Using Static Analysis for Enhancing HLS Security

Luca Collini, Joey Ah-kiow, Christian Pilato, *Senior Member, IEEE*,
Ramesh Karri, *Fellow, IEEE* and Benjamin Tan, *Member, IEEE*

***Abstract*—Due to the increasing complexity of modern integrated circuits, High-Level Synthesis (HLS) is becoming a key technology in hardware design. HLS uses optimizations to assist during design space exploration. However, some of them can introduce security weaknesses. We propose an approach that leverages static analysis to identify a class of weaknesses in HLS-generated code. We show that some of these weaknesses can be corrected through the automatic generation of HLS directives. We evaluate our approach by comparing the static analysis results with formal verification. Our results show that the static approach has the same accuracy as formal methods while being 3× to 200× faster.**

I. Introduction

Systems-on-Chips (SoCs), which integrate multiple Intellectual Property blocks (IP), have increased computer system complexity. Design and verification methodologies and tools have not evolved and scaled as much as the size and complexity of SoC designs [1]. It has thus become increasingly difficult to properly design and verify hardware while meeting time-to-market commitments. High-Level Synthesis (HLS) aims to alleviate the design bottleneck by raising the abstraction level to the algorithmic one [2], [3]. HLS enables designers to start with algorithms expressed in a High-Level Language (HLL) and explore the space of corresponding hardware designs using HLS tool *directives* or source-code *pragmas*. Designers can explore the large design space offered by HLS and evaluate Power, Performance, and Area (PPA) trade-offs, with recent work proposed to help designers explore the large space of configurations [4].

However, current commercial HLS tools generate Register-Transfer level (RTL) designs without considering security. Starting from an algorithm and using different HLS directives, one can obtain different results in terms of latency, resources, and weaknesses. Designers focusing only on traditional metrics can introduce unintentional security weaknesses [5].

With hardware security concerns driving regulations [6], we need security-conscious HLS tools and verification frameworks to systematically assess design security, even when designers are not security experts.

We would like to acknowledge CMC Microsystems for the provision of products and services that facilitated this research. This research work is supported in part by a gift from Intel Corporation. This work does not in any way constitute an Intel endorsement of a product or supplier. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2022-03027.

L. Collini, J. Ah-kiow and R. Karri are with the Center for Cybersecurity, New York University, New York City, NY, 11201 USA. E-mail: {lc4976, joey.a, rkarri}@nyu.edu.

B. Tan is with the Department of Electrical and Software Engineering, University of Calgary, Calgary, AB, T2N 1N4, Canada. Email: benjamin.tan1@ucalgary.ca. J. Ah-kiow was with the University of Calgary for a portion of this research.

C. Pilato is with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, 20133, Italy. Email: christian.pilato@polimi.it

Thus, we propose to use static analysis to detect weaknesses in HLS-generated RTL and an automated repair flow to correct detected weaknesses by generating HLS directives that enable changes to the RTL without modifying the high-level code. We evaluate our work on synthetic benchmarks that highlight how similar high-level patterns can yield different RTL designs and conduct a case study analyzing three block ciphers (AES, serpent, and PRESENT from [7]) to explore the weaknesses that can be introduced by a commercial HLS tool. We show that we can detect these weaknesses automatically and provide fixes to the designers. Our contributions are twofold: (i) An approach for the detection and correction of weaknesses in HLS generated designs (Section III). (ii) A proof-of-concept implementation that we evaluate experimentally on three crypto cores (Section IV).

II. Background and Prior Work

A. HLS-Induced Security Weaknesses

Recent work started to consider the potential for security issues induced by HLS. For example, Pilato et al. [8] and Basu et al. [9] examined the possibility of Hardware Trojan (HT) insertion through compromised HLS tools. HLS is a prime candidate for HT insertion since it is hard to correlate the HLL description to the RTL. HTs can be partially remedied by equivalence checking for mismatches between HLL and RTL; Abderehman et al. [10] propose a C-to-RTL equivalence checking framework by extracting the RTL-level finite-state machine with datapaths (RTL-FSMDs).

Another concern is the *unintentional* introduction of security weaknesses in HLS-generated designs [5]. Design weaknesses are flaws in a design that, if exploited, can lead to vulnerabilities. Unlike HTs, weaknesses are not intentional mismatches and may not be detected through equivalence checking. MITRE maintains a list of known weaknesses called the Common Weakness Enumeration (CWE) [11]. Some vulnerabilities identified by Pundir et al. [5] map to the following CWEs: CWE 1245 (insecure finite state machine); CWE 1300 (improper protection of side channel); CWE 1271 (uninitialized value on reset for registers holding security settings); CWE 1189 (improper isolation of shared resources on system-on-a-chip); CWE 203 (Observable discrepancy). Of these weaknesses, we focus on CWE 203, as it may enable other weaknesses, like uninitialized values on reset or unbalanced pipelines, to leak sensitive data during operation. This is a problem for ciphers like AES that guarantee security *only after* all rounds have been completed, not before.

