

HeisenTrojans: They Are Not There Until They Are Triggered

Akshita Reddy Mavurapu
University of New Hampshire
akshitareddy.mavurapu@unh.edu

Haoqi Shan
Certik
haoqi.shan@certik.com

Xiaolong Guo
University of Kansas
guoxiaolong@ksu.edu

Orlando Arias
University of Massachusetts, Lowell
orlando_arias@uml.edu

Dean Sullivan
University of New Hampshire
dean.sullivan@unh.edu

Abstract—The hardware security community has made significant advances in detecting Hardware Trojan vulnerabilities using software fuzzing-inspired automated analysis. However, the Electronic Design Automation (EDA) code base itself remains under-examined by the same techniques. Our experiments in fuzzing EDA tools demonstrate that, indeed, they are prone to software bugs. As a consequence, this paper unveils HeisenTrojan attacks, a new hardware attack that does not generate harmful hardware, but rather, exploits software vulnerabilities in the EDA tools themselves. A key feature of HeisenTrojan attacks is that they are capable of deploying a malicious payload on the system hosting the EDA tools without triggering verification tools because HeisenTrojan attacks *do not* rely on superfluous or malicious hardware that would otherwise be noticeable. The aim of a HeisenTrojan attack is to execute arbitrary code on the system on which the vulnerable EDA tool is hosted, thereby establishing a permanent presence and providing a beachhead for intrusion into that system. Our analysis reveals 83% of the EDA tools analyzed have exploitable bugs. In what follows, we demonstrate an end-to-end attack and provide analysis on the existing capabilities of fuzzers to find HeisenTrojan attacks in order to emphasize their practicality and the need to secure EDA tools against them.

I. INTRODUCTION

Recently, significant effort in the hardware security community has been paid to the automated analysis of Hardware Trojan vulnerability detection [1], [2] by borrowing concepts from software fuzzing. Less attention, however, has been paid in applying those same concepts to the analysis of the EDA code base itself. EDA tools are complex and sophisticated pieces of software comprising millions of lines of code (MLoC) and heavily used in the community. Moreover, there is a clear correlation between the number of lines of code, third-party library reliance, and number of users for a tool on the one hand and the number of errors and reported vulnerabilities on the other [3], [4]. This insight is, in part, what led us to ask if *can we expect that the EDA tool itself is bug free*.

Our results indicate that, similar to other existing complex code bases, EDA tools contain buggy code. We evaluate a representative set of common EDA tools and found an exploitable bug in 83% of tools analyzed, other instances of bugs that may be exploitable given a motivated attacker with enough time, and still other instances of bugs that, while not fully exploitable for a full system compromise, can be used as means of rendering the tool unusable (Denial of Service).

In this paper we present a new class of hardware attacks called **HeisenTrojans**. To begin, we consider these hardware attacks because the malicious input is embedded in hardware-related components such as a hardware description language (HDL), simulation test vectors, and waveforms. As such, we are using “hardware” in the broad sense to include anything typically involved in the EDA toolchain and hardware manufacturing life cycle. While we say throughout the paper that, for instance, the HDL in a HeisenTrojan is malicious, it is not inherently so as it *does not* generate any

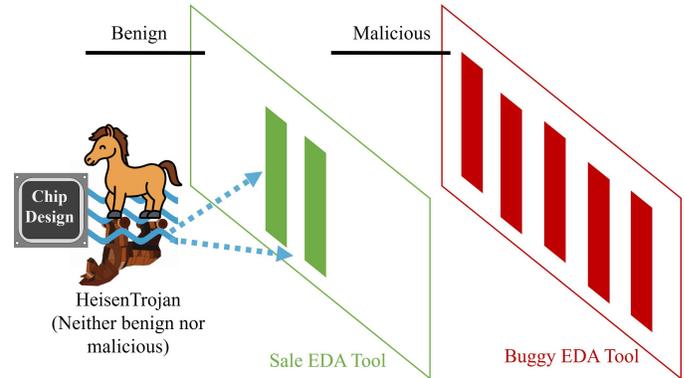


Fig. 1: Borrowing from the famous double slit experiment, a HeisenTrojan projected onto an EDA tool exhibits benign properties if the EDA tool is safe, but is malicious if the EDA tool is vulnerable.

malicious or superfluous hardware. Instead, as shown in Figure 1, a HeisenTrojan aims to exploit vulnerabilities only on buggy EDA tools to deploy a malicious payload in the system running the tool in order to compromise it. This duality property is the inspiration behind the name.

