This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

Hunting Security Bugs in SoC Designs: Lessons Learned

Mohammad Mahdi Bidmeshki, Yunjie Zhang, Monir Zaman, Liwei Zhou, Yiorgos Makris
Department of Electrical and Computer Engineering
The University of Texas at Dallas
Richardson, Texas
{bidmeshki, yunjie.zhang, monir.zaman, lxz100320, yiorgos.makris}@utdallas.edu

***Abstract*—Security of hardware designs is gaining more attention due to the pervasiveness of computing and communication elements in our everyday lives and the constant discovery of previously unknown vulnerabilities, such as Spectre and Meltdown, in hardware computing devices. Consequently, a shift in the thinking of hardware developers is necessary to prioritize security in addition to correct functionality during the hardware design and development cycle. Towards providing insight to other researchers, in this article we present our experience from participating in Hack@DAC, a hardware security contest where teams were tasked with finding security bugs in large System-on-Chip (SoC) designs.**

***Index Terms*—Hardware Security, SoC Vulnerability, Security-Oriented Design**

I. Introduction

Increased complexity of hardware designs has caused the emergence of previously unknown vulnerabilities, which can be exploited by an adversary to develop new attacks for gaining access to private and sensitive information, or disrupting the operation of computer systems. For example, out-of-order execution, branch prediction, and speculative execution of instructions are techniques used by modern microprocessors to fill the empty slots of an execution pipeline and improve performance. While these techniques have been implemented in modern microprocessors for many years, the security implications of their specific implementation were overlooked until 2018, when their vulnerability to attacks known as Meltdown [1], Spectre [2], and Foreshadow [3] was revealed.

Although this example shows an extreme case of overlooked vulnerabilities in high-performance, modern microprocessors, other new vulnerabilities in a wide range of hardware or software systems are being constantly discovered. Unlike software vulnerabilities, hardware vulnerabilities are very difficult, if not impossible, to patch through firmware, software or operating system (OS) modifications. In addition, such software patches for hardware vulnerabilities may disable hardware capabilities or introduce additional overhead, causing performance degradation [2]. In turn, this can result in customer dissatisfaction, as certain features which were paid for to improve performance, e.g., speculative execution, cannot be fully utilized due to security concerns.

Security concerns are not limited to top of the line, high-performance processors and systems. While smart-phones, personal computers and server or cloud systems mostly utilize high-performance processors, embedded and Internet of Things (IoT) devices which have lower processing capabilities are an important part of our connected world and constitute a wide attack surface which can be exploited by adversaries. Most of these attacks may exploit software vulnerabilities; however, hardware features can also provide opportunities to be exploited. Therefore, establishing a security-aware design process, both from the software and from the hardware perspective, is paramount.

Hardware design and development follows a life cycle similar to software, namely requirement analysis, design, manufacturing, testing, distribution, deployment (use and maintenance), and disposal [5]. Attack vectors can be introduced in every stage of this life-cycle [6] and, therefore, hardware security should be studied from different perspectives, e.g., security bugs in the design, hardware Trojans (malicious capabilities planted in the hardware at the design or at the manufacturing stage, which can be exploited later by attacks), or recycled/unreliable devices which may enter the system at the distribution stage.

In 2018, we participated in the Hack@DAC two-phase competition [7] where we were tasked with finding bugs with security implications in the HDL code of two SoC designs. Given a brief requirement description, we had to analyze a considerable amount of HDL source code and report possible security vulnerabilities. This article summarizes our experience in this competition, and points out procedures which may help in extending the hardware developer's mindset to a security-oriented approach. Considering the task at hand, in this article we focus on security vulnerabilities (inadvertent or malicious) in the design stage, i.e., in the HDL source code of the hardware. Since we were not involved in the development of the design, our strategy is centered on streamlining our search, in order to identify and report as many bugs as possible. We also introduce other procedures which could be useful in order to improve our bug-finding approach.