B. RTL Static Analysis

Our work is positioned at the junction of static analysis for weakness detection in RTL and security weaknesses in HLS-generated designs. Prior work [12] showed the use of

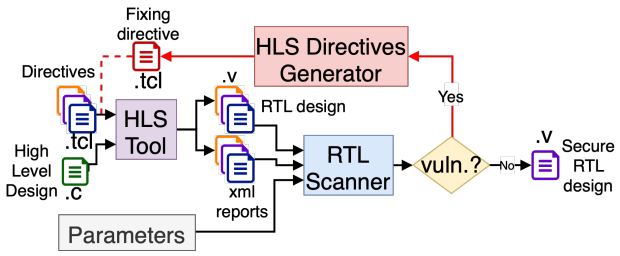

Fig. 1. Our proposed detection and correction flow, where source files in a high-level language like C are passed to an HLS tool with accompanying HLS directives (e.g., in a `.tcl` file). We feed the generated hardware design (RTL) through a scanner to identify weaknesses; if present, we generate a new directive file in an attempt to remediate the weakness.

static analysis of RTL to identify weaknesses. Their work targeted human-written RTL and addressed the challenge of identifying patterns that indicate possible weaknesses, even when designs assume different structures, demonstrating that static analysis can support early detection and fixes, reducing the verification burden in later design stages. In contrast, we focus on HLS-generated code and the template-based nature of commercial HLS tools. We observe that control signals are named consistently by default, and functionally similar patterns are syntactically identical. There is an opportunity for accurate and practical static analysis of the RTL code for HLS-generated designs, since it is likely that there are fewer unique weakness patterns that must be heuristically determined.

III. Weakness Detection and Design Correction

Given the potential introduction of security weaknesses during HLS, we propose a framework to detect/correct weaknesses in HLS designs (Fig. 1). The framework takes a C/C++ design, a set of directive files D for optimizations, and parameters specifying the weaknesses to scan for and environment settings. An HLS engine is run for each directive file $d \in D$ obtaining tuples (synthesized design, report) H, $|H| = |D|$. We feed each tuple to the RTL scanner which analyzes the Verilog for weaknesses. The scanner uses HLS reports to identify primary inputs/outputs and generated RTL. Scanning the outputs of the HLS tool and using built-in directives to fix weaknesses allows us to work with commercial tools since we do not need access to the source code.

A. RTL Scanner Design

At first, a Verilog parser extracts the Abstract Syntax Tree (AST), and an HLS report parser extracts information such as the top module and primary inputs/outputs from the structured report file generated by an HLS tool (e.g., in XML format).

This information is passed to our RTL scanner, which is implemented using the visitor pattern, a common design pattern to explore tree/graph data structures. The RTL Scanner includes one or more Visitors for each weakness; new weakness scanners can be added by implementing new visitors. When exploring the AST, the Visitor looks for patterns that are related to the weakness in scope. Since the analyzed RTL is automatically generated, the scanners can use the patterns and structures characteristic of the specific HLS tool used, obviating the need for generality as desired in prior work [12].

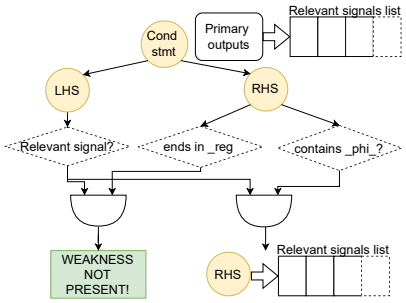

Fig. 2. Scheme of the passthrough visitor.

As discussed in Section II, the passthrough weakness enables other weaknesses as it leaks data to primary outputs before the computation is complete. For this reason, we implemented a scanner for the passthrough weakness.

B. The Passthrough Scanner

Our scanner needs information such as the names of the top module, primary input/outputs. By inspecting synthetic benchmarks with and without the passthrough weakness after using a commercial HLS tool, we observed several patterns. In designs without passthrough, the primary output is assigned with non-blocking substitution; the primary output is assigned in continuous assignment to a signal with post-fix "`_reg`"; the primary output is assigned to a Phi variable ϕ, which has one of the two properties above.

