The Quest to Build Trust Earlier in Digital Design

Benjamin Tan
Department of Electrical and Software Engineering
University of Calgary
Calgary, Alberta, Canada
benjamin.tan1@ucalgary.ca

***Abstract*—The ever-rising complexity of computer systems presents challenges for maintaining security and trust throughout their lifetime. As hardware forms the foundation of a secure system, we need tools and techniques that support computer hardware engineers to improve trust and help them address security concerns. This paper highlights a vision for tools and techniques to enhance the security of digital hardware in earlier stages of the digital design process, especially during design with hardware description languages. We discuss the challenges that design teams face and explore some recent literature on understanding, identifying, and mitigating hardware security weaknesses as early as possible. We highlight the opportunities that emerge with open-source hardware development and sketch some open questions that guide ongoing research in this domain.**

I. Introduction

Designing computer systems is challenging. Not only do designers have to work hard to satisfy functional requirements (often under considerable time pressure), but increasing device interconnectivity and desire for computers in sensitive applications introduce security requirements into the fold. Naturally, we want to identify potential shortcomings in security in earlier stages of digital design, thus building trust in our overall system. Building trust earlier in design by identifying and addressing potential weaknesses is also beneficial because as we progress through the design process, the cost of design changes considerably increases. As best practice, designers should consider adopting a security development lifecycle (SDL) (e.g., [1]) where a security mindset is adopted throughout the design process. Teams need to define security objectives, formulate meaningful threat models, implement, and then verify and validate security mechanisms. Careful thought about support over the lifespan of a released product in the field is needed.

However, while software designers have at their disposal many potential tools to help with security throughout the design flow (e.g., [2]), hardware designers do not yet have such luxury [3]–[5]. In fact, systemization of how we think about security weaknesses is emerging and evolving, with recent efforts like the introduction of the hardware Common Weakness Enumerations (CWEs) [6] and standardization efforts like Accellera's Security Annotation for Electronic Design Integration Standard (SA-EDI) [7] and IEEE's P3164 working group [8] revealing industry-led efforts to tackle security issues. Recent competitions like Hack@DAC [4], [9] seek to raise awareness and engagement with security bugs.

The research program that this work describes is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2022-03027]. Cette recherche a été financée en partie par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG). The research is also supported in part by Alberta Innovates and the University of Calgary, and by a gift from Intel Corporation. This work does not in any way constitute an Intel endorsement of a product or supplier.

Even so, there remains a gap between the accessibility of (hardware) cybersecurity expertise and the need for secure design. As hardware forms the foundation of a secure system, we need tools and techniques that support computer hardware engineers to improve trust and help them address security concerns. *How do we choose what security features to implement? How do we check our designs, even if designs are not yet complete? How do we build trust, even if designers are not security experts?* Such questions are not easily solved.

Towards the goal of building trust in digital systems, this paper highlights a vision for tools and techniques to enhance the security of digital hardware in earlier stages of the digital design process and some of the progress our team has made in this quest for improving security. We discuss the challenges that design teams face and explore some recent literature on understanding, identifying, and mitigating hardware security weaknesses as early as possible. With the emphatic growth in open-source hardware design, there is an opportunity to learn from and contribute to open-source ecosystems in pushing our understanding and handling of security challenges.

The rest of this paper is as follows. Section II provides background on the area of hardware security bugs and the motivation for wanting tools and techniques to support things earlier in design. In Section III, we discuss some of our recent work in the area and highlight some open challenges, and present some related work in Section IV. Section V concludes.

II. Background and Motivation

Hardware security is a wide and varied field, and our understanding of risks continues to evolve. There are potential security issues such as threats in the supply chain (especially given globalized production [10]), the potential for malicious modifications [11], or even perhaps unintentional bugs [4]. Others include issues that can manifest physically (e.g., side-channels [12]). There are many potential solutions for different security challenges such as new mechanisms [13] and the domain features back-and-forth developments, likened to "cat-and-mouse" games for attacks and defenses (e.g., in logic locking [14]). Naturally, to build trusted systems, we want

