that a black-hat HLS tool can be successfully used to maliciously alter electronic circuits to add latency, drain energy, or undermine the security of cryptographic hardware cores. This threat is an important security concern to address.**

***Index Terms*—High-Level Synthesis, Intellectual Property, Hardware Trojan.**

I. Introduction

MODERN System-on-Chip (SoC) architectures are complex systems with growing design complexity and manufacturing costs [1]. These costs are difficult to accept by individual semiconductor design firms, which are becoming fab-less [2]. A fab-less design company leverage several third-party computer-aided design (CAD) tools for its SoC design, which is then shipped to a third-party foundry for manufacturing. This design paradigm reduces the costs, but has security flaws [3]; a rogue employee or a malicious foundry can insert *hardware Trojans* [4]. A hardware Trojan is a malicious modification of the design used to extract sensitive information (e.g., encryption keys) or alter system behavior by introducing system failures or denial-of-service attacks. A hardware Trojan provides an advantage for the attacker by inflicting economic damages to its competitors (e.g., *via planned obsolescence* [5]) or cyber-attacks on national assets [6]. Most CAD companies also provide a portfolio of pre-characterized Intellectual Property (IP) components (e.g., Xilinx IP cores [7]). If a design house develops an IP, which is in direct competition to the CAD company's IP, the latter has a reason to thwart the design house's IP using malicious tools. Chips developed in one country use tools developed by CAD companies in other countries. If CAD company's host country wants to spy on the chip-developer's country, it can force the CAD company to create backdoors in the design using national security interest arguments. A hardware Trojan can be always active or stealthy during typical operation while exceptional conditions trigger its malicious function. Existing protection methods focus on detecting hardware Trojans added *after* the design phase. In this paper, we look at Trojans that can be introduced during *High-Level Synthesis* (HLS). Due to the escalating complexity, cost, and error-proneness of hardware designs, modern SoCs are designed by reusing an increasing number of pre-existing components (more than 90% by 2020 [8]). Automatic HLS is becoming a popular method to support this trend [9]. HLS allows engineers to start from a high-level specification (C/C++/SystemC) and create an optimized hardware description in Verilog/VHDL.

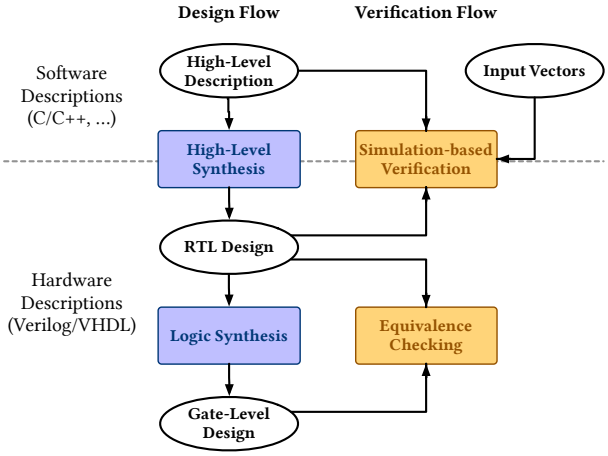


Fig. 1. Front-end HLS design flow for SoC architectures.

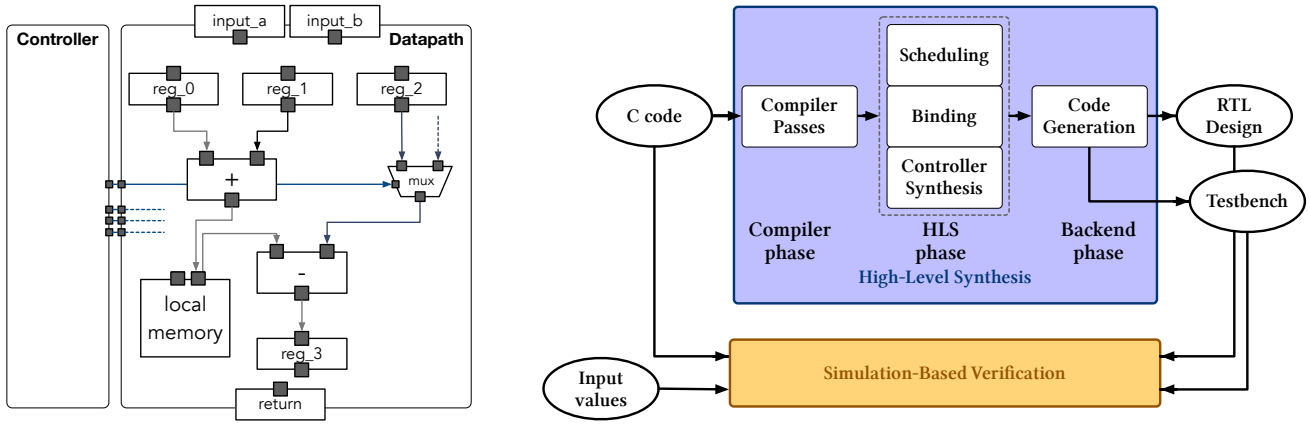
Let us examine the other face of the coin. The RTL code produced during HLS is machine generated and orders of magnitude bigger than the high-level specification. Hence, it is often difficult for an engineer to understand the behavior and correlate the RTL code with the high-level specification. Formal methods are used to correlate the initial high-level description with the machine generated hardware design. Since the HLS tools may restructure and optimize code, this correlation is laborious, if not hopeless [10]. Hence, simulation-based methods are the favored solutions to validate HLS-generated designs. Many commercial and academic tools automatically generate hardware testbenches or co-simulation with software specifications to check whether the generated hardware causes similar results as obtained by the software [9], [11]. This process does not provide complete coverage because exhaustive testing is impossible. This clears the prospect of inserting hardware Trojans at the HLS level without being discovered. Figure 1 represents the front-end design flow for SoCs. Phys-

Manuscript received June 9, 2018; revised October 11 2018; accepted November 19, 2018. This work was partially supported by National Science Foundation (NSF) (award #1526405).

C. Pilato is with the Dipartimento di Elettronica, Informatica e Bioingegneria, Politecnico di Milano, Italy (Contact email: christian.pilato@polimi.it).

K. Basu and R. Karri are with the NYU Center for Cybersecurity (http://cyber.nyu.edu), New York University (NYU), New York, NY, USA.

F. Regazzoni is with the Advanced Learning and Research Institute (ALaRI), Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland.



(a) (b)

Fig. 2. (a) Example of HLS-generated microarchitecture. Dotted lines represent additional command signals from the controller to select multiplexer inputs or to enable register-write operations. (b) HLS flow with simulation-based verification.

ical design and manufacturing follow this step to manufacture the SoC. Once the Trojans escape validation in the front-end, the subsequent verification processes (e.g., equivalence checking after logic synthesis) believe this compromised RTL design as the golden model.

Due to the increasing use of HLS and the complexity of hardware debugging, we want to address the following question: **which is the potential impact of Trojan insertion during HLS?** All commercial HLS companies sell proprietary HLS CAD software, but also a portfolio of IP components. There is no guarantee that the licensed software is not altered by a rogue employee (e.g., with compromised updates) to insert hardware Trojans and undocumented features. Compromised HLS tools can create components that are less competitive on the market due to repeated failures after a specific time-frame (i.e., after the Trojan is activated). Hidden features can be exploited and used in collusion with an IP designer to insert malicious hardware Trojans and compromise the security properties of the IP. For example, hidden features inserted into the design by the HLS developer can be used by the colluding IP designer of a cryptographic core to mark sensitive points of the computation. The real attacker can later access the device and extract the encryption key from the compromised IP. HLS tools are more attractive for Trojan insertion compared to compilers or other CAD tools (e.g., logic synthesis and layout tools) for the following key reasons:

1) While software Trojans can be patched, hardware Trojans are permanent and can create significant economic damages for the users of compromised IPs;
2) HLS can insert a Trojan in early stages of the design flow (e.g., before logic synthesis or layout), and hence, can create a malicious attack that can impact not only functional but also non-functional properties of the design (e.g., latency and energy consumption).
3) There is no formal verification technique able to verify the RTL generated by HLS against the C specification in any circumstances [10], [12]. Therefore, it is more difficult to detect the Trojans inserted during HLS especially when they do not affect the functional behavior.

Answering the question of how much a HLS tool can be trusted has not been previously studied [3] and is the focus of this research. To highlight the problem, this paper will demonstrate that *black-hat HLS tools* are easy to create with minimal alterations to the classic HLS tool flow.

