Detection of Hardware Trojans in SystemC HLS Designs via Coverage-guided Fuzzing*

Hoang M. Le[1], Daniel Große[1,2], Niklas Bruns[2] , Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{ hle,grosse,drechsle } @informatik.uni-bremen.de Niklas.Bruns@dfki.de

***Abstract*—High-level Synthesis (HLS) is being increasingly adopted as a mean to raise design productivity. HLS designs, which can be automatically translated into RTL, are typically written in SystemC at a more abstract level. Hardware Trojan attacks and countermeasures, while well-known and well-researched for RTL and below, have been only recently considered for HLS. The paper makes a contribution to this emerging research area by proposing a novel detection approach for Hardware Trojans in SystemC HLS designs. The proposed approach is based on coverage-guided fuzzing, a new promising idea from software (security) testing research. The efficiency of the approach in identifying stealthy behavior is demonstrated on a set of open-source benchmarks.**

I. Introduction

The increasing adoption of *third-party IPs* (3PIPs) has made embedded systems more vulnerable to *Hardware Trojan* (HT) attacks. In such attacks, adversaries deliberately insert hidden malicious behavior into an IP before it leaves the vendor. The activation of this malicious behavior during system operation might harm the functionality or leak secret information, thus seriously affect the safety and security of the system.

The threat of HTs as well as their countermeasures have been actively researched in the last decade [1]. For HTs in 3PIPs, the detection should ideally be performed before production/manufacturing. A number of presilicon detection approaches (survey [1], recent work e.g. [2], [3]) have been proposed considering the abstraction levels of RTL and below (e.g. gate or transistor-level), as 3PIPs are commonly shipped to the IP consumer at these levels.

On the other hand, to raise design productivity, *High-level Synthesis* (HLS) has recently emerged as an alternative design entry to RTL. HLS designs, often developed using an synthesizable subset [4], [5] of SystemC [6] at a more abstract level, can be automatically synthesized into RTL. The quality of these generated RTLs is mostly comparable to hand-written RTL for the same functionality with much shorter development time [7]. Due to the flexibility in generating multiple variants of the same design , more and more 3PIPs are expected to be delivered as SystemC HLS designs.

This new abstraction level is also susceptible to HTs. A malicious behavior can be coded in SystemC and if undetected, will be synthesized together with normal functionality into RTL. Thus, we refer to HT at this level as *(High-level) Synthesizable Hardware Trojan* (SHT). Since the malicious behavior is preserved by synthesis, it could possibly be detected at RTL or lower levels by existing solutions. However, detection approaches tailored to SHT are still desired, since the detection at the level where a SHT is created should be much less challenging.

Related Work: To the best of our knowledge, the design and detection of SHT has been only recently discussed in [8]. The paper demonstrated the ease of SHT creation using C++ control flow constructs and proposed to use SystemC HLS property checking to detect suspicious control flows that are not exercised by the vendor-supplied verification testbench. A subsequent effort [9] has created the open-source SHT benchmark suite S3CBench to complement the popular Trust-Hub benchmarks for RTL and below. One issue with the detection approach in [8] is that, despite the recent advancements (e.g. [10]–[12]) and the availability of commercial tools (e.g. NEC CyberWorkBench, Calypto SLEC), SystemC formal verification is still not yet mature (i.e. scalability and usability issues exist).

Coverage-guided fuzzing (CGF) is a new promising idea from the software (security) testing community. Popular CGF engines such as AFL or LLVM's LibFuzzer have detected countless hidden vulnerabilities in well-tested software. Thanks to a light-weight coverage-collecting instrumentation, a set of simple but carefully selected input mutations and the integration of a feedback loop, a CGF engine is capable to reach many corners of the *Program-Under-Test* (PUT), that are not possible with conventional testing techniques or even symbolic approaches (due to scalability).

Paper Contribution: Inspired by the success of CGF, we investigate its application in revealing malicious behavior within a SystemC HLS design. The contribution of this paper is multi-fold:

1) For the first time, CGF is applied to SystemC and for HT detection;
2) A new set of mutations tailored for SHT has been developed that increases the effectiveness of CGF significantly;
3) An extensive evaluation on S3CBench including comparison to formal verification is provided.

