Secure High-Level Synthesis: Challenges and Solutions

Nitin Pundir, Farimah Farahmandi, and Mark Tehranipoor
University of Florida
{nitin.pundir, ffarahmandi, tehranipoor}@ufl.edu

***Abstract*—High-level synthesis (HLS) has significantly reduced time and complexity of the hardware design by raising the abstraction to high-level languages (HLL) like C/C++. HLS has allowed non-hardware engineers to quickly prototype and test their algorithmic flow, and enabled hardware developers to build hardware quicker for emerging algorithmic designs such as machine learning (ML) and artificial intelligence (AI) networks. However, current HLS tools were not designed with security in mind as they only optimize the design for area, power, time, and throughput. As a result, security vulnerabilities may be introduced in the HLS-generated RTLs unintentionally. In this paper, we discuss some of the optimizations performed by HLS and present bad design coding practices in HLL that could lead to security vulnerabilities in the RTL. We also explore potential solutions, their limitations, and challenges moving forward to bring attention towards development of automated verification tools and guidelines to ensure secure HLS translation.**

***Index Terms*—high-level synthesis, hardware security, verification**

I. Introduction

High-level synthesis (HLS) takes as input the hardware definition of a design in a high-level language (HLL), i.e., C/C++, and translates it into the hardware description language (HDL), i.e., VHDL/Verilog modules [6]. HLS has significantly reduced the time and complexity involved with the logic design process using HDLs by raising the abstraction to widely adopted and easy to describe HLLs. Deploying HLS has helped entities with small design teams or from software background to compete with diminishing time-to-market (TTM) constraints and to rapidly develop complex hardware components. It has also helped with rapidly evolving designs like machine learning (ML) and artificial intelligence (AI), which lack legacy intellectual properties (IPs) while their algorithmic architecture evolves fast [19].

At inception, HLS was an easy way to test the performance of the algorithmic flow by promptly prototyping the hardware design and testing it on FPGAs. However, due to recent improvements in HLS compilers and competitive TTM constraints, HLS has found its way into IP and system-on-chip (SoC) development either deployed on FPGA-as-a-Service platforms [27] or used as third-party IPs during RTL integration. This wide adoption has raised concerns about the use of HLS compilers for security-critical applications because HLS compilers were not developed with security in mind and have only been optimized for performance constraints such as latency, throughput, area, and power.

There are critical aspects that need to be addressed when transitioning from C/C++ to RTL. For example, clock circuitry, reset circuitry, variables data width, variables sign (signed/unsigned), handling of data representation (fixed/floating point), memory location (register, ROM, RAMs, etc.), type of data paths (pipelined, sequential, parallel), etc. Some constraints are handled automatically by the matured HLS process, but others rely on the user's input. In addition, third-party HLS compilers have their own set of libraries and specifications to generate optimized hardware. Therefore, the user may be unaware of handling the critical aspects of the RTL when coding in C/C++. It could result in either functionally incorrect hardware or functionally correct but insecure hardware with an unintended threat surface. The former is easier to detect and fix by using various functional equivalency tools between HLL and HDL [15], [17]. However, the latter is more challenging to detect because HLS-generated RTLs are more complex to read and understand than a human-written RTL. Moreover, users may be unaware of the translation effect on the design's security, and it is challenging to detect them due to a wide variety of threat models.

Similarly, the skill and background limitations associated with hardware and software coding practices for experts from different domains can result in vulnerable HLS-generated RTL [16]. For example, an expert from the software domain may lack hardware design knowledge, HLS may be black-box to him/her, and may not completely comprehend the RTL design files. Similarly, the lack of software domain coding expertise and increased design complexity for hardware could result in insecure C/C++ coding practices and incomplete security verification efforts to verify the generated-RTL. Fig. 1 illustrates various causes associated with the different domain of expertise that could lead to vulnerable RTL design generation.

HLS has gained prominence in modern SoC design development due to its recent improvements and the ability to generate optimized HDL [14], [23], [31]. It has been actively used to generate RTL designs for security-critical applications such as cryptography, ML/AI networks, hashing algorithms, etc., [3]. The expectation is that such designs must be secure and free from security vulnerabilities. Therefore, it necessitates HLS to be aware of security-critical assets in the design during translation. Such security-critical assets could include encryption/decryption keys, hashing keys, weights and biases of ML/AI networks, etc. Failure to acknowledge these security-critical assets by HLS during optimization and translation