A. Contributions and Paper Roadmap

This paper studies how an attacker can create a black-hat HLS tool that introduces malice when creating IP components, along with the potential functional and non-functional effects. We start by presenting a classic HLS flow (Section II) and follow it up with presenting our main contributions:

- a set of HLS-aided hardware Trojans that can backdoor the generated IP components (Section III).
- a proof-of-concept implementation of three attacks in an open-source HLS tool (Section IV).
- an analysis of the effects of these HLS-generated hardware Trojans on HLS benchmarks. (Section V).

We used BAMBU, an open-source framework for research in HLS [13], to demonstrate that it is easy to compromise HLS tools with disastrous repercussions. Non-functional properties (like latency and energy consumption) are impossible to verify with the current technology, advocating novel specification and validation flows, along with proper security countermeasures.

II. HIGH-LEVEL SYNTHESIS IN A NUTSHELL

The IP blocks are specialized components that have superior performance and energy consumption than the corresponding software. HLS starts from a software-based specification (in C, C++, SystemC, etc.) to generate a Register Transfer Level (RTL) description (in Verilog or VHDL) ready for the rest of the design flow (i.e., logic synthesis and physical design).

As shown in Figure 2a, complex HLS-generated components are organized as modules to reduce the design complexity. The hierarchical organization reflects the sub-function organization of the specification. Each hardware component uses the classical Finite State Machine with Data (FSMD)

model [14] with two major components: *controller* and *datapath*. The controller determines the operations to execute in each clock cycle. The control flow is represented by a finite state machine (FSM) that sends signals to the datapath resources based on a set of conditions. The datapath contains the functional units to implement the accelerator functionality and registers to hold temporary values during the computation. Multiplexers drive the values based on the control flow.

The component generation can be divided into three parts, as shown in Figure 2b: ① the compiler phase, which investigates the input high-level description and applies compiler-level transformations; ② the HLS phase, which determines the microarchitecture, and ③ the backend, which generates the Verilog/VHDL description and the associated test benches. RTL simulation is performed on a set of pre-defined inputs to determine if the generated results match the golden ones obtained from software execution. While HLS-generated IP components are energy-efficient and have high performance, the IP component cannot be altered to add or remove functions after fabrication. Hence, the IP characteristics (e.g., performance and energy consumption) depend on the microarchitecture generated during HLS.

A. Compiler Phase

A traditional HLS flow interacts with compilers (e.g., GCC or LLVM) to parse the input C code, apply compiler optimizations, and produce the resulting intermediate representation (IR) [9]. Most HLS tools exploit the Single Static Assignment (SSA) form [15] so that the IR can be conveniently manipulated and translated into hardware (RTL) by the subsequent HLS steps. Compiler transformations like loop unrolling, constant propagation, function inlining are applied to optimize the code and enable HLS optimizations (e.g., extraction of instruction-level parallelism). The call graph, which represents the relations among the functions remaining after optimization, determines the list of modules to synthesize and their hierarchical interconnection.

B. HLS Phase

Each function of the IR is hierarchically transformed into a hardware module. After the selection of resources (called *allocation*), scheduling is performed to determine the operations to be executed in each clock cycle thereby determining the latency of the circuit. This step also generates the structure of the finite-state machine (FSM) controller, which implements the control-flow management of the accelerator during execution. Operations scheduled in various clock cycles can potentially reuse the same resources. Temporary values spanning the clock boundaries are stored in registers. Different algorithms can be used for *scheduling* and *binding* problems [16]. The last step is *interconnection*, where the functional units and registers are interconnected with multiplexers. Ultimately, the controller FSM is generated (*controller synthesis*). Based on the operations to execute and the microarchitecture, the FSM generates the signals that drive the data in the datapath through the multiplexers in each clock cycle.

C. Backend Phase

The VHDL/Verilog description of the component is hierarchically generated, together with the description of the library components (e.g., custom operators or memory interfaces) used in the design. While C-to-RTL formal verification is still an open research problem [10], RTL simulations is generally preferred to validate the IP behavior. So, HLS tools also generate the corresponding hardware test bench or an interface for co-simulation with the software test bench. A pre-defined set of inputs is used to generate the golden output values, which are then matched with the simulation results. Advanced debugging methods like discrepancy analysis automatically detect the bug location in case of functional mismatches between hardware and software [12].

III. Attacks on High-Level Synthesis CAD

IP components execute specialized functions for which they have been designed. However, an attacker can modify the HLS to add extra functionality and compromise the IP component. Differently from software infections, such malicious functionality cannot be removed after the chip fabrication. This section analyzes three attacks that we envision to be performed during HLS. These attacks influence the performance of the component (*Degradation*), expedite aging or battery consumption (*Battery Exhaustion*), and reduce the security level of the component (*Downgrade*). For each attack, we present the threat model, the malicious impacts and the requirements for its integration within a HLS flow so that the hardware Trojan stays undetectable during IP verification.

A. Threat Model

The use of malicious CAD tools is an emerging design threat [3]. In this work we assume that the attacker is a rogue developer of a HLS tool. While the risk of detection for major CAD developer is high, rogue employees can install malicious modifications into the tools through the alteration of regular updates to specific users. In addition, many semiconductor design houses rely on small companies for the design of specific IP components. There is no guarantee that all these companies are operating in good fait. IP-design companies with internal HLS tools might be interested in delivering altered IP components or license compromised software to some users for personal interests.

B. Degradation Attack

Computation-intensive kernels that are reused across different applications are excellent candidates for hardware acceleration [17]. Since component reuse is one of the key concepts in reducing design costs, many SoC companies either procure ready-to-use IP or use HLS tools on high-level specifications to create reusable IP. The quality of the resulting SoCs and the market revenues depends on the success of these hardware IP. Faulty units are easily detected and the vendor removes them from the market before selling them. On the contrary, it is responsible for the source of premature failures, which are hard to pinpoint, since they are manifested a long time after

the component is in production. Therefore, an attacker is more interested in weakening the HLS tool to impede the long-term quality of the competitor products.

1) Attack Model: A malicious HLS tool can compromise the quality of IP components by inserting a hardware Trojan that degrades the IP performance when activated. This can be used by a rogue employee to force aging degradation after a certain period of time and undermine the perceived quality and the endurance of the components. Consider the following C specification:

```
int FIR(int ntaps, int sum) {
  int i;
  for (i = 0; i < ntaps; i++) {
    sum += h[i] * z[i];
  }
  return sum;
}
```

Fig. 3. FIR filter described in C code.

This describes a simple FIR filter to be accelerated in hardware. Input values are stored in a local memory filled before the component execution. HLS can apply transformations to optimize the performance of the resulting IP component. For example, loop unrolling and multi-port memories can be combined to execute more operations in parallel [18]. An example of resulting FSM is shown in Figure 4a, where the loop body is repeated for a particular number of times based on the value of the parameter **ntaps**.

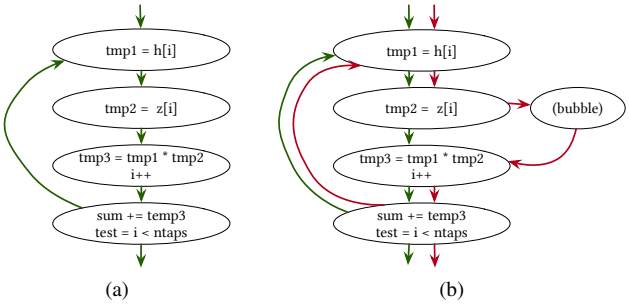


Fig. 4. Degradation attack on the HLS-generated FSM. (a) Original FSM. (b) FSM altered with bubbles.

A malicious attacker can insert alternative paths in specific points of the computation. These paths are meant to contain a few empty FSM states (*bubbles*) instead of useful operations. For example, in Figure 4b, the attacker inserted a bubble after the two memory-read operations. The new path, whose transition edges are colored in red, is executed only when the Trojan is activated, slowing down the computation. From the functional viewpoint, the component performs the exact same function but it takes more cycles to execute when the Trojan is triggered. When the bubbles are inserted in the loop bodies, the latency overhead is repeated as many times as the loop is executed. Since the designer does not possess any information about the FSM before HLS, it is usually hard to determine whether an FSM transition is valid or not. Therefore, traditional logic synthesis-based FSM anomaly detection techniques cannot be applied [19], [20], [21].

2) Effects: Inserting bubbles to degrade performance does not require modifications to the datapath. The area overhead is due to the extra resources needed to implement the bubble states in the controller. Additional flip-flops may be needed to encode the FSM states, while extra logic ports are needed to encode the additional elements of the output and next-state functions. Bubbles have negligible effect on the resource requirements of the IP component. In the example shown in Section III-B1, the area overhead is less than 1% when one bubble is inserted. A similar value is obtained when two bubbles are inserted. The bubbles impact the latency of the computation; no valid computation is performed in these bubble states. Consider an 8-tap FIR filter (i.e., the accelerator is configured to execute with **ntaps** = 8). For this filter, the performance overhead is 3% when one bubble is inserted and 6% when two bubbles are inserted. This overhead is high since the number of cycles required to implement the loop body is relatively small compared to the number of bubbles.

