Fuzzing Hardware: Faith or Reality?

(Invited Paper)

Weimin Fu
Dept. of Electrical and Computer Engineering
Kansas State University
Manhattan, Kansas
weiminf@ksu.edu

Orlando Arias
Dept. of Electrical and Computer Engineering
University of Florida
Gainesville, Florida
orlandoa@ufl.edu

Yier Jin
Dept. of Electrical and Computer Engineering
University of Florida
Gainesville, Florida
yier.jin@ece.ufl.edu

Xiaolong Guo
Dept. of Electrical and Computer Engineering
Kansas State University
Manhattan, Kansas
guoxiaolong@ksu.edu

***Abstract*—Compared to software defects which can be patched in the field, hardware defects are permanent. As hardware iterations accelerate, a leftward shift in hardware testing is necessary. Among all existing techniques, formal methods (both automated and deductive) are the most effective solutions in detecting vulnerabilities in hardware. However, most of the existing formal methods are not scalable to large-scale designs due to the lack of practical automated tools. Very recently, hardware fuzzing solutions are proposed which treat the executable simulation code directly as software and test it with a Fuzz tool such as AFL or Symbolic Execution Engine such as KLEE. In this paper, we survey existing hardware fuzzing studies and discuss whether it is a valuable research direction to pursue. We also review these approaches by identifying potential challenges and gaps, based on which we present visions that can be performed to eliminate these challenges.**

***Index Terms*—Hardware Security, Formal Verification, Hardware Fuzzing**

I. Introduction

The rapid growth of the semiconductor industry put the threat of hardware system front and center among all security concerns. The reasoning behind this issue includes the long hardware supply chain, widely use of intellectual property (IP), explosive increase of Internet of Things (IoT) devices. System-on-chip (SoC) designers or system integrators have unprecedented security concerns with the growth of third-party vendors in the semiconductor industry [1]. As the complexity of SoC design keeps rising, SoC designers have been overwhelmed with the workload of manually diagnosing security vulnerabilities for a long time. Consequently, the industry has an urgent demand for automated methods to detect and evaluate vulnerabilities within the design stage.

On the other hand, the mitigation of vulnerabilities after the design stage leads to increased costs and delayed time-to-market (TTM). With the increased complexity of integrated circuits, hardware engineers rarely design them from scratch. In a rapidly evolving market, economic pressure demand, shorter design cycles, and increasingly complex designs are leading toward the increased adoption of third-party IP (3PIP), exposing systems to more potential vulnerabilities to Hardware Trojans. Innocent or malicious hardware flaws are permanent with little or no way to "patch" the hardware. Replacing flawed integrated circuits – or even repairing it – may be costly and complicated, with estimates ranging wildly [2], [3]. Consequently, a modified chip design process with left-shift verification has become a trend [4], with pre-silicon verification techniques gained increasing prominence.

In the hardware security area, a variety of countermeasures have been developed for the verification and validation of SoC's security at pre-silicon level [5], [6]. Among all existing solutions, formal methods have proven to be effective in detecting various vulnerabilities [5], [6]. However, scalability is still a considerable challenge when applying these methods to complex designs such as SoC [5]. In fact, only very few of the current SoC formal verification approaches are scalable and practical in the industry due to 1) the lack of automatic and efficient tools [7], 2) the use of inappropriate modeling and validating approaches in securing SoC system [8].

In exploring vulnerabilities in software, Fuzzing becomes a popular and scalable solution in recent years due to its capability of handling large applications. The feature that Fuzzing relies on few knowledge of the target and could apply flexibly to different portions of the software world, such as kernel, application, file system, server backend, makes Fuzzer maintained by a great diversity of people, which makes the field of Fuzzing vibrant. For hardware design, the correct design of complex hardware poses serious challenges for chip engineers. Simulation is traditionally the standard for hardware testing. However, it is not feasible to simulate all possible input patterns to verify a hardware design. Moreover, Trojan detection in 3PIP is more challenging than other logic modules, especially that there is no golden chip as a reference model in many application scenarios.