The goal of a HeisenTrojan attack is to gain arbitrary code execution on the machine hosting the vulnerable EDA tool. This, in itself, typically the first step for an attacker who wishes to obtain persistence as in a supply-chain attack [5]. For ethical considerations, in this work we do not show the precise steps to achieve a full system compromise. We do, however, use this as justification to show how a HeisenTrojan can be a significant threat to organizations and companies by presenting how a full HeisenTrojan-based attack can be developed. Further, we consider HeisenTrojans relevant for two additional reasons.

First, HeisenTrojan attacks represent a new attack vector to the hardware security community. We do not claim that triggering software vulnerabilities to gain arbitrary code execution is new, but rather, that targeting software vulnerabilities in EDA tools via maliciously crafted inputs is new. Concern in the hardware security community is typically focused on preventing and/or detecting maliciously crafted HDL that attempts to embed a traditional Hardware Trojan. Little attention, if any, has been paid to the EDA tools themselves¹. We show that this should be a concern. Further, it is an unexplored attack surface by virtue of its newness. To the best of our knowledge, this is the first paper to report an end-to-end attack targeting an EDA tool.

¹Recent work explored attacks stemming from malicious EDA tools but focused on their ability, in such a scenario, to surreptitiously embed hardware Trojans [6]. More will be said regarding this, and other, related works in Section III.

Second, HeisenTrojans can be practically exploited in a variety of relevant scenarios. We will present the details of an end-to-end attack in Section IV-B. As shown in Figure 2, at a high-level, we reason that an attacker can leverage the HDL design, development, and integration process. The first relies on the complexity of HDL designs. For instance, we found we can hide the offending input in a large design, with MLoC, that will unlikely be analyzed line-by-line. Furthermore, by their nature, HeisenTrojans do not generate malicious or superfluous hardware and thus go undetected by verification tools. The second scenario makes use of the need to guard intellectual property (IP) by releasing it to third-party vendors [7]. In such a scenario, it is trivial to embed a maliciously crafted input which can exploit vulnerabilities in EDA tools. The third scenario exploits shared computing environments. FPGA resources are currently scant and expensive. It is not uncommon to turn to cloud providers who offer/lease relatively cheap FPGA resources to users in a hosted environment as an alternative. This allows the attack to arbitrarily upload² malicious input to a cloud-hosted EDA tool.

A. Motivation

We aim to address several motivating points in introducing HeisenTrojan attacks that we hope will further aid its understanding:

How did we discover HeisenTrojan attacks? We were inspired by recent efforts that have demonstrated significant results from the automated analysis of HDL using software fuzzing techniques. However, it is not our aim to improve those results, but rather, to query the EDA tools themselves by treating them purely as pieces of software. For instance, an EDA tool accepts HDL as input for synthesis and a combination of HDL and test vector input during simulation. In each case, we are interested in finding vulnerabilities in the way the EDA tool handles those inputs when guided by a mutational fuzzer.

Why do we target synthesis, simulation, and scripting? Generally, EDA tools take an HDL design and perform synthesis and place-and-route to generate a bitstream that can then be used to configure an FPGA, or alternatively target a technology library to generate an ASIC design. Simulation is performed prior to place-and-route and uses a testbench as a wrapper to handle the flow of input and output to/from the design under test. These two cases allow us to mutate input either in the form of synthesizable HDL or as a test vector input. We also target traditional EDA tool scripting languages because they often act as a front-end to automate the design flow from synthesis to place-and-route, or other types of analysis.

B. Contributions

To summarize, our contributions include:

- We present HeisenTrojan attacks, a unique hardware attack that is benign if the EDA tool is bug-free but malicious if it is not.
- We highlight the practicality of HeisenTrojan attacks by presenting an end-to-end exploit for an EDA tool.
- We present analysis of the effectiveness of fuzzing EDA tools in the discovery of HeisenTrojan attacks.

In what follows, we will first introduce relevant background in Section II before discussing related works in Section III. We then present an end-to-end HeisenTrojan in Section IV. Analysis of the use of fuzzing to find HeisenTrojans is presented in Section V prior to a discussion of future work and conclusions in Section VI.

²Within reason and as prescribed by the cloud vendors rules [8].

II. BACKGROUND

A. EDA Tools

We evaluate 6 EDA tools in total including two synthesis tools (*yosys* [9], *abc* [10]), three simulation tools (*iverilog* [11], *verilator* [12], and *gtkwave* [13]), and a formal analysis tool (*z3* [14]). It is assumed that the readers are familiar with these tools, so we forego a detailed explanation of their functionality and internals. Instead, we provide a high-level overview of them in Table I and refer the reader to their source for further information.

