IoT4CPS – Trustworthy IoT for CPS

FFG - ICT of the Future

Project No. 863129

Deliverable D4.2

Functional and formal checks

The IoT4CPS Consortium:
AIT – Austrian Institute of Technology GmbH
AVL – AVL List GmbH
DUK – Donau-Universität Krems
IFAT – Infineon Technologies Austria AG
JKU – JK Universität Linz / Institute for Pervasive Computing
JR – Joanneum Research Forschungsgesellschaft mbH
NOKIA – Nokia Solutions and Networks Österreich GmbH
NXP – NXP Semiconductors Austria GmbH
SBA – SBA Research GmbH
SRFG – Salzburg Research Forschungsgesellschaft
SCCH – Software Competence Center Hagenberg GmbH
SAGÖ – Siemens AG Österreich
TTTech – TTTech Auto AG
IAIK – TU Graz / Institute for Applied Information Processing and Communications
ITI – TU Graz / Institute for Technical Informatics
TUW – TU Wien / Institute of Computer Engineering
XNET – X-Net Services GmbH

For more information on this document or the IoT4CPS project, please contact:
Mario Drobics, AIT Austrian Institute of Technology, mario.drobics@ait.ac.at

The IoT4CPS project is partially funded by the "ICT of the Future" Program of the FFG and the BMK.

Document Control

Title: Functional and formal checks

Type: Public

Editor(s): Heribert Vallant (JR)

E-mail: heribert.vallant@joanneum.at

Author(s): Thomas Hinterstoisser (SAGÖ), Faiq Khalid (TUW), Stefan Mangard (TUG), Martin Matschnig (SAGÖ), Kai Nahrgang (JR), Katharina Pfeffer (SBA), Herbert Taucher(SAGÖ), Heribert Vallant (JR)

Doc ID: IoT4CPS-D4.2

Amendment History

Version	Date	Author	Description/Comments
V0.1	23.01.2019	Heribert Vallant	Initial version prepared
V0.2	08.01.2020	Heribert Vallant	Document structure updated
V0.3	11.02.2020	Kai Nahrgang, Heribert Vallant, Martin Matschnig, Thomas Hinterstoisser, Herbert Taucher	Chapter 4, 5, 6: Threat modelling, Penetration Test Catalogue and "Dynamically Exchangeable Runtime Checkers in HW"
V0.4	18.02.2020	Faiq Khalid	Chapter 2: Formal hardware property checks
V0.5	28.02.2020	Katharina Pfeffer	Chapter 7: Human Aspects in Automated Model Checking of Security
V0.6	02.03.2020	Stefan Mangard	Chapter 3: Formal verification of side-channel protected hardware implementation
V0.7	03.03.2020	Kai Nahrgang	Summary
V0.8	06.03.2020	Heribert Vallant	Executive Summary
V0.9	23.03.2020	Felix Strohmeier, Heinz Weiskirchner	Document review
V1.0	24.03.2020	Heribert Vallant	Review comments integrated
V1.1	15.04.2020	Heribert Vallant	Comments from AVL integrated

Legal Notices

Content

Abbreviations

3PIPs	3rd party IPs
API	Application Programming Interface
AES	Advanced Encryption Standard
ARM	Advanced RISC Machine
CoAP	Constrained Application Protocol
CPSs	Cyber Physical Systems
DoS	Denial of Service
DPR	Dynamic Partial Reconfiguration
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HT	Hardware Trojan
HK	Human Knowledge
IIoT	Industrial Internet of Things
ILANG	Intermediate Language
ISTQB	International Software Testing Qualifications Board
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
PSL	Property Specification Language
RNG	Random Number Generator
R/S	Rescaled Range
RTL	Register-Transfer Level
SDL	Security Development Lifecycle
SCA	Side-Channel Analysis
SMT	Satisfiability Modulo Theory
SoC	System on Chip
SSA	Static Single Assignment
SUT	System Under Test
TLS	Transport Layer Security
TRNG	True Random Number Generators

Executive Summary

Due to the rise of the Internet of Things, particularly with regard to Cyber Physical Systems, many common and best practice security analysis techniques are becoming obsolete. Thus, to ensure safety and security within the IoT4CPS project, novel approaches have been evaluated in this deliverable. Therefore, different approaches from hardware specific formal verifications and hardware property checks to software and architecture specific threat modelling approaches up to human factors have been analysed. During that, any approach was especially adapted to the CPS environment addressed in IoT4CPS.

At hardware level statistical traffic modelling of communication channels was used to detect hardware Trojans. Therefore Hurst exponent, spatial hop distribution and standard deviation were applied and their potential regarding different hardware Trojans benchmark examined. At the area of formal verification of side-channel protected hardware implementations, the possibility of first order masking with only two random one-bit masks was shown and demonstrated for the design of an AES-128 core and formally verified by the t-probing model. The verification of security properties in reprogrammable hardware has been shown. Therefore the Dynamic Partial Reconfiguration feature of FPGAs was leveraged to enable a set of hardware-based security checks within a resource-constrained device. Within IoT4CPS FPGA functions that assess the randomness of a bit sequence generated by a random number generator were developed and evaluated.

At software and architecture level the threat modelling approach using the STRIDE model was applied. Within this task the default modelling template was extended by integrating the threats which were identified in WP2 and by adding some device specific hardware threats. Thus a threat model, tailored for the IoT4CPS project, has been created and applied to AVL's Device Connect. With the help of various cyber security and software development experts, these threats were afterwards evaluated for their applicableness at the test domain. As a result, a penetration test catalogue containing several reusable test cases for the automotive industry was created.

Furthermore also human aspects in automated model checking of security protocols were addressed. Within this area, the terms of usable security for users and engineers, formal verification of human errors, modelling of human errors, error prone protocols as well as different model checkers were outlined.

1 Introduction

Through its connecting of resource-constrained devices with the global Internet, the Internet of Things (IoT) is rendering many traditional security analysis techniques inapplicable. This rises the need for novel approaches for formally analysing hardware, protocols and system architecture as well as generating test cases.

Side-channel and fault attacks are severe threats against cryptographic keys that are used in IoT devices to provide security. A central challenge when implementing countermeasures against these attacks is to verify that this implementation is correct and provides the desired level of protection. A standard approach is to prototype a design, to do actual measurements and analyses of the side-channel leakage. However, this is a very time consuming and incomplete approach, as testing is done for a specific attack setup and analysis technique. Very recently, progress has been made in using formal methods to show security properties of hardware implementations of countermeasures like masking [1]. Approaches like this allow verifying the security of implementations at design time and providing precise security bounds. Formal verification is also an established method of analysing communication protocols. Therefore, often the Dolev-Yao attacker model is used, assuming perfect cryptography but allowing an attacker to actively interfere in the protocol [2], uncovering weak points in the failing parts.

Strategic assurance also deals with automated test case generation, either as a) model based testing [3], b) code-based test case generation or c) other sources, like interface definitions or executable systems. In recent years, industry is beginning to adopt these approaches (e.g. by Microsoft [4]). Nonetheless, most tests during development in industry are still “handcrafted”. In research, test case generation techniques are evolving from pure functional testing into testing non-functional properties including security. In category c), recently promising results have been achieved using program-analysis guided random testing [5]. During test generation, such tools interact with a system under test (SUT) by generating command calls that are immediately executed, exploring and analysing the actual behaviour of the system under test. The obtained results dynamically form a model, guiding the further test case generation process. Additional analysis steps enrich the model and direct test case generation to relevant hot spots exhibiting a high risk of failure. The generated interaction sequences are stored as error-revealing (on failure) or as passing (on success) ones for regression testing. The guided test generation approach significantly increases coverage and achieves higher defect detection rates than comparable approaches [6].

This deliverable outlines the approaches that were undertaken in IoT4CPS at the area of Strategic Security Assurance covering hardware, protocols, system architecture and human aspects and is structured as follows:

Chapter 2 deals with formal hardware property checks whiles chapter 3 describes the formal verification approach of side-channel protected hardware implementations. Chapter 4 describes the use of dynamically exchangeable runtime checkers as hardware apps. In chapter 5 an IoT4CPS tailored threat modelling approach using the STRIDE model is shown. Based on this threat model in chapter 6 a penetration test catalogue containing several reusable test cases for the automotive industry was created. Chapter 7 discusses the human aspects in automated model checking of security protocols. Chapter 8 summarizes the deliverable.

2 Formal hardware property checks

The interconnectivity between several devices has raised security issues in the IoT, especially the network layer. The security issues in network layers include information stealing, communication channel jamming, spoofing, denial-of-service, etc. Traditionally, these security issues are addressed at the application layer, protocol layers, or system level. However, these techniques cannot ensure the trustworthiness of each component. For example, in the case of compromised hardware, i.e., a small piece of hardware that performs a security attack when it gets a trigger, known as Hardware Trojan (HT), these techniques cannot guarantee the privacy of information. Therefore, it is imperative to ensure the security of the communication at the hardware level.

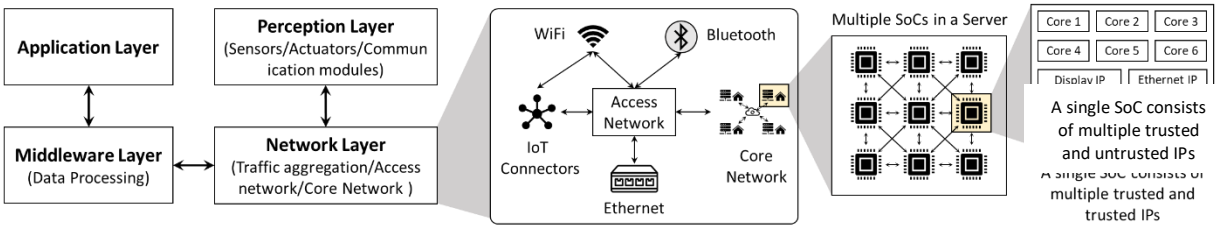


Figure 1: Thematic focus of the proposed technique that introduces the formal hardware checks to secure the SoCs.

The main focus of this work is to provide the security of the basic building block of every component in the network layer, i.e., system-on-chip (SoC), as shown in Figure 1. SoC consists of several trusted and untrusted 3rd party Party Intellectual Properties (3PIPs). Typically, the security of SoC is based on power, delay, and frequency signatures. However, in 3PIPs, it is nearly impossible to extract the golden signatures[1] , but in most of the 3PIP-based SoCs, the facility to integrate the IPs can be trusted. Various IP analysis-based approaches have been proposed to get the golden behaviour but these techniques are costly and cannot ensure the correctness of the golden signatures. To address this issue, several run-time detection approaches have been developed to monitor a SoC for its entire operational lifetime, providing an important last-line of defence. However, most of these techniques are based on side-channel analysis (SCA), which requires precise calibration and relies on the premise that a security attack generates a substantially higher current flow. To avoid the complex requirements of the SCA-based runtime HT detection techniques, we propose to use the communication behaviour because, in real-world scenarios, hardware modules are connected via communication channels and most of the intrusions have an impact on the communication behaviour without affecting the communication protocols. However, run-time monitoring of the communication behaviour of the trusted IPs poses the following research challenges:

1. How to extract the golden behaviour during the design time that can effectively be used for HT detection?
2. How to statistically model the communication behaviour of the trusted IP that can be used for runtime monitoring with minimum overhead?

[1] The normal parametric behavior of the SoC that is obtained from un-intruded IPs.

Note, these concepts can be leveraged to secure against other communication-dependent anomalies that impact the communication profile or signatures.

2.1 Methodology

To address the first challenge, in this work, we assume that at least one of the 3PIPs is trusted and also the SoC integration is performed in a trusted facility. Moreover, trusted 3PIPs are built in house and their communication behaviour is used as golden behaviour during runtime HT detection. To address the second challenge, we propose a novel methodology that leverages the statistical traffic modelling of communication channels in the SoC to sniff the possible anomalies in 3PIPs, named as SIMCom[2]. SIMCom consists of two major phases as depicted by the dotted rectangles in Figure 2.

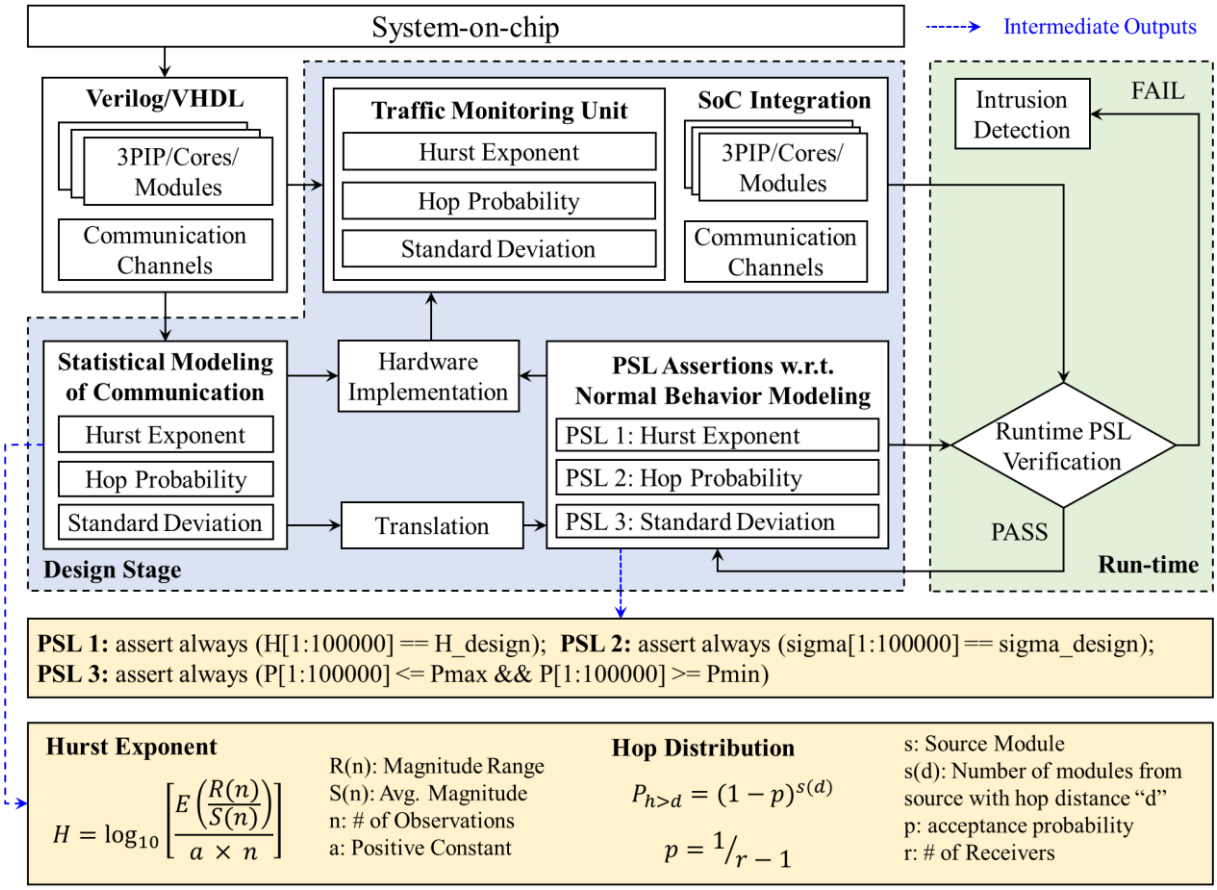


Figure 2: SIMCom: Statistical sniffing of inter-module communication for runtime HT detection.

During the design time, SIMCom requires the following steps to design the runtime monitors:

1. First, it extracts the communication behaviour of the trusted IP and statistically models the extracted communication behaviour of the trusted IP. Note, the traffic modelling is done under the premise that at least one of the IPs is trusted. The statistical modelling of the trusted IP can be used to detect HTs. The statistical communication behaviour of the trusted IPs in a SoC is obtained using the following statistical parameters.

[2] The archive version of the paper is also available online: F. Khalid, et al. "SIMCom: Statistical Sniffing of Inter-Module Communications for Run-time Hardware Trojan Detection." arXiv preprint arXiv:1901.07299 (2018).

a. The input packets are generated with respect to the standard spatial injection distribution, e.g., Gaussian distribution.
b. Next, the Hurst exponent is computed for each communication channel using the R/S method.
c. Hop distribution is computed using the total number of available communication channels and active communication channels.

