Towards a Hardware Trojan Detection Cycle

Adrian Dabrowski, Heidelinde Hobel, Johanna Ullrich, Katharina Krombholz, Edgar Weippl
SBA Research, Vienna, Austria
E-Mail: (firstletterfirstname)(lastname)@sba-research.org

***Abstract*—Intentionally inserted malfunctions in integrated circuits, referred to as *Hardware Trojans*, have become an emerging threat. Recently, the scientific community started to propose technical approaches to mitigate the threat of unspecified and potentially malicious functionality. However, these detection and prevention mechanisms are still hardly integrated in the industry's Hardware development life cycles. We therefore propose in this work a secure hardware development life cycle that assembles methods from trustworthy software engineering.In addition to full traceability from specification to implementation, and down to each gate, we introduce a feedback detection cycle that systematically escorts every single step of the development process. To do so, we integrate different detection methods for each development phase that are derived from a common knowledge base.**

I. Introduction

Software malware has become a daily threat to computer systems and a lot of effort is put into adequate countermeasures. Yet computer systems not only consist of software, but also of hardware. Benign functionality can be realized in hardware or software, likewise, malware can appear as a part of software or hardware.

Wang et al. [1] defined hardware Trojans as hardware modifications which result in malicious functional changes of the respective device. Considering the propagation of integrated circuits in today's world – including domestic appliances like washing machines, means of transportation (e. g. cars or airplanes), clinical devices (e. g. enabling precise diagnosis) and military appliances (e. g. enhancing the effectiveness of weapons) – hardware Trojans impact our everyday life and may even cause life threatening situations. Unlike other errors and malfunctions, Trojans are inserted deliberately. Apart from insider attacks, the economically driven outsourcing of production steps to third party contractors enlarges the attack surface dramatically. Contractors, their employees, and intruders potentially modify the design without the designer's or customer's knowledge. Hitherto existing mitigation approaches (such as [2]) introduce additional manual reviews in different stages of the process, but do not develop specific measures for hardware Trojan detection.

Based on the experience of trustworthy software engineering, the adoption of respective techniques for the hardware development process seems appropriate to encounter the challenges of silicon malware. Due to inherent differences between these domains, the adoption is a non-trivial task. Hardware offers only scarce possibilities of updating after deployment. Furthermore, hardware attacks are typically targeted attacks, i. e. tailored to a specific victim or product, while software malware targets a broad mass of anonymous victims.

Additionally, a drastic change of the familiar and well understood hardware development process requires high financial and organizational effort and therefore unlikely to be applied. Hence, we strove to moderately extend the current process by adapting selected methods known from secure software development.

As a first step, we analyzed typical malware structures to infer the demands for successful detection of hardware Trojans. Then we identified shortcomings in state of the art hardware development processes and evaluated trustworthy software engineering processes in terms of their applicability to the hardware process. Successfully evaluated methods are adopted and then included in the hardware development cycle.

By understanding potential attacks and their point of insertion, we identified feasible methods and requirements for their proper appliance within the hardware development. Finally, the findings were assembled to develop the hardware Trojan detection cycle. Thereby it is taken into account that some properties of malware are easier to detect in artifacts of particular development phases. This is considered by a detection cycle containing adaptive phase-dependent rule sets.

This paper contributes

- a definition of a threat model to identify the demands for a secure hardware development process,
- the introduction of full traceability in hardware development, and
- a detection life cycle for malware in silicon to be included in today's state of the art development processes.

This paper is organized as follows: Section II presents state of the art hardware development processes, types of attackers, differences between hardware and software development as well as an introduction to trustworthy software engineering. Section III introduces the methodology and present the developed detection life cycle. Finally, Section IV provides an evaluation followed by a discussion of results, especially in terms of limitations. Section V provides an overview on related work with respect to hardware Trojans and trustworthy hardware development, followed by the conclusion in Section VI.

II. Background

In this section, we briefly describe the standard industrial hardware development lifecycle. Furthermore, we summarize the recently considered attack vectors, highlight the important differences between software and hardware development, and give an introduction to trustworthy software development.

This work was supported by the FIT-IT program (project number 835922) and the COMET K1 program by FFG (Austrian Research Funding Agency).