To solve these issues, researchers start to apply software fuzzing in addressing threats in the hardware platform recently [9], [10]. Although fuzzing is an effective software approach, there are still limitations and inappropriate assumptions to use it on hardware directly. In this paper, we investigate these approaches and delve more deeply into the compiler and simulator utilized as the infrastructure of hardware fuzzing. The main contributions of this paper are listed below.

- This survey offers a literature review on the formal verification and analysis approaches utilized in securing hardware design and software programs, respectively.
- Technique details of existing hardware fuzzing ap-

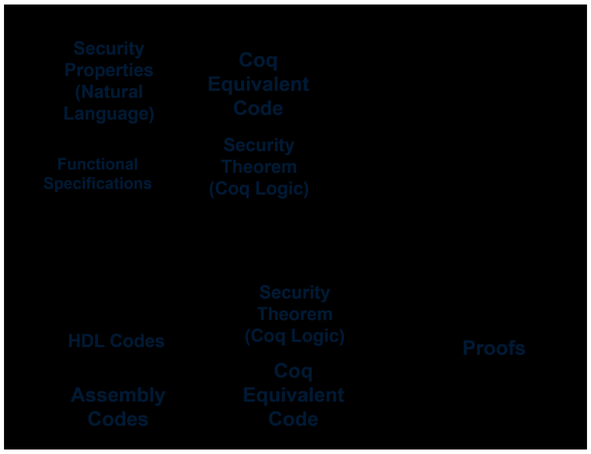
Security Properties (Natural Language)
Coq Equivalent Code
Functional Specifications
Security Theorem (Coq Logic)
Security Theorem (Coq Logic)
HDL Codes
Proofs
Assembly Codes
Coq Equivalent Code

Figure 1: PCH framwork: vendor proves security of hardware and consumer checks the results [5] .

proaches RFuzzing and HW-Fuzzing are summarized. Analysis and discussion are performed in these methods relying on conversion from hardware to software.

- The research trends in applying fuzzing approaches in hardware security are proposed.

II. Pre-silicon Hardware Security Verification

We provide an introduction to formal methods evaluating hardware at the pre-silicon stage, mainly at register-transfer level (RT-Level). Hardware security checking using formal methods briefly includes satisfiability (SAT) solving [11], model checking [12], theorem proving [8], [13], equivalence checking [14], symbolic simulation [15], as well as information flow tracking (IFT) [16]. We mainly discuss IFT, theorem proving and model checking due to their popularity.

A. Interactive Theorem Proving

Theorem provers are used to prove the satisfiability or disatisfiability of the target system against properties. Theorem proving provides a flexible way to model and validate a series of predefined properites in a design. As such, the popularity and demand of these tools have increased over the years, with Coq [17] and Z3 [18] being amongst the latest being introduced. However, it is challenging to apply them to verify the large-scale system due to the enormous time cost and manual workload.

Among formal approaches proof-carrying hardware (PCH) is developed to assure IP trustworthiness using an interactive theorem prover [6]. The idea of PCH approach comes from proof-carrying code (PCC) which was developed for assuring software programs [19]. Authors in [6] proposed the first PCH framework for checking dynamically reconfigurable hardware platforms. This PCH validates the equivalence between the hardware specification and implementation through employing equivalence checking in combinational circuits. Other PCH frameworks are geared towards verifying security properties on untrusted soft-IP cores [5], solving scalability issues [8], and runtime concerns [11]. Software and hardware co-verification approaches have also been proposed [13] to eliminate the semantic boundary between hardware and software. In these frameworks, the Coq proof assistant was used to represent security properties, hardware designs, and formal proofs.Although PCH frameworks have been effective in ensuring the trustworthiness of soft-IP cores [5], the approach is still unable to reach SoC level security checks [8].