Taking into account these observations, one can design a visitor for the passthrough weakness, for example, like one illustrated in Fig. 2 (one can make adjustments for other HLS tools). Our visitor defines a function for analyzing continuous assignments. Before calling the function, the relevant signal list is initialized with the primary outputs. The function then checks if the left-hand side is in the list and if so, it checks if the right-hand side has a post-fix "`_reg`", or includes "`_phi`" in its name. If this condition is met then a flag to signal the weakness is not present is raised and the function exits. If the right-hand side includes "`_phi`", then its name is added to the relevant signal list `phi_list` if not already present. At the end of the traversal, if the list of relevant signals has increased, the AST is scanned again. If the scanning ends without raising the "not present" flag, we infer the weakness is present.

C. Correction via HLS Directives

Two conditions must be met to mitigate the passthrough weakness: (i) a registered output, (ii) appropriate control logic. The registered output is necessary to separate the intermediate output net to the top-level output net. The control logic enables the added register only when the operation is complete. We intuited that weaknesses can be remediated using the directives of an HLS tool, the idea being that after we detect a weakness, we can add the required directive(s) to the corresponding file of the HLS project. After re-running the synthesis, we can scan the generated design again to validate that the weakness has been fixed. If it is present, an error message is raised to get the designer's attention. We manually investigated the directive documentation of a typical commercial tool for directives that make

the outputs registered, and identified `config_interface -register_io scalar_out`. This is a configuration command applied at the solution-level that controls the default IO interface synthesized by the HLS tool for each function, specifying that all scalar outputs must be registered. For secure IP like crypto accelerators, there should not be intermediate updates to the output. Similar directives exist for other tools; the effort to repeat this process for other tools is a one-off.

IV. Experimental Evaluation

We implemented the prototype using Python and a commercial HLS tool[1]. We used Pyverilog to parse the design and extract the AST [13] and the Python ElementTree XML API to parse the XML reports. Our implementation will be open-sourced. We used Cadence JasperGold to run formal verification to verify the passthrough property as a baseline.

A. Benchmarks

For evaluation, we selected three synthetic benchmarks to explore the HLS-induced passthrough weakness. We use an implementation of the Fibonacci sequence and a factorial function to represent designs with loops updating the internal state to compute their results. Fibonacci implementation has the passthrough weakness when implemented with no HLS directives. The Factorial implementation has a similar structure but does not exhibit the weakness. Combining the two functions in a module that allows one to select one of them via a control input, does not present this weakness when the functions are inlined. When not inlined, Fibonacci sequence leaks to the output when selected. These benchmarks serve to illustrate that similar c structures can yield different results, motivating scanning for issues on the generated RTL.

Additionally, we selected block ciphers, AES(-v1), PRESENT and Serpent from [7] as benchmarks. We also selected the open-source AES(-v2) from the Vitis Library [14]. The cipher implementations came with HLS directives for each design as they were HLS-ready. We derived 5 sets of directives: (1) All directives (ALL) from the original set; (2) Register I/O (REG): directives in the original set plus the `config_interface -register_io scalar_out` directive; (3) Only pipelining (PIPE): Original set of directives removing all unrolling directives; (4) Only unrolling (UNROLL): Original set of directives removing pipeline directives; (5) No directives (EMPTY): no directives specified; tool's defaults as in [5]. This allows us to study choices that a designer could make.

B. Experimental Setup and Results

To explore the feasibility of the proposed framework, we take a set of benchmark designs, D, and a set of directives P and check if the combination $< D, p >, p \in P$ contains the passthrough weakness after HLS. If it does, we want to check that $< D, p + f >$, with f being the fixing directive, does not contain the passthrough weakness. We feed the designs through our framework running on 4 cores on an Intel Xeon Gold 6248R with 16GB of RAM. We characterize the benchmarks in terms of their complexity (lines of C code and lines of generated Verilog), their resource requirements in terms of latency, flip-flops, and LUTs, and run times, with results in Table I. For brevity, we report the results of the fix applied only to the original set of directives (REG rows). The results show that the weakness was correctly identified in all cases by the scanner as verified with the formal tool.

[1]We cannot disclose the tool due to license agreements.