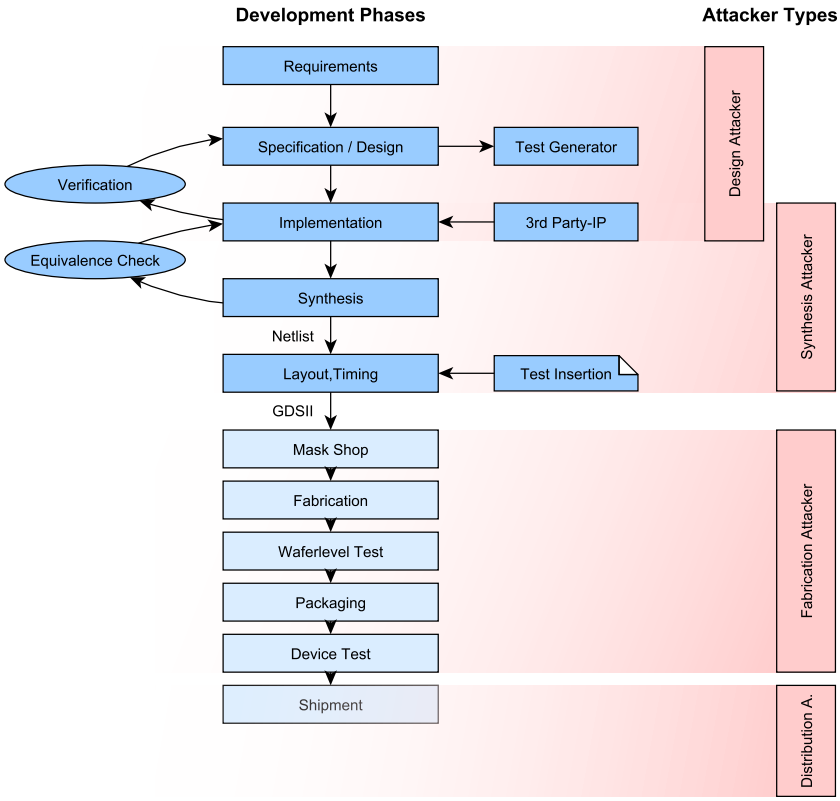

Fig. 1. Industrial Hardware Development Lifecycle

A. Industrial Hardware Development Lifecycle

Figure 1 illustrates the phases of a typical industrial IC development process. In the *requirements phase*, the core features of the future product are defined. In the following *specification and design phase* detailed descriptions and block diagrams are gained from the results of the first phase. Based upon that, disjunctive teams work on test cases and the implementation. During the *implementation phase* HDL (Hardware Description Language) code is written, but also external modules – 3rd party intellectual property (IP) – can be included. Depending on the license, the 3rd party IP can be anything from full HDL source code to a piece of netlist[1] or a development model, whereas the real IP is inserted directly into the mask later on. A *verification* mechanism tests the implementation against the specification in regular intervals (e. g. in nightly regression runs). This is also the phase, in which code reviews, testing and simulation take place.

After *synthesis*, the generated *netlist* is tested against the implementation using an *equivalence check*. The flat netlist is then laid out for a specific production technology and saved as GDS II (Graphic Database System) file. At this point, testing circuits are introduced which support the detection of production errors in later phases, not necessarily Trojans (*Test Insertion*). For a fab-less designer, i.e. an IC designer without fabrication facilities, this is the last step that is done in-house. The *tape out* marks the transfer of the IC design to external contractors. The *mask shop* creates the lithographic masks for the *fabrication*. Wafers are first tested as a whole, then individual ICs are tested. However, even designers with access to a fab source out some of the steps either for economic reasons or because some steps are better performed by very specialized and experienced contractors.

[1]Netlists are a low-level representation of hardware by means of nested graphs of the electronic circuit. They consist of gates and its connectivity wires.

B. Attack types

Attackers are divided into four general groups based on the development phase they are active in, i.e. *design attackers*, *synthesis attackers*, *fabrication attackers* and *distribution attackers* [3]. In Figure 1, these attacker types are visualized in relation to the hardware development phases they are active in and Figure 2 provides an overview.

A *design attacker* is active up to the implementation phase. Thereby, the attacker has full access to design files as well as source code. This access is gained by traditional hacking or by the attacker being an insider with legitimate access. He/She is able to add and remove components or to gain insights into the design for future attacks.

Synthesis attackers compromise CAD (Computer Aided Design) tools or scripts running them, which output a modified representation without modifying the source code. Due to being included into a synthesis tool, these attacks are difficult to discover. Thereby the attacker is able to add Trojan logic, mangle critical logic, metering IPs or theft of respective information.

