Analyzing Security Vulnerabilities Induced by High-level Synthesis

NITIN PUNDIR, University of Florida, USA
SOHRAB AFTABJAHANI, Intel, USA
ROSARIO CAMMAROTA, Intel Labs, USA
MARK TEHRANIPOOR and FARIMAH FARAHMANDI, University of Florida, USA

High-level synthesis (HLS) is essential to map the high-level language (HLL) description (e.g., in C/C++) of hardware design to the corresponding Register Transfer Level (RTL) to produce hardware-independent design specifications with reduced design complexity for ASICs and FPGAs. Adopting HLS is crucial for industrial and government applications to lower development costs, verification efforts, and time-to-market. Current research practices focus on optimizing HLS for performance, power, and area constraints. However, the literature does not include an analysis of the security implications carried through HLS-generated RTL translations (e.g., from an untimed high-level sequential specification to a fully scheduled implementation). This article demonstrates the evidence of security vulnerabilities that emerge during the HLS translation of a high-level description of system-on-chip (SoC) intellectual properties to their corresponding RTL. The evidence provided in this manuscript highlights the need for (a) guidelines for high-level programmers to prevent these security issues at the design time and (b) automated HLS verification solutions that cover security in their optimization flow.

CCS Concepts: • **Security and privacy → Security in hardware;**

Additional Key Words and Phrases: High-level synthesis, security vulnerabilities, threat model

ACM Reference format:
Nitin Pundir, Sohrab Aftabjahani, Rosario Cammarota, Mark Tehranipoor, and Farimah Farahmandi. 2022. Analyzing Security Vulnerabilities Induced by High-level Synthesis. *J. Emerg. Technol. Comput. Syst.* 18, 3, Article 47 (January 2022), 22 pages.
https://doi.org/10.1145/3492345

1 INTRODUCTION

High-level synthesis (HLS) promises to be an effective solution to cope with the ever-growing demand for increasing design and verification productivity of semiconductor design flow. It offers the unparallel possibility of shortening **time-to-market (TTM)** by allowing the design and verification engineers to implement and validate complex custom logic (e.g., hardware accelerators)

This work was supported in part by Semiconductor Research Corporation (SRC) Grant #2019-TS-2910.
Authors' addresses: N. Pundir, M. Tehranipoor, and F. Farahmandi, Florida Institute for Cybersecurity Research (FICS), University of Florida, 601 Gale Lemerand Drive, Gainesville, Florida, 32603; emails: nitin.pundir@ufl.edu, {tehranipoor, farimah}@ece.ufl.edu; S. Aftabjahani, Intel, San Diego, CA; email: sohrab.aftabjahani@intel.com; R. Cammarota, Intel Labs, Hillsboro, OR; email: rosario.cammarota@intel.com.

1550-4832/2022/01-ART47 $15.00
https://doi.org/10.1145/3492345

in higher levels of abstractions (C/C++). For example, government agencies have limited design teams and have relied on third-party **intellectual properties (IPs)** for years to address the shortcomings of their design capacities with the increasing complexity of modern **system-on-chips (SoCs)**. Consequently, the entire SoC is susceptible to vulnerabilities that may arise from any third-party IPs, such as the threat of malicious functionality or information leakage [34, 35]. However, the trend is changing, with the government entities and industry increasingly adopting HLS to accelerate design development and reduce the risk of using third-party IPs [38]. For instance, in a recent white paper, Nvidia's video team reported their motivation to adopt HLS [7]. According to this report, their design complexity (in terms of the number of transistors per chip) is growing by 1.4× per generation while the number of developers on a project has remained relatively constant, causing a design capacity gap. This trend creates a design and verification crisis for traditional RTL flows, which will become even more severe and widespread for emerging SoCs. Nvidia's video team heavily relied on HLS and shifted their design and verification to the C level to address this. Per the report, they simplified the design description by five times and reduced the design and verification time by 40%.

Current research efforts mostly focus on optimizing and increasing the efficiency of HLS. Little efforts have been put to comprehensively investigate potential security vulnerabilities (e.g., information leakage, side-channel leakage) introduced during the HLS translation process. Vulnerabilities introduced during HLS translation come from two primary sources: (1) HLS optimizations that are not security-aware and (2) Non-secure coding practices and HLS constraints due to lack of experience and expertise in designing secure hardware [22, 29]. The impact of induced vulnerabilities will be severe as HLS-generated hardware is increasingly being used in security-critical applications involving cryptography, machine learning, hardware accelerators in the cloud/edge environment, and so on [3, 21, 25, 30]. Using vulnerable HLS-generated RTL blocks can have significant consequences on the overall security of the SoC. Therefore, it is imperative to investigate HLS translation and develop a database of potential security vulnerabilities to analyze future HLS-generated RTLs against these potential vulnerabilities.

In this article, we demonstrate that security-unaware HLS optimizations and different HLS steps (scheduling, binding, control logic extraction, etc.) could generate vulnerable RTL designs. Such security implications could unintentionally increase the attack surface for *information leakage*, *susceptibility to fault injection*, *access control violations*, or *side-channel leakage*. We draw attention to these security vulnerabilities by illustrating the proof-of-concept on cryptographic applications written in C, e.g., an AES encryption engine. We show the example of information leakage vulnerability due to unbalanced pipelines introduced by HLS between assets. Similarly, the adoption of certain coding practices could make the design more susceptible to glitch/fault injection and information leakage attacks. There exist programmer guidelines for producing better hardware (with optimized resource consumption and memory parallelism). It is done with the help of macros and specific data types. However, there is nothing that could assist the programmer in achieving secure hardware. The demonstration of vulnerabilities in this article indicates the need for developing security verification tools and secure HLS coding guidelines. A similar CERT-C coding standard exists for software systems to improve their safety, reliability, and security [32]. However, CERT-C guidelines do not apply to HLS translations, as it does not consider (a) sequential-to-parallel translation and (b) hardware threat model. Similarly, the HLS compiler and its optimization algorithms should be modified to consider security assets and properties during translation. The article aims to facilitate future research endeavors to develop automated HLS security verification tools and secure HLS coding guidelines as a series of "do's" and "don'ts" for the designers.

The rest of the article is organized as follows: Section 2 briefly explains HLS with its various steps and the impact of HLS vulnerabilities on different entities. Section 3 describes the systematic

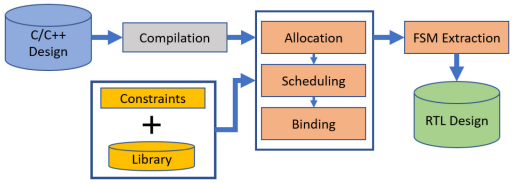


Fig. 1. Overview of generic design steps in high-level synthesis translation.

approach for identifying potential security vulnerabilities introduced by HLS. Section 4 describes HLS-introduced vulnerabilities test cases found in secure C applications. Section 5 discusses and compares with the relevant work in the field. Section 6 discusses the challenges and future work, followed by the conclusion.

2 BACKGROUND

2.1 High-level Synthesis

HLS is an effective solution to address the increasing design and verification complexity of semiconductor design flow. Thus, more than ever, designers rely on HLS to design very complex systems and meet aggressive TTM. HLS consists of a series of individual steps as shown in Figure 1, namely, compilation, allocation, scheduling, binding, and **finite state machine (FSM)** extraction [5, 8, 20].

- *Compilation:* The first step of HLS is to compile the functional specification implemented using a **high-level language (HLL)**. In this step, the provided high-level design specification gets translated into a formal representation illustrating the data and control dependencies of different operations. A **control data flow graph (CDFG)** of the representation helps identify data dependencies and exploit possible parallelism between various blocks. In this step, HLS can additionally apply various optimization techniques such as loop unrolling, loop pipelining, merging, and so on, to optimize the overall throughput, latency, and memory accesses of the design [4].
- *Allocation:* The second step is to allocate operations to different hardware resources such as computation blocks, **digital signal processing (DSP)** blocks, memory elements, connectivity components, and so on, to satisfy the overall design constraints. The allocation of hardware resources may also occur at other stages of the HLS. For example, a connectivity block such as bus or wire could get added before or after scheduling or binding steps as per the HLS tools requirements.
- *Scheduling:* In this step, HLS maps different design operations to specific clock cycles of the design to satisfy data dependencies with the least design constraints (latency or resources). Scheduling can be of a single cycle or multi-cycle based on the executed operation and its data dependency. If there are no dependencies between operations, then they may be scheduled in parallel to increase the design's performance.
- *Binding:* By this step, functional units and hardware resources are already determined to execute design operations. Binding aims at optimizing the total number of such functional units or hardware resources required to implement the design. One can categorize the binding into the module, register, and interconnection binding, depending on the optimized resources. Module binding enables operations scheduled at different clock cycles to reuse the same resources. Register binding maps the temporary values crossing the clock boundaries

to different registers [33]. Finally, at the interconnection binding, the connections between different resources are optimized.
- *Control Logic Extraction:* In the final step, HLS identifies the control signals and extracts the FSMs of the design. This step results in the automatic generation of the FSM that controls the design's datapath and glues the final RTL design together.