2. Then, it uses the statistical model to define the corresponding property specification language assertions. These assertions are inserted into the RTL or Verilog/VHDL of the trusted IP.

The runtime verification of the statistical parameters-based PSL assertion requires the hardware modules that compute the above-mentioned statistical parameters, i.e., the Hurst exponent (H), probability of hop distribution (P), and standard deviation of input injection distribution (σ). Therefore, during the design time, the designer designs these hardware modules and integrated with critical communication channels.

During the run-time, the values of the statistical parameters computed from the corresponding hardware modules are used to verify the associated PSL assertions. Note, for secure communication: All the PSL assertions should be verified. If one of the assertions fails the verification, then the communication channel is considered as intruded. After the verification failures, any of the suitable state-of-the-art recovery mechanisms can be applied, i.e. re-routing the communication, backup communication paths, backup components that are associated with critical computations or communication.

2.2 Details of the Hardware Implementation

To extract the statistical parameters during the runtime for verification of the PSL assertions, we design the hardware modules for computing the Hurst exponent, hop distribution and standard deviation.

Hurst Exponent Estimation: To compute the Hurst exponent, we used one of the most commonly used methods, i.e. the R/S method. The reason behind choosing this method is that it requires less number of observations to estimate the Hurst exponent.

Algorithm 1: Estimation of Hurst Exponent

Input:

n: Number of the observations = 512

$X[0 : n-1]$: number of communication packets per observation

Generate three computation blocks using $X[0 : n-1]$

Computation Block 1:

Compute the mean and the standard deviation of $X[0 : n-1]$

Compute the mean centered series: $h[0 : n-1] = X[0 : n-1] - M$

Compute cumulative deviation by summing up the mean centered values:
$Y[0 : n-1] = \sum_{i=0}^{0} h[i], \sum_{i=0}^{1} h[i], ..., \sum_{i=0}^{n-1} h[i]$

Compute the Range (R) of $Y[0 : n-1]$

Compute R/S

Compute $E(R/S)_0$

Computation Block 2:

Compute the mean (M) and standard deviation (S) of $X[0 : (n/2)-1]$ and $X[n/2 : n-1]$

Compute the mean centered series:
$h_1[0 : (n/2)-1] = X[0 : (n/2)-1] - M$ and $h_2[n/2 : n-1] = X[n/2 : n-1] - M$

Compute cumulative deviation by summing up the mean centered values:
$Y[0 : (n/2)-1] = \sum_{i=0}^{0} h_1[i], \sum_{i=0}^{1} h_1[i], ..., \sum_{i=0}^{(n/2)-1} h_1[i]$
$Y[n/2 : n-1] = \sum_{i=n/2}^{(n/2)} h_2[i], \sum_{i=n/2}^{(n/2)+1} h_2[i], ..., \sum_{i=n/2}^{n-1} h_2[i]$

Compute the Range (R) of $Y[0 : (n/2)-1]$ and $Y[n/2 : n-1]$

Compute $(R/S)_1$ and $(R/S)_2$

Compute $E(R/S)_1 = \frac{(R/S)_1 + (R/S)_2}{2}$

Computation Block 3:

Compute the mean (M) and standard deviation (S) of $X[0 : (n/4)-1]$, $X[n/4 : (n/2)-1]$, $X[n/2 : (3n/4)-1]$ and $X[3n/4 : n-1]$

Compute the mean centered series:
$h_1[0 : (n/4)-1] = X[0 : (n/4)-1] - M$, $h_2[n/4 : (n/2)-1] = X[n/4 : (n/2)-1] - M$,
$h_3[n/2 : (3n/4)-1] = X[n/2 : (3n/4)-1] - M$ and $h_4[3n/4 : n-1] = X[3n/4 : n-1] - M$

Compute cumulative deviation by summing up the mean centered values:
$Y[0 : (n/4)-1] = \sum_{i=0}^{0} h_1[i], \sum_{i=0}^{1} h_1[i], ..., \sum_{i=0}^{(n/4)-1} h_1[i]$
$Y[n/4 : (n/2)-1] = \sum_{i=n/4}^{n/4} h_2[i], \sum_{i=n/4}^{(n/4)+1} h_2[i], ..., \sum_{i=n/4}^{(n/2)-1} h_2[i]$
$Y[n/2 : (3n/4)-1] = \sum_{i=n/2}^{n/2} h_3[i], \sum_{i=n/2}^{(n/2)+1} h_3[i], ..., \sum_{i=n/2}^{(3n/4)-1} h_3[i]$
$Y[3n/4 : n-1] = \sum_{i=3n/4}^{3n/4} h_4[i], \sum_{i=3n/4}^{(3n/4)+1} h_4[i], ..., \sum_{i=3n/4}^{n-1} h_4[i]$

Compute the Range (R) of $Y[0 : (n/4)-1]$, $Y[n/4 : (n/2)-1]$, $Y[n/2 : (3n/4)-1]$ and $Y[3n/4 : n-1]$

Compute $(R/S)_1$, $(R/S)_2$, $(R/S)_3$ and $(R/S)_4$

Compute $E(R/S)_2 = \frac{(R/S)_1 + (R/S)_2 + (R/S)_3 + (R/S)_4}{4}$

Hurst Exponent:

Compute $E(R/S) = \frac{E(R/S)_0 + E(R/S)_1 + E(R/S)_2}{3}$

Compute Hurst exponent $H = 0.37 \times logE(R/S)$

1. After establishing a communication channel, it observes and stores the number of the packets per clock cycle, as denoted by X in Algorithm 1.

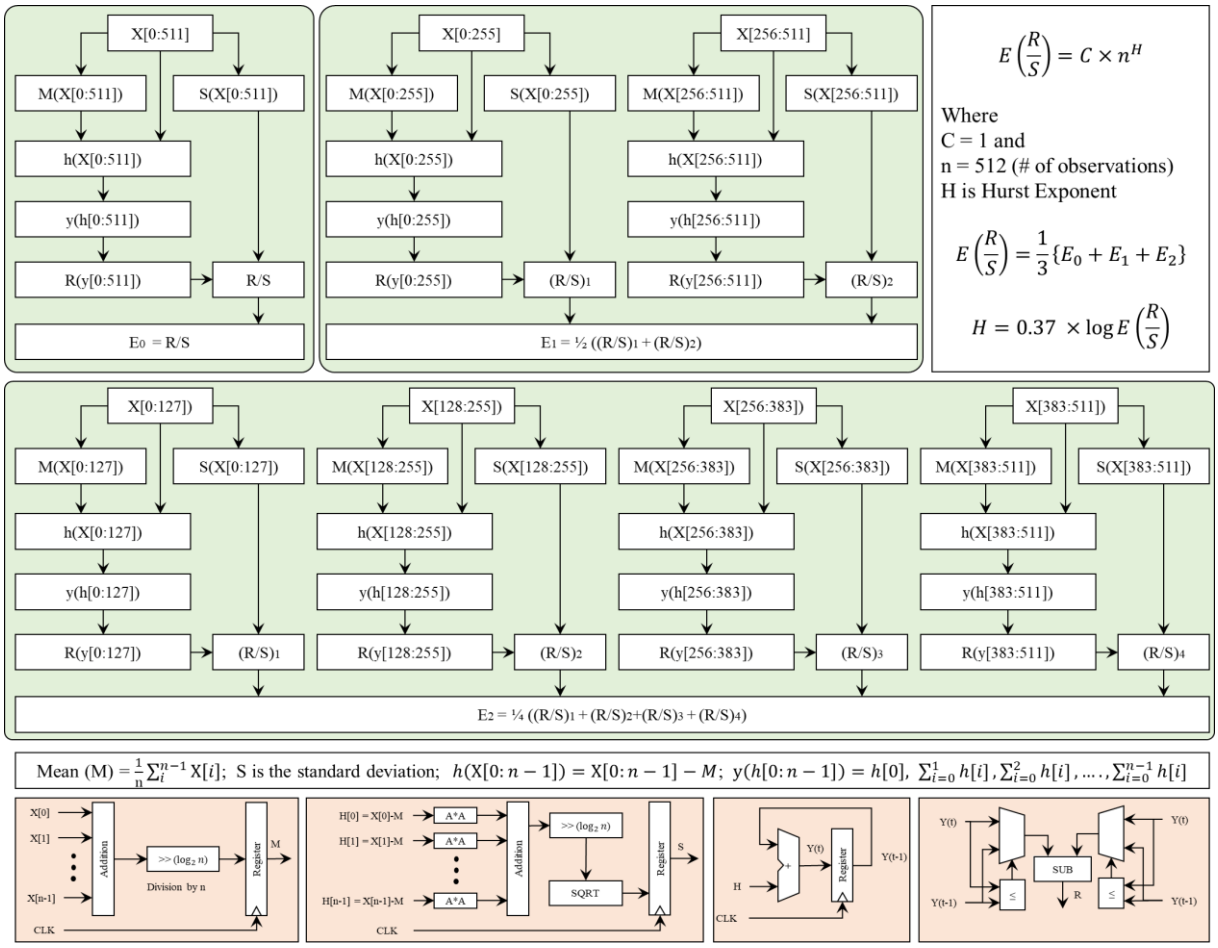


Figure 3: Data flow of the hardware module to compute the Hurst exponent.

In SIMCom, we set the number of observations to 512. The reason behind choosing this value is that for the fast convergence of Hurst exponent, the number of observations should be higher than 300, with minimum memory overhead. Note, to estimate the Hurst exponent, we used one of the most commonly used methods, i.e., the R/S method. The reason behind choosing this method is that it requires fewer number of observations to estimate the Hurst exponent. We implemented a state-of-the-art modified non-restoring algorithm for computing the square root function in standard deviation.

2. To estimate the Hurst exponent, it is important to take the average value of R/S values using the different data distribution obtained from the same data. Therefore, we propose to use three computational blocks, as shown in algorithm 1 and Figure 3. The data flow and hardware modules in each computational block are the same, but the sizes of hardware modules are different. The computational block 1 uses the complete data for estimating the R/S value. The computational block 2 divides the data into two equal parts. Then this block estimates the respective R/S values using each half data and takes the average to estimate the R/S value. Similarly, the computational block 3 repeats the same procedure, but it divides the data into four equal parts.
3. In each computational block, the first step is to compute the mean value (M) and standard deviation (S) of the input data series X. The hardware modules to compute the mean value consists of “n” 64-bit adders, one 9-bit shifter, and one output register, as shown in Figure 3. The hardware module for

computing the standard deviation consists of "n" 64-bit subtractors, "n" 64-bit multipliers, one 9-bit shifter, one module to compute the square root and one output register. Note, we implemented a state-of-the-art modified non-restoring algorithm for computing the square root function in standard deviation.

4. After computing the mean and standard deviation, each computational block computes the mean-centered data series (H) by subtracting the mean value from each input data series (X), as shown in algorithm 1. Then, it computes the cumulative deviation (Y) by summing up the mean-centered data series (H) and computes the magnitude range (R) of the cumulative deviation (Y). Finally, it computes the R/S using a 64-bit divider. Note, there is no extra hardware for the mean-centered series and each computational block uses the values from the standard deviation module. The hardware module of computing Y consists of one 64-bit accumulator, as shown in Figure 3, and the hardware module computing R consists of two comparators, two multiplexers, and one 64-bit subtractor.
5. Finally, it computes the R/S by taking the average of R/S values computed from each computational block. Finally, the average R/S value is used to calculate the Hurst exponent using $H = 0.37 \times \log_{10}(\text{the average value of R/S})$, as shown in algorithm 1 and Figure 3.

Standard Deviation: We have not used a separate block for computing the standard deviation. We use the intermediate output from the standard deviation block of Hurst exponent.

Hop Probability: In order to compute the hop probability, SIMCom computes the number of active channels by counting the acknowledgment signals during the channel establishment. Note, this parameter is effective in the case of denial-of-service, jamming or communication blocking.

2.3 Experimental Results

To illustrate the scalability of the SIMCom, we implemented the three SoCs and tested it for all Trust-Hub HT benchmarks [7], as shown in Figure 4

- SoC1 consists of four single-core MC8051 with UART modules.
- SoC2 consists of four single-core MC8051 linked with each other and AES, Ethernet, memctrl, BasicRSA, RS23s modules.
- SoC3 consists of four single-core LEON3 connected with each other and AES, Ethernet, memctrl, BasicRSA, RS23s modules.

Moreover, we synthesized the SoCs using Cadence Genus (Encounter) tool with the TSMC 65nm library.

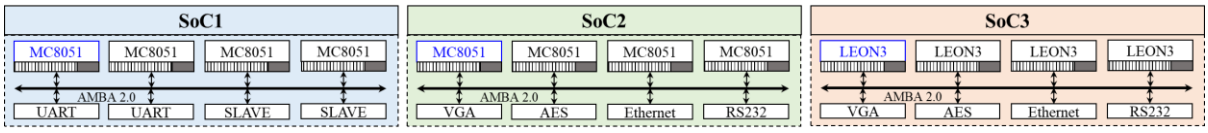


Figure 4: Implemented SoCs to illustrate the effectiveness of the SIMCom.

2.3.1 Experimental Analysis

To elaborate on the practicality of SIMCom, we have computed the statistical communication behaviour of the case study for 100,000 clock cycles, as shown in Figure 5. The figure depicts the communication behaviour with respect to H, σ, and P. This run-time analysis exhibits the following key observations:

1. The Trojan benchmarks exhibit a significant impact on the Hurst Exponent depending on the input data distribution (i.e., Gaussian or Exponential). For instance, in Figure 5, if the input data distribution is Gaussian, then a few of the implemented Trojan benchmarks (i.e., MS8051-T400, T500, and T700) have a significant impact on the Hurst exponent (see: label 1). However, if the input data distribution is exponential, then almost all the HTs exhibit a significant impact on the Hurst exponent (see: label 2).

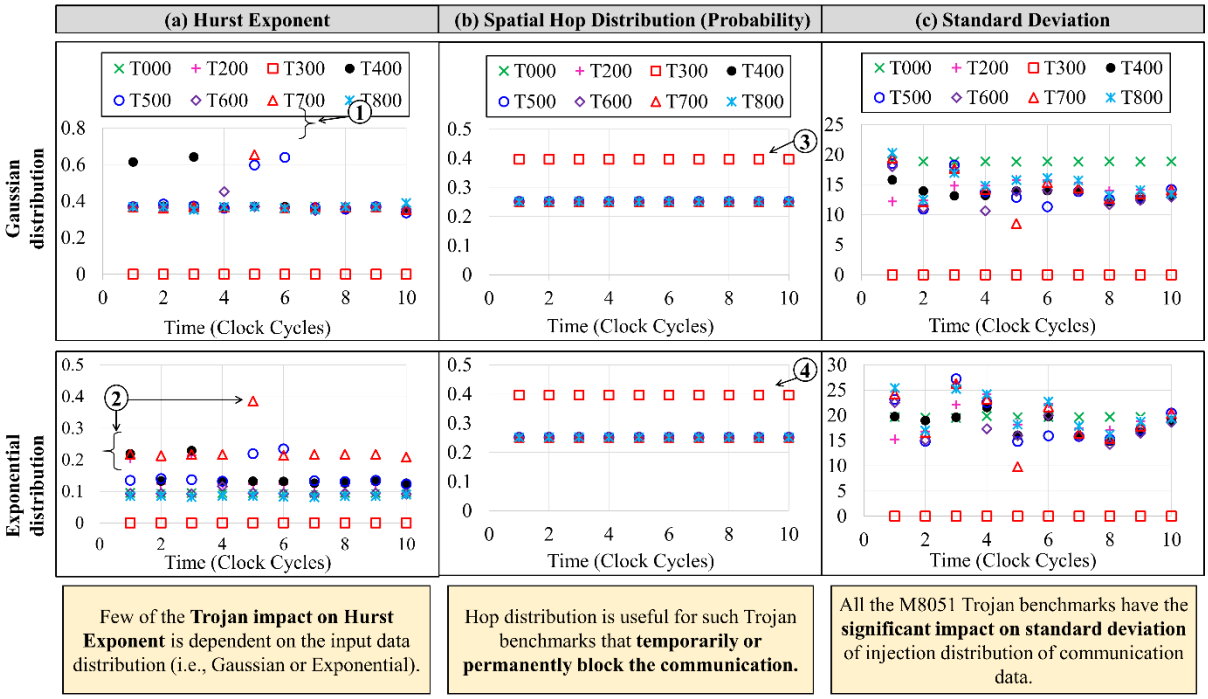


Figure 5: Runtime Impact Analysis of implemented HTs (i.e., MC8051-T200, T300, T400, T500, T600, T700, T800) on the statistical model (H, P, σ) of the implemented case study for 100,000 clock cycles and the values are averaged out for 10,000 clock cycle duration.