While the competition organizers presented a thorough analysis of various testing and verification approaches, and tools to find the introduced or inherent security bugs in SoC designs [7], they were aware of what bugs they are looking for in their analysis, which makes the task easier. We, on the other hand, were the participants in the competition and,

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

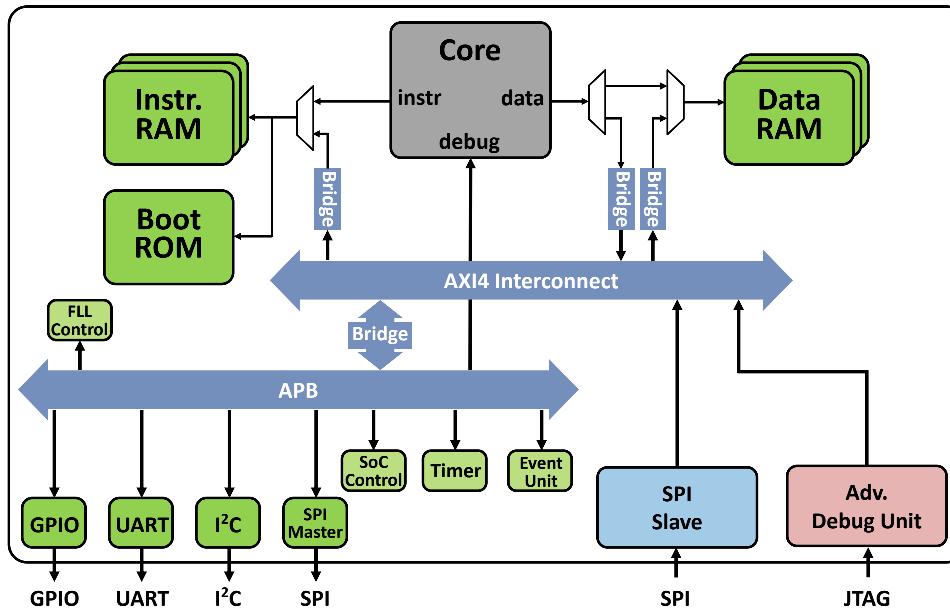

Fig. 1. Block diagram of one of the analyzed SoC designs (i.e., the PULPino platform [4])

therefore, did not have the foreknowledge of the security bugs to develop or evaluate our approach. Also taking into account the time limitations of the competition, we were successful in finding many, but not all of the bugs in those SoC designs. As a result, we believe that our findings in combination with the experience of other participants and the organizers [7], can provide valuable security awareness insight to the SoC designers and the hardware security research community.

II. SoC Structure and Specification

The SoC designs provided in the competition were modified versions of the single-core microcontrollers in the PULP platform [4], specifically, *PULPino* in the initial phase, and *PULPissimo* in the final phase. Both of these SoC designs are based on a RISC-V processor architecture [8], and can be customized as either a 32-bit 4-stage core called RI5CY, or a 32-bit 2-stage core called Ibex (formerly Zero-riscy). These platforms are equipped with various peripherals, such as general purpose input/outputs (GPIOs), universal asynchronous receiver/transmitter (UART), serial peripheral interface (SPI), inter-integrated circuit (I^2C) interface, debug unit and JTAG interface, etc., with the PULPissimo being the higher-end platform equipped with direct memory access (DMA) unit and supporting the addition of hardware processing engines (HWPE), i.e., hardware accelerators. Fig. 1 shows the block diagram of the PULPino platform.

The PULP platform provides the HDL code for the SoC and includes simulation scripts and a software development kit (SDK). In addition to the original specification of the SoC designs and their security requirements, the competition organizers introduced additional requirements, including the protection of the debug unit and GPIOs against unauthorized access. However, these additional specifications were brief and incomplete, which made our bug-hunting task rather difficult. Moreover, the provided code included test benches which allowed loading the software code, running it on the processor in a simulation environment and observing and monitoring the state and the outputs. As expected, at first glance, the normal operation of the platforms was not disturbed by the introduced security bugs.

The attacker modeled in the contest was assumed to have physical access to the device and to be familiar with the overall design of the SoC systems. Also, the attacker could launch attacks on the systems through software, hardware interfaces or a combination thereof; therefore, protecting the debug interface from unauthorized access was added as a security specification to the SoC designs.