3) HLS Implications: The attacker must consider a few details to insert the bubbles into the FSM. First, the degradation effects must be noticeable after a certain amount of time. In fact, if the Trojan is always active and the component is always slowed down, the designer might decide not to purchase the IP core or the HLS tool. On the contrary, to effectively thwart the company using the compromised IP core, the HLS tool should generate a component that is competitive during its normal execution time and becomes slower only after a predefined amount of time or in case of specific input sequences. Then, the IP component should perform the correct computation in any case, only introducing performance degradation. To ensure a correct computation, bubble states must be designed to avoid computational errors during the transitions between the correct FSM states and the added paths with the FSM bubbles. Each FSM state must finalize the operations that terminate in the corresponding clock cycle (storing the output values in the registers) and creating the new ones (enabling the correct values at the input ports of the assigned functional units). If FSM bubbles are designed to maintain this behavior, the resulting FSM escapes even C-to-HDL formal verification. For example, the compromised IP passes sequential equivalence checking because this verification approach is designed to ignore timing introduced by HLS [22].

C. Battery Exhaustion Attack

SoCs are often battery-powered, where efficient energy management is a key design principle. For example, one of the most important criteria when shopping for a new phone is its standby time (the duration of a single battery charge). Hence, SoC-based devices are created with IP cores modified for specialized functions. Contemporary cyber-physical systems involve energy-efficient solutions and may quit operating when the battery drains. This can have repercussions, varying from loss of product competitiveness (planned obsolescence) to system failure. Battery exhaustion can be pulled off by including useless functionality that drains a considerable amount of current from the battery. This translates into extra power consumption when it is switched on, shortening battery time with no impact on the functionality.

1) Attack Model: Battery exhaustion attack entails altering the datapath to consume a substantial amount of current when the Trojan is activated. The Trojan should be designed on one hand to expedite battery discharging and on the other hand to evade power side-channel analysis. The power consumed during normal execution must be acceptable to avoid detecting the Trojan or simply rejecting the component for excessive power consumption. Consider the following C specification:

```
void decode (int input)
{
  int i;
  long int xa1, xa2;
  const int *h_ptr;
  int *ac_ptr, *ac_ptr1, *ad_ptr, *ad_ptr1;
  ilr = input & 0x3f;
  ih = input >> 6;
  d_szl = filtez (d_del_bpl, d_del_dltx);
  d_spl = filtep (d_rlt1, d_al1, d_rlt2, d_al2);
  d_sl = d_spl + d_szl;
  d_dlt = ((long) d_detl * qq4_tab[ilr >> 2]) >> 15;
  dl = ((long) d_detl * qq6_tab[il]) >> 15;
  rl = dl + d_sl;
  ...
}
```

Fig. 5. Initial part of the `adpcm` function **`decode`**.

This code snippet describes the initial part of the Adaptive Differential Pulse Code Modulation (ADPCM) decoder. This algorithm is extracted from the CHStone benchmark suite (`adpcm` benchmark) [23]. The four operations required to compute **`d_sl`**, **`d_dlt`**, **`dl`** and **`rl`** are scheduled in three clock cycles due to some data dependencies. The resulting scheduling is shown in Figure 6a.

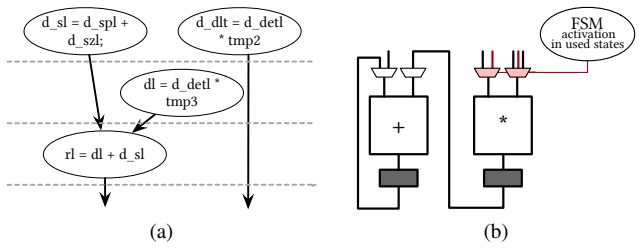


Fig. 6. HLS-generated component with idle functional units that can be reused for battery exhausting attacks. (a) Scheduled DFG. (b) Portion of the datapath.

Since the two multiplications are executed in different clock cycles, they can be assigned to the same functional unit. The same applies to the two additions, which are assigned to the same adder. The resulting portion of the datapath microarchitecture is shown in Figure 6b. When the operations are executed, the results are stored in the output registers. These register-write operations are regulated by the controller, which enables the signal **`wr_en`** during the respective clock cycles. However, these two functional units are not used in all clock cycles. The attacker can thus assign different input values in these clock cycles, triggering the execution of the combinational functionality and enhancing the dynamic power consumption of the component. If the result of this spurious operation is not stored into any register, there are no computational errors and the Trojan is hard to detect. Since most of the logic is reused, it is also more difficult to generate the golden reference model needed to detect this Trojan using power analysis.

2) Effects: Reusing idle functional units is a powerful technique to perform battery exhaustion attacks. These effects are amplified if the selected units have a larger dynamic power consumption, like multipliers. The area overhead for these alterations is negligible and so the additional static power. They only require a multiplexer to select the correct or fake inputs to the unit and the logic to generate these inputs in a manner that forces a high switching activity. For example, the compromised IP generated from the code in Figure 5 has an area overhead of around 2%, while the power consumption is enhanced by almost 10% when only five idle functional units are used for implementing the attack.

3) HLS-Level Implications: Extra functions responsible to consume additional power have no functional side-effects. So, the corresponding functional units might be optimized out by synthesis tools when their results are never used. This can be solved by reusing functional units that are already present in the datapath for implementing the functionality, as mentioned above. Output registers are used in valid computation, while register-write operations are disabled in case of fake operations. These units can be identified by analyzing the scheduling and resource binding resulting after the HLS phase. To force the peak switching activity, we can bit-flip the values currently present as input to the selected functional units. For example, let **`val`** be the current value connected to the input **`in`** of a given functional unit (e.g., the output port of the register where the previous value was stored). The connection can be modified as follows to consume extra current:

`in` = **`sel`** ? ∼**`val`** : **`val`**;

where **`sel`** is a signal representing whether the Trojan is activated. This attack is almost impossible to detect since there are no functional side-effects. Entire sub-modules can be used to consume a substantial amount of power in the same manner as we propose for single functional units. However, the sub-modules must have specific attributes to be used for battery exhaustion attacks. First, they must not have side-effects out of the core, like memory-write operations. Then, the output of the module must be stored in a register and not directly used by the datapath resources of the outermost function. In this way, the Trojan can inhibit the register-write operation of the output value as for the functional-unit operations.

Battery exhaustion attacks can not perform when there are no states with unused functional units. In these cases, a malicious HLS tool must create new states to implement the attack. These states are similar to *bubbles* for downgrade attacks and this is the only case that introduces performance overheads regarding the total number of clock cycles. However, since there is no golden circuit, the designer is not able to detect it.

D. Downgrade Attack

Besides the algorithm to implement, a compromised HLS tool can inject a malicious functionality. For example, malicious functions can weaken the security properties of symmetric key cryptographic algorithms (e.g., AES [24] or SHA [25]).

These algorithms use a typical computational style of bit manipulations repeated a pre-determined number of *rounds*. Reducing the number of rounds allows key recovery and undermines the security. Since this attack involves insight into the algorithm, it cannot be fully automated. This attack can be launched by a rogue in an IP design house that has its own HLS tool. A malicious developer can design hidden directives that can be used during IP development to access the undocumented features and generate compromised IP components. These components can be later accessed by the real attacker to extract sensitive information.

1) Attack Model: To compromise the security of round-based crypto algorithms, it is sufficient to reduce the number of executed rounds. This entails modifying the counter of the executed round. Consider the following C specification:

```
#define R(a,b) (((a)>>(b)) | ((a)<<(32-(b))))
#define CH(x,y,z) (((x)&(y)) ^ (~(x)&(z)))
#define MA(x,y,z) (((x)&(y))^((x)&(z))^((y)&(z)))
#define EP0(x) (R(x,2) ^ R(x,13) ^ R(x,22))
#define EP1(x) (R(x,6) ^ R(x,11) ^ R(x,25))
#define SHArounds 64
// array of constants
static const uint32_t k[64] = {/* omitted values */};
// hash function
void sha256_transform(void *state, const char in[]){
    // internal state variables
    uint32_t a,b,c,d,e,f,g,h,i,t1,t2,m[64];
    // [omitted] setup internal state variables
    // execution of 64 rounds
    for (i = 0; i < SHArounds; ++i) {
        t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
        t2 = EP0(a) + MA(a,b,c);
        h = g; g = f; f = e; e = d + t1;
        d = c; c = b; b = a; a = t1 + t2;
    }
    // [omitted] update external state (*s)
}
```

Fig. 7. C implementation of SHA-256 hash. The main loop has 64 *rounds*.