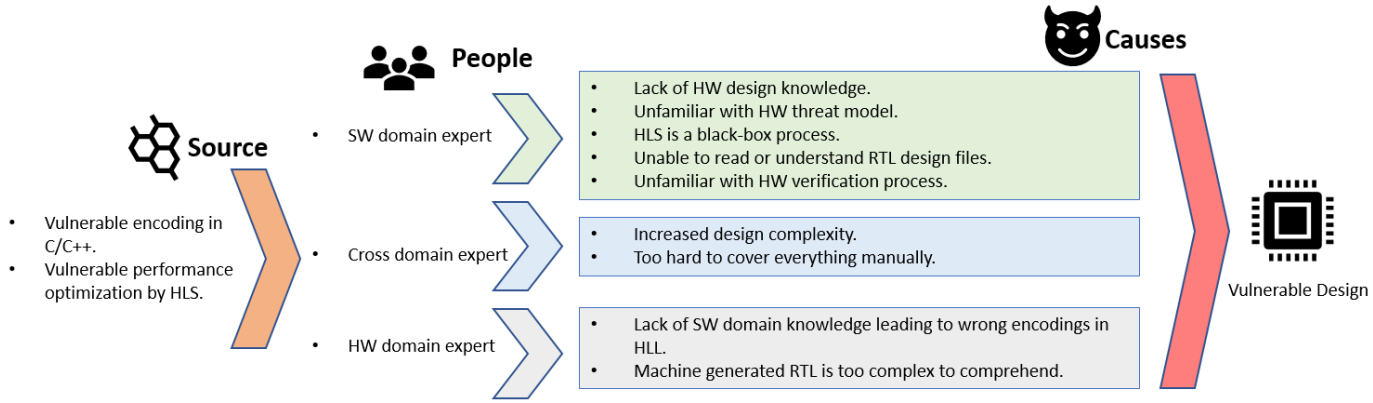

Fig. 1. Different sources, people, and causes leading to generation of vulnerable design using HLS.

could generate vulnerable RTLs [25]. The security implications of using such RTLs can include *information leakage*, *susceptibility to fault injection*, *access control violations*, or *side-channel leakage*. Therefore, there is a need to verify and ensure HLS translation is secure and the generated RTLs are free from security vulnerabilities. However, such verification methods should be universal and scalable to different design types. Therefore, automated security verification tools are required to verify the complex machine-generated RTL codes and ensure secure translation.

In this paper, we review different HLS optimization steps and design coding practices in HLL. We highlight their effect on design security and discuss how, if ignored, certain steps/practices could result in security vulnerabilities in critical applications. We address the limitations of each solution to illustrate the insufficiency of any single method to provide full security coverage during HLS translation. We present the need for security-aware optimization steps and algorithms in HLS. Finally, we outline the challenges present moving forward for implementing verification solutions.

The rest of the paper is organized as follows. Section II discusses the motivation behind making HLS translation secure. Sections III and IV briefly describe the steps involved during HLS translation and discuss different optimizations HLS adopts during translation, respectively. Section V highlights several security concerns in the generated-RTL design due to optimization strategies used by HLS and bad coding practices adopted by the user. Section VI discusses the existing and potential solutions and their respective challenges. Finally, Section VIII concludes the paper.

II. Motivation

Various entities are inclined towards the use of HLS because of the cost/TTM benefits it provides. However, vulnerable generated RTL could introduce severe consequences on the confidentiality, integrity, and availability of the design. Here, we broadly classified all entities into three groups: government, industry, and individuals/researchers. For these entities, there are expected benefits of using HLS, i.e., reduce TTM,

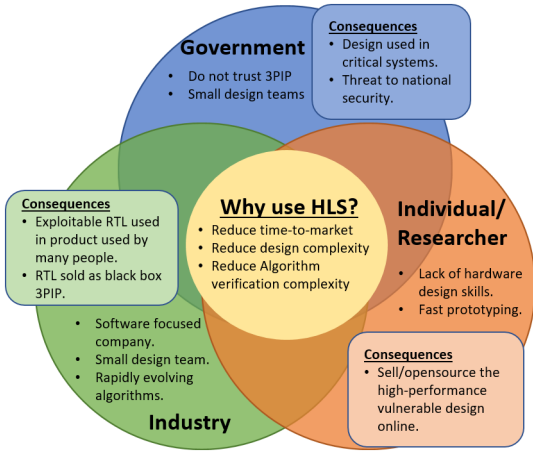

Fig. 2. Classification of different entities' benefits from using HLS and consequences for each for using vulnerable HLS-generated RTL.

