Toward a Trustable, Self-Hosting Computer System

Gabriel L. Somlo
CERT – SEI
Carnegie Mellon University
Pittsburgh, PA 15213
Email: glsomlo@cert.org

***Abstract*—Due to the extremely rapid growth of the computing and IT technology market, commercial hardware made for the civilian, consumer sector is increasingly (and inevitably) deployed in security-sensitive environments. With the growing threat of hardware Trojans and backdoors, an adversary could perpetrate a full system compromise, or privilege escalation attack, even if the software is presumed to be perfectly secure. We propose a method of *field stripping* a computer system by empirically proving an equivalence between the trustability of the fielded system on one hand, and its comprehensive set of sources (including those of all toolchains used in its construction) on the other. In the long run, we hope to facilitate comprehensive verification and validation of fielded computer systems from fully self-contained hardware+software sources, as a way of mitigating against the lack of control over (and visibility into) the hardware supply chain.**

I. Introduction

Hardware vulnerabilities have presented an increasingly dire security threat across the entire user population for the last several decades [1]. The gravity of the problem is compounded by the constantly growing length of microchip design, development, and fabrication supply chains, including outsourcing, the multinational nature of major chipmakers, and the global and highly mobile nature of the workforce they employ. Due to the extremely rapid growth of the microchip market over the last several decades, customers with enhanced security sensitivity (e.g., governments, militaries, and security agencies) are increasingly forced to use chips targeted at the civilian consumer market, and individually no longer have the market size that would put them in a position to demand adequate supply chain security assurances from their vendors [2]. As evidenced by both academic research and practical industry experience [3], [4], [5], [6], [7], [8], carefully planted hardware Trojans or backdoors may allow malicious attackers to completely take over a victim computer system, even if the software could, in theory, be presumed to be perfectly secure and free of bugs.

We propose to build trustable computer systems on top of Field Programmable Gate Arrays (FPGA), which, due to their generic nature, make it qualitatively harder to conceal intentional backdoors implemented in silicon. From there, we leverage the transparency and auditability of Free and Open Source (FOSS) hardware, software, and compiler toolchains to configure FPGAs to act as Linux-capable Systems-on-Chip (SoC) that are as trustworthy as the comprehensive sources used in both their specification and construction.

The remainder of this paper is laid out as follows: Section II presents a brief overview of the hardware development lifecycle, pointing out similarities and contrasts to software development. Section III examines hardware backdoor classification criteria relevant to the solution proposed in Section IV. A proof of concept implementation of our proposed solution, currently still undergoing development, is outlined in Section V. Considerations regarding the performance of our prototype are presented in Section VI. Finally, conclusions and plans for future work are discussed in Section VII.

II. Hardware vs. Software Development

Modern hardware is developed in a way that shares many similarities with software [9]. After an architectural design step, the desired behavior is expressed in a source hardware description language (HDL). HDLs (e.g., Verilog and VHDL) tend to be languages with strong functional and declarative characteristics. Source "code" is then *compiled*, resulting in either photolitographic masks for dedicated application-specific integrated circuits (ASICs), or configuration data (*bitstream*) for field-programmable gate arrays (FPGAs). Debugging and documentation are also stages shared with software development, as well as iterating through all stages multiple times until the desired behavior is accomplished.