2.2 Threat Model

This article follows a white-box approach to identify potential security vulnerabilities and generate exploitable test cases. We assume a security engineer's role in any semiconductor company with complete access to the design files (C/C++, RTL, and gate-netlist) and commonly used verification tools.

As for the threat model, we assume that the attacker could be an end-user interacting with the HLS-generated hardware. The attacker may have physical access to the device or is interacting remotely with the device. We assume that the vulnerabilities introduced by HLS in RTL could be passed down the design cycle to post-silicon and could allow the attacker to leak security assets or interfere with the device's normal execution flow. We define *security assets* as any data variables provided dynamically or stored in a device that the designer wants to protect. Example assets are the encryption/decryption keys in the cryptographic systems or weights/biases of the trained machine learning model. The vulnerabilities introduced by HLS could be divided into four broad categories based on their effect/impact on the design's security.

- *Information Leakage:* This refers to the leakage of security-critical information to untrusted/observable points. For example, leakage of AES key or intermediate round keys at the primary ciphertext port.
- *Control Flow Violation:* This type of vulnerability allows an adversary to break the intended flow of the design execution. An attacker could exploit this to gain unauthorized access to keys, weights/bias of ML model, plaintext, and so on.
- *Fault Injection:* This type of vulnerability may allow an attacker to perform fault injection/glitching attacks to bypass existing security mechanisms of the design [1]. These vulnerabilities may exist in any RTL and not just HLS-generated RTL. However, HLS optimizations could generate RTLs that are more susceptible to these types of attacks.
- *Side-channel Leakage:* This type of vulnerability can result in leakage of side-channel information such as power and time, which an attacker can exploit to infer the design's security assets.

In this work, we show how HLS could generate IPs with a security vulnerability. When these IPs get integrated into an SoC, the SoC will be inherently vulnerable to various attacks. For example, we have shown that HLS-generated AES IP may have unbalanced pipelines that could leak keys. Similar vulnerabilities may happen for any HLS-generated IP with critical security assets.

2.3 Impact of HLS Vulnerabilities on Different Entities

Table 1 illustrates the advantages of HLS for various organizations and the repercussions of vulnerabilities caused by HLS if they are allowed to prevail in their products. The impact of HLS vulnerabilities can be summarized as follows:

- *National Security:* Government and defense entities usually have smaller design teams; hence, they use HLS to develop complex systems. HLS-generated vulnerable IP blocks can pose a significant threat to government agencies if used in systems of national security interest.

Table 1. Impact of HLS Vulnerabilities on Different Entities

Entity	Benefits	Consequence
Government & Defense	Small hardware teams	Dire consequences, since operations are security critical
FPGA-as-a-Service Platforms (AWS, Nimbix, etc.)	End-users not familiar with HDLs	Impact end-users & company reputation
CAD Developers	HLS products for rapidly growing field	Affects all entities using the product
Software Companies (Google TPU)	Small hardware teams, Reduce TTM	Create exploits in products used by millions
Hardware Companies	Smaller Team, Reduce TTM	Create exploits in millions of devices

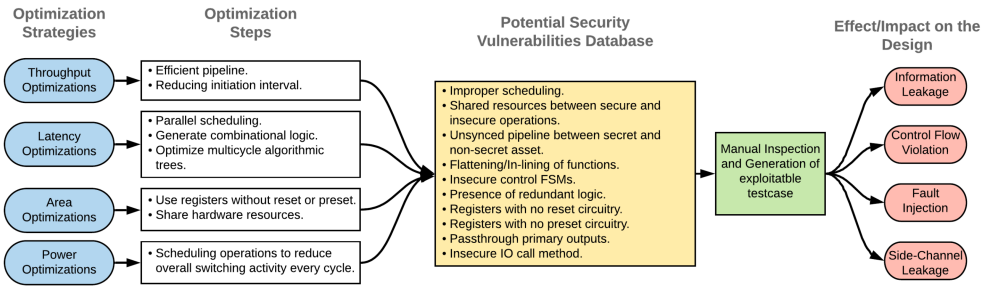


Fig. 2. Overall security assessment framework for identifying security vulnerabilities introduced by HLS translation.

- *Brand Erosion:* HLS gives industrial entities the advantage of simpler design descriptions that significantly decrease design and verification efforts. However, HLS vulnerabilities can compromise their products, harming the company's reputation.

2.4 HLS Optimizations

This section describes various optimization strategies that HLS could adopt to generate efficient RTL, as shown in Figure 2. We classify these strategies into throughput, latency, area, and power, as discussed below.

- *Throughput Optimizations:* HLS attempts to improve the throughput of the generated RTL by optimizing the loops and functions in the HLL design. HLS constructs pipelines for that by inserting registers between loop or function operations. HLS further enhances the performance of pipelines by improving the initiation interval. An initiation interval of one means that the pipeline can receive new data every clock cycle.
- *Latency Optimizations:* Latency, measured as the number of clock cycles required to complete one execution call, is an essential parameter for measuring RTL performance. By default, HLS aims to reduce the overall latency of the design and can adopt some of the following steps to achieve it:
  - Parallel Scheduling: Identifies independent operations and loops with no data dependency and schedules them in parallel to each other.
  - Partial/Full Loop Unrolling: Partially or fully unrolls the loop to execute loop iterations in parallel.
  - Optimize Algorithmic Trees: Identify multi-cycle algorithmic trees such as adder trees and optimize them into a single cycle.
  - Generate Combinational Circuitry: Whenever possible, HLS generates combinational circuity of the design/operations.
- *Area Optimizations:* HLS compiler attempts to reduce the generated RTL area by exploring optimization opportunities. For example, HLS may use registers without reset and

preset circuitry. We observed that the area difference is in the range of approximately 8% (for saed32hvt_ff0p85v25c library) and can significantly impact a design with a large number of registers. Even after using registers with reset and preset, HLS could decide not to connect the reset and preset lines if it is not functionally required. It saves interconnect resources and thus reduces overall area. Similarly, HLS can identify operations that are scheduled at different clock cycles but using similar hardware resources. As a result, HLS can share the resources among those operations rather than allocating new resources for each operation, thus reducing the generated hardware design's overall size.

- *Power Optimizations:* The design's dynamic power consumption directly depends on the design's switching activity, as seen in the following equation:

$$Power = \left[(C_{pd} \times f_I \times N_{SW}) + \sum (C_{LN} \times f_{On})\right] \times V_{CC}^2, \tag{1}$$

where C_{pd} is dynamic power dissipation capacitance, f_I is input frequency, f_{On} is all different output frequencies, V_{CC} is supply voltage, N_{SW} is number of bits switching, and C_{Ln} is capacitances seen at each output. Therefore, to reduce the design's dynamic power consumption, HLS could schedule the operations to minimize the design's overall switching activity in a particular cycle. HLS can also generate RTL components that store data on-chip instead of fetching from external DRAM, which is at least 30× less costly [15]. Similarly, HLS can also convert the floating-point representations to fixed-point/integer to conserve power in the generated RTL.

3 SECURITY ASSESSMENT

In this section, we discuss our step-by-step guide for identifying potential security vulnerabilities that the HLS could introduce in the generated RTL. Subsequently, we highlight our framework to generate tests to exploit the potential security vulnerabilities and how they can be addressed at various abstractions, i.e., HLL or RTL.

