Hiding from Hardware Trojan Detectors by Avoiding Rare Events

Mattis Hasler
(mattis.hasler@barkhauseninstitut.org)
Barkhausen Institut, Dresden, Germany

***Abstract*—An important part of the defense against hardware trojans in commercially available chips is the research of detection techniques as well as the development of trojans challenging detection algorithms. The possible attack scenarios between register transfer level design and implemented silicon are manyfold and cannot be regarded at once. In this work a trojanic structure is presented that is inserted at design time and assumes a defender that is based on rare event detection as well as observability and controllability analysis (SCOAP). The introduced trojan escapes analysis based on these priciples, by avoiding rare events and abnormally large SCOAP values to blend with the original hardware. The proposed infection technique is generally able to alter the output of the attacked module to a malicious output, activated by an activation vector, independent of the module's functionality. Applied to a simple Advance Encryption Standard (AES) module the area overhead introduced by the trojan is 0.15%.**

I. Introduction

The development and detection of hardware trojans are both extremely complex and diverse fields. There is no method of finding every possible hardware trojan, as well as there is no trojan that can dodge every detection mechanism [1]. Like with any other security-related topic, both attacks and defenses are based on an attacker/defender model they are working against. Already the application environment can impose requirements and constraints which narrow down the possible attacks/defenses. For example, a defense can rely on a golden netlist to generate test patterns [2], [3], or a zero-knowledge proofing system only has limited analytic access to an encrypted netlist [4]. Generally, hardware trojans and defenses against them have been spotted in every processing step between RTL design and finished silicon [1], [5].

In this work, it is assumed that a trojan is part of the design that does not get activated during normal operation and stays dormant. Defense mechanisms working with this assumption are based on the detection of rare events and the analysis of the controllability and observability of signals. For these mechanisms, netlists of the design have to be available. For the detection of rare events, the netlist is simulated —preferably with functionality-relevant input values— and the activity (i.e. toggle count) of each signal is recorded. Signals with very low toggling activity (the toggling becomes a rare event) are then suspected to be part of a hardware trojan.

The SCOAP [6] analysis is a technique that statically computes observability and controllability values for each signal in a netlist. The controllability values describe how difficult it is to control a signal with the design's input signals. Similarly, observability values describe how difficult it is for a signal value to have an impact on the designs output. These measures can be used to detect certain trojans. A SCOAP-based detection exploits the fact that a trojan ideally is difficult to activate and thus contains signals with abnormally high controllability values. The mostly dormant state of trojans also leads to high observability values as they describe the difficulty that a signal has an impact on a global output signal. The rarely toggling signals and the dormant state of trojans are the results of the goal for the trojan to not be discovered through random functional testing. A trojan should be only activated under very specific circumstances, that have a low probability of occurring coincidentally. The probability to activate the trojan by chance is called activation probability.

The generic trojan, proposed in this paper, can augment an arbitrary functional unit (FU) with malicious functionality. After augmentation, the infected module will replace the normal output with a malicious output whenever a certain activation key is detected at the inputs. Value and length of the activation key are freely choosable at design time. The added logic is carefully designed to display a reasonable amount of toggling on all signals, mimicking the activity and SCOAP profiles of an uninfected module. It keeps the moderate toggling activity even when not activated, thus combining a lack of rare events with a low activation probability.

The trojan is independent of the infected module. However, it is successfully applied to an open-source Advanced Encryption Standard (AES) implementation, leaking the encryption key upon a specified input. The resulting design size, activity and SCOAP profiles are compared to uninfected and trivial trojan implementation.

II. Related Work

As stated in [1] there is no universal trojan detection procedure. There are as many techniques to insert a trojan into hardware as there are countermeasures. The vast majority of trojans and their detectors are based on rare events [1]–[4], [7]–[11]. In 2009 Chakraborty et.al. presented MERO [3], a trojan detection based on statistical test pattern generation and the use of golden netlists. Likewise, the TARMAC framework [2] also depends on a golden netlist and rare nodes. Focused on untrusted third-party IP, [7] does detection without a golden netlist and [4] in a zero-knowledge environment, but still based on rare events. In more recent times rare events analysis is combined with SCOAP ([6]) analysis to improve results and/or

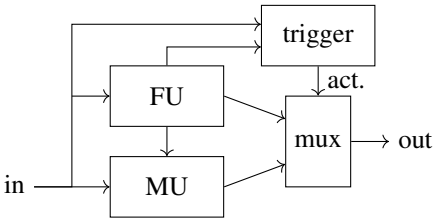

Fig. 1. Model of hardware trojan. The output of the malicious unit (MU) or the real functional unit (FU) is selected by the multiplexer (mux) depending on the trigger function.