DOI 10.1109/SPW50608.2020.00039

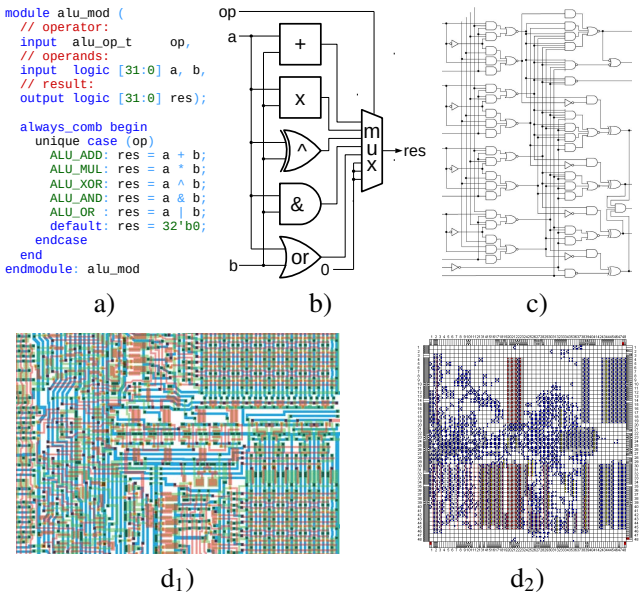

Fig. 1. Hardware Compilation Pipeline.

The main stages of a hardware compilation pipeline are shown in Fig. 1. First, the HDL source code (Fig. 1a) is passed through an *elaboration* stage that builds a graph of standard blocks (Fig. 1b) representing the specified design. Next, *logic synthesis* and *optimization* converts high-level blocks into an optimized graph consisting of basic logic gates (Fig. 1c).

From there, *technology mapping*, *placement*, and *routing* generate either a set of optimized masks, to be etched into dedicated silicon ASICs using an expensive and labor-intensive photolitography process (Fig. 1d$_1$), or bitstream to instruct an FPGA's configurable logic blocks (CLBs) and programmable interconnect elements on how to wire themselves together in order to "act out" the required design (Fig. 1d$_2$). When multiple authors' designs are linked together, they are referred to as either Hard (ASIC masks) or Soft (FPGA bitstream) intellectual property (IP) cores.

While both traditional CPUs and FPGAs are "programmable", the crucial difference is that the former execute a *sequential* stream of instruction opcodes read from RAM, while the latter receive their entire configuration at once. While a CPU's opcode sequence tells it what to *do*, the bitstream tells an FPGA what to *be*. FPGA bitstream resides in a special memory that is written once during configuration, and remains unmodified for as long as the configured device is in operation.

FPGAs are frequently used to debug, test, and validate a hardware design before making the very large financial and time commitments required to fabricate dedicated ASICs at high volume. FPGAs also offer good value when the projected production volume is not expected to cover the sunk costs of ASIC fabrication. Generally, hardware designs deployed on FPGAs are larger, slower, and more power-hungry than equivalent optimized ASICs. However, in Section IV we argue that FPGAs also offer valuable additional security assurance opportunities.

III. Overview of Hardware Vulnerabilities

News outlets and IT trade publications often refer to malicious *firmware* [10], [11] as "hardware" attacks, primarily due to where they are stored (typically, flash on a computer's motherboard), and to their persistent nature (capability to survive a hard disk wipe and OS reinstall). To be precise, from this paper's perspective, those would be considered *software* attacks, as the malicious behavior is still codified in the form of CPU instructions. The hardware vulnerabilities discussed below manifest in the behavior of the CPU (and associated chip set) itself, and occur at or below the CPU's instruction set architecture (ISA) in terms of the abstraction layers affected.

Several classification criteria for hardware vulnerabilities have emerged from industry and academic hardware security research [12], [13], [14], [15], [16]. Without loss of generality, we find the following simplified breakdown relevant:

- **Insertion Method**: Development stage (see Fig. 1) where the vulnerability is introduced:
  - *Design, Implementation*: Vulnerabilities present in source code, introduced accidentally or maliciously; e.g., Spectre [7] and Meltdown [8].
  - *Toolchain*: A compromised HDL compiler toolchain may generate maliciously misbehaving ASIC masks or FPGA bitstream from perfectly clean and innocent sources [5].
  - *Fabrication*: Malicious ASIC fabrication plants may reverse engineer a customer's masks, and carefully alter them to insert backdoors or Trojans into the microchips being produced. Examples include subtly altering a random number generator to weaken encryption [3], or allowing a CPU's privilege mode flag to be flipped during the execution of a prearranged sequence of unprivileged instructions [6].
- **Severity**: Level of damage incurred by the system's owner during an attack:
  - *Denial of Service (DoS)*: An attacker can destroy the system, or otherwise degrade its per-

formance, making it unavailable to its owner; e.g., a "self-destruct" switch or timer.
- *Privilege Escalation*: Any method facilitating an attacker's unauthorized access to a system or its data; includes everything from side channels for data exfiltration to a remote take-over.