B. Model Checking

There are many model checking based approaches utilized in verifying and validating hardware and software applications.In this method, the state-space of the model is explored to check whether a given specification is satisfied. Applying to hardware verification, the Verilog/VHDL code of the hardware accompanied by an initial state is represented as a transition system and its behavioral specification is represented as a temporal logic [8]. If there exists a case where the model does not satisfy the specification, a counterexample is produced by the model checker. At this point, the user is free to utilize this information to correct the design.

Symbolic Model Checking (SMC) is one of the earliest methods for hardware verification [20]. It uses a reduced binary decision diagram (ROBDD) to express transition system states. ROBDDs are a unique, canonical representaiton of a Boolean expression of the system [21]. Another model checking approach, called bounded-model checking (BMC), performs symbolic checking through a SAT solver replacing binary decision diagrams (BDDs) [22]. However, these model checking applications are not effective in checking SoC level hardware due to the state-space explosion issue.

A system-on-chip (SoC) bus protocol verification framework was presented in [12] with the goal of verifying the security properties of SoC bus implementations. The bus protocol specification plays the role of the golden reference. Experimental results on an ARM AMBA protocol demonstrate that the approach is applicable to prevent information leakage and DoS attack by verifying security properties. However, only the bus interconnect is considered during testing. Unfortunately, the method also suffers from the state-space explosion problem resulting in scalability issues.

C. Information-Flow Tracking

Information-flow tracking (IFT) approaches attempt to detect unintended or hidden paths which may lead to the leakage of sensitive information. As such, a non-interference policy is enforced in these solutions, reducing the dependency between lower sensitive outputs and higher sensitive inputs [23]. Existing IFT solutions usually require manual work for either annotating RTL code or proving properties. Users are tasked with selecting sources and targets for potential information leaks. IFT serves as a backbone for the area of language-based security, allowing for the creation of HDLs that can assure the trustworthiness of hardware at the design stage. Various secure RTL programming languages, such as Caisson [24], Sapper [25], and SecVerilog [26] are developed to check the noninterference property based on IFT. The main drawback of IFT is the cost of applying the method. To setup the information flow policy, developers or users must learn the sophisticated tag system used by the languages and tools to manually instrument the design. To alleviate the issue, instead of enforcing non-interference, Guo et al. proposed QIF-Verilog in [27] to relax the property of a hardware description. QIF-Verilog quantifies how much information is leaked by extending the Verilog HDL with a security label. The label is used to calculate an accumulated remaining uncertainty (RU) using the entropy generated during label propagation. Leakages are quantified through the accumulated RU in a given design.

III. Software Security Analysis

Recently, researchers have started to leverage software security analysis techniques to perform security analysis. In this section, we provide an introduction to some of these schemes.

A. *Static Analysis*

Static analysis is a technique which performs checks on source code through semantic analysis, forgoing the execution of the code [28]. Among the first uses was PFORT verifier [29], which was employed to screen the portability Fortran code to detect potential problems. Static analysis tools detect vulnerabilities according to a predefined specific pattern. This poses limitations on what can be accomplished. First, if a rule has not been written to find a particular problem, static analysis tools will not be able to detect it [30]. Second, since a pattern can appear in a non-malicious or vulnerable context, static analysis tools are prone to high false positive rates and may be forced to generalize and miss actual vulnerable code [31]. Third, static analysis presents challenges in terms of model size and functionality which may result in an explosion of states [32], [33].

B. *Dynamic Analysis and Fuzzing*

In contrast to static analysis, dynamic analysis uses a program's state at run-time to detect potential bugs. This scheme can achieve high accuracy in bug detection, but is faced with some fundamental constraints. The most glaring issue is that it is impractical to exhaust all inputs for testing. In writing test cases, users generally consider some common scenarios such as forward test, reverse engineering [34] and boundary value [35].