II. Synthesizable Hardware Trojans

Generally, the malicious behavior of a HT consists of two parts: a trigger and a payload. The trigger monitors a logic

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contract no. 16KIS0821K and by the University of Bremen's Central Research Development Fund and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

```
counter += (value == 0x42424242) ? 1 : 0;
// some lines later ...
sum = (counter > 3) ? (sum += var) : 0;
```

Fig. 1. Example of Hardware Trojan in SystemC HLS design

condition depending on various signals/events of a design. Once this triggering condition of a HT is satisfied, the payload is activated to perform actual malicious behavior. The trigger is designed to be only activated under extremely rare conditions, otherwise an infested design behaves exactly like a HT-free one. For digital HTs, which are in the focus of this paper, the triggering condition can be *sequential*, i.e. evaluated on a sequence of signals/events, or *combinational*. Counter-based is one popular technique of creating sequential HTs.

SHT can be easily inserted into a SystemC HLS design by modifying its control flow. For example, an additional *if-else* with malicious behavior in the if-branch can be inserted. A stealthier way, as proposed in [8], is to leverage conditional assignments, such that all SHT-related statements will be easily covered by a verification testbench without triggering the SHT payload. A simple example of a sequential SHT is shown in Fig. 1. Once a specific value is observed, a counter is incremented. Later in the execution flow, this counter is checked against a threshold value and the payload is activated accordingly, i.e. instead of summing up, the result of the computation is set to zero. Even stealthier SHTs are to be developed in the future.

III. Coverage-guided Fuzzing for SHT Detection

A. Problem Formulation

Presilicon HT detection approaches can only flag suspicious behavior whose (malicious) intent has to be manually analyzed. One of the central issues here is the *availability of a golden model*. If such a model is available, HT detection can look for a functional deviation. Otherwise, it normally tries to flag code/circuit regions that have very low controllability and/or observability. In the context of 3PIPs, a verification testbench and inputs must be supplied together with the IP. So any code/circuit region that is not covered by this testbench is also suspicious.

Previous work on SHT detection [8] assumes there is no golden model. It flattens the SystemC HLS *Design-Under-Test* (DUT) first, to reveal hidden control flows caused by e.g. conditional assignments as described earlier. Then, it tries to reach every branch that is not covered by the supplied testbench. The SHT detection problem is thus reduced to maximization of branch coverage in the flattened SystemC HLS DUT.

While using the same problem formulation, we argue that for an SystemC HLS 3PIP, the IP consumer can opt to develop a golden model. Note that this golden model does not need to be synthesizable but only functionally correct. Thus, its development should not be very time-consuming.

It is worth mentioning that in practice, the inputs from the supplied testbench are mostly provided as files. The testbench parses these files to get raw stimuli to feed the DUT. This practical issue requires adaptations in CGF which will be addressed in the following.

B. Approach Overview

As mentioned in the introduction, CGF consists of three main ingredients: a compiler pass to instrument the PUT with additional code to collect coverage with minimum execution overhead, a set of fuzzing mutations and a feedback loop. This loop, starting with a (possibly empty) set of inputs $\mathcal{I}$, performs the following steps: 1. Select an input I from $\mathcal{I}$; 2. Apply a set of mutations to I to obtain I_m; 3. Execute the instrumented PUT on I_m; 4. If the execution increases the overall coverage, add I_m to $\mathcal{I}$. These steps are repeated until a given time limit is reached or CGF concludes that it cannot improve the coverage. For more details we refer to [13].

The overall flow of the proposed CGF-based SHT detection is shown in Fig. 2. We leverage the instrumentation pass of CGF to also flatten all control flow constructs (CF-Flattening) and obtain the instrumented DUT-Instr. The execution of PUT is substituted with a SystemC simulation of DUT-Instr with the supplied testbench. In the initialization step, DUT-Instr is simulated with the supplied file inputs (TB Inputs) to generate the initial set of CGF inputs and its coverage. Then, the feedback loop with file input selection and mutation is applied. After CGF stops, a set of control flows that are not in the initial coverage together with inputs reaching them are reported as suspicious. Optionally, if a golden model is available, the suspicious set can be further reduced by removing inputs and control flows that do not show any deviation when comparing DUT with the golden model. A reduction of the suspicious set results in a reduced detection time because fewer cases must be checked. CGF engines interpret the content of the input files (included in the testbench) as a byte array, but many times a special format is expected by the testbench. This potential mismatch causes difficulties for the mutation step as will be explained in the next section together with our proposed SHT-oriented mutations.

IV. Mutations for SHT Detection

We first describe the conventional software-oriented mutations geared towards finding security vulnerabilities. Then, we discuss their limitations in the context of SHT detection. Finally, we present a new set of SHT-oriented mutations to overcome these limitations.