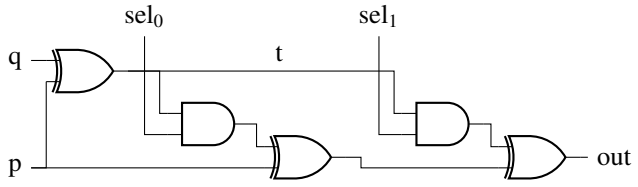

Fig. 2. Schematic of the parity multiplexer. Signal t can toggle a signal between p and q when XOR'd. The two toggle stages are activated by sel_0 and sel_1. If the parity of sel is odd, p is toggled once to q, with an even parity twice to p.

complexity of detectors. Both analyses are being combined using a genetic algorithm in the TRIAGE work [8] and reinforcement learning techniques in [9]. A more analytical approach is presented in [10]. The combinatorial analysis is targeted solely at AES trojans. Similarly, in [12] a detection framework is presented, which analytically tries to extract rarely triggering nets. In [13], [14] RTL features are examined to reveal suspicious hardware blocks. Apart from the RTL-level detection, there is a whole field of fabrication step trojans and their detection [5].

Traditionally ([1], [15]), and also in recent work ([11]), hardware trojans focused on the lowering of trigger probability by using rare events. In [15] a balance between the trigger probability and rareness of signals is found to minimize the detectability. The trojan presented in [16] tries to dodge several detection techniques, for example by actually avoiding rare events but using a combination of counters and shift registers to lower the trigger probability with not-so-rare events. In this work, this idea is extended to a point where every wire is seemingly toggling randomly without compromising on the activation probability.

III. Trivial Hardware Trojan

The goal of the regarded hardware trojan class is to modify the output of a functional unit (FU) for a specific input vector. A conceptional block diagram of the resulting module is displayed in Fig. 1. Although most trojans are not implemented using these distinguished blocks, they usually can be decomposed to fit this representation. The original FU is left unmodified. Instead, its output is given to a multiplexer (mux) together with the output of the malicious unit (MU). The task of the MU is to produce the output that should be propagated to the module's output in case the trojan gets activated. The selection between FU's original output and the MU's malicious output is controlled by the trigger block. It uses the activation signal (act.) to control the mux, which selects the correct output. The module's input is forwarded to each of the three blocks: trigger, FU and MU.

The described trojan structure can be used to implement a trivial augmentation of an AES module to leak the encryption key on the main output whenever a specific input vector is discovered. In this trivial implementation, the MU constantly outputs the encryption key:

```
assign mu_out = ENC_KEY;
```

The trigger contains a simple comparator that compares the module's input with the activation key:

```
assign act = inp == ACT_KEY;
```

And finally, the mux is a simple multiplexer selecting between the original and the malicious output.

```
assign out = act ? fu_out : mu_out;
```

Although this would work as expected and also has a good activation probability of $p = 1/2^{128}$, assuming an input/output size of 128-bit of the AES module, it would be very easily detectable by rare-event-based detectors. The main reason is the activation signal between the trigger and the mux block. Under the assumption of random input, the toggling rates in the AND gate cascade, that implements a comparator, decrease from layer to layer. At the tip of the cascade, the activation signal is not toggling at all, giving a trojan detector a strong hint.

IV. Avoiding Rare Toggling

In the trivial trojan implementation, the rarely toggling activation signal poses a problem, because it can easily be detected in netlist simulation.