III. Security Analysis Procedure

While various methodologies can be adopted in security bug finding in hardware designs, we suggest the following steps which we followed to the extent that time constraints of the contest and our access to specialized tools allowed. While we are advocates of a security-aware development process and we believe that security analysis should be part of the design and development process, these steps might provide guidelines for a scenario where the hardware is already developed and further scrutiny of the SoC design is performed by independent teams, as was the objective of this competition. Moreover, following guidelines and procedures for developing verifiable designs, as described in [9], can also help in streamlining the verification of security-related functions. Fig. 2 summarizes the suggested procedure which is described next.

A. Specification and Requirement Analysis of the Design

The first step in analyzing a design is to understand its specifications and requirements. In the contest, since our first

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

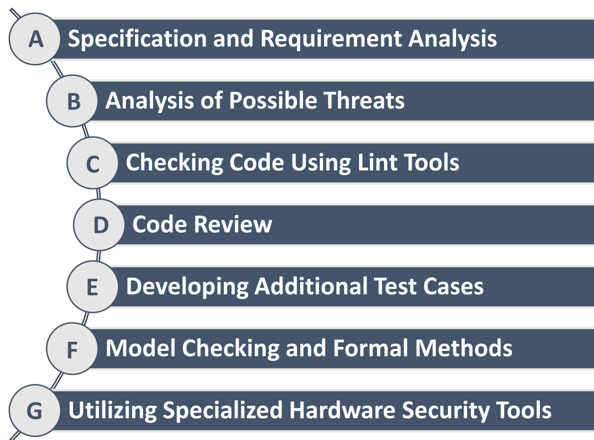

Fig. 2. Our suggested bug-finding procedure

```
1  csr_restore_mret_i: begin // MRET
2    unique case (mstatus_q.mpp)
3      PRIV_LVL_U: begin
4        mstatus_n.uie=mstatus_q.mpie;
5        priv_lvl_n   =PRIV_LVL_U;
6        mstatus_n.mpie=1'b1;
7        mstatus_n.mpp=PRIV_LVL_U;
8      end
9      PRIV_LVL_M: begin
10       mstatus_n.mie=mstatus_q.mpie;
11       priv_lvl_n   =PRIV_LVL_M;
12       mstatus_n.mpie=1'b1;
13       mstatus_n.mpp=PRIV_LVL_U;
14     end
15     default:;
16   endcase
17   epc_o = mepc_q;
18 end // csr_restore_mret_i
```

Fig. 3. Privilege level handling for `MRET` instruction in RI5CY core

encounter with the PULP platform was during the initial competition phase, we had to perform a detailed analysis of the design, understand the system and the processor architecture, its peripheral and debug units, and interconnections used for communication with the processor and the instruction and data memories.

Moreover, we performed analysis on the security specifications, both from the original design point of view and as related to the additional security requirements of the contest. As an example, the debug unit does not have any access restriction in the original platform and any entity with knowledge of the system can communicate with the PULP system and perform debugging operations. However, access restriction was added to the specifications of the altered PULP platforms provided in the contest. On the other hand, the security specifications are sometimes (in the case of this contest, deliberately) incomplete. Therefore, in such cases, we tried to make reasonable assumptions. In a general security analysis, such incomplete specifications should be revised to be more comprehensive, if at all possible.

In addition, the degree of protection a system requires, should be defined. For example, side-channels might be a concern in an application of the SoC design, and therefore, it may require special preventive measures, which may be implemented at the design or the circuit level. While another application may be only concerned about direct data leakage.

B. Analysis of Possible Threats

Based on the specifications of the design, its functional units and peripherals, in this step we seek to envision possible security threats for these units and the system as a whole. In this process, learning about security vulnerabilities that previously have been discovered in similar systems is really helpful. For example, as we mentioned earlier, various implementations of advanced features, such as out-of-order execution, branch prediction, and speculative execution of instructions in modern CPUs, can be vulnerable and enable attacks leading to the violation of data isolation or causing information leakage, as shown by attacks such as Meltdown [1], Spectre [2], and Foreshadow [3]. If a processor in an SoC design implements such advanced features, such known vulnerabilities should also be considered while analyzing its design. However, additional thought has to be given on other vulnerabilities that may be specific to the design at hand.