Fuzzing (short for fuzz testing), which was first proposed by Miller [36] is "an automatic testing technique that covers numerous boundary cases using invalid data (from files, network protocols, application programming interface (API) calls, and other targets) as application inputs to better ensure the absence of exploitable vulnerabilities" [37]. Fuzzers automate the generation of inputs which are used to dynamically test a program. This incurs low labor costs coupled with high test efficiency. Furthermore, inputs generated by fuzzers are unpredictable. Fuzzing technology is a typical application of the law of large numbers. With the massive progress of fuzzing test, it is bound to detect zero day vulnerabilities [38]–[40].

C. *Behavior of Fuzzers*

The core concept behind fuzzers has not changed over time. However, software fuzzers have been iterated many times, with new features being added. New fuzzing frameworks have a feedback scheme which drives a genetic algorithm for input generation. Figure 2 shows the traditional fuzzing test process.

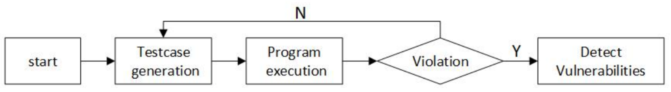

Figure 2: Work process of fuzz testing: the fuzzer will generate inputs until a violation occurs. The violation results in a crash that can be recorded and analyzed for potential vulnerabilities.

A Fuzz test starts with a set of generated program inputs. The testcases should, on the one hand, satisfy the required input format of the program as closely as possible, and on the other hand, bring the program close to failure. Once the input is generated, it is fed to the target program. Fuzzer automatically starts the target program and monitors the running status until it knows that the target program has stopped executing. After this, Fuzzer verifies the existence of violation by signals from the operating system. If there is, Fuzzer records the vulnerability found, if not, then another test is run.

Common instrumentation tools such as AddressSanitizer [41] and LeakSanitizer [42] provide extra safety checks on software at the cost of some performance overhead. When violation of a running program is detected by the instrumentation of these frameworks, a forcible program termination is issued.

D. *Types of Fuzzers*

Fuzzers can be classified by the method of testbench generation, the degree of target source code dependency, and the exploration strategy. A fuzzing framework may fall within one or more of these categories.

1) Input Generation: Fuzzers can be categorized by the way they generate inputs for the test program. A generation-based fuzzer utilizes entries in a configuration file as inputs. This method improves testing efficiency as the most common inputs can easily be provided to the program. Examples of this type of fuzzer include Peach [43], Trinity [44], Sulley [45], Csmith [46]. Unfortunately, the configuration file may not be exhaustive and may allow for vulnerabilities to go undetected.

Mutation-based fuzzers utilize a single input and will automatically mutate it on every round of testing. This type of fuzzer requires little knowledge of the program being tested, and can automatically generate new test inputs by using the used input as a seed. Examples of this type of fuzzer include American Fozzy Lop (AFL) [47], Driller [48], and Mayhem [49]. However, mutation-based fuzzers are difficult to apply to programs that can accept multiple different file types [50].

2) Target source code dependency: Fuzzers can also be classified by their dependency to the source code of the application in the test harness. White box fuzzers, such as Peach [43], require access to the source code of the program, gaining information through static analysis. Black-box fuzzers, such as LibFuzzer [51], perform fuzzing tests directly on executables, without access to the source code. Gray-box fuzzers such as AFL instrument the source code of the program being tested during compilation. After this point, the original source code is no longer used. AFL uses the instrumentation in the binary to monitor the execution of the program and generate a coverage map of the code being executed.

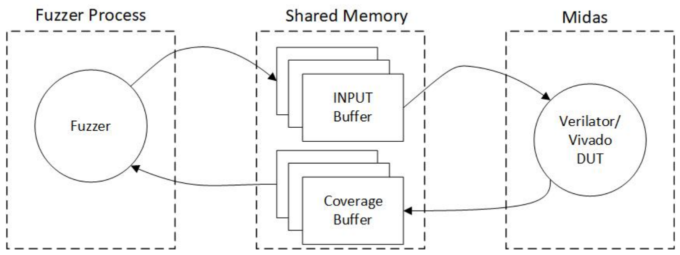

Figure 3: Structure of RFuzz [10]. RFuzz uses Midas as a bridge between the simulator and the fuzzer. A buffer is used to handle the communication between the simulator and fuzzer.