To circumvent rare events, the proposed trojan uses parity-valued logic for those signals that would produce rare events on common-valued logic. In common-valued logic the level of a wire usually maps directly the logic value of a signal, e.g. "low" maps to logic-0, "high" to logic-1. In parity-valued logic, the parity of a wire pair is mapped to logic values, so that odd parity becomes logic-0, even parity logic-1. This way the wires can toggle without changing the logic value of the signal. Starting from the trivial trojan implementation, the activation signal is replaced with parity-valued logic, to suppress suspicious (not) toggling behavior. To achieve this, the comparator in the trigger block and the multiplexer are replaced with parity-valued variants, namely "Parity Comparator" and "Parity Multiplexer".

A. Parity Multiplexer

The parity multiplexer selects between two signals based on a parity-valued selection signal. A signal with the value being either `p` or `q` can be toggled to the respective other by XOR'ing a signal `t = p ^ q` to it. As displayed in Fig. 2 the parity multiplexer consists of two toggling stages that each toggle the input value between `p` and `q` depending on the two input wires of the `sel` signal. Even parity of `sel` causes two

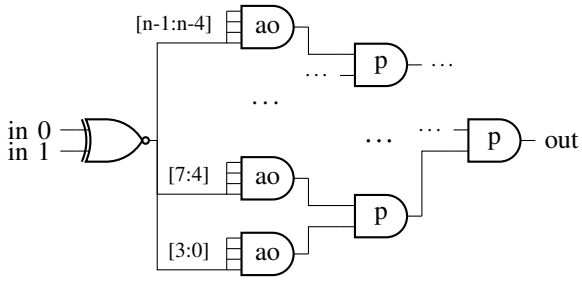

Fig. 3. Comparator with 2-bit parity result. A bit-wise equality between the inputs is created with an XNOR gate. All-one (ao) gates reduce 4-bit slices of that equality to 2-bit parity values with even parity (logic-1) when all four inputs are high. Parity-And (p) gates implement an AND gate for 2-bit parity signals with logic-1 mapped to even parity.

toggles, which produces `p` to the output. With odd parity and thus a single toggle, `q` emerges at the output.

B. Parity Comparator

The parity comparator's output signal describes if the two input signals are equal. In contrast to a common comparator, the output signal is a 2-bit parity-valued signal. Further, all signals including the output have a high probability of toggling upon an input change, even if the output is not changed logically. First, as seen in Fig. 3, a bit-wise equality signal is created by XNOR'ing the inputs. This signal must then be converted to parity-valued logic and reduced to a single value. To convert the signal to parity-valued logic, slices of four wires are fed into custom "all-one" gatess. The resulting vector of parity-valued signals is then reduced to a single signal with a cascade of parity AND (pAND) gates. The "all-one" gates check if all input signals are one, to then produce a double one (`2'b11`), which is a parity logic-1. The fact that it will never use the double zero (`2'b00`), which also is a parity logic-1, is exploited by the following pAND gates. Apart from this unused input value, both "all-one" and "pAND" are very similar, as can be seen in their Karnaugh diagrams in Fig. 4. The functions of the two output wires are described by the red and the blue coloring. Only when all four inputs are set, both outputs are activated. All other cases are separated into two sets of either output being active. The (almost) equal size of these sets assures a frequent toggling of wires upon input change. For the pAND gate, the cases with one input pair being zero are undefined and can be ignored. The resulting simplified hardware can be implemented with only four NAND gates, compared to the "all-one" gate, which uses an additional NOR gate.

V. Evaluation

The presented trojan structure can be used to augment any functional unit. For this paper, it is used on an open-source Advanced Encryption Standard (AES) module [17]. The AES module is synthesized in three variants: (1) unaltered, (2) infected with the trivial trojan (Section III), and (3) infected with the proposed trojan (Section IV). Synthesis is done with Yosys open synthesis suite [18]. The netlists are then simulated with Icarus Verilog Simulator [19] to confirm the correctness

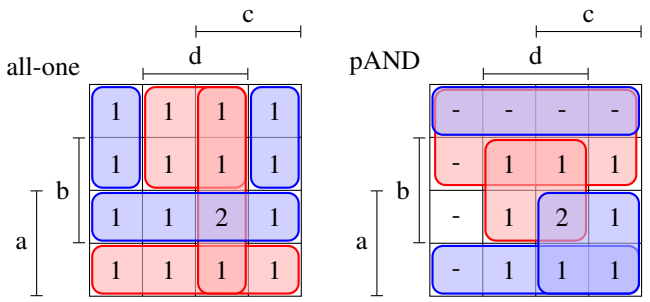

Fig. 4. Karnaugh map of the all-one and the pAND gates. The output functions are color coded. For most input values exactly one output function is active, except when all inputs are one, in which case both functions activate. The pAND gate is a simplified version taking some impossible input values into account as don't cares.