Another aspect that deserves special attention in this step is threat analysis of the system as a whole. For example, if access to the system requires authentication at one hardware interface, we need to ensure that there is no other interface or backdoor in the system that can provide similar access, or if such other interface exists, that it is also protected.

In our case, one of the security features of the processor was the implementation of several privilege levels. RISC-V specifications define four levels of privilege, namely user/application-, supervisor-, hypervisor- and machine-level [10]. Only the implementation of machine level is mandatory. Programs with the highest privilege level, i.e., the machine level, can have access to all memory addresses and manipulate critical control registers, e.g., specific control and status registers (CSR). On the other hand, a user-level program must not be allowed to execute with a higher privilege in which it can read or corrupt memory locations with unauthorized access. The RI5CY core in the PULP platform implements two of the above privilege levels and, therefore, correct implementation of this feature is important for the security of this platform. As an example, the code excerpt in Fig. 3 shows the handling of privilege level upon the execution of the `MRET` instruction, i.e., return from trap (interrupt) in machine mode.

As shown by this code, the privilege level after the execution of the `MRET` instruction is determined by the value of `mstatus_q.mpp`, which consists of the bits of the status register storing the privilege level before entering the trap. Interrupt-enable bits are also manipulated by this instruction, according to the RISC-V specifications.

Privilege levels may also be utilized for limiting physical addresses accessible by a lower-privilege context, which is im-

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

plemented by an optional physical memory protection (PMP) unit, as defined by the RISC-V specification [10]. To our knowledge, the RI5CY processor provided in the altered PULP platforms in the contest did not implement a PMP unit and, therefore, we did not perform a memory protection analysis in this case.

Among all the peripherals and functional units, we paid more attention to the debug unit and the GPIOs, as those were specified as in need of access protection in the security specifications of the contest. Specifically, external access to the debug unit through the JTAG interface was protected by a password checking mechanism, whose correct implementation and operation was crucial for the security of the SoC.

C. Checking Code Using Lint Tools

Static analysis tools, also known as Lint tools, were mainly developed for checking software source code and are also available for hardware description languages (HDLs) such as Verilog, SystemVerilog and VHDL. These tools check compliance of the code with a series of customizable rules and guidelines and are, nowadays, part of the hardware development and verification process [9]. Lint tools check how the HDL constructs are used in the design. For example, a rule may check whether a signal is used but never assigned, or whether there is a mismatch in the bit-length of two signals in a compare statement. Devising high-quality rules and enforcing them by revising code to comply with Lint checks can reduce the number of errors that may occur and may be captured in later simulation steps, thereby making functional verification much easier. In the same way, efficient use of Lint tools can also reduce the number of bugs with security implications. As we show later, a few of the introduced bugs in the provided PULP platforms could have been captured by Lint checking.

D. Reviewing Code Implementing Critical Security Aspects

Code review is part of any production level software or hardware development process and, in such reviews, paying attention to the security aspects of the design is important for reducing the number of security bugs which can appear in the end system. Anything suspicious is documented in this step and corrections or remedies are recommended. This can lead to revising the specifications and/or implemented code, as well as adding to additional test cases to cover the points of concern.

In the context of this competition, while we reviewed most of the units and their code as much as time allowed, we prioritized code review of the functional units based on the threat analysis of Section III-B. We went through the implementation of the privilege levels in the processor and checked whether changes in the privilege level are in accordance with the specifications of RISC-V and only occur during execution of authorized instructions or interrupts. We also checked whether register access for each privilege level complies with the specifications. Other units for which security specification were provided and/or required, including the password checking methodology for accessing the debug unit, as well as the mechanisms for accessing GPIOs, also caught our attention and were subjected to more detailed scrutiny and review. We documented any part of the code that we found suspicious for the next step of our analysis.

E. Developing Additional Test Cases

Directed (deterministic) and random tests are among the test types that are used in the hardware design process. While directed tests are effective in verifying the anticipated corner cases, random tests can be used to find complex problems arising from fine interactions between the functional units or complex sequences of multiple simultaneous events [9]. Adding new test cases based on the analysis described in the previous steps is, generally, a good idea. Indeed, such tests can document any identified concerns in the previous steps, and can be used to verify security aspects of the design in current and future revisions.