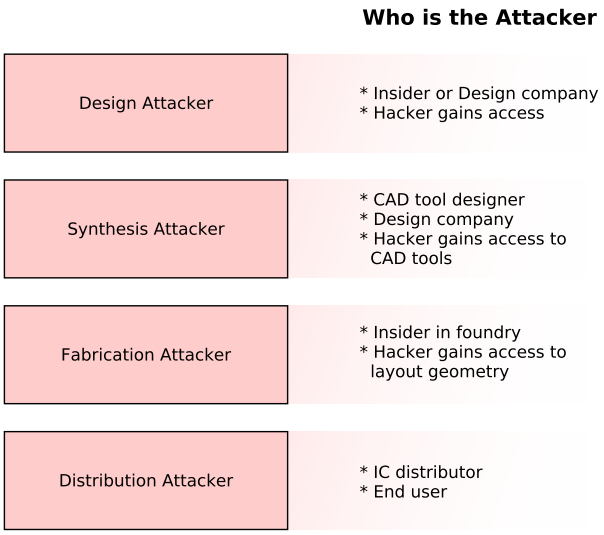

Fig. 2. Attacker Types, based on [3]

A *fabrication attacker* unfolds his/her activities after tape-out and is typically external to the IC designer. Attackers are able to remove/add components via layout geometry modification, reverse engineering or IC metering.

The fourth group, the *distribution attackers*, sell counterfeit products with modified circuitry.

C. Differences between Software and Hardware Development

In general, software engineering is more flexible than most engineering professions due to being independent of physical artifacts. This also leads to differences in the development process and does not allow the simple adoption of trustworthy software engineering to hardware development. Physically producing an ICs is a time-consuming and costly task. Furthermore, testing the final good requires significantly more resources than in software engineering. Thus, engineers try to mitigate all kind of errors before production by means of planing and simulation. This leads to a production process where the actual production remains a comparably small step at the end.

After roll-out of the respective product, updates and fixes such as performed regularly in operating systems and browsers are impossible because they would require physical changes or replacement which cannot be performed remotely.

Coherently with these aspects, iterative development processes are not fully applicable for hardware. As an extreme, agile software development methods start with a rough design, which expands modular and subsequently. The implemented modules are refined based on the use case just before implementation. In contrast, IC engineering prefers a more classic top-down engineering approach based on the waterfall model, where each phase is strictly performed after each other. Changes in previous phases are possible, but have to trickle down the waterfall and hold the risk of a long tail of other adoptions, such as retrofitting tests.

In a nutshell, IC development phases gain similarity to software development the further away the are from the final product. As a conclusion, methods from trustworthy software development are more likely to be adaptable for the first stages of hardware development.

D. Trustworthy Software Development Lifecycle

In software a *secure development process* is often referred to as *trustworthy*, since *secure* is also associated with a quality control so that the end product does not contain design- or implementation-specific security vulnerabilities, such as buffer overflows or command injections. In this paper we focus on the first interpretation, even though they often correlate.

The method of *Requirement Traceability* has been researched and popularized by Gotel and Finkelstein [4]. In brief, it describes the ability to track all people, decisions and artifacts that lead to a certain requirement, as well as all artifacts (e.g. code and tests) involved in fulfilling the requirement in the final product. The latter part is called *post requirements specification* and can be as detailed as for each single code block or code line.

A vast number of implementations exists for all popular development platforms. They typically bridge or unite other requirements, specifications, testing and source code versioning tools. The integration into the development environment (IDE) forces developers to stick to a certain work-flow. With an enforced work-flow, developers might not be able to commit changes into a source code management tool without logging into a specification, test case or change request. Together with debug symbols for the binary, this method creates an uninterrupted traceability from the requirement, through the specification, the test cases and the implementation down to the binary code and vice versa. Each compiler-generated CPU instruction can be traced to a specific source code line or module and then to all the authors, specifications, requirements, change requests and test cases associated with it.

The term *Continuous Integration* describes a software development infrastructure where code is automatically built and tested in short intervals - usually several times per day or at least every night (i. e. *nightly build*). This leads to a usable product very fast, but without the full feature set that grows additively, which allows for early testing. This method is often combined with *test-driven development*. Here, unit tests are written before the actual implementation and automatically tested.

III. Securing the Hardware Development

The goal is the enhancement of the state-of-the-art hardware development process to increase hardware security by adapting methods from trustworthy software engineering. However, due to the differences between hardware and software development (see Section II-C), the respective methods have to be chosen carefully. This way, an adequate hardware development process is constructed to decrease the possibility of Hardware Trojans.In detail, four steps are taken:

Threat modeling: Based on the attack types (see Section II-B) and Hardware Trojan descriptions, summarized by [5], we analyze their point of insertion. The entity of these points are referred to as *attack surface*. The analysis reveals the advantages and disadvantages of introducing malware at a certain

phase of the development process. Thereby, the perspectives of both stakeholders, i. e. developers and attackers, are included to get a comprehensive picture.