TABLE I: Open-source EDA tools analyzed in this work.

EDA Tool	Description
<i>iverilog</i>	A verilog HDL compiler for the IEEE-1364 standard
<i>gtkwave</i>	A GTK+ based wave viewer
<i>yosys</i>	Open source synthesis suite for Verilog
<i>abc</i>	A synthesis and verification suite
<i>z3</i>	An SMT solver developed by Microsoft
<i>verilator</i>	A verilog HDL simulator

We focus our evaluation on open source EDA tools because they are easily obtained at no cost and are supportive of bug reporting. We have reported our results to the tool maintainers. However, they are still in the patching process so, in this paper, we limit our discussion to number of bugs found and type. We remove all details of the EDA tool, insofar as possible, in the end-to-end exploit to abide with responsible disclosure procedures.

B. Fuzzing

Fuzzing is one of the most successfully employed techniques for software bug discovery [15] and currently an active area of research in both the hardware and software communities. Fuzzers are usually categorized as black-box, white-box, or grey-box depending on the amount of information they have of the underlying program. A black-box fuzzer knows nothing about the internal structure of the program, whereas a white-box fuzzer knows everything about the program's internal structure. A grey-box fuzzer sits in-between the two in that it has limited knowledge of the internals of the program behavior via coverage-guided feedback provided by some form of program instrumentation.

In this paper, we rely heavily on coverage-guided fuzzing. Briefly, the fundamental steps taken in coverage guided fuzzing include:

- 1) **Instrumentation:** The program being fuzzed is augmented with code to record its control-flow during execution.
- 2) **Seed pool Generation:** The fuzzer generates candidate inputs for the program, which can either be user supplied or randomly generated. The correct generation of an initial seed pool is an active area of research [16].
- 3) **Input mutation:** After each round, selected inputs that exhibit unique code coverage are mutated (e.g. addition, deletion of bytes or flipping, rotating bits among several others).
- 4) **Coverage-guided feedback:** Code coverage information is recorded as the seed inputs are executed. This information is binned, ordered, and then selected (e.g. best/new code coverage) to generate the new input corpus for the next fuzzing round.

Using open-source tools allows us to use gray-box fuzzers with little effort. Closed-source EDA tools require the use of black-box fuzzers which are slow and unreliable. Since our objective is to demonstrate the dangers of HeisenTrojans we believe this is a fair compromise.

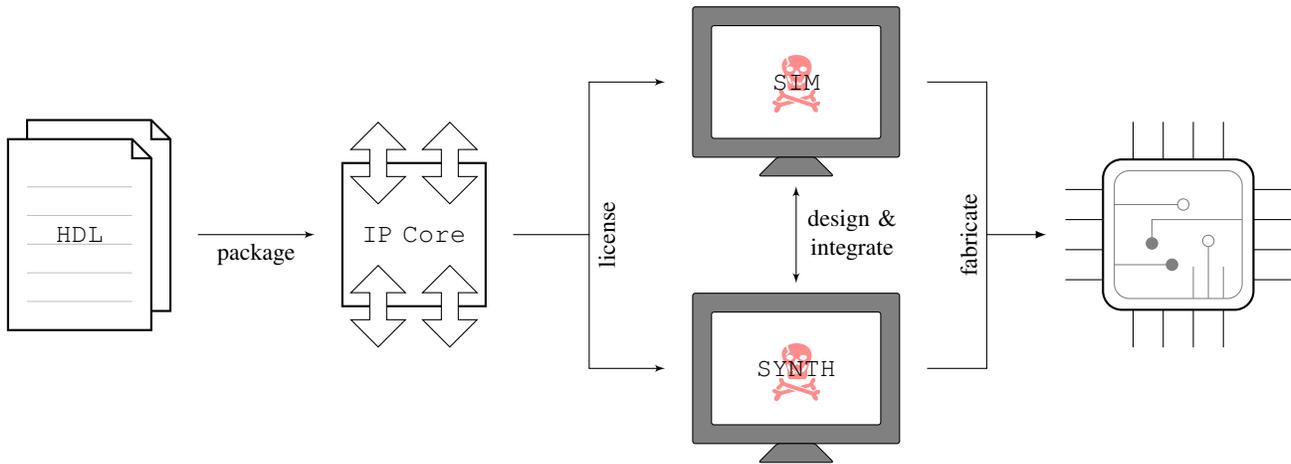


Fig. 2: The HeisenTrojan makes use of standard HDL workflows in the industry. Malicious HDL is packed as part of an IP core. This HDL does *not* aim to generate malicious hardware. Instead, it triggers and exploits bugs in an HDL synthesis or simulation tool in order to compromise the system where the tool runs. If fabricated, ICs including HeisenTrojan affected IP cores do not exhibit malicious behavior.