Consistent with this principle, in the altered PULP platforms of the competition we identified several points of concern with regards to the specified security features and we developed various test cases to verify or disprove these concerns. This was streamlined by the excellent project architecture of the PULP platform, which allowed us to add and run our custom developed test cases for the altered platform without much difficulty.

F. Employing Model Checking and Formal Methods

Model checking and formal methods could be effective approaches for verification and security validation, especially for security-critical blocks such as the privilege-level implementation or the authentication mechanism for the debug unit in our altered PULP platforms. This requires formalizing the properties to be verified using assertions in temporal logic, using Property Specification Language (PSL) or SystemVerilog Assertions (SVA).

Due to time limits, we did not use formal verification in our analysis. However, the altered PULP platform included a few assertions used in simulations and we also added a few more in our test cases to understand and verify critical and security-related functionality of the provided code in the simulation of our test cases. We could have formalized some of the critical security functionalities and employed formal verification tools in our efforts, if we had enough time. On the other hand, formal verification's ability to capture security bugs also has its limits. As reported by the organizers of Hack@DAC [7], formalizing some of the desired security properties might not be easy or even possible, hence, preventing the formal verification tools from capturing certain security bugs. We believe that as security gains more attention in hardware designs, testing and verification tools, including formal verification methodologies, will be augmented and enhanced to fill this gap.

G. Utilizing Specialized Tools and Methods Developed for Hardware Security

The recent attention paid by the research community to the security of hardware has stimulated the development of

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

special tools and methodologies for ensuring hardware trust. Depending on the application and the type of the design, utilization of such tools can provide higher security assurance for the design.

As an example, SecVerilog [11] and its improved variants introduce an information flow tracking (IFT) methodology for securing a hardware design against timing channels. This method enforces information flow policies by introducing a type system, wherein type annotations determine the security levels of the signals and can be defined as a function of digital signal values, i.e., it employs dependent types. It is essentially Verilog, which has been extended with type annotations. SecVerilog is powerful and provides a very accurate information flow model. It can be effectively utilized to enforce isolation properties ensuring that sensitive and secure data do not leak to public sites.

As another example, VeriCoq and VeriCoq-IFT [12] are tools based on a proof-carrying hardware intellectual property (PCHIP) framework and can be utilized to develop and check the validity of security properties for the hardware designs in an interactive theorem prover such as Coq. Due to time limitations, in our analysis for this competition, we did not utilize any specialized tools such as the aforementioned solutions.

IV. Examples of Bugs Found

In this section, we provide examples of bugs that we identified in the modified PULP platforms, i.e., PULPino and PULPissimo. We tried to follow the approach described in Section III as much as possible. However, as mentioned, we did not employed Lint tools, model checking and specialized hardware security tools in the competition. While the bugs were deliberately inserted in these SoC designs for the purpose of the competition, a brief review of some of them may give insight regarding the consequences of incomplete verification of a design, especially from a security point of view.

A. Debug Unit

As mentioned earlier, both PULP platforms are equipped with a debug unit which is accessible through a JTAG interface and requires protected access. In our analysis, we found that this protected access is implemented by requiring the user to provide a secret binary sequence, i.e., a password, to enable the JTAG controller to accept and execute commands. Assuming that such an access control mechanism is sufficient for this design, which may not be the case in general due to shortcomings such as having a single, short, hardcoded password for all chips, we still found several flaws in the implementation of this mechanism in both platforms, as we describe below.