3.1 Assessment Framework

Figure 2 shows an overview of our approach for identifying potential security vulnerabilities introduced during HLS translation. HLS compilers target to produce efficient and optimized RTL designs from HLL specifications. For this, HLS compilers integrate various optimizations in the toolchain. However, these optimizations do not consider the security of the design. These optimizations are independent of using and propagation of critical assets (such as secret keys) in the design. Therefore, to recognize security issues that HLS might introduce, it is first necessary to identify such optimizations. We classified these optimization strategies and various steps associated with them, as shown in Figure 2. We then map these steps to potential security vulnerabilities that could be introduced in the translated security-critical application. These mappings are part of our growing database of potential vulnerabilities. The database serves as a standard to verify the generated RTL designs for potential security vulnerabilities based on the optimization strategies and steps used during the translation. In this article, we resort to manual inspection using formal tools and simulation test benches for proof-of-concept purposes. However, in future work, We envision developing security verification tools at different abstractions of translation to automatically analyze security vulnerabilities associated with HLS translation. First, the tools at C/C++ can help analyze the source code and the constraints to identify the potential vulnerabilities in the generated RTL. For example, vulnerabilities due to no reset/preset constraints, critical function in-lining, call methods, and so on. Second, the tools during translation can help analyze the intermediate design representation after each HLS step and verify the impact on the design's security.

Table 2. Potential Security Vulnerability Causes and Their Countermeasures

Security Vulnerability Potential Cause	Source	Potential Countermeasure
Flattening of functions	Insecure Coding	Use pragma/directive to prevent flattening of critical function.
Registers with no reset/preset	Insecure Coding	Enable global configurations to use reset registers. Crosschecking if the library has desired register types.
Passthrough Primary Outputs	Insecure Coding	Enable pipeline dataflow architecture. Making output latching dependent on condition in C/C++.
Insecure Call Methods	Insecure Coding	Identifying and understanding the accurate threat model of different components.
Non-secure Scheduling	Insecure Translation	Development of security-aware scheduling heuristics.
Non-secure Resource Sharing	Insecure Translation	Development of security-aware binding and resource sharing heuristics.
Unbalanced Pipeline	Insecure Translation	Tool enhancement required. Temporary fix is to implement pipelined design (#pragma HLS dataflow), which prevents intermediate leaky register to directly connect with the primary output port.
Insecure Control Logic Extraction	Insecure Coding/ Translation	Tool enhancement to implement fault-resistant FSMs.

For example, security assessment after schedule, resource allocation, binding, and so on. Finally, the tools at RTL can help to analyze the susceptibility to information leakage, fault-injection, side-channel leakage, and so on. Though, at RTL, various verification tools (such as SecVerilog [39], QIF-Verilog [11], etc.) exist to analyze for security verification and can be used to assist in security verification of HLS-generated RTLs. However, HLS-generated RTLs present unique challenges that will require improvements in the existing tools or developing new tools. For example, the HLS-generated RTLs are very complex and large (in terms of code size) compared to the human-written RTLs for the same functionality, making it difficult to comprehend and apply countermeasures directly at the RTL. It would require the mapping of vulnerabilities to higher abstraction to apply the countermeasures.

3.2 Potential HLS-induced Security Vulnerabilities

In Section 2.4, we discussed a series of optimizations undertaken by the HLS compiler to effectively translate the HLL design specifications into a functionally equal RTL design. In this section, we discuss the security vulnerabilities introduced by HLS. We classify these potential security scenarios based on the source of vulnerability, as summarized in Table 2.

3.2.1 Designer-introduced Vulnerabilities. This section gives an overview of potential security issues that may arise in the HLS-generated design due to default configurations of the tool or the user's lack of knowledge towards generating secure hardware design. The user could mitigate these security issues by providing certain HLS-specific directives/pragmas or secure coding practices to develop the algorithmic flows in C/C++.

(1) *Flattening of Functions:* One of the latency optimizations performed by HLS is the in-lining of the function calls during hardware translation. Flattening different functions can present different types of security challenges. If a function is security-critical, then it is essential to verify all registers associated with information leakage. However, if HLS flattens each function call, then it significantly increases the number of security-critical registers in the design. This increase in security-critical registers could increase the verification efforts of ensuring none of these registers result in information leakage.
(2) *Registers with No Reset/Preset:* HLS uses registers in the generated design to create efficient pipelines, synchronize design to the system clock, or use them as memory units. These registers may hold sensitive information, such as keys, passwords, and so on. Therefore, it is

crucial to clear these registers when the design is reset or at the beginning of the new execution cycle. However, the number of registers or memory elements in the generated design may exponentially increase in an algorithmic complex design. As a result, the HLS compiler may resort to registers with no reset or preset to reduce the area overhead. HLS may also choose to disconnect the reset and preset lines to the registers if it is not required functionally to reduce the interconnect resources. HLS may choose to adopt that strategy for selected registers or all registers, depending on its underlying selection algorithm. Registers with no reset may cause them to retain sensitive information increases the risk of leakage. Whereas registers with no preset can cause the register to be in an insecure state for a short period right after the reset. It is best illustrated by example in CWE-1271,[1] as shown in Listing 1.

Listing 1. CWE-1271: No-preset register vulnerability scenario.

```
always @(posedge clk) begin
    if (en) lock_jtag <= d;
end
```

The code shows that the flip-flop is in an unknown state until "enable" and "D" signals update the flip-flop state. An attacker can reset the device until the JTAG interface is unlocked, allowing him/her access to the test interface until the logic drives the "lock" signal to a known state.

(3) *Passthrough Primary Outputs:* The HLS-generated FSM controls the data flow and regulates when the results get latched to the primary outputs. However, these FSMs may not be secure to optimize the design's latency and area. For example, HLS may directly latch the intermediate computation to the primary output.

This optimization improves the design's area by not implementing the additional circuitry needed to control, so the output value is only latched at the end-of-execution. It also improves the design's latency by one cycle, as now the extra step of latching the output does not need to be performed.

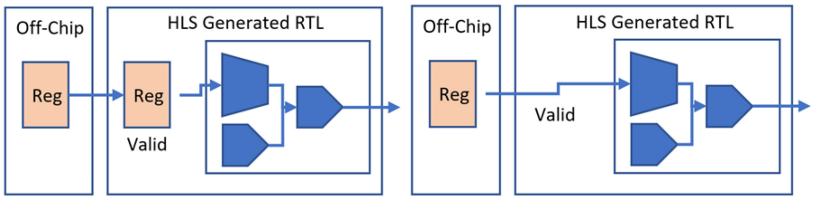
Listing 2. Example C code to show direct latching of intermediate values to primary output.

```
int add(int *arr){
    int sum = 0;
    for( int i=0; i<len; i++){
        sum = sum + a[i];
    }
    return sum;
}
```

Listing 2 shows a simple HLL code, where the "add" function returns the final computed sum at the end of the function execution. For the "add" function, HLS generates input and output ports for "arr" and "sum" variables, respectively. Since "sum" updates at every iteration, the intermediate "sum" values get reflected at the output port due to a lack of safe data flow control. This behavior is unsafe for sensitive applications, such as AES, because ciphertext should only update after the final AES round. If the output port latches intermediate round values, then this could enable an attacker to guess the key.

(4) *Lack of Top-level Control Signals:* HLS compilers automatically introduce block-level control signals and the FSMs into the generated RTL module. Block-level signals are vital to assist data handshaking and integration of generated IPs in the SoC. These block-level signals are inextricably linked to the HLS-generated FSMs in the design and impact the design behavior. However, these signals could be modified or disabled by the user using macros

[1]CWE is a community-developed list of software and hardware weakness types. https://cwe.mitre.org/.



(a) Valid passed by value. (b) Valid variable passed by reference.

Fig. 3. The impact of top-level arguments defined in the HLL specification on the generated RTL.

(pragmas/directives) and could have a security-critical impact on the design. For example, in Section 4, we showed how the absence of start-signal could cause data-latching problems in the design, leading to information leakage.

