Detecting Hardware Trojans using Model Guided Symbolic Execution

Ruochen Dai
ruochendai@ufl.edu
University of Florida
Gainesville, FL, USA

Tuba Yavuz
tuba@ece.ufl.edu
University of Florida
Gainesville, FL, USA

ABSTRACT

We present an automated approach for detecting two types of Hardware Trojans (HTs) in hardware designs. Malicious adversaries often hide HTs under rare triggering conditions such as timing-based or input-based conditions to avoid their detection by state-of-the-art analysis techniques. Our Trojan detection method employs fuzzing and static analysis to generate models of suspicious hardware elements that are used as oracles to guide symbolic execution. Experimental evaluation on diverse hardware designs demonstrates significant speed-ups compared to existing approaches, achieving on average a 445X speed-up for timing-based HTs and on average 27X speed-up for input-dependent HTs.

CCS CONCEPTS

• **Security and privacy → Formal security models**.

KEYWORDS

Hardware Trojan, Symbolic Execution, Fuzzing, Static Analysis

ACM Reference Format:
Ruochen Dai and Tuba Yavuz. 2024. Detecting Hardware Trojans using Model Guided Symbolic Execution. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Globalization of the Integrated Circuit supply chain and the role of Third-party Intellectual Property in SoC development opened up possibilities for various attacks in the hardware domain. One type of attack is injecting malicious logic, known as Hardware Trojans (HTs), into the hardware designs. The HT is generally designed to be activated on a rare-condition. Once the condition gets satisfied, the payload causes malfunctioning of the design, such as Denial of Service, Secret Leakage, Privilege Escalation, and so on.

Hardware fuzzing [5, 8, 9] has recently become popular in analyzing hardware designs. Static analysis is a well-known technique and is often used for data-flow analysis for hardware. The challenge with HTs is that the location where they are implanted as well as the triggering logic are not known so it is difficult to use fuzzing and static analysis directly to detect HTs.

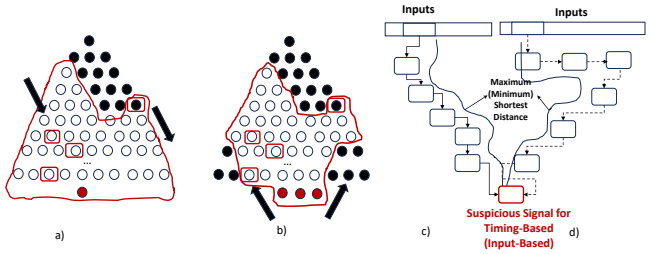

Figure 1: Exploration of symbolic execution tree for a) Timing-based Trojans, b) Input-based Trojans. Black filled circles represent states whose locations covered with fuzzing, red rounded rectangles denote the states with suspicious signal updates, red filled circles represent states violating the correctness property, the covered area shows states that will be prioritized during forward (a) or backward (b) symbolic execution, c) Data-flow graph, d) Control-flow graph.

In this paper, we focus on the automated detection of two types of HTs: timing-based and input-dependent, where the former type of HTs get activated after a number of clock cycles and the latter type of HTs get triggered after a specific sequence of inputs. Our approach uses a combination of fuzzing, static analysis, and symbolic execution to scale the analysis. Specifically, we use fuzzing and static analysis as oracles that guide symbolic execution into suspicious parts of the design, which allows symbolic execution to scale while providing a precise analysis. We have evaluated our approach on a set of benchmarks from OpenCores and Trusthub. Results show that our approach for detecting timing-based HTs is very efficient, especially for those with large trigger depth. Also, our approach for detecting input-based HTs improves over naive symbolic execution and fuzzing.

In short, the contributions of this paper are:

- An oracle guided symbolic execution based approach for detecting timing-based and input-dependent HTs and exploring the effectiveness of fuzzing and static analysis in model extraction.
- Extending the data-flow dependency analysis of SVF tool with control-flow analysis and introducing a dependency metric that is effective for identifying suspicious signals,
- Implementation of the proposed approach in a tool, called OruGuiTas[1] [2] using state-of-the-art analysis tools AFL, SVF, and KLEE.