design complexity, and algorithm verification complexity. Government and defense entities usually have smaller design teams and rely on third-party IPs, introducing trust issues in their supply chain [30], [32]. Therefore, the use of HLS for them obviates the need to rely on third-party IP providers. However, the security vulnerabilities introduced during the HLS process can pose a major threat to government agencies depending on the systems in which such HLS-generated IPs are used. For industrial entities, HLS gives them the advantage of simpler design descriptions that allows for more coverage in terms of verification in less time. However, HLS vulnerabilities can potentially compromise all software-hardware products and threaten its user's security and privacy. It also can harm the company's reputation. Similarly, for individuals and researchers, HLS is an effective way to test their algorithms by fast prototyping them into hardware (e.g., an FPGA) as many may be unfamiliar with RTL coding. However, they could avail their IPs as open-source or on the marketplace while these IPs may contain security vulnerabilities. Fig. 2 highlights the benefits of using HLS for all entities and

the associated consequences if the generated-RTL module is vulnerable. Therefore, it is imperative that we identify these potential vulnerabilities and automatically detect them in the generated RTL.

III. High-level Synthesis: A Brief Overview

HLS has been proven to rapidly generate RTLs from the C/C++ description optimized for area, latency, throughput, and power. Fig. 3 shows the overall flow of the RTL generation using HLS. The flow typically includes the C/C++ design and the testbench to ensure that the design's functionality is correct per the user's requirement. HLS then takes HLL design and generates functionally equivalent HDL. HLS is a complex task consisting of a series of individual steps: compilation, allocation, scheduling, binding, and FSM extraction [5], [8], [18]. Each of these steps is briefly discussed below.

- *Compilation:* The first step in the HLS translation is the compilation. HLS in the backend uses C/C++ compilers to compile the code. It helps to ensure the code is statically correct, and then software-level optimizations are applied. The code is then transformed into a formal representation of the control data flow graph (CDFG) to help identify the data dependencies between different operations. The CDFG is used to identify possible operations that could be scheduled in parallel. Depending on user constraints, HLS can additionally apply other optimization techniques such as loop unrolling, loop pipelining, etc., [4].
- *Allocation:* HLS identifies different hardware resources (e.g., adders, multipliers, memory elements, buses, etc.) needed for different operations. The HLS performs this within the technology library's constraints or the resource pool provided by the user.
- *Scheduling:* HLS schedules different operations of the design to different clock cycles while satisfying the data dependencies for operations. HLS performs scheduling to obtain the least design latency possible.
- *Binding:* Prior to this step, each design's operation is mapped to a functional unit or hardware resource. Binding aims at optimizing the total number of such functional units or hardware resources required to implement the design. The binding could be categorized into module binding, register binding, 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 [29]. Finally, at the interconnection binding, the connections between different resources are optimized.
- *Control Logic Extraction:* In the final step, the control signals are identified, and the finite state machines (FSMs) of the design are extracted. It results in the automatic generation of the FSM that controls the design's overall flow, and glues the final RTL blocks together.

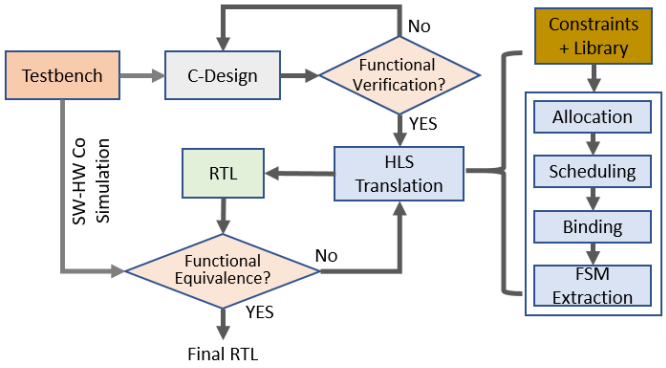

Fig. 3. Overview of HLS translation.

IV. HLS Optimization Strategies

HLS users expect to obtain highly-optimized RTL designs with the least area, timing, and power overhead. Hence, HLS performs various steps to generate the optimized RTLs. Understanding these optimization strategies adopted by HLS is critical for identifying potential security problems in the generated-RTL. In this section, we briefly discuss these optimization strategies and the underlying steps within each strategy.

A. Throughput Optimization

HLS attempts to improve the throughput of the RTL design. To achieve this, HLS inserts registers between operations of the functions and loops to create pipelines. HLS further tries to optimize when new data could be fed into the pipelines, also called the initiation interval. By default, HLS compilers target to achieve an initiation interval of one so that pipeline could start processing new data every clock cycle.