The corresponding microarchitecture is shown in Figure 8. Such round-based security algorithms can be compromised by *downgrading* them [26], [27]. For example, message pairs with collision can be generated for SHA-256 when reducing the rounds from 64 to 18 rounds [26], while key recovery for AES-128 is simplified when reducing the rounds from 10 to 7 [27]. This can be easily accomplished by modifying the loop constant (**`SHArounds`**) or by pre-loading a value into the counter variable **`i`** greater than 0. In both cases, the component executes a reduced number of rounds. In Figure 8, the part affected by the modification is highlighted in red.

This attack requires a collusion between the HLS developer and the IP designer to identify how many rounds must be executed and where the modification must be inserted (e.g., which are the counter variable and the constant defining the number of rounds). However, the IP designer, who is usually responsible for IP validation, can simply ignore the mismatch between the expected and the real outputs when it occurs.

2) Effects: Downgrading minimally impacts area since the modification is limited only to the round counter logic. So, the effects on the power consumption are limited and, when the Trojan is inactive, power analysis is not able to identify

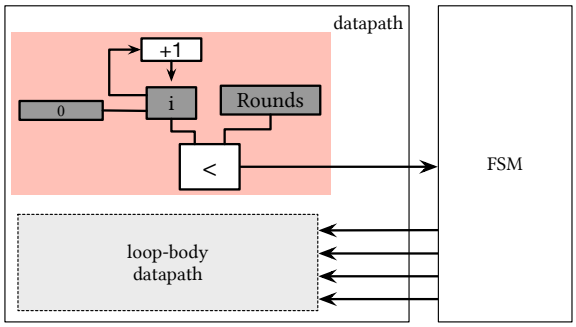


Fig. 8. Round-based microarchitecture of SHA-256.

any anomalies. When the Trojan is activated, the component executes a reduced number of rounds and this compromises the security of the IP core. For example, AES encryption with a reduced number of rounds aids key recovery [27]. Reducing the number of rounds also reduces the energy consumed. However, this is a benevolent effect that is hardly perceived by the user as a problem to be further investigated.

3) HLS Implications: Designing backdoors that support downgrading requires not only an analysis of the microarchitecture generated by HLS, but also an understanding of the computation. While it is possible to analyze at the compiler-level to automatically identify the parts of the specification involving the round computation, it is easier to bring insights from the IP developer including approaches to compromise it. Once the rogue IP developer specifies which counter must be "downgraded" and the maximum number of new rounds, the black-hat HLS tool can create and insert the backdoor. The backdoored design is created in a way to escape the verification steps following HLS even when this is not performed by the IP designer. As discussed in Section II, C-to-RTL verification is not established and designers rely on simulation-based approaches. Hence, the rogue IP developer can create a Trojan that is activated by a very rare condition, which is not provided as a test case to the user.

IV. Implementing Black-Hat HLS

This section outlines how to build a black-hat HLS tool on top of BAMBU, an open-source HLS framework [13] based on GCC, and insert the Trojans discussed in Section III. BAMBU is a command-line tool with a modular organization. Additional optimizations can be activated by passing specific flags when invoking the tool, while each synthesis step is implemented as a transformation step, called *pass*. As illustrated in Figure 9, these attacks are implemented as additional passes introduced by the rogue HLS developer. These passes can be activated to inject Trojans. To achieve a *Degradation Attack*, black-hat BAMBU determines the specific points in the computation where bubbles are inserted and modify the FSM accordingly (details in Section IV-A). To perform *Battery Exhaustion*, black-hat BAMBU selects the functional units that are idle for a particular number of clock cycles in order to generate the microarchitecture with high switching activity (see Section IV-B). Compromising the security standard of round-based crypto algorithms requires identifying the variables and create the microarchitecture to enforce a reduced

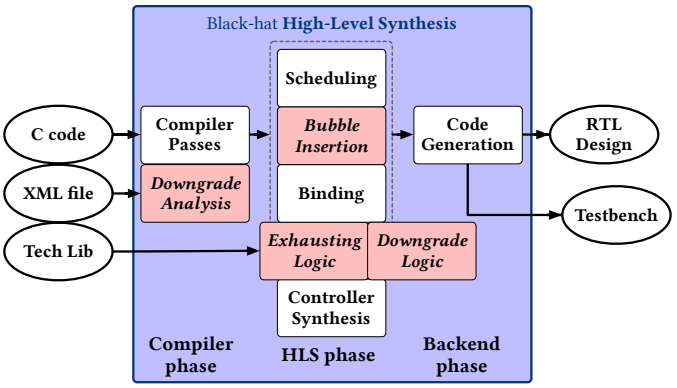


Fig. 9. Black-hat HLS constraints during classic HLS flow.

```
S_5 :                          S_5 :
begin                          begin
 wrenable_reg_9 = 1'b1;         wrenable_reg_9 = 1'b1;
 _next = sel ? S_9 : S_6;       _next_state = S_9;
end                            end
S_9 :                          S_9 :
begin                          begin
 _next = S_6;                   _next = (sel & s2) s? S_9:S_6;
end                             s2 = 0;
S_6 :                          end
begin                          S_6 :
 ...                           begin
                                ...
        (a)                            (b)
```

Fig. 10. Example degradation attack: (a) FSM with degradation attack that does not introduce performance degradation when the Trojan is inactive; (b) degradation attack that escapes detection using code coverage analysis.

number of rounds (see Section IV-C). All these rogue steps can be enabled via command-line options and are executed between the HLS phase and the back-end phase.

A. Degradation Attack

To maximize the impact of the FSM bubbles, they must be inserted in those points of the computation that are repeated multiple times (e.g., loop bodies). Therefore, black-hat BAMBU uses the scheduling information and implements the algorithm shown in Algorithm 1. The algorithm starts from the baseline FSM resulting from the scheduling of the input functionality and inserts a pre-defined number of bubbles defined by the HLS developer (N). Each alternative path is selected when the Trojan is activated.

ALGORITHM 1: Insert FSM bubbles for a degradation attack.

```
Procedure InsertFSMBubbles(FSM, N, trigger)
    Data: FSM is the baseline FSM as it comes out from the scheduling
          step; N is the number of bubbles to be added; trigger is the
          activation signal of the Trojan
    Result: FSM_d is the FSM modified with bubbles
    BBList ← ComputeBBList(FSM)      // extract the BB
     list
    FSM_d ← CopyFSM(FSM)             // create a copy of the
     current FSM
    for b = 1; b ≤ N; b++ do
        foreach BB_i ∈ BBLis do
            C(BB_i, b) ← EstimateBBCost(BB_i)   // compute
             the cost of each BB
        end
        BB_sel ← ExtractMaxCost(BBList, C)      // Extract
         the BB with max cost
        (state_1, state_2) ← SelectState(BB_sel) // Select the
         pair of states where to insert the bubble
        FSM_d ← InsertBubble(FSM_d, state_1, state_2, trigger)
         // Add the bubble
    end
    return FSM_d
```

This algorithm works at the basic-block level. A basic block represents a part of the computation with a single entry point and a single exit point. Each basic block is scheduled and translated into a sequence of states executed serially. The *cost* of the basic block BB_i is defined as follows:

$$C(BB_i, b) = \frac{states_{BB_i} * exec_{BB_i}}{b+1} \quad (1)$$

where $states_{BB_i}$ is the number of states required to perform the basic block BB_i, $execBB_i$ is an estimate of the number of times the basic block executes (e.g., the number of iterations of the loop). b counts the bubbles already introduced in the same part of the code. This is useful when assigning the bubbles in the various parts of the computation. The algorithm selects a pair of states to introduce a bubble. In each basic block, it selects the state with fewer operations executed during the corresponding clock cycle to minimize the number of FSM signals used. Each bubble has two states to maintain functional correctness. The first state has the signals of state $state_2$ to finalize the operations in state $state_1$ (e.g., register-write operations on the output values of the functional units). The second state has the activation signals of the operations launched in the $state_2$. The first bubble state attaches to $state_1$ based on the trigger signal and the second bubble state attaches to $state_2$ as target state.