[1]Means caterpillars in Spanish.
[2]https://github.com/RuochenDai78/OruGuiTas.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Evaluation of OruGuiTas in comparison to state-of-the-art analysis techniques based on fuzzing, symbolic execution, and model checking.

2 RELATED WORK AND BACKGROUND

Hardware Trojan detection. Various heuristics on how often Trojans get activated have been used to detect Trojans. These heuristic-based approaches include identification of unused circuits [4], suspicious signals [11], suspicious wires [10], and dedicated triggers [12]. Formal verification approaches to Trojan detection require either functional specifications [15] or security relevant specifications [7]. Self-referencing techniques [6], which eliminate the need for a golden chip, leverage various physical characteristics of the circuits to detect Trojans during post-silicon analysis. HWDCT [3] focuses on only don't-care transition triggered Trojans. Recently, fuzzing [8, 9] has been used to analyze hardware designs. While [9] works on a software representation of the hardware, TheHuzz [8] simulates the hardware design while guiding fuzzing with a variety of coverage information.

Symbolic Execution. Symbolic execution is a program analysis technique that can potentially achieve high coverage of the system under analysis. It labels inputs as symbolic and typically interprets the instructions of an intermediate language, such as the LLVM IR, so that expressions that involve symbolic values are manipulated according to the semantics of the instructions. We have implemented our approach on top of the KLEE symbolic execution engine [2].

American Fuzzy Lop. American Fuzzy Lop (AFL) is a mutational fuzzer that employs instrumentation in the target program to generate a control-flow based coverage map [1]. AFL uses genetic algorithms to mutate user-provided inputs using byte-level operations. These mutations are guided by coverage information obtained from running the instrumented program on the generated inputs. The interesting mutants are saved and mutated again and the process continues with the newly generated inputs.

Static Value-Flow (SVF). SVF is a tool that enables scalable and precise inter-procedural data-flow and control-flow analysis for C programs by leveraging recent advances in sparse analysis [16]. SVF is implemented on top of the LLVM IR . It provides two data structures: 1) The Sparse Value Flow Graph (SVFG) in which the nodes represent the LLVM IR instructions and the edges represent data-flow dependencies between those, and 2) Inter-procedural Control-Flow Graph (ICFG) in which the nodes also represent the LLVM IR instructions and the edges represent control-flow dependencies such as branching and function calls.

3 APPROACH

3.1 Threat Model

We assume that the Trojan may be inserted into the hardware design by a rogue designer or by a third-party intellectual property (3PIP) vendor. For both types of Trojans, we assume that once activated the payload violates a functional property of the design. While we assume that in the case of input-based Trojans the attacker needs to have access to the circuit at run-time to deliver the triggering input sequence, for timing-based Trojans the attacker does not need to have an active participation to trigger it.

Algorithm 1: GenerateOracleModel: Generate suspicious instructions as an oracle for Trojan detection.

Input: P: HW Design, $mode$: {Fuzzing,Static,FuzzingStatic}, $type$: {Timing, Input}
Output: Oracle Model

$om \leftarrow \emptyset$: ▷ Set of Instructions;
if $mode$ = *Fuzzing or* $mode$ = *FuzzingStatic* **then**
  $om \leftarrow Locs(P) \setminus FuzzingCov(P)$
end
if $mode$ = *Static or* $mode$ = *FuzzingStatic* **then**
  if $type$ *is Timing* **then**
    $dfMD2I \leftarrow SVFExtended(P, DF, max)$;
    $cfMD2I \leftarrow SVFExtended(P, CF, max)$;
  end
  else
    $dfMD2I \leftarrow SVFExtended(P, DF, min)$;
    $cfMD2I \leftarrow SVFExtended(P, CF, min)$;
  end
  $om \leftarrow om \cup \{si \mid ss \in dfMD2I \cap cfMD2I \wedge si$ *is a store instruction in* P *accessing* $ss\}$;
end
return om;

3.2 Overall workflow

Our approach takes as input a hardware design specified in Verilog HDL and one or more functional correctness properties specified in SystemVerilog Assertions (SVAs). We first translate the hardware designs into their C++ implementations using Verilator [18], and their C++ representations are then combined with separate testbenches to generate either executable binary for fuzzing or LLVM bitcode for symbolic execution and static analysis.