B. Latency Optimization

By default, the main goal of HLS compilers is to reduce the total execution clock cycles of the design and could resort to several techniques to achieve it. Some of these techniques are briefly described below.

- *Parallel Scheduling:* Identifies independent operations and loops that have no data dependency and schedule 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.

C. Area Optimization

The area is one of the key features which HLS could try to optimize during translation. Some of the key steps include:

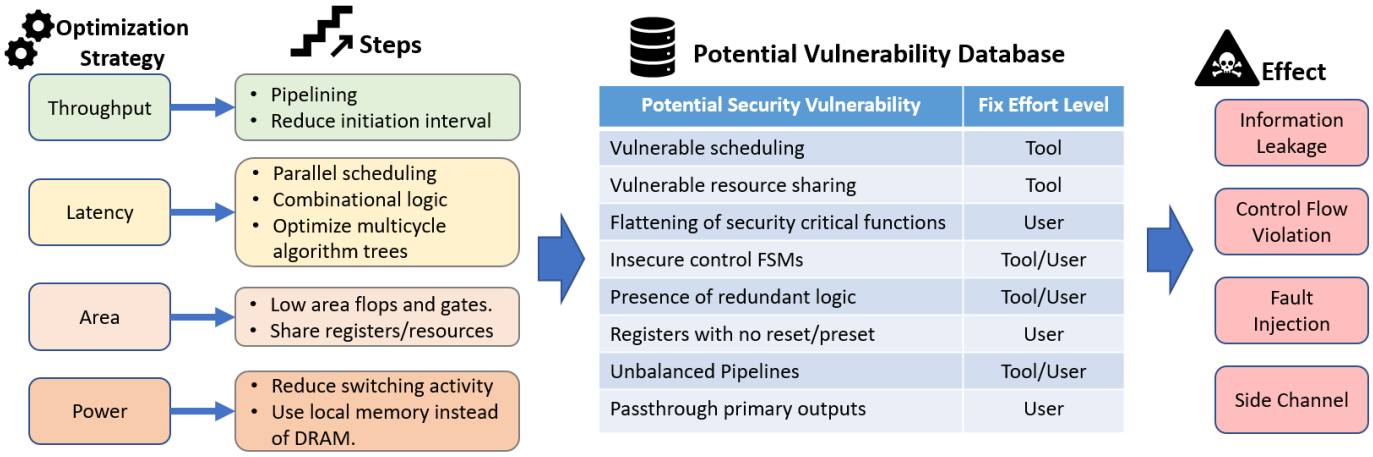

Fig. 4. HLS optimization strategies, potential security vulnerability database, and associated effort level to apply the potential fix.

TABLE I
AREA COMPARISON OF DIFFERENT TYPES OF FLIP-FLOP STANDARD CELLS OF SYNOPSYS 32NM LIBRARY.

TYPE	Standard cell	AREA μm^2
DFF	DFFX1	6.60774
	DFFX2	7.62432
DFF with reset	DFFARX1	7.116032
	DFFARX2	7.878464
DFF with preset	DFFASX1	7.116032
	DFFASX2	8.132608
DFF with reset and preset	DFFASRX1	7.62432
	DFFASRX1	8.132608

- Registers with low area overhead: HLS can opt for registers with a low area overhead. Table I shows area for different types of D flip-flops for 32nm library.
- Resource sharing: HLS can identify operations scheduled in different clock cycles and share hardware resources between them.
- Relying on external memory: Instead of using on-chip memory or generating ROMs for arrays, HLS could rely on external memory to fetch data and reduce the size of the generated RTL.

D. Power Optimization

Power is a critical component for modern SoC designs for which HLS could try to optimize the designs. HLS could spread out the high switching activity operations to reduce the dynamic power consumption of a design. HLS can also store data on-chip instead of fetching from external DRAM, which is at least 30x less power costly [12]. Similarly, HLS can also convert the floating-point representations to fixed-point/integer representations to conserve power in the generated RTL.

V. SECURITY VIOLATIONS

Earlier, we discussed a series of optimizations undertaken by the HLS compiler to effectively translate the HLL design specifications into a functionally-equivalent RTL design. These optimizations do not consider security when optimizing the design for throughput, latency, area, or power. Unfortunately, these optimizations could result in a violation of security requirements. In this section, we show cases of such security vulnerabilities in the generated-RTL. We classify these potential security vulnerabilities based on the effort-level required to mitigate them, as summarized in Fig. 4.