(5) *Insecure Call Methods:* HLL design specification allows function arguments to be provided either by value or by reference. When "passed-by-value," the function creates a copy of the argument to be used within the function. In contrast, when "passed-by-reference," a reference to the memory element storing argument value is passed to the function. However, inappropriate call methods could become a source of vulnerability in hardware, depending on the threat model, as discussed below.
   - Pass-by-value: HLS inserts a register at the input buffer when an argument is "passed-by-value" to create a copy of the argument value from the external location of the memory (external to the HLS-generated RTL) as shown in Figure 3(a). The copied value is used by design throughout the entire execution without being refreshed by the external source. If an attacker can inject a transient fault when the design makes a copy, then the entire execution will use the incorrect argument value. It also simplifies the attacker's effort to time the fault injection, as he now has to inject the fault at the start of the design execution. Therefore, if the system's threat model assumes that an attacker can snoop the input to the HLS-generated RTL and inject fault, then the variable should be passed-as-reference. It will complicate the attacker's work to time the attack, and the design will always use the refreshed value from the external source.
   - Pass-by-reference: HLS does not generate a copy of the arguments when "passed-by-reference" but will retrieve from the external memory whenever necessary during the execution, as shown in Figure 3(b). The design would expect the external referenced memory (ROM, RAM, or register in other modules) to remain stable throughout the execution. Failure to do so will cause the design to fetch incorrect values. Therefore, if the system's threat model assumes that an attacker could compromise the external memory location during design execution, then the variable should be pass-by-value. It will allow the design to make a correct local copy of the value at the start of execution for use throughout the execution.

   As discussed above, the variable call method depends on the system's threat model. Therefore, the designer should be cautious with the system's requirement to prevent HLS from generating RTL, which is vulnerable to the discussed attacks.

3.2.2 Tool-introduced Vulnerabilities: These security issues could arise in the HLS-generated design, because HLS was never designed with security in mind and is unaware of security-critical information/operations in the design. Mitigation of these issues requires changes in the HLS engine and optimization algorithms.

Listing 3. Snippet of AES OpenSSL code.

```
1  void rijndaelEncrypt(const u32 rk[/*4*(Nr + 1)*/], int
       Nr, const u8 pt[16], u8 ct[16]) {
2        u32 s0, s1, s2, s3, t0, t1, t2, t3; int r;
3        s0 = GETU32(pt     ) ^ rk[0]; s1 = GETU32(pt +
                 4) ^ rk[1];
4        s2 = GETU32(pt +  8) ^ rk[2]; s3 = GETU32(pt +
                12) ^ rk[3];
5    /* round 1: */
6        ...
7    /* round 2 - 8: */
8        ...
9    /* round 9: */
10       ...
11   if (Nr > 10) {
12       /* round 10 - 11: */
13       ...
14       if (Nr > 12) {
15       /* round 12 - 13: */
16           ...}
17   }
18   rk += Nr << 2;
19   /* apply last round and map cipher state to byte
          array block: */
20       s0 = ... ^ rk[0]; PUTU32(ct     , s0);
21       s1 = ... ^ rk[1]; PUTU32(ct +  4, s1);
22       s2 = ... ^ rk[2]; PUTU32(ct +  8, s2);
23       s3 =... ^ rk[3];  PUTU32(ct + 12, s3);
24 }
```

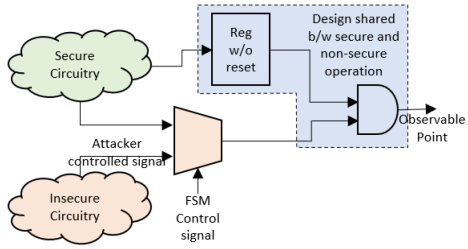


Fig. 4. Illustration of resource-sharing vulnerability in HLS-generated RTLs.

(1) *Non-secure Scheduling:* As previously described, HLS tries to optimize the overall latency of the generated design by scheduling multiple operations in parallel. This optimization may overlook that certain operations are sensitive and should be scheduled after fulfilling some prior conditions. This simple optimization may overlook the security rule "If A happens, then only B should happen," leaving security holes in some critical applications. For example, in security protocol "HMAC then decrypt" [12], decryption should only occur after the ciphertext authentication. Assuming the HMAC takes eight cycles and the decryption takes 11 cycles, the overall design latency will be 19 cycles. However, to optimize the design's latency, HLS can schedule certain independent operations of the decryption ahead of time. At the end of the execution, the assumption is that the design will discard those pre-computed results based on whether or not the HMAC was successful. However, since partial or complete results are already in the internal registers, they are susceptible to leakage, depending on the attacker's capability. This optimization nearly defeats the algorithm's primary purpose, i.e., not initiating decryption if HMAC fails. A similar observation is made in HLS-generated AES implementation [9], where one can assume that the last round key will be fetched only after compilation of previous rounds. However, as shown in Listing 3, Line 18 has no prior data dependency, and HLS schedules this operation at the very first clock cycle to fetch and store the last round key in the internal registers. HLS performs this optimization to reduce the number of clock cycles required for the last AES round.

(2) *Non-secure Resource Sharing:* The binding process that allocates real hardware resources to operations does not recognize the design's security-critical and non-critical assets/operations. HLS may share the available hardware resources such as adders, multipliers, DSPs, and so on, between the different design operations to optimize the area overhead. This sharing may cause secure and non-secure operations to use the same hardware resources. Similarly, to reduce the overall register consumption of the design, HLS may also share registers between different operations during the register binding stage [13]. This resource sharing between the secure and insecure part of the design may allow an adversary to compromise the design's security concerning **confidentiality, integrity, and availability (CIA)**.

For example, suppose an attacker can control a non-secure operation that uses the same hardware resources used by secure operation in earlier cycles. In that case, there is a possibility of residual leakage from the hardware resources. For example, Figure 4 illustrates the HLS-generated RTL where certain resources (blue-region) are shared between secure and non-secure operations. Note that the secure and non-secure operations distinction is only for a developer as HLS is unaware of its presence. The design's FSM controls if the shared resources process the secure or non-secure signal. If an attacker can control to force the "attacker-controlled signal" to "1" using fault-injection techniques [24, 42], then one can leak the register value at the observable point.

(3) *Unbalanced Pipelines:* As mentioned earlier in this section, HLS automatically identifies pipelining opportunities and inserts additional registers to gain performance. However, HLS does not understand the concept of secure and non-secure assets. As a result, multiple pipeline paths can be created for different variables (secure and non-secure). However, no attempt is made to synchronize those paths to preserve the security of critical assets in the design. As a result, one variable may arrive early at the processing unit (DSP, adders, MULT, etc.) while the other is still in the pipeline. If an attacker can manage to flush the pipeline before other variables catch up, then he/she may cause information leakage; e.g., leakage of intermediate values (which may leak information about the assets) in the pipeline. The evidence of this vulnerability is shown in Section 4.

(4) *Control Logic Extraction:* As mentioned, an HLS-generated design is no longer sequential compared to the C/C++ code. To govern the complex data paths in the RTL, HLS generates control FSM. The user could implement binary, gray code, or one-hot encoded FSM in the generated RTL. However, such automatically created control FSMs mainly exploit parallelism or optimize area and may not be immune to fault injection attacks. For example, HLS-generated RTLs could have "don't care" states in the design to synchronize data paths. These optimizations could make the generated-RTL FSM vulnerable to single event upsets and fault injection attacks [23, 24, 36]. It could cause FSM to bypass some states that could cause privilege escalation, information leakage, or control flow violation.

(5) *Side-channel Leakage:* HLS performs power optimizations that are aimed to reduce the overall power consumption of the generated RTL. However, to the best of our knowledge, no HLS compiler considers power side-channel leakage while performing HLS optimizations. As a result, by default, HLS-generated RTLs are susceptible to power side-channel leakage attacks. Moreover, HLS optimizations for performance and latency could make the side-channel countermeasure implementations in C/C++ vulnerable after the translation [41].

4 EXPERIMENTAL RESULTS

In this section, we present case studies to demonstrate that potential security vulnerabilities discussed in Section 3.2 could exist when HLS is used to generate RTL for secure C applications without proper checks. We present cases of *unbalanced pipelines, generation of a combinational circuit, lack of reset, lack of control signal, and side-channel leakage.* These vulnerabilities cause the leakage of an asset (in our case, a cryptographic key) in the HLS-generated RTL. For our case studies, we chose the OpenSSL implementation of AES-128 because it is an open-source, widely used, and most updated cryptographic repository [9]. The implementation was also free from any known security vulnerability when writing the manuscript. In addition, the AES-128 C code used is a T-table-based implementation, extensively adopted by various cryptographic libraries.