1) Existence of Another Open Interface: As seen in Fig. 1, in addition to the JTAG interface, PULPino provides an SPI Slave interface which has direct access to the AXI4 interconnect. This allows direct access and modification of the processor registers through its debug interface using the SPI slave, which essentially gives the SPI slave interface a similar capability as what is provided by the JTAG interface.

```
1  // ...
2  reg [31:0] idcode_reg;
3  reg [31:0] tmp_pwd;
4  //sequential logic
5  always @(posedge tck_pad_i or
6           negedge trstn_pad_i) begin
7    if(trstn_pad_i == 0) begin
8      // ... reset logic
9    end else begin
10     idcode_reg[counter] <= tdi_pad_i;
11     counter <= counter + 1'b1;
12   end
13 end
14 always @(*) begin // combinational logic
15   if(counter == 5'b11111) begin
16     pwd_check = (tmp_pwd[7:0]
17       == idcode_reg[7:0]);
18   end
19   logic_reset = (idcode_reg[counter]
20          != tmp_pwd[counter])? 1: 0;
21 end
```

Fig. 4. JTAG password checking bugs

In our analysis, we found that, unlike the JTAG interface, there is no external access control mechanism implemented in the SPI slave unit. Accordingly, based on the aforementioned requirements, this poses a security risk.

2) JTAG Password Checking: The implementation of password checking in both of the provided PULPino and PULPissimo platforms suffered from several bugs. The code snippet in Fig. 4 shows the part in the PULPino platform where the input password is compared with the hardcoded on-chip password, and the result is assigned to the `pwd_check` signal. From lines 16-17 in this code, it is evident that only 8 bits of the input password are compared to the hardcoded one, reducing the effectiveness of this mechanism and enabling brute-force password hacking using only 256 tries.

Even if this issue is resolved, we found another bug related to the reading of the user password from the JTAG input. Through close inspection of this code, we realized that serially reading in the user-provided password is coded in sequential logic (lines 5-13 in Fig. 4), while validating the password is done in combinational logic (lines 14-21). As a result, when the `counter` reaches the value of `5’b11111`, the last bit of the input password is assigned to the MSB of `idcode_reg` in the next clock cycle, while the password comparison is performed in the current cycle. Therefore, the MSB of the input password is not correctly compared with the hardcoded password. One potential mitigation of this issue is to implement the validation logic to perform comparison after the full password is read.

Another bug we found in this interface was the incorrect initial reset value of the `pwd_check` signal, upon activation of the reset input of the JTAG interface. Such bugs may be prevented by proper use of Lint tools and its associated rule checks. Fig. 5 shows a sample simulation code to exploit this bug. Normally, to have a successful write to an AXI address

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

```
initial begin
  // ...
  logic [31:0] my_pwd;
  logic [31:0] tmp_do;
  my_pwd = 32'h9010_0101;
  #400ns;
  adv_dbg_if.jtag_reset();
  /* adv_dbg_if.
     jtag_cluster_dbg.shift_nbits_noex
       (32, my_pwd, tmp_do); */
  adv_dbg_if.jtag_softreset();
  adv_dbg_if.init();
  adv_dbg_if.axi4_write32
    (32'h1A11_0000, 1, 32'h0001_0001);
  // ...
end
```

Fig. 5. Sample simulation code to exploit JTAG bugs

through the JTAG interface (lines 13-14), the JTAG password has to be sent in serially first. However, as can be seen in commented lines of this code (8-10), the password checking mechanism can be bypassed just after JTAG reset. The other bugs can be exploited similarly by uncommenting lines 8-10 to allow sending in the JTAG password, and changing the value assigned to `my_pwd` in line 5 of the code in Fig. 5 to try 8-bit or 31-bit passwords accordingly, as described in the bugs.

Similar bugs were also found in the implemented password checking mechanism for the PULPissimo platform provided in the competition.

B. AXI4 Interconnect

The AXI4 protocol, which is used as the interconnect in the PULPino platform, defines different levels for access permissions, including flags indicating secure/not-secure, and privileged/non-privileged access. We found that special considerations were implemented for accessing the status register through the debug unit, including having another layer of password-checking for such access. However, the security flags for transactions accessing the status register were not set correctly in the code.

C. GPIO Protection

As mentioned earlier, GPIO access protection through the debug unit was another security requirement of the provided SoC designs. In our analysis, we found several bugs in the implementation of this requirement, as shown in Fig. 6.