A. Vulnerable Scheduling

HLS attempts to reduce the overall latency of the design by scheduling multiple operations in parallel. HLS could identify operations executed in later stages of the design but have no data dependency from any prior operation. It could schedule such operations at the very beginning of the execution and store their results in internal registers, which are used in later stages of the design execution. Such scheduling sometimes could violate the security policy of "If A succeeds, then only B". For example, in security protocol "HMAC then decrypt" [10], decryption should only occur after the ciphertext has been authenticated. Though the decryption occurs after HMAC validation due to conditional statements in the C/C++ code, however, HLS could schedule certain operations of the decryption function in advance due to no data dependency.

Possible Solution: The user should identify security-critical operations in the C/C++ and establish data dependency to ensure they could only be scheduled one after the other. Moreover, the HLS scheduling algorithm could be updated appropriately to take tags from the C/C++ level to ensure certain operations could only be scheduled in the expected order. For example, in the case of "HMAC then decrypt", all operations related to the decryption function (at TAG 2) could only be scheduled after the completion of all HMAC operations (TAG 1).

B. Vulnerable Resource Sharing

During optimization, HLS may be asked to minimize the overall area of the design. To achieve this, it tries to share

registers and other hardware resources between operations. Since HLS have no idea of secure and non-secure operations in the design, resource sharing could lead to new attack surfaces. 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.
Possible Solution: HLS binding and resource sharing algorithms should be appropriately updated to identify and segregate the secure and non-secure operations and assets of the design to perform resource sharing securely.

C. Flattening Security Critical Functions

HLS could perform the inlining of functions during function calls to improve latency and avoid cycle overhead on function calls. If a function is security-critical, all the registers associated with that function should be verified for information leakage. However, if HLS flattens the function each time it is called in the main module, it significantly increases the design's security-critical registers. It could increase the verification efforts of ensuring none of these registers result in information leakage.
Possible Solution: The user can address this issue by identifying critical function calls in the design and ensuring HLS generates separate IP for such functions. It will help in consolidating verification efforts.

D. Non-secure FSM

HLS automatically extracts the FSM to control the datapath. Such FSMs are optimized to exploit parallelism, reduce area, or the number of states needed in a design. For example, HLS could implement binary or grey code encoded FSM instead of one-hot encoding or introduce additional don't care states. This could be done to reduce the state register size or synchronize data paths. However, these optimizations could make the FSM vulnerable to single event upsets and fault injection attacks [22].
Possible Solution: The FSM extraction should be performed by HLS more securely, and if possible, the user should be given a choice of FSM encoding styles. Nahiyan et. al proposed using "Finite-state FFs circuit" to implement secure FSMs resilient towards fault injection attacks [22].

E. Presence of Redundant Logic

Most HLS compilers follow a template-based translation, where multiple templates could be glued together, based on HLL code, to generate the final RTL. In many cases, these templates could result in redundant logic/variables and hanging nets in the design. If these hanging nets are exposed at the top-level, false load and drivers could be attached to them during RTL integration. As a result, they may get passed down the design cycle and may have an unexpected security impact on the design. For example, the redundant logic may lead to a slight imbalance in power consumption, causing power balancing based power side-channel resistant implementations like wave dynamic differential logics (WDDL) [13] vulnerable.
Possible Solution: The HLS compiler could perform a two-stage synthesis, while the latter stage focuses on cleaning up the redundant logic that may cause security violations in the generated RTL. The user can synthesize the individual IP to identify the hanging logic and nets in the design and remove them.

F. Registers with No Reset/Preset

HLS optimizes the design's area and may sometimes use hardware resources that have a smaller area than other variants. One such example is the use of registers with no reset, and this could present security challenges. For example, the registers could still hold secret information (encryption, decryption keys, etc.) even after applying the reset, making information susceptible to leakage.
Possible Solution: Most HLS compilers present their users with configuration options to choose which type of registers to use in the design. Users should identify and decide if registers values should be cleared after applying reset to prevent leakage but suffer area overhead penalty. Moreover, underlying HLS algorithms could be updated to choose the resource type if the particular resource will be handling the security asset. For example, suppose the security assets could propagate to specific registers. In that case, those registers should be deemed critical and should clear values at reset, and all other non-critical registers could retain values at reset.

