Towards Zero Trust: An Experience Report

Jason Lowdermilk
Chip Scan, Inc.
New York, NY
jlowder@chipscan.us

Simha Sethumadhavan
Chip Scan, Inc.
New York, NY
simha@chipscan.us

2021 IEEE Secure Development Conference (SecDev) | 978-1-6654-3170-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SecDev51306.2021.00027

***Abstract*—Risk from supply chain attacks have gained prominence. In response to these attacks, regulators have suggested building systems on the principles of "zero-trust", an aspirational motto that urges system designers to take measures to minimize trust. But, to what degree can one minimize trust in realistic systems? The answer to this question, of course, depends on the context. In this paper, we explore this question in the context of a satellite ground station front end processor – a critical component in satellite ground stations, in both standalone and cloud settings. Based on our design and implementation experience that spanned 18 months, we observe that it is possible to achieve a significant reduction in trust as measured by the lines of code. We also find that minimizing the lines of code improves productivity and the performance of our design. Finally, we find trust can be minimized to a greater extent for standalone systems than cloud systems.**

***Index Terms*—hardware, security, HLS, high-level synthesis, debloating, zero-trust**

I. Introduction

A number of prominent supply chain attacks have dominated news headlines in recent years. In 2018, the Bloomberg Businessweek's story regarding implanted devices in Supermicro motherboards created widespread concern [1]. Even though the allegations have been largely debunked, it still demonstrates a widespread risk in global supply chains [2]. More recently, the SolarWinds supply chain attack has brought these concerns back into focus, demonstrating the fragility of our critical assets arising within an increasingly complex global supply chain [3].

One appropriate defense against these types of attacks is to adopt a Zero Trust security model [4]. When this model is applied to its fullest extent, the Zero Trust paradigm requires practitioners to eliminate implicit trust in any one element of a computer system, including its hardware elements. Critics of the Zero Trust paradigm argue that even if the software in a system could somehow be fully trusted through verification and validation, there are still potential trust issues at the hardware level: hardware may also contain exploitable bugs and malicious intrusions similar to software, and issues like Spectre and Meltdown have caused unexpected trust issues with hardware [5], [6]. Furthermore, there are concerns that third-party IP which has been integrated into a System on a Chip to represents another large attack vector for hardware [7]. And, even if these design stage issues are resolved, the hardware may still be tampered with during manufacturing, such as via stealthy dopants. [8]. The essence of this line of argument is that *trust is a race to the bottom*.

In this paper we report on our experiences at implementing full-system Zero Trust. We detail design choices we made to minimize trust in a context of a realistic engineering effort and under reasonable cost constraints. We decided the best way to think about eliminating trust is from the hardware up: in other words, the way to win the race to the bottom is to start from the bottom! In this paper we show how this hardware-up methodology worked and concretely describe the difficulties we encountered in applying this methodology, and also attempt to quantify the benefits of this method.

For our case study, we chose a satellite ground station "front-end processor" system. The reason for choosing this system was that we are quite familiar with the workings of satellite systems, and it is a real, substantial system that is a critical piece of the satellite ecosystem. At a high-level, this system processes health and status data from the satellite (Telemetry) and sends out commands. Thus hackers who can take control of the ground control system can cause havoc in a number of ways. For example, two orbiting satellites could crash into each other splintering their debris into thousands of pieces, that in turn cause the debris to crash in other satellites, causing a catastrophic chain reaction that brings down GPS, weather, defense, surveillance, science, and communication and satellites, technologically setting us back to the 1950s [1].

To achieve full-system Zero Trust using our hardware-up method, we made the following design decisions:

- Our first major decision was to implement the system on an FPGA because we can trust the reconfigurable portion of the FPGA fabric not to have backdoors because it is difficult for the foundry to envision all possible designs that can be mapped on the FPGA and break them in a predictable manner. Further, unlike CPUs, FPGAs reconfigurable fabric does not have speculative execution so it is not vulnerable to some modern CPU vulnerabilities of this genre.
- To make programming these FPGAs easier, we used High-Level Synthesis. We converted legacy C code and Clash (Haskell-like language) code directly into FPGA bitstreams. Using a functional language and an imperative language allowed us to study the generality of the hardware-up method for achieving Zero Trust. The conversion was done using a widely used commercial high-level synthesis compiler. We wished to formally verify the correctness of our implementation but

[1]This scenario is known as the Kessler's Syndrome and has gained some attention with New Space companies like SpaceX launching several small sats

could not find easily usable tools to complete the work in the time frame for the project.