3) Exploration strategy: Fuzzers can also be classified as directed fuzzers or coverage-based fuzzers. Directed fuzzers aim to generate test cases that cover the target path of the program, with the expectation of faster testing of the program. Coverage-based fuzzers, on the other hand, aim to generate test cases that cover as many execution paths as feasible, expecting more exhaustive testing and maximizing the detection of bugs.

Table I: Comparison of testing techniques. Fuzzing has a low entry cost while yielding scalable results and high accuracy.

Technique	Deployment	Accuracy	Scalability
static analysis	easy	low	good
dynamic analysis	hard	high	unknown
fuzzing	easy	high	good

IV. Hardware Fuzzing

Recently, researchers have tried to apply the concepts behind software fuzzing into RTL code with the objective of finding flaws in hardware. We will now examine the two most prominent solutions: RFuzz [10] and HW-Fuzzing [9].

A. RFuzz

RFuzz [10] is a part of Flexible Internal Representation for RTL (FIRRTL) [52]. It is a project that aims to investigate the use of coverage-directed fuzzing for RTL pre-silicon testing. As shown in Figure 3, the structure is like a normal software fuzzing test. Inspired by American Fuzzy Lop (AFL) [47], multiplexer are chosen as a branch coverage point similar to edge coverage in software fuzzing. This choice is justified in that multiplexers feature very similar characteristics to branches in software. Branches in software determine the execution path potentially altering the value of variables, whereas multiplexers act on signals much of the same way. RFuzz uses Midas [53] as a back-end. Midas provides a performance and power evaluation platform which can use different HDL simulators to simulate the hardware design and an interface which allows the fuzzer engine to communicate with the unit under test. RFuzz can use Verilator [54] for emulation purposes or Xilinx Vivado [55] for FPGA acceleration. The latter is done in an attempt to solve the problem of slow simulation of large-scale hardware designs. RFuzz employs AFL as the fuzzing engine for input generation and mutation.

However, testing through the supervision of simulation software is not ideal. In the software fuzzing domain, processes are Device Under Test (DUT) and crashes are a result of the operating system signaling an improper action. Compared with the traditional software FUZZ, RFuzz is comparable to use a virtual machine hypervisor as the testing harness for a program. Much like the illegal actions in the program do not crash the hypervisor, the simulated hardware does not crash the simulator. Simulation software will steadily produce an output for any input given a DUT.

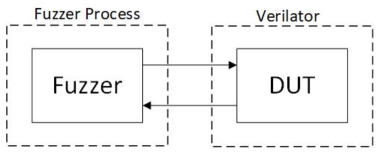

Figure 4: HW-Fuzzing model [9]. HW-Fuzzing uses Verilator as a target directly. Both fuzzer and simulator are running on the same platform so buffer is omitted.

Although the authors claim that RFuzz works close to the RT-level, choosing a multiplexer as the means of detecting coverage puts the translation closer to a netlist level. After RTL synthesis, much of the semantics of higher-level languages are lost. The recovery of RTL level semantics has proven to be an NP-Hard problem [56]. RFuzz employs FPGA-accelerated fuzzing in an attempt to speed up the fuzzing procedure. This may increase the cost of fuzzing design, since specialized hardware must be used. This further faces the issue that hardware structures do not necessarily map properly to an FPGA's fabric [57], which raises questions in the capabilities of the system. RFuzz is further limited by the language support in Midas due to latter's dependency on FIRRTL. This is currently restricted to Chisel, Verilog, and portions of SystemVerilog [58].