G. Unbalanced Pipelines

To optimize dataflow and throughput, HLS inserts registers between operations and tries to create efficient pipelines. However, the pipelines between secure and non-secure variables could be unbalanced, resulting in the secret variable reaching a hardware resource early while the non-secret asset is still in the pipeline. For example, in AES, the round key reaching the XOR early while plaintext is still in the pipeline going through substitution, shiftrows, and mixcolumns. It could result in leakage of the secret if an attacker can manage to flush the pipeline.
Possible Solution: The HLS should be updated to allow tagging of the secure and non-secure variable so that the scheduling algorithm could ensure pipeline depths between secure and non-secure variables is matched. Automatic security verification approaches (such as information leakage analysis) should be developed/used to identify such violations.

H. Passthrough Primary Outputs

To optimize the latency and reduce area, HLS directly latches the intermediate register values to the primary outputs. It saves the additional FSM required to control when the output should be latched and cycle overhead.

For example, Listing 1 shows a simple HLL code, where the "foo" function returns the final computed sum at the end of the function execution. If the function "foo" is passed to the HLS as the top-level module, the variables "a" and "sum" will result in the input and output ports. As "sum" is changed for each iteration, this may result in the intermediate value of the

sum being mirrored at the output port due to a lack of safe data flow control. Such behavior is unsafe for sensitive applications, such as AES, where the state (ciphertext) registers should only be latched to the output port at the end of the final round. Otherwise, the encryption key or some part of it may be leaked.

Listing 1. Example C code.

```
1 int foo(int *a){
2     int sum = 0;
3     for( int i=0; i<len; i++){
4         sum = sum + a[i];
5     }
6     return sum;
7 }
```

Possible Solution: HLS compiler needs to be updated to generate more secure FSMs that prevents intermediate latching of data to output ports. As another solution, the user can identify such blocks and enable pipelining.

VI. A Brief Summary of Solutions and Limitations

As discussed previously, some vulnerabilities in the HLS-generated design could arise due to the user's negligence during coding the design in C/C++. It could be due to the user's lack of security expertise, increased design complexity, the difficulty of analyzing the complex machine-generated code, etc. However, some other vulnerabilities could be due to HLS itself as it is not designed with security in mind and its unawareness of security assets in the design. Solutions to the former may be easy to develop and implement compared to the latter case, which may require a complete overhaul of the HLS compiler.

In this section, we discuss some of the potential solutions (existing and future) that could be implemented to prevent the vulnerabilities in HLS-generated RTL designs. We also discuss the challenges and limitations associated with each solution.

Solution #1: HLS Security Vulnerability Database: A comprehensive database for HLS security vulnerabilities needs to be developed. Security vulnerabilities can be mapped to different HLS optimization strategies (throughput, area, time, and power), different strategy steps, and various C/C++ coding practices. The development of such a database is crucial for understanding various threat surfaces, developing security properties, and generating rules for C/C++ developers. These properties and rules could become part of automated verification tools.

Limitations: Developing such a database is a time-consuming and continuous process where new vulnerabilities could arise due to updates to the HLS tool or the adoption of HLS in the newer fields. During the development phase, the engineer could spend huge time identifying vulnerabilities and generating an exploitable test case.

Solution #2: Secure HLS Guidelines: Secure coding and constraint guidelines could be developed for HLS users as a series of do's and don'ts. These guidelines could be similar to CERT-C [26] but focused on securely coding a parallel hardware design in sequential C/C++ language. Some examples include:

- Security-critical functions should not be flattened.
- Security-critical arrays should be local to the design instead of fetching/storing from/to external memory.
- All registers on the sequential path of security assets should have global reset.

Limitations: Such guidelines can only be developed for standard practices and do not account for vulnerabilities that could be introduced due to HLS optimizations. The user constraints (pragmas, directives, libraries) and coding styles vary with the HLS compiler, posing challenges to accommodate such variations.

Solution #3: Formal Verification: Formal verification at RTL can ensure that design is secure and satisfies the security properties even after serial-to-parallel translation, FSM insertion, and hardware optimizations. Unlike simulation-based verification, formal verification is output-driven and complete in nature, making it ideal for security verification. Whenever evidence of security vulnerabilities are found, a set of security properties need to be generated. It ensures that future designs of C/C++ or RTL can be formally verified against these properties to ensure that the particular vulnerability is not present.

Limitations: Generation of such properties can take time, be done for each design individually, and may not be exhaustive to provide complete coverage. Moreover, not all security vulnerabilities could be expressed using formal properties, thus requiring the need for other verification methods.

Solution #4: Assertion-Based Verification: Security assertions and properties could be made part of the C/C++ design specification. For example, in C/C++ of AES design user can include assertion to ensure output port (ciphertext) has data only after the final round. The assertions get translated with the design and accommodate for any change in variable names. The HLS compiler ensures that the generated RTL also satisfies the assertions. An example of an industry-grade HLS compiler that supports such verification methodology is Catapult [9].