• Since we wanted to reduce the trust on the compilation and conversion tools, we used a commercially available tool (ESPY) to check that no backdoors were inserted into the design during this process.

Our findings are mixed. Compared to implementing the system with Linux running on a traditional server CPU/NIC, we found that implementing the system in hardware (specifically, on commodity FPGAs using commodity tools) substantially reduces the lines of code and improves the speed of the system. This is somewhat unsurprising and expected. However, a surprising and unexpected result was that hardware design technology had matured to a point where it was not only possible to complete the design and validate it in a shorter amount of time than equivalent software, but also provide a sufficient level of assurance that it is free of backdoors and trojans using current commercial tools. This level of assurance is simply not possible with software today. We also found that if the redesigned system were to be hosted on the cloud, a significant amount of trust would need to be placed on the cloud provider because of how FPGAs are integrated into the cloud system. This level of trust could be reduced further by implementing a few simple redesigns into the cloud architecture.

The rest of the paper is organized as follows: in Section 2 we provide background on satellite ground stations. In Section 3 we describe the design decisions and rationale in our ground station design; in Section 4 we describe the results of comparisons with typical software implementations of our system. In Section 5 we discuss related work, and conclude in Section 6.

II. Background: Satellite Architectures

Satellite Ground Segments: Satellite communication consists of telemetry and commanding. Telemetry refers to data being received from the satellite, whereas commanding refers to the data being transmitted to the satellite. For this paper we will focus on telemetry, which generally requires higher data rates than commanding.

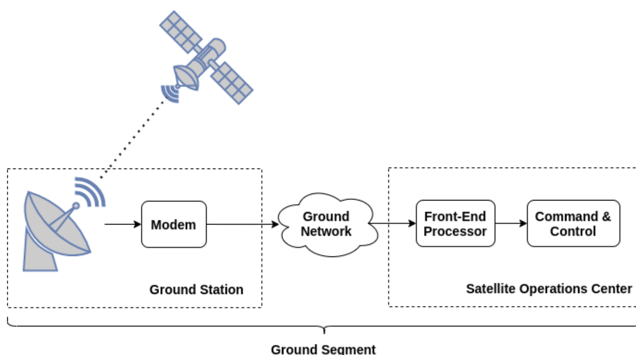

Fig. 1. Components of the Ground Segment

The ground segment of a typical satellite system contains several terrestrial-based pieces of equipment: antennas and modems reside in ground stations, which are located physically in places where they have visibility to the satellite as it orbits the Earth. Analog signals are received by the antenna and then demodulated and digitized by the modem.

The resulting digital data stream is transmitted across a network to the Satellite Operations Center[2] (SOC) which is often centrally located and staffed with flight operations personnel. Inside the SOC, two major components are the front-end processor (FEP) and the command and control (C2) system. The FEP is responsible for signal processing, forward error correction, and decryption of the telemetry data, whereas the C2 system provides the user interface for flight operations.

Front-End Processors (FEP): The FEP is considered a "cross-domain equipment" since it handles the decryption of the telemetry, and therefore resides at the boundary of two different classification levels: unclassified (black) as telemetry arrives in encrypted form, and classified (red) as it leaves the FEP after decryption. This usually requires the FEP to be split into two physically separate devices, a red FEP and a black FEP, in order to satisfy red-black separation requirements. An approved cryptographic device is placed between these to perform the decryption function.

Additionally, there is often a need for plaintext information to travel between the black and red FEPs. In some cases this is nothing more than metadata items, such as timestamps and bit rate information, which were inserted by the modem. In other cases, some portions of the arriving telemetry data frames may arrive from the satellite as plaintext. In these cases, the plaintext data must pass through an approved cross-domain device known as the data guard which validates that the data is in an acceptable format.

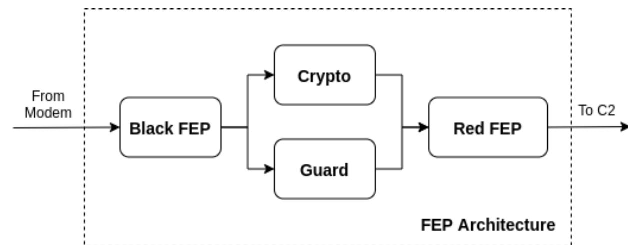

Fig. 2. FEP Architecture

The functions performed by the red and black FEPs are very different. The black FEP is responsible for signal processing and managing the transmission of data through the crypto and data guard paths. The red FEP must merge the data arriving from the crypto and data guard and then process the data as required to support the interface to the C2 system. This often involves demultiplexing data packets from different virtual channels.