Experimental Setup: For our experiments, we used Xilinx's Vivado HLS [37] and Mentor's Catapult HLS [10]. Both compilers are widely used and commercially available. Although the RTL

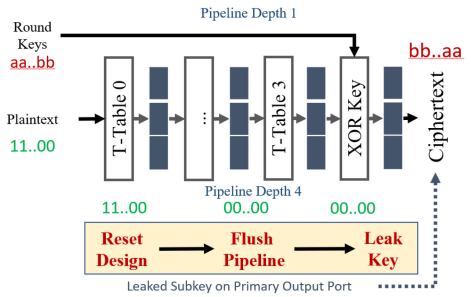


Fig. 5. Illustration of unbalanced pipeline in HLS-generated RTL of OpenSSL AES implementation.

output of HLS compilers is hardware-independent, the Vivado HLS contains binding information of Xilinx FPGAs and Catapult of specific ASIC technology nodes, making them suitable for targeting FPGAs and ASICs. The standard translation steps and different optimization strategies are also consistent across various HLS compilers. As a result, the security implications discussed in this article apply to other HLS compilers as well. The discussed vulnerabilities arose in the generated RTLs when compilers were used in their default configurations (out-of-the-box), and the user provides no additional constraints. Since compilers have hundreds of constraints to apply locally to functions or globally, we highlight a few prominent ones enabled/disabled in the default configuration. For example, Vivado HLS includes function inlining disabled, pipeline optimizations disabled, loop unrolling/optimizations disabled, array optimization disabled, and control signals enabled. However, Catapult includes function inlining enabled, control signal disabled, transactional signals enabled, speculative execution enabled, pipelines, and loop optimization disabled.

4.1 Unbalanced Pipeline Depths

Through the HLS optimization assessment phase as discussed in Section 3, we realized that HLS makes no efforts to balance pipelines between secure and non-secure assets of the design. In AES round operations, plaintext goes through different operations (subkey, shift rows, and mix column) before being XORed with a round key. Therefore, there should be a mismatch between the plaintext and the round key depths for each AES round.

We wrote formal properties to check if the round key could be leaked after the first round. Since Vivado HLS generates passthrough output ports (as explained in Section 3.2), using formal verification, we were able to leak round key via *reset* and *flush* attack through the ciphertext ports. In this attack, flushing the pipeline after resetting registers at a specific time leaks the intermediate key values. The vulnerability arises from unbalanced pipeline depths between round keys and other round operations, as shown in Figure 5. Here, the round keys reach the ciphertext output before the intermediate round states can catch up, allowing the adversary to leak the values by resetting and flushing. Figure 6 shows the functional validation of the leak in the AES encryption module where part of the first round key is directly leaked at the ciphertext output port. The abbreviations *rk*, *pt*, and *ct* in the figure represent the round keys, plaintext, and ciphertext, respectively. Whereas, *ap_clk*, *ap_rst*, and *ap_start* are the top-level control signals introduced by the HLS compiler. As one can see, the leaked keys are in the reverse order due to the array's reverse indexing from C to RTL. We also evaluated the decryption block of the OpenSSL AES by performing the same series of analyses. As shown in Figure 7, we were able to perform the same attack and leak first-round keys to the plaintext output port.

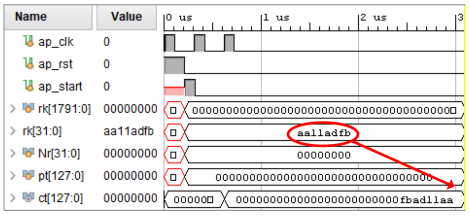


Fig. 6. Leakage of round 0 key at the ciphertext output using reset and flush attack in the Vivado-generated OpenSSL AES encryption block.

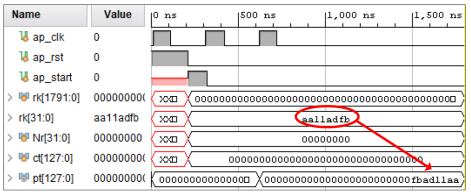


Fig. 7. Leakage of round 0 key at the ciphertext output using reset and flush attack in the Vivado-generated OpenSSL AES decryption block.

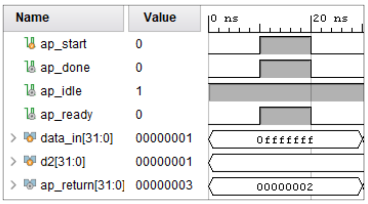


Fig. 8. Vulnerability introduced by HLS while parsing conditional statement. The computed result is reflected on the primary output even before "done" signal is asserted.

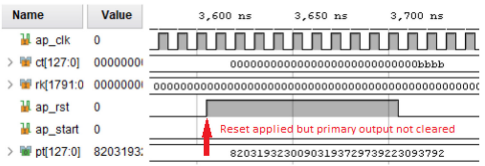


Fig. 9. Not clearing of intermediate registers and I/O registers on applying reset. Previously decrypted value available on output port even after applying reset in Vivado-generated RTL.

4.2 Generation of Combinational Circuits

We examined the translation of conditional statements by the HLS compiler. In its default configuration mode, we found that the Vivado HLS compiler translates conditional statements into combinational circuits to optimize performance and reduce latency. In addition to the generated RTL, Vivado HLS also automatically adds some block-level control signals to the design, such as ap_done, ap_idle, ap_start, and so on, to manage module operations and reflect valid data availability at the I/O ports.

Our findings on these generated RTLs showed that the computed values are reflected on the primary output even before the valid output signal is confirmed by design, as shown in Figure 8. The primary cause of the problem was the combinational circuit generation to reduce each conditional branch's latency. It could become a source of security problems by causing information leakage if such HLS-generated RTL modules were integrated into the SoC revealing additional information at the output port when it is not supposed to happen.

4.3 Uncleared Intermediate and I/O Registers

All intermediate and I/O registers should be cleared to prevent leakage of sensitive residual information for security reasons. However, both Vivado HLS and Catapult HLS do not clear the intermediate register values when used in their default configuration to reduce the created RTL area overhead. We understand that intermediate register values leakage is not intuitive and highly depends on the complete SoC implementation. However, even after applying reset, the mere presence of sensitive information in registers raises concerns about the SoC's overall security.

The situation gets severe when we noticed that Vivado HLS does not even clear the I/O port registers on applying reset. Figure 9 shows the AES decryption block's functional simulation where the plaintext output port retains the previously decrypted value even after the reset. These results

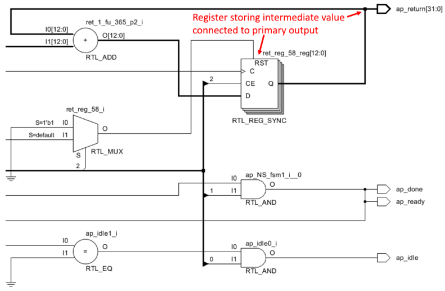


Fig. 10. RTL schematic of secure string comparison function showing connection of output port.

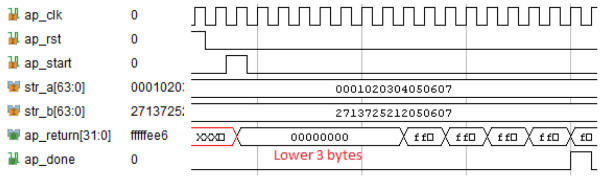


Fig. 11. RTL behavioral simulation of secure string comparison function.

emphasize that if HLS is used for security-critical applications, more preventive checks should be in place, because the default behavior of HLS compilers is to optimize the design for performance and area.

We tested the HLS-generated RTLs for "Passthrough Primary Outputs," as discussed in Section 3.2. Using Vivado HLS, we generated the RTL of a secure string comparison function (shown in Listing 4) and analyzed the output port's connection to the rest of the design. This C function can be considered secure, because it takes constant time for the comparisons, thus preventing timing attacks, which could reveal at which bytes the strings differ based on when the function finishes execution. As shown in Figure 10, the primary output port "ap_return" was directly connected to the register, which stores the intermediate byte comparison results. Since there is a lack of secure data control mechanism that could govern latching of result to the output port, the intermediate comparison results are reflected on the output port, as seen in Figure 11. Even though the entire execution takes constant time, the attacker could precisely know which byte of the strings differ.