First, we observe that ASIC fabrication attacks (capable of inserting both DoS and privilege escalation backdoors) can not be prevented without ownership and control of the chip foundry, an increasingly difficult and unlikely proposition. Proposed mitigation attempts include: visually inspecting a microscopic die-shot of the silicon [17], obfuscating the chip's interaction with the external world [18], implementing "voting" across multiple redundant chips exposing the same interface [19], and split manufacturing of different layers on the die [20]. However, most of these techniques have met with limited success, due to factors of cost, reliability, and the ease of hiding the small number of components required by modern exploits (as low as 20) among the *billions* of components contained in modern microchips.

A second observation is that privilege escalation vulnerabilities are *qualitatively* more severe than DoS attacks. Having one's computer die during a DoS attack may range from inconvenient to dangerous; however, a privilege escalation attack will silently make a system turn on its owner, betraying them to the adversary while appearing to operate correctly!

IV. Improving Hardware Assurance for Sensitive Applications

To work around the lack of control over chip foundries, we propose that security sensitive hardware deployments be limited to FPGA-based *soft* IP cores. Since doing so limits a foundry's role to fabricating FPGAs, it removes the attacker's ability to derive useful details about the final design, which would be necessary to mount a successful privilege escalation hardware attack [21]. The malicious foundry would have no way of predicting e.g., where a soft-IP CPU's privilege mode flag would be laid out on an FPGA, and would be relegated to at worst inserting DoS hardware attacks. Since the FPGA consists of a regular grid of identical CLBs, even the components of a DoS attack would be much easier to detect visually using imaging techniques [17], compared to the much more diverse and irregular structure of ASICs.

By way of analogy, if the 20-some transistors and one capacitor used to implement a privilege escalation attack against an ASIC CPU such as the A2 Trojan [6] can be compared to a "needle in a haystack", perpetrating a similar attack against an FPGA-based soft-IP CPU would require *many* needles to be scattered throughout the entire (FPGA) haystack just in case, and would thus be much harder to successfully conceal.

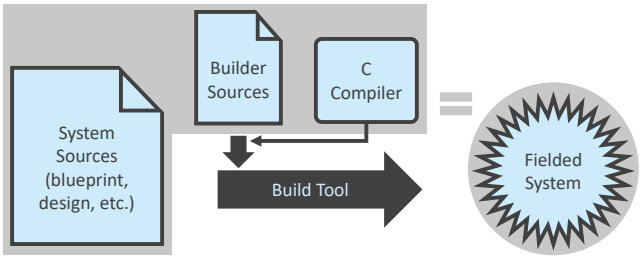

Fig. 2. Trust Anchor for Fielded Computer Systems.

Having thus mitigated against malicious foundry attacks, the actual degree to which we can trust a fielded computing system depends not only on the trustability of the cumulative source code to all its hardware and software, but also on the trustability of the toolchains used to compile and build those sources. A tool's degree of trustability depends on its own sources and build environment – a recursive process illustrated in Fig. 2, which terminates with the question of whether we can trust our C compiler (Ken Thompson's "Trusting Trust" thought experiment [22]).

