MALWARE TOLERANCE: DISTRIBUTING TRUST OVER MULTIPLE DEVICES

by

Michael Denzel

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
May 31, 2018

ABSTRACT

Current security solutions try to keep the adversary out of the computer infrastructure. However, with zero-day exploits and undetectable intrusions like certain rootkits [4, 48, 67, 68] the assumption that attacks can be blocked does not hold any more.

This work presents the concept of malware tolerance accepting the fact that every device might be compromised at some point in time. The concept aims to distribute trust over several devices so that no single device is able to compromise security features by itself. Simply put, malware tolerance tries to eliminate every single point-of-failure in an architecture.

I create three malware-tolerant techniques to demonstrate the feasibility of the concept. This thesis introduces a trusted input system which delivers keystrokes securely from the keyboard to a recipient even if one of its components is compromised. The second approach is the design of a self-healing Industrial Control System, a sensor-actuator network to securely control a physical system. If an adversary manages to compromise one of the components, the Industrial Control System remains secure and can even recover from attacks. In order to assess the self-healing technique, I evaluate my open-source proof-of-concept implementation built on top of *FreeRTOS*. Lastly, this thesis proposes a mesh network architecture aimed at smart-home networks without assuming any device in the network invulnerable to attacks. It applies isolation mechanisms to otherwise flat mesh networks and can automatically quarantine compromised devices.

To analyse the security of these approaches, I develop a new model, the multiple Trusted Computing Bases model. This thesis gives formal security proofs based on this model with state-of-the-art protocol verifier ProVerif. The proof scripts are open-source.

Malware tolerance proved to be an effective new way of thinking that highlights security insufficiencies by analysing architectures with regard to single points-of-failure. These fragile points are the weakest ones of an architecture and should be eliminated or strengthened early on since their deficiency exposes the rest of the infrastructure.

ACKNOWLEDGEMENTS

First of all, I would like to express my appreciation to my supervisor, Prof. Mark Ryan, for his help and advice. He provided very helpful guidance throughout my PhD.

I am also very grateful to Dr. Alessandro Bruni for helping me with the proofs of Smart-Guard and teaching me ProVerif. Without his introduction to ProVerif, this thesis would not have been possible the way it is now. In this regard, I wish to thank Dr. Eike Ritter, too, with whom I modelled the self-healing ICS architecture in ProVerif/StatVerif. Both of my co-authors contributed valuable feedback to this work.

A special thanks goes to Federico De Meo for continual, very helpful discussions and suggestions. Also, Dr. Jiangshan Yu provided me with valuable feedback and advice during my time at the University of Birmingham.

I wish to acknowledge the help of Dr. Flavio Garcia and Dr. David Parker who were part of my supervision group and supported me with useful suggestions.

I would like to thank the research group of RITICS/SCEPTICS/CAPRICA/MUMBA/CEDRICS who allowed me to present my work to them and gave some very helpful feedback.

Furthermore, I would like to thank my parents, my sister, and my grandparents for their constant support and for proofreading my thesis.

Lastly, I want to mention that the work on Smart-Guard was partially supported by the EPSRC project “Analysing security and privacy properties”.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
APT	Advanced Persistent Threat
BIOS	Basic Input/Output System
BSS	Block Started by Symbol
CAN	Controller Area Network
COM	Communication Port
CRT	Chinese Remainder Theorem
CVE	Common Vulnerability and Exposures
DH	Diffie-Hellman Key Exchange
DMA	Direct Memory Access
DoS	Denial of Service
DRTM	Dynamic Root of Trust for Measurement
EPC	Enclave Page Cache
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
FSA	Finite State Automaton
GCM	Galois Counter Mode
GKA	Group Key Agreement

ICS	Industrial Control System
IDS	Intrusion Detection System
I/O	Input/Output
IoT	Internet of Things
IPS	Intrusion Prevention System
IRE	Intrusion Resilient Encryption
IRQ	Interrupt
IRS	Intrusion Resilient Signature
MAC	Message Authentication Code
MANET	Mobile Ad-Hoc Network
MITB	Man-in-the-Browser
MITM	Man-in-the-Middle
MMU	Memory Management Unit
mRSA	mediated RSA
NIST	National Institute of Standards and Technology
NS	Non-Secure
PLC	Programmable Logic Controller
QR	Quick Response
RNG	Random Number Generation
RSA	Rivest-Shamir-Adleman Cryptosystem
RTOS	Real-Time Operating System
SCR	Secure Configuration Register
SDN	Software Defined Network
SGX	Software Guard Extensions

SMC	Secure Monitor Call
SSH	Secure Shell
SVM	Secure Virtual Machine
SWI	Software Interrupt
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TC	Trusted Computing
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TTP	Trusted Third Party
TZIC	TrustZone Interrupt Controller
VANET	Vehicular Ad-Hoc Network
VLAN	Virtual Local Area Network
VM	Virtual Machine
WANET	Wireless Ad-Hoc Network
WSN	Wireless Sensor Network
ZitMo	Zeus in the Mobile

PUBLICATIONS

The following papers were published in course of the thesis. Especially Chapters 3, 4, 5, and 6 present these papers and, thus, reproduce and quote them.

1. M. Denzel, A. Bruni, and M. D. Ryan. Smart-Guard: Defending User Input from Malware. In *Intl IEEE Conf. on Advanced and Trusted Computing*, pages 502–509. IEEE, 2016. doi: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.97. URL `http://ieeexplore.ieee.org/abstract/document/7816885/`. accessed: 2017-09-15

2. M. Denzel, M. Ryan, and E. Ritter. A Malware-Tolerant, Self-Healing Industrial Control System Framework. In *IFIP Advances in Information and Communication Technology ICT Systems Security and Privacy Protection*, volume 502. Springer, 2017. doi: https://doi.org/10.1007/978-3-319-58469-0_4. URL `https://link.springer.com/chapter/10.1007%2F978-3-319-58469-0_4`. accessed: 2017-07-23

3. M. Denzel and M. Ryan. Malware tolerant (mesh-)networks. 2018. (submitted to CANS 2018)

CHAPTER 1

INTRODUCTION

During the last decades, we saw new attacks appearing, amongst them are CryptoLockers and Advanced Persistent Threats (APTs). While the former encrypt a system and ask for ransom, the latter are stealthy attacks often based on rootkits which quietly spread over the network and escalate their privileges [128]. These attacks targeted critical infrastructures – Industrial Control System (ICS) – like power plants (e.g. Stuxnet [85, 86]), medical institutions, public transport, and communication services (e.g. Wannacry [64]). While those incidents received a lot of media attention, there are more examples like Duqu, Flame, Red October, MiniDuke [140], Gauss, Energetic Bear, Epic Turla [79], the attack on the Maroochy Water Services [1], and the attack on a German steel mill [23]. Some attacks, like Operation Aurora [47, 21, 113, 127] or Stuxnet, are supposedly supported by governments.

These attacks demonstrate that security is not equal to systems being invulnerable to attacks – every system can be potentially compromised. Several projects started to raise awareness about this matter: The Shodan search engine began to crawl for vulnerable ICSes (see Fig. A.2 in the Appendix) and Conti et al. [31] analysed the security of an entire city. They cross-examined electricity supply, water purification, sewage treatment, natural gas, oil industry, customs and immigration, hospital and health care, and food industry to gain knowledge on their vulnerability and dependencies. It became evident that these attacks harm more than only a few home computers.

In addition to these issues, there are proof-of-concept rootkit attacks which cannot (yet) be prevented; those are Debug-Register (DR) rootkits [4], System Management Mode (SMM) rootkits [48], Advanced Configuration and Power Interface (ACPI) rootkits [67], and Peripheral Component Interconnect (PCI) rootkits [68]. The current PC architecture cannot defend against these attacks and a solution would likely include partly re-developing the computer itself. This will doubtfully happen in the near future, leaving all computers vulnerable to these rootkits.

Previous research focused on mitigating attacks by blocking them with a firewall, using anti-virus and Intrusion Detection Systems (IDSes) to detect them, and preventing a few of them with signature, anomaly, or specification-based network intrusion detection systems. Critical systems were isolated from the corporate network without wired or wireless connection. This so-called air-gap was a last resort to prevent attacks on high risk systems. Nevertheless, Stuxnet [85, 86] skipped all mitigation techniques including the air-gap and demonstrated how they can be circumvented. There is also indication that coffee machines [8] were used by a CryptoLocker to bridge the air-gap to an industrial plant.

To secure the computer architecture in general, special chips and hardware features were introduced to provide a *root of trust*. The first such chip was the Trusted Platform Module (TPM) chip followed by ARM TrustZone and Intel Software Guard Extensions (SGX). Those chips provide hardware enforced security properties to bootstrap a computer securely from a small starting point. Although they minimise the components which need to be trusted, they have not been successful due to two drawbacks. They are not flexible enough because they lock down the architecture too intensively. The other shortcoming is that they are not invulnerable to attacks. ARM TrustZone was exploited [116, 15] and the recent *meltdown* and *spectre* attack [89] exploited vulnerabilities in Intel processors. This suggests that the Intel and ARM architecture are complex and, therefore, one must assume that even hardware features are potentially vulnerable.

While all these mitigation techniques are a viable defence line and should stay in place, they are not sufficient. This led us to re-thinking how secure systems should be built, considering that no part of them can be assumed trustworthy. Baize et al. [14] already suggested that security must be “built-in” and compromising one system element should not compromise the overall security of the system.

The research objective is, thus, to examine if systems can be built without assuming any component invulnerable to attacks. There should be no single component on which the security of the entire architecture depends. This work calls the idea *malware tolerance.*

In practice, this could look like this: Let us consider a malware-tolerant system consisting of e.g. three diverse computers (one with TrustZone, one with SGX, one with a TPM). Diversity is necessary to prevent multiple devices being compromised with the same exploit. If one computer is compromised – even if this includes its TPM and all of its cryptographic keys – the remaining two computers must be able to keep the overall system secure and running. This includes preventing leakage of data and resources of the overall architecture, although all part-secrets of the compromised computer are leaked. This means evidently that data, resources, and keys must be securely distributed over the three computers. In other words, there must be no single point-of-failure.

Malware tolerance is not limited to three computers, the following chapters also demonstrate the concept with only one computer and with many computers in a network. This thesis is aimed at elaborating the concept of malware tolerance regarding its feasibility. Its goal is not to provide a full implementation.

Contributions:

- Architectures should be built keeping in mind that every component will potentially be compromised. This thesis proposes the concept of malware tolerance (Chapter 3): All single points-of-failure should be replaced by distributing trust upon several, independent components or devices. Components that are similar or depend on each other, i.e. that can be compromised with the same exploit, count as one part.
- I further demonstrate how to apply this concept in three scenarios: trusted input (Chapter 4), Industrial Control Systems (Chapter 5), and mesh networks (Chapter 6).
- To give proof of the security guarantees, the state-of-the-art protocol verifier ProVerif[1] was used in each scenario. These ProVerif scripts are available online as open-source (see Table 1.1).
- I also implemented a basic proof-of-concept operating system for ICS which utilises FreeRTOS and ARM TrustZone to provide self-healing capabilities. I released it as open-source (see Table 1.1).

Topic	Link
Proofs Smart-Guard	`https://github.com/mdenzel/smartguard`
Proofs malware-tolerant, self-healing ICS	`https://github.com/mdenzel/malware-tolerant_ICS_proofs`
Proofs malware-tolerant mesh network	`https://github.com/mdenzel/malware-tolerant_mesh_network_proofs`
Self-healing RTOS	`https://github.com/mdenzel/self-healing_FreeRTOS`

Table 1.1: Links to ProVerif proof scripts and the proof-of-concept implementation

[1] `http://proverif.inria.fr`

CHAPTER 2

BACKGROUND

This chapter gives a general overview of previous security concepts. The following chapters also compare this work to corresponding approaches and give more details.

I focus on defence techniques against malicious attacks. There are also techniques to prevent common faults but they usually fail versus a methodical adversary and are omitted in this overview.

2.1 Literature Review

The idea to spread information across various devices originated from distributed systems (e.g. [40]). Initially, this was done to recover from failures through Byzantine fault tolerance where $2 * f + 1$ replicated servers can tolerate failures of a fraction f of the servers [139]. Redundancy was also applied to tolerate attacks in which case $3 * f + 1$ systems were used [6]. However, it is then essential to diversify these replicas [57, 58], otherwise all replicas could be compromised with the same exploit. To achieve diversity, software was developed multiple times by different teams, so-called *N-version programming* [24, 118]. But, there is evidence that N-version programming is not effective against attacks because teams make related errors [112].

N-variant systems on the other hand diversify systems wilfully e.g. by running on two different CPUs (x86 and PowerPC) [33], using different stack directions [111], or changing the representation of the data [142]. This proved to be more effective against attacks, since diversity is deliberately built-in.

2.1.1 Intrusion Tolerance for Critical Infrastructures

Simultaneously, Byzantine fault tolerance [6] was developed into intrusion tolerance. Initially, this was used in IDSes, i.e. systems that identify malicious attacks. Adversaries attacked these systems first before compromising the actual target because IDSes are an early warning system. To counter attacks turning off detection systems, IDSes evolved to intrusion tolerant systems which utilised redundancy and restarted copies in case of failures [83, 77].

Wang et al. [141] proposed an intrusion tolerant proxy architecture to shield servers off from malicious requests. The architecture can switch to backup proxies and backup servers if some of them fail. Audit control suggests the authors considered automatically reviewing and resetting the architecture and they also mentioned diversity, however, both topics were not explained in detail.

Verissimo et al. saw intrusion tolerance more from a recovery perspective incorporating IDSes, secure communication, replication, recovery, and fail-safe behaviour [137]. Verissimo created a model for distributed systems communication [138] by assuming messages to be only partially synchronised. This model was called the wormhole subsystem. They experimented with proactive recovery – i.e. reinstalling the system every now and then – but concluded that it is not sufficient [120]. The group developed a proactive and reactive recovery approach [119] where replicas can additionally trigger recovery of other potentially faulty replicas. They combined all their work to create a recovering, intrusion-tolerant firewall using a component they called *critical utility infrastructure resilience (crutial) information switch* [16, 121].

Similarly, Platania et al. [107] created an architecture to recover from attacks. They periodically reinstalled machines via a netboot and diversified replicas with a compiler.

Replication and recovery requires additional machines which is costly and, therefore, is more aimed at critical infrastructures and Industrial Control System than consumer devices.

2.1.2 Industrial Control Systems

Industrial Control Systems (ICSes) are sensor-actuator networks or also cyber-physical systems, i.e. computers monitoring and controlling processes of the physical world. The priority of these systems is to ensure safety and reliability of the architecture but recently also protection against malicious attacks became essential. The term *secure control* describes techniques ensuring integrity and availability in the face of attacks [26, 27].

The core components of ICSes are so-called Programmable Logic Controllers (PLCs), which nowadays are essentially commodity computers with specialised software to satisfy the requirement for high availability and real-time operation. Due to these requirements, they cannot run common defensive measures like an anti-virus. Defensive mechanisms have, thus, to be deployed (less effective) elsewhere in the network. Moreover, PLCs have a long lifetime (10-20 years) and are not usually patched to avoid downtime and destroying the devices [122] through updates. A corrupted patch can render a PLC unusable possibly leading to a shutdown of part of the network which is potentially life-threatening.

Homeland Security [70] as well as the National Institute of Standards and Technology (NIST) [122] recommend a strategy called *defence in depth* that tries to deploy defences at every layer of the network. Figure 2.1 shows an example layout with different network zones. Firewalls and IDSes isolate these zones from each other resulting in a layered network with IDS at intersection points. The field site with PLCs, sensors, actuator, and physical systems is the so-called control loop. This part of the architecture is the actual cyber-physical system and has hardly any (often no) defensive measures apart from the firewall in front of it due to availability and real-time constraints. The control loop can be at a different location than the control centre. The corporate network is entirely separated to prevent attacks from escalating to the physical system.

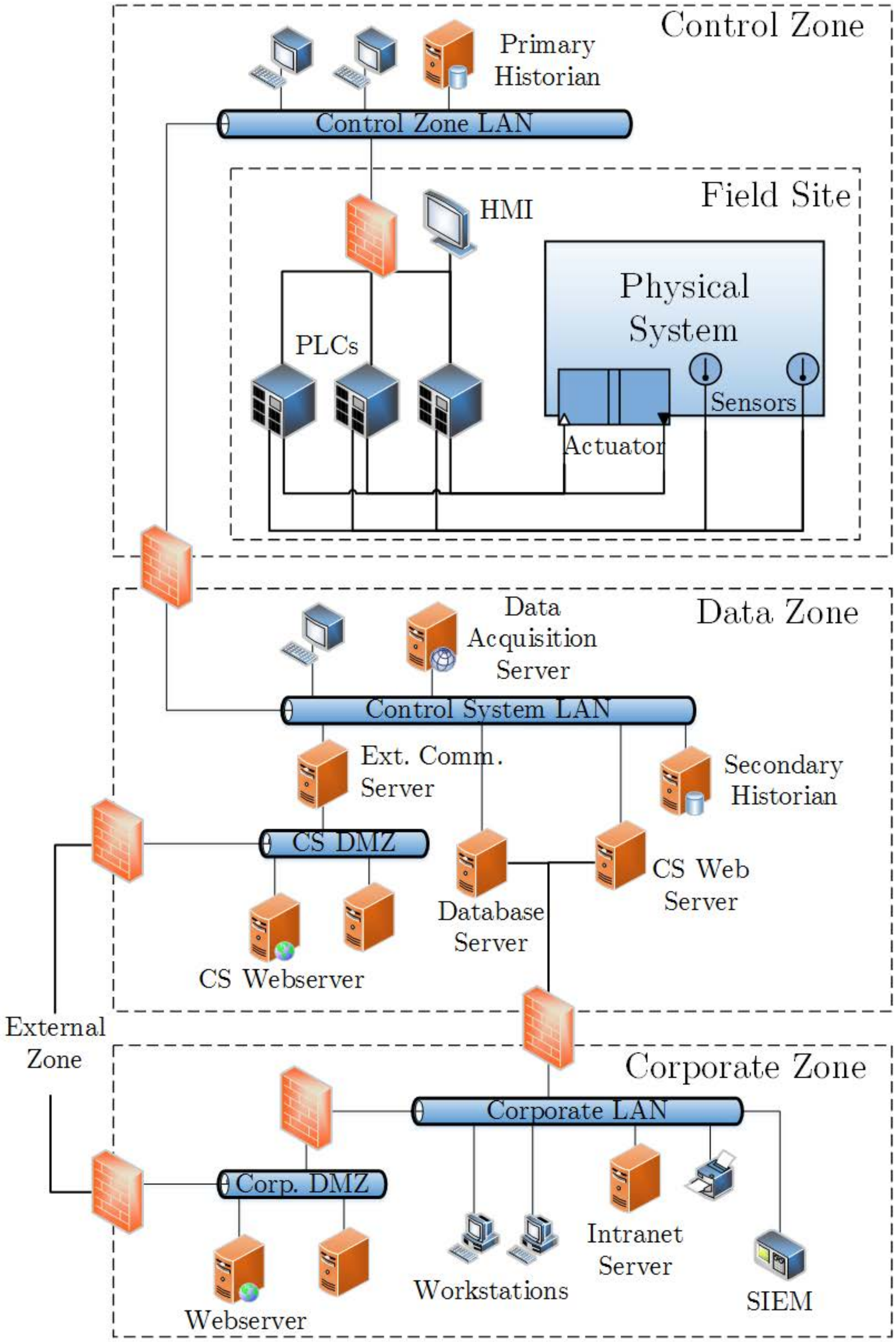


Figure 2.1: Example Industrial Control System architecture reproduced from [70]

An example for such a control loop would be a temperature control system with e.g. some water tanks which should neither freeze nor boil. The PLC would read the temperature from a sensor and adjust the heating/cooling of the water to maintain a temperature between 0 to 100°C.