B. HW-Fuzzing

HW-Fuzzing [9] follows the RFuzz model while also exhibiting some of its shortcomings. As shown in Figure 4, with a nearly identical model, the authors make three major changes to the design. HW-Fuzzing narrows the application of the fuzzer engine to the unit under test only rather than the applying it to the entire test harness. Authors also modify the coverage statistical method to reduce tool runtime. Lastly, HW-Fuzzing employs predefined SystemVerilog assertions as simulation termination condition.

HW-Fuzzing proposed the idea that the test target of the fuzzer should not contain the simulator itself. However, HW-Fuzzing continues using Verilator as the tool and claims to acquire the equivalent model of the C++ language for Verilog code through this approach. Sadeghi [59] questions those approaches and leaves the problem of equivalence between the software model and the hardware model to be proven. The HW-Fuzzing team uses Verilator as an agent to compile SystemVerilog HDL into C++ code, which can then be built by standard C++ compilers. However, Verilator further requires the linking of additional auxiliary libraries as part of generating the final executable. This may be treated as an *emulated* model, rather than a *simulated* model, which brings back the question of accuracy, and whether sending and receiving data from the model is sufficient to determine fuzzer coverage.

Second, the group claims that the coverage of multiplexers lacks a proof of equivalence with the coverage of design verification. As such, they opt for using the default coverage statistics in AFL, namely that of the instrumentation on the source code of the simulation software during the compilation process. This is fine as long as the result of the conversion is an equivalent model. Verilator is a cycle-accurate emulator that does not guarantee the software model and the hardware model would yield the same state at any given moment [54]. Such requirements are not constraints for Verilator.

Lastly, since hardware models do not directly crash a simulator, authors introduce SystemVerilog assertions to the simulation treating them as a crash. This limits the testing scope available through HW-Fuzz as extra manual instrumentation is required. For instance, authors are only able to utilize OpenTitan [60] to test their proposed mechanism. Due to the nature of hardware and simulators, questions can be raised about the equivalence of a SystemVerilog assertion and a software crash. Moreover, existing tools such as Cadence JapserGold [61] or Synopsys VC Formal [62] can perform the same type of evaluation at coverage with similar or less required workload [63]. These tools, much like HW-Fuzz, may have a difficult time finding vulnerabilities in hardware because of the manual instrumentation required. Assertions will only trigger based on the particular condition they are checking. Testers are responsible for their creation.

V. Overcoming Limitations and Envisioning Hardware Fuzzers

If it is desired to build a hardware fuzzing system based on an existing fuzzer, the simulator should provide a feedback model similar to the crash model used by regular software fuzzers. In software testing, since the test environment is the same as the actual running environment, the fuzzer can determine the termination of software operation by an OS-generated process termination signal. The execution environment provides the means for the fuzzer to obtain the reason for the termination, such as crash and hang-off. In contrast, HDL simulators and emulator do not offer an equivalent termination for fuzzers to exploit: there is no crash or hang-off in the simulation utilized in RFuzz and HW-Fuzz. Even if a fuzzer detects the satisfiability of the predefined assertion, it checks a known flaw rather than explores an unknown threat.

Checking of the assertion at each cycle of the hardware simulation poses two problems. Unlike a software crash which indicates a failure at a specific point in the program, an assertion being triggered during simulation is a result of a series of events that occur in parallel. Although the assertion signals a failure in the design, it does *not* reveal the underlying cause of the detected problem. This takes us to the next issue. Some of these processes are the result of multi-cycle actions. For example, the reflection of the change in the architectural state on a CPU due to a single instruction may take several clock cycles, including those required for decoding, dispatching, execution, and committing of the effects. The process may be even longer if the instruction was incorrectly issued due to a misprediction, in which case any temporary changes to the microarchitecture must be rolled back. Current assertion mechanisms are unable to "follow" on the multi-cycle operation of sequential hardware, further limiting the types of checks that can be performed.

We consider that a proper hardware fuzzing mechanism must overcome these issues. It must be able to robustly handle multi-cycle computations, and be able to recognize the inherent differences between hardware and software. This way, any possible errors on the design and potential vulnerabilities can be readily detected. That is, we propose "fuzzing hardware as hardware".