Diverse double compilation (DDC) [23] has been proposed as a method to detect whether a C compiler may have been subjected to a malicious "Trusting Trust" compromise. The method uses a second, independent C compiler that is unlikely to have been compromised by the same type of attack. The suspect compiler's sources are repeatedly built using both its own binary, and the second, independent compiler's binary, until a bit-by-bit comparison of the resulting outputs can be used to ascertain the absence of malicious compromise, with a fairly high degree of confidence. From here, we can bootstrap a trustable system via the following steps:

1) On a host system, using DDC, build clean C [cross]-compilers for both the host and target environments
2) [Cross-]compile the HDL toolchain for both host and target systems
3) Build target system hardware as FPGA bitstream (soft IP cores)
4) Cross-compile OS (kernel, system libraries and utilities) for target system
5) Boot target OS on FPGA configured with the generated bitstream

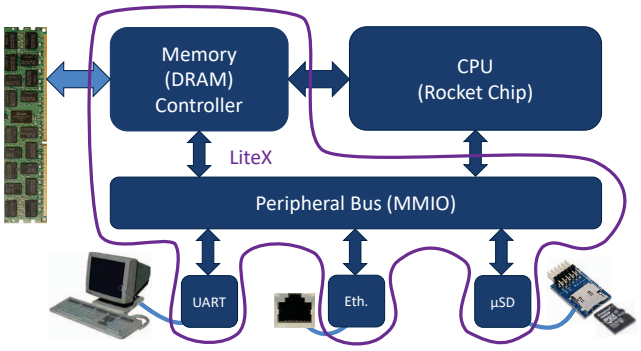

Fig. 3. Trustable Computer with LiteX and Rocket Chip

Once the target system is booted up, it can operate as a self-hosting "clean room" environment, capable of rebuilding and supporting further development of any of its hardware or software components, without relying on external tools, services, or infrastructure. The system is *as trustable as* the cumulative sources to the C compiler, HDL toolchain, and target OS components (kernel, system libraries, and utilities).

V. Proof of Concept: Bootstrapping a Free, Self-Hosting Linux Computer

A combination of FPGA based soft-IP hardware and fully buildable sources to *everything* (i.e., hardware, software, and toolchains) should be a key addition to customers' IT acquisition requirements for security sensitive applications. First, however, we demonstrate the feasibility of this approach by building a usable self-hosting, cleanroom prototype system from scratch, relying exclusively on Free and Open Source (FOSS) components:

- For the system's CPU, we selected the open RISC-V architecture [24], [25]. Since our ultimate goal is to build a fully Linux-capable computer, and since most Linux distributions (e.g., Fedora [26], Debian [27]) have standardized on the 64-bit "rv64gc" feature set, we ended up using Rocket Chip [28], an open source, reference implementation of the RISC-V specification.
- For the chipset surrounding the CPU, and providing essential functionality such as a system bus, memory controller, and peripheral interfaces (serial console or UART, Ethernet, and µSDcard), we selected the LiteX SoC builder [29], [30]. We made significant upstream changes adding Rocket Chip support, to the point where a LiteX+Rocket computer (see Fig. 3) can boot a standard, unmodified upstream Linux kernel all the way into a fully functional user shell provided by the BusyBox [31] utility.
- To build the HDL sources of the combined LiteX+Rocket system into functional FPGA bitstream, we use the FOSS Yosys/Nextpnr toolchain [32], [33], [34], [35].
- At the time of this writing, the only FPGAs large enough to fit a Litex+Rocket bitstream supported by the open source toolchain are Lattice's ECP5 series, although a project is underway [36] (and has made significant progress) toward adding open source toolchain support to Xilinx's Artix7 series FPGAs.
- The majority of commercially available FPGA development boards built around Lattice ECP5 (or Xilinx Artix7) chips include under 512MB of RAM (the typical amount of RAM included tends to be 128MB). While sufficient for booting Linux into a BusyBox shell, experiments conducted on the x86_64 host development platform suggest that, to build Litex+Rocket HDL into FPGA bitstream, the toolchain processes will utilize a resident memory footprint in the range of 1.3GB. Currently we use a custom-designed, open source development board designed around the largest available Lattice ECP5 FPGA [37], which comes equipped with 1GB of RAM memory. Optimization efforts are currently underway to reduce the run-time memory requirements of Yosys [33], which will eventually allow a LiteX+Rocket HDL-to-bitstream build to fit into a sub-1GB resident memory footprint.