Adoption of methods: Methods which are identified as applicable, will have to be modified. In this step, we will prove the feasibility of the method's introduction and define further the requirements which have to be fulfilled.

Definition of detection cycle: After all pre-requirements have been presented, the detection cycle is assembled and described in detail. Further, the combination of automation and human intervention is explained.

Evaluation: The final evaluation demonstrates that the detection cycle fulfills the target to successfully detect hardware Trojans. Thereby we rely on test implementations of malware from a *Hardware Trojan Kit* [6]. We exemplify the evaluation of one Hardware Trojan implementation. The evaluation also reveals the limitations of our approach.

A. Threat Model

As described in Section II-B, malware can be introduced in different stages of the production process. Each stage has its own representation and bears its own risks and advantages for an attacker. Malfunctionality planted into the (machine readable) specification or design needs to be hidden in a very elaborate way, since specifications are seen and checked from designers, testers and developers, and are heavily reviewed.

An attack in the implementation phase (i.e. design attack) allows an attacker to access high level functions as well as low level signals. The main advantage for the attacker is the low effort of integrating the malfunction, but bears the risks of being detected by unit tests. In reaction, an attacker can insert its modifications into the glue logic between modules or spread parts of the Trojan across different modules.

Netlist modifications are hard to comprehend for humans and require the attacker to implement its Trojan in very low level terms. This can however lead to a lean and sophisticated modification with a minimum number of changed gates and interconnects. Besides that, the synthesis tool itself can be Trojanized and simply include and hide the hardware Trojan everytime the chip synthesis is performed (synthesis attacker). Mask modifications and attacks during fabrication introduce even more subtle changes [7] to the circuit, but require a very deep understanding of the circuitry and the production process (fabrication attacker).

We assume that the earlier in the production stage a Trojan is inserted, the more hints for its existence will be scattered through the project artifacts.

B. Traceability in Hardware

We propose to use traceability in hardware development like in software development (Section II-D). The detailed specification is developed based on the requirements and recorded in a first table (Figure 3, left table). Every specification document has its own history. Single authors are linked to single specifications or paragraphs. These specifications are the basis for creating tests. Test cases are related to the specification points they cover (middle table), the involved test engineers and later on also to the covered source code lines. Implementation is again based on the specifications. A source code management system is able to track each version and to relate source code parts to single authors and the specification or change request, effectively ensuring traceability of every revision of each source code line.

Even the frequent use of 3rd party IPs does not require special treatment, as third-party libraries are also known in software development. However, the circuit- and netlist generation work in a completely different way than in software development. The generated logic and circuitry tend to be heavily optimized. *Nested netlists* (often generated for visualization) contain some meta-information (e.g. the module or process name). In this stage it is trivial to attach source code references to the elements - in fact many of today's development environments do this to some degree. However, flattened or technology-mapped netlists are optimized regardless of the module's boundaries. Depending on the target platform, the output of the synthesis might be individual gates or lookup-tables (in FPGAs). An optimizer has to merge these source code reference labels accordingly, e. g. labels for merged elements accumulate in the resulting element. However, the output signal of an entirely removed individual element is typically substituted with a static connection to either logic 0 or 1 as the input for the next element. If the latter, this input inherits the labels, not the global logic 0 or 1 source. In other cases, such as entirely removed address bus lines accompanied by resized or removed address decoders, collecting all labels doesn't seem so helpful and needs some balance.

Preserving meta-data in external facilities provides some additional challenges. For example, a standardized extension to the GDS II (Graphic Database System) file format is required. GDS II is the de facto industry standard for IC layout-related data.

We believe that the novel approach of uninterrupted traceability from requirements to source code to each individual transistor is very helpful in a number of (debug) tasks, not only in the particular one we describe in the next section.

C. Detection Cycle

As attacks occur on many different levels and in different development phases providing various possibilities of detection, we propose a multi-phase detection feedback cycle - similar to those used for machine learning (Figure 4).

In a first step, a knowledge base for properties and working principals of Hardware Trojans is implemented. It includes expert knowledge, theoretical and practical descriptions from literature, real world examples, implementations and lessons learned example implementations (see Section IV-A for further explanations and examples).