1) Attack Detection: **Sequential Equivalence Checking (SEC)** methods [28] aim at matching the behavior of specifications at different levels of abstraction, ignoring timing differences introduced by HLS. Since this attack does not introduce any behavior modifications but only timing differences, it cannot be detected with SEC methods. **Code coverage** can detect the baseline degradation attack. However, it is possible to implement a variant that cannot be detected by code coverage at the cost of a small additional overhead. In this variant, the bubble is traversed once in normal operation and multiple times when the Trigger is on. The two variants are shown in Figure 10. In the first version (see Figure 10a), there is no performance overhead when the Trojan is inactive but code coverage analysis identifies that the line in state `S_9` is not covered during normal execution. This might induce the designer to perform further analysis. On the contrary, when the tool adopts the solution of Figure 10b, code coverage analysis obtains the same results as in the baseline version (i.e., the one without Trojan) and the Trojan stays undetected. However, this solution has a small overhead (one cycle per execution) when the Trojan is inactive.

B. Battery Exhaustion Attack

To implement battery exhaustion attacks, the black-hat HLS tool must select which functional units are used to increase the power consumption. Similar to the performance degradation attacks, battery exhaustion is implemented as part of the classic HLS. After module binding, it is possible to determine which

functional units are idle in each clock cycle. The algorithm for selecting such idle functional units is shown in Algorithm 2 that considers technology-related information (e.g., estimated dynamic power consumption of each type of functional unit).

ALGORITHM 2: Algorithm to increase the power consumption of unused functional units.

Procedure `InsertExhauster`($arch$, $Bind_{FU}$, lib, FSM, $trigger$, $power$)
- **Data:** $arch$ is the current microarchitecture; $Bind_{FU}$ is the current FU binding; lib is the technology library with power information; FSM is the current FSM; $trigger$ is the activation signal of the Trojan; $power$ is the amount of additional power to be consumed
- **Result:** $arch$ is the updated microarchitecture
- $FUList \leftarrow$ `ComputeFUList` ($Bind_{FU}$) `// extract the FU list`
- $p \leftarrow 0$ `// initialize the additional power`
- **while** ($p < power$) **do**
  - $fu \leftarrow$ `SelectFU` ($FUList, lib, FSM$) `// select the FU to modify`
  - $p \leftarrow p +$ (`GetPowerFU` (fu, lib)$*$`UnusedEstimate` (fu, FSM)) `// save the extra power`
  - $arch \leftarrow$ `AddExhaustingLogic` ($arch, fu$) `// add the extra logic around` fu
- **end**
- **return** $arch$

Given a power consumption budget ($power$), the algorithm modifies the datapath and the FSM controller using a twofold objective: (1) select the functional units that maximize the dynamic power consumption, (2) minimize the number of functional units that are extended with the extra logic to increase the switching activity while minimizing the extra hardware. In function `SelectFU`, the algorithm selects the functional units by estimating the dynamic power consumption (`GetPowerFU` – based on library information) and the number of cycles in which they are idle during normal execution (`UnusedEstimate` – based on profiling details). The information concerning power consumption comes from the technology library used for synthesis, while profiling information is gathered to prefer functional units that are unused regardless the control flow. The resulting contribution is used to update `p` until the power budget is reached. In terms of complexity, the algorithm iterates over all functional units of the design ($FUList$) in the worst case, i.e., the worst case complexity is linear in number of functional units.

Figure 11 shows the microarchitecture that is added to each functional unit to increase dynamic power consumption.

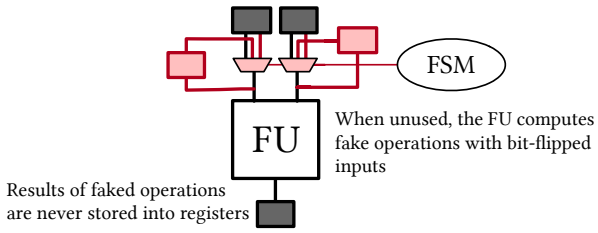


Fig. 11. Modifications to the functional units to increase power consumption.

The functional-unit inputs are stored in extra register with bit-flipping. The bit-flipping with NOT gates and extra registers guarantees the maximum switching activity for each combinational functional unit. Most of the functional units are connected to flip-flops that contain the input values. Often, these flip flops have an additional port with the complement of the output that can be directly connected to the extra logic.

1) Attack Detection: **Sequential Equivalence Checking:** Similarly to degradation attacks, this attack does not introduce any behavior modifications and SEC methods are not able to detect anomalies. **Code Coverage:** Since all instructions are executed with the same frequency, *code coverage* [29] is not able to determine any potential anomaly in the design. The single line of code shown in Section III-C3, which activates the Trojan, is also covered.

C. Downgrade Attack

The modifications to the IP for the downgrade attack are not automatically implemented inside black-hat BAMBU but rely on annotations from the colluding IP designer. So, to create a compromised IP component, the rogue IP designer marks the variable that is used to track the number of executed rounds and the black-hat HLS flow introduces the microarchitecture to execute a reduced number of rounds. The variable, along with its scope (i.e., the function where it is executed) and the number of rounds are specified through an XML file. Then, the attacker may have an interest in accessing the device, activating the Trojan, and extracting private data only in specific situations. Therefore, we design a trigger to activate the Trojan only in rare conditions. Such a microarchitecture is shown in Figure 12 and is implemented by a multiplexer that pre-loads a non-zero value into the counter.

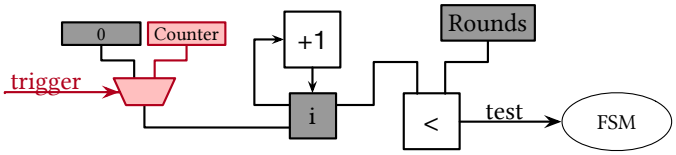


Fig. 12. Microarchitecture of the backdoor that triggers the downgrade attack. At the beginning of each execution, the counter is preloaded with a value different (including zero) to reduce the number of rounds.

This implementation and the associated overhead are independent from the number of rounds to be executed. Let $Rounds$ and $Reduced\_Rounds$ be the number of rounds normally executed and the number of rounds that undermine the security, respectively. The counter is pre-loaded with the value $Counter$ computed as follows:

$$Counter = Rounds - Reduced\_Rounds \qquad (2)$$

The FSM is modified to execute a reduced number of iterations with a dedicated implementation in case of partial or complete unrolling. Instead of modifying the constant to terminate the loop, this implementation operates on the register variable and enables optimizations on the comparator.

1) Attack Detection: **Sequential Equivalence Checking**: The compromised IP generates behavioral mismatches, which can be exploited to ease key recovery, only when the Trojan is activated, i.e., the attacker gains control of the IP and provides the activation sequence. As long as the Trojan is not triggered, the IP produces correct results and the Trojan stays

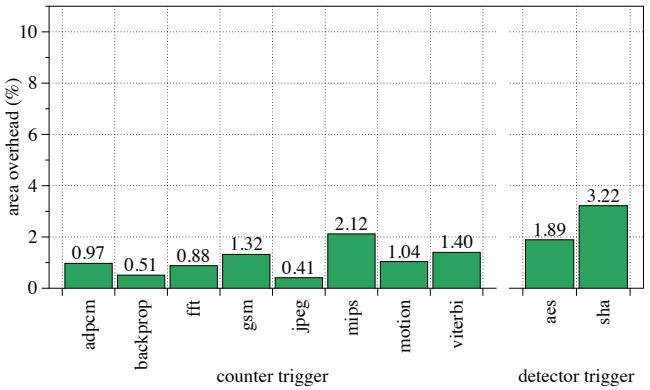


Fig. 13. Area overhead of the two trigger functions with respect to the **`baseline`** implementations.

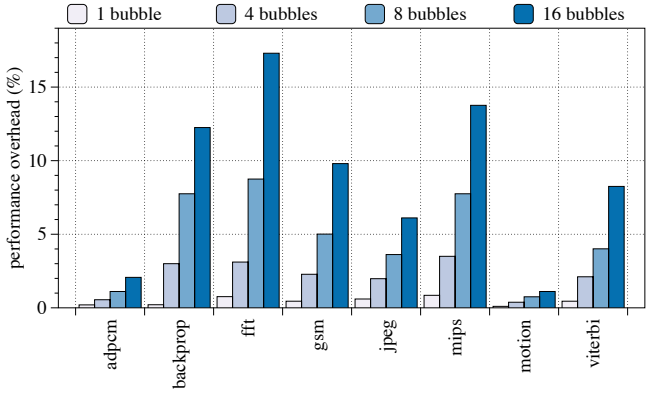


Fig. 14. Performance overhead caused by the degradation attacks when varying the number of bubbles.

undetected. **Code Coverage**: The only design modification affects the value of the counter (see Figure 12). However, the corresponding HDL line is always executed to determine to value to pre-load into variable **`i`**. All lines are thus covered.

V. Effects of HLS Attacks

We evaluated the attacks implemented in black-hat BAMBU. We added new command-line options to selectively activate the *malicious passes*. These options are not listed in the `help` menu, mimicking the situation where malicious and undocumented features are added to the HLS tool.