Once built [38], the system described above runs at a clock cycle of up to 65MHz, and is capable of reliably booting a standard, 64-bit RISC-V Linux kernel. Terminal I/O is available over the serial UART, and, from the BusyBox userspace shell, we are able to bring up the included Ethernet interface, ssh into and out of the system, and mount a remote NFS filesystem. Complete and up-to-date build instructions for the system are available online at http://www.contrib.andrew.cmu.edu/~somlo/BTCP.

While the Rocket Chip offers the option of a hardware floting-point unit (FPU), the largest Lattice ECP5 FPGA does not have enough capacity to accomodate it. Therefore, floating-point opcodes unsupported by the available CPU configuration will trap, and are subsequently handled by a software FPU emulator provided by the BBL machine-mode hypervisor module [39].

	CPU	MHz	CoreMark	Linpack (KFLOPS)		nbench P5-90		nbench K6-223			Notes
				Single	Double	Int	Float	Mem	Int	Float	
1.	P5	90	-	-	-	1.00	1.00	-	-	-	nbench calibration, P5-90
2.	K6	233	-	-	-	-	-	1.00	1.00	1.00	nbench calibration, K6-233
3.	Xeon	2400	12489.07	1679090	1618198	109.59	112.60	34.36	23.05	62.46	native 2.4GHz Xeon E5645
4.	rv64gc	-	1468.86	21520	20964	13.38	1.67	2.80	3.81	0.93	Qemu 2.4GHz Xeon E5645
5.	P5	133	282.63	13227	8923	1.77	0.90	0.35	0.53	0.50	Dell Dimension GsMT5133
6.	Rocket	65	47.45	48	31	0.31	.003	.077	.079	.001	LiteX+Rocket (no FPU)
7.	Rocket	60	103.89	84	79	0.47	.003	0.11	0.12	.001	LiteX+Rocket (FPU)
8.	Rocket	50	103.58	5709	4492	0.92	0.67	0.19	0.26	0.37	lowRISC (FPU, max.cache)

TABLE I
BENCHMARKING THE LITEX+ROCKET SOFT-IP COMPUTER.

VI. PERFORMANCE ASSESSMENT

As a soft-IP FPGA implementation, we expected our computer to have relatively low performance. A 65MHz Linux-capable CPU inevitably invokes memories of mid-1990s Intel 486 and first-generation Pentium processors. While it is well understood that benchmarks are at best an imprecise way of measuring a system's performance, we decided to use a set of popular benchmarks to measure and compare the performance of our Litex+Rocket computer (row #6 in Table I) against a list of reference machines:

- Contemporary hardware (2.4GHz Xeon E5645) running natively (row #3), and also emulating a 64bit RISC-V machine with QEMU (row #4),
- Pentium machine (Dell Dimension GsMT5133) from the mid-1990s, running at 133MHz (row #5),
- Rocket+LiteX computer with hardware FPU, compiled for a Xilinx Artix-7 FPGA on a Nexys 4 DDR board, using proprietary vendor tools (row #7),
- Rocket Chip with FPU and maximized L1 cache (using up all available spare capacity of the Artix-7 FPGA) as part of the lowRISC SoC project [40], [41] (row #8).