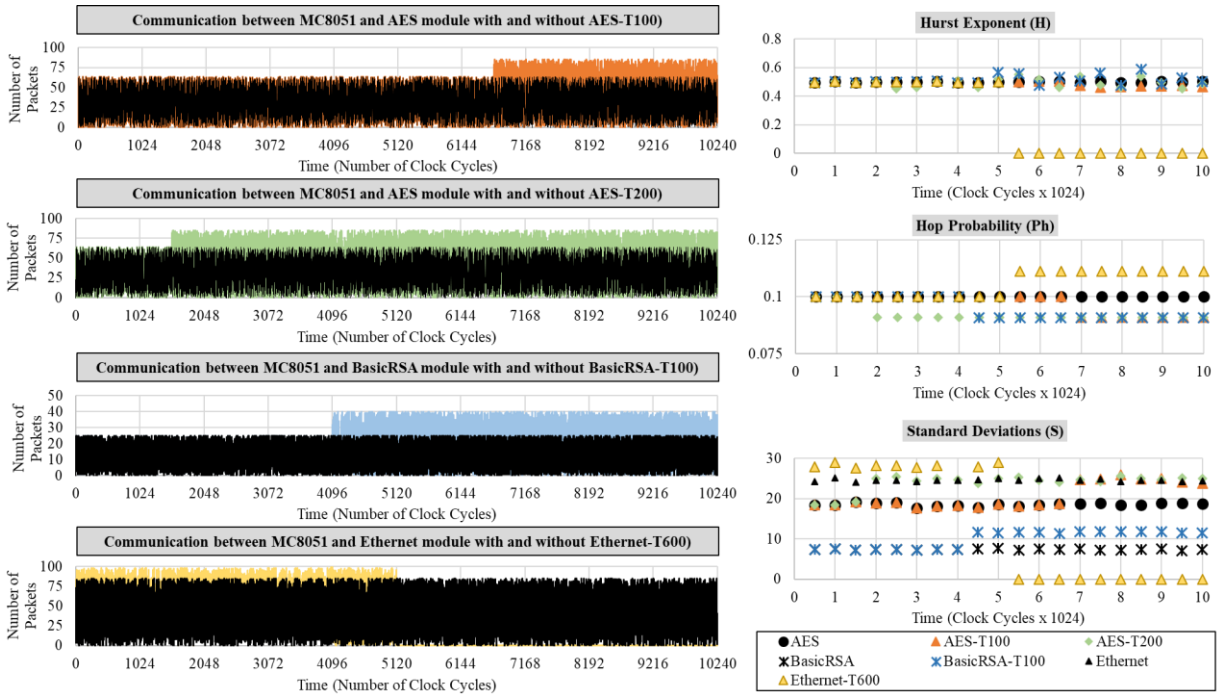


Figure 6: Experimental results to show the effect of HT benchmarks, i.e., AES-T100, AES-T200, BasicRSA-T200, and EthernetMAC10GE-T600, on communication between **MC8051** and AES, Ethernet and basic RAS module respectively. Note, in this analysis, the number of the observations for Hurst exponent estimation is 512.

2. All the implemented Trojans exhibit no impact on the probability of hop distribution, except the MC8051-T300 because when it is triggered, it halts a communication channel, as shown in Figure 5 (see: labels 3 and 4).
3. All the implemented Trojans deviate from the original values because all of them affect the input data injection, as shown in Figure 5. For instance, MS8051-T500 and MS8051-T700 replace valid data with intruded data. However, MC8051-T400 disables the interrupt handling, MS8051-T600 interrupts the jump, and MS8051-T800 manipulates the stack pointer, which indirectly disrupts the input data or respective control modules.

The values of H and σ for MC8051-T300 are “0” because upon triggering MS8051-T300 blocks the UART communication altogether, and hence no data traffic flows at all, resulting in the corresponding '0' values for H and σ. Similar observations are noted in the case of SoC2 and SoC3, as shown in Figure 6 and Figure 7, respectively.

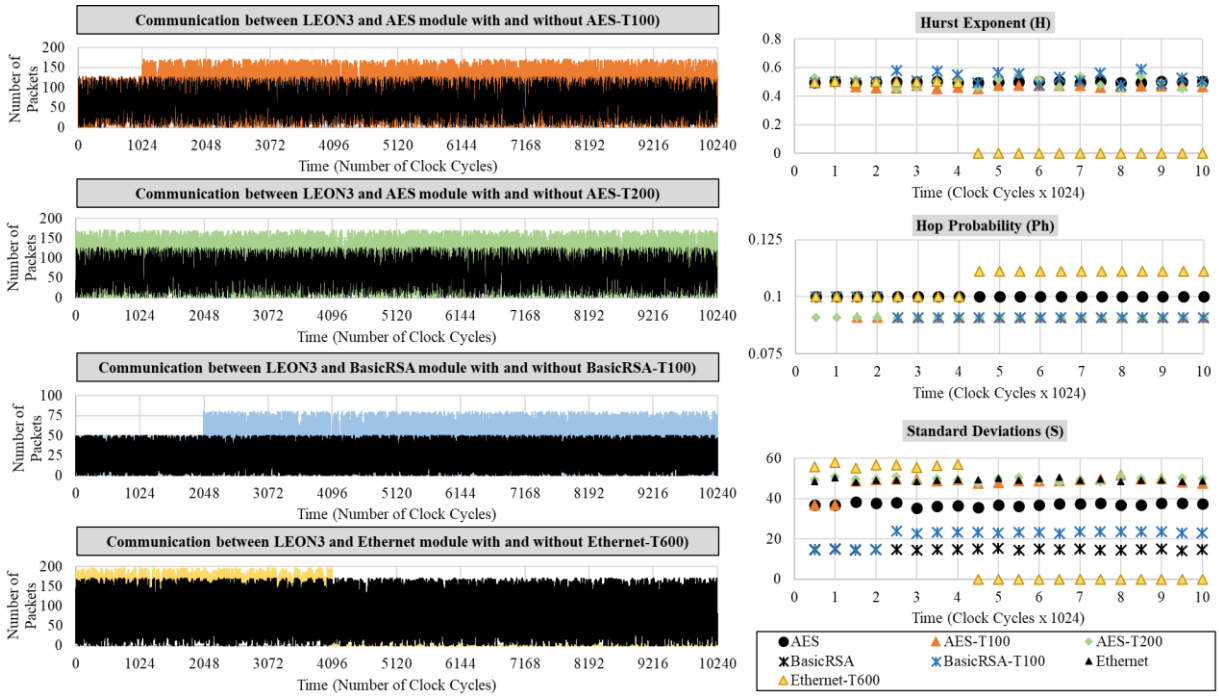


Figure 7: Experimental results to show the effect of HT benchmarks, i.e., AES-T100, AES-T200, BasicRSA-T200, and EthernetMAC10GE-T600, on communication between **LEON3** and AES, Ethernet and basic RAS module respectively. Note, in this analysis, the number of the observations for Hurst exponent estimation is 512.

2.3.2 Overhead Analysis

The overhead analysis presented in Figure 8 shows that the overhead associated with SIMCom in the SoCs with the same number of communication channels relatively decreases with the increase in the complexity of the modules. For example, in SoC3, the area and power overhead is less than 1%. Note, if the number of communication channels increases, the overhead of the SIMCom also increases. To address this issue, SIMCom can use the state-of-the-art methodology for distributing the runtime monitors in the SoC.

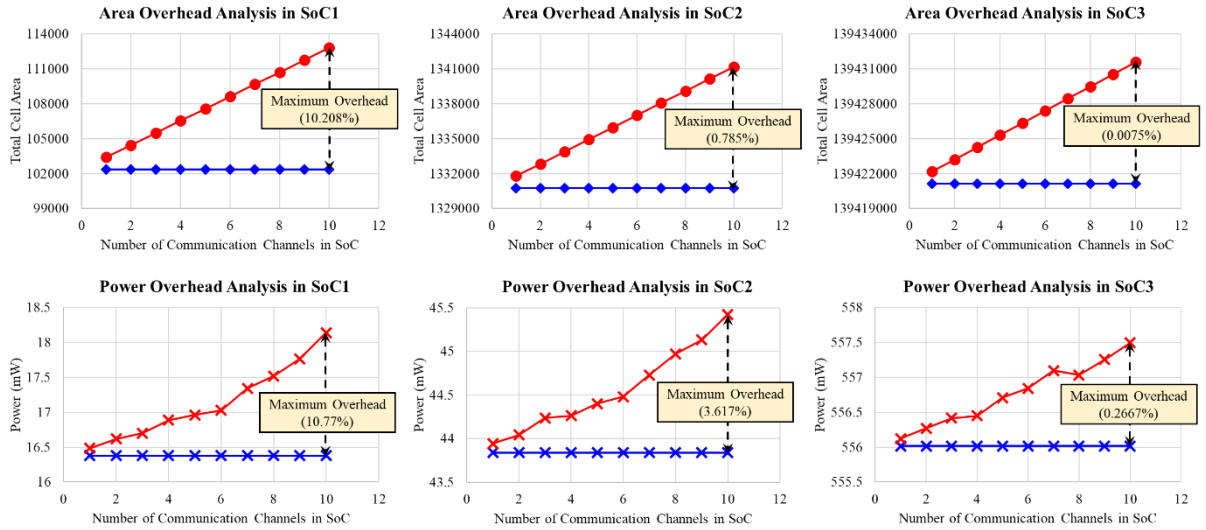


Figure 8: Overhead analysis of implemented SoCs. For this analysis, we synthesized the SoCs using Cadence Genus (Encounter) tool with the TSMC 65nm library.

3 Formal verification of side-channel protected hardware implementation

Side-channel attacks in general and in particular power-analysis attacks are a severe threat to implementations of cryptographic algorithms. Unless countermeasures are implemented, these attacks allow revealing the secret key that is used in a device by observing the power consumption during the execution of a cryptographic algorithm. The attacks are in particular relevant to IoT devices, which are often deployed in a non-protected environment and easily accessible for attackers.

The last decades of research on countermeasures have essentially led to two main approaches towards counteracting power-analysis attacks:

- **Protocol-level countermeasures:** These countermeasures aim at preventing power analysis attacks by redefining the use of cryptographic primitives in cryptographic protocols. The basic idea is to prevent an attacker from observing multiple executions of a cryptographic primitive with the same key. A promising example of a cryptographic protocol to prevent power analysis attacks is ISAP [7]. We study efficient and secure implementation of this countermeasure in the context of WP3.
- **Masking:** The second approach towards counteracting power analysis attacks is to mask the intermediate results of a cryptographic algorithm. During the last two decades, many approaches to efficiently mask implementations of cryptographic algorithms have been proposed. However, also many attacks have been found. This is why currently, there is a significant focus on formally verifying masked implementations. In [8] we have published a novel approach towards verifying masked hardware implementations. In the context of IoT4CPS, we have extended this work and analysed its applicability in an industrial framework. In particular, we have validated and tested the approach within NXP. It has turned out that the approach is indeed a practical approach to test the security of masked implementations with a low masking order. Numerous extensions for improving the performance of the verification have been explored.

In this deliverable, we focus on one particular result, namely the approach of minimizing the number of fresh random masks, while maintaining security in the t-probing model by Ishai, Sahai, and Wagner. In this model, an adversary is allowed to probe up to t intermediate values in an implementation. One is considered secure against such an adversary, if those t wires reveal no secret information.

One important drawback of classical masking schemes is their implementation costs because of their high demand for fresh randomness. Since the creation of large amounts of fresh random bits requires additional time, chip area, energy, et cetera, a lot of research has been done on more randomness-efficient masking. Most of the existing work, however, focuses on the randomness optimization for specific masking gadgets, like masked AND gates, and do not consider the minimization of the overall randomness costs. An interesting result from prior work is the proof by Faust et al. that first-order masking with only one bit of randomness is impossible. They also demonstrated the theoretical possibility of masking with constant randomness cost.

Even more of the masking implementation papers only consider the so-called online randomness costs spent on producing fresh randomness to secure the computation once the initial sharing of the input data, e.g., plaintext, ciphertext, or data and key material, has been performed. There is, to the best of our knowledge, no

prior work that considers the minimization of randomness costs when taking the masking of the input data into account or that tries to minimize the overall randomness costs.

We demonstrate that first-order masking is theoretically possible with only two random one-bit masks. As a first practical contribution, we design a masked AND gate that allows reusing randomness from its inputs safely.

Based on our findings, we introduce a simple rule-based system. These rules can be encoded in SMT2 statements and they are then used to automatically check whether the masking approach is directly applicable to an unprotected implementation or if modifications (mask changes) are required. Upon acceptance, our tool synthesizes a securely masked implementation for a given set of additional constraints like the used mask encoding.

We then show how our approach can be applied to larger implementations and demonstrate its feasibility and its impact on a full AES-128 encryption-only implementation. With our approach, we successfully designed the first formally verified AES S-box design that requires only two random bits for the initial sharing of its inputs and requires no online randomness to achieve first-order security in the probing model. Even when going for a full AES implementation, the randomness requirements do not increase further. However, since existing formal tools are not yet efficient enough to digest a fully unrolled AES implementation, we instead verify each building block of our design using the maskVerif tool of Barthe et al. [9] for a predefined mask encoding of its inputs and outputs. Ensuring the same mask encoding for each input and output allows us to argue about the security when putting the components together in the full AES implementation.

3.1 Masking without Online Randomness

The goal of masking is to make the power consumption (and other side-channels related to the power consumption) independent of security-sensitive information. For this purpose, the security-sensitive information is first combined with uniformly random sampled data in an invertible masking function, such that the representation of the data itself becomes uniformly random distributed. In the case of Boolean masking, the sensitive information s, for instance, is combined with a random mask m by using the Boolean exclusive-or (XOR) operation. The resulting masked value $s_0 = s \oplus m_0$ thus becomes statistically independent of s, i.e., the mutual information between s and s_0 becomes zero. For this reason, any computation on s_0 trivially results in power consumption that is statistically independent of s as long as m_0 is not recombined with s_0.

Adversary Model. The security of masked implementations is often expressed in the so-called t-probing model [10] which assumes that an attacker can make up to t observations in the implementation (place up to t probes on the circuit). It has been verified in the past that this formal model also implies security against a differential side-channel analysis attacker that has access to noisy side-channel leakage traces. We assume in the following a first-order attacker, i.e., an attacker that can place a single probe on the device.

3.2 Computation on Masked Data

To realize computations that are not only secure against side-channel analysis but also correct, the computed masked function needs to take the mask into account but in a way that does not unmask the data. For

example, when calculating the XOR of two sensitive variables as $q = a \oplus b$, where a is shared in the two shares a_0 and a_1 and b is shared as b_0 and b_1, the correct and securely masked realization is trivial:

$q_0 = a_0 \oplus b_0$, $q_1 = a_1 \oplus b_1$

With Equal Masks. The situation changes when assuming that both masked variables use the same mask $m_0 = m_1$, which trivially reveals a and b in the equation of q_0.
$q_0 = a_0 \oplus b_0 = (a \oplus m_0) \oplus (b \oplus m_0) = a \oplus b$

Most state-of-the-art masking works assume that shares are produced using independent random masks which helps to avoid such situations. Our first and admittedly rather trivial observation is that the amount of accumulated randomness is unnecessarily high. One can realize the same function in a secure and correct shared way by simply alternating two random masks m0 and m1 in such a way that at no time an intermediate result is formed that depends on the secret value without a mask. One possible realization is to use m0 to mask a and use m1 for the remaining variables:

$q_0 = (a \oplus m_0) \oplus (b \oplus m_{1)} \oplus (c \oplus m_1) \oplus \ldots (z \oplus m_1) = a \oplus b \oplus c \oplus \cdots \oplus z \oplus m_t$
$q_1 = m_0 \oplus m_1 \oplus m_1 \oplus \cdots \oplus m_1 = m_t$

where $m_t = m_0$ if the number of inputs is odd (and thus the number of m_1 masks is even) and else $m_t = m_0 \oplus m_1$.