[2]also called Mission Control Center (MCC) or Control Center (CC)

III. Motivation: FEP Attack Surface

Modern FEPs are often deployed on commodity servers running Linux. The data guard consists of one or more additional servers, also running Linux. Cryptos are generally implemented as application specific network appliances, not on general-purpose computers. The total attack surface of a FEP consists of the combined hardware, firmware, and software making up these commodity servers.

In terms of software, the Linux kernel weighs in at over 20 million lines of code. For a typical Linux distribution, the kernel represents 10% or less of the total, with the rest consisting of libraries, utilities, frameworks, and user-space applications. The total amount of code in a full Linux distribution can therefore exceed 200 million lines of code.

Even though Linux is open-source and frequently reviewed for security, a recently discovered decade-old privilege escalation vulnerability in sudo [9] has made it clear that even well-reviewed code that is directly related to security can still contain severe vulnerabilities which remain undetected for years at a time.

Another significant attack vector is the FEP software itself. Since FEPs are deployed in a physically secure environment with strong perimeter defenses, security is often not a top priority for FEP developers. Because the FEP will not be directly connected to the internet, the types of security precautions deemed necessary in other public-facing deployments are not made a priority. As a result, any malicious code brought into the SOC through the supply chain or other means (since the task of the FEP is to receive inputs) may have a relatively easy time locating targets.

IV. (Re)-Design of the Front-End Processor

A. Inner workings of a FEP

The FEP is responsible for performing digital signal processing on the raw data arriving from the modem (see Fig 3). The data arriving from the modem can be viewed as a stream of bits which are not byte-aligned, and begins at an arbitrary point within the data stream where the satellite signal was acquired. Frame synchronization is performed to align the data on frame boundaries, effectively discarding all data up to the beginning of the first complete frame. Derandomization is then necessary to remove the pseudo-random pattern that has been mixed with the data before transmission from the satellite. The randomization is done to increase binary data transitions, making data extraction at the modem more reliable. Forward error correction is then performed to correct any bit errors incurred during transmission through the atmosphere. Integrity verification consisting of a CRC check is then performed to ensure that the data has been fully corrected; any frames which still contain bit errors at this point are dropped. Generally, each frame contains one or more packets or packet fragments, and each frame belongs to a set of virtual channels which are multiplexed together to form the telemetry downlink. The packet extraction step demultiplexes and aggregates the packet fragments to produce complete packets, which are then forwarded to the C2 system.

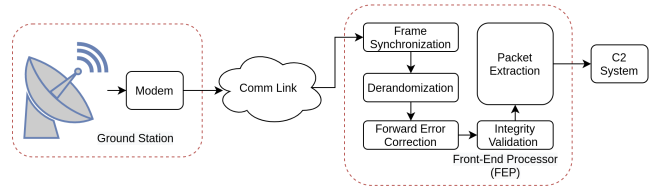

Fig. 3. FEP Inner Workings.

B. Design Methodology

The primary method of reducing the overall code size (and trust) is to replace software with hardware. The functionality that was previously performed in software is now implemented instead using hardware primitives on an FPGA, without a general purpose processor. This eliminates vast amounts of software required to run the application in the form of the operating systems, hypervisors, libraries, and support utilities required to implement the runtime environment necessary to run software.

To implement the FEP, we faced a familiar set of questions: should we develop new code to implement on the FPGA, or reuse existing code? And, which programming languages are the most appropriate in each case? For the first question, we determined that we would reuse code to implement certain aspects of the FEP such as encryption/decryption and forward error correction, since these would be too time consuming to implement and test from first principles. For these functions, we decided to either reuse existing RTL implementations or use High-level Synthesis to convert existing high-level code to FPGA bitstreams.