A. SW-oriented Mutations

We describe the basic ideas of the mutations employed by AFL [13]. An input is interpreted as a byte array. To maximize the execution speed, AFL applies a *trim* mutation that tries to reduce the size of this array without changing the coverage of the input. For the discovery of new behavior (and thus new coverage), AFL uses a family of *bitflip* mutations that flip a different number of bits in an input at the same time. The *arithmetic* mutations take a portion of bytes from the input, interpret these bytes as an integer value and subtract/add a small integer from/to this value. The *interesting value* mutations try to overwrite a portion of bytes with the byte representation of some special value (e.g. $0, -1, INT\_MAX$, etc.). The *havoc* mutation is a combined mutation that apply between 2 and 128 individual mutations to an input at random positions. The *splice* mutation is only used when the other

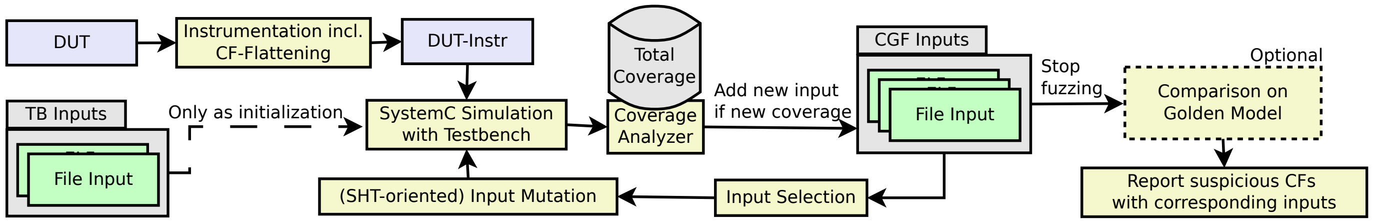
DUT
Instrumentation incl. CF-Flattening
DUT-Instr
Total Coverage
TB Inputs
File Input
Only as initialization
SystemC Simulation with Testbench
Coverage Analyzer
Add new input if new coverage
CGF Inputs
File Input
Stop fuzzing
Optional
Comparison on Golden Model
Report suspicious CFs with corresponding inputs
(SHT-oriented) Input Mutation
Input Selection

Fig. 2. Coverage-guided Fuzzing for SHT detection

mutations could not help to discover new coverage. It chooses an input, applies the well-known genetic crossover operator cut-and-splice, then invokes the havoc mutation.

B. *Limitations of SW-oriented Mutations for SHT Detection*

Format unawareness: The SW-oriented mutations work at the byte level and do not have any knowledge about the input format. As a result, most mutated inputs are invalid or only “almost” valid. This is very useful for robustness/vulnerability testing, but inadequate for the SHT detection. SHTs are already hard to trigger with valid inputs and thus much harder for CGF to detect if the testbench aborts the simulation already at the format checking stage.

Strong bias for small inputs: Most of the AFL mutations have minimal impact on the size of the inputs. Among the few exceptions, the trim mutation only reduces the size. Both, the havoc mutation and the splice mutation can shorten or expand an input, but shortening is much more likely.The reason for this bias towards reducing the size of inputs is to optimize the speed of each individual execution of CGF. However, this bias of the mutations makes it very hard to trigger sequential counter-based SHTs, which require bigger inputs (i.e. long sequences of stimuli).

Difficulty with tight comparisons: Well-hidden hard-to-trigger SHTs, both combinational and sequential, often include in their trigger conditions tight comparisons, i.e. comparisons to a single value or a small set of values such as `if (value == 0x42 && ...)`. CGF is known to have difficulties in generating values satisfying these comparisons. This difficulty is even more amplified in our context considering the direct relationship between SHTs and tight comparisons.

Too optimistic byte exclusion: The 8-bit bitflip (byteflip) mutation is used additionally as a heuristic. If a byteflip at a position does not change the coverage, no other deterministic mutation will be used again at that position. In combination with the format unawareness, this often leads to the exclusion of all or too many byte positions from the search space.

C. *SHT-oriented Mutations*

To overcome these limitations we have devised a set of new mutations that are tailored for SHT detection.

Pump mutation: Targeting (counter-based) sequential SHTs, the pump mutation is designed such that it increases the size of the to-be-mutated input I stepwise. In this process, a new input is obtained by appending I multiple times to itself. By doing so, the mutated input will also have a valid format most of the time if I is also valid.