3.3 Application to Nonlinear Gates

There exists a vast variety of first-order masked AND gates in the literature which form the simplest class of nonlinear functions and are used to construct more complex functions. These realizations of masked AND gates usually vary regarding online randomness requirements and the number of used input and output shares. The underlying functionality is of course always the same and, in the case of a realization with two shares, it requires the secure evaluation of four multiplication terms (where $\wedge$ represents a single AND operation):
$q = a \wedge b = (a_0 \oplus a_1)(b_0 \oplus b_1) = a_0 \wedge b_0 \oplus a_0 \wedge b_1 \oplus a_1 \wedge b_0 \oplus a_1 \wedge b_1$
Any direct combination of either two multiplications terms (e.g., $a_0\, b_0 \oplus a_0\, b_1$) is insecure because it leads to a function that statistically depends on the secret a or b. Most of the existing masked AND gadgets thus use fresh random masks to realize the secure evaluation.
Without Fresh Randomness. There also exist realizations of a masked AND gate that do not require any fresh randomness. As an example, we consider the following equations from Biryukov et al. [11] where $\vee$ is the OR operation:
$q_0 = a_0 \wedge b_0 \oplus (a_0 \vee \neg b_1)$
$q_1 = a_1 \wedge b_0 \oplus (a_1 \vee \neg b_1)$
A closer look at the properties of this realization from Biryukov et al. reveals that, while the masking itself is secure, a further (linear) combination with shares or combinations of shares from a and b (barring $a_0 \oplus a_1$) can

make the sharing insecure again. Chaining of masked AND operations by carefully selecting (or changing) between two different masks is not possible with this masked AND gate.

3.4 Construction of a New Masked AND

We first transform the secure equations of Biryukov et al. such that we can directly observe what happens to the multiplication terms.

$q_0 = a_0 \wedge b_0 \oplus (a_0 \vee \neg b_1) = a_0 \wedge b_0 \oplus \neg(\neg a_0 \wedge b_1) = a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1) \oplus 1$

$q_1 = a_1 \wedge b_0 \oplus (a_1 \vee \neg b_1) = a_1 \wedge b_0 \oplus \neg(\neg a_1 \wedge b_1) = a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1) \oplus 1$

It can be verified that the terms $a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1)$ from q_0 and $a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1)$ from q_1 , considered separately, are securely masked by b_1 (= m_1 , in the masking representation).

New construction. The design idea to ensure that the resulting sharing behaves similarly to the masked XOR gate is to securely combine all multiplication terms in a single share of q, together with a single mask. However, adding q_0 and q_1 directly together is insecure because this results in a ∧ b without any mask. We therefore first add a_1 (= m_0) to the second term (q_1) and then, both terms can be added without leaking information. The result (our new q_0) is only masked with a single mask m_0. To achieve correctness the second share (the new q_1) is set to m_0 (or equivalently a_1). This then results in the following masked AND gate:

$q_0 = (a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1)) \oplus ((a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1)) \oplus a_1) = (a \wedge b) \oplus m0$

$q_1 = a_1 = m0$

Further optimization. We find that under given circumstances (possible mask configurations associated with the input shares), another optimization is possible:

$q_0 = (a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1)) \oplus (a_1 \wedge b_0 \oplus ([m_0 \vee m_1])$

Security. The security of the masked AND gate can be easily verified by hand. In addition to the manual inspection of the masked AND gate, we also performed a formal verification by using the tools by Bloem et al. [8] and Barthe et al. [9] which gave us the same results. Furthermore, we did the same verification for the composition of the AND gate with an XOR (q ⊕ b) and with another AND (q ∧ b).

By combining the findings for the XOR and the AND gates we can mask arbitrary implementations, and as we will show in the next section, we can also derive simple rules to synthesize securely masked implementations from unprotected ones.

3.5 Synthesis of First-Order Secure Implementations

Manually tracking the masks as they propagate through the implementations quickly becomes a very complex task as the implementation size increases. We thus decided to develop an automated approach to create a masked implementation when possible, or to indicate which signals need to be changed otherwise. As a first step, the tool reads the description of a Boolean program in static single assignment (SSA) form in Verilog syntax such that each instruction is either a one-bit signal assignment or a two-bit XOR, XNOR, or AND gate.

The Boolean circuit is then represented as an SMT problem which is fed to the Z3 theorem prover. Z3 searches for a possible solution for the mask encoding of the input signals so that for each gate the inputs have different masks. Furthermore, it allows ensuring a desired mask encoding for the input and output signals.

Checking of the model and creating the masked implementation. When the Z3 theorem solver finds a secure model that fulfils our constraints, it constructs the mask assignments for a masked implementation. The translation of the unprotected scheme to a secure masked implementation is then rather straightforward. At first, we duplicate all input and output ports of the module and additionally add the two masks m0 and m1 as input signals. For each instruction of the SSA input file we replace the original code by its masked variant according to the masked gates introduced in Section 2. As a further optimization, the second share of each instruction is (optionally) replaced by the resulting mask of the output signal which helps to save unnecessary instructions that would result in one of the three mask encodings anyway.

3.6 Masking the AES

To demonstrate the practicality of our approach, we target the AES-128 (encryption-only) as an example. Since none of the existing formal verification tools are yet powerful enough to verify a full AES encryption, we decide to use a modular implementation and verification approach. To justify the security of the overall design when bringing the modules together, we restrict the mask encoding for each input and output byte of every function to be equal.

Our software implementation is partially based on earlier work by Schwabe and Stoffelen [12]. In their paper they describe various optimized assembly implementations targeting the 32-bit ARM Cortex-M3 and Cortex-M4 microcontrollers. One implementation is masked using 2 Boolean shares.

The most complicated part of the AES is its SubBytes layer which can be implemented as 16 instances of S-box modules. A suitable design for our bit-wise approach, is the design of Boyar and Peralta [13] which is already constructed in SSA form.

Result of the Tool. After running our synthesis tool on this S-box design without any further optimizations, the resulting masked design consists of 96 AND gates, 228 XOR gates, and 4 NOT gates (because XNORs are decomposed to one XOR followed by a not gate in Yosys' ILANG). The 96 AND gates result from the fact that the masked AND triples the number of AND gates compared to the unmasked design. Also, each masked AND gate introduces 4 XOR gates which in total results in 128 additional XOR gates. The masking of the XOR and XNOR gates, on the other hand, do not introduces additional circuitry since the second output share can simply be assigned to the third mask (i.e. unused by the inputs). Some additional XOR gates are required because at some points we need to change the masking of a signal by introducing additional XOR instructions to receive a satisfiable Z3 model and thus a securely synthesizable implementation, and to ensure that the input and output mask encoding is equal.

After running an optimization pass in Yosys, which maps gates implementing the same function to a single gate and thus eliminates duplications, the number of gates could be reduced to 86 AND gates, 1 OR gate, 225 XOR gates and 4 NOT gates. We rerun the verification after this optimization to ensure that the implementation remains secure. The NOT gates can be moved to the key schedule such that they are not executed for every

encryption/decryption call with the same key. The total overhead for the masking of the S-box is thus about a factor 2.79 regarding arithmetic instructions.

3.7 Results

For the entire AES encryption, we measure on average 3,387.6 cycles per block (or 211.7 cycles per byte) under the exact same test conditions as in [12]. This is a speed improvement of roughly 54%. Moreover, the stack requirements are lowered from 1588 bytes to only 188 bytes, a decrease of over 88%.

Schwabe and Stoffelen also provided an unmasked bitsliced AES-128-CTR implementation as an intermediate step. This took only 1617.6 cycles per block (or 101.1 cycles per byte) on the Cortex-M4. The overhead cost of adding first-order masking is therefore still almost a factor 2.1.

	Platform	Speed [cycles]	Overhead factor	ROM [bytes]	RAM [bytes]	Random [bits]
This Work	Cortex-M4	3387	2.1	25.2k	188	2
[Schwabe & Stoffelen 2016]	Cortex-M4	7422	4.6	39.9k	2.0k	10.5k

3.8 Formal Verification in the t-Probing Model

For the verification of the side-channel security of our approach, we used the formal verification tool maskVerif of Barthe et al. [9] on the synthesized modules. Since maskVerif is originally designed to verify sharing-based implementations, the outcome of our synthesis tool creates a verification wrapper that is later on modified to represent the correct masking for the input signals of the actual masked implementation. The verification wrapper thus takes two shares per input of the masked module and creates the correct masking by first adding the mask as defined by the mask encoding and subsequently the second share of the input.

For the input in the maskVerif tool, the implementation is read by the Yosys open synthesis tool. The circuit is then mapped to Yosys' internal gate representation (ILANG) and subsequently flattened such that a single module is created that contains all gates. The resulting circuit is then returned in ILANG format for which input, output and mask signals are annotated before it is fed into maskVerif. The implementations are validated for the probing model of Ishai et al. without glitches.

All of the modules on which our entire AES-128 encryption depends, are probing secure as intended. ShiftRows is only rewiring (readdressing) in hardware and just a bit permutation in software, which does not influence the probing security. With the input and output constraints for our synthesis tool, we also ensure that the mask encoding for each byte is the same, and we can thus safely compose these modules without creating flaws in the probing model for first-orders. However, we note that this composition argument is only true for first-order implementations for which a probing attacker is restricted to a single probe. This means that multivariate probes are of no concern and thus probes occur only in a single submodule. The reuse of randomness has no influence on the output distributions of cascaded gates, as long as the mask encoding is done with precision. Our synthesis tool creates implementations which, by construction, ensure that the mask encoding is fixed at the inputs and outputs of submodules. Our submodules have been formally verified for these encodings. Therefore, combined with the fact that probes can only be placed on a single submodule, this ensures that the entire AES implementation is first-order secure.

We have proven the security of our scheme using formal verification tools and demonstrated that randomness can almost completely be eliminated for first-order security within the t-probing model.

4 Dynamically Exchangeable Runtime Checkers in HW

Following section is about runtime verification of security properties in reprogrammable hardware. It shows how the **Dynamic Partial Reconfiguration** feature of FPGAs is leveraged to enable a set of hardware-based security checks within a resource-constrained device to enable safe and secure IoT-based applications.

4.1 Hardware Apps

For almost any modern mobile device like Smartphones or Tablets, Software Apps are available. These small programs can individually enhance the default functionality of the device on demand. Apps can be purchased and installed via an App-Store that is maintained by a service provider. These well-known and widely accepted concepts of the mobile communication domain are increasingly introduced to other sectors such as Automotive, Industrial Control and Automation, where more stringent safety and security requirements need to be considered. Especially in systems where battery runtime is of less importance, reconfigurable hardware devices are integrated in many cases. FPGAs (Field Programmable Gate Arrays) offer great flexibility with respect to application openness, since they can be reconfigured on-the-fly during runtime. Latest FPGA products offer the possibility to even exchange only portions of the device while the rest of the system continues its operation without interference. This Dynamic Partial Reconfiguration feature is very powerful and is available in common state-of-the-art SRAM-based FPGAs, e.g. Intel STRATIX or XILINX Zync Ultrascale+.

The concept of Hardware Apps extends standard Software Apps with hardware accelerators instantiated within reconfigurable FPGA devices. Consequently, this new kind of App basically consists of two parts, both available via App-Store:

- Software App
- Hardware configuration (FPGA bit stream suitable for partial reconfiguration)

Hardware Apps are applicable wherever spare FPGA resources are available in order to accelerate SW-Apps or simply to offload the CPU. Possible fields for the application of HW Apps are:

- In-field data analytics (e.g. dynamic update of algorithms extracted from cloud to devices)
- Real-time control functions (e.g. different algorithms depending on dynamic/changing environment of products)
- Crypto accelerators (e.g. supporting multiple algorithms in sequence)
- Communication (e.g. update of selected interface types to adapt for changing communication partners)

4.2 Dynamic Partial Reconfiguration (DPR)

SRAM-based FPGAs can be partially reconfigured using a specifically generated partial bit stream. This bit stream includes instructions for the DPR controller and data for the configuration memory. In other words, DPR modifies all parts of configuration memory that configure a selected PR-region. Following **Fehler! Verweisquelle konnte nicht gefunden werden.** shows the physical structure of an INTEL SoC-FPGA with one PR-region.

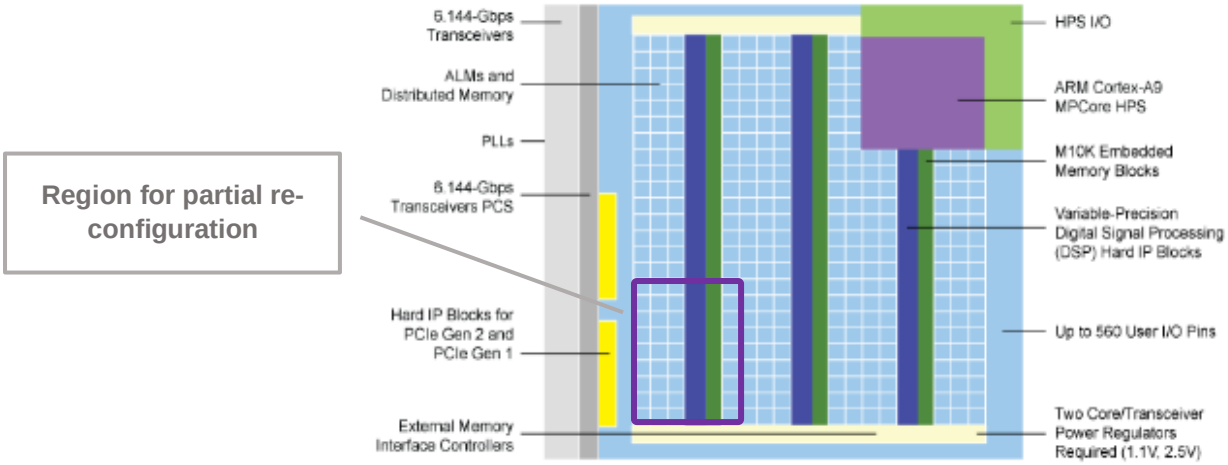


Figure 9: One region for PR in an INTEL SoC-FPGA

Size and location of a PR-region in the floorplan are defined by the user and it is possible to create a design with several PR-regions where each region can be re-configured separately. The example below shows a design with 2 modules B and D which can be reconfigured using DPR. For example, the functions of module B are B1 and B2 and the functions of module D are D1 and D2. DPR allows exchanging the function B1 with B2 or vice versa for module B while the rest of the device continues running. Same for module D, where function D1 can be replaced with D2. Important to note is that it is not possible to place functions of module B in the slot of module D – these are strictly reserved for one region. The partial bit streams are always generated for a specific slot and cannot be loaded in other places, even in case the size is the same.

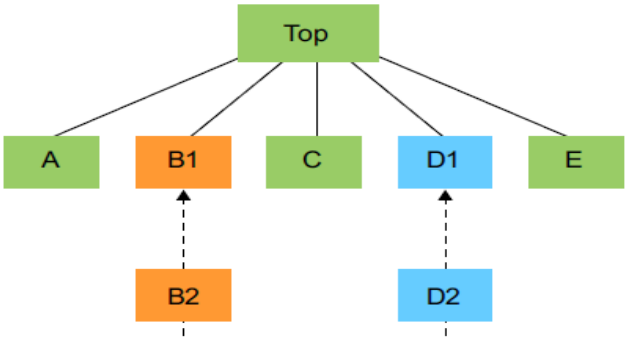


Figure 10: Design with two DPR-regions B and D (hierarchical structure)

One function of a PR-region can be considered as a HW-App. There is no limit on the number of HW-Apps for one PR-region, each HW-App has its bit stream generated for the configuration of a dedicated PR-region.

Partial Reconfiguration enables:

- Design permutations that do not operate simultaneously
- Time-sliced sharing of the same FPGA resources
- Use-cases like data-analytics in HW, real-time control algorithms, applications for a selected set of external interfaces
- Generic platform-like FPGA design instead of many specific design variants
- Smaller devices instead of large devices for complex all-in-one FPGA designs

4.3 Random Numbers in FPGA

True Random Number Generators (TRNG) are essential security building blocks that are used to generate random numbers with high entropy. In cryptographic applications they are applied for the generation of secret or public keys, initialization vectors and seeds for various cryptographic primitives and pseudo-random number generators. Low entropy of the generated numbers has negative influence on the level of security of any security measure built upon. Measuring entropy, resp. the 'quality' of random numbers is a complex task where a lot of research was already done, and several evaluation suites are available. Often, FPGAs are used to host cryptographic functions when random numbers are needed, and many different mechanisms and physical effects can be exploited to realize TRNGs in FPGA reconfigurable logic. Especially for FPGA implementations it is very hard to estimate respectively measure the entropy for a TRNG because it may depend on the actual final placement within the available FPGA fabric. Even device variations can have significant impact on the quality of extracted random numbers.