For forward error correction, we decided to reuse an existing open source implementation in C language from the package libfec by Phil Karn. The libfec (Forward Error Correction, RS 255,223 code) version includes support for dual-basis mapping and virtual fill, but not code block interleaving. Our first step was to add interleaving support to the C implementation. Then, we refactored the code to partition the functionality into loading, decoding, and storing portions. This allows the code to function in a byte-streaming manner where each invocation provides one byte to buffer for the next decode operation, and simultaneously returns one byte from the result of the previous decoding operation. This allows the decoder to operate in a fixed amount of memory and behave with reliable and predictable timing. In order to support high data rates, we created a parallel decoding mechanism that uses up to 64 decoder cores with a round-robin scheduler, enabling multiple frames to be decoded simultaneously while maintaining a byte-streaming interface capable of operating at the full data rate.

We considered both Bambu and Vitis HLS for conversion to RTL, and ultimately chose Vitis HLS. In both cases, the resulting RTL was able to produce a result after about 140,000 clock cycles to decode a single 255-byte code block. However, the Bambu version failed to signal the result properly due to unknown reasons, whereas the Vitis HLS version did not share this problem.

For the encryption and decryption functions, we reused a proven RTL implementation of AES available as open source from SecWorks [10]. This implementation conforms with NIST FIPS 197 and supports both 128 and 256-bit keys. The interface to this module is based on reading and writing a bank of 32-bit registers to load keys and data, configure the encryption/decryption functions, and control the device. Our first step was to implement an adapter component that presents a byte-streaming interface to the rest of the system while using a finite-state machine to interface with the AES register bank in order to configure and control all aspects of the AES core. We again found it necessary to deploy multiple AES accelerators with a round-robin mechanism in order to achieve high data rates. In effect, this created a parallel ECB encryptor/decryptor unit. Our final step was to add an additional component layer to manage nonce values in order to implement CFB encryption and decryption.

For less-complex aspects of the FEP such as frame synchronization, CRC validation, BCH code generation, and space packet demultiplexing, we decided to implement these features using HLS. Because hardware behaves in a manner similar to functional programming in which a module's input signals resolve to output signals on every clock cycle, we wanted to use an HLS based on a functional language. We initially considered Chisel (Constructing Hardware in a Scala Embedded Language), which is an HLS that has gained popularity in recent years. We also considered a newer language called Clash, which is based on Haskell. Compared to Chisel, Clash provides clearer separation of sequential and combinational logic through the use of Signal domains and through the use of Mealy and Moore transformations which are available as part of the language. We ultimately selected Clash as the HLS to use for implementing all new code, including the AES functions described above.

C. Synthesis and Checking

The HLS tools used for hardware design and synthesis provide options to utilize different types of memory that are available to the FPGA device, such as external DDR4 memory, on-chip high-speed memory or block RAM, and flip-flop memory. These options manifest themselves in the generated RTL as "hints" inserted at appropriate places in the code which will be subsequently interpreted by the synthesizer to generate interfaces for the specified memory type. For the FEP, we chose flip-flop memory in order to avoid the use of 3rd-party IP for memory controllers.

Since we do not have any 3rd-Party IP, the main concern for backdoors is through the corruption of the HLS process. To mitigate this concern we ran the ESPY hardware security verification tool after synthesis. At a high level, the tool checks for wires in the design that do not influence the outputs and flags them as potential candidates for backdoors.

D. System Integration

The system described above can be deployed on any standalone FPGA. However, we decided to implement the FEP on a cloud instance since we wanted to be able to connect to a real antenna and a real satellite, and the only reasonable cost solution is to rent these services through a cloud service provider. We used the system architecture described in Figure 4 and deployed this on the Amazon Cloud. As shown in the figure, the ground station currently transmits only to a service running on the host CPU. Similarly, to send the data to the C2 System, it needs to go through a CPU host application. In the future if Amazon (or another cloud provider) allows the FPGA instance to talk directly to other network services, we can avoid the unnecessary communication through the host CPU.