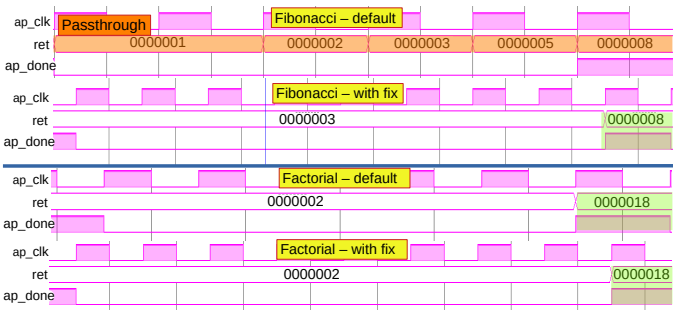

Fig. 3. Waveform of Fibonacci sequence (top) and factorial (bottom) with both default settings and register I/O directive. Notice how Factorial does not exhibit the weakness but adding the directives adds latency to the computation.

Synthetic Benchmarks. As shown in the waveforms in Fig. 3, when the default settings are used, both the Fibonacci sequence and the factorial exhibit the passthrough weakness. Without inlining, only the Fibonacci sequence leaks data on the output during the computation. When both functions are inlined the passthrough weakness is absent. This is illustrated in Fig. 4. For complex scenarios, it is hard to predict the security outcomes by looking only at HLL code.

Serpent. This block cipher exhibited a passthrough when the directive to register all outputs was not used. However, using such a directive increased latency from 36→337 cycles.

PRESENT. This block cipher also exhibited passthrough in all cases without the directive to register the output registers. The directive increased latency from 73→110 cycles.

AES-v1. This also exhibited passthrough in all cases that did not include the directive to register the output registers. The directive increased latency from 532→563 cycles.

AES-v2. This also exhibited passthrough in all cases where we did not include the directive to register the outputs. Fig. 5 shows the passthrough of the subkey used at round 1. In this case, the directive had a minimal latency increase: 166→170.

Overall, our scanner identified all instances of the passthrough weaknesses. The scanner's bottleneck is the Verilog parser. We used Pyverilog [13] since it is open-source, while faster commercial Verilog parsers could reduce this bottleneck. However, this is not the focus of this work. Furthermore, our fixing directive increases latency from 1.1× to 9× in the worst case, emphasizing the need to scan for the weakness rather than applying it universally.

V. Conclusion

We proposed a framework to detect and correct a class of security weaknesses that can be induced by high-level synthesis. Our experimental results on synthetic algorithms and realistic block cipher benchmarks show that our prototype is as effective at detecting the weakness in generated Verilog as more cumbersome formal verification while being up to

TABLE I
EXPERIMENTAL RESULTS SHOWING BENCHMARK CHARACTERISTICS AFTER HLS AND WEAKNESS PRESENCE AS DETECTED BY OUR SCANNER. DESIGN COMPLEXITY IS REPRESENTED BY THE LINES OF CODE (LOC). TCL SUMMARIZES THE DIRECTIVE SET USED FOR SYNTHESIS.

DESIGN	LOC (.c)	TCL	LOC (.v)	Passthrough	Design Characteristics			Time (s)		
					Latency	FF	LUT	Scanner Parse	Scanner Scan	Formal
Factorial	9	DEFAULT	597	N	**	104	172	1.225	0.0015	3.934
		REG	497	N	**	171	177	1.222	0.0020	3.757
Fibonacci	11	DEFAULT	314	Y	**	98	159	1.206	0.0009	3.678
		REG	249	N	**	181	144	1.219	0.0012	3.803
Combined	23	INLINE OFF	1067	Y	**	255	352	1.241	0.0015	3.866
		INLINE OFF + REG	928	N	**	388	353	1.211	0.0012	4.062
		REG	786	N	**	338	313	1.256	0.0016	3.831
		EMPTY	922	N	**	206	307	1.230	0.0021	3.977
PRESENT [7]	205	ALL	5405	Y	73	539	2105	1.323	0.0052	4.140
		REG	5535	N	110	808	2135	1.362	0.0057	80.66
		PIPELINE	4488	Y	205	681	1697	1.357	0.0051	4.274
		UNROLL	3463	Y	130	766	1598	1.317	0.0044	4.334
		EMPTY	1717	Y	5956	659	2340	1.316	0.0031	14.174
serpent [7]	331	ALL	4129	Y	36	667	1543	1.243	0.0024	5.399
		REG	5303	N	337	4433	1766	1.276	0.0034	300.83
		PIPELINE	4129	Y	36	667	1543	1.274	0.0026	5.871
		UNROLL*	-	-	-	-	-	-	-	–
		EMPTY	2574	Y	21944	4434	5964	1.248	0.0025	12.414
AES-v1 [7]	372	ALL	6751	Y	532	1266	5860	1.287	0.0062	4.344
		REG	7038	N	563	1983	5953	1.298	0.0062	204.19
		PIPELINE	6325	Y	584	1629	11346	1.312	0.0062	4.552
		UNROLL	4516	Y	1048	1453	6343	1.307	0.0055	4.427
		EMPTY	10535	Y	10542	2015	15082	1.304	0.0053	4.571
AES-v2 [14]	303	ALL	3660	Y	166	1979	1824	1.215	0.0016	6.669
		REG	3387	N	170	2368	1800	1.222	0.0019	189.241
		PIPELINE	3660	Y	166	1824	1824	1.205	0.0016	6.054
		UNROLL	3190	Y	67	1290	1594	1.249	0.0032	7.334
		EMPTY	2989	Y	1103	2247	27388	1.228	0.0024	6.234