III. RELATED WORKS

We are unaware of any existing effort to fuzz EDA tools and craft HeisenTrojan attacks. Project F4PGA [17], [18] aims to reverse engineer bitstreams for Lattice and Xilinx FPGAs using what amounts to fuzzing, but has a completely different goal than our work. However, we are not claiming complete ingenuity. Analysis of software using a fuzzer is common-place and likely performed in-house by EDA tool vendors, or at the very least during the development process for reporting and fixing bugs prior to release. With that said, we are only aware of one bug report [19], but there are probably many that go unreported publicly.

Recent research [20] investigates logic synthesis tools for the correctness of their output via equivalence checking. The research is constructed around a tool, Verismith, that generates semantically correct and deterministic Verilog allowing for comparison between the generated design and its synthesized netlist. If they differ, then a bug is registered. A similar analysis is performed for high-level synthesis tools [21]. Our work, however, is not interested in the correctness of output for a given logic synthesis tool, but rather, whether a given EDA tool contains bugs that can triggered via specially crafted input to gain control of the system. We show that many such bugs can indeed be found for a variety of commonly used tools (synthesis, simulation, verification) and present an end-to-end exploit to demonstrate their impact.

An orthogonal line of research was proposed in [6], [22] that investigates the attack surface of malicious EDA tools. The authors demonstrate the potential for attacks by, for instance, making minor modifications to the intermediate files generated by the EDA toolchain. This research highlights and provides insight into the vulnerabilities that can be exploited by attackers if the EDA toolchain is compromised. An interesting approach may be to combine HeisenTrojan attacks with this line of research to first gain control of the EDA tool in order to then embed a traditional Hardware Trojan.

IV. BUILDING A HEISENTROJAN

In what follows, we first introduce our adversarial model. We then discuss the steps taken to build a full end-to-end HeisenTrojan attack chain on an EDA tool. Due to page limitations, we are limited to discussing a single attack against a given EDA tool. However, we found 12 exploitable bugs in total, see Table II, in which we are able to craft end-to-end exploits across the tools outlined.

A. Adversarial Model and Assumptions

HeisenTrojan introduces a new class of hardware attacks where an adversary can ship an IP core written in any given HDL, a simulation test vector, or script for tool management that contains a specially crafted payload. However, unlike traditional Hardware Trojans the bundle in question *does not* generate any malicious hardware. Instead, the adversary's goal is to *compromise the computing system of the licensee*. This may be done with the purpose of spying, sabotaging operations, or stealing data. The HeisenTrojan shipped as part of the IP core exploits a vulnerability in the tooling utilized by the victim to establish a permanent presence in their computing system. When processed by a non-vulnerable tool, the HeisenTrojan is not triggered nor does it generate extraneous or superfluous hardware, thereby remaining hidden from detection.

B. Finding an Exploitable Bug

We examined the results of our fuzzing campaign on open-source EDA tools until we found a crash that was the result of an error that could be readily exploited. Suitability for exploitation was determined by examining the crash and performing a two factor test:

- whether the behavior causing the crash is controllable in ways that do not trigger a denial of service; and
- whether the behavior causing the crash allows for the corruption of memory areas containing code pointers.

A suitable vulnerability was found in tool *REDACTED*³. The vulnerability allows us to perform a write anywhere in the program's stack, which is conducive to a type of attack under the umbrella of code-reuse attacks called *return-oriented programming* (ROP) [23], [24].

C. Preparing, Deploying, and Synthesizing the HDL

For purposes of deployment the HDL can be wrapped in IEEE 1735-2023 [25] but this is not strictly necessary. The goal is *not* to hide a traditional Hardware Trojan which generates malicious logic, but to deploy HDL which generates legitimate hardware. The HeisenTrojan payload aims to exploit bugs on the EDA toolchain being used to compromise the victim's infrastructure. As such, for

³We have followed responsible disclosure procedures for all vulnerabilities found and reported them to the respective maintainers. The tool name is omitted to comply with procedures.

our purposes, the HDL can be sent in plain-text, as long as the portion of the HDL which triggers the bug in the HDL tool is innocuous.