VI. Conclusion

Fuzzing is becoming a promising field of research for detecting hardware bugs. Some concessions are usually made by hardware fuzzers due to limitations in simulation and emulation models. This paper summarizes and analyzes the drawbacks of existing typical hardware fuzzing approaches RFuzz and HW-Fuzz. Research trends are given as handling multi-cycle computations and eliminating the inherent differences between hardware and software.

Acknowledgments

Portions of this work were supported by the National Science Foundation through the FMitF Project (CNS-2019310) and the PPoSS Planning Project (NSF-2028910).

References

[1] 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, 2017.

[2] D. Price, "Pentium fdiv flaw-lessons learned," *IEEE Micro*, vol. 15, no. 2, pp. 86–88, 1995.

[3] R. R. Collins, "The pentium `f00f` bug," 1997.

[4] "Shift-left, right." [Online]. Available: "https://www.acuerdo.co.uk/shift-left-right"

[5] X. Guo, R. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, June 2015, pp. 1–6.

[6] S. Drzevitzky, "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration," in *International Conference on Field Programmable Logic and Applications*, 2010, pp. 255–258.

[7] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate-abstraction and refinement techniques for verifying rtl verilog," *IEEE TCAD*, vol. 27, no. 2, pp. 366–379, 2008.

[8] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Automatic code converter enhanced pch framework for soc trust verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, pp. 3390–3400, 2017.

[9] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," *arXiv preprint arXiv:2102.02308*, 2021.

[10] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[11] X. Guo, R. G. Dutta, J. He, and Y. Jin, "Pch framework for ip runtime security verification," in *2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2017, pp. 79–84.

[12] J. He, X. Guo, T. Meade, R. G. Dutta, Y. Zhao, and Y. Jin, "Soc interconnection protection through formal verification," *Integration*, vol. 64, pp. 143–151, 2019.

[13] X. Guo, R. G. Dutta, and Y. Jin, “Eliminating the hardware-software boundary: A proof-carrying approach for trust evaluation on computer systems,” *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 2, pp. 405–417, 2017.

[14] F. Farahmandi, Y. Huang, and P. Mishra, “Trojan localization using symbolic algebra,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 591–597.

[15] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, “Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution,” in *2018 IEEE International Test Conference (ITC)*. IEEE, 2018, pp. 1–10.

[16] W. Hu, A. Ardeshiricham, and R. Kastner, “Hardware information flow tracking,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.

[17] INRIA, “The coq proof assistant,” 2010, `http://coq.inria.fr/`.

[18] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[19] G. C. Necula, “Proof-carrying code,” in *POPL ’97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 106–119.

[20] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, “Symbolic model checking,” in *International conference on computer aided verification*. Springer, 1996, pp. 419–422.

[21] F. Towhidi, A. H. Lashkari, and R. S. Hosseini, “Binary decision diagram (bdd),” in *2009 International Conference on Future Computer and Communication*. IEEE, 2009, pp. 496–499.

[22] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Advances in computers*, vol. 58, pp. 117–148, 2003.

[23] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[24] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, “Caisson: a hardware description language for secure information flow,” in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 109–120.

[25] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, “Sapper: A language for hardware-level security policy enforcement,” in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 97–112.

[26] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, “A hardware design language for timing-sensitive information-flow security,” *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[27] X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin, “Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment,” in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019, pp. 91–100.

[28] B. A. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. W. R. Marsh, “Industrial perspective on static analysis,” *Software Engineering Journal*, vol. 10, no. 2, pp. 69–75, 1995.

[29] B. G. Ryder, “The pfort verifier,” *Software: Practice and Experience*, vol. 4, no. 4, pp. 359–377, 1974.