For example, one popular and straight forward method for implementing TRNGs within FPGA is based on ring oscillators. Especially for this sort of TRNGs the physical placement of the large number of ring oscillators affects the quality of results.

4.4 HW Checker Apps

Dynamic Partial Reconfiguration can be used to build an FPGA Design where individual checker modules are realized as HW-Apps that operate like a firewall. These checkers are located next to the interfaces of critical modules (e.g. encryption engine or random number generator) and observe data transfers. In case malicious activity is detected an alarm is triggered via an interrupt line. **Fehler! Verweisquelle konnte nicht gefunden werden.** depicts a simple demo design that was built as proof of concept. Here the entropy of a sequence of random numbers included within an encrypted data stream is checked with different algorithms.

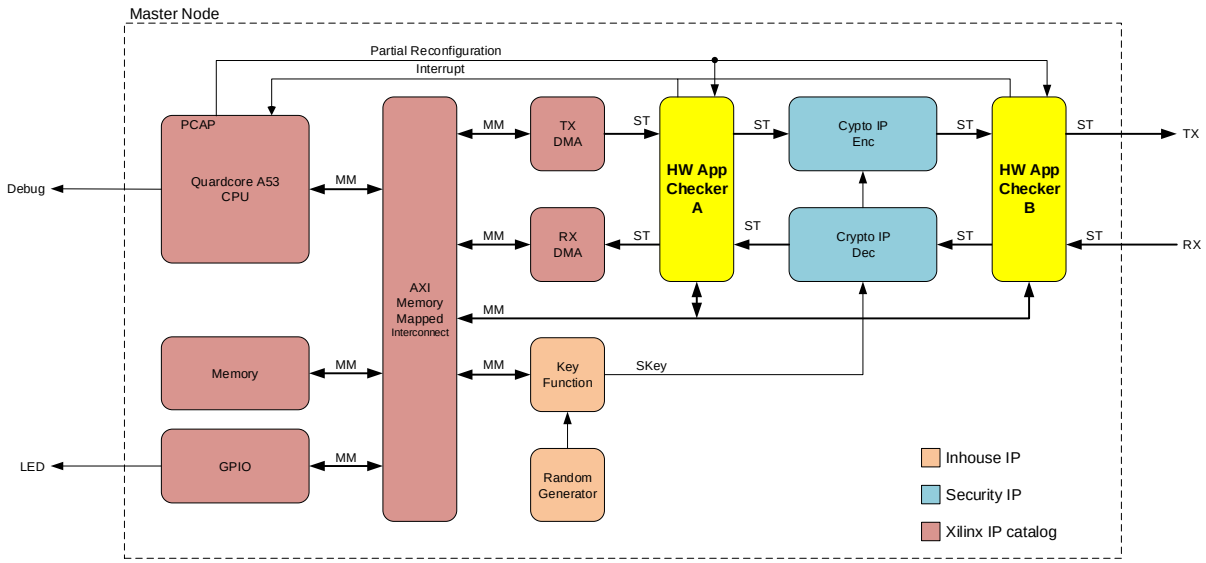


Figure 11: Example Design for HW-App checkers

4.5 Implemented Functions

The HW_App Checkers (**Fehler! Verweisquelle konnte nicht gefunden werden.**) provide functions to assess the randomness of a bit sequence generated by a random number generator (RNG). This bit sequence is a part of encrypted data packets. The bit sequence to be tested (RNG_data) will be extracted from input data under consideration of the input data format and protocol. The functionality for the extraction of RNG_data is common for all Checkers and is implemented in a separate sub-module. The output are RNG_data with accompanying data_valid signal. The width of RNG_data depends on the input data format and is variable.

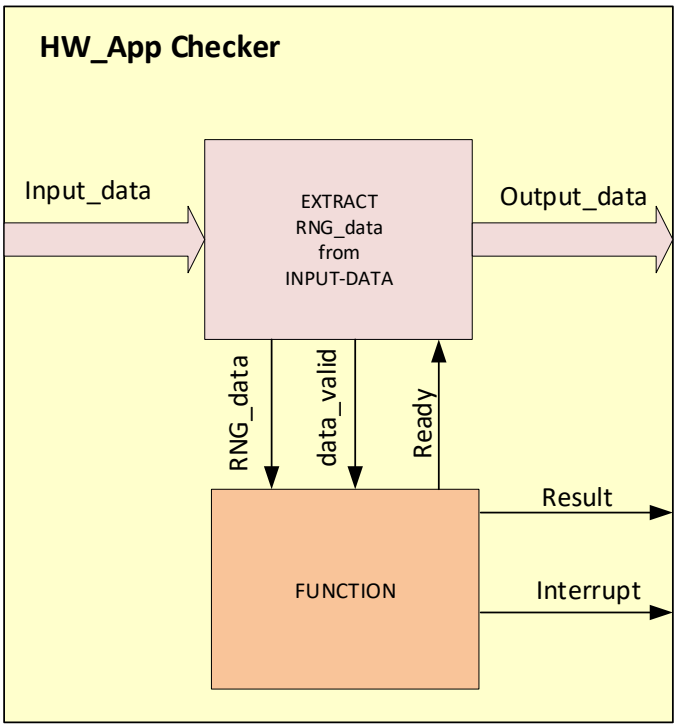


Figure 12: common structure of RN checkers

The block “FUNCTION” represents the function for checking the randomness of RNG_data, using generic parameters that can be configured:

- Width of input RNG_data
- Total length of input bit sequence for each evaluation

Outputs of the FUNCTION-Block are

- the “Result” signal indicating passed or failed tests
- “Interrupt” that will be active in case the test failed
- Handshake signal “Ready” = 1 means the FUNCTION-Block is ready to receive RNG_data

The implemented functions are selected from a NIST test suite for statistical evaluation of randomness. The NIST test suite includes 15 functions defined in Bassham et al. [44]:

The NIST test functions are mainly suitable for SW. For their implementation in HW, particularly for FPGA, some simplifications are necessary. The selection of simplified NIST-functions, suitable for the implementation in FPGA, originates from Veljkovic et al. [45]

Currently, there are three selected functions available:

1. Frequency Test (Monobit)
2. Cumulative Sums Test
3. The Longest Run of Ones in a Block

For all implemented functions the value α = 0.01 has been chosen for the threshold value for P_value. Parameters for each test are chosen with the consideration of the input size recommendations given by NIST.

4.5.1 Frequency Test (Monobit)

The purpose of this test is to determine whether the number of ones and zeros in a sequence are approximately the same as expected for a truly random sequence. The test assesses the closeness of the fraction of ones to ½, that is, the number of ones and zeroes in a sequence should be about the same. In each evaluation a sequence length (n) of 20000 bits is assessed. According to the simplification of this test, the number of ones (ε) for sequence of 20000 bits should be in the range of [9818, 10182].

Implementation of the function in HW:

1. Counter for bit sequence length: For monitoring the input sequence length there is one counter counting a total length of input data (e.g. up to 20000 bits). The counter marks start and end of one evaluation cycle.
2. Counter of ones: This counter counts ones in RNG_data within one evaluation cycle (ε).

Interpretation of results:

If ε is in range [9818, 10182], the test passed and the outputs are:

result_o = '0' and int_o = '0';

If the test failed, the outputs are:

result_o = '1' and int_o = '1' (the interrupt will be set active).

4.5.2 Cumulative Sums Test

The focus of this test is the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted (-1, +1) digits in the sequence. The purpose of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behaviour of that cumulative sum for random sequences.

The function calculates the total sum of all bits in the sequence and stores maximal excursions from zero.

Applying simplification of the function, the calculation of the maximal excursion from zero of the random walk (for the sequence length of n = 20000 bits) should result in +- 379.

Implementation of the function in HW:

Following inequality is implemented in HW

−397 <= Sk <= +397

Sk is the maximal or minimal partial sum calculated over all bits in the sequence.

The function is implemented using a counter for the total length of the bit sequence, an adder with comparator to monitor min and max values of Sk and a register to store the min and max values.

At the end of one evaluation, the largest of the absolute values of Skmin and Skmax is used for comparison in the inequality −397 <= Sk <= +397.

The function provides only "forward" mode, the "backward" mode is not available.

Interpretation of results:

If Sk is in range [-379, +379], the test passed and the outputs are:

result_o = '0' and int_o = '0';

If the test failed, the outputs are:

result_o = '1' and int_o = '1' (the interrupt will be set active).

4.5.3 The Longest Run of Ones in a Block

The focus of the test is the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. Note that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Therefore, only a test for ones is necessary.

Function parameters:

- n the length of a bit sequence (RNG_data)
- M the length of each block
- N the number of blocks in accordance with the value of M (n = N*M)
- K degrees of freedom

…are selected from the table:

n	Block length-M	Num. of Blocks-N	Deg. of freedom-K
128	8	16	3
6400	128	50	5
750000	10^4	75	6

- n = 128
- M = 8
- N = 16
- K = 3

The function detects the longest run of ones (L) in a block and records this value. The L values are assigned to categories (vi) and the number of blocks assigned to each category are counted.

The number of available categories (vi) depends on the selected parameter K respectively M according to the following table:

v_i	M = 8	M = 128	$M = 10^4$
v_0	≤ 1	≤ 4	≤ 10
v_1	2	5	11
v_2	3	6	12
v_3	≥ 4	7	13
v_4		8	14
v_5		≥ 9	15
v_6			≥ 16

Table 1: The Longest Run of Ones in a Block – Categories

Consequently, for M = 8, there are 4 categories v0 … v3, i.e. 4 counters which count the number of assignments to the category depending on value L.

The simplified function with selected parameters computes the inequation for one evaluation including 16 blocks:

$$\sum_{i=0}^{K=3} \nu_i^2 \left(\frac{1}{\pi_i}\right) \leqslant 437.52$$

where values for (πi) vary depending on K. According to the NIST-document, the values for K = 3 are as follows:

K=3, M=8

classes	probabilities
{ν≤1}	$\pi_0 = 0.2148$
{ν=2}	$\pi_1 = 0.3672$
{ν=3}	$\pi_2 = 0.2305$
{ν≥4}	$\pi_3 = 0.1875$

Table 2: Probability for classes (K=3, M=8)

Implementation of the function in HW:

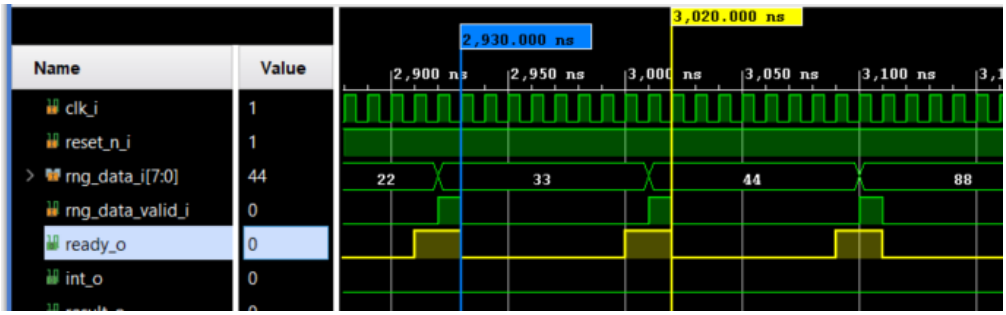
- Logic to detect the longest run of ones in one block of 8 bits (L)
- Four counters v0 … v3 counting assignments of the L value e.g. if L = 2, the counter v1 will be incremented
- Multipliers and adders for calculation of the inequation

Parameter settings:

- g_data_width: 8, 16 or 32 bit
- g_sequence_length = 128

The input RNG_data can be received every 9 clock periods for all 3 possible values of g_data_width. In case g_data_width is 16 or 32 bits, they will be processed in parallel with 2 or 4 data-blocks.

If signal ready_o = '1', the function block is ready to receive new data earliest in the next clock cycle. The sender sets the valid signal active for one clock period.



Interpretation of results:

The criterium for passing the test is given by the mentioned inequation. In case the test failed, the interrupt will be set active and the output result_o = '1'.

5 Threat Modelling

Usually security testing is limited by resources (i.e. time, complexity of the system under test, etc.). To overcome these limitations, an actual IoT/IIoT system can be abstracted as a model. The model is an abstraction of the real-world IoT/IIoT system under test, which should optimally re-enact the expected behaviour of the system. In this chapter, the threat modelling approach within IoT4CPS is described.

5.1 Methodology

Threat modelling is an approach with the goal to identify threats and vulnerabilities within IT system architectures [14] and was also introduced by Microsoft as part of their Security Development Lifecycle (SDL) concept. The Microsoft Threat Modelling tool, which was used within IoT4CPS, creates threats that are divided into six categories, which are defined in the, at Microsoft developed, STRIDE model [15]:

- Spoofing identity. A user or service illegally accesses and uses other authentication information to gain illegitimate access to a system or data.
- Tampering with data. Data tampering occurs when data is malicious modified. This includes data at rest, data in use as well as data in transit.
- Repudiation. This means that an entity may plausibly deny an action that it has taken. Countering these threats usually requires a combination of authentication, authorization and logging, ideally in a cryptographically secured way.
- Information disclosure. Refers to any information exposed to unauthorized users.
- Denial of service (DoS). DoS attacks deny services availability to valid users.
- Elevation of privilege. These threats occur when unprivileged users gain privileged access and, thus, are able to compromise an entire system.

5.2 Modelling Setup

Within IoT4CPS this tool was selected under **Task 4.1 - Strategic Security Assurance** to model the main industrial use case the Device.CONNECT™. Figure 13 gives an overview of the Device connect set-up and outlines the data flow between the backend system (Device Connect Framework), the middleware (MQTT Broker) and the client system (Smart Hub) at the customer's side.

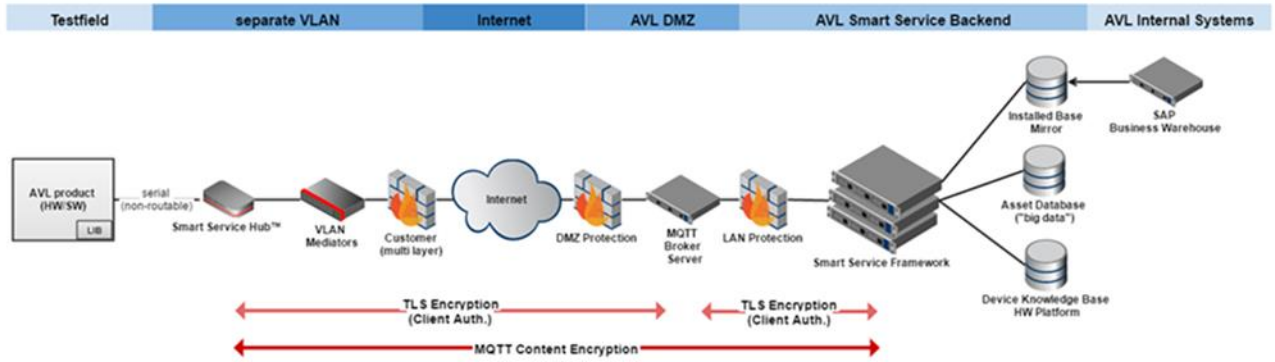


Figure 13: Device Connect

In addition to conceptual model outlined in Figure 13 a basic threat model was provided by AVL, which is shown in Figure 14.