From this code, we see that the range-checking for protecting the read access to the GPIO address range is not inclusive of the start and end addresses (lines 3-4). Also, password checking in Fig. 6 is essentially ineffective since lines 9-12 in this code assign `data_out_reg` to `data_o` irrespective of `pwd_check` value or range checking of lines 3-4. Furthermore, we found that although an attempt was made to implement a read protection mechanism for the GPIO range, there was not any protection against the write access through the debug unit, which may create security concerns based on the provided requirements.

```
always_comb begin
  if (!(pwd_check)) begin
    if((addr_i > `GPIO_START_ADD)
      && (addr_i < `GPIO_END_ADD))
    begin
      data_o = 64'b0;
    end
  end
  if (AXI_DATA_WIDTH == 64)
    data_o=data_out_reg;
  else if (AXI_DATA_WIDTH == 32)
    data_o={32'h0,data_out_reg};
end
```

Fig. 6. Security bugs in GPIO access protection

V. Conclusion

This article summarized our experience from participating in the 2018 Hack@DAC competition, where we were tasked with finding security-related bugs in two SoC designs. While the bugs were deliberately inserted into the designs, because of the relatively large size of the SoCs, finding them was challenging. We demonstrated a few of the bugs that we identified, with the hope that their description may be helpful to other researchers in this area. Certainly, our hunting effort may have not been complete and may have missed several other bugs. Nevertheless, we feel that a disciplined approach, such as the one we presented here, as well as proper utilization of security and verification tools and methodologies, can result in successful finding of many more complicated issues which we may have missed.

References

[1] 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 *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, Aug. 2018, pp. 973–990.

[2] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, Aug. 2018, p. 991–1008.

[4] PULP platform. Accessed July 10, 2020. [Online]. Available: https://pulp-platform.org/

[5] S. L. He, N. H. Roe, E. C. L. Wood, N. Nachtigal, and J. Helms, “Model of the product development lifecycle,” Sandia National Laboratories, Tech. Rep. SAND2015-9022, September 2015, accessed July 10, 2020. [Online]. Available: https://prod-ng.sandia.gov/techlib-noauth/access-control.cgi/2015/159022.pdf

[6] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, “Hardware Trojan attacks: Threat analysis and countermeasures,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[7] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, “HardFails: Insights into software-exploitable hardware bugs,” in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019, pp. 213–230.

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MDAT.2020.3013727, IEEE Design and Test

[8] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, 2014.
[9] L. Bening and H. Foster, *Principles of verifiable RTL design*. Springer, 2001.
[10] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual volume II: Privileged architecture version 1.9," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*, 2016.
[11] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, 2015, pp. 503–516.
[12] M.-M. Bidmeshki, X. Guo, R. G. Dutta, Y. Jin, and Y. Makris, "Data secrecy protection through information flow tracking in proof-carrying hardware IP - part II: Framework automation," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 12, no. 10, pp. 2430–2443, Oct 2017.

Mohammad Mahdi Bidmeshki received his Ph.D. in computer engineering from The University of Texas at Dallas in Richardson, Texas in 2018, where he is currently a post-doctoral research associate. His current research includes hardware-based security, trusted hardware design, formal methods in security and verification, and the applications of machine learning in computer security.

Yunjie Zhang is pursing his Ph.D. in electrical and computer engineering at The University of Texas at Dallas. His research interests include applications of machine learning in workload forensics and malware detection. He is a student member of the IEEE.

Monir Zaman received his Ph.D. in the field of digital hardware security and cross-layer power estimation improvement, and M.S. in electrical engineering from The University of Texas at Dallas in Richardson, Texas, in 2019 and 2011 respectively.

Liwei Zhou received his Ph.D. and M.S. degrees in electrical engineering from The University of Texas at Dallas, Richardson, Texas in 2018 and 2013 respectively. His research interests lie in trustworthy security-enforced computer architecture for system security applications, e.g., computer forensics and malware detection.

Yiorgos Makris is a Professor of Electrical and Computer Engineering at The University of Texas at Dallas, Richardson, TX, USA. His research focuses on applications of machine learning in test, reliability, and security of ICs, with particular emphasis in the analog/RF domain. Makris has a PhD in computer engineering from the University of California San Diego, La Jolla, CA, USA. He is a Senior Member of the IEEE.

2168-2356 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on August 04,2020 at 23:37:37 UTC from IEEE Xplore. Restrictions apply.