Format-aware trim mutations: A new family of format-aware trim mutation aims to prevent the corruption of the input format through trimming. When the input is expected in a line-based format (e.g. one integer per line), in contrast to the conventional trim mutation, the new trim mutation does not trim arbitrary parts but only whole lines. As a result, a corruption of the input structure can be avoided. For SystemC HLS designs targeting multimedia or imaging applications, the testbench requires the input to be in some specific format, e.g. bitmap. For such format, a repair pass is applied on the mutated input to avoid format corruption. Otherwise, the raw data part is either trimmed or filled with new random bytes to match the bitmap size.

Design-based interesting number mutation: As mentioned earlier, SHTs often use tight comparisons with “magic numbers” as triggers. These magic numbers must be also somehow encoded into the SystemC HLS code of a design. We leverage the Clang compiler front-end to extract the numbers from the source code. Furthermore, these numbers are inserted into an input in the mutation stage in a manner that does not corrupt the input format.

V. Experimental Evaluation

We have implemented the proposed approach based on AFL version 2.52b and evaluated our implementation on the open-source benchmark suite S3CBench [9]. We have compared our approach AFL-SHT to the unmodified version of AFL as well as to SHT detection approaches based on formal verification. The obtained results are shown in Table I, which is divided into three parts. In the following, we describe the benchmarks and both comparisons in more detail.

A. *Benchmark Description*

The first column shows the name of each benchmark. The type of the inserted SHT is encoded as the last component of the name (more details in [9]). The second column **ST** shows the stealthiness of each SHT as defined in [9]: the probability that 10,000 random inputs can trigger the SHT. The majority of the inserted SHTs is apparently very hard to detect with random stimuli. As reported in [9], the supplied testbenches do not trigger inserted SHTs but have a very high statement coverage (100% in most cases, not shown in the table).

B. *Effectiveness of SHT-oriented Mutations*

The purpose of this comparison of AFL-SHT and AFL is to assess the effectiveness of the SHT-oriented mutations. The results have been obtained in a KVM-virtualized environment on an AMD A10 Series A10-7890K host using a time limit of

TABLE I
SHT DETECTION RESULTS ON S3CBENCH USING COVERAGE-GUIDED FUZZING AND FORMAL VERIFICATION

Benchmark		Coverage-guided Fuzzing						Formal Verification	
		AFL			AFL-SHT			PropCheck	EqCheck
Name	ST	BCOV	#INP	Result	BCOV	#INP	Result	Result	Result
adpcm-swm	0.05%	97.1%	451563	T.O. ✗	100%	423	1.71s ✓	T.O. ✗	12s ✓
adpcm-swom	0.05%	97.1%	450839	T.O. ✗	100%	414	1.67s ✓	19s ✓	19s ✓
aes-cwom	0.00%	100%	50544	888.29s ✓	100%	22	0.04s ✓	105s ✓	1294s ✓
bubble-sort-cwom	0.02%	100%	118	4.82s ✓	100%	39	0.05s ✓	36s ✓	283s ✓
bubble-sort-swm	0.02%	100%	19826	337.36s ✓	100%	108	0.11s ✓	105s ✓	554s ✓
disparity-cwm	0.06%	37.1%	36391	T.O. ✗	93.2%	327	63.70s ✓	T.O. ✗	T.O. ✗
disparity-cwom	0.10%	38.3%	36736	T.O. ✗	93.8%	327	65.97s ✓	T.O. ✗	T.O. ✗
filter_FIR-cwom	0.01%	93.8%	207	8.51s ✓	93.8%	41	0.07s ✓	7s ✓	16s ✓
interpolation-cwom	0.02%	68.8%	1706325	3731.07s ✗	68.8%	2325402	4569.27s ✗	ERR ✗	ERR ✗
interpolation-swm	0.00%	100%	89	0.16s ✓	100%	47	0.90s ✓	ERR ✗	ERR ✗
interpolation-swom	0.00%	100%	89	0.16s ✓	100%	47	0.89s ✓	ERR ✗	ERR ✗
kasumi-cwom	0.00%	100%	910	3.08s ✓	100%	316	1.32s ✓	93s ✓	98s ✓
kasumi-swm	0.00%	100%	918	3.01s ✓	100%	345	1.32s ✓	8s ✓	103s ✓
sobel-swm	0.00%	95.2%	1191	457.35s ✓	95.2%	182	58.65s ✓	ERR ✗	ERR ✗
sobel-cwm	0.02%	94.2%	11362	4504.25s ✓	94.2%	10330	4154.07s ✓	ERR ✗	ERR ✗
sobel-cwom	0.07%	98.1%	1217	464.58s ✓	98.1%	190	64.02s ✓	ERR ✗	ERR ✗
uart-swm	0.00%	85.4%	172	8.82s ✓	85.4%	51	0.18s ✓	ERR ✗	ERR ✗

2 hours each. The column **BCOV** presents the branch coverage value achieved by AFL(-SHT). The column **#INP** shows the number of inputs tried by AFL(-SHT) upon completion. The column **Result** includes the computation time used by each approach and a ✓ if malicious behavior detected, a ✗ otherwise. **T.O.** denotes that the time limit is reached.