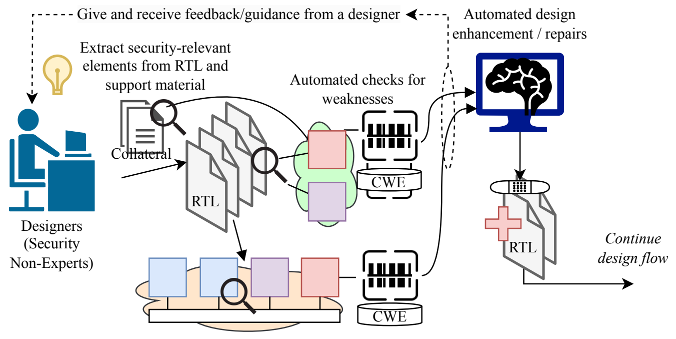
Give and receive feedback/guidance from a designer
Automated design enhancement / repairs
Extract security-relevant elements from RTL and support material
Automated checks for weaknesses
Collateral
RTL
CWE
CWE
RTL
Designers (Security Non-Experts)
Continue design flow

Fig. 1. A vision for tools and techniques to help with building trust in early stages of design

to choose and combine the "best" security solutions available. However, what is "best" is determined on a case-by-case basis; designers might need to favor one design metric over another, such as keeping area overhead low, maximizing performance, or improving usability—sometimes at the cost of security.

Given the plethora of options available, security-driven design remains largely a human-in-the-loop endeavor. Choosing how best to go about improving trust requires creativity and value-based judgment. Even the process of deciding what is important (i.e., identifying *assets*) is subjective and requires some level of security expertise and a handle on designer intent [15]. However, when there are humans involved, there is the risk of unintentional mistakes – in other words, there is a potential for *bugs*, a design defect that might result in unintended behavior (noting, of course, that designer intent is often imperfect, incomplete, or implicit [16]). Design bugs can appear throughout the design lifecycle, ranging from improperly captured (or even defined) specifications to literal typographical errors in the code.

How can we effectively find and deal with bugs? This question motivates the "quest" outlined in this paper. While one can (and should) think about security throughout the lifecycle, our team's work is especially concerned with what is possible at *early* stages of design, i.e., during RTL implementation and earlier. *Why?* If we find bugs early in design, we can make lower-cost changes to reduce risk and avoid calamities should an exploitable vulnerability "escape" in a final product. Working at RTL and earlier improves the likelihood that we have at our disposal more markers of designer intent, ranging from code comments, signal names, specification documents, and such – we can work **with** designers to give guidance (and hopefully improve security awareness) as well as receive guidance (for example, on the validity of an identified weakness). How early do we envision? As early as possible, potentially even before we finalize security objectives and threat models – this entails adaptable and flexible analyses. As of today, there is no panacea when it comes to identifying security weaknesses [4], [5]. Fig. 1 illustrates a vision for tools and techniques that can assist designers in building trust.

Recent efforts like the hardware CWEs [6] provide a means to categorize and reason over security weaknesses in a common framework, but how one can use the framework, e.g., for implementing automated detection tools or informing designers, remains an open challenge. Another industry-led effort, the SA-EDI standard [7] (now transitioned to an in-progress IEEE proposed standard, IEEE P3164), captures the idea that security is a collective responsibility by aiming for a standardized format for security-related collateral associated with an IP – a system *integrator* can use the information to make an "informed decision at the time of IP integration" and lead to actions like "implement[ing] mitigations" or even deciding that the risks are out of scope" [7]. To the best of our knowledge, uptake of the standard has been low, perhaps partly due to the onerousness of populating the fields required (by hand). *Can we automate things? If so, how much automation should we have?* In light of everything discussed so far, some of the open questions include (and are not limited to):

- **Bugs, bugs, bugs**: How can we detect different kinds of security weaknesses and bugs? Are there some types of bugs that are inherently more (or less) amenable to certain types of analyses?
- **Humans-in-the-loop**: How can we maximize human expertise and intent in building trust? Is there a space between the extremes of having everything manually crafted and fully automated? How do we support the design processes currently used by design teams?

III. Recent Directions

Given our discussion of the motivation for our work, this section provides an overview of some of the directions we have been pursuing, emphasizing what we consider to be some open challenges. We also provide interested readers with a non-extensive overview of related work.

A. Static Analysis

In pursuing security analysis at earlier design stages, we looked at *static analysis* [5]. Static analysis focuses primarily (if not solely) on source code, thus obviating the need for other design collateral (such as testbenches and a functioning simulation environment). In some cases, analyzing *incomplete* code is even possible. In that work, we identified five CWEs, *CWE-1234*: Hardware Internal or Debug Modes Allow Override of Locks, *CWE-1271*: Uninitialized Value on Reset for Registers with Security Settings, *CWE-1245*: Improper Finite State Machines (finite state machines (FSMs)) in Hardware Logic, *CWE-1280*: Access Control Check Implemented After Asset is Accessed and *CWE-1262*: Improper Access Control for Register Interface, as amenable to pattern recognition in a "context-less" fashion. This means that we could craft scanners to identify potential instances of these weaknesses without requiring additional context from a designer, such as specific assets or design intent, as these weaknesses could be considered "general".