We use fuzzing as an oracle for both the timing-based and input-dependent based HTS. We hypothesize that code that could be covered by fuzzing are easier to reach. So, we identify code locations that are difficult to reach by filtering out those that could be reached with fuzzing. We also use static analysis to identify suspicious code locations for both types of HTs. However, how we identify the suspicious signals for each type of Trojan differs slightly. For timing-based HTs, suspicious instructions define values of signals with maximum distance to some input signal(s) both in terms of data-flow and control-flow dependencies. For input-based HTs, on the other hand, suspicious instructions define values of signals with minimum distance to some input signal(s) both in terms of data-flow and control-flow dependencies.

Figure 1 summarizes our Trojan detection approach. To detect timing-based HTs, OruGuiTas first performs one-clock cycle approximate symbolic execution, where both the input signals and the internal signals get symbolized. This starts the execution from arbitrary states possibly including the unreachable states. The successors of these states according to one-clock cycle symbolic execution are explored by prioritizing the states that are difficult to reach and those that define suspicious signals. Then forward execution with concretized input values is used to search for violation of the properties. To detect input-dependent Trojans, OruGuiTas performs backward symbolic execution starting from the states that violate the correctness properties until it reaches the reset state. For both types of Trojans, OruGuiTas explores the candidate states by prioritizing the difficult to reach states as suggested by static analysis (as shown by c) and d) in Figure 1) and fuzzing.

Table 1: Benchmark Descriptions of Timing-based Trojan and Input-based Trojan.

Benchmark	Description
Time-based Trojan	
Arbiter [1,2]	Four level, round-robin arbiter, trigger flips the output gnt0
UART [1,2]	8-bit UART, trigger flips the first bit of the received data
IMA_ADPCM [1,2]	Audio compression algorithm, trigger flips the last bit of step index
AES-T2500 [3]	128-bit AES algorithm, trigger flips the first bit of encrypted key
RISC16F84 [1,2]	RISC microcontroller, trigger flips the first bit of RAM write data
AE18_core [1,2]	AE18 8-bit Microprocessor, trigger flips data write back logic
Input-dependent Trojan	
RS232-T600 [3]	Trigger: 8'hAA, 8'h55, 8'h22 , 8'hFF. Payload: sticks xmit_doneH at 1.
RS232-T700 [3]	Trigger: 8'hAA, 8'h00, 8'h55, 8'hFF. Payload: sticks xmit_doneH at 0.
RS232-T900 [3]	Trigger: 8'hAA, 8'h55, 8'h22, 8'hFF. Payload: blocks transmission.
RS232-T901 [3]	Trigger: 8'hAA, 8'h00, 8'h55, 8'hFF. Payload: blocks transmission.

[1] The timing-based Trojan has been generated by modifying the original design according to the counter-based trigger mechanism from AES-T2500.

[2] Benchmark from OpenCores[14].

[3] Input-dependent benchmark from Trust-Hub[17].

Algorithm 2: DetectTimingBasedHT: Oracle Guided Symbolic Execution for Timing-based Trojan Detection.

Input: P: HW Design, $Spec$: Specifications, $mode$: {Fuzzing,Static,FuzzingStatic}, τ: Timeout
Output: Detected Timing-based Trojan (if any)

1 $om \leftarrow GenerateOracleModel(P, mode, Timing)$;
2 Let I denote inputs of P;
3 Let S denote registers and wires in P;
4 Let $state$ denote initial state of P, where I and S marked symbolic;
5 $succs \leftarrow SymExForOneClockCycle(state)$;
6 $Active \leftarrow \{s' \mid s' = s[(I,S) \leftarrow Solve(s.PC)] \wedge s \in succs \wedge s.pc \in om\}$;
7 **while** $Active \neq \emptyset$ *and* τ *not reached* **do**
8 $cur \leftarrow prioritizeAndChooseNext(Active, om)$;
9 $succ \leftarrow SymExForOneClockCycle(cur)$;
10 **if** $CheckViolation(succ, Spec)$ **then**
11 **return** true ▷ Trojan Found;
12 **end**
13 $Active \leftarrow Active \cup \{succ\} \setminus \{cur\}$;
14 **end**
15 **return** false ▷ No Trojan Found;