Fielder et al. [52] studied the effectiveness of the defence in depth strategy from a theoretical perspective turning the scenario into an optimisation problem between attacker and defender. They analysed how the defence in depth strategy performs compared to a critical component defence against a methodical attacker. The authors concluded that defence in depth is more effective against a greedy attacker; critical component defence is better suited against a methodical adversary. That means for an ICS with valuable PLCs (i.e. the single point-of-failure), critical component defence should be applied. However, if there are multiple valuable targets for the adversary, defence in depth is favourable.

Coexisting to Defence in Depth, there is also a *Defence in Breadth* which is not clearly defined but is described as the use of multiple instances of a security technology within a security layer [100].

While ICSes employ sensors usually in a controlled, small area, sensors can also be spread over a wider area using mesh routing. This is called a Wireless Sensor Network (WSN).

2.1.3 (Mesh-)Networks: Wireless Ad-Hoc Networks and Wireless Sensor Networks

Mesh networks are highly interconnected networks where every node has routing capabilities. These networks can have different layouts, e.g. there are hierarchical WSNs and distributed WSNs [25] (see Fig. 2.2). Hierarchical ones usually rely on a base station to collect the sensor data from multiple clusters and forward it to the data sink. Each of the clusters is managed by a cluster head. Distributed WSNs on the other hand have no fixed structure.

Also, the communication varies depending on the network. There is pair-wise com-

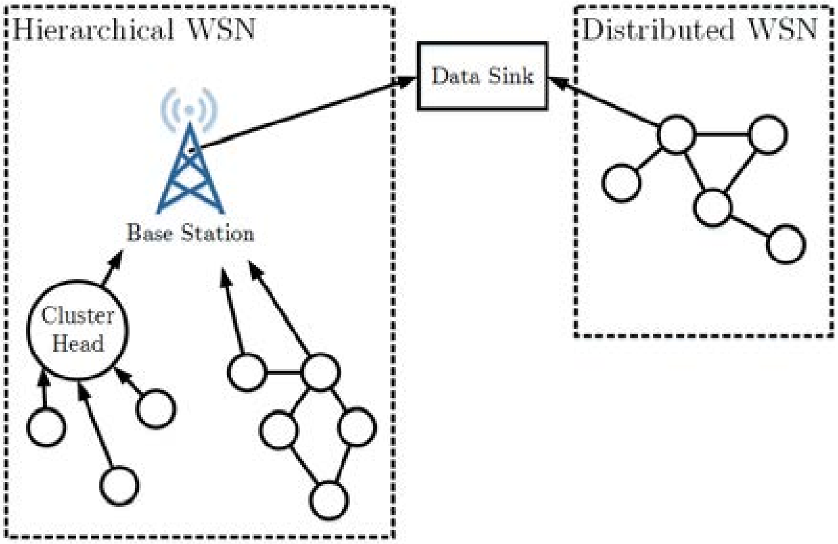


Figure 2.2: Hierarchical and distributed Wireless Sensor Network: Reproduction of Camtepe et al. [25]

munication, multicast within clusters, or broadcast communication. If mesh networks spontaneously change their geographic layout, they are called Wireless Ad-Hoc Networks (WANETs). The terms can also overlap, so a WSN could also be a WANET or could use a different routing technique than mesh routing. This work focuses on mesh networks.

Since these networks are only loosely connected and nodes are rather insecure – also due to their potential physical exposure to attackers – these networks are vulnerable to attacks like blackhole, routing loop, wormhole, node capture, clone and further attacks. In WSNs the nodes often have limited power resources making deployment of costly cryptographic solutions difficult. Key distribution and revocation becomes a major issue as systems need to be built keeping in mind that an adversary potentially compromises several nodes during operation [72, 105]. The key distribution concepts range from probabilistic over hybrid to deterministic approaches [25]. However, they often rely on pre-shared keys or a master key.

To recover from attacks, mesh networks, especially WSNs, utilised self-healing techniques which are often induced by the base station.

2.1.4 Self-healing Techniques

Self-healing approaches are techniques where a system can recover to a good state after it detected an invalid state. Ghosh et al. [61] classified self-healing approaches into three fields: (1) maintaining system health, (2) discovery of non-self or system failure, and (3) system recovery. Redundancy, probing, diversity, and log analysis are used to maintain system health. Sometimes the systems are also formally modelled. Failures are discovered according to the utilised maintenance mechanism; this can be by e.g. detecting a missing component or a missing response. Similarly, recovery takes place corresponding with the initially used maintenance technique. If there is redundancy, another system could take over. Otherwise, the faulty component is contained or the software state is recovered.

Self-healing was especially proposed for critical systems like ICSes to achieve intrusion tolerance (as already mentioned above). Furthermore, self-healing is used in WSNs in particular.

Di Pietro et al. [41] proposed a proactive self-healing technique for WSNs where the sink, i.e. the system collecting the data from the nodes, re-initialises nodes with a new cryptographic key or secret seed. This randomness is also forwarded to a few neighbouring nodes in order to update key material. The authors later improved the technique with a moving target defence system [42] by moving and encrypting the data. Symmetric keys additionally evolve with the help of the sink. While the sink represents a Trusted Third Party (TTP), this approach demonstrates how self-healing can be combined with other techniques, in this case moving target defence.

Self-healing was also deployed to recover a single machine instead of a network. Grizzard [65] developed a host-based self-healing technique. A virtual machine monitor observes the system and can recover to a known good state. Abnormal states are detected with a machine learning algorithm which is based on the control flow graph and can identify deviations from good states.

2.1.5 Trusted Execution Environments and Trusted Path

In order to provide a starting point for secure architectures, special software and hardware features were necessary. They are summarised under the term Trusted Computing (TC).

Trusted Execution Environments (TEEs) and trusted input/output or trusted path are the building blocks of TC. Vasudevan et al. [134] defined five properties such secure systems should fulfil: isolated execution, secure storage, remote attestation, secure provisioning, and a trusted path. Rijswijk and Deij [132] suggested adding local attestation to the user as sixth property to the five properties of Vasudevan et al.

TC is also seen as a countermeasure for APT attacks. Virvilis et al. [140] confirmed this by analysing five APT attacks – Stuxnet, Duqu, Flame, Red October, and MiniDuke – for their common criteria and concluded that TC, patching, network segregation, whitelisting against Command and Control servers, and filtering dynamic content execution can defend against these attacks.

TEE techniques can be categorised into three sub-areas: hardware-based, software-based, and integrated hardware. The following section introduces them in this order. Trusted path is omitted because Chapter 4 evaluates and compares various trusted path techniques in more detail.

2.1.5.1 Hardware-based Trusted Execution Environments

Probably the most popular hardware-based TEEs are used in online banking. Hardware tokens with PIN pad e.g. by Vasco [133] or USB based tokens like the Seal One [114] are commonly used by banks for two factor authentication. These tokens exist with various configurations and interfaces e.g. as USB sticks for two factor authentication, encryption, and signing [62]. There are also SD cards to enhance smartphones [63] and general purpose smart-cards with NFC or Bluetooth [131].

The major difference between the banks is hereby whether these tokens are used to authenticate the entity (verifying each recipient) or the transaction (verifying each bank transfer) [30].

While these techniques are feasible, there are already attacks against two factor authentication with so-called Man-in-the-Browser (MITB) attacks [2]. Trojans like Zeus or Zeus in the Mobile (ZitMo) [78] compromise online banking and simultaneously hide their malicious actions from the victim by displaying correct results.

Additionally to commercial hardware solutions, there is research on new secure hardware architectures. One of the first architectures was AEGIS [124], a single-chip processor architecture which verifies and encrypts off-chip memory and has a secure context manager to manage processes if the operating system is untrusted.

Lee et al. [87] designed a microprocessor architecture to protect sensitive data. For this, they defined new processor features and an implementation of a trusted path but relied on two additional buttons and LEDs. The hardware is trusted. Champagne and Lee [28] then developed Bastion which shields a hypervisor off from software and physical attacks while achieving better performance than integrated TEEs (like the TPM). The authors achieved this by adding secure hardware registers for hypervisor hashes and keys, a memory encryption module, a new type of hypervisor calls on Translation Lookaside Buffer (TLB) misses, and two hardware cryptographic engines between cache and memory. Later, Szefer and Lee [126] also created a Virtual Machine (VM) environment, called HyperWall, to protect guest operating systems from the hypervisor. The technique relies on modifications to the microprocessor and Memory Management Unit (MMU). While the hypervisor can manage VMs (i.e. start, pause, stop, change memory assignment), the hardware protects confidentiality and integrity of the memory of the guest VMs.

Sancus [98, 99] by Noorman et al. is aimed at Internet of Things (IoT) devices and provides software isolation, remote attestation, secure communication, secure linking, and confidential software deployment. Their approach relies on key management and memory access control of two additional units in the CPU. Start of execution is only possible at certain entry points and text sections are only readable during their execution. The authors implemented their architecture as zero-software Trusted Computing Base (TCB) in a Field Programmable Gate Array (FPGA).

2.1.5.2 Software-based Trusted Execution Environments

For software-based TEEs there are two approaches, either security guarantees are given by a compiler or by virtualisation.

TEEs through compilers were shaped by the group around Agten, Strackx, Jacobs, Patrignani, and Piessens. They developed a compiler which protects memory accesses by fine-grained control of the program counter. It restricts user-level attacks to public method calls of the programming language [3]. The authors improved their method and produced Fides [123], a hypervisor approach to defend against kernel-level attacks. They coined the term *fully abstract compilation* where compilation preserves the security properties of the high level language also in the low level language. The goal hereby is to restrict an attacker injecting assembly code at kernel-level to the legitimate interface. Fides achieves this through binding access rights to the program counter, only allowing to start software modules at valid entry points, and cryptographically hashing public sections of the memory. The authors achieved full object orientation by adding dynamic memory allocation, exceptions, inheritance, and inner classes [103]. They also gave a formal proof of fully abstract compilation [104].

The need for TC created virtualisation- and hypervisor based techniques as they were available early on. Terra [59] manages multiple, isolated VMs with a trusted virtual machine monitor to achieve TC. Later, AMD and Intel developed hardware support for virtualisation.

2.1.5.3 Trusted Execution Environments based on Integrated Hardware

Virtualisation was quickly overtaken by approaches based on integrated hardware. There are three special systems at the moment: the TPM chip, ARM TrustZone, and Intel SGX. They are introduced here briefly, a more detailed overview is given by Murdoch [96].

Trusted Platform Module (TPM): The first integrated TEE was the TPM chip, an additional, secure chip in the computer providing cryptographic functionality and limited secure storage. Sailer et al. [110] practically demonstrated how to use the TPM to verify the integrity of executable code starting at the Basic Input/Output System (BIOS). They implemented an integrity measurement system for Linux using the TPM which can detect rootkits.

Kauer [81] removed BIOS and bootloaders from the trust chain, so-called Dynamic Root of Trust for Measurement (DRTM), to minimise the TCB. He implemented the *OSLO* architecture on top of AMD Secure Virtual Machine (SVM).

The group around McCune and Perrig very actively researched TEE and the TPM. Their first approach, Flicker [91], used a minimal secure environment which (1) temporarily suspended the operating system, (2) switched ("flicked") to the TEE, (3) executed the security sensitive code, and (4) resumed the operating system. Their implementation was based on AMD SVM and the TPM.

Together with Parno et al. [102], McCune used Flicker to protect the state of a software module. They leveraged the Non-Volatile Random Access Memory (NVRAM) of a TPM to store the state history of the module in order to replay the last changes after a crash.

Due to the performance overhead of Flicker, McCune et al. developed TrustVisor [93], a special hypervisor based on a DRTM. The architecture relies on software based TPMs called microTPMs for performance and a hardware TPM to provide security. They demonstrated an application of TrustVisor with a sandbox that protects host operating system as well as the guest-application [88].

Vasudevan et al. generalised together with McCune the idea and development of a secure hypervisor. They proposed *DRIVE* [135], a methodology for designing hypervisors for integrity verification. Such hypervisors require six properties: (1) modularity and (2) atomicity of initialisation and interrupt handlers, (3) memory access control protection, (4) correct initialisation, (5) proper mediation – i.e. memory access protection is active whenever attacker-controlled programs execute – and (6) safe state updates. The authors extended TrustVisor to a proof-of-concept *DRIVE* hypervisor and verified it with a software model checker.

ARM TrustZone: TrustZone is a security feature of modern ARM CPUs which splits the architecture into two "worlds": the normal, unprivileged, world and the secure world. While the secure world has access to the entire architecture, the normal world can be restricted.

Since TrustZone was developed after the TPM chip, Winter [143] attempted to port the approach of Trusted Computing Group (TCG) (who developed the TPM) to TrustZone. He used a hypervisor in order to manage multiple isolated trusted engines, as required by the TCG, with only the two worlds of TrustZone. The main obstacle was that TrustZone does not require a secure bootloader. While some vendors provide this, it is an additional feature and is not present on all platforms. However, Winter experimented with the FriendlyARM Mini6410 [144] and analysed its security and the secure bootloader including one-time writable registers. Together with Fitzek et al. [54], Winter created a monolithic operating system running in the secure world of TrustZone while a Linux-based operating system runs in parallel in the normal world. They proposed to use this kind of architecture to send data securely from the ICS to the manufacturer.

Also the group around Vasudevan (McCune, Perrig) et al. [134] evaluated TrustZone. As mentioned earlier, they defined five properties TEEs should provide (isolated execution, secure storage, remote attestation, secure provisioning, and trusted path). They identified a few inaccuracies in the TrustZone architecture: To provide security, all components of TrustZone have to be present but most off-the-shelf mobile devices do not include all of them. In some devices the secure world is turned off completely. Also, manufacturers and carriers do not consider Direct Memory Access (DMA) attacks leaving this attack vector open. Lastly, due to the cost of ARM tools, the open-source community did not get involved in it, preventing wide-spread adoption.

One of the few practical examples for TrustZone is a privacy preserving payment framework of Pirker et al. [106]. They combined NFC with TrustZone on smart-phones based on prepaid credits. Normal world apps would request payment from the anonymous payment token in the secure world. After the payment protocol, the secure world module receives a bar-code with a proof of payment from the service provider.

Intel Software Guard Extensions (SGX): Towards the end of this work, the first Intel SGX-enabled CPUs were released. SGX provides secure remote computation, i.e. executing software on a system controlled by an untrusted party. Protected software containers, so-called enclaves, shield software off from operating system and hypervisors. For this, SGX stores the enclave data in the Enclave Page Cache (EPC). The state is hashed and protected by encryption when leaving the TEE. I refer the interested reader to Costan et al. [32], who analysed SGX in depth.

As SGX is fairly new, there are only few research studies. The main source is research by Intel. They demonstrated isolated execution [94], remote attestation [7], and an example application [69] providing one-time passwords, distribution of sensitive documents, and secure video conferencing.

On the academic side, Brenner et al. [22] ported Apache ZooKeeper, a cloud coordination service, to SGX to protect user-data. They intercepted the message processing pipeline of ZooKeeper and redirected it to an enclave. Additional enclave encryption helps keeping the data secure. Together with Arnautov et al., a few of the authors continued to work on SGX and created *SCONE* [13], a secure Linux container management for Docker. Containers are protected from the environment by partly moving system calls into the corresponding enclave. Their implementation also support asynchronous system calls via a request queue and a response queue. A special operating system thread inside SCONE processes these system calls and returns the results in the response queue.

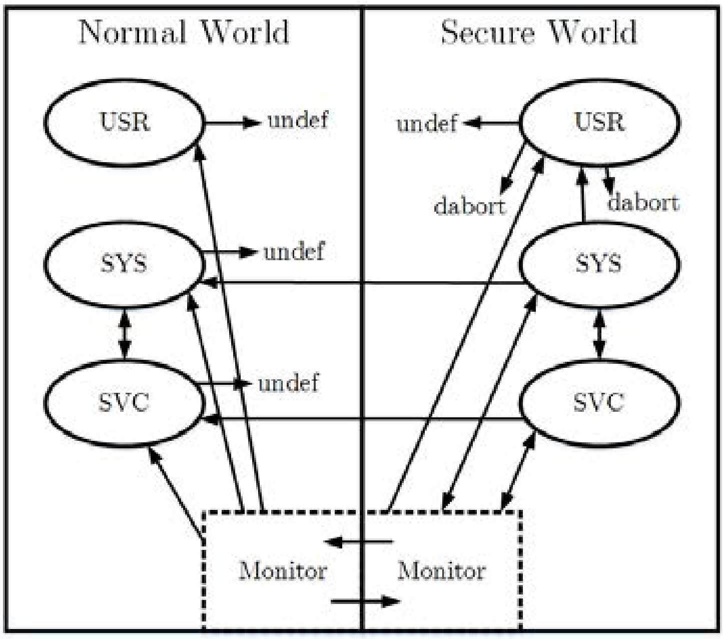


Figure 2.3: ARM mode switches: *undef* stands for the undefined instruction exception and *dabort* stands for a data abort.

2.2 Overview of ARM TrustZone

The following section introduces ARM TrustZone in more detail. I collected this information mainly from the official documentation [12, 10, 9] and from my own experiments with the ARM Cortex-A8 of a FreeScale i.MX53 Quick Start Board. Some information is taken from Genode operating system [51, 50, 60]. Section 5.2.3 presents a small performance analysis I conducted on ARM TrustZone.

As already mentioned, ARM TrustZone is a TEE consisting of two separated environments: the *secure world* and the *normal world*. While the secure world has full access to the system, the normal world can be restricted in its capabilities. The switch between the two worlds is handled by the so-called *monitor*. TrustZone chips usually come with the TrustZone Interrupt Controller (TZIC) and functionality to manage memory, i.e. a TrustZone-aware MMU, routines to forbid DMA, and so on. However, the official documentation does not specify a secure boot mechanism. This, if existent, is delivered by the particular chip vendor.

ARM TrustZone is internally realised via an additional 33rd bit, the Non-Secure (NS) bit, propagated over the system bus. This bit defines in which world (normal world or secure world) the processor is currently running. Additionally to the world, there are different modes in which the processor can be. The main modes are user mode (USR), system mode (SYS), supervisor mode (SVC), and monitor mode (MON). Since worlds and modes are separated from each other, only certain mode switches are allowed (see Fig. 2.3). All other mode and world switches have to use a Software Interrupt (SWI) to switch the mode or a Secure Monitor Call (SMC) to access the monitor and switch world/mode.

In total, there are eight exceptions:

1. Reset (indicates system boot)
2. Fast Interrupt Request (FIQ)
3. Interrupt (IRQ)
4. External Data Abort
5. External Prefetch Abort
6. Undefined Instruction
7. Software Interrupt (SWI)
8. Secure Monitor Call (SMC)

Some of these exceptions can be configured to trap into monitor mode and, thus, the world in which they are handled can be set [9, 51].