Listing 1: Simplified view of the vulnerability.

```
char buffer[1024], * p = buffer;
va_list ap;
/* ... */
p += vsnprintf(p, buffer+sizeof(buffer)-p, fmt, ap); /* ❶ */
p += snprintf(p, buffer+sizeof(buffer)-p, "\n"); /* ❷ */
```

We make use of a write anywhere vulnerability we found in the implementation of the `f_REDACTED_r()` function of one of the synthesis tools we tested. By crafting a specific payload and exploiting this vulnerability we are able to overwrite a return address in the program’s stack, thereby allowing us to perform a control-flow attack. Through repeated usages of this vulnerability, we craft a payload which results in a chain of gadgets that achieve our desired result. In our case, we wish to print an innocent message in the terminal to signify a successful attack.

When synthesized, the HDL in question makes use of one of the vulnerabilities presented in Section V, Table II. A simplified version of the bug is shown in Listing 1. The code in question attempts to perform safe string concatenation through the use of the `snprintf()` family of functions. The next available location in the buffer is found by advancing the pointer `p` by the return of the `vsnprintf()` function in the line labeled ❶. However, it is imperative to notice that these functions *do not* return the value of characters added to the buffer, but the number of characters that *would have been added* were there enough room in the buffer. As such, on the call to `vsnprintf()` in ❷ the pointer may point to an address outside the bounds of the buffer. This gives us a *spatial* memory error which can be exploited to perform a write to a location of memory.

The spatial memory error in question allows us to overwrite any byte in memory to have a value of `0x00`. Because the buffer in question is allocated in the stack of a function automatic storage variables and any code pointers stored in the stack, such as return addresses, are prime targets. Since the vulnerability in question allows us to freely move a pointer to any address in the stack area, we can safely bypass commonly deployed defenses such as stack protection [26] while constructing the desired ROP-chain. This allows us to gain arbitrary code execution by chaining gadgets (e.g. small instruction sequences) together from the existing code base.

D. A Word about Full System Compromise

We show how we can achieve arbitrary code execution by exploiting bugs in the synthesis environment. A system compromise relies on a payload which can cause permanent changes to the OS introducing malware. Achieving this goal further requires escalation of privileges through a kernel vulnerability. The latter are common [27], [28], [29], [30] and do not require our payload to directly exploit them. Our payload can simply launch another application (such as through the `execve()` family of functions) which can more readily exploit such vulnerabilities.

V. ON FUZZING EDA TOOLS

In what follows, we present a discussion on how we found bugs on each tool we examined as well as a description of the process and tooling used. We summarize the bugs found in Table II as well as present an analysis of coverage achieved.

TABLE II: Number, type, and exploitability of bugs per EDA tool.

EDA Tool	Bugs [†]	Types	Vulnerable [‡]
Z3	1	null pointer dereference	×
GTKWave	7	heap overflow null pointer dereference	✓ ×
verilator	3	stack overflow null pointer dereference	✓ ×
iverilog	11	stack overflow null pointer dereference	✓ ×
ABC	9	null pointer dereference	×
Yosys	6	heap overflow stack overflow null pointer dereference	✓ ✓ ×
Total	37		12

[†] Number listed corresponds to unique bugs.

[‡] Indicates if a bug creates a vulnerability which can be used to perform an attack other than a Denial of Service.

A. Experimental Tools

We used `honggfuzz` [31] to fuzz each EDA tool due to its comparative performance in recent literature [32], [33]. `Honggfuzz` is a multi-threaded, grey-box coverage-guided fuzzer. It takes input corpora, checks them for new coverage, and then feeds the files to a simple in-memory corpus directory. It utilizes randomly chosen inputs from this memory and mutates them to start fuzzing.

We used `llvm-cov` [34] to generate coverage information for each EDA tool. The tools are instrumented to emit profile and coverage information by compiling them with `clang` using the `-fprofile-instr-generate` and `-fcoverage-mapping` flags. Coverage information is generated by running the instrumented EDA tool normally. A raw profile file is created that is then converted using the `llvm-profdata merge` tool. Finally, a JSON-formatted file is exported to collect the profiled edge coverage during the measured interval.