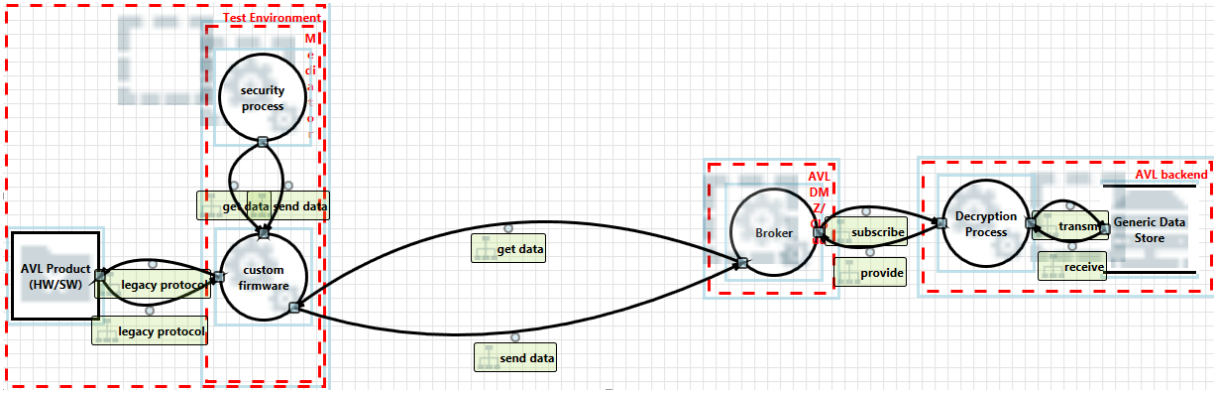


Figure 14: Device Connect - Basic Threat Model

5.3 Model Extension

The MS Threat modelling tool offers the possibility to extent the modelling template with new components and communication lines. For IoT4CPS, the identified processes of the use case were added to the template and in addition to that, the 85 threats identified during the requirements analysis step in WP2 were aligned to the components of the template. A sample of how these extension were implemented can be seen in Table 3. Here, three threats in different categories and their implementation queries can be seen. Having a look at the queries, some for IoT4CPS created components like "Key Storage", "Security Controller", "AVL Product" and "Custom Firmware" can be found. Finally, this implementation yields to an IoT4CPS model of potential threats consisting of the standard model provided by the tool as well as device-specific threats and additional modelling of the protocols in use.

Category	Title	Microsoft TM Query

System design exploits	Encryption-deprecated cryptographic algorithms	(source is [Key Storage] and target is [Security Controller]) or (source is [Security Controller] and target is [Key Storage])
Compromise of external connectivity	External wireless interfaces-Interference	(source is [AVL Product] or source is [Custom firmware]) and flow.[Physical Network] is 'Wi-Fi'
Human factor and social engineering	Misconfiguration/Erroneous use or administration	(source is [Browser] or source is [Human User]) and (target is [Generic Process])

Table 3: MS Threat Model Extension

5.4 Result

In Figure 15 the advanced threat model covering all identified processes, data flows and data storage elements of the Device Connect use case is shown. This model yield to a final list of 358 threats. All of these threats were reviewed in cooperation with AVL in order to have full insight if the respective threats are applicable or not. Thus, out of the 358 generated threats, 246 were duplicates, which resulted in the same mitigation even though the threats might occur on different places of the system architecture, 74 threats were not applicable or has been already addressed by AVL, resulting in 38 new threats that will be addressed.

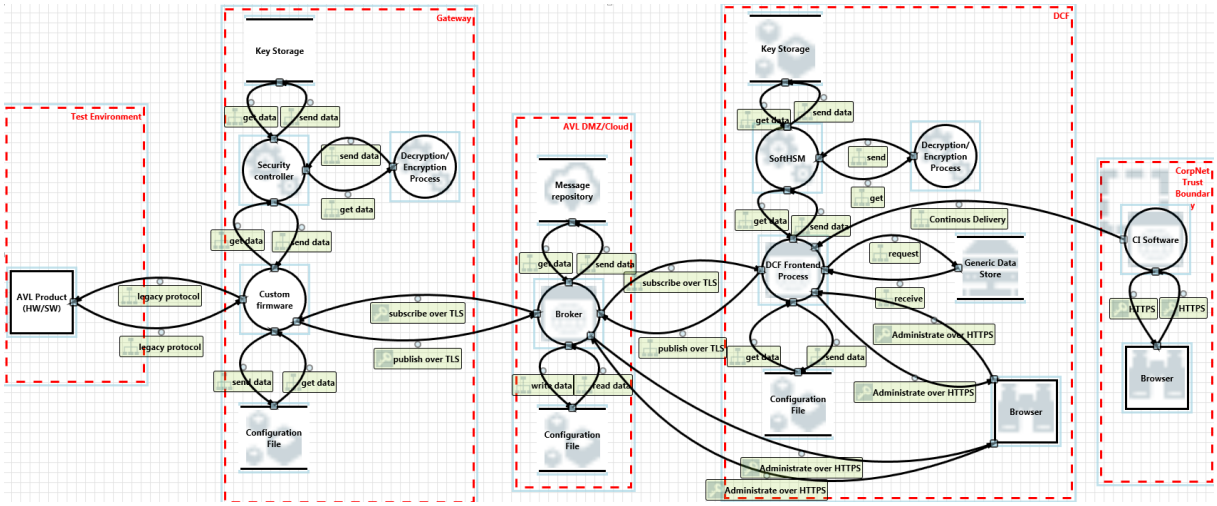


Figure 15: Advanced IoT4CPS threat model

As an example of generated threats within the threat model, a closer look to the model describing the DFC Frontend Process and the interaction with the aligned configuration file (Figure 16) can be taken. The interaction shows the data flow between these two components leading to two different potential threats (1) *Weak Access Control for a Resource* and (2) *Spoofing of Source Data Store Configuration File.*

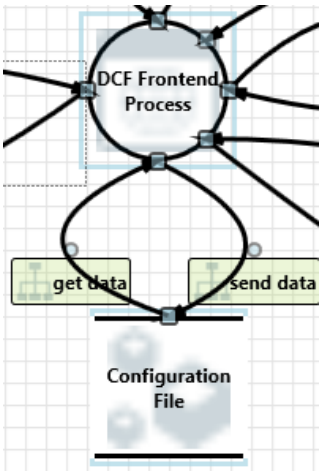


Figure 16: DFC Frontend Process Configuration model

Table 4 list these threats with the additional information provided by the threat model. These threats then have to be evaluated with regards to their applicableness. Thus, a review with security and software experts was taken for each of the generated threats. The results are described within section 6.

1. Weak Access Control for a Resource	Priority: High
Category	Information Disclosure
Description:	Improper data protection of Configuration File can allow an attacker to read information not intended for disclosure. Review authorization settings.
Justification	<no mitigation provided>
Short Description	Information disclosure happens when the information can be read by an unauthorized party.
2. Spoofing of Source Data Store Configuration File	Priority: High
Category	Spoofing
Description:	Configuration File may be spoofed by an attacker and this may lead to incorrect data delivered to DCF Frontend Process. Consider using a standard authentication mechanism to identify the source data store.
Justification	<no mitigation provided>
Short Description	Spoofing is when a process or entity is something other than its claimed identity. Examples include substituting a process, a file, website or a network address.

Table 4: Potential Threats of DFC Frontend Process Configuration

6 Penetration Test Catalogue

A penetration test catalogue is the idea to create test cases describing their prerequisites and test steps in order to help cyber security experts, system administrators, software developer and software testers not only to test their products in regards to cyber security but also to increase the awareness and communication during the development and testing processes. In addition, these test cases should be able to be re-used on similar test cases. In this section, the penetration test catalogue approach of IoT4CPS is described.

6.1 Methodology

Taking the in Section 5 described threat modelling approach to the next step, a penetration test catalogue has been created. This catalogue addresses each identified threat with a respective mitigation in order to help software developers, software testers as well as system architects to improve the modelled architecture in regards to cyber security during development. In addition, cyber security experts can use the penetration test catalogue as a help when performing a penetration test, which is the process of testing computer systems as well as human resources (social engineering) to identify security threats and possible vulnerabilities. These tests can be performed from the inside and from the outside of the systems under test to ensure, that all possible options of an attacker are covered. The aim of each penetration test is to specify guidelines and recommendations, which address the identified issues.

To perform a more effective penetration test and to keep the costs down, the IoT4CPS project pursues the strategy to perform a grey box penetration test obtained by the threats identified via the threat modelling process. Therefore, the identified threats are used to specify a set of tests according to the International Software Testing Qualifications Board (ISTQB). The ISTQB describes each test with an ID, a description, the needed prerequisites, the test steps which are used to perform the test, needed test data as well as the status and the results of the test [16]. A template of an ISTQB test case can be seen in Table 5.

Name	**Description**
Test case ID	Unique identification of the test case
Test case description	The objective of this test case
Prerequisites	Prerequisites to perform the test (HW, SW, etc…)
Test steps	Exact specification procedure to follow
Test data	Input data for the specific test case
Expected Results	Expected result and post conditions
Actual Results	Obtained results and post conditions
Status	Pass or fail
Created By	Name of the person who specified the test case
Date of creation	Date of test case creation
Executed By	Name of the person who executed the test case
Date of execution	Date of test case execution

Table 5: ISTQB Template

To create the penetration test catalogue according to the ISTQB specification, each from the threat model generated threat had been reviewed with regards of their applicableness. To do so, JR in cooperation with experts of the modelled systems from AVL reviewed each threat.

6.2 Results

As a result, Table 6 shows the applicable test cases created from the threat model. As the table is missing the non-applicable test cases, the IDs of the test cases are not continuously. However, the complete list of the penetration test catalogue, including the non-applicable test cases can be found in the Annex.

Name	**Description**
Test Case ID	6
Test case description	Data Flow is not interrupted by network issues
Prerequisites	traffic generator (e.g. ostinato)
Test steps	1. Interrupt the data flow by removing the connection physically 1a. Remove router/switch from the network 1b. Remove network cable 2. Use traffic generator to simulate high capacity
Test data	-
Expected Results	1. Dataflow is not interrupted due to missing redundant connectivity 2. High capacity does not result in packet loss
Test Case ID	7
Test case description	Excessive Resource Consumption
Prerequisites	Test data is needed and must be defined by the development team
Test steps	1. Test the service with an unusual amount of data and requests
Test data	Unusual amount of data that can be consumed
Expected Results	The consumption of the data does not cause a denial of service or other unexpected behaviour
Test Case ID	8
Test case description	Elevation of Privilege Using Remote Code Execution (Broker)
Prerequisites	1. Any fuzzer (like sfuzz or AFL) 2. define fuzzing range based on the application
Test steps	1. Use fuzzer to test input validation
Test data	
Expected Results	Device is not affected in an unexpected way by the input: - invalid data is rejected - valid data is processed - no access to functions the user is not privileged for
Test Case ID	9
Test case description	Network design - internet ports left open

Prerequisites	1. Internet access 2. NMAP or other port scanner 3. Necessary ports must be defined by AVL
Test steps	1. nmap -p- -Pn -sU <ip>
Test data	-
Expected Results	Any unnecessary port is closed. Used ports: • Smart Hub: SSH, NTP, DHCP • Broker: 22, 8883 • Backend: -
Test Case ID	36
Test case description	Input Validation for Custom firmware
Prerequisites	1. Any fuzzer (like sfuzz or AFL) 2. define fuzzing range based on the application
Test steps	1. use fuzzer to test input validation
Test data	-
Expected Results	Invalid data gets rejected from service
Test Case ID	47
Test case description	Potential Lack of Input Validation for DFC Frontend Process
Prerequisites	1. Identify input possibilities (API endpoints, User Input, etc.) 2. Identify valid and invalid input data based on the endpoint (XSS, SQLi, XML injections, etc.) 3. Use certain frameworks to automate test steps: (Xenotix XSS exploit framework, SQLMap, XEE frameworks: https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XXE%20Injection)
Test steps	1. Send invalid data to identified endpoints
Test data	
Expected Results	Invalid data is rejected
Test Case ID	48
Test case description	Cross Site Scripting
Prerequisites	1. OWASP Xenotix XSS Exploit Framework 2. Identify possible XSS input fields
Test steps	Use XSS Framework to automatically search for vulnerabilities
Test data	
Expected Results	No XSS vulnerabilities are found
Test Case ID	49
Test case description	Process Crash or Stop for Broker

Prerequisites	-
Test steps	Simulate a process crash for the device by e.g. killing the process during production time
Test data	-
Expected Results	1. The process recovers in a short period within a self-healing process 2. Log files are available to investigate the issue, which caused the shutdown 3. Monitoring software notifies the responsible system administrators
Test Case ID	55
Test case description	Data Flow Sniffing
Prerequisites	1. Network access 2. network sniffer (Wireshark/tcpdump)
Test steps	1. Use network sniffer to sniff the data flow (i.e. Wireshark, tcpdump) 2. UI: Wireshark 3. tcpdump: tcpdump -i <interface> host <ip>
Test data	
Expected Results	Sniffed dataflow is encrypted
Test Case ID	58
Test case description	Process Crash or Stop for custom firmware
Prerequisites	see ID49
Test steps	
Test data	-
Expected Results	-
Test Case ID	65
Test case description	Process Crash or Stop for DFC Frontend Process
Prerequisites	see ID49
Test steps	
Test data	
Expected Results	
Test Case ID	79
Test case description	Services from back-end server cannot be disrupted by a (Denial of Service)-Attack on the back-end server
Prerequisites	1. Network Perimeter to disarm DoS attacks 2. DoS Tool (e.g. LOIC, XOIC)
Test steps	1. Perform a DoS attack on the service
Test data	
Expected Results	The service is not disrupted by a DoS attack
Test Case ID	86

Test case description	Spoofing of Destination Data Store Cloud Storage
Prerequisites	1. Enable TLS 2. Spoofing tools (e.g. sslstrip, yersinia)
Test steps	Try to spoof the connection using the desired tools
Test data	-
Expected Results	Spoofing attacks failed
Test Case ID	87
Test case description	Spoofing of Destination Data Store Configuration File
Prerequisites	see ID86
Test steps	
Test data	
Expected Results	
Test Case ID	91
Test case description	Authenticated Data Flow Compromised
Prerequisites	1. Enable TLS 2. Network Sniffer
Test steps	1. Use network sniffer to sniff the data flow (i.e. Wireshark, tcpdump) 2. UI: Wireshark 3. tcpdump: tcpdump -i <interface> host <ip>
Test data	
Expected Results	TLS prevents that data flow gets compromised
Test Case ID	103
Test case description	Data flow during administration is interrupted
Prerequisites	see ID6
Test steps	
Test data	
Expected Results	
Test Case ID	114
Test case description	Data Flow subscribe over TLS Is Interrupted
Prerequisites	see ID6
Test steps	
Test data	
Expected Results	
Test Case ID	115
Test case description	Spoofing of Destination Data Store Key Storage
Prerequisites	see ID86
Test steps	

Test data	
Expected Results	
Test Case ID	123
Test case description	Spoofing of the AVL Product (HW/SW) External Destination Entity
Prerequisites	see ID86
Test steps	
Test data	
Expected Results	
Test Case ID	124
Test case description	Elevation by Changing the Execution Flow in Broker
Prerequisites	
Test steps	Smart Hub A publishes under the Topic which is assigned of Smart Hub B
Test data	
Expected Results	Smart Hub A cannot publish a Topic B.
Test Case ID	145
Test case description	External Entity AVL Product (HW/SW) Denies Receiving Data
Prerequisites	-
Test steps	1. Send requests to the service 2. Make sure every request gets logged
Test data	-
Expected Results	1. Errors and information events are logged 2. Debug events are not logged during production
Test Case ID	155
Test case description	Weak Access Control for a Resource
Prerequisites	1. Several users with different permissions 2. Developers and administrators have to define restrictions of resources/groups/users
Test steps	1. Create unit tests to test the access of different resources with several users
Test data	Test resources which are restricted differently
Expected Results	Permissions are enforced correctly
Test Case ID	166
Test case description	Spoofing the Custom firmware Process
Prerequisites	see ID86
Test steps	
Test data	
Expected Results	
Test Case ID	167
Test case description	Introduce new software or overwrite existing software

Prerequisites	Several users with different permissions
Test steps	1. Try to install new software on the system 2. Try to uninstall software from the system 3. Use different users with different privileges for this task 4. Try to exploit the API to inject malicious updates
Test data	
Expected Results	Only users with certain permissions are able to install/uninstall software It is not possible to inject fault updates
Test Case ID	208
Test case description	Misconfiguration/Erroneous use or administration
Prerequisites	see Input Validation
Test steps	
Test data	
Expected Results	
Test Case ID	218
Test case description	Misuse of updates- software manipulation
Prerequisites	see ID167
Test steps	
Test data	
Expected Results	
Test Case ID	222
Test case description	Misuse of updates-Compromise of local/physical software update procedures
Prerequisites	see ID167
Test steps	
Test data	
Expected Results	
Test Case ID	231
Test case description	Unintended transfer of data
Prerequisites	Monitoring software
Test steps	1. Send (common/unusual) data from a service to another 2. Review monitoring system
Test data	
Expected Results	Monitoring is aware of unusual data sent from the service and system administrators get notified
Test Case ID	238
Test case description	Disrupt systems or operations
Prerequisites	see ID49

Test steps	
Test data	
Expected Results	
Test Case ID	245
Test case description	Broker used to attack DCF Frontend Process (Tampering)-Abuse of privileges by staff (insider attack)
Prerequisites	1. Users with different privileges 2. Network monitoring see ID124
Test steps	1. Use several different users to attack other devices 2. Use unprivileged (non-root) user to modify log files
Test data	
Expected Results	1. Log files contain IP address (src, dst), port (src, dst), timestamp and username 2. Monitoring software alerts administrators 3. Log files cannot be modified by unprivileged users
Test Case ID	246
Test case description	Broker used to attack DCF Frontend Process (Tampering)-Unauthorised internet access to the server
Prerequisites	see ID124
Test steps	1. Try to access the service through the internet
Test data	
Expected Results	The service is not accessible from the internet. Side note: If the service needs to be accessible, make sure to monitor each attempt and secure the service using whitelisting
Test Case ID	251
Test case description	Data held lost "data leakage" / compromised (Information Disclosure)-Unauthorised internet access to the server
Prerequisites	see ID246
Test steps	
Test data	
Expected Results	
Test Case ID	255
Test case description	DCF Frontend Process used to attack Broker (Tampering)-Unauthorised internet access to the server
Prerequisites	see ID246
Test steps	
Test data	

Expected Results	
Test Case ID	330
Test case description	Spoofing of CMS protocol
Prerequisites	Sender knows public key of recipient
Test steps	1. Create a new payload, encrypt it, and wrap the key for the recipient using the recipients public key
Test data	
Expected Results	The recipient verifies the senders' public key and thus does not decrypt the message and processes the payload.