Listing 4. Secure string comparison function.

```
#define ABS(N) ((N<0)?(-N):(N))
int strncmp_c(char str_a[8], char str_b[8]){
        int ret=0;
        strncmp_c_label0:for(int i=0; i<8; i++){
                ret += ABS(str_a[i] - str_b[i]);
        }
        return ret;
}
```

4.4 Latching of Intermediate Values

The vulnerability also persisted in the RTL generated for the AES-128 module, emphasizing the severity. The AES-128 takes round keys ("rk") and the plaintext ("pt") as inputs. At encryption's beginning, the plaintext is XORed with the first round key, and the result is internally stored in variables "s0, s1, s2, and s3." The encryption loop iterates five times and computes two AES rounds each time. As seen in Figure 12, which shows the behavioral simulation of generated RTL, the

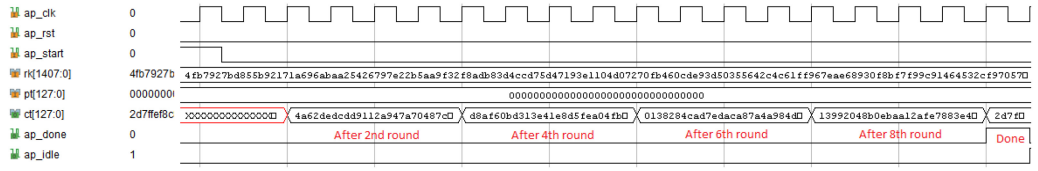


Fig. 12. AES behavioral simulation.

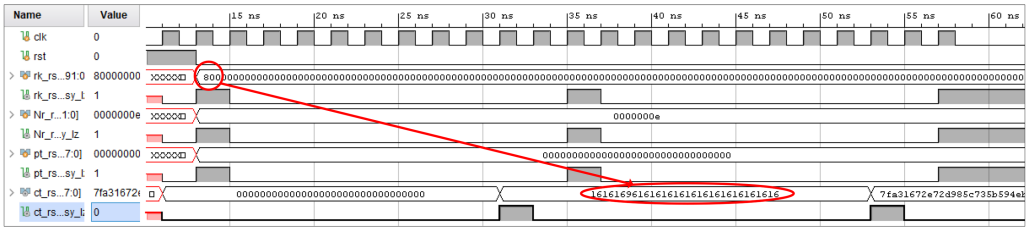


Fig. 13. Improper latching of input values due to lack of "start" control signal causes bypassing of intermediate round operations and leaking of intermediate information.

ciphertext state after every two AES rounds latches to the output port. It presents a significant risk because if the attacker can retrieve the ciphertext state after the second round, it significantly decreases the attacker's effort to obtaining the original key or the plaintext.

4.5 Lack of Top-level Control Signals

We found that Catapult HLS automatically generates transactional signals in the default configuration, such as "start" and "done." Transactional signals indicate when the module finishes reading each specific input or produces a specific output. The lack of a "start" signal makes the generated RTL dependent on the reset signal to latch the input data. The module automatically latches the input data when reset is disabled and goes into a free-running mode (always running) until reset is again enabled. It can be exploited to cause data latching problems and leaking information at the primary output. For example, the functional simulation of the catapult generated RTL of the OpenSSL AES encryption module showed that the latching of input signals at a specific time causes the module to bypass some of the intermediate round operations to leak a unique intermediate value. On further analysis, we found that the leaked value was 128'h161616 ...16 when the last round key was all zeros, and a single bit change in the final round key represented a single bit change in the leaked information, as shown in Figure 13. The arrow in the figure shows the single bit change in the final round key (4'h0->4'h8) and the corresponding change in the leaked intermediate value (4'h1->4'h9).

The gathered information helped us infer that the last round key was directly XORed with some intermediate value to leak a patterned value due to bypassing intermediate round operations. Then, we examined how different plaintext input values affect the leaked value. We found that plaintext had no impact on the leaked value. It was sufficient to orchestrate an attack where an adversary could leak the last round key by utilizing the gathered information, as shown in Figure 14. The unknown round key leaks some random key information, which then XORed with 128'h1616161...16 can reveal the last round key. The leaked last round key could help the attacker by reducing the complexity of guessing the original key.

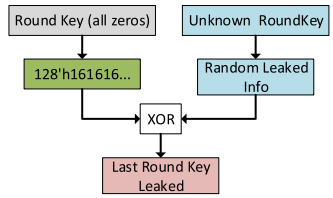


Fig. 14. Illustration of the usage of leaked intermediate information to retrieve the last roundkey.

Listing 5. Snippet of AES code used for insecure call methods test case.

```
void rijndaelEncryptRound(const u32 rk[60], int
     &Nr, u8 block[16], u8 ct[16], int rounds) {
    s0 = GETU32(block ) ^ rk[0];
    s1 = GETU32(block + 4) ^ rk[1];
    ...
    if(rounds==10 || rounds==12 | rounds==14)
        r = rounds-1;
    else
        r = rounds;
    ...
    /* Perform Arbitrary AES rounds */
    done =1;
    if (done && rounds == Nr){
    /* Perform Final Round*/
    }
}
```

Table 3. Unbalanced Pipeline Analysis for Vulnerable AES RTL

SD	KeyBit Early	Plaintext Early	Both at Same Depth
1			
2	Ct[32–127]		
3	Ct[0–31]		
4	Ct[0–31, 40–127]		Ct[32–39]
5	Ct[8–23, 88–95, 112–119]		Ct[0–7, 56–87, 104–111]
6	Ct[8–23, 64–79, 88–127]	Ct[56–63]	Ct[0–7, 24–55, 80–87, 104–111]
7	Ct[24–31, 56–63]		Ct[0–23, 64–127]
8		Ct[56–63]	Ct[0–55, 64–127]
9	Ct[56–63]		Ct[8–39, 64–127]
10		Ct[56–63]	Ct[0–55, 64–127]
11	Ct[56–63]		Ct[8–31, 64–79, 88–127]
12		Ct[56–63]	Ct[0–55, 64–127]

4.6 Insecure Call Method

As discussed in Section 3.2, insecure call methods can become a source of vulnerabilities, depending on the threat model. To show the proof of concept for this source of vulnerability, we chose the AES test case shown in Listing 5. We assume that the HLS-generated AES core will be a secure core, and the rest of the IPs in the SoC are insecure for the threat model. Therefore, it is safe to assume that the external variables (plaintext, keys, rounds, etc.) supplied to the AES core should be held stable throughout the execution to prevent security implications. The test case AES design allows performing arbitrary AES rounds to behave as a source of entropy or use it as typical AES-128/192/256. The "Nr" variable determines if the final round is part of the AES-128/192/256 and, therefore, should be performed without mix column operation. If "Nr" is passed-as-value, then it would be fetched at the start of the design execution and stored internally to provide a stable value to compare throughout the execution. However, if passed as a reference, then it is fetched from an external location whenever needed. Suppose a single-bit fault is inserted in the external memory/register storing the "Nr" variable during the last round comparison. It can force to bypass the final round and cause the latching of the ninth round state to the ciphertext port. An attacker can manually perform the sbox and shift row operations on the leaked value and XOR it with the correct ciphertext to leak the final round key. Figure 15 shows the simulation for the generated RTL, where a single bit fault at the "Nr" variable causes the latching of the ninth round state to the output port.

4.7 Countermeasure Implementation

The main purpose of this manuscript is to demonstrate that security vulnerabilities can be introduced either due to HLS optimizations for performance gains or incorrect user constraints due to the lack of security guidelines to assist secure translation. Showing the traces of various vulnerabilities is the first step to identify the potential security pitfalls of current HLS tools and move

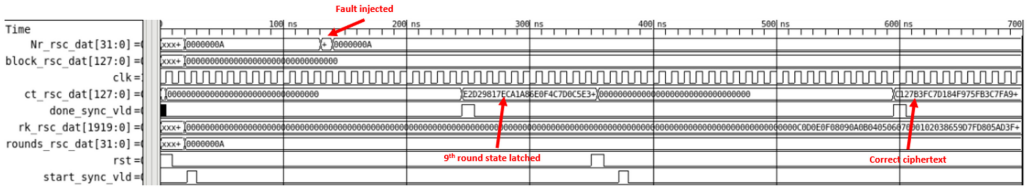


Fig. 15. Insecure call method simulation results.