We used a combination of `AFLTriage` [35] and `afl-tmin` [36] to determine the uniqueness of the bugs found. `AFLTriage` is first used to minimize the number of crashing inputs by performing crash deduplication using a heuristic that selects between file and line number and the address of the first interesting stack frame. This approach is imperfect and may lead to missing unique crashes (false negatives) but avoids labelling aliasing crashes as unique (false positives). `afl-tmin` is then used to minimize a crashing input to its bare minimum (in terms of bytes). This allows us to more easily determine how to control the bug by modifying the minimized input.

B. Experiment Setup

We instrumented the source code for each EDA tool, used a trivial input seed (e.g. an appropriate “hello, world” for the tool), and applied flags where necessary to ensure proper functionality of the tool with the fuzzer (e.g. disabling `gtkwave`’s GUI) while fuzzing. Each EDA tools was evaluated 5 times for a 24 hour period on a 24 core Intel i7-12850HX with 32 GiB of memory running Ubuntu 22.04.2 LTS for a total of 20,000 compute hours per recommendations [33].

C. Unique Crashes and Crash Types

Table II shows the number of unique crashes, crash types, and exploitability of the crash for each EDA tool. In total, we found 37 unique crashes. Limited manual analysis was used to determine that 12 of these bugs were exploitable from userland. This does not mean

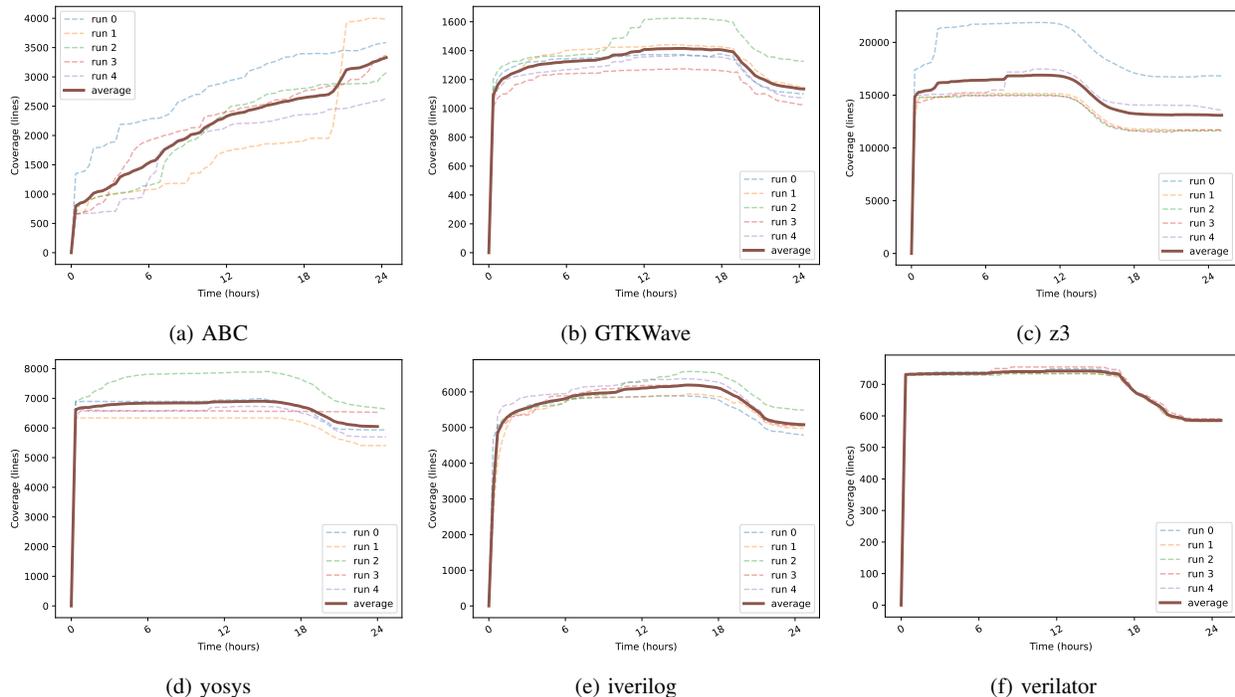


Fig. 3: Coverage plots for evaluated EDA tools.