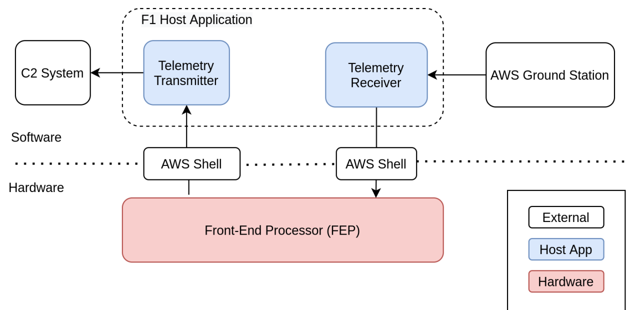

Fig. 4. System Architecture

V. Results

A. Debloating and Code Size Reductions

In a standalone mode, i.e., not the Amazon cloud deployment, the FEP architecture can run without an hypervisor or OS, just requiring some basic firmware/BIOS for management functions such as loading the bitstream and initializing the FPGA and the cards. This alone leads to a debloating of millions of lines of code. In a cloud setting, two things need to happen to achieve code size reductions comparable to the standalone setting 1) FPGAs are permitted to host services and communicate with other services bypassing the CPU and 2) confidential computing capabilities such as SGX or isolation primitives for FPGAs, are extended to the FPGA. These are not serious technical obstacles. In fact, there are research papers on how to do these for heterogeneous accelerators such as the GPU [11], and extending them to FPGAs would not be a significant task for the cloud providers if they choose to do so.

The second source of code increase is because of High Level Synthesis. The code in high level language is compact but it is converted to RTL for every component in the FEP. Clash language was used to develop all new modules, and Vitis HLS was used to convert the Reed Solomon decoder. The results are summarized in table I. Code written in Clash produced RTL with 4.5 times more lines of code, and Vitis HLS produced an increase of 4.8x to RTL. This represents a significant savings in terms of the development time required, and is fairly typical of the 5-10x ratio that is often advertised by HLS vendors. On average, each line of HLS code produced 36 logic gates.

TABLE I
HLS CODE GENERATION

Module Name	HLS Lines	RTL Lines	Logic Gates
killbuffer	59	563	109375
crc	50	110	408
derand	51	109	354
framesync	87	316	1736
roundrobin	44	124	286
deserial8	37	99	82
deserial32	36	98	390
serial8	42	146	1175
serial32	30	83	571
tlmpreproc	49	170	3704
demux	139	984	1377
rsdecoder	4324	20804	58952

B. Productivity

We developed the FEP over a period of 18 person-months, including the development of a simple telemetry simulator and integration with a C2 system. Based on our past experience, a conservative estimate for the development of a typical modern software FEP is roughly 2 person-years, given a typical environment consisting of development work shared over multiple engineers. We believe this reduction in schedule can be attributed to increased productivity due to the use of HLS and hardware-up techniques.

C. Security benefits: Memory Safety

Improper use of dynamic memory allocation can result in buffer overflows, use-after-free errors, type confusion errors, and many other errors that can become security vulnerabilities [12]. For the development of our FEP, we chose to avoid dynamic memory allocation altogether and use only statically allocated memory. This was possible because all data buffering in the system occurs only at the network interfaces, and the maximum size required can be determined. The network interface between the modem and the FEP is UDP datagrams containing raw, non byte-aligned data; the FEP receives datagrams which have a fixed size dictated by the Maximum Transmission Unit (MTU) length. The data from each datagram is placed in a static MTU-size memory buffer while it is passed through a FIFO to the hardware. The FPGA ultimately converts the raw bits into packets, which are returned to the software layer and again stored in a static memory buffer before being passed to the C2 system. These network packets are multiplexed over a maximum of 64 channels, and each packet has a maximum size of 64 kilobytes. This represents a total of 4 megabytes of static memory in a worst-case scenario to store the maximum amount of data from every virtual channel.

D. Trust Benefits

We used ESPY to perform a security scan on every component produced by HLS techniques. The results are summarized in table II. While no backdoors were detected and the ratio of false positives were reasonable, the RTL generated from Mealy transformations in Clash code consistently generated false positives. This is because of the highly-specific comparators generated by the state machine RTL.

TABLE II
ESPY SCAN RESULTS

Module Name	Backdoors Detected	False Positives
killbuffer	0	4
crc	0	4
derand	0	1
framesync	0	16
roundrobin	0	3
deserial8	0	0
deserial32	0	0
serial8	0	4
serial32	0	1
tlmpreproc	0	9
demux	0	20
rsdecoder	0	5

E. Performance

The FEP uses a bit-serial dataflow architecture in which one bit (maximum) is transferred on every clock cycle. When deploying to the Amazon EC2 F1 environment[3], the clock period used for custom logic (CL) has an 8 nanosecond period. This represents a maximum data rate of 125 Mbps.