`CWEAT` [5] showed that certain CWEs can be detected during the early RTL Implementation, where we were able to highlight 180 instances of potential weaknesses, reducing the search space for manual checking. However, while promising, several challenges emerged – as with any imperfect detection system, there is the risk of false positives that can distract, confuse, or burden a human operator. As such, more work is needed to reduce the "noise." We did, however, find that similar scanning could be used in cases where the RTL code is more structured, such as that generated through HLS [17]. In investigating the CWEs, we found that several entries are quite "broad" – given that we only looked at five of (as of writing) 108 hardware-relevant CWEs, we surmised that the remaining require much more *context*. In other words, scanners need to incorporate design- or project-specific information to guide the identification of areas of concern. As of now, we lack robust solutions for context-inclusive scanning.

B. Large Language Models (LLMs)

The recent emergence of large language models (LLMs) has inspired a flurry of research activity (readers might find the recent survey [18] useful). Keeping the focus on *source code*, we have investigated LLMs for detecting potential security bugs [19], repairing bugs [20], and assertion generation [21]. In many ways, these works could be considered a type of "static analysis," as previously discussed, at least when using LLMs without "feedback" (such as from a simulation or formal verification tool).

Our proof-of-concept implementations[1] show that there is the potential for using models to help designers, although sometimes with numerous requests to the LLMs. We speculate that their usefulness will increase over time, at least while new LLM models continue to exhibit increased performance in general. However, like other LLM-centric solutions, LLM shortcomings remain, such as hallucination or mixed-quality in their outputs (such as security [22]). Prompt engineering and verification/evaluation of LLM outputs are some of the challenges we continue to face. As of now, we postulate that the more "interactive" nature of LLMs (e.g., for "chat") can provide opportunities to more directly inform designers or act as an interface for designer guidance. For example, our recent work investigated using LLMs to explain EDA tool error messages to novice users [18].

C. Learning from Open Source Processes

With the rise in open-source hardware projects (e.g., OpenTitan [23]) and heterogeneous SoCs, there is an opportunity to learn from hardware development. This is especially pertinent from an academic point of view, given that access to internal details of commercial designs is understandably unlikely. Having access to designs written in HDLs is very useful for experimentation and analysis, and efforts like Hack@DAC (part of the Hack The Silicon series [9]), TrustHub [24], CAD4Security [25] and CAD for Assurance [26] have enabled scores of research, including ours. Even so, the quantity of open-source hardware remains orders of magnitude less than software, with datasets of known bugs even more scarce.

What we do have available, however, can be very interesting. In our recent work [27], we have begun to look beyond the RTL code and into the *discussions* accompanying digital design, where human developers identify potential bugs, discuss them, and implement fixes. By looking at behaviors, such as the nature of identified issues, the magnitude of code changes between commits, and the levels of discussion, we can start to build a more holistic view of the development process. When we manually examined OpenTitan, we found that 53% of the bugs identified during its development (in the period that we examined) had potential security implications and that 55% of all bug fixes changed only a single file. We think that more consideration of open-source projects can reveal new insights, and as more projects emerge and activity increases (and we hope they do), this opportunity will grow.

[1]e.g., https://zenodo.org/records/10416865

IV. Selected Related Work

Finding hardware security bugs in the design stage at RTL requires considerable security expertise, especially for manual analysis [3], [4], [28]. There exist some specialized approaches that require experts to devise information flow properties for formal verification and simulation [29]. Security invariants are mined in [30], and testing approaches like concolic testing (e.g., [31]) or fuzzing (e.g., [32]) are emerging. There are a few approaches for security analysis during RTL design, e.g., the construction and analysis of hyperflow graphs [33], and progress toward the automation of various security tasks, such as asset identification [34] and security property reuse [35]. Notably, few works attempt to deal with security feedback as you go; the automated verification environment for HDLs is far less mature compared with the state-of-the-art for higher-level computer programming (which has several security-focused static analysis tools—e.g., nearly 100 listed on OWASP [2]). Several tools provide linting capabilities for RTL (e.g., [36]), but these do not yet focus on highlighting security weaknesses. As previously discussed, LLMs provide opportunities for early-stage security assistance [37], such as property generation [38].