TABLE I
Synthesized Module Sizes

module	GE*	diff. to AES	% growth
AES	343272	0	–
\w trivial	343640	368	0.107
\w proposed	343792.5	520.5	0.151

* Gate equivalent (GE): The area of one NAND gate.

of the AES implementation and the functionality of the trojan augmentation. A full value change dump (VCD) from the simulation is analyzed to extract wire activity values and finally, the netlist is analyzed to extract observability and controllability values.

In each simulation the correctness of the module's output is checked for 1000 random input values. To test the data leakage by the trojan, another simulation is performed with one input value set to the activation key and the appearance of the secret key at the output is observed. The activity values shown in Fig. 5 are taken from simulations without trojan activation. The signal activity values of the proposed trojan group themselves around 0.5, which represents a signal with an equal distribution of high and low values. Especially it is to be noticed, that no signal from the trojan has an activity below or even close to 0.1 which is a common value for considering a signal to be rare. In contrast, the trivial trojan implementation does have a significant number of signals that do toggle very seldomly, even some that do not toggle at all.

In Fig. 6 the SCOAP analysis of all three netlists is shown as a histogram of the controllability and observability values. All three histograms are very similar, neither of the trojans do have a suspicious profile. Especially neither has any outliers of extremely high values that would give a hint to a trojanic structure.

The implementation cost of both the proposed and the trivial trojan are 520.5 and 368 gate equivalents, which is 0.15% and 0.11% of the AES module's size, respectively (Tab. I). At this scale, the increased size of the proposed trojan of around 50% compared to the trivial one is a small price for the increased stealthiness. The other part of the stealthiness, namely the activation probability of the proposed design is

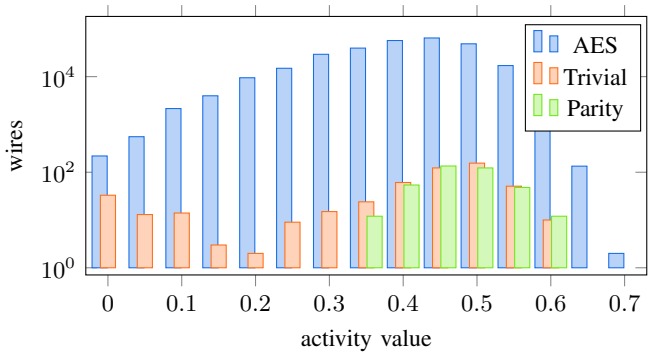

Fig. 5. Histogram of wire activity values (toggle probability). The proposed parity trojan does not include any rare, or even close to rare (activity < 0.1) signals. The trivial implementation has a large group of those signals.

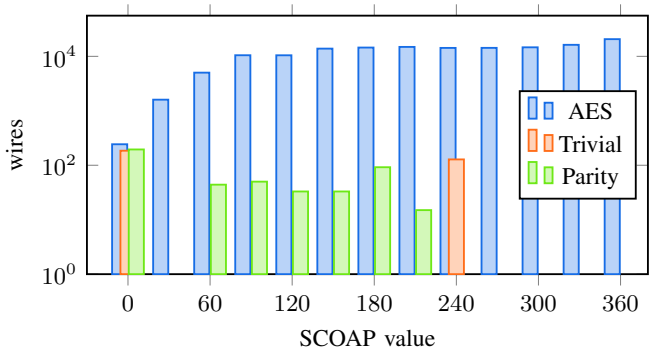

Fig. 6. Comparison of SCOAP profiles, AES module, the trivial and the parity HT implementation. Neither the proposed nor the trivial trojan show abnormally large values.

unaltered compared to the trivial implementation. Both activate at exactly one input value out of the 2^{128} possible input values, setting the activation probability to $1/2^{128} = 2.94 \times 10^{-39}$.