The concept of the interrupts FIQ, IRQ, SWI, and SMC are specific for ARM processors. Hereby, SWI and SMC realise mode and world switches while the others are ordinary interrupts. FIQs are fast interrupts and have priority over the normal IRQ. Furthermore, FIQs can be configured to trap into the monitor and, thus, the secure world.

The last ARM specific concept are co-processors. In order to access functionality external to the CPU (e.g. the MMU), there are 16 co-processors. The system control co-processor *CP15* is used to configure TrustZone and it also stores the Secure Configuration Register (SCR) allowing the secure world in privileged mode to access[1] the NS bit. If the

[1] More information is listed at the ARM documentation [12] under the MRC/MCR instruction (move ARM register to co-processor and vice versa).

normal world or user mode tries to access the bit (even read-only access), an undefined exception happens. That means, the normal world has no means to discover in which world the processor is running or if the processor is even able to run TrustZone. To the normal world it always looks like the CPU does not support TrustZone.

An ARM CPU always boots in secure world to enable setting up the stack of all modes, Block Started by Symbol (BSS) segment, TrustZone monitor, SMCs, FIQs, IRQs, and TZIC.

CHAPTER 3

THE MALWARE TOLERANCE CONCEPT

3.1 Research Questions

At the beginning of this research the following questions were raised:

1. Can a malicious system be used securely?
2. In what circumstances, and for what purposes, might it be possible to securely use a platform which is suspected to have malware?
3. Is it possible to split trust upon multiple components so that none of them alone can compromise the system? This includes – depending on the underlying system – protecting the confidentiality of the secret(s) as well as preserving the integrity of data, resources, and/or environment (e.g. cyber-physical systems).
4. How can the tolerance of a system towards malware be evaluated?

3.2 Definition and Example

The basic idea of malware tolerance is to distribute trust upon several, independent devices interacting in such a way that the individual device cannot meaningfully tamper with the data or resources. Thus, the method tolerates devices which are compromised by

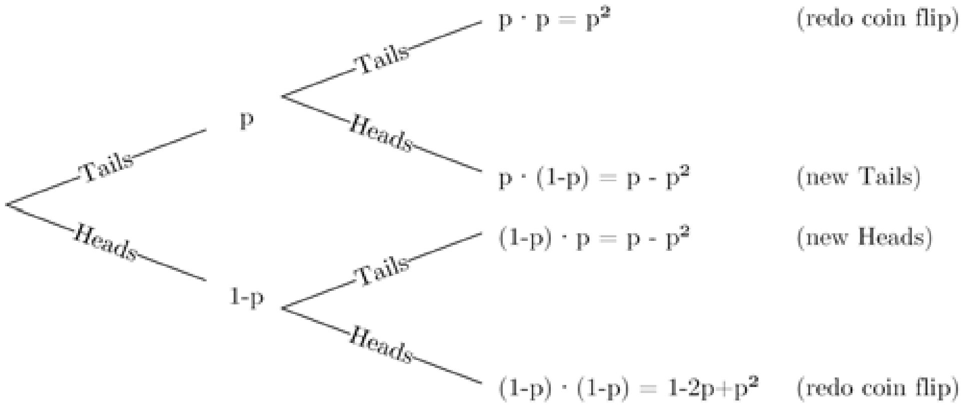


Figure 3.1: Probabilities of two coin flips

malware. No part of the architecture is assumed invulnerable to attacks. An adversary would have to compromise multiple devices to successfully attack the system. This is significantly more difficult than manipulating a single device.

Let us start with an example: Suppose we flip a coin but we suspect that the coin was manipulated and is biased. We cannot find out the exact bias of the coin but the coin flip should be perfectly fair, i.e. *heads* should be as likely as *tails*.

While this first sounds impossible, the solution is simply to throw the coin twice. We define heads followed by tails as the new heads and tails followed by heads as the new tails. In all other cases, one starts again.

If the coin is biased and lands with probability $1 - p$ on heads and with probability p on tails, then both new cases have the same likelihood to occur. The probability for heads followed by tails is $(1 - p) * p = p - p^2$ and tails followed by heads is $p * (1 - p) = p - p^2$. Independently from p both cases have the same probability. Figure 3.1 shows the probabilities as a tree diagram.

Essentially in this little experiment, the adversary was controlling the hardware (the coin) while we wrote the software (the coin flip). Considering that a computer is dealing with zeros and ones (heads or tails), this research aims to transfer the principle to computers.

3.3 Classification and Differentiation

The NIST [97] divides security incidence management into four areas: identify, protect/prevent, detect, and respond/recover. Malware tolerance draws ideas from detection by using IDS techniques, prevention with TC (i.e. TPM, SGX, ARM TrustZone), and recovery by employing so-called self-* approaches, especially self-healing. I would like to explain differences and common terms between these research fields and malware tolerance.

Early intrusion tolerance as seen in IDS and Intrusion Prevention System (IPS) developed from fault tolerance and is, thus, focusing on detecting and preventing attacks. It is, contrary to malware tolerance, not allowing attacks. The early versions of it simply restarted the IDS on faults. While this helps against ordinary faults, it is certainly not secure against malware on the IDS itself.

Prevention (e.g. from IPSes) and resilience are terms I would describe with malware resistance. Instead of being tolerated, malware is defended against or resisted. Malware tolerance is resilience taken to the maximum and is better described with *absorbing potential* or *recovery potential*. This expresses that malware incidences are allowed to happen but are not necessarily harmful.

The latest re-definition of intrusion tolerance (in contrast to the early intrusion tolerance of IDSes) overlaps with malware tolerance substantially. It was notably shaped by Verissimo and his research group. To accurately compare both techniques I present the exact definition of Verissimo et al. [137]:

> "A new approach has slowly emerged during the past decade, and gained impressive momentum recently: Intrusion Tolerance (IT). That is, the notion of handling – react, counteract, recover, mask – a wide set of faults encompassing intentional and malicious faults (we may collectively call them intrusions), which may lead to failure of the system security properties if nothing is done to counter their effect on the system state. In short, instead of trying to prevent every single intrusion, these are allowed, but tolerated: The system has the means to trigger mechanisms that prevent the intrusion from generating a system failure."

Malware tolerance is closely related to intrusion tolerance and a sub-category thereof.

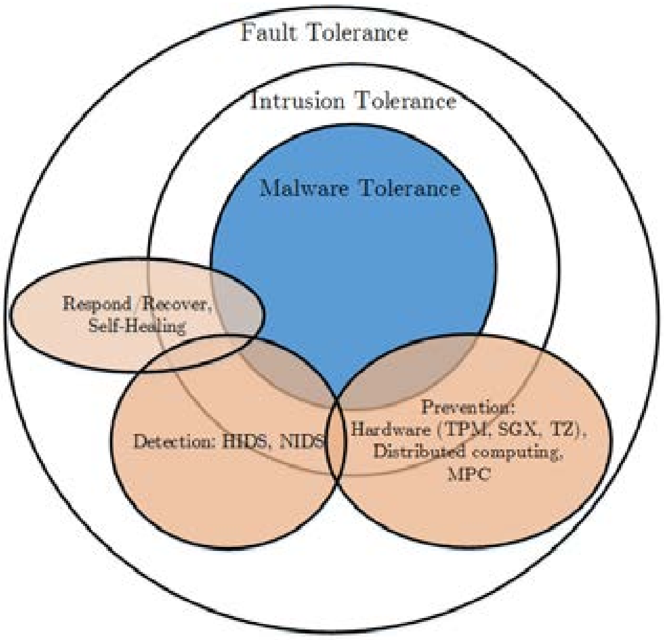

Fault Tolerance
Intrusion Tolerance
Malware Tolerance
Respond/Recover, Self-Healing
Detection: HIDS, NIDS
Prevention: Hardware (TPM, SGX, TZ), Distributed computing, MPC

Figure 3.2: Classification of malware tolerance: The circles in the diagram indicate sets. Malware tolerance is a sub-set of intrusion tolerance using ideas of several related topics, e.g. self-healing, IDSes, and trusted hardware.

It specifies more how to achieve such tolerance – namely by removing every single point-of-failure where necessary. I propose that recovery mechanisms, if present, should happen either always or automatically and never manually. The goal is to prevent a single intrusion of one component from generating a complete system failure. Malware tolerance is intrusion tolerance by distributing trust over several components and automatic recovery if possible. It provides security through replication of the core components and a form of decision making or voting, thus, avoiding single points-of-failure.

An overview how malware tolerance is aligned to other research areas is displayed in Fig. 3.2. Malware tolerance is a sub-category of intrusion tolerance and fault tolerance and uses techniques of self-healing, IDSes, trusted hardware, and distributed computing. The circles in the figure stand for sets of techniques. For example, an IDS is part of the detection circle while an IPS belongs to the intersection between detection and prevention, it first detects threats and then blocks (prevents) them. This stops attacks before they take effect. Self-healing, on the other hand, recovers from faults or intrusions after these happened. Therefore, the two circles for self-healing and prevention do not overlap.

3.4 Models

3.4.1 Attacker Model

The basis of attacker models is usually the Dolev-Yao attacker who is situated on the network and can intercept, change, and redirect every message sent on this network. In particular, the adversary can read and manipulate plaintext messages. However, the Dolev-Yao attacker alone is insufficient for certain types of attacks [101], e.g. node capture attacks in a WSN.

Malware tolerance operates in a post-compromise scenario where the attacker already gained access to the secrets of a device. Therefore, the Dolev-Yao attacker by itself is inadequate for malware tolerance and we extend the model with further capabilities. The initial goal is to be malware tolerant towards one compromised device, i.e. we assume the attacker gained a first foothold in the architecture.

1. This work assumes a Dolev-Yao attacker [46] on the network who interacts with software-side technologies. The attacker has no physical access to the facilities and cannot change cabling or remotely introduce electrical signals directly into wires (apart from Assumption 2).

2. Additionally, the attacker can choose one[1] device of which he gains full control – i.e. also access to corresponding cryptographic keys and "software" access to the physical wires connected to the chosen device. The attacker can manipulate the hardware of a chosen device once (during production) but has no physical access afterwards any more. If the chosen device has network access, the attacker can update software and firmware.

3. We are initially not aware which device the attacker chose.

[1] This thesis only shows the basic case of an attacker compromising one device. Tolerating attacks on multiple devices is more challenging but similarly possible.

3.4.2 General Assumptions

Furthermore, the following assumptions are made:

1. Multiple components only offer additional protection if they are not controlled by the same party. Thus, I stipulate, that for any malware-tolerant technique the devices are manufactured by different vendors, so-called heterogeneity. If diversity of suppliers is not granted, the manufacturer controls the entire architecture and compromising this one manufacturer would break all security measurements.
2. Every computational device is able to execute cryptography (especially asymmetric cryptography) and attacks on cryptography and phishing attacks are out of scope.
3. The user or operator of the infrastructure is a trusted entity and is allowed to access and change secrets, data, and configuration.

The proposed techniques of the consecutive chapters additionally assume the following.

Chapter 4 (trusted input):

5. Keyboard and smart-card are assumed to have no other communication channels apart from the ones to the computer (e.g. USB). This could be verified by inspecting the devices. Note that this does not forbid a separate or built-in hardware keylogger.
6. The recipient is trusted since he should see the plaintext data.

Chapter 5 (malware-tolerant Industrial Control System):

5. The 2-out-of-3 circuit is hardware-only and in scope of verification.
6. Furthermore, while this work only speaks of one actuator, no critical infrastructure would only rely on one such component. In practice, multiple actuators exist.
7. PLCs and sensors work synchronously or are buffered.

Chapter 6 (malware-tolerant mesh network):

5. We assume the network has a high connectivity and it is not partitioned. Low connectivity enables the adversary to compromise a device that is situated at interconnections enabling him to block messages completely. As light-bulbs are normally in every room of a flat or house, this assumption is justifiable in a smart-home setting with smart light-bulbs (and further smart devices).

3.4.3 Multiple Trusted Computing Bases Model

In order to reason about malware-tolerant systems, this research analyses approaches based on multiple Trusted Computing Bases (TCBs). A TCB is the minimum set of honest components needed to secure the system. Multiple redundant sets can exist. Devices are grouped into a TCB, if they can produce the desired result even when all other devices are compromised. A system is malware-tolerant if there are at least two disjoint TCBs that provide the same property. If one TCB is compromised, another honest TCB would give the correct result. Each independent component of the system can be part of multiple TCBs – i.e. mostly entire devices as e.g. the CPU depends on the computer and is not an independent component.

In contrast to the classical single TCB model, my model consists of several TCBs with flexible trust assumptions. If any 1 (of N) TCBs is secure, the system is secure. In other words, TCBs are *OR*'ed together. If TCB_1 is secure *OR* TCB_2 is secure then the system is secure. An adversary has to compromise all TCBs to be successful which is significantly harder than attacking a single TCB.

E.g. if device d_1 and d_2 are part of TCB_1 and device d_2 and d_3 are part of TCB_2, then the system is secure if $(d_1 \wedge d_2) \vee (d_2 \wedge d_3)$ is secure. As one can see here, d_2 is part of all TCBs and is, thus, the single point-of-failure. The system is not malware-tolerant because there are no disjoint TCBs (TCB_1 and TCB_2 overlap).

3.5 Brief Feasibility Study

Since malware tolerance first seems difficult (or even impossible) to achieve, this section briefly estimates to what extent it is indeed possible.

Intuitively, the upper bound of malicious parts to tolerate is at maximum $N - 1$ for N parts in total; one part has to be honest in order to detect or prevent attacks of the others, report to the user, recover, or serve as root-of-trust. Contrary to other approaches, malware tolerance does not stipulate which part this has to be and the TCB is flexible.

The $N - 1$ border is a rough estimation and unrealistic. It needs to be stated more precisely. To estimate it more exactly, I make use of two scenarios: voting and multi-party computation.

In voting, the majority of the voters determine the result. If an adversary controls more than 50% of the nodes, he is in charge of the outcome. Consequently, no technique can prevent this kind of attack when allowing nodes to be malicious – which malware tolerance deliberately allows. Through this (negative) example, we know that malware tolerance *in the general case* can only permit $< 50\%$ malicious parts.

When a set of entities collectively calculates a function securely over given input values, one speaks of multi-party computation. A real-world application was shown by Bogetoft et al. [17] They developed a multi-party protocol for sugar beet farmers in Denmark to agree on a fair price with the only company to process sugar beets in Denmark. The protocol hides the submitted prices and protects the farmers from the monopoly of the company.

The upper bound of passive attackers a_p such techniques can tolerate is $a_p < 50\%$ (we have already seen this above). For active attackers a_a, it is $a_a < 33\%$, i.e. it is only possible to prevent attacks generally for less than a third of the architecture being malicious.

To understand the 33% border, the idea behind the proof as given by Damgard [36] is reproduced here: Let us assume a three party protocol with A, B, and C. All parties should output a bit $b \in [0, 1]$. A is in possession of b and broadcasts it to B and C. B asks C for the value. But, an honest B could not decide who is lying if the values of A and C do not match. Thus, B can only detect an attack but cannot prevent it. The interested reader is referred to Cramer et al. [34] for more details about multi-party computation and the proofs. This example is close to our scenario in Chapter 4 where a user (A) has a message b and types it into a keyboard (B) which forwards it to a smart-card (C).

The formalisation of multi-party computation also matches the empirical result of fault tolerance (as seen in Chapter 2) where $2 * f + 1$ replicated servers can detect failures of a fraction f ($< 50\%$ malicious) of the replicas and tolerate attacks for $3 * f + 1$ servers ($< 33\%$ malicious).

While this is only an estimation, it implies that *general purpose* malware tolerance cannot tolerate more than 33% malicious components. Only in certain cases we can achieve better results than this.

With this knowledge about the theoretical concepts, this thesis demonstrates the practical application of malware tolerance in three scenarios: trusted input, ICSes, and mesh networks in particular smart-home networks. Although there are other examples like online banking and electronic voting, I consider the above mentioned scenarios more promising from a research perspective because common solutions, like two factor authentication or an anti-virus, are mostly not applicable.

3.6 Methodology

3.6.1 ProVerif Proofs

To prove the security guarantees of the approaches following in the next chapters, this thesis utilises ProVerif[1], a state-of-the-art protocol verifier. I also released the proof scripts as open-source.

In ProVerif, protocols are represented with *messages* which are sent over public or private *channels* between *processes*. During a protocol execution user-defined *events* can happen. Predicates, so-called *queries*, can be formulated and ProVerif tries to verify them. To do so, ProVerif has a built-in *attacker* which is able to participate on public channels. Internally, ProVerif represents protocols through Horn clauses enabling formal verification.

Figure 3.3 shows a ProVerif script of a simple protocol where A sends a secret message m over a public channel ch to B. The queries test if the attacker gets m (confidentiality) and if messages B received are from A (authentication). Obviously, this protocol is insecure and ProVerif correctly returns that the attacker can compromise authentication and confidentiality (see Fig. 3.4).

[1] `http://proverif.inria.fr`

```
(* public channel *)
free ch: channel.
(* secret message *)
free m: bitstring [private].
(* events *)
event A_sent(bitstring).
event B_received(bitstring).

(* queries *)
(* does the attacker get m? *)
query attacker(m).
(* if B received any message x, does that imply that A sent it? *)
query x:bitstring; event (B_received(x)) ==> event (A_sent(x)).

(* processes *)
let A =
    event A_sent(m);
    out(ch, m). (* send m on channel ch *)
let B =
    in(ch, m: bitstring); (* receive m on channel ch *)
    event B_received(m).
(* start *)
process A | B
```

Figure 3.3: ProVerif example

```
(...)
RESULT event(B_received(x)) ==> event(A_sent(x)) is false.
(...)
RESULT not attacker(m[]) is false.
```

Figure 3.4: ProVerif output

3.6.2 Cryptographic Notation

To improve readability, cryptographic algorithms are abbreviated in the following chapters. $E(k, m)$ refers to an authenticated symmetric encryption of message m with key k, e.g. AES-GCM. $D(k, c)$ is a decryption and $m_i|...|m_0$ represents a concatenation. $E(k^{pub}, m)$ denotes an asymmetric encryption, $S(k^{priv}, m)$ is a digital signature of m, and $H(k, m)$ is a Message Authentication Code (MAC). $RSA(param, k^{pub}, m)$ stands for the common Rivest-Shamir-Adleman Cryptosystem (RSA) cryptosystem. The parameters are displayed deliberately here, because they are needed in Chapter 4. All cryptographic algorithms are used in their appropriate form, i.e. with initialisation vector, nonce, padding, etc. but this is omitted for brevity. Chapter 6 introduces the Intrusion Resilient Signature (IRS) scheme which is abbreviated with $IRS(k^{priv}, m)$. Figure 3.1 shows an overview of the utilised notation.

Furthermore, the protocols are presented as message sequence charts. If a device has some secret keys, they are displayed over the device at the top of the diagram.