The following list of benchmarks was used:

- CoreMark [42], [43], a test designed to measure the perfomance of a processor's core features. The test produces a single numerical result intended to facilitate comparison between CPU cores.
- Linpack [44], a test designed to measure a CPU's floating point performance. The test uses matrix multiplication, and produces Floating Point Operations per Second (FLOPS) measurements for both Single and Double precision multiplications. When used on a FPU-less CPU core, these tests are an indirect measure of integer arithmetic performance, as the FPU is emulated in software.
- nbench (née BYTEMark) [45], designed to measure a CPU core's integer, floating point, and memory access performance. Results are presented as the ratio of the tested CPU's performance to that of a *reference* CPU (either a 90MHz original Pentium, or a 233MHz AMD K6). This is illustrated by the "calibration" rows (#1, #2) in Table I, where the hypothetical P5-90 and K6-233 CPUs would obtain result values of 1.0 in their respective columns.

As expected, our LiteX+Rocket system built with the FOSS Yosys/nextpnr toolchain for the ECP5-based TrellisBoard (row #6) obtained the lowest performance. The primary explanation is the lack of an FPU, and the default L1 cache size included with the Rocket Chip. Even the largest ECP5 FPGA is a relatively small chip, and does not have enough spare capacity for a Rocket Chip configured to include a "hardware" FPU, or enough spare Block RAM (BRAM) elements to significantly increase the default Rocket L1 cache.

We also built Litex+Rocket for a Xilinx Artix-7 FPGA (on a Nexys 4 DDR development board, using the vendor's proprietary HDL-to-bitstream toolchain), which does have the spare capacity to support the inclusion of Rocket's built-in FPU. Without an FPU, the ECP5 and Artix-7 builds exhibited no significant differences in benchmark results. Adding the FPU (row #7), we observe that CoreMark and Linpack values are roughly double those in row #6. Results for nbench are inconclusive,

due to the noisiness of the "ratio" method of reporting in extreme low performance conditions, compounded by the narrower DRAM memory bus width available on the Nexys board.

We also tested lowRISC [40], [41] (row #8), an alternative SoC built around the Rocket Chip, specifically targeted at the Artix-7 FPGA on a Nexys 4 DDR board. The project activates the Rocket Chip's FPU option, and also utilizes all available BRAM elements on the FPGA to maximize the L1 cache available to the Rocket CPU core. On the downside, lowRISC relies exclusively on the proprietary vendor toolchain, and on some of the included proprietary soft-IP library blocks (e.g., the "mig7series" DRAM controller). The benchmark results are within range of those of an original Pentium chip running at 133MHz (row #5), which is encouraging: with the projected availability of Yosys/nextpnr support for Artix-7 FPGAs, we should soon be able to build trustable, self-hosting versions of LiteX+Rocket with hardware FPU and a large L1 cache, with performance similar to that of lowRISC and the original Intel Pentium.

VII. Conclusions and Future Work

We have demonstrated the feasibility of building a soft-IP, FPGA-based computer system that mitigates against hardware privilege escalation exploits. The system is currently capable of running an unmodified upstream 64-bit RISC-V Linux kernel at a clock frequency of 65MHz, on a development board equipped with a Lattice ECP5 FPGA and 1GB of RAM. The system's current performance, as measured by a set of popular benchmarks, is within less than an order of magnitude of the original Intel Pentium CPU running at approximately the same clock frequency.

While these results are highly encouraging, much work still remains. Currently in progress is an effort to integrate LiteX's μSDcard support into our system, which would enable it to boot a full-fledged existing Linux distribution (e.g., Fedora or Debian). With a "real" Linux distro, we gain the availability of native builds of the Yosys/nextpnr toolchain, and the system could then be used to rebuild its *own* FPGA bitstream, achieving our ultimate goal of *self-hosting*.

Next, there is still the "big picture" problem of empirically proving equivalence of trustability between the collected sources to the system and its build tools on one hand, and the actual fielded system on the other, starting with DDC. This problem is further complicated by the fact that the HDL sources to Litex and the Rocket Chip are not actually written in Verilog, ready to be digested by the Yosys compiler. Instead, Rocket is written in Chisel, which in turn depends on Scala (and ultimately Java), while LiteX is written in Migen, which is in turn written in Python. Fortunately, the complexity this adds to the problem is only quantitative (more sources to build on top of each other), rather than qualitative (everything ultimately still boils down to having a clean C compiler).