From this knowledge base, a set of rules and patterns is extracted for the different design phases and verification steps. These include but are not limited to: design rule checks, negative source code patterns, negative netlist sub-graphs, structural rules, and formal verification rules. These rules are then applied to the design and implementation artifacts in regular time intervals (e.g. in automated nightly test runs). It automatically selects the appropriate examination methods for the right development phase.

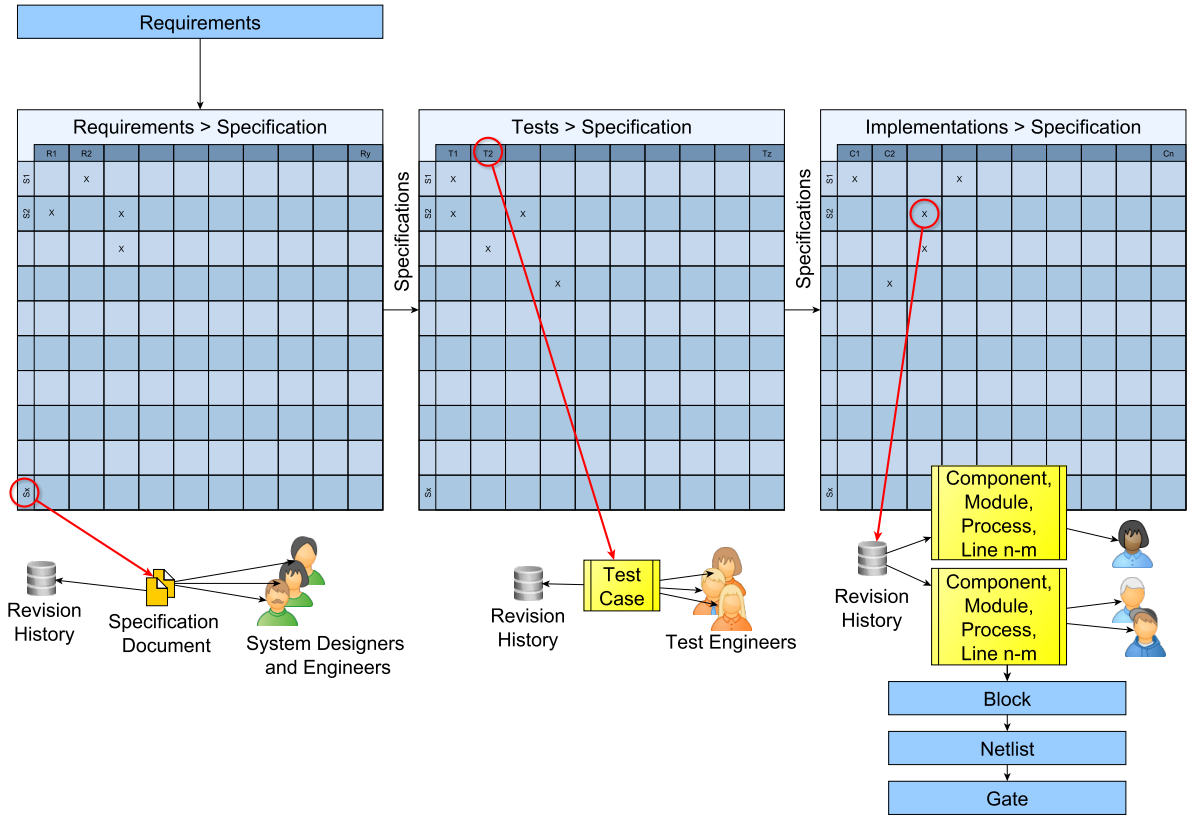

Fig. 3. Proposed traceability in hardware development

Suspicious structures flagged by the testing procedures are then presented to a senior tester or senior quality assurance engineer. Thanks to the previously introduced full traceability, they can trace back these structures to the source code, the author(s) of the appropriate lines, every change that was made (history), the design requirement supposedly covered by the code, and the respective test cases. After the assessment the tester either flags the structure as malicious or as a genuinely wanted artefact. The latter is followed by an error analysis. As the precision of detection rules is not perfect and false-positives are going to occur, this information is fed back into the loop. In case the incident is specific to the project, an exception into the rule set is added or a specific construct (e.g. netlist area or lines of source codes) is white-listed. If the findings (true- or false-positive) lead to a new insight or knowledge, the knowledge base is extended.

IV. Evaluation and Discussion

For evaluation, we used the Hardware Trojan Kit [6] which allows the modular construction of hardware Trojans based on the attributes *activation*, *covert communication*, *payload* and *detection*. Based on the implemented modules, various rules and properties that are used to construct a first knowledge base for Trojan detection are inferred. While Trojans will rarely come in neatly capsuled modules, they are helpful in analyzing structures and generating new variants.