Abbreviation	Meaning
$E(k, m)$	Authenticated symmetric encryption of message m with key k
$D(k, c)$	Decryption and verification of ciphertext c with key k
$m_i\|...\|m_0$	Concatenation of message m_0 to m_i
$E(k^{pub}, m)$	Asymmetric encryption of message m with public key k^{pub}
$D(k^{priv}, c)$	Asymmetric decryption of ciphertext c with private key k^{priv}
$S(k^{priv}, m)$	Digital signature of m with private key k^{priv}
$H(k, m)$	Keyed hash of m with key k
$RSA(param, k^{pub}, m)$	RSA encryption with parameters $param$ of message m with public key k^{pub}
$IRS(k^{priv}, m)$	IRS signature of message m with private key k^{priv}

Table 3.1: Cryptographic notation

CHAPTER 4

USER INPUT: SMART-GUARD

The first step of this thesis is to protect the user input. A security system (e.g. an Intel SGX enclave) can only trust inputs if it can distinguish malware input from user input.

In the literature, this field is known as *trusted input*. It is defined as the problem of securing user input from device end-point (e.g. a keyboard) to program end-point. *Trusted output* specifies the stream in the other direction (program to device). Both terms are also subsumed by the expression *trusted path* or *trusted I/O* and belong to the wider domain of *trusted execution* [69, 145].

Trusted input techniques are already utilised to harden the infrastructure in fields such as online banking and electronic voting but are also applicable to Virtual Private Networks (VPNs) or commands to a server (e.g. via SSH) or to an ICS. Authentication of the exact origin of the input is essential in all of these scenarios as the following thoughts demonstrate:

- Recent online banking trojans (ZitMo [78]) spread from PCs to smart-phones to compromise two-factor authentication. Afterwards they impersonate the banking customer.

- Malicious or bogus commands to ICSes can damage these systems (see e.g. Stuxnet [85, 146]) and potentially lead to catastrophes especially since ICSes are used in the energy, transport, and health sector.

- Keyloggers are able to intercept login credentials and hijack connections [140].

- After compromising an administrator account in a company network, an adversary can pretend to be the administrator and gain widespread access to assets and resources [73].

- Forged, i.e. wrongly authenticated, votes in electronic voting compromise the entire voting system [49].

To improve on these scenarios the input source needs to be exactly identified, i.e. we have to be able to distinguish between authorised users and a potentially compromised PC.

Current approaches [29, 53, 90, 92, 145, 108] (detailed explanation in Section 4.4) for trusted path usually omit hardware attacks which means firmware attacks, keyloggers and similar are still effective. Moreover, they make strong and inflexible assumptions about which parts of the system are trustworthy. For critical resources like a power plant or electronic votes, we need stronger security guarantees and cannot purely rely on the trustworthiness of a single component.

Contributions:

- This chapter introduces Smart-Guard, a technique to secure input even if malware controls some parts of the system in real time. Smart-Guard distributes trust over several components of the system, and guarantees integrity and authentication when one out of three components is controlled by the attacker (see Section 4.1). It can also ensure confidentiality under certain conditions. No single component is required to be invulnerable to attack; an attack on one component can be resisted if other components are trustworthy. This means Smart-Guard does not rely on a single trusted computing base, but allows flexibility about the trust assumptions.

- A formal proof[1] of the security claims for Smart-Guard is provided using the state-of-the-art protocol verification tool ProVerif (Section 4.3.1 gives details).

[1] https://github.com/mdenzel/smartguard

4.1 Overview of Smart-Guard

Smart-Guard is a trusted input system consisting of a computer, a smart-card, and an encryption-capable keyboard[1], i.e. a keyboard with built-in encryption module which encrypts typed data. Keyboard and smart-card (smart-card reader) are both connected to the computer via USB or similar.

One could argue that an encryption-capable keyboard could simply encrypt or sign keystrokes by itself, but this would only shift the trust from the operating system driver to the keyboard. Malware tolerance aims to achieve stronger assurances and, thus, trust is distributed upon multiple devices in order to tolerate localised attacks. An adversary would have to compromise more than one device to be successful. In particular, Smart-Guard can resist (confidentiality and integrity) an infected computer, provided keyboard and smart-card are not compromised.

4.1.1 Basic Procedure of Smart-Guard

Consider a scenario where an authorised user wants to send a message or a command to a recipient from his or her commodity computer. The recipient could be the user's bank, an election server, or the flow control system of an oil pipeline.

The user types the characters of the message into the encryption-capable keyboard which sends them signed and encrypted to the smart-card to prevent the computer from tampering with the data. The smart-card verifies the keystrokes by displaying them via any form of confidentiality preserving output, like e.g. ARM TrustZone with a TrustZone-aware screen or an encrypted message to a phone. When typing is finished, the user confirms the displayed input by entering a short string.

Smart-card and keyboard each produce one partial signature of the message. Those partial signatures can only be combined into a valid signature if the two devices agree on the same input. The combined cryptographic signature is verified by the recipient.

[1] e.g. `https://www.nordicsemi.com/eng/Products/Bluetooth-Smart-Bluetooth-low-energy/nRFready-Desktop-2-Reference-Design`

4.1.2 Objective

The primary goal of Smart-Guard is to protect integrity and authentication of the user-given keystrokes. The recipient of the keystrokes should be able to verify the origin of the input. The system must at least guarantee these security properties if one of the three participating devices (user PC, smart-card, keyboard) is malicious. Smart-Guard also provides confidentiality under stricter conditions but focuses on authenticating the input.

4.2 Smart-Guard Protocol

The protocol consists of three phases:

1. An input phase which distributes user input to the devices. This step runs for every input character.
2. A transition phase marking the end of input.
3. A signature and encryption phase which only takes place once per message and can also run in the background.

The protocol phases are now introduced one by one. They are displayed in Fig. 4.1 and 4.2.

4.2.1 Setup

The encryption-capable keyboard and the smart-card share (1) a symmetric key k_{ks} which pairs the two devices. They could be directly connected by inserting the smart-card into a slot in the keyboard to establish k_{ks}. Both also receive (2) a partial key (respective k_1 and k_2) of a mediated RSA (mRSA) [19, 20] algorithm.

mRSA is an asymmetric cryptosystem and splits the private key into two shares $k_{priv} = k_1 + k_2$. This way, signatures can be created out of two partial signatures ps_1 and ps_2 (see Eq. 4.1). Note that the plaintext m has to be appropriately padded (details in [19, 20]).

$$signature = ps_1 * ps_2 = m^{k_1} * m^{k_2} = m^{k_1+k_2} = m^{k_{priv}} \ . \tag{4.1}$$

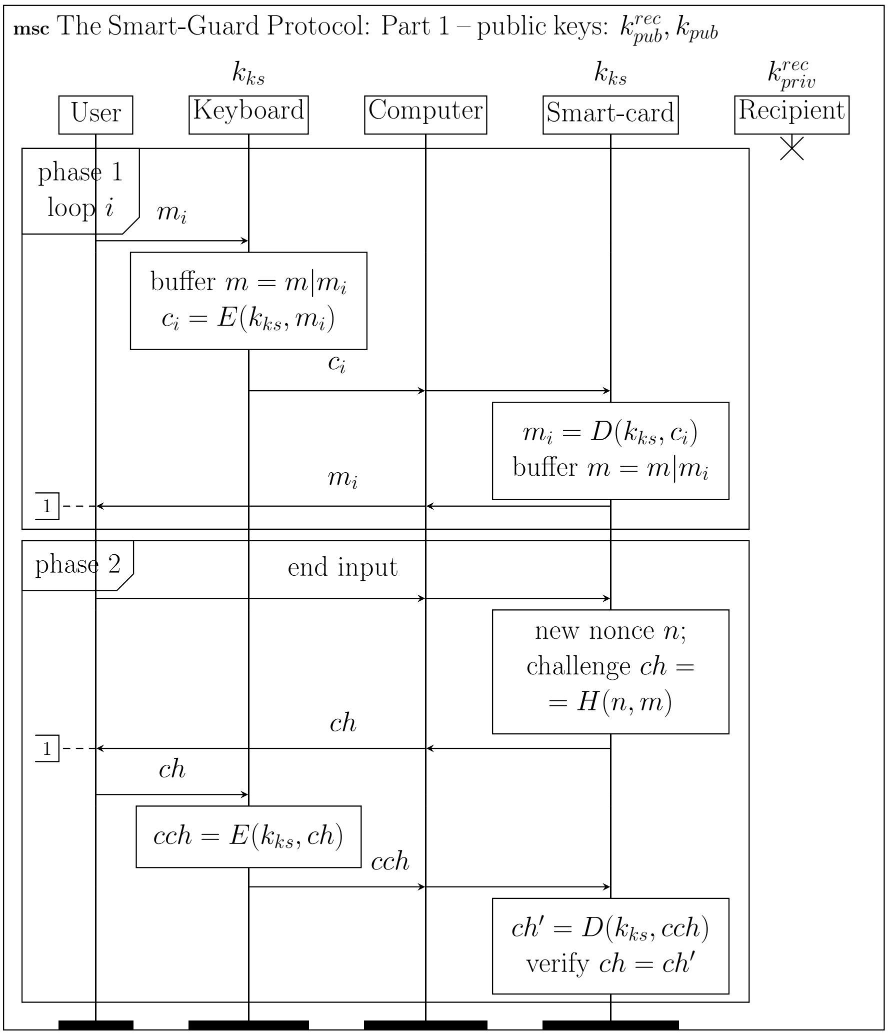


Figure 4.1: The Smart-Guard protocol – part 1: The user types in keystrokes m_i which are distributed to the encryption-capable keyboard and the smart-card. An end-input event (e.g. a mouse click) indicates that the user wants to proceed. The user verifies the keystrokes being displayed via secure output (label 1), by typing a challenge into the keyboard. Afterwards phase 3 (see Fig. 4.2) is run.

4.2.2 Phase 1

Phase 1 (see upper half of Fig. 4.1) distributes the input characters to keyboard and smart-card. Every character the user types into the keyboard is encrypted with the previously setup stream cipher (e.g. AES-GCM) and sent to the PC. The PC forwards them to the smart-card which decrypts them. Smart-card and keyboard buffer the characters for later signing. At this stage, only the keyboard verified the input. To allow the user to confirm the typed keystrokes, the smart-card displays them via confidential output to hide them from the computer. Confidential output can be provided, for example, by an ARM TrustZone enabled device or by sending them encrypted to a phone. Even if the secrecy of the *output* is compromised, the integrity of the *input* is unaffected (assuming no further colluding attacks).

4.2.3 Phase 2

After the user indicates that he wants to end the input (e.g. by a mouse click), phase 2 is run (see lower part of Fig. 4.1). The smart-card creates a fresh nonce, hashes the input with it, and displays the result via confidential output to the user who then types this verification string – estimately 4-8 characters – into the keyboard. The keyboard can only forward these characters via the PC to the smart-card as they appear random. This way the smart-card can independently verify that the user agrees with the keystrokes. A hash with a nonce has two advantages: (1) it cannot be forged by keyboard or PC and (2) it is clear to which input string it corresponds.

4.2.4 Phase 3

After keyboard and smart-card both received (phase 1) and verified (phase 2) the keystrokes, they can generate a shared signature (phase 3). This protocol is displayed in Fig. 4.2.

To create the shared signature, keyboard and smart-card must first establish a new key (k' in Fig. 4.2), that is verifiably fresh, in order to encrypt the message m with e.g. AES-GCM. Phase 3 generates this key k' with a Diffie-Hellman key exchange on the encrypted channel of pre-shared key k_{ks}. Since keyboard and smart-card contribute to Diffie-Hellman, both know that the resulting key is fresh.

Afterwards, the keyboard encrypts user input m with k' and encrypts (RSA) the key k' with the public key of the recipient k_{pub}^{rec} – similar to common e-mail encryption schemes. The keyboard also creates a partial mRSA signature ps_1. To enable the smart-card to verify the encryption (see also Section 4.2.5), the encryption parameters pr are encrypted

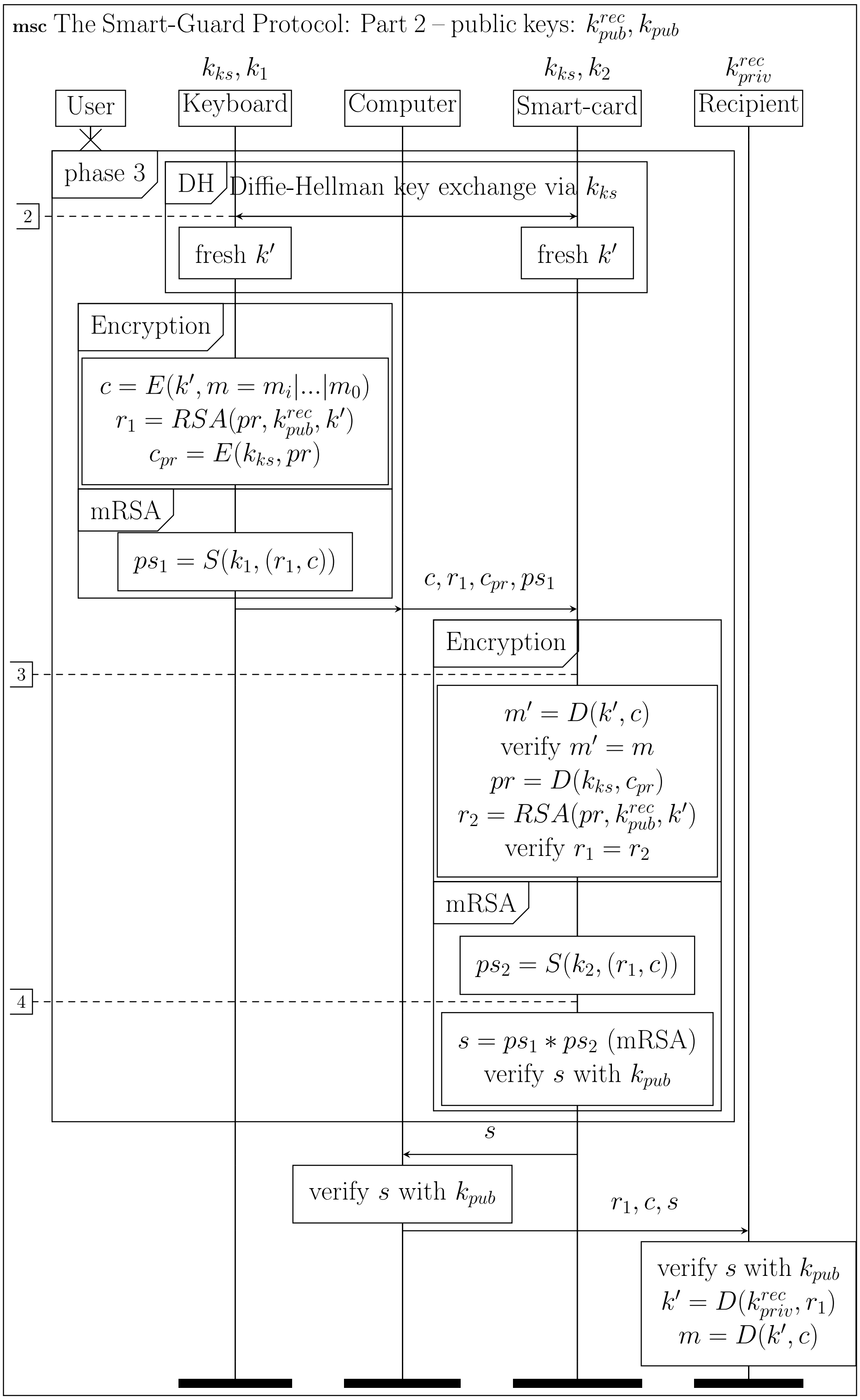

msc The Smart-Guard Protocol: Part 2 – public keys: k^{rec}_{pub}, k_{pub}
k_{ks}, k_1
k_{ks}, k_2
k^{rec}_{priv}
User
Keyboard
Computer
Smart-card
Recipient
phase 3
DH
Diffie-Hellman key exchange via k_{ks}
2
fresh k'
fresh k'
Encryption
$c = E(k', m = m_i|...|m_0)$
$r_1 = RSA(pr, k^{rec}_{pub}, k')$
$c_{pr} = E(k_{ks}, pr)$
mRSA
$ps_1 = S(k_1, (r_1, c))$
c, r_1, c_{pr}, ps_1
Encryption
3
$m' = D(k', c)$
verify $m' = m$
$pr = D(k_{ks}, c_{pr})$
$r_2 = RSA(pr, k^{rec}_{pub}, k')$
verify $r_1 = r_2$
mRSA
$ps_2 = S(k_2, (r_1, c))$
4
$s = ps_1 * ps_2$ (mRSA)
verify s with k_{pub}
s
verify s with k_{pub}
r_1, c, s
verify s with k_{pub}
$k' = D(k^{rec}_{priv}, r_1)$
$m = D(k', c)$

Figure 4.2: The Smart-Guard protocol – part 2: After phase 2 (see Fig. 4.1) the keyboard and the smart-card are supplied with a verified copy of the user input. Encryption-capable keyboard and smart-card now generate a fresh, shared key k'. The keyboard encrypts the input with k' and adds a partial mRSA [19, 20] signature, which is completed by the smart-card if it agrees.

and sent together with ciphertext and partial signature ps_1 to the smart-card.

The smart-card decrypts the ciphertext and verifies that it is indeed the message m. It then decrypts the encryption parameters pr and recalculates the RSA encryption. If the RSA ciphertext of the keyboard matches the recalculated RSA ciphertext of the smart-card, the smart-card accepts message and ciphertext. The structure is completed with the second part of the mRSA signature ps_2 by the smart-card. Before sending the entire ciphertext with signature s to the recipient, the smart-card verifies the complete signature s to test if ps_1 was correct. The computer can also optionally verify the signature.

The three phases are the logical steps to create a shared, signed ciphertext. Phase 1 distributes the input to the participating devices which sign and encrypt the input in phase 3. The second phase seems unnecessary at first but prevents the keyboard from appending characters to the message.

4.2.5 Important Details

It is worth mentioning some important details in the protocol which are essential to achieve the security guarantees. These details are labelled 1 – 4 in Fig. 4.1 and 4.2.

1. For confidentiality, the characters should be displayed with a technique for confidential output. Integrity protection, and with it trusted output, is not needed. If any device changes the output, the user will quickly realise that the characters on the screen do not match to the typed ones. Also, the displayed hash will be wrong and the smart-card will not sign the input. It will be impossible for the user to type the message. However, if the output is not confidential, neither is the message. The message is still integrity protected but not secret.

2. The Diffie-Hellman key exchange is executed via the encrypted and authenticated channel of k_{ks} (e.g. AES-GCM) to prevent Man-in-the-Middle (MITM) attacks. This key exchange generates a fresh key k' of which keyboard and smart-card both know that it is indeed fresh.

3. The smart-card recalculates the RSA encryption of k' to prevent the keyboard from adding additional keys. A malicious keyboard could e.g. send $\overline{r}_1 = RSA(k_{pub}^{attack}, k')$. This cannot be distinguished from the original $r_1 = RSA(k_{pub}^{rec}, k')$ without recomputing it. To repeat and verify this calculation, it is necessary to send the random parameters c_{pr} (encrypted) to the smart-card.
4. The smart-card creates the signature and immediately verifies it afterwards. This reveals malicious behaviour of the keyboard.