3.3 Static Analysis Extensions

Since Verilator uses a specific implementation of hardware design signals using a `struct` data type, we can identify the specific hardware signals in the SVFG/ICFG by matching the index that corresponds to the specific field representing the signal in the struct type with the index in the GetElementPtr instructions. Next, on the SVFG we find all the paths starting from the `load` instructions of the input signals to the `store` instructions accessing other signals and compute a metric called Minimum Distance to Inputs (MD2I) to quantify the data-flow dependency between a signal and the input signals, excluding the **clock** and **reset** signals. MD2I corresponds to the length of the shortest path. We also extended SVF to extract the control-flow dependencies from the ICFG and the LLVM IR. In essence, we initiate the search from the `store` instructions for the internal signal and identify branch instructions that control these instructions, as well as those that control the `load` instructions that have a path to these instructions in accordance with the ICFG.

Timing-based Trojans exploit timing behavior variations of a design to disrupt its normal operation. For instance, a counter-based timing Trojan counts the number of clock cycles since reset. When the counter reaches a specific value, the Trojan becomes activated.

3.4 Timing-based Trojan Detection

Figure 1 depicts our approach for timing-based Trojans at a high-level and Algorithm 2 provides the technical details. Our approach can leverage individual oracles, `Fuzzing` and `Static`, or combine them into a hybrid one, `StaticFuzzing`. The oracle generates some information that helps the symbolic execution steps in prioritizing paths that are likely to detect the Trojan. Specifically, as shown in Algorithm 1, we get the code covered during the fuzzing campaign and compute the code that is not covered (line 3). For the static oracle, we compute the signals with the maximum MD2I in terms of data-flow (line 7) and those with the maximum MD2I in terms of the control-flow (line 8) under the assumption that such signals may get used in the Trojan trigger. If there are no signals with a certain type of dependence, e.g., not found, then `SVFGExtended` returns all the signals. Then we get the intersection of such signals and find the `store` instructions that access them. So, *om* denotes the oracle model and includes the locations of instructions to prioritize during symbolic execution.

Once the oracle model, *om*, gets generated, in Algorithm 2, the next step labels both the inputs and the registers as symbolic to start the symbolic execution from an arbitrary state rather than the reset state, which makes the analysis approximate. Performing one clock-cycle symbolic execution (line 5) yields a set of successor states, which get filtered (line 6) based on whether the program counter (pc) refers to a Trojan relevant code location, *om*. For those that are filtered, the path constraint, *PC*, is used to derive concrete inputs using the SMT solver as the focused forward analysis work on these states while using concrete inputs. So, our approach performs full symbolic execution for one clock cycle and then explores these states for multiple clock cycles until either the specification violation is found or the timeout is reached. In the focused forward step (lines 7-14), paths that execute Trojan relevant code locations are prioritized (line 8).

3.5 Input-dependent Trojan Detection

Input-dependent Trojans are designed to be triggered by specific inputs to the system, and may be designed to remain dormant until the desired input pattern is detected.

Figure 1 depicts our approach for detecting input-dependent Trojans at a high-level and Algorithm 3 provides the details. It first generates the oracle model (Line 3) by executing Algorithm 1, which uses the minimum distance to inputs when identifying suspicious code elements using static analysis. Then Algorithm 3 obtains the initial state `is` (Line 3) and subsequently performs one clock-cycle symbolic execution from a state where both the registers and the inputs are marked symbolic (Lines 5 and 6). The generated states are filtered to include only states in which the specification fails (Line 7). It uses repeated forward symbolic execution and chaining states backward to simulate backward symbolic execution. Performing backward symbolic execution from these error states may yield states that match the initial state of the hardware, and, hence, leading to a conclusion that the input-dependent Trojan is detected (Lines 10-12), or would lead to new states to work on them backward (Lines 15-17).

Algorithm 3: DetectInputBasedHT: Backward Symbolic Execution for Input-dependent Trojan Detection.