[30] B. Chess and G. McGraw, “Static analysis for security,” *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[31] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[32] M. Dwyer and D. Schmidt, “Limiting state explosion with filter-based refinement,” in *Proceedings of the 1st International Workshop on Verification, Abstract Interpretation and Model Checking*. Citeseer, 1997.

[33] M. Kusano and C. Wang, “Thread-modular static analysis for relaxed memory models,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 337–348.

[34] C. E. Silva and J. C. Campos, “Combining static and dynamic analysis for the reverse engineering of web applications,” in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, 2013, pp. 107–112.

[35] A. Bhat and S. Quadri, “Equivalence class partitioning and boundary value analysis-a review,” in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 2015, pp. 1557–1562.

[36] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[37] P. Oehlert, “Violating assumptions with fuzzing,” *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[38] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning, “Experiences with model inference assisted fuzzing.” *WOOT*, vol. 2, pp. 1–2, 2008.

[39] J. Yan, Y. Zhang, and D. Yang, “Structurized grammar-based fuzz testing for programs with highly structured inputs,” *Security and Communication Networks*, vol. 6, no. 11, pp. 1319–1330, 2013.

[40] N. Palsetia, G. Deepa, F. A. Khan, P. S. Thilagam, and A. R. Pais, “Securing native xml database-driven web applications from xquery injection vulnerabilities,” *Journal of Systems and Software*, vol. 122, pp. 93–109, 2016.

[41] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Address-sanitizer: A fast address sanity checker,” in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[42] C. Team, “Leaksanitizer-clang 12 documentation,” 2020.

[43] M. Eddington, “Peach fuzzer: Discover unknown vulnerabilities.”

[44] D. Jones, T. Rantala, and V. Weaver, “Trinity: A linux system call fuzz tester,” 2016.

[45] P. Amini, “Openrce/sulley,” *original-date*, vol. 28, p. 57Z, 2012.

[46] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, June 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[47] M. Zalewski, “american fuzzy lop (2.52 b),” *Retrieved April*, vol. 10, p. 2020, 2019.

[48] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.” in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[49] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[50] C. Miller, Z. N. Peterson, *et al.*, “Analysis of mutation and generation-based fuzzing,” *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[51] “Libfuzzer: A library for coverage-guided fuzz testing,” Feb 2019. [Online]. Available: http://llvm.org/docs/LibFuzzer.html.(2019)

[52] P. S. Li, A. M. Izraelevitz, and J. Bachrach, “Specification for the firrtl language,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html

[53] D. Kim, C. Celio, D. Biancolin, J. Bachrach, and K. Asanovic, “Evaluation of risc-v rtl with fpga-accelerated simulation,” in *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[54] W. Snyder, “Verilator: Open simulation-growing up,” *DVClub Bristol*, 2013.

[55] Xilinx, “Vivado.” [Online]. Available: ”https://www.xilinx.com/products/design-tools/vivado.html”

[56] T. Meade, K. Shamsi, T. Le, J. Di, S. Zhang, and Y. Jin, “The old frontier of reverse engineering: Netlist partitioning,” *Journal of Hardware and Systems Security*, vol. 2, no. 3, pp. 201–213, 2018.

[57] H. Wong, V. Betz, and J. Rose, “Quantifying the gap between fpga and custom cmos to aid microarchitectural design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2067–2080, 2013.

[58] C. Alliance, “Flexible internal representation for rtl,” https://github.com/chipsalliance/firrtl, 2021.

[59] A.-R. Sadeghi, J. Rajendran, and R. Kande, “Organizing the world’s largest hardware security competition: Challenges, opportunities, and lessons learned,” in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 95–100.

[60] “Opentitan: Secure boot SoC design,” https://opentitan.org/, 2019.

[61] Cadence, “Jaspergold.” [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[62] Synopsys, “Vc formal.” [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html

[63] OpenTitan, “Dv simulation summary results, with coverage (nightly).” [Online]. Available: https://reports.opentitan.org/hw/top_earlgrey/dv/summary.html