F. Lessons Learned

Need for a FPGA cloud or FPGA confidential compute: Currently, for a cloud-based deployment such as Amazon AWS F1, it is not possible to fully eliminate the software stack after the FPGA is programmed. The operating system is required to first load the FPGA and then must remain active during the life of the FPGA session, otherwise the AWS EC2 framework will reclaim the F1 instance. Other architectures have the potential to allow the operating system and related software to be atrophied after the FPGA is programmed. In particular, this would be possible with non-cloud (standalone) implementations and possibly with other cloud services that provide a network offloading capability, such as Microsoft Azure who recently started providing satellite antenna capabilities. With these alternative architectures, it would be possible to implement the network interfaces completely within the FPGA and eliminate the need for any software running on the computer's processor at runtime. This would have the desired effect of debloating the 200+ million lines of code required for a full Linux installation even for the cloud deployment.

Need for isolation features within FPGAs: The FPGAs in our design are highly underutilized (less than 50% utilization). As such we can integrate more features from the ground station into the FPGA. Ideally, we could eliminate the need for red/black separation between FEPs, data guards, and cryptos, and allow all of these functions to be performed securely within a single FPGA rather than over multiple physical commodity servers. FPGAs provide the "Moats and Drawbridges" isolation primitives which could effectively isolate

[3]The Amazon EC2 F1 service provides a cloud-based FPGA computing environment

these different parts, viz., FEP, crypto, and guard components to ensure that side channel paths due to signal entanglement are not present. Unfortunately, none of the available FPGA cloud hosting services currently provide FPGAs that are compatible with this mechanism. As a result, it is still necessary to maintain a red/black boundary using physically separate servers at this time. This presents a future opportunity for FPGA designers to provide higher levels of consolidation and security.

Building on productivity gains: In our design, the FEP uses a dataflow architecture consisting of many independent modules. In our design, receiving modules must always be available to receive data even if that means it has to temporarily buffer data internally until it is ready. We chose this design because pushing back all the way to the data source (the satellite) is not possible, and any push-back mechanism between internal modules would only delay an inevitable overflow while complicating the design of each module. However, during implementation we discovered that this type of modular design is not ideal for HLS implementations since it masks one of the main benefits of HLS: automatic management of timing between stages. Since every module is implemented separately and then connected using a standard Verilog toplevel, the HLS compiler is not able to manage the timing between modules. Standard RTL development tools were still necessary in order to achieve the necessary timing characteristics between modules, including waveform viewers, virtual JTAG interfaces, and standard SystemVerilog testbenches. This diluted the overall promise of using HLS to some degree. However, other benefits of HLS such as increased productivity and higher-level logic were still realized. A concrete recommendation for HLS compilers would be to perform timing management across modules.

VI. Related Work

Alternate approaches to enhancing trust: In this work, we equated minimizing trust to minimizing the amount of code, and then checking that the remaining code is trustworthy by using automated tools. However, trustworthy execution can be achieved irrespective of the code size as long as formal verification techniques can be applied for reactive systems. This area has been improving rapidly and soon may offer a promising replacement for achieving trust [13], [14]. Another promising trust reduction technique is homomorphic execution [15], [16]. While homomorphic execution zeroes out trust during program execution, the creation of homomorphic programs on host computers requires only a minimal amount of trust that might be acceptable. While the minimize-and-verify potpourri approach described in the paper does not have the concisely articulated guarantees offered by end-to-end formally verified systems or encrypted execution, a nice property of this approach is that it is applicable today.

High-level Synthesis: The complexity of today's hardware designs is driving designers to use high-level synthesis (HLS) techniques [17]. HLS provides a higher level of abstraction for designers by allowing hardware components to be designed and implemented in a way that is more similar to software than traditional register-transfer level (RTL) descriptions[4]. With HLS, the specification is provided in a high level language such as C language and automatically translated or compiled into RTL descriptions. The RTL descriptions are then used to synthesize, technology map, and place & route as is typical for a hardware development workflow.

HLS has a long history that goes back to the 1970s [18]–[20]. Over time, HLS has seen many perturbations and competitive vendor offerings. Today, HLS is a mature and effective alternative to RTL workflows. Supported high level languages generally include C and C-like languages, as well as newer HLS tools that are based on functional languages. Examples that use C language include Vitis HLS and Bambu. Functional language examples include Chisel (based on Scala) and Clash (based on Haskell). For our hardware-up method, we selected Vitis HLS to use for converting legacy C code into RTL, and Clash for any new hardware development.