A. Setup

The modules were developed and tested on several Xilinx FPGAs. The sources as well as the synthesized artifacts were then analyzed for typical properties and characteristics and revealed typical malware structures, which may serve as a strong indicator to reveal malicious hardware and are provided in the following list:

- **Asynchronous latch:** A latch not clocked by one of the typically low number of global clocks.
- **Gated wire or output:** A signal filtered by another gate to falsify its output.
- **Ring oscillator:** A combinatorial loop without a constant frequency.
- **Unused pin or bond wire:** Can be used for dissemination, e. g. for electromagnetic radiation.
- **Hidden finite state machine (FSM) state(s):** Depending on the encoding (e. g. one-hot- v.s. sequential encoding) can be hard to detect.
- **Latch or Flipflop independent from global reset**
- **Local or gated clock:** Typically most gates are controlled by one of the global clocks.

B. Use-case

We exemplify our findings based on a ring oscillator (RO). In its simplest case, a RO is a cycle of an odd number of inverters feeding the output back to the input. It will then start to oscillate with a frequency depending on the gate and transmission delays. This specific structure can be found in some Trojan examples (e. g. side channels, independent

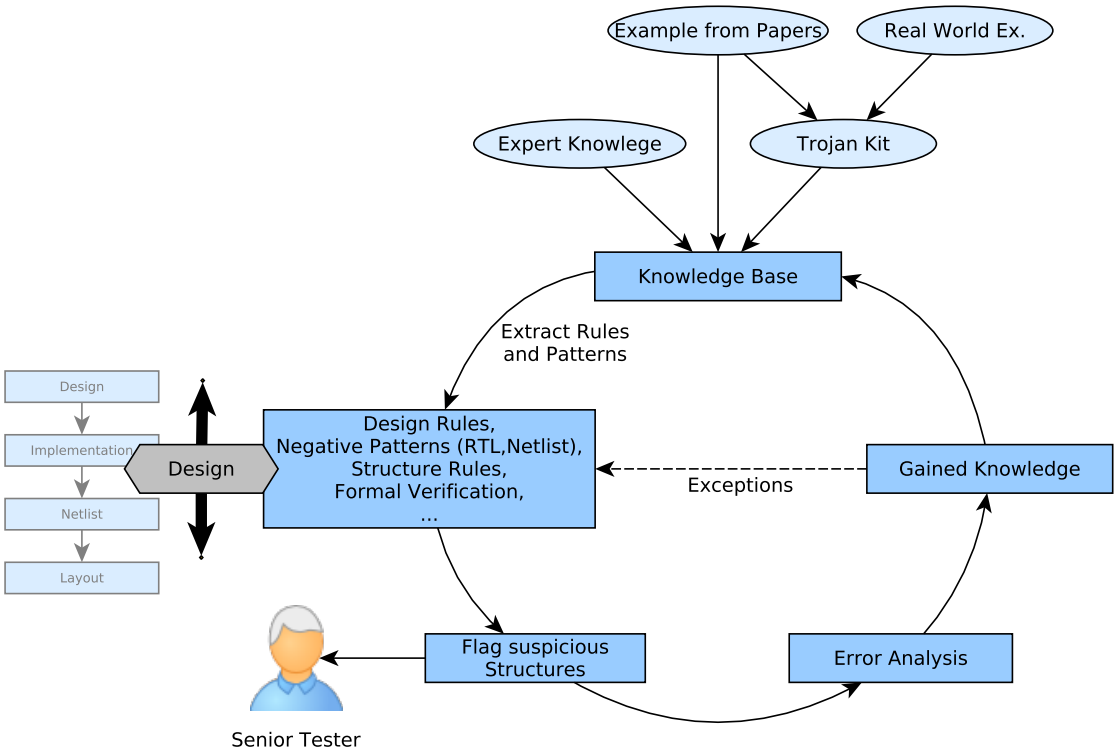

Fig. 4. Proposed detection cycle

clock, ambient temperature measuring) as well as legitimate uses (e. g. random number generators, physical unclonable functions). However, since this is a structure used in multiple Trojan examples, it is included into the knowledge base. Multiple rules can be inferred to detect such structures. We derived the following rules that are aimed at mitigating the risk of using a ring oscillator:

- In formal verification, a specific rule forbids sub-structures with outputs, but without digital inputs.
- On source code level, a rule reports the use of optimizing-inhibiting compiler- or source code flags[2]. In this special case, most synthesis tools will try to optimize `NOT(NOT(A))` back into `A`, effectively removing the RO.
- A structural graph pattern searches the netlist for ring sub-graphs of this type.