Input: P: HW Design, $Spec$: Specifications, τ: Timeout
Output: Detected Timing-based Trojan (if any)

```
is: initial state, om: set of instructions;
om ← GenerateOracleModel(P, mode, Input);
is ← ResetDesign(P);
Let S denote registers and wires in P;
Let state denote initial state of P, where I and S marked symbolic;
succs ← SymExForOneClockCycle(state);
Active ← {succ | succ ∈ succs ∧ ¬Spec holds in succ};
while Active ≠ ∅ and τ not reached do
    cur ← prioritizeAndChooseNext(Active, om);
    if cur.PC holds in is then
        return true                    ▷ "Trojan Found";
    end
    succs ← SymExForOneClockCycle(state);
    for each succ ∈ succs do
        if cur.PC holds in succ then
                                ▷ succ acts as a predecessor of cur;
            Active ← Active ∪ {succ}
        end
    end
    Active ← Active \ {cur}
end
return false                       ▷ "No Trojan Found";
```

Table 2: Comparison of different oracle modes (F=Fuzzing-guided, S=static analysis-guided, FS=Fuzzing+Static Analysis-guided) for Timing Trojan detection with *OruGuiTas*, time-out=6 hours, trigger depth=2^{16}.

Benchmark	AFL time (min)	Static time (min)	SymEx time (min)			Best Total time (min)
			F	S	FS	
Arbiter	0.217	1.49	0.661	0.635	0.533	0.878 (**F**)
UART	0.233	0.97	0.475	0.465	0.367	0.708 (**F**)
IMA_ADPCM	0.483	1.32	0.507	0.506	0.480	0.99 (**F**)
AES-T2500	1.767	28.69	216.47	212.94	180.89	211.347 (**FS**)
RISC16F84	9.65	7.44	2.75	2.94	2.46	10.38 (**S**)
AE18_core	19.45	18.12	2.26	2.08	2.01	20.2 (**S**)

A naive approach to detecting input-dependent Trojans is fully forward symbolic execution, as shown in Section 4.2, starting the symbolic exploration of input-based Trojans from the error state provides better performance compared to the naive approach, which faces the path explosion problem

4 EVALUATION

4.1 Timing-based Trojan Detection

Impact of Oracles. We ran OruGuiTas for each benchmark with three possible oracle modes as mentioned in Algorithm 2. We computed Fuzzing-guided only (F), Static-guided only (S), and Fuzzing+ Static-guided (FS) in OruGuiTas using *AFL time + F time*, *Static time + S time*, and *AFL time + Static time + FS time*, respectively. We set the Trojan trigger depth to 2^{16} (the value of the counter when the Trojan gets activated), and the result is shown in Table 2. The *Best Total Time* column refers to the minimum time among the different oracle modes.

Fuzzing-guided only performs the best for Arbiter, UART, and IMA_ADPCM, Static-guided performs the best for RISC16F84 and AE18_core, and Fuzzing+Static-guided performs the best for AES-T2500. We also find that considering the symbolic execution time only (Columns 4-6), using both oracles has an advantage over using a single oracle as Fuzzing+Static-guided search can achieve an average speed up of 1.14 and 1.13, and a maximum speed up of 1.29 and 1.26, compared to Fuzzing-guided only and Static-guided only, respectively. We think that the knowledge of suspicious signals improves the performance as the design complexity increases. However, the overhead of static analysis may make fuzzing more advantageous due to its lower analysis overhead for smaller designs.

Table 3: Comparison of different oracle modes (F=Fuzzing-guided, S=static analysis-guided, FS=Fuzzing+Static Analysis-guided) for Input-dependent Trojan detection.

Benchmark	AFL time (min)	Static time (min)	SymEx time (min)			Best Total time (min)
			F	S	FS	
RS232-T600	0.26	0.53	1.51	1.38	0.86	1.65 (**FS**)
RS232-T700	0.23	0.53	1.73	1.46	0.87	1.63 (**FS**)
RS232-T900	0.29	0.54	1.68	1.34	1.04	1.87 (**FS**)
RS232-T901	0.25	0.56	1.67	1.70	0.97	1.78 (**FS**)

Comparison with other Approaches. We first attempted to compare OruGuiTas to naive forward symbolic execution. However, the naive approach timed out at a trigger depth of 2^8 for simple designs like the arbiter, and at 2^6 for more intricate designs like AES, all within a 24-hour period.