Limitations: The security assertions at the C/C++ level can only include variables available in C/C++. Assertions related to HLS inserted variables and FSMs cannot be included in the design. It could allow vulnerabilities introduced by insecure HLS translation to be overlooked during verification. This methodology also requires the generation of input test vectors to satisfy the design's response against security assertions/properties. Designs with large input vector widths can pose timing limitations against exhaustive testing.

Solution #5: Secure Optimization Algorithms: HLS algorithms responsible for scheduling, resource allocation, binding, loop optimization, etc., could be updated to incorporate security assets in the design. For example, resource allocation and register binding algorithms can ensure that the same resources

or registers are not shared between secure and non-secure operations. Similarly, the scheduling algorithm could be updated to ensure certain security-critical operations are scheduled in a specific order. The algorithms should be modified to take some user input about security assets in the design and consider them during translation.

Limitations: The HLS algorithms are optimized to reduce area, timing, and power, making them constraint to security properties that could incur significant overheads. Not only one, but all the underlying algorithms should be conscious of the security assets in the design to ensure secure translation. Such modifications could be time-consuming and may not be feasible in some cases due to the involved cost and timing constraints.

Solution #6: Automated Verification Tools: There is a need to develop automated security verification tools for HLS translation [7]. Such tools could exist at both C/C++ and RTL levels. Tools at C/C++ can help users identify early if insecure HLS coding practices are used. They could analyze user constraints (pragmas and directives) to help users visualize the hardware that will be generated after the HLS optimizations. For example, such tools could identify the data dependency between operations to create a map when different operations are scheduled. It could help users to ensure scheduling optimizations do not violate security policies such as "If A then only B". In short, C/C++ level tools may help create an abstract model of ought to be generated RTL to facilitate security verification and early ability to adjust user constraints to achieve the desired hardware after translation. Tools at RTL-level could ensure that the final RTL is free from vulnerabilities by analyzing for information leakages, fault injection [20], control flow violations, side-channel leakages [11], [21], etc. These tools can be more accurate than C/C++ level tools because they could consider the logic and variables introduced during translation, such as FSM, registers, etc.

Limitations: Tools developed for C/C++ design code can only consider algorithmic flow and user constraints during analysis. FSMs and other additional logic/variables introduced after translation will be hard to model and could vary across HLS compilers. The user constraints such as pragmas and directives are also unique to the HLS compiler, thus, posing a challenge to develop such tools scaled across different HLS compilers. Certain verification tools at RTL-level may be inaccurate due to a lack of targeted technology information at the RTL-level, for example, side-channel analysis (power, electromagnetic, etc.).

VII. Challenges Ahead

Security solutions and their limitations discussed in the previous section established that a single verification methodology is not sufficient to provide completeness in limited timing constraints. Therefore, there is a need for a combination of automated security verification tools and guidelines to ensure secure HLS translation. The development of such tools and guidelines is a non-trivial task and faces many challenges. Some of these challenges which affect future work in this domain are discussed below:

- *Closed System:* There is a scarcity of open-source HLS compilers that could match industry standards. Chisel, which is widely used to generate RTL for SoCs and processors, is not HLS but a scala-based language to describe hardware design [1]. Widely used commercial HLS compilers such as Vivado HLS [33], Catapult [9], Stratus [2], etc., are protected closed systems. The solutions which require modifications in the compiler can only be possible after collaboration between industry and academia. Further, solutions implemented for one compiler may not be implemented on others due to protected copyright issues.
- *Non-standardized Translation:* Every compiler comes with its own set of specifications and coding style. Cadence Stratus [2], for example, requires that clock and reset signals be defined in C/C++ whereas Vivado HLS [33] and Catapult [9] do not, and clock and reset signals are generated in the RTL automatically. HLS compilers have their own set of library files to optimize the translation of certain logic, such as floating points, custom data widths, DSPs, etc. The non-standardization that prevails across HLS compilers poses a unique challenge towards developing uniform solutions and guidelines that could be implemented across different HLS compilers.
- *Protected Algorithms:* The underlying algorithms used by HLS compilers are protected trade secrets. Thus, gaining access to such algorithms and modifying them can pose unique challenges. The proof-of-concept could be demonstrated on widely researched HLS algorithms such as As-soon-as-possible (ASAP), As-last-as-possible (ALAP), simulated annealing, etc., [24], [28]. But integration into existing products is only feasible after strong industry-academia collaboration.