VI. Conclusion

A hardware trojan-hiding mechanism is presented that is agnostic to both the original and the malicious functionality. In contrast to most other hardware trojans the presented work tries to avoid rare events. The trojan activation mechanism uses parity-valued logic which maps the parity of wire pairs to the logic values zero and one. Using parity-valued logic the detection of the activation key and the selection of the malicious result can be implemented avoiding rare events without compromising on the activation probability. It is shown that the trojan's impact on area consumption, activity profile and SCOAP values of an AES unit is negligible. The area consumption is 520 gate equivalents or 0.15% of a targeted AES module. The activation probability stays unaltered compared to a trivial trojan implementation of $1/2^{128}$.

References

[1] S. Bhasin and F. Regazzoni, "A survey on hardware trojan detection techniques," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2015, pp. 2021–2024.

[2] Y. Lyu and P. Mishra, "Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 7, pp. 1287–1300, 2020.

[3] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Cryptographic Hardware and Embedded Systems-CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*, Springer, 2009, pp. 396–410.

[4] D. Mouris, C. Gouert, and N. G. Tsoutsos, "Zk-sherlock: Exposing hardware trojans in zero-knowledge," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, 2022, pp. 170–175.

[5] A. Jain, Z. Zhou, and U. Guin, "Survey of recent developments for hardware trojan detection," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2021, pp. 1–5.

[6] L. H. Goldstein and E. L. Thigpen, "Scoap: Sandia controllability/observability analysis program," in *Proceedings of the 17th Design Automation Conference*, 1980, pp. 190–196.

[7] M. Tehranipoor, H. Salmani, X. Zhang, M. Tehranipoor, H. Salmani, and X. Zhang, "Hardware trojan detection: Untrusted third-party ip cores," *Integrated Circuit Authentication: Hardware Trojans and Counterfeit Detection*, pp. 19–30, 2014.

[8] M. Nourian, M. Fazeli, and D. Hély, "Hardware trojan detection using an advised genetic algorithm based logic testing," *Journal of Electronic Testing*, vol. 34, pp. 461–470, 2018.

[9] Z. Pan and P. Mishra, "Automated test generation for hardware trojan detection using reinforcement learning," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021, pp. 408–413.

[10] L. Kampel, P. Kitsos, and D. E. Simos, "Locating hardware trojans using combinatorial testing for cryptographic circuits," *IEEE Access*, vol. 10, pp. 18 787–18 806, 2022.

[11] N. Zhang, Z. Lv, Y. Zhang, H. Li, Y. Zhang, and W. Huang, "Novel design of hardware trojan: A generic approach for defeating testability based detection," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, IEEE, 2020, pp. 162–173.

[12] S. A. Islam, L. K. Sah, and S. Katkoori, "A framework for hardware trojan vulnerability estimation and localization in rtl designs," *Journal of Hardware and Systems Security*, vol. 4, pp. 246–262, 2020.

[13] H. S. Choo, C. Y. Ooi, M. Inoue, N. Ismail, M. Moghbel, and C. H. Kok, "Register-transfer-level features for machine-learning-based hardware trojan detection," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 103, no. 2, pp. 502–509, 2020.

[14] X. Chen, Q. Liu, S. Yao, *et al.*, "Hardware trojan detection in third-party digital intellectual property cores by multi-level feature analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1370–1383, 2017.

[15] J. Zhang and Q. Xu, "On hardware trojan design and implementation at register-transfer level," in *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*, IEEE, 2013, pp. 107–112.

[16] Y. Zhang, M. Ge, X. Chen, J. Yao, and Z. Mao, "Blinding ht: Hiding hardware trojan signals traced across multiple sequential levels," *IET Circuits, Devices & Systems*, vol. 16, no. 1, pp. 105–115, 2022.

[17] H. Hsing. "Opencores.org tiny_aes." (2013), [Online]. Available: https://opencores.org/projects/tiny%5C_aes (visited on 03/17/2023).

[18] C. Wolf, *Yosys open synthesis suite*, 2016.

[19] S. Williams and M. Baxter, "Icarus verilog: Open-source verilog more than a year later," *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.