Equivalent rules have also been found for the other malicious structures described in Section IV-A, enabling their successful detection.

C. Discussion

As described in Section III-C, the detection life cycle accompanies the development process through different phases with a phase-dependent rule set derived from a common knowledge base. This way, our approach is effective against *design-*, *synthesis-* and partly against *fabrication attackers* because the life cycle is applied in the respective phases of the hardware development process. It will not work on *distribution attacks*.

We assume, that every included Trojan leaves some evidence or hint of its existence in the project's artifacts. However, each detection rule has only a certain probability of detecting a particular Trojan based on a hint. The more hints there are, or the earlier in the development the Trojan is inserted, the more scans they are going to be subject to. Therefore the rare case of multiple Trojans injected into one IC is going to leave more hints, thus making the detection even easier.

Another problem are dual-use structures. However, since we already established that a RO as well as all of the patterns listed above might have legitimate uses, the flagged structures are presented to the senior engineer for approval.

The above example also demonstrates why it might be favorable to have stricter rules: it is better to produce false-positives (which are then reviewed) than to produce false-negatives that might slip through the detection process.

D. Limitations

One limitation of our detection cycle approach is that it depends on the quality of the verification, property and rule sets to detect Trojans. Therefore it is designed as feedback loop: as it is used, it accumulates new knowledge, refines its rules and matures over time. Many of the examinations and checks can be automated. It therefore complements approaches like [2] and partly incorporates [8]. Rule sets and knowledge base can be shared among industry, similar to personal computer anti-virus companies sharing their knowledge and fingerprints with each other.

[2] e. g. `keep`- and `noprune`-attributes in VHDL

V. Related Work

A number of publications has been presented to provide methods for Hardware Trojan detection. Applying *formal verification*, i. e. verifying the equivalence of two design representations, leads to the method of *Structural Checking* [9], [10]. Other methods target Trojan activation by finding the rare events which serve as trigger. Additional methods reduce the overall test effort, e. g. via statistical analysis or state space obfuscation [11], [12]. It is also possible to compare physical parameters of a chip to a Trojan-free reference chip – the *golden model* – allowing the detection of side channels [13]–[15]. *Invasion* refers to the insertion of additional circuits into a design without changing its original functionality in order to test it after production [13], [16]. In combination with Trojan detection, their localization is also of interest. Thereby, *activation* means maximizing the Trojan activity while reducing the remaining parts' activity [17]–[19], whereas *mensuration* means gaining the regional information from side channel measurements [14], [15]. All these papers focus on the technical aspects of detection and mitigation strategies. However, currently this knowledge has not been systematically integrated into development processes, which is required to encounter Trojans in an organized way. Our life cycle allows to systematically integrate them now.

In [2] Khattri et al. present a Hardware Security Development Lifecycle. Identifying the lack of adequate tools for static analysis of hardware description languages and a comprehensive collection of hardware threats, a life cycle consisting of five phases was developed based on the experience of secure software engineering. An initial *security assessment* is followed by an *architecture review* and a *design review*. Finally, pre-silicon and post-silicon testing is performed in an *implementation review* and in a *penetration test*. It remains unclear how this penetration testing works without a comprehensive thread collection. Our approach overcomes this by adopting a reinforcement learning technique into the detection cycle, similar to the one used in machine learning. Thereby, the lack of a threat collection is handled by creating it during the iterations of the presented process.

VI. Conclusion

Hardware Trojans are a type of malware which are realized in silicon and are a severe thread to our daily life. While sophisticated experience and knowledge regarding secure software development are available, respective counterparts for hardware developing is still lacking.

In this paper, we proposed a Hardware Trojan Detection cycle which is applicable in the traditional hardware development process. Within the iterative cycle, rules and patterns leading to a rule set are extracted from a knowledge base. Automatic tests flag suspicious structures, which are forwarded to a senior tester. He/She is able to trace back these structures and decide whether they are malware. If identified as malware, an error analysis is performed and the gained knowledge is fed back to the knowledge base. The detection cycle accompanies the development phases while constantly extending its knowledge and adapting. Thereby it is taken into account, that some properties of malware are easier to detect in artifacts of particular development phases which is considered by a detection cycle containing adaptive phase dependent rule sets. It is effective against design and synthesis attackers and partly against fabrication attackers.