Table 4. Unbalanced Pipeline Analysis for AES RTL after Countermeasure Constraints

SD	Keybit Early	Plaintext Early	Both at Same Time
1			
...			
15			Ct[0-127]

toward changing/modifying the C/C++ source code or the HLS engine to generate RTLs that are inherently secure. For this purpose, we have used a set of test designs and went through the trial and error for several threat models (information leakage, timing, and power side-channel, access controls, etc.) to manifest the security pitfalls of current HLS. In future works, we will present automated holistic approaches to identify these vulnerabilities and apply countermeasures to generate secure RTLs. However, after exhausting manual testing and trying different combinations of constraints, we suggest using some constraints and modifications in C code to avoid demonstrated security vulnerabilities in the generated AES design. The constraints highlighted in Listing 6 offer the most security during AES code translation while requiring no modifications to the original code.

Listing 6. HLS constraints for generating countermeasure AES RTL.

```
1) #pragma config_rtl -reset_all
2) #pragma HLS pipeline (flush and rewind disabled)
3) #pragma config_rtl -encoding gray
```

The first constraint ensures that all registers are cleared on applying reset to avoid information leakage or unauthorized access to the design assets. The second constraint, when used with "enable_flush" and "rewind" functionality disabled, protects against "passthrough vulnerability" and "unbalanced pipeline" for this particular test case. It transforms AES rounds into different pipeline stages where one stage register is only connected to the next stage register. It prevents the direct connection of AES state registers to the ciphertext output port to prevent latching of intermediate state register values to the ciphertext port. Furthermore, in the generated RTL, the pipeline depths for key bits and plaintexts to reach the ciphertext port are also balanced. It minimizes the risk of glitches trumping the leakage resistance countermeasures in the design. Table 3 shows the pipeline depth analysis for the vulnerable AES for a $key - bit[0]$ and $plaintext[0]$. It can be seen that the key bit can reach ciphertext ports early at sequential depths 2 and 3, which is exploited to leak the first-round keys. Table 4 shows that pipeline depths for the key bit and plaintext are balanced in the countermeasure AES RTL. The third constraint directs HLS to use gray encoding to implement the state FSM. Since gray encoding ensures only 1-bit change during state transition, it makes the design's FSM resilient towards setup-and-hold time violation attacks [24]. The use of constraints to generate considerably secure RTL results in an area overhead of 20% and an increase in the latency by two clock cycles.

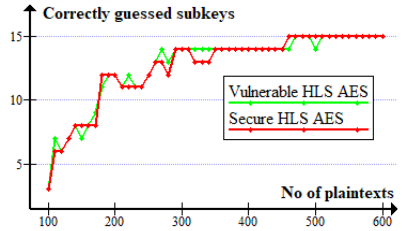


Fig. 16. SCA comparison between AES RTLs with and without vulnerabilities.

4.8 Side-channel Leakage

We performed preliminary **side-channel analysis (SCA)** on the HLS-generated RTLs to highlight the need for **power side-channel (PSC)**-aware translation. We compared the generated AES RTL with and without vulnerabilities to perform the SCA. Since the scope of the paper is limited to the generated RTLs, we performed pre-silicon **correlation power analysis (CPA)** [2, 31]. We targeted the design's first clock cycle, as this is where the first T-table substitution occurs, and the design is noise-free from subsequent round operations. To obtain the power traces in the pre-silicon (RTL) environment, we hypothesized the design's dynamic power consumption based on the nets' switching activity [14]. We first used the RTL-simulation tool (VCS) to generate files in the **Switching Activity Interchange Format (SAIF)** that reports the toggle count of all signals within a specific period. We then parsed those files to measure the entire system's cumulative toggle count related to hardware design's dynamic power consumption. Figure 16 shows the relationship between the number of plaintexts and correctly guessed subkeys for both the AES RTLs. It is clear that both the designs are equally leaking PSC information.

Performing PSC analysis at the pre-silicon environment shows that HLS makes no efforts or provides constraints to assist in PSC-aware translation. However, HLS can be guided to drive the generation of masking circuits by developing the path-based masking structures by analyzing the algorithmic path to provide first-order resiliency to overall generated AES up to 1B traces [18]. In addition, HLS can help reduce the **signal-to-noise ratio (SNR)** by inducing additional noise during the execution of critical operations and perform PSC-aware scheduling to implement timing-rearrangement countermeasure to obfuscate the sequence of critical operations. We will present a set of tools and techniques for assessing the security of generated RTL codes and changing HLS flows for each type of threat (including side-channel leakage) in our future work.

5 RELEVANT WORK

To the best of our knowledge, only a few prior works have investigated security-related aspects of HLS. For example, Zhang et al. [40] have analyzed the effects of HLS optimizations on power side-channel leakage. The authors applied different HLS optimizations on AES S-Box architecture and evaluated their power side-channel leakage characteristics. However, they did not analyze the effect of different underlying HLS processes for security but only focused on the resources used for implementing the design against power leakage. Jiang et al. [16] proposed an HLS framework named "Assure," which aims to develop a language-based **information flow control (IFC)** to enforce security policies during HLS translation. "Assure" flow changes the design process at the C level, is not compatible with existing commercial HLS tools, and only focuses on information leakage. Konigsmark et al. [17] analyzed the power leakage of HLS-generated RTLs and proposed to minimize the leakage by correcting relevant imbalanced branches in the C code. Peter et al. [26] proposed an LLVM-based framework to balance the execution paths in the C/C++ code to

ensure timing-attack-resistant HLS translation. Pilato et al. [28] proposed "TaintHLS" to automatically generate **dynamic information flow tracking (DIFT)**-enabled hardware accelerators for heterogeneous architectures. HLS has also been utilized for Trojan detection [6] and logic obfuscation [27]. There also has been quite a bit of work done to analyze/prevent information leakage at the RTL. Li et al. [19] proposed Caisson HDL for describing designs with secure information flow. However, it does not address the challenges faced in describing complex algorithmic designs for which HLS is predominantly used. Zhang et al. [39] proposed "SecVerilog," a formal verification-based framework to enforce secure information flow. It helps to mathematically ensure that an asset in RTL at a higher security level does not flow to a lower security level. However, formal methods face scalability issues; therefore, Guo et al. [11] proposed QIF-Verilog, which uses a quantitative information flow model to verify the system against given security rules. However, none of these works have investigated (i) how user practices could result in vulnerable RTL, (ii) how the HLS optimizations themselves could unintentionally introduce security vulnerabilities, and (iii) how to detect these vulnerabilities.

6 DISCUSSION

Our results and test cases have clearly shown that HLS is not security-aware, and HLS translation could introduce many security vulnerabilities. Therefore, there is a need to develop tools to verify generated-RTLs for security vulnerabilities. Moreover, it is critical to ensure that HLS is made security-aware to ensure secure translation in the first place. Below, we briefly discuss some of these future efforts.

- *Secure Coding Guidelines:* It is clear that coding practices in C/C++ and the use of pragmas/directives highly control the way HLS will generate the RTL. The knowledge obtained from identifying vulnerabilities should be transferred as a series of "Do's" and "Don'ts" to the developers to minimize the risk of generating vulnerable-RTL using HLS.
- *RTL Verification Tools:* The development of a potential security vulnerability database will help generate security properties for the RTL verification. The security properties could be translated as formal properties to check if the generated-RTL satisfies the security property. Moreover, tools could be developed to check for the dedicated security properties to ensure that the generated-RTLs are free from any known vulnerability.
- *Security-aware HLS:* It is clear that current HLS compilers and their algorithms are optimized for throughput, latency, area, and power. They are not security-aware, nor consider the presence of secret assets in the design while performing optimizations. The HLS algorithms for scheduling, binding, resource sharing, and so on, should be updated to consider security assets and security-critical information during translation. Secure heuristics for these algorithms should be developed that consider not only latency and resource constraints but also security.

6.1 Challenges

As we discussed, HLS-based design development reveals a diverse set of security implications and offers a plethora of opportunities to develop security guidelines and countermeasures. However, HLS is still emerging and poses various unique challenges for security assessment, as listed below.

- HLS is still primitive and does not support all the features widely used in HLLs, such as pointer arithmetic, dynamic allocations, and so on. As a result, most of the secure open-source applications are not directly compatible with HLS compilers and need to be modified, which could introduce unknown vulnerabilities in the C/C++ design.