Table 6: Applicable Test Cases

7 Human Aspects in Automated Model Checking of Security Protocols

In order to obtain secure cryptographic protocols for CPS, human factors must be taken into account. Ultimately, it is the human who is responsible to design, deploy, use, and maintain these systems. Attackers commonly target humans operating protocols instead of machines since the former are more error-prone [17]. Humans therefore represent a weak point which, if left unattended, can have a negative impact on all other parts of the system [18].

However, it is challenging for CPS protocol designers to consider possible human errors in different environments and understand how they play together with the broad threat landscape surrounding the IoT ecosystem [19]. In general, the security analysis of IoT protocols is a hard task, as vendors often react quickly on market demands and frequent code changes are required. Moreover, many different types of adversaries must be considered, depending on the deployment environment. For instance, some environments must only deal with insecure wireless channels, while others face adversaries capable to compromise long-term keys or corrupt random number generators [20]. Due to a lack of sound security analysis in the protocol design face, many of today's consumer and industrial IoT devices contain severe security vulnerabilities [21].

In order to prevent the release of buggy protocols, the scientific community came up with various decision procedures aiming to help developers to identify protocols weaknesses and possible attacks already in the design phase. Engineers frequently use pen-and-paper proofs for security analysis of IoT protocols. However, this method is time-consuming and error prone. Small parameter changes can lead to vast changes in the outcome of the analysis [22]. In contrast, tool-based automated formal analysis is more effective and flexible as it works with protocol abstractions.

Symbolic model checking is a useful approach for automatically detecting security flaws in cryptographic protocols, which has been recently used to uncover security vulnerabilities in several deployed security protocols (e.g., MQTT and CoAP [22], the vehicular networking (V2X) revocation protocol [23], TLS [24], and Yubikey protocols [25]). Albeit available model checking tools are promising, they cannot be directly applied to this project as they still face some challenges in verifying real-world protocols [26] and do not sufficiently take human factors into account.

First, the possibility of human error is not reflected in state-of-the-art symbolic model checking tools by default. Second, usability challenges of these tools are among the main reasons why they have not yet gained wide acceptance in industrial use cases and are currently only applied in academic circles. We therefore discuss the first attempt to integrate human behaviour into automated model checking tools and its suitability for verifying IoT protocols in Section 7.3. Novel approaches to overcome usability challenges of automated model checking tools and the evaluation of their applicability for IoT protocols in comparison to the original tools is presented in Section 7.4.

7.1 Symbolic Model Checking

In order to generate automated proofs of a cryptographic protocol, symbolic model checking tools require a mathematical, generic model (formal model), which formalizes the operational semantics of the protocol, the network, and the security properties [27] [28]. Protocol designers must construct such a formal model and implement it in the input language of the used tool. Thereby, implementation details of the protocol are abstracted.

The **formal model** represents cryptographic primitives as function symbols (terms), which are considered as black-boxes and handled as idealized mathematical constructs, i.e., perfect cryptography is assumed. A protocol is modelled as a set of roles. Each role can be played by one or more agents, which follows a specified sequence of events (as defined by the role). Agents can play multiple roles and each role can be instantiated any number of times. A protocol can only be verified in regards to the security properties it is supposed to fulfil. Properties are specified as part of the tools' input file. Verifiable properties include secrecy and different forms of authentication.

Based on the formal model, an automated tool verifies the protocol and identifies real-world attack scenarios caused by poor design or implementation choices [29]. Automated model checkers search for protocol traces that violate defined security properties. If any are found, they output possible attacks, if none are found, the security of the protocol is verified for the specified properties.

Currently, there are multiple tools available which offer automatic symbolic model checking of cryptographic protocols for an unbounded number of sessions. Tamarin [30] and ProVerif [31] are the most prominent ones. Kim et al. [22] assessed the applicability of different model checking tools for verifying IoT protocols and concluded, that Tamarin is the most suitable. ProVerif is efficient but does not guarantee termination for complex protocols. In contrast, Tamarin usually terminates even for complex protocols and can still deliver unbounded verification for various adversaries. Moreover, Tamarin also supports bilinear pairing and group key schemes [32], which is helpful for modelling IoT protocols. Furthermore, Tamarin is the only tool that offers an extension to model human error, which will be discussed in Section 7.3.1.

To give an example for an input language of an automated model checking tool, we briefly discuss **Tamarin's modelling language**. Protocol messages in Tamarin are defined as multiset rewriting rules and the properties as axioms or lemmas. The multi-set-rewriting rules consist of a left-hand-side, labels, and right-hand-sides. The left-hand sides describe input to the network and the right-hand-sides consumption of messages from the network. A simple example is shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**. A fresh value *nonce1* is constructed and sent to the network. Moreover, a message which contains *messageValue* is consumed from the network. Finally, the nonce and the received value are stored.

```
rule Step1:
        [ Fr(~nonce1),
        In(messageValue)]
           -->
        [ Out(~nonce1),
        StoreValues(~nonce1, messageValue) ]
```

Figure 17. Simple example protocol Tamarin input language.

Figure 18 illustrated the definition of the security property of secrecy. Basically, this lemma describes that in order for the protocol verification to succeed it should not be possible that somebody claims to have setup secret session keys, but the adversary knows at least one of them.

```
lemma session_key_secrecy:
        "not(
            Ex A B SKeNB #n.
            SessionKeys(A, B, SKeNB) @ n
            & (Ex #i. K(SKeNB) @ i)
        )"
```

Figure 18. Tamarin input file, specification of secrecy property.

7.2 Usable Security

In the last decade, the usable security research community has investigated areas where human failure jeopardizes security and privacy of IT systems. Thereby, it is the main assumption, that human errors are consequences of system properties and cannot be avoided by training humans, but by re-designing systems. In order to shape systems in a way to prevent human errors, the first step is to understand which knowledge is required by humans to securely use a system in its current state. The next step is to produce ideas how the system can be made more robust to human errors by requiring less knowledge from the user and designing the interfaces accordingly [33]. This is important, since humans usually react to interfaces and do not follow protocol specifications.

7.3 Formal Verification of Human Errors

Humans that operate security protocols often do not behave compliant to protocol specifications due to a lack of knowledge, prior experiences with similar technologies, or carelessness. However, the possibility of human failure is rarely considered by protocol designers. Hence, human errors are often responsible for security pitfalls. For example, phishing attacks where humans are tricked into revealing secrets (e.g., passwords, banking details) are reported frequently [34]. This is because a human will likely enter the requested information without thinking about authenticating the communication partner. When using IoT devices and when dealing with CPS, users play an important role as they must deal with interfaces for device monitoring or receive sensitive information from remote devices. Hence, a formal method for factoring in human errors is needed to ensure secure and robust operation of CPS.

7.3.1 Tamarin Extension for Modelling Human Errors

So far, automated model checking tools, which are commonly applied to verify protocol security, did not consider the possibility of misbehaving humans. Basin et al. [17] were the first to introduce an extension to Tamarin Prover, which includes human behaviour in the automated protocol verification process. Therefore, they formalize humans, taking into account their knowledge of a system, possible low attention and poor motivation. Human error is defined as deviation of a human from the protocol specification. They specify a human whose behaviour always deviates from the specification as *fallible* human, and a human that always behaves according to the specification as *infallible* human.

Humans are modelled as extended agents playing a specific role. Thereby, the human knowledge (HK) is tracked, storing tag-value pairs. At any point in time, the human can share HK and the adversary can query and update HK In order to model degrees between infallible and fallible humans and define more fine-grained how humans can deviate from protocol specifications, two possibilities exist. One can either specify a *skilled human* (modelled as infallible human that can make a fixed number of mistakes) or a *rule-based human* (modelled as fallible human that follows a set of rules). A skilled human could be somebody, who actually knows all steps of an authentication procedure, but skips specific parts (e.g., due to laziness). A rule-based human could be somebody who does not have any deeper knowledge about an authentication protocol and only knows that when he receives a message with a specific tag, he should compare it with a message in his HK that has the same tag. In Figure 19 an example is presented, where this rule is specified for a skilled human in Tamarin's input language.

ICompare(H,tag) := ∀Receive(*H,l,P,<t,m>*) ∈ *tr,m'*: <t,m> $\vdash_H$ <*tag,m'*> => InitK(*H,<tag,m'>*) ∈ *tr*

Figure 19. Rule for a skilled human in Tamarin.

Basin et al. used their extension to analyse an authentication protocol (MP-Auth) for transactions between a human and a server that uses a trusted device to store the server's public key. They revealed attacks caused by human error that could not have been found without their tool. When assuming a fallible human, they showed that it is feasible for an attacker to conduct a non-legitimate transaction *t'*. A fallible human would see *t'* displayed on the device and would - although the protocol specification says that the human should recognize the incorrect data - accept the transaction. Nevertheless, when assuming a rule-based human that follows the rule illustrated in Figure 19, such an attack can be proven infeasible.

Moreover, Basin et al. evaluated different authentication protocols and discussed their resistance to human error when assuming infallible, fallible, rule-based, and skilled humans. Based on this evaluation, they created heuristics which can be applied by secure authentication protocols in general to prevent human error. These heuristics also apply to IoT protocols and should be taken into account when designing authentication protocols for CPSs. In particular, Basin et al. suggested to enforce crucial human operations such as carrying out certain checks as far as possible in order to minimize the space for skipping these steps. For instance, the human can be forced to enter a code instead of being requested to compare two codes which can be skipped by fallible humans.

The long-term goal is to design protocols in a way that human errors are not possible anymore. On a short run, the results from model checking human error possibilities can be used to construct clear guidelines for what humans operating a specific protocol must do or not do.

7.3.2 Automatic Generation of Human Error Models

In order to simplify the cumbersome process of modelling human errors in protocols, Denzler [35] introduced a tool that automatically generates all set of possible errors for a specific protocol. Before this tool has been released, engineers had to manually go through a protocol model and specify possible errors for each human role. The tool enables engineers to specify an upper and lower bound for human errors. Therewith, human

behaviour can be modelled more accurately, and different backgrounds and environments can be considered. Hence, this tool is a crucial step towards usable model checking tools that take human factors into account. It can be applied in the realm of CPS protocols since it offers possibilities to model heterogeneous environments and human knowledge bases.

7.3.3 Security by Design: Guidelines for Human Error Prone Protocols

In general, the aim is to use the output of automatic model checking tools to design protocols in a way that are as resistant to human errors as possible. However, finding a trade-off between security-by-design (i.e., hiding complexity from the user) and making complexity visible to users must be assessed for each protocol separately to maximize protocol security.

Craggs et al. [36] formulated guidelines which can serve as a starting point when designing cryptographic protocols for CPS where humans are involved. They suggest proactive security ergonomic to prevent human security errors before they occur instead of mitigating negative effect afterwards. Moreover, they propose to tightly integrate security in the design of systems, instead of designing them as an add-on. Furthermore, they argue that the design should encourage secure behavior through secure default settings. Additionally, insecure behavior should be prohibited as far as possible. For instance, it should not be possible for a human to proceed to the next step when choosing a weak password. Finally, external validation of the protocol design (e.g., through automated testing tools) is required to ensure that human failure of protocol engineers cannot stay undetected. Thereby, automated model checking can be of great help.

7.4 Usable Symbolic Model Checking for Engineers

Another human factor to consider in symbolic model checking is that protocol engineers are in charge of finding a suitable protocol abstraction, i.e., over-approximate the protocol into a symbolic model (see Section 7.1), so that the protocol can be checked by the respective tool [37]. Therefore, they are in charge to model the message flow, network characteristics and possible attackers, as well as the required security properties. These symbolic models need to be sound in order to enable protocol verification. It is crucial that all characteristics which can possibly lead to attacks are preserved in the symbolic models and the security properties are defined correctly. However, it is currently an error-prone and time-consuming process to construct such models in the input languages required by the different tools. The languages of the currently most widely used model checkers, Tamarin and ProVerif, are not intuitive as they were mainly constructed for academics (see **Fehler! Verweisquelle konnte nicht gefunden werden.** and Figure 18). Moreover, the presentation of the results (attack scenarios) are currently hard to interpret by tool users.

To overcome these issues, we investigated novel user-centred approaches for automated symbolic model checking and assessed their applicability for IoT use cases.

7.4.1 Noise Explorer

Kobeissi et al. [38] introduced a publicly available online tool called Noise Explorer that automatically formally verifies protocols created with the Noise Framework [39]. This is a framework for designing cryptographic protocols by describing protocol messages in a simple language, from which a secure protocol is automatically

generated. The aim of the Noise Framework is to standardize a methodology to build protocols. Thus, creating multiple protocols with the same primitives and security goals (defined in the simple input language of the Noise framework) always yields the same result.

An example for a simple handshake protocol specified in the Noise Framework via the Noise Explorer user interface is given in Figure 20. This example illustrates a simple handshake protocol, where the receiving party shares his static public key s. Then, the sending party sends a fresh ephemeral key *e*, a Diffie Hellman shared secret *es* (of e and the responder's s), his public static key *s*, and a Diffie Hellman shared secret *ss* (of his and the responder's *s*).

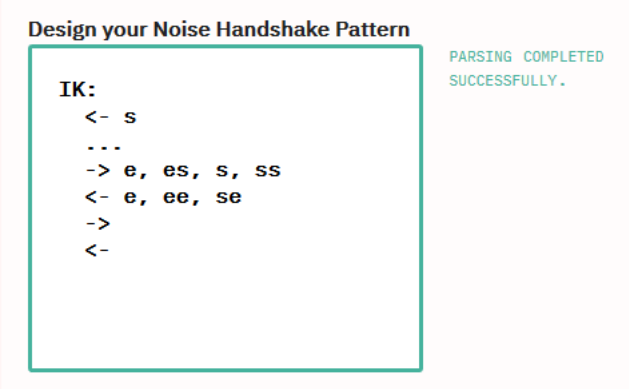


Figure 20. Simple handshake protocol specified in Noise Explorer (successful)

While defining a protocol, the user is actively guided to avoid user error. The tool provides real-time feedback on security vulnerabilities or typos, so that the model can be changed accordingly (see Figure 21). Moreover, users can inspect security properties of single messages of different handshake patterns in an extended view.