Being a precondition for the appliance of the Detection cycle, we introduced the novel technique of gap less traceability from every requirement of the specification down to each single transistor. Its implementation demands extensions to today's synthesis tools. Further, it will require a gentle tradeoff between label explosion due to optimization and the level of insight. Once implemented it is able to provide valuable insight and debug features even beyond the purpose of Hardware Trojan detection, e. g. re-usability of modules.

Acknowledgments

This work was supported by the FIT-IT program (project number 835922) and the COMET K1 program by FFG (Austrian Research Funding Agency).

References

[1] X. Wang, S. Narasimhan, A. R. Krishna, T. Mal-Sarkar, and S. Bhunia, "Sequential hardware trojan: Side-channel aware design and placement," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, 2011, pp. 297–300.

[2] H. Khattri, N. K. V. Mangipudi, and S. Mandujano, "Hsdl: A security development lifecycle for hardware technologies," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 116–121.

[3] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A case study in hardware Trojan design and implementation," in *International Journal of Information Security*, 2010, vol. 10, pp. 1–14.

[4] O. C. Gotel and C. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*. IEEE, 1994, pp. 94–101.

[5] C. Krieg, A. Dabrowski, H. Hobel, K. Krombholz, and E. Weippl, "Hardware malware," *Synthesis Lectures on Information Security, Privacy, and Trust*, vol. 4, no. 2, pp. 1–115, 2013.

[6] A. Dabrowski, P. Fejes, J. Ullrich, K. Krombholz, H. Hobel, and E. Weippl, "Poster: Hardware trojans - detect and react?" in *Network and Distributed System Security (NDSS) Symposium, 2014, Extended Abstract and Poster Session*. Internet Society, 2014.

[7] G. Becker, F. Regazzoni, C. Paar, and W. Burleson, "Stealthy dopant-level hardware trojans," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, ser. Lecture Notes in Computer Science, G. Bertoni and J.-S. Coron, Eds. Springer Berlin Heidelberg, 2013, vol. 8086, pp. 197–214.

[8] M. Rathmair and F. Schupfer, "Hardware trojan detection by specifying malicious circuit properties," in *Proceedings of 2013 IEEE 4th International Conference on Electronics Information and Emergency Communication*, 2013, pp. 394 – 397.

[9] S. Smith and J. Di, "Detecting Malicious Logic Through Structural Checking," in *IEEE Region 5 2007: Proceedings of the Region 5 Technical Conference*, 2007, pp. 217–222.

[10] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *HOST 2011: Proceedings of the IEEE Hardware-Oriented Security and Trust Symposium*, 2011, pp. 67–70.

[11] R. S. Chakraborty and S. Bhunia, "Security against hardware Trojan through a novel application of design obfuscation," in *ICCAD 2009: Proceedings of the International Conference on Computer-Aided Design*, 2009, pp. 113–116.

[12] M. Banga and M. Hsiao, "A region based approach for the identification of hardware Trojans," in *HOST 2008: Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 40–47.

[13] C. Lamech, R. Rad, M. Tehrani, and J. Plusquellic, "An Experimental Analysis of Power and Delay Signal-to-Noise Requirements for Detecting Trojans and Methods for Achieving the Required Detection Sensitivities," in *IEEE Transactions on Information Forensics and Security*, 2011, vol. 6, pp. 1170–1179.

[14] X. Zhang, N. Tuzzio, and M. Tehranipoor, "Red team: Design of intelligent hardware trojans with known defense schemes," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, 2011, pp. 309–312.

[15] F. Koushanfar and A. Mirhoseini, "A Unified Framework for Multimodal Submodular Integrated Circuits Trojan Detection," in *IEEE Transactions on Information Forensics and Security*, 2011, vol. 6, pp. 162–174.

[16] H. Salmani, M. Tehranipoor, and J. Plusquellic, "A Novel Technique for Improving Hardware Trojan Detection and Reducing Trojan Activation Time," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2011.

[17] M. Banga and M. Hsiao, "A Novel Sustained Vector Technique for the Detection of Hardware Trojans," in *VLSI Design 2009: 22nd International Conference on VLSI Design*, 2009, pp. 327–332.

[18] H. Salmani, M. Tehranipoor, and J. Plusquellic, "A layout-aware approach for improving localized switching to detect hardware Trojans in integrated circuits," in *WIFS 2010: Proceedings of the International Workshop on Information Forensics and Security*, 2010, pp. 1–6.

[19] S. Wei, S. Meguerdichian, and M. Potkonjak, "Gate-level characterization: Foundations and hardware security applications," in *DAC 2010: Proceedings of the 47th Conference on Design Automation*, 2010, pp. 222–227.