There are many benefits to using HLS. Developers realize an increase in productivity by using already-familiar languages and workflows. Existing software components which have already been functionally verified can be directly converted into hardware, improving reuse. Additionally, the time required for functional verification time can be reduced with HLS due to using larger and more comprehensive software testbenches, rather than RTL testbenches.

Hardware Trust Tools and Techniques Hardware development is susceptible to supply-chain attacks at several points in the development workflow. Backdoors can be introduced directly at the RTL level by corrupted EDA tools (such as HLS compilers) or through a malicious actor. Backdoors can also be introduced at the synthesis, mapping, and place & route steps by corrupted EDA tools.

Hardware backdoors are very difficult to detect through traditional testing techniques due to the size of the search space involved. Since any input value may be used to trigger the backdoor by waiting for a specific data pattern to accumulate over time, the search space increases exponentially on every clock cycle. This property enables the stealth required for a backdoor to remain undetected during testing. If a backdoored circuit survives the testing phase, it can be placed into the commercial supply chain and ultimately end up being used in critical defense or weapon systems.

For our hardware-up method, we used a security scanning tool called ESPY to ensure that all RTL is free of backdoors [21], [22]. ESPY measures the stealth of every wire in a hardware design in order to detect areas of the circuit that rely on stealth to either become active or to shield activity from affecting the primary outputs of a circuit. This is an effective way to scan RTL designs to ensure they are free from stealthy backdoors.

VII. Conclusions

Trust can be viewed as a set of axioms used to define the security of the system against specific attack vectors. While

[4]usually Verilog or VHDL

Zero Trust is ideal, the goal of security designers has been to minimize the level of trust. In this paper we showed that it is possible to minimize trust to a very large degree by using modern technologies and hardware oriented techniques. We conservatively estimate a 100x reduction in code size, and a significant level of security against hardware backdoors. Admittedly this is one example, and more applications of the methodology described in the paper would be needed to validate the results in the paper.

References

[1] D. Mehta, H. Lu, O. P. Paradis, M. A. M. S., M. T. Rahman, Y. Iskander, P. Chawla, D. L. Woodard, M. Tehranipoor, and N. Asadizanjani, "The big hack explained: Detection and prevention of pcb supply chain implants," *J. Emerg. Technol. Comput. Syst.*, vol. 16, Aug. 2020.

[2] "Supermicro rebuttal." https://www.supermicro.com/en/pressreleases/supermicro-statement-bloombergs-claims.

[3] NIST, "CVE-2020-10148.." Available from NIST, CVE-ID CVE-2020-10148., December 2020.

[4] "Department of defense (dod) zero trust reference architecture." https://dodcio.defense.gov/Portals/0/Documents/Library/(U)ZT_RA_v1.1\(U)_Mar21.pdf.

[5] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018.

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, (USA), p. 973–990, USENIX Association, 2018.

[7] S. Ray, E. Peeters, M. M. Tehranipoor, and S. Bhunia, "System-on-chip platform security assurance: Architecture and validation," *Proceedings of the IEEE*, vol. 106, no. 1, pp. 21–37, 2018.

[8] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *CHES*, pp. 197–214, Springer, 2013.

[9] NIST, "CVE-2021-3156.." Available from NIST, CVE-ID CVE-2021-3156., April 2021.

[10] "secworks / aes." https://github.com/secworks/aes.git.

[11] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th USENIX Symposium on Operating Systems Design and Implementation*, October 2018.

[12] "We need a safer systems programming language." https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/.

[13] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, p. 107–115, June 2010.

[14] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[15] C. Gentry, *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, Stanford, CA, USA, 2009.

[16] "Intel HEXL (release 1.1.1)." https://arxiv.org/abs/2103.16400, Mar. 2021.

[17] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. Smullen IV, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. S. Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D.-k. He, C. R. Ho, R. W. Huffman Jr., E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor, N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V.-M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. K. Wu, "Warehouse-scale video acceleration: Co-design and deployment in the wild," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, (New York, NY, USA), p. 600–615, Association for Computing Machinery, 2021.

[18] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, pp. 18–25, 07 2009.

[19] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[20] "Vitis high-level synthesis." https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[21] A. Waksman, M. Suozzo, and S. Sethumadhavan, "Fanci: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), p. 697–708, Association for Computing Machinery, 2013.

[22] "ESPY / chipscan products." https://chipscan.us/products/.