*Output does not update **Latency depends on inputs

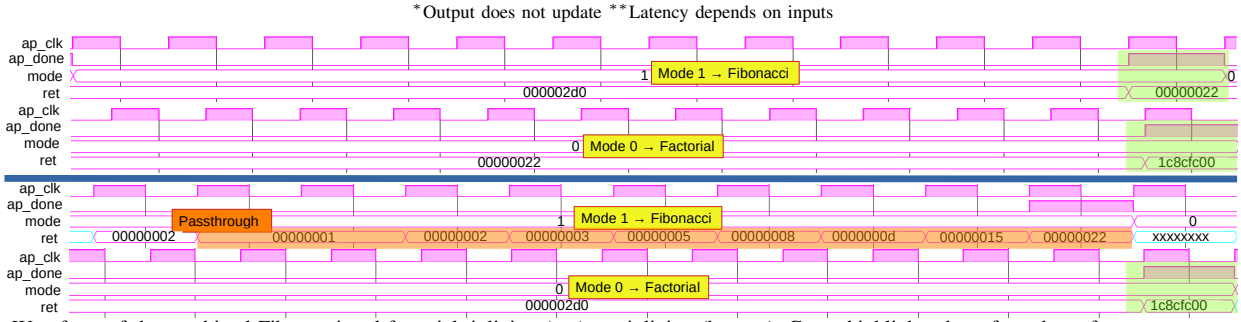

Fig. 4. Waveform of the combined Fibonacci and factorial, inlining (top), no inlining (bottom). Green highlights the safe update of output.

Fig. 5. Waveform of AES from Vitis Libraries [14], with (top) and without (bottom) fix. Orange shows passthrough leaking the first sub key in round 1.

200× faster even without a fast Verilog parser. This shows the potential for static analysis for weakness detection on HLS generated designs that can take advantage of template-based code structures, motivating research for new static scanners. In further exploring this work, we aim to generalize the support for other commercial HLS tools and increase the number of scanners to cover more vulnerabilities.

REFERENCES

[1] G. Dessouky *et al.*, "HardFails: Insights into Software-Exploitable hardware bugs," in *28th USENIX Security Symp. (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230.

[2] R. Nane *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, 2016.

[3] Siemens, "Working smarter, not harder: Nvidia closes design complexity gap with hls."

[4] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, 2020.

[5] N. Pundir *et al.*, "Analyzing Security Vulnerabilities Induced by High-level Synthesis," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 18, no. 3, pp. 1–22, Jul. 2022.

[6] M. Bartock *et al.*, "Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST IR 8320, 2022.

[7] P. Socha, V. Miškovský, and M. Novotný, "High-level synthesis, cryptography, and side-channel countermeasures: A comprehensive evaluation," *Microprocessors and Microsystems*, vol. 85, p. 104311, 2021.

[8] C. Pilato *et al.*, "Black-Hat High-Level Synthesis: Myth or Reality?" *IEEE Trans. VLSI Syst.*, vol. 27, no. 4, pp. 913–926, Apr. 2019.

[9] K. Basu *et al.*, "CAD-Base: An Attack Vector into the Electronics Supply Chain," *ACM Transactions on Design Automation of Electronic Systems*, vol. 24, no. 4, pp. 38:1–38:30, Apr. 2019.

[10] M. Abderehman *et al.*, "BLAST: Belling the Black-Hat High-Level Synthesis Tool," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3661–3672, Nov. 2022.

[11] The MITRE Corporation, "CWE - CWE-1194: Hardware Design (4.1)," https://cwe.mitre.org/data/definitions/1194.html, 2022.

[12] B. Ahmad *et al.*, "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design," Sep 2022, http://arxiv.org/abs/2209.01291.

[13] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Proc. of ARC*, 2015, pp. 451–460.

[14] Xilinx, "Vitis accelerated libraries," https://github.com/Xilinx/Vitis_Libraries/tree/master.