As can be seen, AFL-SHT outperforms AFL significantly. AFL-SHT can cover more behavior (higher BCOV) by generating a remarkably smaller number of inputs and thus in much shorter time. Consequently, the SHT detection rate of AFL-SHT is also superior: 16/17 vs. 12/17.

C. Comparison to SHT Detection with Formal Verification

For this comparison, we have used an anonymized commercial formal verification tool with SystemC HLS support. Unfortunately, due to license restrictions, the results have been obtained on a different machine (Intel Xeon E3-1240 V2 @ 3.40GHz). Although this machine is more powerful than the one used in the last comparison, we still apply a time limit of 2 hours for each run.

We have employed the tool in two different modes. In **PropCheck** mode, we try to reproduce the results from [8]. We invoke the available SystemC HLS property checking engine to detect SHT with a single assertion at the inserted SHT location. Unfortunately, [8] reported no performance data. In **EqCheck** mode, we use the available SystemC HLS equivalence checking engine to detect SHT as a difference of the design with SHT and the golden design. The results for PropCheck and EqCheck are reported using the same format as for AFL-SHT: computation time and ✓ if malicious behavior detected, otherwise ✗. Both PropCheck and EqCheck perform worse than AFL-SHT w.r.t. computation time and SHT detection rate (7/17 and 8/17, respectively). The earlier mentioned usability issue of SystemC formal verification tools can be observed as three designs *interpolation*, *sobel* and *uart* have caused internal errors (denoted as **ERR**).

VI. Discussion and Future Work

We have demonstrated the applicability of Coverage-guided Fuzzing for Hardware Trojan detection in SystemC HLS designs. By developing a new set of fuzzing mutations tailored to the problem at hand, all Trojans but one from the only currently known benchmark suite S3CBench can be detected. Compared to detection approaches based on SystemC formal verificaion, fuzzing with the new mutations performs significantly better. Nevertheless, the potential is not yet fully realized as there is still much room for improvement in other fuzzing components. Further research should focus on these components as well as on the development of new kinds of Trojans for HLS that are even harder to detect. Moreoever, we expect that hybrid techniques combining fuzzing with formal verification, symbolic execution and/or constrained random verification (e.g. [14]) will be needed for these next-generation HLS Trojans.

References

[1] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *TODAES*, vol. 22, no. 1, pp. 6:1–6:23, 2016.
[2] L. Piccolboni, A. Menon, and G. Pravadelli, "Efficient control-flow subgraph matching for detecting hardware trojans in RTL models," *TODAES*, vol. 16, no. 5s, pp. 137:1–137:19, 2017.
[3] S. K. Haider, C. Jin, M. Ahmad, D. Shila, O. Khan, and M. van Dijk, "Advancing the state-of-the-art in hardware trojans detection," *TDSC*, 2018 (Early Access).
[4] A. S. Initiative, "SystemC synthesizable subset version 1.4.7," 2016.
[5] P. Coussy, A. Takach, M. McNamara, and M. Meredith, "An introduction to the SystemC synthesis subset standard," in *CODES+ISSS*, 2010, pp. 183–184.
[6] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.
[7] S. Lahti, P. Sjvall, J. Vanne, and T. D. Hmlinen, "Are we there yet? a study on the state of high-level synthesis," *TCAD*, 2018 (Early Acccess).
[8] N. Veeranna and B. C. Schafer, "Hardware trojan detection in behavioral intellectual properties (IP's) using property checking techniques," *TETC*, vol. 5, no. 4, pp. 576–585, Oct 2017.
[9] ——, "S3CBench: Synthesizable security SystemC benchmarks for high-level synthesis," *Journal of Hardware and Systems Security*, vol. 1, no. 2, pp. 103–113, 2017.
[10] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, 2018 (Early Access).
[11] ——, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.
[12] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models – a case study," in *DATE*, 2016, pp. 1160–1163.
[13] M. Zalewski, "Technical whitepaper for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt.
[14] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *SoC*, 2012, pp. 1–7.