4.3 Security Analysis

Smart-Guard is a malware-tolerant input architecture and consists of three devices. As defined in Section 3.4.3, multiple TCBs must exist. They are displayed for Smart-Guard in Table 4.1. To e.g. compromise integrity, an attacker has to gain control over two TCBs. For integrity, the PC is not part of any of them meaning that its security (including the security of the output) is irrelevant.

Integrity:	Confidentiality:
$TCB_1 = \{$ smart-card $\}$	$TCB_1 = \{$ PC, smart-card $\}$
$TCB_2 = \{$ keyboard $\}$	$TCB_2 = \{$ smart-card, keyboard $\}$
	$TCB_3 = \{$ PC, keyboard $\}$

Table 4.1: Trusted Computing Bases

The TCB model for integrity can be interpreted as shown in Eq. 4.2. Both, keyboard on its own or smart-card on its own provide integrity. Only one of them has to be honest.

$$\begin{aligned} \text{keyboard secure} &\Rightarrow \text{integrity}\ . \\ \text{smart-card secure} &\Rightarrow \text{integrity}\ . \end{aligned} \tag{4.2}$$

The confidentiality side of Table 4.1 is equal to Eq. 4.3. Assuming a confidentiality preserving technique for output, the attacker needs to control three TCBs to compromise confidentiality. This corresponds to two of three devices as every device is part of two TCBs.

$$\begin{aligned} \text{PC secure} \land \text{smart-card secure} &\Rightarrow \text{confidentiality} . \\ \text{smart-card secure} \land \text{keyboard secure} &\Rightarrow \text{confidentiality} . \\ \text{PC secure} \land \text{keyboard secure} &\Rightarrow \text{confidentiality} . \end{aligned} \tag{4.3}$$

4.3.1 Proofs

In contrast to classical ProVerif proofs, it was necessary to employ two attackers for Smart-Guard: One is controlling some of the three devices (PC, keyboard, smart-card), while the other one passively waits for messages. This chapter refers to them as the *offline* and *online* attacker respectively. This characteristic simulates the fact that smart-card or keyboard cannot communicate to the network without the PC (see assumptions in Section 3.4.1). As a result, the compromised devices (i.e. the offline attacker) have to trick the PC to communicate to the second attacker – assuming the PC is honest. A malicious PC is modelled by making all communication channels of it public in ProVerif.

Due to the proofs for the protocol becoming fairly long, they are split accordingly to Fig. 4.1 and 4.2 into two parts. The ProVerif scripts and all required files are available online[1].

4.3.1.1 Proofs for Phase 1 and 2

The results of phase 1 and 2 are the pre-conditions of phase 3. Three properties have to be tested:

- Does the keyboard have the correct input? (message integrity keyboard)

- Does the smart-card have the correct input? (message integrity smart-card)

[1] `https://github.com/mdenzel/smartguard`

- Does the protocol protect the input from an online attacker? (confidentiality)
- To verify confidentiality, another property called “strong confidentiality” was introduced. It tests if the offline attacker gets the input (makes four properties to test).

Either the keyboard or the smart-card have to know the correct message in order to proceed to phase 2. Integrity is, therefore, the union of message integrity at the keyboard and message integrity at the smart-card. The four properties are shown as ProVerif queries in Fig. 4.3.

```
(* integrity *)
query m:char;
  event (sc_pass(m)) ==> event (user_pass(m)).
query m:char;
  event (kb_pass(m)) ==> event (user_begin(m)).

(* confidentiality *)
query mess(ch_att, new m).
(* strong confidentiality *)
query attacker(new m).
```

Figure 4.3: ProVerif queries phase 1 and 2: The code excerpt defines four queries:
Query 1: If the smart-card accepts the message m (event sc_pass), the user must have accepted it on the screen output (event user_pass).
Query 2: If the keyboard accepts the message m (event kb_pass), the user must have created that message (event user_begin).
Query 3: The offline attacker does not get the message m.
Query 4: The online attacker does not receive message m.

4.3.1.2 Proofs for Phase 3

Phase 3 assumes the results of phase 2. This should be that the keyboard or the smart-card have a user-verified copy of the message. The message m might be compromised by the offline attacker but was not sent to the online attacker. The protocol will create a joint ciphertext and signature. The proofs verify that the signature at the recipient side is indeed correct and that the online attacker cannot retrieve the plaintext of the message m (see Fig. 4.4).

```
(* integrity *)
query m:char;
   event (rec_end(m)) ==>
          event (kb_begin(m)) ||
          event (sc_begin(m)).

(* confidentiality *)
query mess(ch_att, new m).
```

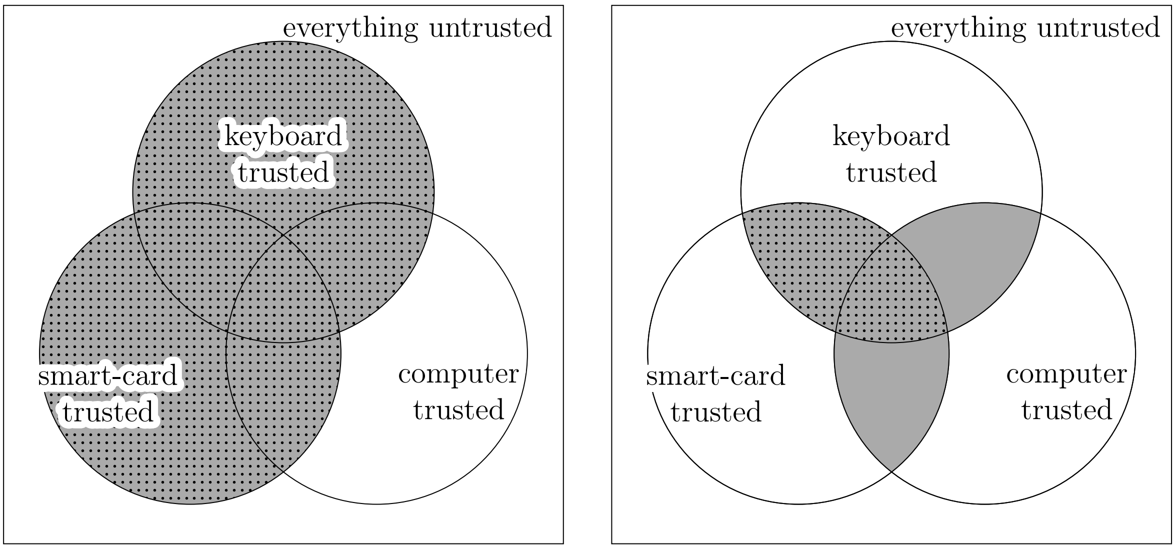
Figure 4.4: ProVerif queries phase 3: The queries are:
Query 1: If recipient accepts message m (event rec_end), it came from the keyboard (event kb_begin) or from the smart-card (event sc_begin).
Query 2: The online attacker does not receive the message m. (or: the offline attacker cannot communicate m via channel *ch_att*)

4.3.1.3 Combined Results

The results of the ProVerif queries are shown in Table 4.2. Important for this part is that either smart-card or keyboard have a valid message (integrity column) and that the attacker on the internet did not receive the message (confidentiality column).

No	Compromised Devices	Confidentiality	Integrity	End reached
1	None	✓	✓	✓
2	Keyboard	✓	✓	✓
3	Smart-card	✓	✓	✓
4	PC	✓	✓	✓
5	PC, Smart-card		✓	✓
6	PC, Keyboard		✓	✓
7	Smart-card, Keyboard	(✓)		✓
8	All			✓

Table 4.2: ProVerif results: Brackets indicate that a property does not hold in phase 3 of the protocol.



(a) Integrity

(b) Confidentiality

Figure 4.5: Trust model for Smart-Guard: The "keyboard trusted" circle represents the cases in which the keyboard is trusted, and similarly for the other two circles. Thus, the very centre of the diagram means that all devices are trusted.

Grey area: protocol satisfies integrity (Fig. 4.5a) or confidentiality (Fig. 4.5b).
Dotted area: protocol defends against hardware keyloggers.

The results are summarised in Fig. 4.5a and 4.5b. Each circle in the Venn diagram represents that a device is trusted while the complement stands for the device being compromised. The grey area marks cases satisfying the particular property labelled below the figure. Smart-Guard guarantees integrity for either a trustworthy keyboard or a trustworthy smart-card (Fig. 4.5a) and confidentiality for one malicious device out of three devices (see also Fig. 4.5b).

It was also tested if Smart-Guard defends against hardware keyloggers. A hardware keylogger corresponds to the channels between PC and the other two devices (usually USB channels) being public. ProVerif verified these queries and, thus, Smart-Guard defends against hardware keyloggers in some cases (see dotted area in Fig. 4.5a and 4.5b).

4.3.2 Performance Estimation

To estimate performance I use the crypto benchmark of Petr Svenda et al. [35, 125] and assume a Softlock SLCOS InfineonSLE78 smart-card. The measurements of the benchmark are shown in Table 4.3.

Function	Average time
Random Number Generation (RNG) (256B)	32.12 ms
SHA2-256 hash (256B)	36.56 ms
AES256 encrypt (256B)	2.46 ms
RSA1024 CRT decrypt	56.27 ms
RSA1024 CRT encrypt	5.43 ms
RSA1024 CRT SHA PKCS1 PSS sign	124.01 ms
RSA1024 CRT SHA PKCS1 PSS verify	81.69 ms
DH (ALG_EC_SVDP_DH/ALG_EC_SVDP_DH_KDF)	85 ms

Table 4.3: Cryptographic operations benchmark of Softlock SLCOS InfineonSLE78 smart-card according to [35, 125]

Phase 1: In the first phase, we only distribute the characters to the devices, for this, a random number generation, an AES encryption, and an AES decryption are necessary: $RNG + AES256 + AES256 = 32.12\ ms + 2.46\ ms + 2.46\ ms = 37.04\ ms$. This is equivalent to an average typing speed of $60\ s/37.04\ ms * 256\ bytes \sim 415000$ characters per minute.

Phase 2: The main bottleneck for the short string is the delay imposed by the user. Thus, the cryptographic performance is negligible: $RNG + SHA256 + RNG + AES256 + AES256 = 32.12\ ms + 36.56\ ms + 32.12\ ms + 2.46\ ms + 2.46\ ms = 105.72\ ms \sim 0.1\ s$.

Phase 3: To generate the resulting ciphertext, the protocol executes (in order): a Diffie-Hellman key exchange, an AES encryption with random number generation, a

RSA encryption (for the key) including random number generation, an AES encryption of the initialisation vector with random number generation, a RSA signature, an AES decryption, an AES decryption of the initialisation vector, a RSA encryption with the same random number as before, a RSA signature, and a RSA verification.

The calculation is show in Eq. 4.4, assuming a message of 256 bytes and Chinese Remainder Theorem (CRT) for RSA with key length 1024 bytes.

$$\begin{array}{llllllll}
\text{DH} & + \text{RNG} & + \text{AES256} & + \text{RNG} & + \text{RSA(enc)} & + \text{RNG} & + \text{AES256} & + \\
85\text{ ms} & + 32.12\text{ ms} & + 2.46\text{ ms} & + 32.12\text{ ms} & + 5.43\text{ ms} & + 32.12\text{ ms} & + 2.46\text{ ms} & + \\
 & & & & & & & \\
\text{RSA(sign)} & + \text{AES256} & + \text{AES256} & + \text{RSA(enc)} & + \text{RSA(sign)} & + \text{RSA(verify)} & & \\
124.01\text{ ms} & + 2.46\text{ ms} & + 2.46\text{ ms} & + 5.43\text{ ms} & + 124.01\text{ ms} & + 81.69\text{ ms} & = 531.77\text{ ms} &
\end{array} \tag{4.4}$$

These numbers are a rough estimate and have to be considered carefully. However, they indicate that performance is not a major concern.

4.4 Related Work and Comparison

Table 4.4 gives an overview of the existent techniques for trusted path and compares them based on the information the papers provided.

Comparison:

1. *BitE* – McCune et al. [90]: *Bump in the Ether (BitE)* combines a PC, an encryption-capable keyboard, and a mobile phone to achieve a trusted path. User input is sent from the keyboard to the trusted phone which forwards the keystrokes (encrypted) to the operating system. There, they are distributed to the correct applications. The trusted phone also serves as trusted output.

   The technique only prevents user-space malware with phone, PC, kernel, and keyboard being trusted entities. However, it is also the oldest approach of the presented ones.

Technique	Trusted input	Trusted output	Confidentiality	Integrity	Software attacks	Hardware attacks	Trusted Computing Base
BitE [90]	✓	∼[1]	✓	✓	∼[2]		keyboard, phone, PC, operating system
Bumpy [92]	✓	∼[1]	✓	✓	✓		keyboard, Flicker, TPM
UTP [53]	✓			✓	✓		keyboard, Flicker, TPM, hypervisor
DriverGuard [29]	✓	✓	✓	✓	∼[3]		I/O devices, PC, hypervisor
KeyScrambler [108]	✓		✓	✓	∼[2]		keyboard, PC, operating system
Zhou et al. [145]	✓	✓	✓	✓	✓		hypervisor, hand-held device, TPM
TrustZone [11]	✓	✓	✓	✓	∼[4]		secure world, monitor, bootloader, TrustZone, I/O devices
Smart-Guard	✓	∼[5]	✓	✓	✓	✓	multiple/flexible (see Section 3.4.3)

[1] limited by phone [2] only user-space malware blocked [3] attacks of drivers possible
[4] secure world/monitor attacks possible [5] relies on other trusted output techniques

Table 4.4: Comparison of trusted path techniques: Similar techniques are grouped together (dashed line). Ticks indicate the referred property is secure while a tilde means that it has drawbacks.

2. *Bumpy* – McCune et al. [92]: The authors used a computer with Flicker [91] in combination with a special keyboard to deliver passwords securely to a recipient (e.g. webserver). A phone serves again as output to indicate the receiving application and the webserver. The technique consists of two stages: collecting the keystrokes and encrypting or hashing them in order to send them to the recipient.

   Flicker, a minimal secure environment with hardware support of a TPM, removed the kernel from the TCB. The result is secure also against kernel-space attacks (in contrast to BitE).

3. *UTP* – Filyanov et al. [53]: The authors worked together with McCune to realise trusted input with Flicker. Their approach, Uni-directional Trusted Path (UTP), is aimed at confirmations for banking transactions. UTP switches temporarily away from the operating system to Flicker which securely displays a confirmation dialog and signs the user's input. The result is sent to the bank where it is verified.

   The approach is not a full trusted path since it is normally inactive and only gives the receiver security guarantees.

4. *DriverGuard* – Cheng et al. [29]: In contrast to other techniques, DriverGuard aims to achieve a trusted path in software and shields I/O drivers with a hypervisor. So-called privileged code blocks have access rights to I/O resources while accesses of the guest operating system are denied.

   A problem of the solutions is that privileged code blocks (Drivers) have access to any I/O port. Compromising any driver, thus, compromises all trusted paths. Since DriverGuard solely relies on software, there is no hardware root-of-trust.

5. *KeyScrambler* – QFX Software [108, 80]: The commercial tool KeyScrambler intercepts keystrokes at the keyboard driver, encrypts them during processing of the operating system, and decrypts them in the actual application. The difference to DriverGuard is that KeyScrambler only handles input but manages without a hypervisor. It partly defends against user-space keyloggers.

6. Zhou et al. [145]: The authors of TrustVisor [93] created a user-verifiable trusted path. The approach uses TrustVisor to shield drivers at a lower level than DriverGuard protecting I/O ports, device memory access, device configuration space, and interrupts. Additionally, the authors enabled the user to verify the system with a custom hand-held device with a red/green indicator LED and a TPM. The hand-held device requests attestation from the TPM about the honesty of the platform. The technique defends against software attacks while maintaining usability during run-time.

7. *TrustZone* [11]: TrustZone is a TEE of ARM CPUs. The system is split into two zones, normal and secure world, with different privileges managed by the so-called monitor. Combined with TrustZone-aware components (like a TrustZone-aware touchscreen) it can provide Trusted I/O if the entire TrustZone architecture is trusted.

8. *Smart-Guard*: The technique of this thesis defends against software, hardware, and even attacks of the own devices. The TCB model differs from previous research in the fact that there are flexible trust assumptions: Trust is distributed over multiple TCBs of several devices. If one of the devices is compromised, the others will prevent attacks automatically (Section 3.4.3). The limiting factor is that Smart-Guard relies on other techniques for trusted output.

Problems of Trusted Output: To recapitulate, trusted output is the problem of delivering data securely, i.e. confidential and integrity protected, from a trusted environment to the user or rather a display.

All presented trusted output techniques either rely on a phone or a hypervisor to secure the output. I suspect this cannot be improved without output devices supporting cryptography; e.g. computer screens and graphic cards need to encrypt or decrypt. Otherwise, the device endpoint for output is not secure and can only be assumed uncompromised. The two (part-)solutions all presented papers rely on are hence (1) to shield the graphic pipeline with a hypervisor, shifting the trust to the hypervisor, or (2) additionally displaying the output via e.g. a phone.

The aim of malware tolerance for trusted output is to display data securely without a single point-of-failure. But, the output device (the screen) itself is a single point-of-failure without guarantees about confidentiality because general purpose screens are incapable of cryptography. Integrity can be provided by displaying the output on two devices (e.g. computer screen and a phone) and relying on the user to compare them. But, this is inconvenient and unfeasible for general purpose scenarios.

Since Smart-Guard requires confidential output, this can so far only be provided via any other non-malware-tolerant technique for trusted output. It is less secure than a potential malware-tolerant output but this is doubtfully achievable without display hardware evolving. Nevertheless, Smart-Guard provides integrity fully malware-tolerant and independent of the confidentiality of the output.

4.5 Summary

This chapter presented Smart-Guard, the first malware-tolerant protocol to protect user input from malware and hardware attacks. It is especially useful to authenticate user input. Smart-Guard consists of three devices – a PC, a keyboard, and a smart-card – and guarantees security properties even in the context of attacks of one of the underlying devices. The protocol is designed to be secure under several sets of trust assumptions, providing flexibility and avoiding a single point of failure making it malware-tolerant. To provide evidence of the claims, the protocol was formally verified with ProVerif and analysed its security properties.

This first practical chapter demonstrates how several devices can interact in a way that prevents any individual device from compromising the resources – *malware tolerance.*

CHAPTER 5

SENSOR/ACTUATOR NETWORKS: SELF-HEALING INDUSTRIAL CONTROL SYSTEM

Having seen how to securely send strings like commands to a recipient, we will now look at the execution thereof in a specialised scenario: Industrial Control Systems (ICSes).

To recapitulate, ICSes are sensor-actuator networks controlling physical systems. They consist of PLCs, which are computers with specialised software to provide high availability and real-time operation. Unpatched PLCs with long lifetimes and historic protocols without even basic authentication (like the Modbus protocol [55]) create an insecure environment. As soon as adversaries gain network access, they are in charge of the architecture. Governmental organisations [70, 122] recommend the defence in depth strategy by trying to deploy defences at every layer of the network.

Malware tolerance goes one step further and aims to distribute trust over several independent components in a way that an individual component infected with malware cannot break the security policy. Simply put, every single point-of-failure at critical intersections should be removed throughout the entire ICS architecture. The secondary goal of this chapter is to enable the architecture to automatically repair ordinary and malicious faults (self-healing). With this approach, it is also possible to recover from corrupted or incomplete patches.