We employed the original BAMBU and black-hat BAMBU HLS frameworks to create accelerators for selected kernels from CHStone [23] and MachSuite [17], two HLS benchmark suites. All benchmarks are specified in C language. BAMBU generates hardware test benches and performs simulation-based validation on an array of input values provided by the user. In each experiment, we used the Synopsys SAED 32nm Generic Library at 500 MHz. Results pertaining to performance are obtained through RTL simulation with Mentor Graphics ModelSim SE 10.3. The same simulations are also used to compare the golden results obtained by executing the C code on the input values and the ones obtained from the hardware circuits. Area and power results are obtained after logic synthesis and power analysis with Synopsys Design Compiler J-2014.09-SP2. These values represent the **`baseline`** of our experiments. The area overhead is computed only on the logic resources, without considering the SRAM. RTL simulations are used to extract SAIF back-annotations with accurate switching activity for precise power analysis. We used Synopsys VCS simulator for code coverage analysis. In all cases, we obtained the same results as in the baseline versions.

Next, we explain the activation functions of the Trojans and the effects of the attacks in terms of performance, power and security when they are activated.

A. Trojan Triggers

The trigger functions of the Trojans must be designed according to the use of the generated IP core and how the attacker plans to compromise the resulting design. For degradation and battery-exhaustion attacks, the attacker is not interested in accessing the device, but only to compromise its execution after a predefined amount of time. In these two cases, we use a *counter-based trigger* (**`counter`**) that activates the Trojan after 65,536 executions (i.e., the counter has 16 bits). Designing Trojans that are activated after a larger number iterations requires simply to add a few bits to the counter register. For downgrade attacks, we design an *input sequence detector* (**`detector`**), where the colluding IP designer can specify the input sequence that must be provided to activate the Trojan. The implementation of this unit is simple as it is conceptually equivalent to a string detector.

In both cases, the size of the trigger modules is independent of the functionality of the IP core. Figure 13 reports the impact of such modules on the area of the entire IP cores (**`baseline`** versions). These results demonstrate that trigger functions minimally impact the area (less than 4% in the worst case, being negligible in most cases).

The trigger functions are completely orthogonal to the effects of the three attacks and more complex trigger mechanisms can be easily integrated. However, we have shown that these simple triggers are enough to activate the intended Trojans in the different designs.

B. Trojan Payloads

We will discuss the functional and non-functional effects of the proposed Trojans.

Degradation Attack: In case of degradation attacks, we created several variants of compromised IP cores. Each of them has a different number of FSM bubbles. We evaluated the area and performance overheads of these malicious modifications.

The area overhead is negligible in all cases (less than 1%). Indeed, there are no modification to the datapath, while the controller only needs few extra logic gates for the transition and output functions.

Figure 14 illustrates the performance overhead with respect to the **`baseline`** versions when varying the number of bubbles. As expected, the overhead is significant and grows as we increase the number of FSM bubbles introduced in the designs. The loop-intensive IP cores are more affected by the bubbles, such as the `fft` reaching a performance overhead up to 17%. Trojans inserted in cores where the sequential

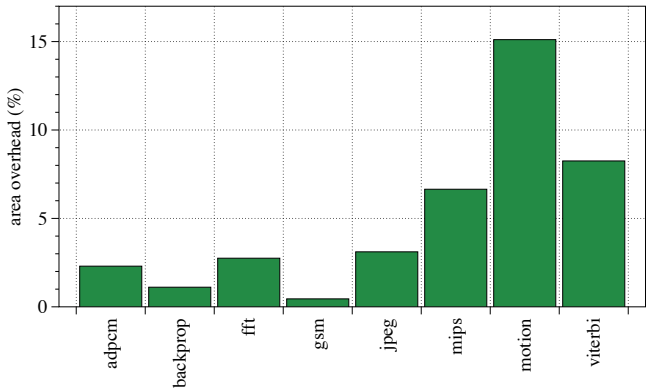


Fig. 15. Area overhead for battery-exhaustion attacks when imposing a 30% more power budget with respect to the **baseline**.

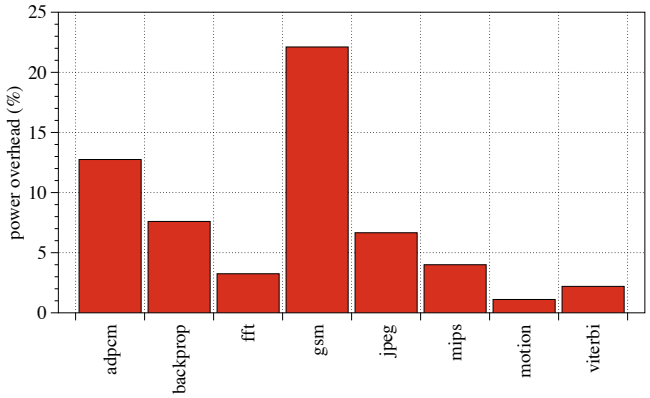


Fig. 16. Power overhead when only four functional units participate in the battery exhaustion attacks.

execution is much larger than the code contained in the loops have a minimal impact (see `motion`). In these cases, the loops have a small impact and cannot maximize the bubble effects. Occasionally, the algorithm does not insert the bubbles in proper points (e.g., `adpcm`). This is due to a heavy use of pointers and submodules that complicate the computation of the BB costs defined in Equation 1. The optimization of the cost function and the automatic identification of the appropriate number of bubbles to insert are still open problems. The RTL simulations report correct results in all cases, indicating that the degradation attacks introduce no computation errors. Thus, neither simulation-based verification, nor sequential equivalence checking can detect these Trojans.

Battery Exhaustion Attack: We designed two sets of experiments. First, we determined a particular power budget and evaluated the area overhead necessary to achieve it applying the black-hat algorithm proposed in Section IV-B. Then, we used a simplified version of the same algorithm, where we selected the four most significant functional units (function `SelectFU` of Algorithm 2) and evaluated the power overhead that can be obtained. In the first set of experiments, we estimated the power consumed by each **baseline** implementation and assigned an extra 30% power budget to be consumed when the Trojan is activated. Figure 15 reports the area overhead to implement the extra logic to increase the power consumption (see Section IV-B). The area overhead is generally quite low and in some cases, negligible. For `gsm`, a considerable amount of power consumption can be achieved by exploiting a small set of functional units with an area overhead less than 1%. On the contrary, adding extra logic to the benchmarks with many Boolean operations (e.g., `motion`) has a larger impact on the area (around 15%). Indeed, the logic added to increase power consumption (see Figure 11) has an area comparable with the relative functional units.

The second set of experiments reduces the area overhead by modifying only up to four functional units. This way, we obtained an area overhead that is less than 3%. The resulting power overheads with respect to the **baseline** implementations are illustrated in Figure 16. As in the previous case, the benchmarks with idle functional units can achieve a substantial power overhead (over 20% in case of `gsm`). This attack has a limited impact on simple benchmarks composed of Boolean operations, like the `motion`. RTL simulations report correct results in all experiments, confirming that battery exhaustion attacks do not introduce computational errors that can be detected with either simulation-based verification or by sequential equivalence checking.

TABLE I
DOWNGRADE ATTACKS TO SHA-256 AND AES-128 ALGORITHMS.

Algorithm	No. Rounds	Trust Level	Overhead Area	Overhead Power
SHA-256	80	✓	-	-
	64	✓	+2.72%	<0.01%
	48	✓	+2.01%	<0.01%
	18	✗	+2.21%	<0.01%
AES-128	10	✓	-	-
	9	✓	+2.97%	<0.01%
	8	✓	+2.74%	<0.01%
	7	✗	+2.84%	<0.01%

Downgrade Attack: We applied the black-hat BAMBU to implement the downgrade attack described in Section IV-C on two cryptographic cores: SHA-256 and AES-128. They are two of the most popular (round-based) algorithms for which several hardware implementations were explored and also compromised [26]. Starting from the `C` descriptions publicly available, we specified the round counter variables in the XML file that is passed on to black-hat BAMBU, together with the list of four rare input values to generate the **detector** trigger of the Trojans (see Section V-A). Black-hat BAMBU generates the microarchitecture of these two cores augmented with the Trojan logic shown in Figure 12. The trust-level of these algorithms depends on the number of iterations that are executed. If the algorithms execute a number of rounds below a certain value, the cores become vulnerable. For each benchmark, we designed different versions concerning the number of rounds to execute. Table I reports the values that we used together with the area/power overhead. From the functional perspective, we performed extensive RTL simulations of the generated hardware with several input values that were randomly generated. These simulations did not activate the Trojans in any cases, indicating that the **detector** trigger and the given input sequence is robust against accidental activations during validation or normal execution. Next, we provided the trigger sequence of input values to activate the

TABLE II
COMPARATIVE ANALYSIS OF THE THREE HLS-BASED HARDWARE TROJANS GENERATED BY BLACK-HAT BAMBU.