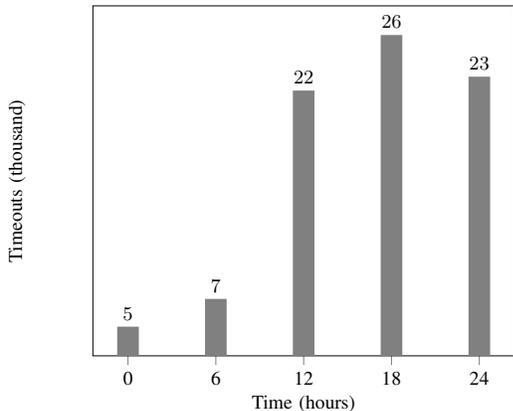


Fig. 4: Number of timeouts per fuzzing interval for z3

that the remaining bugs are not exploitable, but rather they were not immediately exploitable given our analysis. A motivated attacker under the correct conditions could exploit the bugs.

We categorized the bugs broadly as either null-pointer dereferences, heap, or stack overflows. With respect to the latter two classes, we made no distinction between a out-of-bounds access or an overflow for clarity. We found, however, that the heap overflows were typically write-anywhere vulnerabilities that crashed while accessing invalid memory. Of these, we found that 62% were controllable - we could read/write to an arbitrary location in mapped memory.

D. Evaluation of Coverage

Coverage plots for each EDA tool are provided in Figure 3. It is apparent from these plots that, in the majority of cases, the fuzzing campaign shows diminishing returns as it progresses in time. This can be explained, in part, as a side-effect of the method of logging coverage information. After each logging interval, the saved corpora in the default directory used by the fuzzer are removed and saved in an accumulated directory of corpora from prior intervals. This

was done to avoid analyzing prior corpora for coverage information and, thereby, save time. However, we also reason that the fall-off in coverage over time is due to the highly structured input requirements for EDA tools. The initial seed, while trivially constructed for our evaluation, represents a valid starting point for the fuzzer. This can be seen by the immediate increase in line coverage. However, as time elapses, that input becomes less structured under mutation by the fuzzer. We observed that those programs which show diminishing returns experience a concomitant increase in number of timeouts. This effectively prevents the fuzzer from making forward progress during each interval of coverage evaluation.

Only one tool, `abc`, shows steady growth. Others show almost no growth after initialization. The `z3` SMT solver performs worst in that it shows diminishing returns after only 12 hours of fuzzing. This makes sense considering its strict input requirements. We plot the number of timeouts over time for `z3` as a topical explanation for these results in Figure 4. Notice that the number of timeout increases at 12 hours. We observed a similar trend for the other EDA tools too.

TABLE III: Percent line coverage achieved while fuzzing compared to total line coverage.

EDA Tool	12h (%)	24h (%)
iverilog	21	17
GTKWave	15	13
Yosys	11	9
ABC	15	23
Z3	13	9
verilator	8	5

We also show the achieved line coverage compared to total line coverage as a percentage in Table III. In most cases, our results indicate that the fuzzer was only capable of shallow analysis. Improving the seed corpus would perhaps improve these results per discussions in related work [16]. However, a more obvious, yet more complicated, solution would be to develop a custom interface capable of handling

highly-structured input for the fuzzer.

VI. FUTURE WORK AND CONCLUSIONS

There is a clear trend between a tool's code base and complexity of its user ecosystem, and the number of reported memory vulnerabilities for that tool [3], [4]. HeisenTrojan attacks exploit this trend to launch end-to-end attacks against common EDA tools. Therefore, in order to defend against them, an EDA tool vendor could eliminate all memory vulnerabilities in its code base. Research, however, suggests that its unlikely memory corruption can be completely eliminated from runtime efficient languages without paying a severe performance loss [37] or switching entirely to a memory safe language [38]. The latter is unlikely, though gaining traction, and the former untenable. Another mitigation strategy against HeisenTrojan attacks includes code-reuse mitigations. However, these too are often imperfect or negatively impact performance [39].

Another promising strategy would be to incorporate fuzzing into the EDA tooling ecosystem. As Section V demonstrated, current fuzzing infrastructure is ill-suited to handle the highly structured input requirements for the common EDA tools. An interesting area of research, partially explored in [20], includes the creation of domain-specific front-ends capable of outputting correctly formatted data. This would provide a sane mutational block with which the fuzzer can work. This would not only resolve the diminishing returns observed during our experimentation, but also allow deeper exploration of the code base with respect to coverage.

To conclude, in this paper we introduced a new class of Hardware Trojan which we call HeisenTrojans. Unlike traditional hardware attacks, HeisenTrojans do not generate spurious or malicious hardware but aim to exploit vulnerabilities in EDA tools to infect and establish a presence on computer systems. We caution that HeisenTrojans can be used as vectors to enable spying, sabotaging, or stealing data of a potential rival or organization by attacking their computing infrastructure. We further showcase how, with some effort, HeisenTrojans can be developed to target a wide array of EDA tools. Although in this work we focus on open-source tools, commercial closed-source tools are likely not bug-free and vulnerable to the same types of attack. We hope that this work raises awareness of this attack vector and jolts the industry into better securing their tools.