Contributions:

- This chapter presents the architecture of a malware-tolerant ICS that has no single point-of-failure at critical intersections and can self-heal failed or (maliciously) misbehaving PLCs.
- It also gives formal proofs of the network architecture with state-of-the-art protocol verifier ProVerif. The proofs can be found online[1].
- To achieve the architecture, I develop a self-healing mechanism which detects incorrect behaviour by verifying invariants, and recovers to a good state. I adjusted FreeRTOS[2] to include this mechanism and released the implementation as open-source[3].

5.1 Proposed Architecture

The approach is an extension to already existent firewalls, network zones, IDSes etc. and changes the control loop at the field site. Figure 5.1 displays the infrastructure with the changes being highlighted in red. The concept adds hardware in form of reset-circuits; data by images and policies; and software in form of a self-healing Real-Time Operating System (RTOS) and the netboot firmware of the reset-chip. Additionally, the architecture leverages existent redundancy of PLCs and a 2-out-of-3 (2oo3) circuit which are already in place in some ICS facilities.

Basis of the malware-tolerant architecture are three diverse PLCs combined with trusted computing. The 2-out-of-3 hardware circuit combines the results of the PLCs and forwards them to the actuator. That means none of the PLCs has to be invulnerable to attacks or failures, it is enough if two of the three work. The PLCs must differ in their

[1] `https://github.com/mdenzel/malware-tolerant_ICS_proofs`

[2] `www.freertos.org`

[3] `https://github.com/mdenzel/self-healing_FreeRTOS`

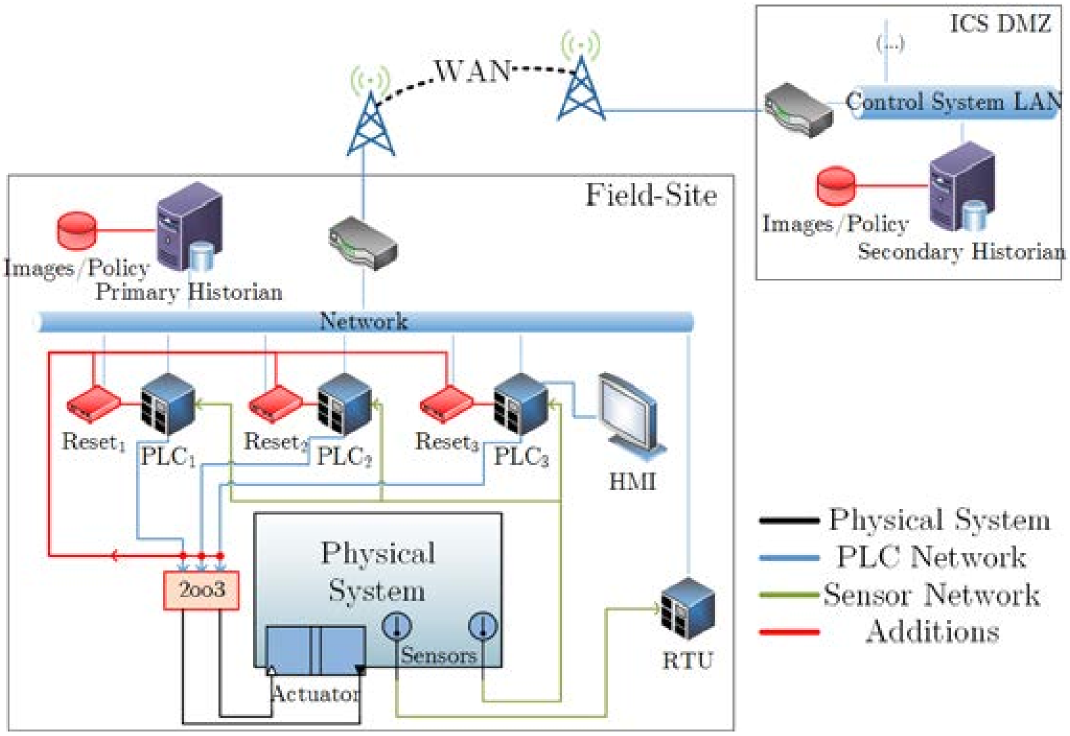


Figure 5.1: Proposed Industrial Control System architecture

soft- as well as hardware which is achieved with a special kind of N-variant system and diverse hardware (details in Section 5.3.2).

I also added self-healing functionality to recover failed and compromised PLCs with (1) a RTOS based on ARM TrustZone that can reset user level tasks and (2) a network protocol and reset-circuits to defend against attacks on system level and on the TEE.

5.1.1 Self-healing Real-Time Operating System

To demonstrate the RTOS, I created a proof-of-concept implementation based on the FreeRTOS operating system which I ported to ARM TrustZone to protect critical functionality like scheduling and interrupts.

Figure 5.2 shows the control flow of my TrustZone-aware RTOS. Periodically, the TZIC generates a timer interrupt (1.) which is set up as an FIQ trapping into monitor mode (2.). The monitor saves the context and jumps to the interrupt handler (3.) for timer interrupts calling the FreeRTOS scheduler (4.). After scheduler (5.) and interrupt

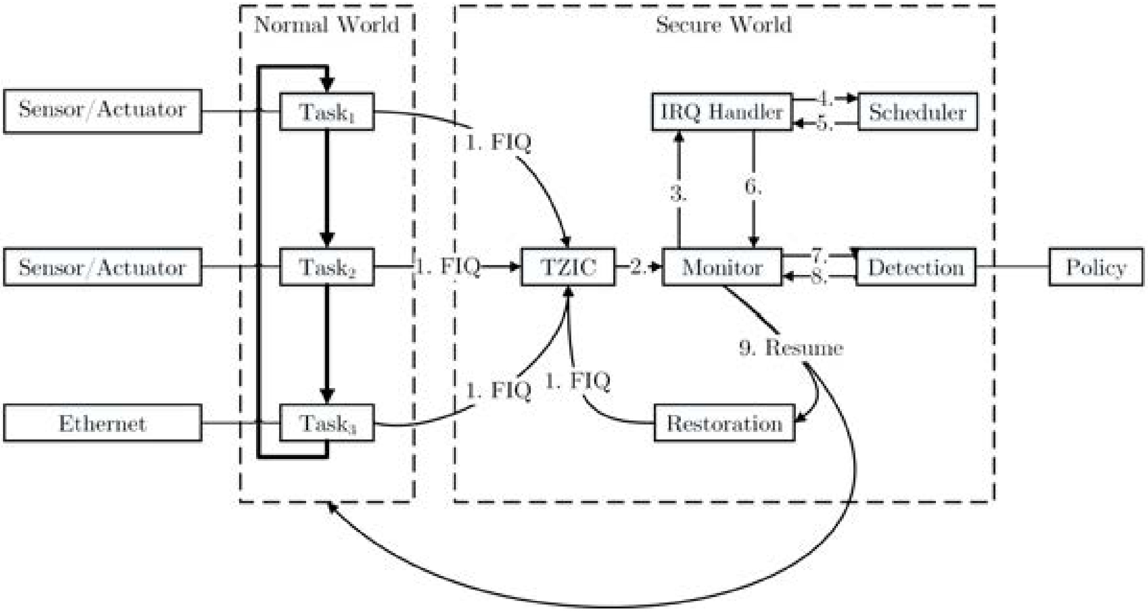


Figure 5.2: TrustZone-aware Real-Time Operating System

handler (6.) return, the next *task* is determined. At this point, the monitor invokes a detection routine (7.).

To reveal faults or malicious behaviour of certain tasks, the detection routine checks various system variables and external values (e.g. sensor values) against invariants which are stored in form of a policy, cryptographically signed, on at least two servers. These invariants are implicitly given by the set-points the operator of the system placed. For the water tank example, the operator could e.g. set the temperature t to 0 to $100°C$, forbid heating for $t > 50°C$, and forbid cooling for $t < 50°C$. If the temperature is below or above this range and the task does not enable the actuator, the task is faulty.

The result of the detection routine is returned to the monitor (8.) which then (9.) either runs the task or dispatches a restoration routine if the task was misbehaving. The restoration routine only runs during the time-slice of the misbehaving task ensuring availability of the rest of the system including other tasks and operating system functionality. The restoration terminates the task and loads an image of the original task from a protected memory region inside the secure world. Lastly, the task is added to the scheduler again. The critical steps are run inside the TrustZone secure world to protect them from manipulation.

To avoid unnecessary resets due to false positives, I created a specification-based detection technique that verifies the status of invariants. Specification-based techniques (e.g. signatures and finite state machines) have a lower rate of false alarms than non-specification-based ones (e.g. anomaly detection) but might miss some attacks [115]. If the presented online self-healing mechanism fails, the network level self-healing approach (see following paragraph) restores the particular PLC at the cost of a restart.

5.1.2 Reset-Circuits and Network Protocol

The basic idea of the reset-circuits is to reboot the PLCs and load a digitally signed and verified image from the network. The proposed circuits consist of a network boot chip (e.g. iPXE[1]) and a logical circuit to control resets (Fig. 5.3). To restrict resets to a certain interval, a low frequency clock signal was AND'ed. Optionally, the inputs to the circuit (label 1. in Fig. 5.3) can be replaced with flipflops to enable synchronising the PLCs. Circuits for PLC_2 and PLC_3 can be similarly derived.

Since network-based detection indicates that system level self-healing (taking place beforehand) failed, I intentionally clear the state to recover from the attack. The state can either be re-initialised by discovery or by requesting it from the other PLCs. In the temperature management example, discovering the temperature and adjusting the actuator is straightforward. For more complex scenarios, the reset PLC would request the state from the other two PLCs and compare it.

The message sequence chart of the malware-tolerant, self-healing network protocol is presented in Fig. 5.4: Every PLC reads the current sensor value s (for simplicity only one sensor was drawn but multiple sensors are possible) and computes the adjustment a_i of the actuator. This is sent to the 2-out-of-3 circuit which forwards the end result a to the actuator. Parallel, each reset-circuit receives the three response values of the PLCs and checks if the corresponding PLC needs resetting ($r \overset{?}{=} 1$). A reset flashes a PLC with netboot image from the network. Figure 5.4 displays a reset of PLC_1 as an example.

[1] www.ipxe.org

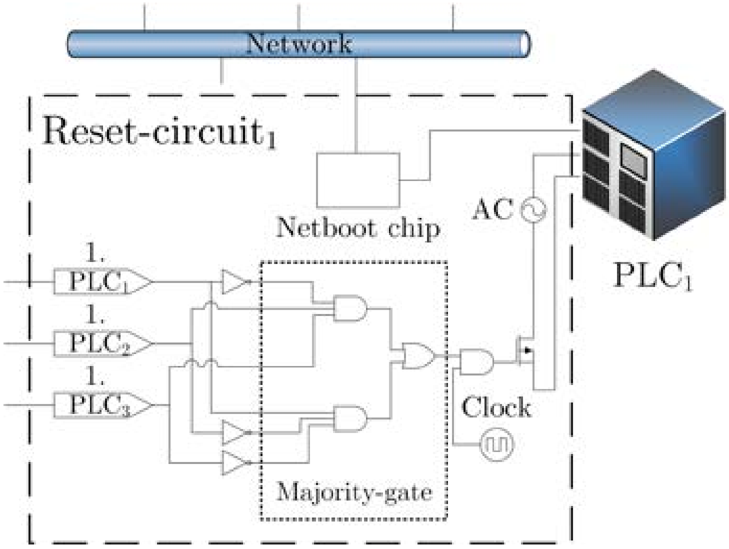


Figure 5.3: Reset-circuit for PLC_1

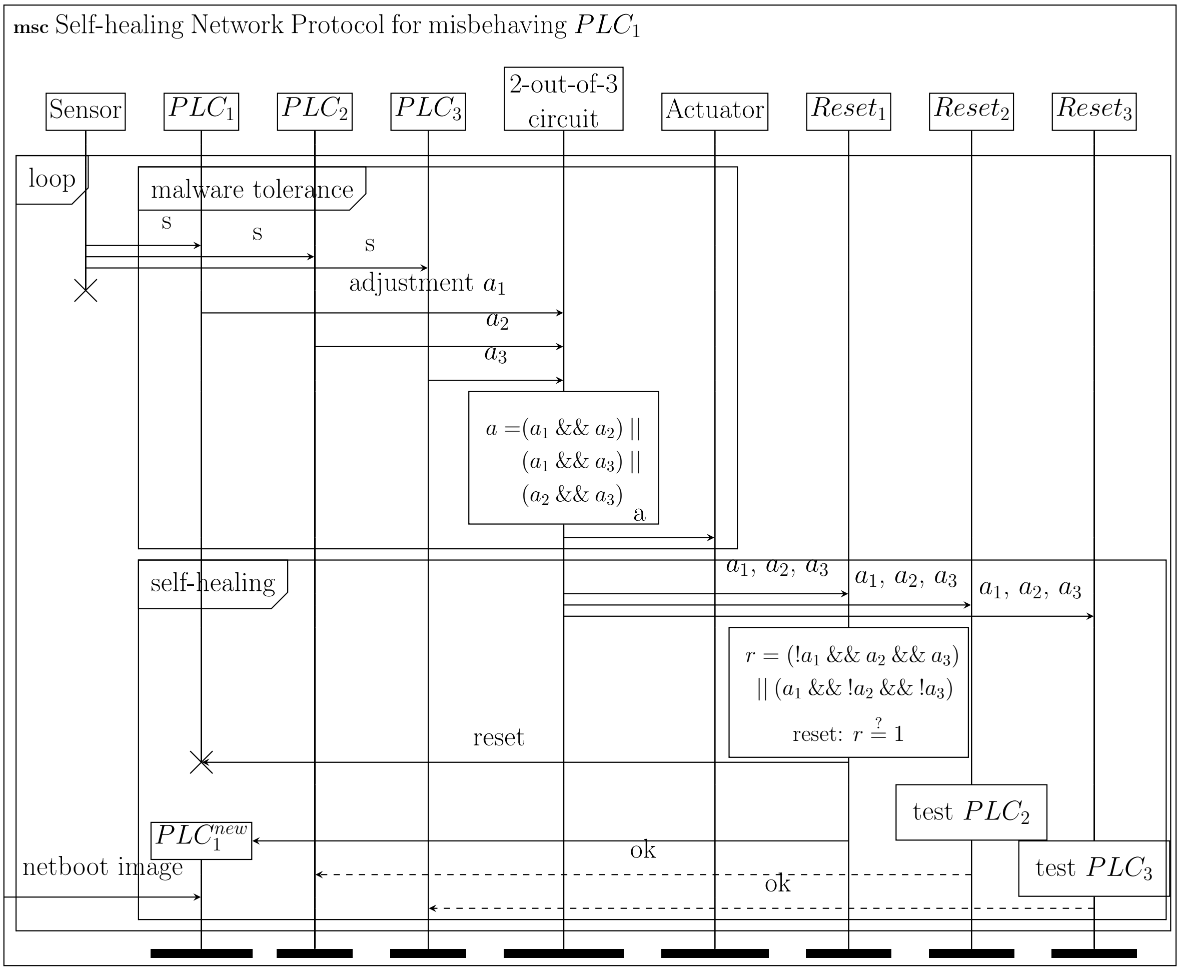


Figure 5.4: Malware-tolerant, self-healing protocol

5.2 Security Analysis and Results

5.2.1 ProVerif Proofs

To give proof of the security features of the illustrated architecture, ProVerif was utilised to test the network protocol. Figure 5.4 shows the modelled protocol for various configurations (see Table 5.2) granting the attacker control over different sets of devices.

The TCB model (see Section 3.4.3) of the proposed ICS is shown in Expression 5.1. R_i stands for the reset-circuit of PLC number i. Since R_i controls PLC_i, one has to consider (R_i, PLC_i) pairs as they are the smallest subset of *independent* components. The system is malware tolerant, because no (R_i, PLC_i) pair is part of all TCBs.

$$\begin{aligned} TCB_1 &= \{(R_1, PLC_1), (R_2, PLC_2)\} \, . \\ TCB_2 &= \{(R_1, PLC_1), (R_3, PLC_3)\} \, . \\ TCB_3 &= \{(R_2, PLC_2), (R_3, PLC_3)\} \, . \end{aligned} \tag{5.1}$$

The architecture was formally verified by testing five properties of the protocol (Fig. 5.4) with ProVerif (results shown in Table 5.2):

1./2. $1^{st}/2^{nd}$ iteration: As ProVerif cannot verify loops, two iterations of the protocol were modelled. These two iterations are sufficient, since computations are independent from each other and resets only affect the next loop iteration. For each iteration, the proof scripts test if the actuator received the correct value.

3./4. Self-healing: The self-healing column in Table 5.2 consists of two proofs; the absence of type I errors and the absence of type II errors.

- Type I error (false positive): Regarding the protocol, a false positive is the case where the hardware circuits reset an honest PLC. In ProVerif this has to be expressed as: For all *reset* events of PLC_i, PLC_i misbehaved.

  At first, this seems not to prove that *if PLC_i misbehaved, a reset happens* but in combination with the type II error and knowing that reset is a binary event (it can either happen or not happen), the property is proven.

- Type II error (false negative): False negative refers to the case where a misbehaving PLC is not reset (missed attack). In ProVerif: For all *not_ reset* events of PLC_i, PLC_i behaved correctly.

  Again, it seems that *if PLC_i is honest, no reset happens* is not proven. This is applicable, similar to before, by the absence of type I errors.

  Figure 5.1 displays the proofs of the absence of Type I and Type II errors as four squares diagram.

5. End reached: The property tests if the protocol runs through. This is commonly done in ProVerif by detecting the deliberate leak of a secret value at the end of the protocol.

	PLC misbehaving	PLC honest
reset	(proved with ProVerif)	false positive
no reset	false negative	(proved with ProVerif)

Table 5.1: Type I and type II errors: To prove the absence of type I and type II errors in ProVerif, the scripts prove that: (3.) if a reset happened, the PLC *always* misbehaved and (4.) if no reset happened, the PLC was *always* honest.

The proofs (Table 5.2) verify that the physical system, i.e. the actuator, is supplied with correct values for the cases where the adversary controls one device (cases 2–4 and 6–8) or one $(PLC_i, Reset_i)$-pair (cases 9–11). Self-healing works for one compromised PLC but functional reset-circuits (cases 2–4). Also, everything works if there is no attack (case 1). More hypothetical cases are tested as sanity checks, e.g. the case where the attacker can physically change the 2-out-of-3 circuit (case 5) which is a validation of the assumption. The proof fails as expected since one hands the asset to the attacker from the very beginning. Case 24 is a special sanity check where the adversary has control over literally everything. Both cases (case 5 and 24) fail as predicted.