Attack	Attacker's goal	Trigger	Implementation & Main effect	Side effects	Benchmark Impact
Degradation	Worsen the performance after a pre-defined amount of time	Counter of IP executions	Extra FSM states with no operations to increase the execution latency, perceiving a component degradation	Limited area overhead in the controller due to extra encoding flip-flops and related logic	Low impact on heavily sequential circuits with small loops
Battery Exhaustion	Increase power consumption to accelerate battery consumption after a pre-defined amount of time	Counter of IP executions	Fake operations in unused FUs to increase dynamic power consumption	Limited area overhead to provide fake inputs with high switching activity; no performance overhead unless all FUs are always used	Large overhead in case of many Boolean operations
Downgrade	Compromise security of hardware IP cores to extract sensitive information (e.g., encryption key)	Input sequence detector	Pre-loading of the round counter with a value greater than zero to execute less rounds	Negligible area/power overhead; behavioral mismatches only when the Trojan is active	Attack independent of the number of rounds

Trojans and we verified that the IP executes the reduced number of iterations requested by the attacker.

The area overhead of the compromised IP cores is negligible (less than 3% – see Table I) regardless the trust-level of the cores (i.e., the number of iterations). The logic for the triggers (see Section V-A) accounts for most of the overhead (more than 2%), while the payload of these Trojans (the microarchitecture outlined in Section IV-C) is minimal since it requires only an extra multiplexer (less than 1%). Also the power overhead is minimal. It is worth noting that the key recovery is not performed with power side-channel attacks, but with analysis of the output values.

Final Remarks: Table II compares the three attacks. For each of them, we describe the attacker's goal and the type of trigger used for activation. Also, we report a brief description of the Trojan implementation along with its main effect when it is activated. Finally, we describe the the side effects (e.g., the overheads) when they are injected.

VI. RELATED WORK

Hardware Trojans are a serious concern for the security of SoC architectures [30]. They may have different characteristics and activities, varying from simple modifications of the circuits to alteration of the manufacturing process [31]. While there are several techniques to detect hardware Trojans (including machine-learning based approaches [32]), the problem of guaranteeing a Trojan-free chip is still unsolved. Hardware Performance Counters (HPCs) are architectural features that are used to monitor the system execution. However, they are not suitable for detecting Hardware Trojans. HPCs can detect malware assuming that the underlying hardware is trusted. If the hardware is untrusted (i.e., Hardware Trojans exist), this technique can not apply.

Existing threat models assume an attack during manufacturing since design houses outsource the chip fabrication to third-party foundries. Until now CAD tools were mostly assumed trustworthy [33]. HLS can generate additional logic to detect security violations or thwart attack [34], [35]. Trustworthy HLS-generated designs are also vulnerable to a wide range of security violations [8]. Malicious logic synthesis has been explored in [20], [36]. The research showed that logic synthesis tools can insert a Trojan by modifying the FSM of the design. However, the authors did not discuss about robustness of these attacks against post-synthesis verification tools like Logical Equivalence Checkers (LEC). We devised an experiment to determine whether commercial state-of-the-art LEC tools can detect these attacks. Let us consider a sequence detector, as shown using the FSM in Figure 17a. 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 logic-synthesis tool can take advantage of undefined transitions and create an FSM with an extra transition, as shown in red in Figure 17b. These types of attacks are generally known as *Transition-Based Attacks*. These extra transitions are used by a malicious synthesis tool to create vulnerabilities in the design without changing the original functionality. In the FSM described in Figure 17a, if one wants to travel from state B to state A, she has to travel via state C. In the modified FSM in Figure 17b, the malicious CAD tool creates a back-door to transition from B to A.

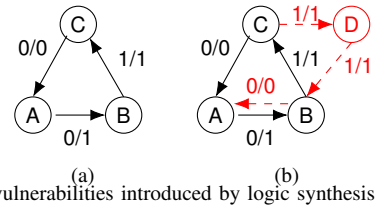


Fig. 17. FSM vulnerabilities introduced by logic synthesis tools.(a) Original FSM to detect pattern 001. (b) Transition attack. (c) Extra state attack

The malicious logic synthesis tool can introduce another attack, which entails adding an Extra State [19]. The malicious synthesis tool finds whether extra states can be added. Consider the FSM in Figure 17a. 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 red in Figure 17b. The additional state adds a back-door to the design. This attack (which we call Extra state attack) is similar to the "Degradation attack" proposed in Section III-B. These are the two attacks introduced by logic

```
LEC> compare
================================================================================
Compared points      PO     DFF    DLAT      Total
--------------------------------------------------------------------------------
Equivalent           1      2      0         3
--------------------------------------------------------------------------------
Non-equivalent       0      0      3         3
================================================================================
LEC>
```

Fig. 18. Screenshot of Cadence Conformal LEC report on comparison of designs after logic synthesis attack. The primary output and flip-flop counts are similar. However, there are new transitions in the FSM in Figure 17b, which gets detected as "non-equivalent".

synthesis tools as described in [19], [20], [21].

We used the original FSM in Figure 17a and synthesized it to a netlist. Both sets of vulnerabilities are introduced in the netlist. We use Cadence Conformal LEC tool for equivalence checking between the RTL and the netlist. Conformal LEC was able to detect both vulnerabilities, thus, validating our claim. The Conformal LEC report, shown in Figure 18, detects non-equivalent portions in the two designs.

Moreover, researchers have proposed several techniques to detect these class of attacks by logic synthesis tools [19], [20], [21]. Hence, logic synthesis attacks can be detected.

Although HLS-inserted Trojans are similar to logic synthesis-inserted o1nes, the proposed attacks are robust against these techniques since the designer does not have a FSM in the original design. Since the designer uses high-level languages like C, C++ or System C for HLS, he/she is agnostic to the states and transitions in the resulting FSM. Hence, the concepts of "extra transitions" and "extra states" don't exist for HLS. Although HLS is becoming popular as a productive tool for design [9], related verification methods are pre-mature. In the future, security verification must consider functional and non-functional properties of HLS-generated modules.

Layout tools can insert Trojans by modifying the GDSII files. They can use the vacant spaces in the design and insert a Trojan, which can be triggered using existing wires and signals [37]. This threat model is restrictive in that the attacker cannot increase the dimensions of the chip. He/she has to leverage white spaces in the layout to realize this attack. The layout-based attack is difficult to realize for two reasons:

1) The attacker has to rigorously reverse engineer the layout to identify signals that could enable the trigger.
2) The attacker does not have information about the test cases used to test the IP. The testing company might design an intelligent test case to detect this Trojan.

HLS-based Trojans can make larger modifications to the designs. During HLS, it is known which functional units are used/unused and it is easier to create extra connections without altering the functionality.

In the software domain, a compiler can insert a backdoor in a login program. This would allow a login whenever a special password (known only to the Trojan Horse designer) is entered [38], undermining the security of the system. This is similar to the downgrade attack. Extra routines can be added to perform useless computation and degrade performance or waste power consumption. However, these attacks can be identified and removed by substituting the compromised binary.

HLS verification techniques compare the golden function with the generated hardware to detect mismatches. Discrepancy analysis is a novel technique to identify differences between software and hardware execution [11]. However, one cannot identify non-functional discrepancies and attacks like performance degradation and battery exhaustion. In downgrade attacks, we assume collusion with the HLS developer. If the HLS tool is compromised, discrepancy analysis may be compromised as well. Multi-level attacks wherein two or more parties conspire have been demonstrated [39].

VII. Concluding Remarks

We presented a proof-of-concept implementation of a black-hat HLS framework that can introduce a variety of hardware Trojans that are challenging to detect. Classic HLS flows can be altered minimally to aid increase in latency or power consumption and to compromise the security of the generated IP cores. For example, the proposed Downgrade Attack can be used to obtain confidential information from the design. HLS and other CAD tools in general are potential attack surfaces advocating methods to verify functional and non-functional properties of the IP, along with proper countermeasures.

References

[1] S. Heck, S. Kaza, and D. Pinner, "Creating value in the semiconductor industry," *McKinsey on Semiconductors*, pp. 5–144, Oct. 2011.

[2] J. Hurtarte, E. Wolsheimer, and L. Tafoya, *Understanding Fabless IC Technology*. Elsevier, Aug. 2007.

[3] I. Polian, G. T. Becker, and F. Regazzoni, "Trojans in early design steps – an emerging threat," in *Proceedings of the Conference on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE)*, 2016.