- Vulnerabilities discovered in HLS-generated RTL need to be prevented as soon as possible, i.e., at HLL design. Mapping the RTL vulnerabilities to the HLL design-code/constraints is not trivial, as C/C++ execution is sequential and RTL is parallel. Moreover, a single HLS optimization could have a drastic impact on the design's CDFG, making it challenging to map the changes. Machine-learning-driven HLS translation and vulnerability identification can help to improve the security of HLS-generated RTLs. However, it is currently limited due to the availability of a sufficiently large dataset of security vulnerabilities so machines can be trained based on them. Therefore, our efforts focus on identifying as many of these vulnerabilities as possible to be later used to train machine learning models to automate the verification for new designs. Identifying security implications that can arise during sequential to parallel translation and development of HLS-specific vulnerability database is a step forward towards developing such large sets for machine learning algorithms. Once established, machine-learning can help map the intermediate representation of the design (which resembles a graph with nodes being the resources and their connections as edges) to security vulnerabilities.
- Widely used commercial HLS compilers are protected closed systems. It is challenging to implement changes to the core compiler without collaboration between academia and industry.

7 CONCLUSION

High-level synthesis is becoming an industry norm for rapid hardware design development due to reduced design complexity and time-to-market. In this article, we discussed security vulnerabilities that could be introduced during HLS translation. We showed examples of how various optimization (e.g., performance, latency, area, and power) algorithms of HLS can unintentionally introduce vulnerabilities in the security-critical applications. Therefore, HLS needs to have secure programming rules and security analysis tools to detect and notify likely weaknesses that may occur during translation from HLL design specification to RTL.

REFERENCES

[1] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. 2012. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proc. IEEE* 100, 11 (2012), 3056–3076.
[2] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 16–29.
[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 33–36.
[4] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. 2009. An introduction to high-level synthesis. *IEEE Des. Test Comput.* 26, 4 (2009), 8–17.
[5] Philippe Coussy and Adam Morawiec. 2008. *High-level Synthesis: From Algorithm to Digital Circuit*. Springer Science & Business Media.
[6] Xiaotong Cui, Kun Ma, Liang Shi, and Kaijie Wu. 2014. High-level synthesis for run-time hardware Trojan detection and recovery. In *Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[7] J. C. Li Frans Sijstermans. 2019. Working Smarter, Not Harder: NVIDIA Closes Design Complexity Gap with High-Level Synthesis. Retrieved from https://go.mentor.com/4N9cP.
[8] Daniel D. Gajski and Loganath Ramachandran. 1994. Introduction to high-level synthesis. *IEEE Des. Test Comput.* 11, 4 (1994), 44–54.
[9] github. 1998–2021. openssl. Retrieved from https://github.com/openssl/openssl/tree/master/crypto/aes.
[10] Mentor Graphics. 2021. Catapult HLS. Retrieved from https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.
[11] Xiaolong Guo, Raj Gautam Dutta, Jiaji He, Mark M. Tehranipoor, and Yier Jin. 2019. QIF-Verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 91–100.

[12] Peter Gutmann. 2014. Encrypt-then-MAC for transport layer security (TLS) and datagram transport layer security (DTLS). *Req. Comm.* 7366.
[13] Yuko Hara-Azumi, Toshinobu Matsuba, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2014. Impact of resource sharing and register retiming on area and performance of FPGA-based designs. *Inf. Media Technol.* 9, 1 (2014), 26–34.
[14] Miao He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Tehranipoor. 2019. RTL-PSC: Automated power side-channel leakage assessment at register-transfer level. In *Proceedings of the IEEE 37th VLSI Test Symposium (VTS)*. IEEE, 1–6.
[15] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. DOI: https://doi.org/10.1109/ISSCC.2014.6757323
[16] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. 2018. High-level synthesis with timing-sensitive information flow enforcement. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.
[17] S. T. Choden Konigsmark, Deming Chen, and Martin D. F. Wong. 2017. High-level synthesis for side-channel defense. In *Proceedings of the IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 37–44.
[18] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. 2014. Gate-level masking under a path-based leakage metric. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 580–597.
[19] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A hardware description language for secure information flow. *ACM SIGPLAN Not.* 46, 6 (2011), 109–120.
[20] Michael C. McFarland, Alice C. Parker, and Raul Camposano. 1988. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*. 330–336.
[21] Mentor. 2020. Machine Learning at the Edge: Using HLS to Optimize Power and Performance. Retrieved from https://s3.amazonaws.com/s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_106005.pdf.
[22] M. Rafid Muttaki, Nitin Pundir, Mark Tehranipoor, and Farimah Farahmandi. 2021. Security assessment of high-level synthesis. In *Emerging Topics in Hardware Security*. Springer, 147–170.
[23] Adib Nahiyan, Farimah Farahmandi, Prabhat Mishra, Domenic Forte, and Mark Tehranipoor. 2018. Security-aware FSM design flow for identifying and mitigating vulnerabilities to fault attacks. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 38, 6 (2018), 1003–1016.
[24] Adib Nahiyan, Kan Xiao, Kun Yang, Yeir Jin, Domenic Forte, and M. Tehranipoor. 2016. AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.
[25] Daniel H. Noronha, Bahar Salehpour, and Steven J. E. Wilton. 2018. LeFlow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks. In *Proceedings of the 5th International Workshop on FPGAs for Software Programmers*. VDE, 1–8.
[26] Steffen Peter and Tony Givargis. 2016. Towards a timing attack aware high-level synthesis of integrated circuits. In *Proceedings of the IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 452–455.
[27] Christian Pilato, Francesco Regazzoni, Ramesh Karri, and Siddharth Garg. 2018. TAO: Techniques for algorithm-level obfuscation during high-level synthesis. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.
[28] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2018. TaintHLS: High-level synthesis for dynamic information flow tracking. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 38, 5 (2018), 798–808.
[29] Nitin Pundir, Farimah Farahmandi, and Mark Tehranipoor. 2021. Secure high-level synthesis: Challenges and solutions. In *Proceedings of the 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 164–171.
[30] Nitin Pundir, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2021. What is all the FaaS about? Remote exploitation of FPGA-as-a-service platforms. *Cryptology ePrint Archive* (2021). https://eprint.iacr.org/2021/74.
[31] Rajat Sadhukhan, Paulson Mathew, Debapriya Basu Roy, and Debdeep Mukhopadhyay. 2019. Count your toggles: A new leakage model for pre-silicon power analysis of crypto designs. *J. Electron. Test.* 35, 5 (2019), 605–619.
[32] Robert C. Seacord. 2008. *The CERT C Secure Coding Standard*. Pearson Education.
[33] Leon Stok. 1994. Data path synthesis. *Integration* 18, 1 (1994), 1–71.
[34] M. Tehranipoor and F. Koushanfar. 2010. A survey of hardware trojan taxonomy and detection. *IEEE Des. Test Comput.* 27, 1 (2010), 10–25.
[35] M. Tehranipoor and Cliff Wang. 2011. *Introduction to Hardware Security and Trust*. Springer Science & Business Media.
[36] Huanyu Wang, Henian Li, Fahim Rahman, Mark M. Tehranipoor, and Farimah Farahmandi. 2021. SoFI: Security property-driven vulnerability assessments of ICs against fault-injection attacks. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* DOI: 10.1109/TCAD.2021.3063998

[37] Xilinx. 2021. Vivado HLS. Retrieved from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[38] David Zaretsky. 2007. A high level synthesis tool for FPGA design from software binaries. *SBIR Award, Department of Defense*. https://www.sbir.gov/sbirsearch/detail/106849
[39] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A hardware design language for timing-sensitive information-flow security. *ACM SIGPLAN Not.* 50, 4 (2015), 503–516.
[40] Lu Zhang, Wei Hu, Armaiti Ardeshiricham, Yu Tai, Jeremy Blackstone, Dejun Mu, and Ryan Kastner. 2018. Examining the consequences of high-level synthesis optimizations on power side-channel. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1167–1170.
[41] Lu Zhang, Dejun Mu, Wei Hu, Yu Tai, Jeremy Blackstone, and Ryan Kastner. 2019. Memory-based high-level synthesis optimizations security exploration on the power side-channel. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 39, 10 (2019), 2124–2137.
[42] Haissam Ziade, Rafic A. Ayoubi, and Raoul Velazco. 2004. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.* 1, 2 (2004), 171–186.

Received February 2021; revised July 2021; accepted October 2021