No	Compromised Devices	1^{st}	2^{nd}	Self-		End
		Iteration	Iteration	healing		reached
		(1.)	(2.)	(3.)	(4.)	(5.)
1	None	✓	✓	✓	✓	✓
2	PLC_1	✓	✓	✓	✓	✓
3	PLC_2	✓	✓	✓	✓	✓
4	PLC_3	✓	✓	✓	✓	✓
5	2oo3					✓
6	R_1	✓	✓			✓
7	R_2	✓	✓			✓
8	R_3	✓	✓			✓
9	PLC_1, R_1	✓	✓			✓
10	PLC_2, R_2	✓	✓			✓
11	PLC_3, R_3	✓	✓			✓
12	PLC_1, R_2	✓				✓
13	PLC_1, PLC_2					✓
14	PLC_1, 2oo3					✓
15	2oo3, R_1					✓
16	R_1, R_2	✓				✓
17	PLC_{1-3}					✓
18	PLC_{1-2}, 2oo3					✓
19	PLC_1, 2oo3, R_1					✓
20	2oo3, R_{1-2}					✓
21	PLC_1, R_{1-2}	✓				✓
22	PLC_1, R_{2-3}	✓				✓
23	R_{1-3}	✓				✓
24	All					✓

Table 5.2: ProVerif results

5.2.2 Evaluation of Self-healing FreeRTOS

Since I am not aware of any Common Vulnerability and Exposures (CVE) for FreeRTOS, I could not test any real-world attacks against my extended FreeRTOS operating system (a broader analysis of attacks follows in Section 5.3.1).

To test the system level self-healing capability of the proof-of-concept implementation, I introduced a buffer overflow in the Input/Output (I/O) driver using the vulnerable C-function *strcpy* (see Fig. 5.5). I exploited this vulnerability by overflowing the buffer and overwriting the settings in the simulated PLC. I chose this attack as it is the most common vulnerability in C and is similar to a range of attacks, e.g. format string attacks and return-oriented-programming.

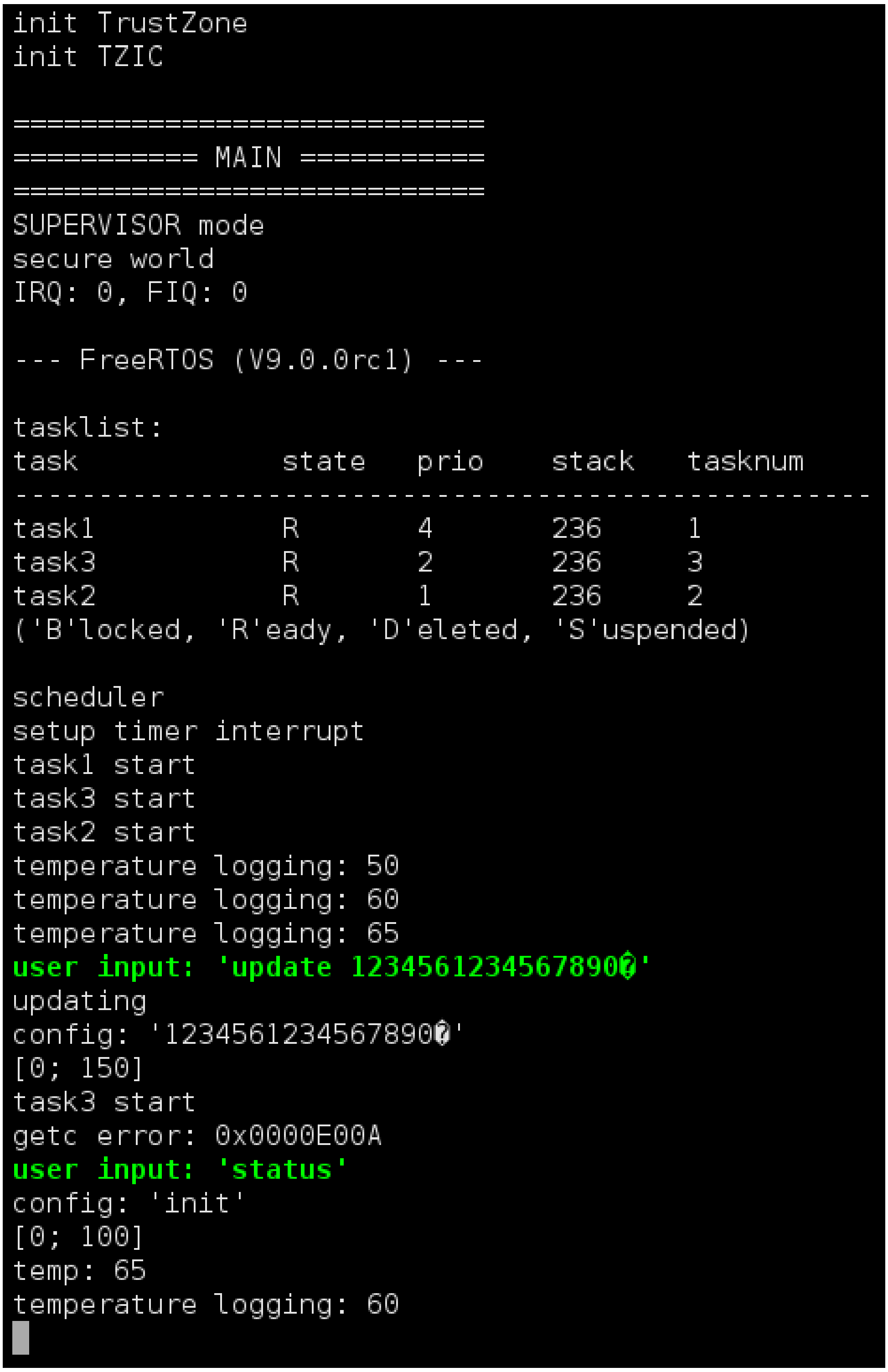
```c
typedef struct temperature{
  char info[16]; //to introduce a buffer overflow
  int max;
  int min;
} temperature;

void set_config_temperature(char* str){
#ifdef BUFFEROVERFLOW
  strcpy(temp_config.info, str); //buffer overflow (on purpose)
#else
  strncpy(temp_config.info, str, 15);
#endif
}
```

Figure 5.5: Example bufferoverflow as introduced into the I/O driver

The simplified detection routine checks that the settings are within an accepted range and otherwise triggers restoration. Task and temperature driver are reset to their original if the maximum temperature is changed to values outside the range.

Figure 5.6 shows an example run of the attack. After the operating system boots (i.e. when scheduler and timer interrupt are set up), three tasks are started. Task1 toggles an LED to indicate that the system is alive. Task2 is logging the temperature which is simulated by reading it from a random function in the driver. The temperature is initially set to be between 0 and 100. Task3 is managing the user input via the COM port. In the example, I sent the user command (green in Fig. 5.6) to update the configuration name including a buffer overflow attack. The last, non-printable character of the input string is the binary value of 150. As we can see in the next rows, the temperature is now set to $[0; 150]$. Upon the next task switch, the self-healing routine runs and verifies the invariant $temperature_{max} <= 100$. Since the invariant is false, the responsible task (task3) is reset including its driver. This can be seen by the repetition of "task3 start" and the *getc* error indicating that the driver ignored a character. The status command after the attack shows



```
init TrustZone
init TZIC

============================
=========== MAIN ===========
============================
SUPERVISOR mode
secure world
IRQ: 0, FIQ: 0

--- FreeRTOS (V9.0.0rc1) ---

tasklist:
task            state   prio    stack   tasknum
---------------------------------------------------
task1           R       4       236     1
task3           R       2       236     3
task2           R       1       236     2
('B'locked, 'R'eady, 'D'eleted, 'S'uspended)

scheduler
setup timer interrupt
task1 start
task3 start
task2 start
temperature logging: 50
temperature logging: 60
temperature logging: 65
user input: 'update 1234561234567890�'
updating
config: '1234561234567890�'
[0; 150]
task3 start
getc error: 0x0000E00A
user input: 'status'
config: 'init'
[0; 100]
temp: 65
temperature logging: 60
```

Figure 5.6: Simulated attack on the self-healing operating system: User input was highlighted in green.

that the configuration was already recovered with temperature borders being $[0; 100]$.

Due to self-healing routine always running in between two tasks and the user input task sleeping directly after getting input, an attacker does not succeed with this attack. The adversary would have to attack the input function of task3 directly to prevent the sleep.

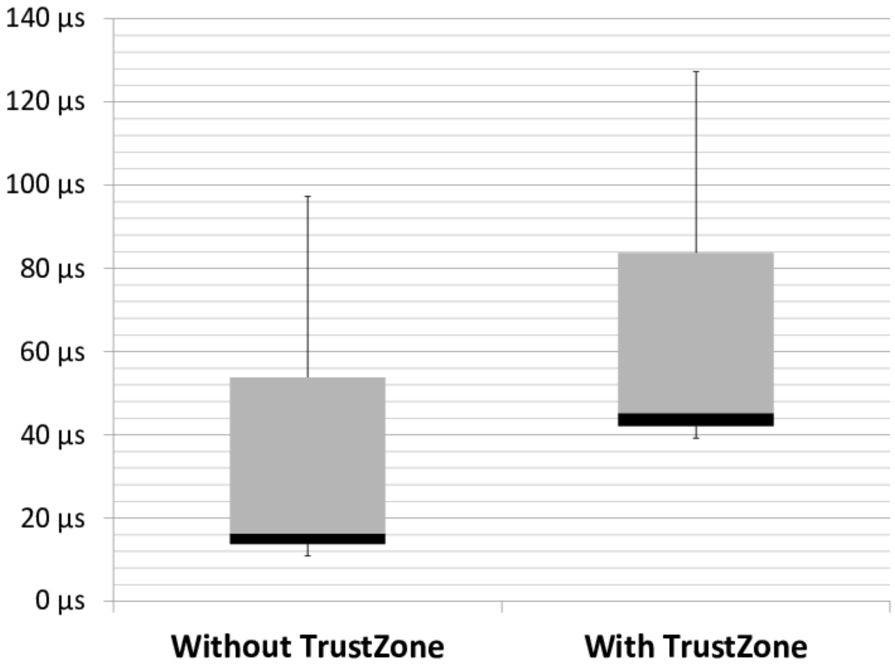
My source code can be found online[1]. However, I emphasise that it is a proof-of-concept implementation and is not entirely secure. Firstly, I omitted common security features like the MMU to speed up development. Secondly, FreeRTOS made some insecure design decisions by running all tasks at kernel privilege level and by only performing memory management in the idle thread risking Denial of Service (DoS) attacks.

5.2.3 Performance Analysis of TrustZone

I conducted a performance analysis of the TrustZone world switch on a FreeScale i.MX53 Quick Start Board with 1 GB DDR3 SDRAM running a 1 GHz ARM Cortex-A8. For this, the time of 1531 task switches was measured on a system running four tasks and FreeRTOS. The measurements start from the timer interrupt until restoring the context of the next task. To measure the time accurately, the program read the CCNT-register which stores the cycle count. The overhead of the timing function calls was 0.9 µs, allowing an accuracy of microseconds. This experiment was executed twice, once with and once without TrustZone. The average overhead of a TrustZone task switch was 29 µs. Figure 5.7 presents the overhead as box-and-whisker diagram and as absolute values.

A TrustZone task switch in comparison to a non-TrustZone one is equal to 3.6 *malloc* system calls (average time for *malloc* on the system: 8 µs) overhead; in other words memory management overhead is comparable to TrustZone.

[1] `https://github.com/mdenzel/self-healing_FreeRTOS`



(a) Box-and-Whisker Diagram

Value	Without TrustZone	With TrustZone
Maximum	(97 ± 1) µs	(126 ± 1) µs
75%-Quantile	(54 ± 1) µs	(84 ± 1) µs
Mean	(16 ± 1) µs	(45 ± 1) µs
25%-Quantile	(14 ± 1) µs	(42 ± 1) µs
Minimum	(11 ± 1) µs	(39 ± 1) µs

(b) Data Values of the Box-and-Whisker Diagram

Figure 5.7: Performance analysis of time for task switches

5.2.4 Security Analysis of TrustZone

Even though all modern ARM CPUs come with TrustZone, a lot of vendors disable it by switching to the normal world immediately at boot. I had to order two different development boards because there are not many boards with enabled TrustZone and the first ordered board disabled TrustZone. Vasudevan et al. [134] confirmed this and criticised that locked down boards prevented the open-source community from developing applications and, thus, prevent a more widespread usage of TrustZone.

Also, TrustZone is only a framework to enable security platforms, it does not provide a fully secure design or implementation. The ARM specification leaves secure storage, secure boot and fuse registers, remote attestation, and trusted path to the chip vendors (confirmed by [134, 51]). Furthermore, TZIC and a TrustZone-aware MMU are not necessarily present and DMA can, if not disabled or controlled through the secure world, compromise security. Feske et al. [51] concluded that using TrustZone does not imply security.

It is difficult to produce platform independent code since the ARM specification is too open leading to various chip configurations with varying security. While this also enables vendors to update security, cost-effective production prevents re-developing many security features.

I welcome the idea of TrustZone as a security enhancement, but digital rights management introduced a few concepts hindering security, e.g. TrustZone is entirely hidden from the normal world appearing as if the CPU does not support TrustZone. Accessing the NS bit even results in a non-intuitive CPU exception jeopardising system stability for Digital Rights Management.

Lastly, mobile phone vendors hardly update their software leading to weaker systems. TrustZone, and its security, depend on updated software and firmware. Bugs can quickly escalate into exploits like the TrustZone exploit of Shen [116] or Google Project Zero [15].

ARM could vastly enhance security by specifying the gaps in the documentation: necessary hardware parts, secure storage, secure boot and fuse registers, remote attestation, trusted path, and update framework.

5.3 Discussion

5.3.1 Attacks

This section examines how the architecture behaves against different attack classes.

- Attacks changing invariants: Let us assume an adversary compromises a PLC but changes some invariants – e.g. he overflows a buffer and inserts a new task but the policy states that there are only N tasks. The system level self-healing immediately resets the tasks, removing the malicious task. This type of defence was demonstrated in a simple version by the buffer overflow above (Section 5.2.2).

- Stealthy attacks (APTs, backdoors, rootkits, trojans etc.): Suppose an attacker manages to deploy a stealthy rootkit on a PLC without being detected. He can now manipulate the PLC as he likes, but the other two PLCs continue to function correctly. If the rogue PLC affects outputs, its reset-circuit will notice and reinstall the PLC, thus removing the rootkit. If the rootkit resides in the normal world, it will be removed even earlier by the system level self-healing.

- Firmware/hardware attacks: Suppose the attacker deploys a firmware or hardware rootkit in a PLC or in a reset-circuit, then software-based self-healing becomes impossible. However, if the other two PLCs remain operational, then manipulations of outputs are still detected and outvoted by those other two PLCs (through the 2-out-of-3 circuit).

- Attacks on network protocol flaws (e.g. Modbus): Since all PLCs presumably have to support the same protocols, exploits targeting the protocol itself (in contrast to its implementation) are not prevented. To defend against these attacks, one has to modify the protocol standard which is beyond the scope.

- Attacks on the policy/administrator account: The system relies on the information in the (cryptographically signed) policy. If the account is compromised, the adversary can change the policy freely. To prevent this, trusted input techniques as in [145] or Chapter 4 should be utilised.

- Attacks on the netboot image and signing key: All netboot images must be cryptographically signed to enable the reset-circuits to verify the image. To avoid attacks, the images are stored at two different locations in the network (see Fig. 5.1) and it is recommended to split the signing key of the images into multiple shares with e.g. mRSA. If, despite these measurements, the adversary gains control over images or signing key, all PLCs are compromised.
- DoS: Since the attacker was deliberately granted full control over some devices, he already has the ability to turn these devices off, but this work aims at not enabling further DoS attacks.

5.3.2 Diversity of Programmable Logic Controllers

It is crucial for the illustrated architecture that PLCs are diverse in their software as well as hardware. To achieve this, I suggest using different reset-chips and different CPU architectures, e.g. ARM TrustZone, Intel SGX, and a PowerPC with a TPM chip. Since the architectures are different, an adversary has to craft different exploits, however, he can still use the same exploit idea to attack the software of all PLCs.

N-version programming was shown to be ineffective against malicious attacks [24, 112] as people make correlated mistakes. Hence, I suggest to use a form of artificial N-variant systems where N systems are crafted such that they are distinct by design [142]. Cox et al. [33] used address space partitioning and instruction set tagging to create different programs that cannot be compromised with the same exploit. Salamat et al. [111] proposed to invert the stack and demonstrated this by a special compiler. Another compiler [84] splits stack into data and control structures.

By utilising N-variant system techniques, we can artificially create distinct software, that cannot be compromised with the same exploit idea – e.g. a RTOS with the stack growing downwards, one with the stack growing upwards, and one with separate data and control stack cannot all be exploited with the same stack-based attack. If each netboot image is additionally diverse from the last one, similar to Sousa et al. [121], an attacker would need to learn the properties of the new image to compromise it. That means it is considerably harder for an attacker to compromise two PLCs at the same time.

5.3.3 Implications

The practical implications of the proposed architecture for the real world are that an attacker would have to find twice the amount of vulnerabilities for an ICS since he has to compromise two different devices (e.g. an ARM TrustZone and an Intel SGX PLC). Hence, the illustrated system would double the cost for the attacker but not for the defender. Considering that many companies already have redundant PLCs, the hardware cost for the company would roughly stay the same. Diverse software versions can be created similar to the artificial N-variant systems. As the special compilers used to generate these variants [33, 111] demonstrate, software can be automatically diversified except for architecture dependent code. In the proof-of-concept implementation based on FreeRTOS, 7.8% is platform specific code (780 lines Assembly; 9956 lines C-code)[1].

Another advantage is that network-based self-healing prevents destroying PLCs through software updates. If a PLC is partially flashed with an image and crashes, it would automatically reboot, triggering the netboot chip to reinstall the image. Thus, patching of the PLCs can now be done conveniently by central image servers with signed images.

5.3.4 Security of Real-Time Operating Systems

RTOSes are time-bound operating system ensuring consistency of the execution. It is crucial that these systems minimise delays and have a high throughput. They come with either soft or hard real-time constraints. While soft real-time operating systems usually

[1] measured with the CLOC Linux tool

meet a deadline, hard real-time operating systems must be deterministic [82].

Unfortunately, security solutions like ARM TrustZone or the TPM did not pick up this requirement and have varying execution times. However, I argue that also these ICSes should be secured on the device itself otherwise defence mechanisms can be skipped as e.g. Stuxnet demonstrated.

Although most security solutions like an anti-virus or IDS impose an intolerable delay on RTOSes, there are other techniques which can be constructed to behave deterministically. The proposed detection mechanism based on invariants has a fixed runtime and also a state-less firewall is deterministic. This shows that some security solutions are applicable or could be developed.

5.3.4.1 Operating System Architectures for Trusted Execution Environments

To evaluate operating system kernels objectively, this section briefly introduces the four general operating system architectures existing nowadays:

1. Monolithic Kernels: Windows, Mac OS, and Linux are all monoliths. They incorporate a large amount of features and drivers into one kernel. These operating systems are convenient from a user perspective but they perform slower than specialised operating systems. An example of a monolithic RTOS is VxWorks. However, with their large TCB, monoliths are difficult to secure and expose a wide attack surface.

2. Microkernel: The most popular operating system architecture for RTOSes are microkernels. These operating systems are small and only provide memory management, scheduling, and inter process communication. Examples for microkernels are FreeRTOS[1], QNX[2], and Genode [60, 50].

   Due to their small size, microkernels can be better secured than monoliths and are even in the scope of verification (e.g. the seL4 microkernel[3]).