Related literature deals with hardware Trojans (HTs) detection [11]; if we consider bugs to be "unintentional" artifacts, HTs are complementary "intentional" malicious insertions. There are techniques for HT detection (e.g., [39]–[41]) that attempt to localize suspicious design parts using heuristics or ML techniques. While they can serve as a starting point for security big detection, they do not usually apply to earlier design stages or propose repair techniques for security weaknesses. While research into automatic program repair in software engineering is mature [42], similar efforts for hardware design lag, but recent work is promising [20], [43], [44].

V. Conclusions

We gave insights into our vision for tools and techniques to enhance the security of digital hardware in earlier stages of the digital design process. We discussed recent literature on understanding, identifying, and mitigating hardware security weaknesses as early as possible and outlined some ongoing challenges and opportunities, especially those that continue to emerge with open-source hardware development.

References

[1] V. Dorsey and C. Morhardt, "Intel Security Development Lifecycle," Intel Corporation, Tech. Rep., 2020. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2020/10/sdl-2020-whitepaper.pdf

[2] OWASP, "Source Code Analysis Tools | OWASP Foundation." [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[3] M. M. Bidmeshki *et al.*, "Hunting Security Bugs in SoC Designs: Lessons Learned," *IEEE Design & Test*, vol. 38, no. 1, pp. 22–29, Feb. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9154739/

[4] G. Dessouky *et al.*, "HardFails: Insights into Software-Exploitable hardware bugs," in *USENIX Security Symp.* USENIX Association, Aug. 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[5] B. Ahmad *et al.*, "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design," in *IEEE/ACM Int. Conf. on CAD*, Dec 2022, p. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3508352.3549369

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

[7] Accellera Systems Initiative, "Security Annotation for Electronic Design Integration Standard," Jul. 2021. [Online]. Available: https://www.accellera.org/images/downloads/standards/Accellera_SA-EDI_Standard_v10.pdf

[8] IEEE P3164 Working Group, "P3164 Standard for Security Annotation for Electronic Design Integration." [Online]. Available: https://standards.ieee.org/ieee/3164/11106/

[9] "Home - Hack The Silicon," 2024. [Online]. Available: https://hackthesilicon.com/

[10] M. Rostami, F. Koushanfar, and R. Karri, "A Primer on Hardware Security: Models, Methods, and Metrics," *Proc. IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug. 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6860363/

[11] K. Xiao *et al.*, "Hardware Trojans: Lessons Learned after One Decade of Research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 6:1–6:23, May 2016. [Online]. Available: http://doi.org/10.1145/2906147

[12] F.-X. Standaert, "Introduction to Side-Channel Attacks," in *Secure Integrated Circuits and Systems*, I. M. Verbauwhede, Ed. Boston, MA: Springer US, 2010, pp. 27–42. [Online]. Available: https://doi.org/10.1007/978-0-387-71829-3_2

[13] B. Tan, "Challenges and Opportunities for Hardware-Assisted Security Improvements in the Field," in *2022 23rd Int. Symp. on Quality Electronic Design (ISQED)*, Apr. 2022, pp. 90–95, iSSN: 1948-3295. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9806254

[14] A. Chakraborty *et al.*, "Keynote: A Disquisition on Logic Locking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, pp. 1–1, 2019.

[15] IEEE P3164 Working Group, "Asset Identification for Electronic Design IP," *Asset Identification for Electronic Design IP*, pp. 1–26, Apr. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10496567

[16] D. G. Widder and C. L. Goues, "What is a "bug"? subjectivity, epistemic power, and implications for software research," Feb. 2024, arXiv:2402.08165 [cs]. [Online]. Available: http://arxiv.org/abs/2402.08165

[17] L. Collini *et al.*, "Using Static Analysis for Enhancing HLS Security," *IEEE Embedded Systems Letters*, pp. 1–1, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10308602/

[18] S. Qiu, B. Tan, and H. Pearce, "Explaining EDA synthesis errors with LLMs," Apr. 2024, arXiv:2404.07235 [cs]. Accepted to appear at the 1st IEEE Int. Workshop on LLM-Aided Design (LAD'24). [Online]. Available: http://arxiv.org/abs/2404.07235

[19] B. Ahmad *et al.*, "FLAG: Finding Line Anomalies (in code) with Generative AI," Jun. 2023, arXiv:2306.12643 [cs]. [Online]. Available: http://arxiv.org/abs/2306.12643

[20] ——, "On Hardware Security Bug Code Fixes By Prompting Large Language Models," *IEEE Trans. Inf. Forensics Security*, pp. 1–1, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10462177

[21] R. Kande *et al.*, "(Security) Assertions by Large Language Models," *IEEE Trans. Inf. Forensics Security*, 2024, preprint: https://arxiv.org/abs/2306.14027. [Online]. Available: https://ieeexplore.ieee.org/document/10458667

[22] H. Pearce *et al.*, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *2022 IEEE Symp. on Security and Privacy (SP)*, May 2022, pp. 754–768.