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

[5] A. Hadhazy, "Here's the truth about the 'planned obsolescence' of tech," *BBC Future*, Jun. 2016.

[6] S. Bhunia and M. Tehranipoor, *The Hardware Trojan War - Attacks, Myths, and Defenses*, S. I. Publishing, Ed., 2018.

[7] Xilinx Inc., "Vivado design suite user guide - designing with ip (ug896)," 2017.

[8] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni, "Securing hardware accelerators: a new challenge for High-Level Synthesis," *IEEE Embedded Systems Letters*, vol. 3, no. 10, pp. 77–80, September 2018.

[9] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. 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, no. 10, Oct. 2016.

[10] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, Oct. 2017.

[11] P. Fezzardi, M. Castellana, and F. Ferrandi, "Trace-based automated logical debugging for high-level synthesis generated circuits," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Oct. 2015, pp. 251–258.

[12] P. Fezzardi, C. Pilato, and F. Ferrandi, "Enabling automated bug detection for IP-based designs using high-level synthesis," *IEEE Design & Test*, vol. 35, no. 5, pp. 54–62, Oct. 2018.

[13] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, Sep. 2013, pp. 1–4.

[14] J. Zhu and D. Gajski, "A unified formal model of ISA and FSMD," in *Proceedings of the International Workshop on Hardware/Software Codesign (CODES)*, 1999, pp. 121–125.

[15] D. Novillo, "Design and implementation of Tree SSA," in *GCC Developers' Summit*, 2004, pp. 119–130.

[16] L. Stok, "Data path synthesis," *Integration, the VLSI Journal*, vol. 18, no. 1, pp. 1–71, Dec. 1994.

[17] 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 International Symposium on Workload Characterization (IISWC)*, Oct. 2014, pp. 110–119.

[18] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. Carloni, "System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 435–448, Mar. 2017.

[19] 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 the ACM/IEEE Annual Design Automation Conference (DAC)*, 2016, p. 89.

[20] F. Farahmandi and P. Mishra, "Fsm anomaly detection using formal analysis," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 313–320.

[21] X. Sun, P. Kalla, and F. Enescu, "Word-level traversal of finite state machines using algebraic geometry," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2016, pp. 142–149.

[22] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *Proceedings of the ACM/IEEE Annual Design Automation Conference (DAC)*, 2009, pp. 460–465.

[23] 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, Oct. 2009.

[24] U.S. Department of Commerce and National Institute of Standards and Technology (NIST), *Advanced Encryption Standard - AES: Federal Information Processing Standards Publication 197*, Nov. 2001.

[25] ——, *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*, Mar. 2012.

[26] 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., 2008, pp. 130–143.

[27] 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*, 2013, pp. 371–387.

[28] A. Mathur, M. Fujita, M. Balakrishnan, and R. Mitra, "Sequential equivalence checking," in *Proceedings of the International Conference on VLSI Design (VLSID)*, Jan. 2006.

[29] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 36–45, Jul. 2001.

[30] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, Oct. 2010.

[31] G. Becker, F. Regazzoni, C. Paar, and W. Burleson, "Stealthy dopant-level hardware trojans: extended version," *Journal of Cryptographic Engineering*, vol. 4, no. 1, pp. 19–31, Apr. 2014.

[32] A. Kulkarni, Y. Pino, and T. Mohsenin, "Adaptive real-time trojan detection framework through machine learning," in *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2016, pp. 120–123.

[33] M. Potkonjak, "Synthesis of trustable ICs using untrusted CAD tools," in *Proceedings of the ACM/IEEE Annual Design Automation Conference (DAC)*, ser. DAC '10, 2010, pp. 633–634.

[34] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, "Belling the cad: Toward security-centric electronic system design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1756–1769, Nov. 2015.

[35] Y. Lao and K. K. Parhi, "Obfuscating dsp circuits via high-level transformations," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 5, pp. 819–830, May 2015.

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

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

[38] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

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

Christian Pilato is an Assistant Professor at Politecnico di Milano. He was a Post-doc Research Scientist at Columbia University from 2013 to 2016, and at the ALaRI Institute of the Università della Svizzera italiana until 2018. He received the Laurea degree in Computer Engineering and the Ph.D. degree in Information Technology from Politecnico di Milano, in 2007 and 2011, respectively. His research interests include high-level synthesis, reconfigurable systems and system-on-chip architectures, with emphasis on memory and security aspects. In 2014 Dr. Pilato served as program chair of the Conference on Embedded and Ubiquitous Computing (EUC) and he is currently involved in the program committees of many conferences on embedded systems, CAD, and reconfigurable architectures (e.g., FPL, DATE, CASES). He is a member of the ACM.

Kanad Basu received his Ph.D. from the department of Computer and Information Science and Engineering, University of Florida. His thesis was focused on improving signal observability for post-silicon validation. Post-Phd, Kanad worked in various semiconductor companies like IBM and Synopsys. At IBM, he was responsible for the design on IBM Power and Z Processors. At Synopsys, Kanad helped in development of DFTMAX Ultra, the state of the art low pin hardware test solution. Currently, Kanad is an Assistant Research Professor at the Electrical and Computer Engineering Department of NYU. He has authored 2 US patents, 2 book chapters and several peer reviewed journal and conference articles. Kanad was awarded the "Best Paper Award" at the International Conference on VLSI Design 2011. Kanad's current research interests are hardware and systems security.

Francesco Regazzoni is a Senior Researcher at the ALaRI Institute of University of Lugano (Lugano, Switzerland). He received his Master of Science degree from Politecnico di Milano and his Ph.D. degree at the ALaRI Institute of the Università della Svizzera italiana. He has been assistant researcher at the Université Catholique de Louvain and at Technical University of Delft, and visiting researcher at several institutions, including NEC Labs America, Ruhr University of Bochum, EPFL, and NTU. His research interests are mainly focused on cyber-physical and embedded systems security, covering in particular side channel attacks, cryptographic hardware, and electronic design automation for security.

Ramesh Karri is a Professor of Electrical and Computer Engineering at New York University. He co-directs the NYU Center for Cyber Security (http://cyber.nyu.edu). He also leads the Cyber Security thrust of the NY State Center for Advanced Telecommunications Technologies at NYU. He co-founded the Trust-Hub (http://trust-hub.org). He organizes the Embedded Systems Challenge (https://csaw.engineering.nyu.edu/esc), the global red-team-blue-team hardware hacking event.

Ramesh Karri has a Ph.D. in Computer Science and Engineering, from the University of California at San Diego and a B.E in ECE from Andhra University. His research and education activities in hardware cybersecurity include trustworthy ICs; processors and cyber-physical systems; security-aware computer-aided design, test, verification, validation, and reliability; nano meets security; hardware security competitions, benchmarks, and metrics; biochip security; additive manufacturing security. He has published over 200 articles in leading journals and conference proceedings.

Ramesh Karri' work on hardware cybersecurity received best paper award nominations (ICCD 2015 and DFTS 2015), awards (ITC 2014, CCS 2013, DFTS 2013 and VLSI Design 2012, ACM Student Research Competition at DAC 2012, ICCAD 2013, DAC 2014, ACM Grand Finals 2013, Kaspersky Challenge and Embedded Security Challenge). He received the Humboldt Fellowship and the National Science Foundation CAREER Award.

Ramesh Karri co-founded the IEEE/ACM Symposium on Nanoscale Architectures (NANOARCH). He served as program/general chair of conferences including IEEE International Conference on Computer Design (ICCD), IEEE Symposium on Hardware-Oriented Security and Trust (HOST), IEEE Symposium on Defect and Fault Tolerant Nano VLSI Systems (DFTS), NANOARCH, RFIDSEC, and WISEC. He serves on several program committees (HOST, ITC, VTS, ETS, ICCD, DTIS, WIFS).

Ramesh Karri served(s) as the Associate Editor of IEEE Transactions on Information Forensics and Security (2010-2014), IEEE Transactions on CAD (2014-), ACM Journal of Emerging Computing Technologies (2007-), ACM Transactions on Design Automation of Electronic Systems (2014-), IEEE Access (2015-), IEEE Transactions on Emerging Technologies in Computing (2015-), IEEE Design and Test (2015-) and IEEE Embedded Systems Letters (2016-). He served as an IEEE Computer Society Distinguished Visitor (2013-2015). He served on the Executive Committee of the IEEE/ACM Design Automation Conference leading the Security@DAC initiative (2014-2017). He has given invited keynotes, talks, and tutorials on Hardware Security and Trust (ESRF, DAC, DATE, VTS, ITC, ICCD, NATW, LATW, CROSSING, HIPEAC).