As shown in Figure 2, we compare the Best Total analysis time of our approach with two tools: EBMC [13], and HW-Fuzzing [9], for varying Trojan trigger depth ranging from 2^4 to 2^{16}. The time it takes for OruGuiTas to detect the Trojans is not sensitive to the trigger depth due to using a one-clock cycle full symbolic execution during the approximate analysis as the focused forward analysis uses concrete inputs. Also, while the oracle times depend on the design and code size they do not depend on the trigger depth. However, for EBMC and Hw-Fuzz, total analysis time grows exponentially as the trigger depth increases.

Specifically, for EBMC, it can be very efficient when the design complexity and the trigger depth are both small, like Arbiter, UART, and IMA_ADPCM. But when dealing with large designs, like AES, it quickly runs out of memory even for a small trigger depth of 2^6. For Hw-Fuzzing, it can also detect specification violations very fast when the trigger depth is small and does not depend much on design complexity, however, when the trigger depth is larger than 2^{14}, it would quickly timeout. So, our approach performs much better than EBMC and Hw-Fuzz when the trigger depth is larger than 2^{14} clock cycles, and achieves an average speedup of 445 times compared to both techniques.

4.2 Input-dependent Trojan Detection

Impact of Oracles. Similar to timing-based Trojan detection oracle modes, OruGuiTas analyze each benchmark in three different oracle configurations, as delineated in Algorithm 1. We evaluated Fuzzing-guided (F), Static-guided (S), and Combined Fuzzing and Static-guided (FS) configurations in OruGuiTas, quantified by *AFL + F time*, *Static + S time*, and *AFL + Static + FS time*, respectively. As shown in Table 3, the *Best Total Time* column captures the optimal timing across various oracle modes. Our analysis indicates that

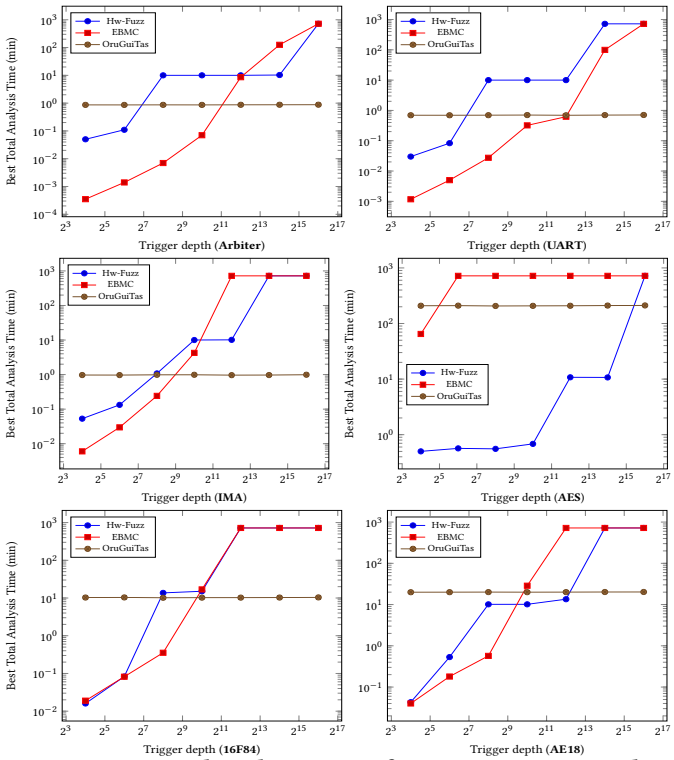

Figure 2: Best Total analysis time of EBMC, HW-Fuzz, and OruGuiTas v.s. Trigger depth, Timeout is 12 hours, Memory out of bound is 16GB.

using both oracles is advantageous (lines 4-6), as the combined approach achieves an average and maximum speedup of 1.10 and 1.16 against F-only, and 1.19 and 1.27 against S-only, respectively. We attribute this to the richer information pool derived from dual oracles, enabling more effective prioritization in symbolic execution.