Our proposed method of building trustable computing systems prevents insertion of hardware privilege escalation exploits, and facilitates detection of hardware DoS vulnerabilities. Once widely adopted, this method will make it easier to build systems with increased security assurance levels, for both large, "state actor" organizations and for private individuals concerned about retaining control over their computing devices.

Since our method is predicated on the use of soft-IP FPGA based hardware, the resulting computer systems will have relatively low performance compared to high-end systems built with optimized ASICs. In other words, the workloads for which our trustable computers will be suited tend toward embedded systems, communications, industrial control, and lightweight personal computing – as opposed to HPC or datacenter deployments.

Finally, the availability of a fully self-hosting FOSS hardware+software computer system will hopefully spur further efforts and developments in formal verification and validation of the entire system as a cohesive unit. Sources can be studied and analyzed (manually, but more likely using automated tools), knowing there is a guarantee that trustability of the analyzed sources will be fully equivalent to the trustability of the system deployed in the field.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM20-0254

References

[1] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008. [Online]. Available: http://dx.doi.org/10.1109/MSPEC.2008.4505310

[2] D. Chesebrough, "Trusted microelectronics: A critical defense need," *National Defense: NDIA's Business & Technology Magazine*, Oct 2017. [Online]. Available: http://www.nationaldefensemagazine.org/articles/2017/10/31/trusted-microelectronics-a-critical-defense-need

[3] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'13. Springer-Verlag, 2013, pp. 197–214.

[4] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, ser. LEET'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 5:1–5:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387709.1387714

[5] C. Krieg, C. Wolf, and A. Jantsch, "Malicious LUT: A stealthy FPGA trojan injected and triggered by the design flow," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD'16. New York, NY, USA: ACM, 2016, pp. 43:1–43:8. [Online]. Available: http://doi.acm.org/10.1145/2966986.2967054

[6] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, ser. SP'16, 2016, pp. 18–37.

[7] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[8] M. Lipp *et al.*, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[9] E. Klingman, "Fpga programming step by step," *Embedded Systems*, Mar. 2004. [Online]. Available: https://www.embedded.com/print/4006429

[10] C. Domas, "The memory sinkhole - unleashing an x86 design flaw allowing universal privilege escalation." Las Vegas, NV, USA: Black Hat, 2015. [Online]. Available: https://www.blackhat.com/us-15/briefings.html#christopher-domas

[11] D. Oleksiuk, "Building reliable smm backdoor for UEFI based platforms," http://blog.cr4.sh/2015/07/building-reliable-smm-backdoor-for-uefi.html, 2015.

[12] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, ser. HLDVT'09, Nov 2009, pp. 166–171.

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

[14] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[15] M. Beaumont, B. Hopkins, and T. Newby, "Hardware trojans - prevention, detection, countermeasures," Defence Science and Technology Organisation, Edinburgh, Australia, Tech. Rep., 2011.

[16] S. Bhasin and F. Regazzoni, "A survey on hardware trojan detection techniques," in *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 2021–2024. [Online]. Available: http://ieeexplore.ieee.org/ielx7/7152138/7168553/07169073.pdf

[17] R. Torrance and D. James, "The state-of-the-art in IC reverse engineering," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES'09, vol. 5747. Springer-Verlag, Sep 2009, pp. 363–381.

[18] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP'11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 49–63. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.27

[19] H. A. Amin, Y. Alkabani, and G. M. Selim, "System-level protection and hardware trojan detection using weighted voting," *Journal of Advanced Research*, vol. 5, no. 4, pp. 499–505, 2014, cyber Security. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2090123213001446

[20] S. Mitra, H. S. P. Wong, and S. Wong, "The trojan-proof chip," *IEEE Spectrum*, vol. 52, no. 2, pp. 46–51, Feb 2015.