[23] lowRISC, "Opentitan," 2024, last accessed on 05/05/2024. [Online]. Available: https://github.com/lowRISC/opentitan

[24] "Trust-Hub.org." [Online]. Available: https://trust-hub.org/#/home

[25] "CAD4Security – CAD4Security." [Online]. Available: http://cad4security.org/

[26] "CAD for Assurance – CAD for Assurance of Electronic Systems." [Online]. Available: https://cadforassurance.org/

[27] J. Ah-kiow and B. Tan, "An Investigation of Hardware Security Bug Characteristics in Open-Source Projects," Feb. 2024, arXiv:2402.00684 [cs]. [Online]. Available: http://arxiv.org/abs/2402.00684

[28] M. Fischer *et al.*, "Hardware Penetration Testing Knocks Your SoCs Off," *IEEE Design Test*, vol. 38, no. 1, pp. 14–21, Feb. 2021.

[29] "Cycuity | Security assurance starts here." [Online]. Available: https://cycuity.com/

[30] R. Zhang *et al.*, "End-to-End Automated Exploit Generation for Validating the Security of Processor Designs," in *2018 51st Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 815–827. [Online]. Available: https://ieeexplore.ieee.org/document/8574588/

[31] X. Meng *et al.*, "RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, pp. 1–1, 2021.

[32] K. Laeufer *et al.*, "RFUZZ: coverage-directed fuzz testing of RTL on FPGAs," in *Proc. the Int. Conf. on Computer-Aided Design*. San Diego California: ACM, Nov. 2018, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3240765.3240842

[33] A. Meza and R. Kastner, "Information Flow Coverage Metrics for Hardware Security Verification," Apr. 2023, arXiv:2304.08263 [cs]. [Online]. Available: http://arxiv.org/abs/2304.08263

[34] N. Farzana *et al.*, "SAIF: Automated Asset Identification for Security Verification at the Register Transfer Level," in *2021 IEEE 39th VLSI Test Symp. (VTS)*, Apr. 2021, pp. 1–7, iSSN: 2375-1053. [Online]. Available: https://ieeexplore.ieee.org/document/9441039

[35] R. Zhang and C. Sturton, "Transys: Leveraging Common Security Properties Across Hardware Designs," in *2020 IEEE Symp. on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1713–1727. [Online]. Available: https://ieeexplore.ieee.org/document/9152775/

[36] "Verilator User's Guide — Verilator 5.024 documentation." [Online]. Available: https://verilator.org/guide/latest/#

[37] Z. Wang *et al.*, "LLMs and the Future of Chip Design: Unveiling Security Risks and Building Trust," May 2024, arXiv:2405.07061 [cs]. [Online]. Available: http://arxiv.org/abs/2405.07061

[38] X. Meng *et al.*, "Unlocking Hardware Security Assurance: The Potential of LLMs," Aug. 2023, arXiv:2308.11042 [cs]. [Online]. Available: http://arxiv.org/abs/2308.11042

[39] T. Trippel *et al.*, "Bomberman: Defining and Defeating Hardware Ticking Timebombs at Design-time," in *2021 IEEE Symp. on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2021, pp. 970–986. [Online]. Available: https://ieeexplore.ieee.org/document/9519417/

[40] T. Han, Y. Wang, and P. Liu, "Hardware Trojans Detection at Register Transfer Level Based on Machine Learning," in *2019 IEEE Int. Symp. on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.

[41] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *Proc. the 2013 ACM SIGSAC Conf. on Computer & communications security*. ACM, Nov. 2013, pp. 697–708. [Online]. Available: https://doi.org/10.1145/2508859.2516654

[42] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, Jan. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8089448/

[43] H. Ahmad, Y. Huang, and W. Weimer, "Cirfix: automatically repairing defects in hardware design code," in *ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, Feb 2022, p. 990–1003. [Online]. Available: https://doi.org/10.1145/3503222.3507763

[44] K. Laeufer *et al.*, "RTL-Repair: Fast Symbolic Repair of Hardware Design Code," in *Proc. the 29th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Volume 3*. La Jolla CA USA: ACM, Apr. 2024, pp. 867–881. [Online]. Available: https://dl.acm.org/doi/10.1145/3620666.3651346