REFERENCES

- [1] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3219–3236.
- [2] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254.
- [3] C. Woody, R. Ellison, and W. Nichols, "Predicting software assurance using quality and reliability measures," *CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST*, Tech. Rep., 2014.
- [4] "Beware the browser within," <https://www.infoworld.com/article/2672106/beware-the-browser-within.html>, accessed: 2022-10-23.
- [5] A. Cao and B. Dolan-Gavitt, "What the fork? finding and analyzing malware in github forks," in *Proceedings of the NDSS*, vol. 22, 2022.
- [6] S. Sunkavilli, Z. Zhang, and Q. Yu, "Analysis of attack surfaces and practical attack examples in open source fpga cad tools," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2021, pp. 504–509.
- [7] J. Speith, F. Schweins, M. Ender, M. Fyrbiak, A. May, and C. Paar, "How not to protect your ip—an industry-wide break of ieee 1735 implementations," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1656–1671.
- [8] Amazon, "Overview of aws ec2 fpga development kit," <https://github.com/aws/aws-fpga>, 2023.
- [9] C. Wolf, "Yosys open synthesis suite," 2016.
- [10] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.
- [11] S. Williams and M. Baxter, "Icarus verilog: open-source verilog more than a year later," *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.
- [12] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.
- [13] T. Bybell, "Gtkwave electronic waveform viewer," 2010.
- [14] L. De Moura and N. Björner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [15] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.
- [16] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.
- [17] K. E. Murray, M. A. Elgammal, V. Betz, T. Ansell, K. Rothman, and A. Comodi, "Symbiflow and vpr: An open-source design flow for commercial and novel fpgas," *IEEE Micro*, vol. 40, no. 4, pp. 49–57, 2020.
- [18] S. Kashif, T. Ahmed, M. Ismail, and F. A. Karim, "Chipshop: A cloud based gui for accelerating soc design."
- [19] "Unsigned bit extension in if statement," June 2019. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006iHwwSSAS/vivado-20191-unsigned-bit-extension-in-if-statement?language=en_US
- [20] Y. Herklotz and J. Wickerson, "Finding and understanding bugs in fpga synthesis tools," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 277–287.
- [21] Y. Herklotz, Z. Du, N. Ramanathan, and J. Wickerson, "An empirical study of the reliability of high-level synthesis tools," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 219–223.
- [22] S. Sunkavilli, Z. Zhang, and Q. Yu, "Fpga security: Security threats from untrusted fpga cad toolchain," in *Frontiers of Quality Electronic Design (QED) AI, IoT and Hardware Security*. Springer, 2023, pp. 551–573.
- [23] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.
- [24] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.
- [25] "Ieee approved draft recommended practice for encryption and management of electronic design intellectual property (ip)," *IEEE P1735/D2*, May 2023, pp. 1–100, 2023.
- [26] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*, vol. 1, 2003.
- [27] "CVE-2023-35788." Available from MITRE, CVE-ID CVE-2023-35788., June 2023. [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-35788>
- [28] "CVE-2023-3812." Available from MITRE, CVE-ID CVE-2023-3812., July 2023. [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-3812>
- [29] "CVE-2023-4004." Available from MITRE, CVE-ID CVE-2023-4004., July 2023. [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-4004>
- [30] "CVE-2023-4147." Available from MITRE, CVE-ID CVE-2023-4147., August 2023. [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-4147>
- [31] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," <https://github.com/google/honggfuzz>, 2017.
- [32] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "{UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2777–2794.
- [33] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM joint meeting on European software*

engineering conference and symposium on the foundations of software engineering, 2021, pp. 1393–1403.

- [34] LLVM, “llvm-cov - emit coverage information,” <https://llvm.org/docs/CommandGuide/llvm-cov.html>, 2023.
- [35] “Afltriage,” <https://github.com/quic/AFLTriage>, accessed: 2023-6-19.
- [36] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “{AFL++}: Combining incremental steps of fuzzing research,” in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.
- [37] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.
- [38] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, “Benefits and drawbacks of adopting a secure programming language: Rust as a case study,” in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, 2021, pp. 597–616.
- [39] P. Larsen and A.-R. Sadeghi, *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018.