Comparison with naive symbolic execution. Table 4 shows the results of input-dependent Trojan detection using naive forward and backward symbolic execution approaches. It's clear that naive backward analysis performs much better than naive forward analysis in terms of detection time. However, OruGuiTas performs better than the naive approaches. Specifically, oracle-guided backward analysis achieves up to 36X and 5.6X and on average 27X and 5.2 X speedup for compared to naive forward and naive backward symbolic execution, respectively.

Comparison with other Approaches. As also shown in Table 4, we compare the backward analysis time of our approach with HW-Fuzz on different Trojan designs listed in Table 1. We excluded EBMC from this table as it generated false positives with a counter-example length of one for all the benchmarks in this category. It is clear that OruGuiTas outperforms HW-Fuzz with a maximum speedup of 2.5X and an average speedup of 2.2X.

4.3 Discussion

OruGuiTas achieves a larger speedup for Timing-based HTs compared to that achieved for Input-based HTs. This is because hardware fuzzing, which is the next best performing approach in our evaluation, is more effective for detecting Input-based HTs then Timing-based HTs, which depends on an internal signal that gets updated independent of the inputs. OruGuiTas, on the other hand, handles both input dependency and less covered design elements.

Table 4: Results on Input-dependent Trojan Detection time with pure backward symbolic execution (BSymEx), forward symbolic execution (FSymEx), HW-Fuzz, and OruGuiTas.

Benchmark	FSymEx time(min)	BSymEx time(min)	HW-Fuzz time(min)	OruGuiTas time(min)
RS232-T600	48.67	8.21	4.17	1.62
RS232-T700	58.56	9.08	3.42	1.63
RS232-T900	40.81	9.45	3.88	1.87
RS232-T901	41.21	9.14	3.92	1.78

5 CONCLUSION

We present oracle guided symbolic execution techniques for the detection of timing-based and input-dependent HTs and show that our approach improves upon hardware fuzzing and bounded model checking with zero false positives.

ACKNOWLEDGMENTS

This project has been partially funded by NSF Award 2019283.

REFERENCES

[1] American fuzzy lop. [n. d.]. "http://lcamtuf.coredump.cx/afl/".
[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX OSDI, San Diego, California, USA*. 209–224.
[3] Ruochen Dai and Tuba Yavuz. 2022. A Symbolic Approach to Detecting Hardware Trojans Triggered by Don't Care Transitions. *ACM Trans. Des. Autom. Electron. Syst.* (aug 2022). https://doi.org/10.1145/3558392
[4] Hicks Matthew et al. 2010. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *IEEE Symposium on S&P*.
[5] Hoang M Le et al. 2019. Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.
[6] Liu Yu et al. 2014. Hardware Trojan Detection through Golden Chip-Free Statistical Side-Channel Fingerprinting. In *Proceedings of the 51st Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '14)*.
[7] Rajendran Jeyavijayan et al. 2015. Detecting malicious modifications of data in third-party intellectual property cores. In *Design Automation Conference (DAC)*.
[8] Rahul Kande et al. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 3219–3236.
[9] Trippel Timothy et al. 2022. Fuzzing hardware like software. In *31st USENIX Security Symposium*. 3237–3254.
[10] Waksman Adam et al. 2013. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. 697–708.
[11] Xuehui Zhang et al. 2011. Case study: Detecting hardware Trojans in third-party digital IP cores. In *Proceedings of the 2011 IEEE HOST, San Diego, California, USA*. 67–70.
[12] Zhang Jie et al. 2015. VeriTrust: Verification for Hardware Trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 7 (2015), 1148–1161.
[13] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. 2015. Hardware verification using software analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 7–12.
[14] OpenCores. [n. d.]. https://opencores.org/.
[15] Michael Rathmair, Florian Schupfer, and Christian Krieg. 2014. Applied formal methods for hardware Trojan detection. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*.
[16] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*.
[17] TrustHub. [n. d.]. https://www.trust-hub.org/#/home.
[18] Verilator. [n. d.]. https://www.veripool.org/verilator/.