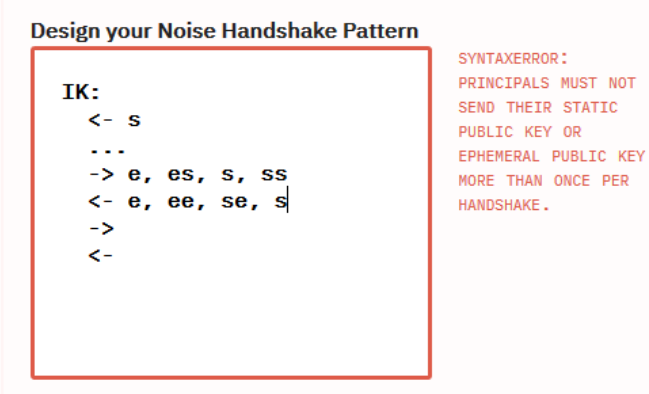


Figure 21. Handshake protocol with an user error in Noise Explorer

Noise explorer offers the possibility to automatically construct formal models of such protocols. Hence, the engineers do not need to manually implement formal models anymore. The generated model is checked by ProVerif. Noise Explorer parses the results of ProVerif and provides a detailed interactive page describing the analysis results. In comparison to current tools, it is easier for engineers to understand which attacks are possible due to poor protocol design and how it can be fixed. Moreover, there is an option to automatically generate software implementations in Go or Rusk.

As Noise explorer can only be used for handshake protocols, it's applicability for model checking protocols of CPS is limited. However, the tool is a step in the right direction to make automated model checking tools more usable.

7.4.2 Verifpal

In addition to Noise Explorer, Kobeissi et al. [40] presented a novel software called Verifpal for verifying security protocols, which aims at making automated model checking more intuitive and usable for engineers. Verifpal has been used for verification of real-world-protocols such as Signal or TLS 1.3. In order to lower the entry barriers for engineers to formal verification, Kobeissi et al. introduce a new modelling language which is closer to human language than the input languages of current automated model checking tools. It is not possible to define custom cryptographic primitives, as this is assumed as error prone. Instead, default cryptographic functions for symmetric and asymmetric encryption and decryption (e.g., AEAD_ENC, AEAD_DEC) are provided. Figure 22 shows an example of a simple protocol defined in Verifpal. When specifying roles, Verifpal uses intuitive terms related to human languages such as the terms *knows* or *generates.* In this example protocol, the principals Alice and Bob communicate with each other. Alice generates a random value *a* and carries out a Diffie Hellman exponentiation. Then, Alice sends the result to Bob. Bob as well generates a random value *b*, carries out a Diffie Hellman exponentiation, encrypts the result and sends an encrypted message *e1* to Alice. Alice in turn decrypts the message.

```
attacker[active]
principal Bob[]
principal Alice[
  generates a
  ga = G^a
]
Alice -> Bob: ga
principal Bob[
  knows private m1
  generates b
  gb = G^b
  e1 = AEAD_ENC(ga^b, m1, gb)
]
Bob -> Alice: gb, e1
principal Alice[
  e1_dec = AEAD_DEC(gb^a, e1, gb)?
]
```

Figure 22. Simple protocol in Verifpal.

Verifpal's specification language is more intuitive than the one used in traditional tools (see Figure 13 for a comparison). In the input language of current tools, the attacker model must be manually defined. In Verifpal, users do not have to define any attacker properties, but must only choose between the definition of an *active* or *passive* attacker. On the one hand, this simplifies the tool usage for developers. On the other hand, custom definitions of attacker models are not possible in Verifpal. However, in real-world scenarios such fine-grained definitions are usually not needed. In order to specify security properties, engineers can formulate authentication or confidentiality queries such as illustrated in Figure 19. This is an usability improvement to

current automated model checking tools, where the properties have to be manually specified in the input language of the used tool (see Figure 14 for a comparison).

```
queries[
        confidentiality ? message1
        authentication ? Alice -> Bob: value1
]
```

Figure 23. Security property specification in Verifpal.

Apart from that, the verification results of Verifpal are more readable for engineers than the results of Tamarin or ProVerif. It is easier to conduct changes in the protocol specification in accordance to the verification results, since the attacks are tied to real-world scenarios. Hence, Verifpal offers multiple usability enhancements in comparison to traditional automated model checking tools and can be applied for user-friendly modelling of CPS protocols.

7.4.3 Automated model checking and protocol standardization

Some academic papers show how current automated model checking tools can be applied to standardized protocols (e.g., TLS [24]). However, the usage of these tools by standardization delegates has not yet become established in practice. In order to explore reasons for this, Henda et al. [41] conducted a study investigating the suitability of three popular formal verification tools (Scyther [42], Tamarin Prover [30], ProVerif [31]) during the standardization process of a real-world security protocol. Thereby, they reported how capable the different tools are to model a rapidly changing system as it is developed. Henda et al. concluded that Tamarin and ProVerif are most applicable for their use case. However, usability challenges of both tools are discussed as reasons for why current tools can not readily be applied to model real-world protocols. When using Tamarin, despite a graphical user interface (GUI) being available, the users still have to be familiar with some aspects of the tool's theoretical foundation in order to correctly use it. For ProVerif, they mention that a GUI is missing, but an extensive tutorial is available. It has to be noted, that since the release of the paper, Tamarin also published a new and more user-friendly manual with increased usability for engineers.

Currently, the implementation of standardized cryptographic algorithms is an error-prone and hard task for engineers. Security guarantees and assumptions of cryptographic algorithms are complicated, and it is not intuitive to turn them into sound and secure implementations. Standards usually do not specify interfaces (APIs) through which cryptographic algorithms can be implemented, which in practice often leads to errors. As a solution to these challenges, Bhargavan et al. introduced the formal verification language hacspec [43]. Hacspec is the result of a workshop which brought together crypto library developers and verification framework researchers with the goal to find a suitable trade-off between usability and ease of formal verification. The language is similar to pseudocode in cryptographic standards and can thus easily be understood by developers and engineers.

Hacspec makes standards easy to read and implement for engineers. At the same time, it facilitates the usage of formal verification and therewith, makes standards and implementations comparable to reference models.

Currently, even if an engineer succeeds in constructing a sound formal model, it is hard to check the soundness of the model (e.g., by comparing it to a standard or to another model) since different tools follow different rules. With hacspec, the standards itself provide formal reference models. When pseudocode of standards is written in hacspec, this is beneficial for two reasons. First, the algorithm itself can be tested against a static type system and syntax checking tool before the standard is released. Second, this enables developers to check their implementations of the respective standard for compliance (i.e., functional equivalence) with the formal model.

8 Overall Summary

Due to rising new issues of the Internet of Things and the increasing need for connection and cooperation in cyber physical systems, the need of novel approaches of security analysis techniques growths. Therefore, this document describes new approaches for formally analysing hardware, protocols, system architecture as well as test case generation.

Chapter 2 shows the use of formal hardware property checks in order to provide security on basic of the building block of components in the network layer to detect hardware Trojans.

Then, chapter 3 describes the formal verification of side-channel approach, which resulted in the first formally verified AES S-box design that requires only two random bits for the initial sharing of its inputs and requires no online randomness to achieve first-order security in the probing model.

Chapter 4 describes hardware apps and the use of dynamically exchangeable runtime checkers as hardware apps, in which several functions were implemented.

In chapter 5 the threat modelling approach using the STRIDE model is shown. Based on extended template, a threat model tailored for the IoT4CPS project has been created.

The results are then taken to chapter 6, where these threats were evaluated in regards of applicableness with the help of various cyber security and software development experts. As a result, a penetration test catalogue containing several reusable test cases for the automotive industry was created.

Chapter 7 discusses the human aspects in automated model checking of security protocols. Here, symbolic model checking, the formal verification of human errors as well as usable symbolic model checking for engineers were evaluated.

This document reflects the variety of approaches that were developed in IoT4CPS at the area of functional and formal checks addressing different areas ranging from human factors, protocol level, hardware & software up to system architectures and test case generation. This was done in close cooperation between industrial partners and research institutions in order to provide the foundation for safe and secure IoT-based applications.

9 References

[1] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub and R. Zucchini, “Strong Non-Interference and Type-Directed Higher- Order Masking,” in *ACM Conference on Computer and Communications Security*, Vienna, 2016.

[2] D. Dolev and A. C. Yao, “On the security of public key protocols,” in *IEEE Transactions on information theory*, IEEE, 1983, pp. 198-208.

[3] M. Utting, B. Legeard and A. Pretschner, “A Taxonomy of model-based testing,” in *Software Testing Verification and Reliability*, 2012, pp. 297-312.

[4] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann and L. Nachmanson, “Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer,” in *Formal Methods and Testing*, Heidelberg, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39-76.

[5] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner and R. Ramler, “GRT: Program-Analysis-Guided Random Testing,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, 2015.

[6] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe and M. Yamamoto, “GRT at the SBST 2015 Tool Competition,” in *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*, Florence, Italy, 2015.

[7] H. S. M. Tehranipoor, “trust-HUB,” [Online]. Available: https://www.trust-hub.org/. [Accessed 2020].

[8] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel and T. Unterluggauer, “ISAP - Towards Side-Channel Secure Authenticated Encryption,” in *IACR Trans. Symmetric Cryptol*, Ruhr-Universität Bochum, 2017, pp. 80-105.

[9] R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard and J. Winter, “Formal Verification of Masked Hardware Implementations in the Presence of Glitches,” in *EUROCRYPT*, 2018, pp. 321-353.

[10] G. Barthe, S. Belaïd, P.-A. Fouque and B. Grégoire, “maskverif: a formal tool for analyzing software and hardware masked implementations,” in *ESORICS*, 2019.

[11] Y. Ishai, A. Sahai and D. A. Wagner, “Private circuits: Securing hardware against probing attacks,” in *CRYPTO*, 2003.

[12] A. Biryukov, D. Dinu, Y. L. Corre and A. Udovenko, “Optimal first-order boolean masking for embedded iot devices,” in *CARDIS*, 2017.

[13] P. Schwabe and K. Stoffelen, “All the AES you need on Cortex-M3 and M4,” in *SAC*, 2016.

[14] J. Boyar and R. Peralta, “A Small Depth-16 Circuit for the AES S-Box,” in *IFIP AICT*, 2012.

[15] L. Bassham, A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, N. Heckert and J. Dray, “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,” NIST, Gaithersburg, 2010.

[16] F. Veljkovic, V. Rozic and I. Verbauwhede, “Low-Cost Implementations of On-the-Fly Tests for Random

Number Generators," in *Design, Automation and Test in Europe*, Dresden, 2012.

[17] A. Shostack, Threat Modeling - Designing for Security, Crosspoint: Wiley, 2014.

[18] M. Howard and D. LeBlanc, Writing secure code, Redmond: Microsoft Press, 2014.

[19] "istqb," 2020. [Online]. Available: https://www.istqb.org/. [Accessed 11 02 2020].

[20] D. Basin, S. Radomirovic and L. Schmid, "Modeling human errors in security protocols," *2016 IEEE 29th Computer Security Foundations Symposium (CSF),* 2016.

[21] B. Craggs and A. Rashid, "Smart cyber-physical systems: beyond usable security to security ergonomics by design," *IEEE/ACM 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS),* 2017.

[22] Abdul-Ghani, H. Akram, D. Konstantas and M. Mahyoub, "A comprehensive IoT attacks survey based on a building-blocked reference model}," *IJACSA International Journal of Advanced Computer Science and Applications,* 2018.

[23] J. Wurm, K. Hoang, O. Arias, A.-R. Sadeghi and Y. Jin, "A comprehensive IoT attacks survey based on a building-blocked reference model," *21st Asia and South Pacific Design Automation Conference (ASP-DAC),* 2016.

[24] J. Deogirikar and A. Vidhate, "Security attacks in IoT: A survey," *International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC),* 2017.

[25] J. Y. Kim, R. Holz, W. Hu and S. Jha, "Automated analysis of secure internet of things protocols," *Proceedings of the 33rd Annual Computer Security Applications Conference,* 2017.

[26] J. Whitefield, L. Chen, F. Kargl, A. Paverd, S. Schneider, H. Treharne and S. Wesemeyer, "Formal analysis of V2X revocation protocols," *International Workshop on Security and Trust Management,* 2017.

[27] C. Cremers, M. Horvat, J. Hoyland, S. Scott and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security,* 2017.

[28] R. Künnemann and G. Steel, "YubiSecure? Formal security analysis results for the Yubikey and YubiHSM," *International Workshop on Security and Trust Management,* 2012.

[29] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. J. F. Cremers, K. Liao and B. Parno, "SoK: Computer-Aided Cryptography," *IACR Cryptology ePrint Archive,* 2019.

[30] D. Basin, C. Cremers and C. Meadows, "Model checking security protocols," *Handbook of Model Checking,* 2018.

[31] B. Blanchet, "Security protocol verification: Symbolic and computational models," *International Conference on Principles of Security and Trust,* 2012.

[32] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. J. F. Cremers, K. Liao and B. Parno, "SoK: Computer-Aided Cryptography," *IACR Cryptology ePrint Archive,* 2019.

[33] S. Meier, B. Schmidt, C. Cremers and D. Basin, "The TAMARIN prover for the symbolic analysis of security

protocols," *International Conference on Computer Aided Verification,* 2013.

[34] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security,* 2016.

[35] B. Schmidt, R. Sasse, C. Cremers and D. Basin, "Automated Verification of Group Key Agreement Protocols," *IEEE Symposium on Security and Privacy,* 2014.

[36] L. Schmid, "Human errors in secure communication protocols," *ETH Zürich,* 2015.

[37] B. B. Gupta, A. Tewari, A. K. Jain and D. P. Agrawal, "Fighting against phishing attacks: state of the art and future challenges," *Neural Computing and Applications,* 2017.

[38] A. Denzler, "Automatic Analysis of Communication Protocols with Human Errors," 2016.

[39] B. Craggs and R. Awais, Smart cyber-physical systems: beyond usable security to security ergonomics by design, 2017.

[40] B. T. Nguyen, C. Sprenger and C. Cremers, "Abstractions for security protocol verification," *Journal of Computer Security,* 2018.

[41] N. Kobeissi, G. Nicolas and K. Bhargavan, "Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols," *IEEE European Symposium on Security and Privacy (EuroS&P),* 2019.

[42] T. Perrin, "The noise protocol framework," 2016.

[43] N. Kobeissi, "Verifpal: Cryptographic Protocol Analysis for Students and Engineers," *Cryptology ePrint Archive, Report 2019/971,* 2019.

[44] N. B. Henda, K. Norrman and K. Pfeffer, "Formal Verification of the Security for Dual Connectivity in LTE," *IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering,* 2015.

[45] C. Cremers, "The Scyther Tool: Verification, falsification, and analysis of security protocols," *International conference on computer aided verification,* 2008.

[46] K. Bhargavan, F. Kiefer and P.-Y. Strub, "hacspec: towards verifiable crypto standards," *International Conference on Research in Security Standardisation,* 2018.

[47] IOT4CPS Consortium, *Project description for proposals: Trustworthy IoT for CPS,* 2017.

[48] M. Drobics, "IoT4CPS Common Reference Architecture," IOT4CPS Consortium, 2008.

[49] IOT4CPS Consortium, "IOT4CPS Assets," [Online]. Available: https://portal.ait.ac.at/sites/AHIT/IoT-LP/_layouts/15/start.aspx#/Wiki/Asset%20Management.aspx. [Accessed 30 8 2018].

[50] AIOTI Consortium, "AIOTI WG11 – Smart manufacturing," 2015.

[51] Verein Industrie 4.0 Österreich, "Ergebnispapier "Forschung, Entwicklung & Innovation in der Industrie 4.0"," 2018.

[52] ECSO Consortium, "ECSO European Cybersecurity Strategic Research and Innovation Agenda (SRIA) for a contractual Public-Private Partnership (cPPP)," 2016.

[53] ECSEL Consortium, "ECSEL 2017 Multi Annual Strategic Research and Innovation Agenda for ECSEL Joint Undertaking," 2017.

[54] Siemens, “Charter of Trust”.

[55] European Commission, “European Commission C-ITS Platform phase I final report of January 2016”.

[56] European Commission, “European Commission C-ITS Platform phase II final report of September 2017,” 2017.

[57] ECSEL JU, “Multi-Annual Strategic Plan (“MASP”) 2018, ECSEL JU”.

[58] ERTRAC, “Strategic Research Agenda: Input to 9th EU Framework Programme”.

[59] Josef Affenzeller et al., “Austrian Research, Development & Innovation Roadmap for Automated Vehicles (BMVIT)”.

[60] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner and R. Ramler, “GRT: An Automated Test Generator Using Orchestrated Program Analysis,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, 2015.