VIII. Conclusions

HLS is gained much popularity for rapidly generating hardware designs by easily coding in HLLs, thus reducing design complexity and time-to-market. In this paper, we discussed HLS generic flow and different optimizations it can perform to improve design throughput, area, latency, and power. We also discussed how insecure coding practices in HLL and some HLS optimizations could become the source of vulnerabilities in the translated RTL. Finally, we discussed solutions and challenges to mitigate those vulnerabilities, paving the way towards secure HLS translation.

Acknowledgement

This work was supported in part by Semiconductor Research Corporation (SRC) Grant #2019-TS-2910.

References

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[2] Cadence. Stratus. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[3] A Cortes, I Velez, and A Irizar. High level synthesis using vivado hls for zynq soc: Image processing case studies. In *2016 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2016.

[4] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[5] Philippe Coussy and Adam Morawiec. *High-level synthesis: from algorithm to digital circuit*. Springer Science & Business Media, 2008.

[6] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[7] Farimah Farahmandi and Mark Tehranipoor. *CAD for hardware security*. Springer, 2021.

[8] Daniel D Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, 1994.

[9] Mentor Graphics. Catapult hls. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[10] Peter Gutmann. Encrypt-then-mac for transport layer security (tls) and datagram transport layer security (dtls). *Request for Comments*, 7366, 2014.

[11] Miao He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Tehranipoor. Rtl-psc: Automated power side-channel leakage assessment at register-transfer level. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2019.

[12] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[13] David D Hwang, Kris Tiri, Alireza Hodjat, B-C Lai, Shenglin Yang, Patrick Schaumont, and Ingrid Verbauwhede. Aes-based security coprocessor ic in 0.18-$muhboxm$ cmos with resistance to differential power analysis side-channel attacks. *IEEE Journal of Solid-State Circuits*, 41(4):781–792, 2006.

[14] Brucek Khailany, Evgeni Krimer, Rangharajan Venkatesan, Jason Clemons, Joel S Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, et al. A modular digital vlsi flow for high-productivity soc design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[15] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating high-level synthesis. In *International Conference on Computer Aided Verification*, pages 459–472. Springer, 2008.

[16] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011.

[17] Anmol Mathur, Masahiro Fujita, Edmund Clarke, and Pascal Urard. Functional equivalence verification tools in high-level synthesis flows. *IEEE Design & Test of Computers*, 26(4):88–95, 2009.

[18] Michael C McFarland, Alice C Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 330–336, 1988.

[19] Kevin Morris. Hls powers ai revolution. https://www.eejournal.com/article/hls-powers-ai-revolution/.

[20] Adib Nahiyan, Farimah Farahmandi, Prabhat Mishra, Domenic Forte, and Mark Tehranipoor. Security-aware fsm design flow for identifying and mitigating vulnerabilities to fault attacks. *IEEE Transactions on Computer-aided design of integrated circuits and systems*, 38(6):1003–1016, 2018.

[21] Adib Nahiyan, Jungmin Park, Miao He, Yousef Iskander, Farimah Farahmandi, Domenic Forte, and Mark Tehranipoor. Script: A cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 25(3):1–27, 2020.

[22] Adib Nahiyan, Kan Xiao, Kun Yang, Yeir Jin, Domenic Forte, and Mark Tehranipoor. Avfsm: a framework for identifying and mitigating vulnerabilities in fsms. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[23] Erdal Oruklu, Richard Hanley, Semih Aslan, Christophe Desmouliers, Fernando M Vallina, and Jafar Saniie. System-on-chip design using high-level synthesis tools. 2012.

[24] Pierre G Paulin and John P Knight. Scheduling and binding algorithms for high-level synthesis. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 1–6, 1989.

[25] Nitin Pundir, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi. Analyzing security vulnerabilities induced by high-level synthesis in socs, 2020.

[26] Robert C Seacord. *The CERT C secure coding standard*. Pearson Education, 2008.

[27] Amazon Web Services. F1 instances. https://aws.amazon.com/.

[28] Azeddien M Sllame and Vladimir Drabek. An efficient list-based scheduling algorithm for high-level synthesis. In *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*, pages 316–323. IEEE, 2002.

[29] Leon Stok. Data path synthesis. *Integration*, 18(1):1–71, 1994.

[30] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[31] Kazutoshi Wakabayashi and Takumi Okamoto. C-based soc design flow and eda tools: An asic and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1507–1522, 2000.

[32] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 15–19. IEEE, 2008.

[33] Xilinx. Vivado hls. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.