[21] S. Trimberger and J. Moore, "FPGA security: Motivations, features, and applications," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1248–1265, 2014. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2014.2331672

[22] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984. [Online]. Available: http://doi.acm.org/10.1145/358198.358210

[23] D. A. Wheeler, "Fully countering trusting trust through diverse double-compiling," Ph.D. dissertation, George Mason University, Fairfax, VA, 2009.

[24] "RISC-V: The free and open RISC ISA," http://riscv.org, The RISC-V Foundation, accessed: 2018-05-30.

[25] K. Asanovic *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[26] "Fedora/RISC-V," http://fedoraproject.org/wiki/Architectures/RISC-V[/FPGA], Fedora Project, accessed: 2018-05-30.

[27] "Debian/RISC-V," https://wiki.debian.org/RISC-V, Debian Project, accessed: 2018-05-30.

[28] "Rocket chip generator," https://github.com/chipsalliance/rocket-chip, CHIPS Alliance, accessed: 2020-01-20.

[29] F. Kermarrec, S. Bourdeauducq, J.-C. Le Lann, and H. Badier, "Litex: an open-source soc builder and librarybased on migen python dsl," in *Proceedings of the 2019 Workshop on Open Source Design Automation*, ser. OSDA'19, 2019.

[30] "LiteX SoC builder," https://github.com/enjoy-digital/litex, Enjoy Digital, accessed: 2020-01-20.

[31] "Busybox – the swiss army knife of embedded linux," https://busybox.net, accessed: 2020-01-19.

[32] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: an open source framework from verilog to bitstream for commercial fpgas," in *Proceedings of the 27th IEEE International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM'19. IEEE, Apr 2019, pp. 1–4.

[33] "Yosys open synthesis suite," https://github.com/YosysHQ/yosys, Yosys Headquarters, accessed: 2020-01-20.

[34] "Project trellis: Documenting the lattice ecp5 bitstream," https://github.com/SymbiFlow/prjtrellis, SymbiFlow, accessed: 2020-01-20.

[35] "nextpnr: A portable fpga place and route tool," https://github.com/YosysHQ/nextpnr, Yosys Headquarters, accessed: 2020-01-20.

[36] "Project x-ray: Documenting the xilinx 7-series bitstream,"

https://github.com/SymbiFlow/prjxray, SymbiFlow, accessed: 2020-01-20.
[37] D. Shah, "Trellisboard: Ultimate ecp5 board," https://github.com/daveshah1/TrellisBoard, accessed: 2020-01-20.
[38] G. L. Somlo, "A trustworthy, free/libre, linux capable, self-hosting 64bit risc-v computer," http://www.contrib.andrew.cmu.edu/~somlo/BTCP, accessed: 2020-01-20.
[39] "RISC-V proxy kernel and boot loader," https://github.com/riscv/riscv-pk, The RISC-V Foundation, accessed: 2020-01-20.
[40] "The lowRISC community interest company," https://www.lowrisc.org, lowRISC CIC, accessed: 2020-01-20.
[41] "lowRISC chip," https://github.com/lowRISC/lowrisc-chip, lowRISC CIC, accessed: 2020-01-20.
[42] S. Gal-On and M. Levy, "Exploring coremark – a benchmark maximizing simplicity and efficacy," https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf, Embedded Microprocessor Benchmark Consortium, accessed: 2020-01-20.
[43] "CoreMark benchmark," https://github.com/eembc/coremark, EEMBC.org, accessed: 2020-01-20.
[44] "Linpack benchmark," http://www.netlib.org/benchmark/linpackc.new, Netlib Repository at UTK and ORNL, accessed: 2020-01-20.
[45] U. F. Mayer, "nbench linux/unix benchmark," https://www.math.utah.edu/~mayer/linux/bmark.html, accessed: 2020-01-20.