[1] `www.freertos.org`

[2] `www.qnx.com`

[3] `https://www.sel4.systems/`

3. Exokernel/Virtualising kernel: Exokernels focus exclusively on multiplexing and protecting the hardware. Since applications have to provide all necessary features, user-space libraries are regularly used for this purpose. A special type of exokernel is a virtualising kernel which allows multiple operating systems to run on the same machine. The Exokernel[1] and Nemesis[2] are two exokernel operating systems. Although it is labelled as microkernel, I classify Qubes [109] as exokernel or more specifically as virtualising kernel, because it uses a hypervisor to run multiple operating systems.

   With isolation being a central feature of virtualisation, these kernels already offer one security guarantee by design. But a problem for RTOS is the reduced performance.

4. Library operating system/Unikernel: The fourth operating systems concept are library operating systems where functionalities are provided via libraries instead of providing a full kernel. Programs are compiled together with the libraries into a so-called unikernel – a program that can run on the hardware. E.g. MirageOS[3] is a library operating system.

   While MirageOS demonstrates that also unikernels are suitable for isolation, hardware features like MMU have to be initialised securely. MirageOS realises this through a hypervisor which belongs to the domain of virtualising kernels.

All in all, microkernels and exokernels seem to be more suitable for secure operating systems. Therefore, I chose a microkernel as basis for my proof-of-concept implementation.

5.3.4.2 Analysis of FreeRTOS

Even though microkernels seemed promising at the beginning, I found a couple of weaknesses in FreeRTOS. It is a microkernel with flat privileges and without memory protection sacrificing security for extra performance. However, I doubt this trade is beneficial for safety or security. During the development of the self-healing operating system, I e.g.

[1] `https://pdos.csail.mit.edu/archive/exo/`

[2] `https://www.cl.cam.ac.uk/research/srg/netos/projects/archive/nemesis/`

[3] `https://mirage.io/`

realised that the idle thread of FreeRTOS is freeing memory. That means if any thread hangs, the idle thread starves and memory will no longer be freed. This leads to a system crash at some point – especially considering embedded systems with limited resources – and subverts availability and safety. This permits an adversary to execute simple DoS attacks. Additionally, forgoing basic security features like privilege hierarchy, MMU, Address Space Layout Randomisation, etc. weakens all devices running FreeRTOS.

Apparently, these are not only concepts of FreeRTOS but RTOSes in general since also VxWorks runs programs as kernel tasks on the highest privilege level without standard memory protection (see Zhu et al. [146]).

The presented performance measurements (Section 5.2.3) indicate that memory management overhead is comparable to TrustZone. Thus, I recommend to built at least deterministic security features into RTOSes, particularly in face of attacks like Stuxnet.

5.4 Related Work and Comparison

5.4.1 N-version System Solutions

Wang et al. [141] proposed an N-version system for distributed services using redundancy and diversity. Their architecture consists of four stages: (1) a cluster of proxies accepts client requests and forwards them to (2) the servers handling the request. The reply is sent to the so-called (3) acceptance monitors which verify the replies. If there are conflicts, the replies are forwarded to (4) the ballot monitors. A voting system is applied to determine which reply is sent back to the client. The proxy servers are synchronised using Javaspaces and a reconfiguration stub is included to manage systems. However, the architecture is not fully intrusion-tolerant since software (Javaspaces) and hardware are not diverse. With diversity of the devices, this technique could develop into a fully intrusion-tolerant approach.

Totel et al. [129] proposed a similar system. They replaced the acceptance monitors with an IDS and diversified the servers. Their architecture can tolerate one intrusion on the servers. Since they only used one proxy and IDS, the architecture is still not entirely intrusion-tolerant. Proxy and IDS are single points-of-failure.

5.4.2 Self-healing Systems

The field of self-healing is less well-established, especially considering security. Ghosh et al. [61] gave a detailed overview of existing techniques. The closest ones to the technique of this chapter are Finite State Automaton (FSA) approaches [130, 71] which model the systems as an FSA and rejuvenate it when invalid states are reached.

Tu [130] suggested a new hardware component, embryonic arrays, consisting of several chips working together. If one chip (cell) fails, another one can take over. The model is biologically inspired by the human immune system and uses a FSA to model states, tolerance conditions, and the generation of tolerance conditions. With this, the chip array should detect failed cells and self-heal automatically. However, the paper did not specify details how a whole system shall be translated into a FSA.

More practically, Hong et al. [71] suggested that the FSA state of a computer can be modelled by the free memory. The authors sample data from sensors (i.e. free memory) and rejuvenated the system based on the memory status. If a certain threshold was reached, the service was restarted. After continuous violations, a system restart was carried out. Hong et al. demonstrated their approach with a proof-of-concept implementation which they tested by inserting artificial memory leaks into an Apache web server.

The self-healing technique of this thesis based on invariants is more efficient than FSA as it does not actively keep track of the state. Since invariants can be declared as regular expressions and every FSA is translatable into a regular expression [95], the technique is as representative as FSA.

5.4.3 Intrusion Tolerance

Two systems are similar to the proposed architecture of this chapter:

- Bessani et al. [16] use proactive-reactive rejuvenation to restore intrusion-tolerant *Crutial Information Switches* (a firewall device) throughout the network. The system consists of several such switches which are (1) reinstalled proactively every few time-frames and (2) recovered reactively if faults or intrusions are detected. It is an example of a hybrid distributed system [138].

  While their system targets the firewall in front of critical devices like PLCs, the approach of this research is aimed at PLCs directly. Additionally, the wormhole devices of Bessani et al. storing the cryptographic key are single points-of-failure. The authors assumed that the wormhole subsystem cannot be corrupted.

- Platania et al. [107] proposed a rejuvenation architecture similar to the one of this chapter. Instead of self-healing upon detected misbehaviour, they proactively rejuvenate PLCs periodically – each on its own to ensure availability of the whole system.

  While the rejuvenation device is separated by air-gap from the rest of the network, it is still a single point-of-failure which could be exploited with e.g. compromised USB sticks.

Periodic system resets can defend against attacks before visible effects occur – independently of any detection algorithm – however, they impose an overhead on the system even though the system is mostly in a valid state. In contrast to Bessani et al. and Platania et al., the approach of this work removes single points-of-failure and keeps the system running as long as possible through system level reactive measurements making it more applicable to scenarios where availability is a major concern.

The differences are design decisions slightly shifting the target of the approach. Bessani et al. concentrated on firewalls and Platania et al. focused on robust ICSes whereas malware tolerance strengthens high availability ICSes.

5.5 Summary

This chapter presented a malware-tolerant Industrial Control System architecture without single points-of-failure at critical intersection points. To achieve this, it relies on diverse, redundant PLCs and a 2-out-of-3 circuit. The infrastructure can push an attacker out of any single PLC using its offline self-healing abilities on the network level. By also employing online self-healing at system level, high availability is maintained during basic failures or simple attacks. ProVerif was utilised to prove the claims and the self-healing capabilities were implemented as proof-of-concept on top of FreeRTOS and ARM TrustZone. Proofs as well as RTOS implementation are open-source.

In this chapter, we have seen how malware tolerance can be applied in a specialised scenario, ICSes, where redundancy is already in place. In contrast to Chapter 4 which only considered one computer, this chapter demonstrated how to strengthen a small network.

CHAPTER 6

MALWARE-TOLERANT MESH NETWORKS

From the specialised network of an ICS, we proceed with networks in general. This chapter presents a malware-tolerant mesh network and also analyses this approach in regard to other types of networks (Section 6.3.2).

Recently, smart-homes and IoT devices gain popularity through e.g. Google's Nest[2], Samsung SmartThings[3], Philips Hue[4], and Amazon Echo[5]. These smart-homes increasingly employ mesh routing, a technique where every device can forward messages. Google WiFi[6] routers already form mesh networks for performance reasons and to support non-uniformly shaped network areas (like a long stretched, narrow flat).

Mesh networks were originally used in isolated settings, e.g. by rescue teams in remote areas, and thus the focus was on reliability and safety. Nowadays, these networks are more and more applied in consumer scenarios like smart-home networks, wireless ad-hoc networks, vehicular ad-hoc networks, and wireless sensor networks. However, with features and time-to-market influencing the strategy of the IoT industry, devices often lack security features. Various forms of attacks like black hole attacks, DoS, routing table overflows, and impersonation exist [72]. As mesh networks are usually flat, compromised devices immediately supply an adversary with access to the entire network. This is in

[2] https://nest.com/uk/about/

[3] http://www.samsung.com/uk/smartthings/

[4] http://www2.meethue.com/

[5] https://www.amazon.co.uk/Amazon-SK705DI-Echo-Black/dp/B01GAGVIE4

[6] https://madeby.google.com/wifi/

particular of concern when taking node capture attacks into account, where the adversary hijacks a device in wide area networks (e.g. a network of weather stations).

While previous research already proposed self-healing techniques for mesh networks [41, 18], these approaches rely on a single point-of-failure – usually a trusted base station – which could be attacked by an adversary to still successfully compromise the architecture.

The aim of malware tolerance is to ensure that most of the network still operates securely when parts of it are entirely controlled by an adversary and for the attack to be contained and limited from spreading further. There shall be no ultimate authority with access to everything – malware tolerance.

Contributions:

1. I present the design of a malware-tolerant mesh network that isolates devices into groups. Groups are able to communicate with each other via a special bridge group which can enforce security properties and filtering like a firewall. The network also applies automatic containment based on voting to identify and quarantine threats, and is in this regard self-managing.
2. To prove the network architecture, the protocol verifier ProVerif is utilised similarly to previous chapters. The proofs can be found online[1].

6.1 Overview

Let us consider a smart-home setting with smart light bulbs, smart fridge, smart door locks, entertainment zone, tablets, phones, laptops, etc. (see Fig. 6.2a). The WiFi of the gateway (the router) does not reach all devices (e.g. due to the long stretched layout of the house and interference) and a flat mesh network similar to Google WiFi is used: All devices route packets. While the laptops and PCs in the network employ a firewall and an anti-virus, light bulbs, fridge, and so on lack these defences and are easier to compromise. Thus, an attacker might get access to a light bulb and start infecting the network.

[1] `https://github.com/mdenzel/malware-tolerant_mesh_network_proofs`

To protect the other devices from attacks (e.g. the data on the laptop), I suggest to (1) isolate different device types. I propose to set up a group for the light bulbs, one for the entertainment devices, one for the work devices, etc. This already limits the adversary to the group of the compromised device.

If an old light bulb is compromised and starts attacking the newer ones, (2) the attack should be automatically contained when this behaviour is detected. All devices should be incorporated in the detection. This is achieved by voting against dishonest devices.

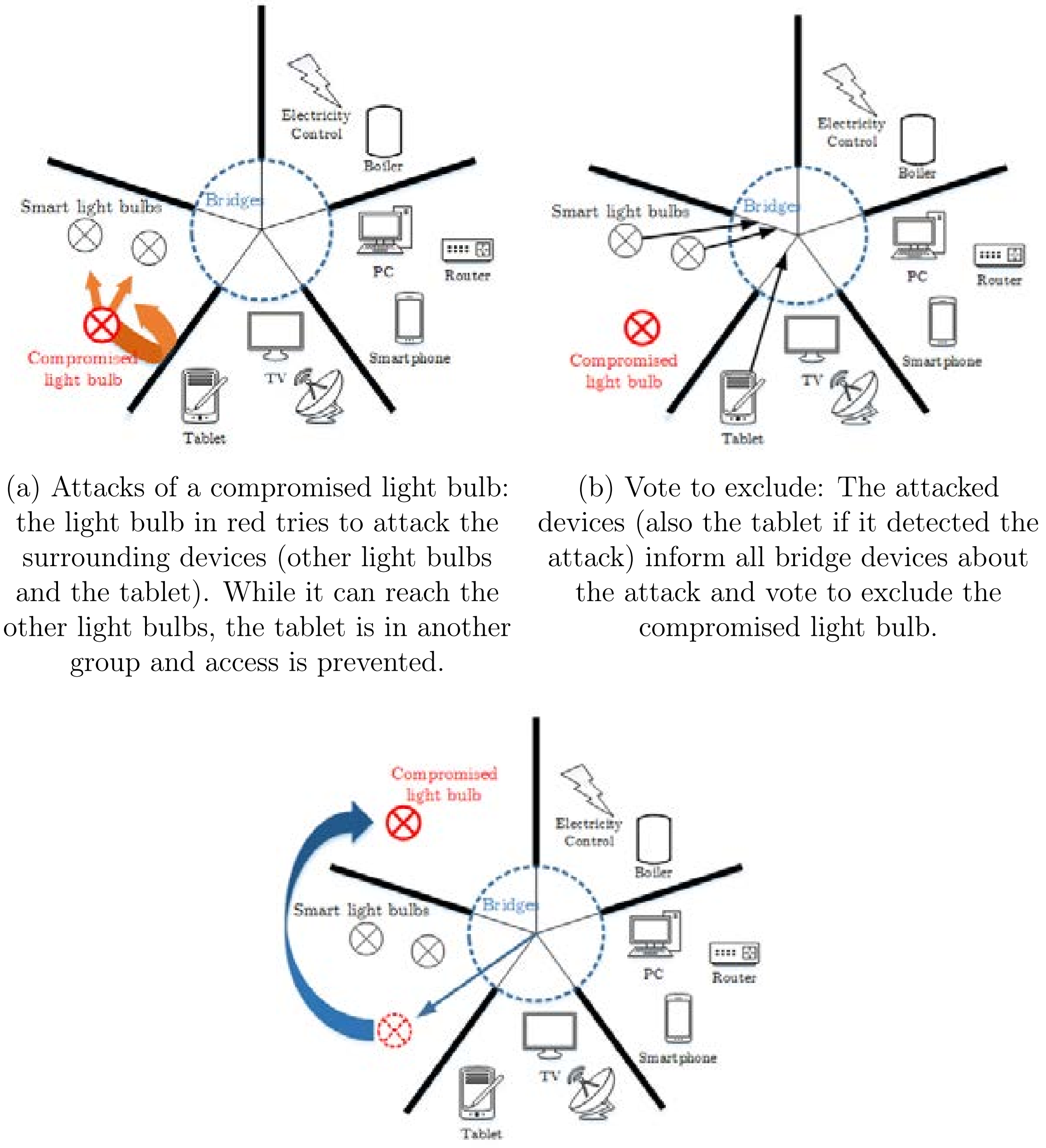
Lastly, there should also be (3) no single point-of-failure (malware tolerance) because this would only shift the target. E.g. if we used the PC to set up the entire network, the adversary would only have to compromise the PC to succeed. Figure 6.1 displays a sketch of a network architecture with isolation and automatic containment.

6.1.1 Proposed Architecture

The proposed network architecture consists of two virtual types of devices: non-privileged devices and privileged "bridge" devices. Non-privileged devices are clustered into groups with every group having one symmetric group key – like a standalone WiFi network. All messages are encrypted (as it is also common nowadays in WiFi networks) and devices can only communicate with devices in their group. To enable groups to talk to each other, a small amount of devices of each group (≥ 2) is promoted to bridge devices.

These bridges have two tasks:

- First, they enable communication between the device groups. A bridge device of one group and a bridge device of another group can re-encrypt and forward messages between those two groups. Filtering, similar to a firewall, works in a distributed fashion, namely at bridges.
- Second, bridges certify keys of non-privileged devices. For this, bridges are divided into two subgroups: bridge group A and B, each of which has an asymmetric key. A key of a non-privileged device is valid, if it was certified by both bridge groups A and B (i.e. by at least two different bridge devices).



(a) Attacks of a compromised light bulb: the light bulb in red tries to attack the surrounding devices (other light bulbs and the tablet). While it can reach the other light bulbs, the tablet is in another group and access is prevented.

(b) Vote to exclude: The attacked devices (also the tablet if it detected the attack) inform all bridge devices about the attack and vote to exclude the compromised light bulb.

(c) Isolation in separate group: The bridge devices react to the attack and isolate the particular light bulb in a new, separate group. Afterwards the malicious light bulb can only make requests to the bridge devices (if one allows this).

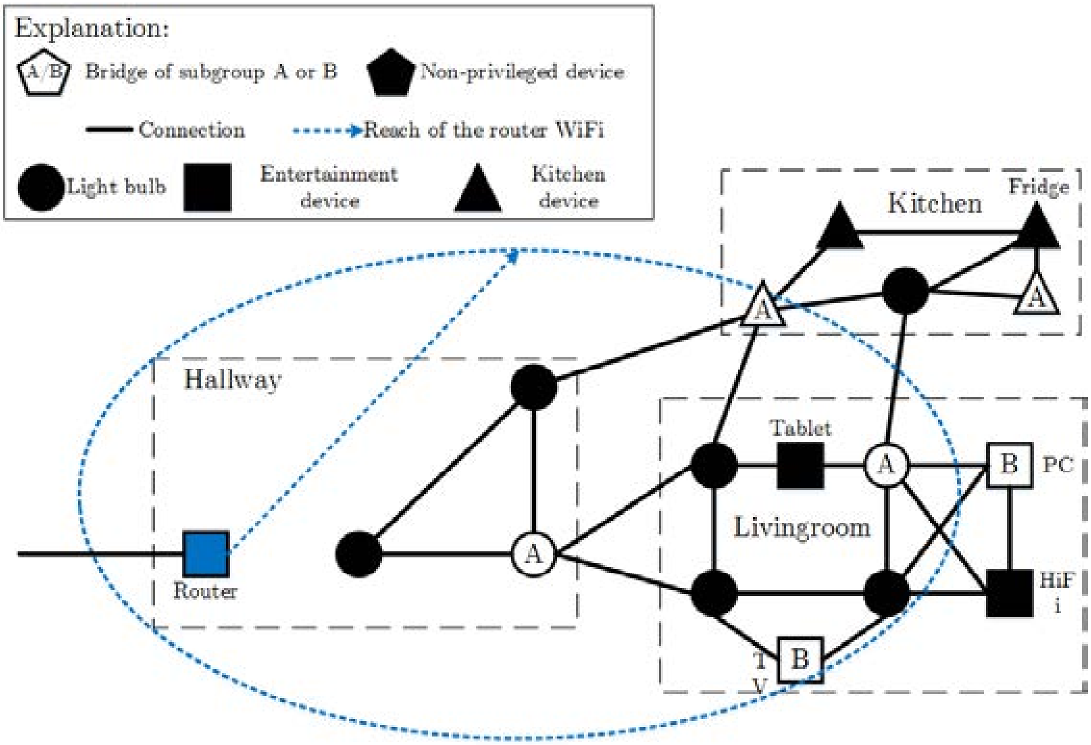
Figure 6.1: Sketch of a network architecture with isolation: Devices are clustered into separate groups which are isolated from each other through a centralised group consisting of several "Bridge" devices. Communication between the groups and management of the network is handled by these bridges. The diagram was inspired by drawings of the microkernel operating system Genode [60] (the original pictures are in Appendix A.1).

Through this, a virtual overlay over the geographical layout of the network was created. An example is shown in Fig. 6.2 where the network is divided into three groups: light bulbs (circles), kitchen devices (triangles), and entertainment devices (squares). Non-privileged group devices are black shapes while bridge devices are white with their subgroup (A or B) written inside the shape. Connections are shown as black lines and as blue dotted circle for the router. Not every device is connected to any other one because walls and electronic currents limit connectivity. E.g. the router cannot directly reach the HiFi stereo system. Fig. 6.2 (b) displays the virtual layout of this network with cryptographic keys.

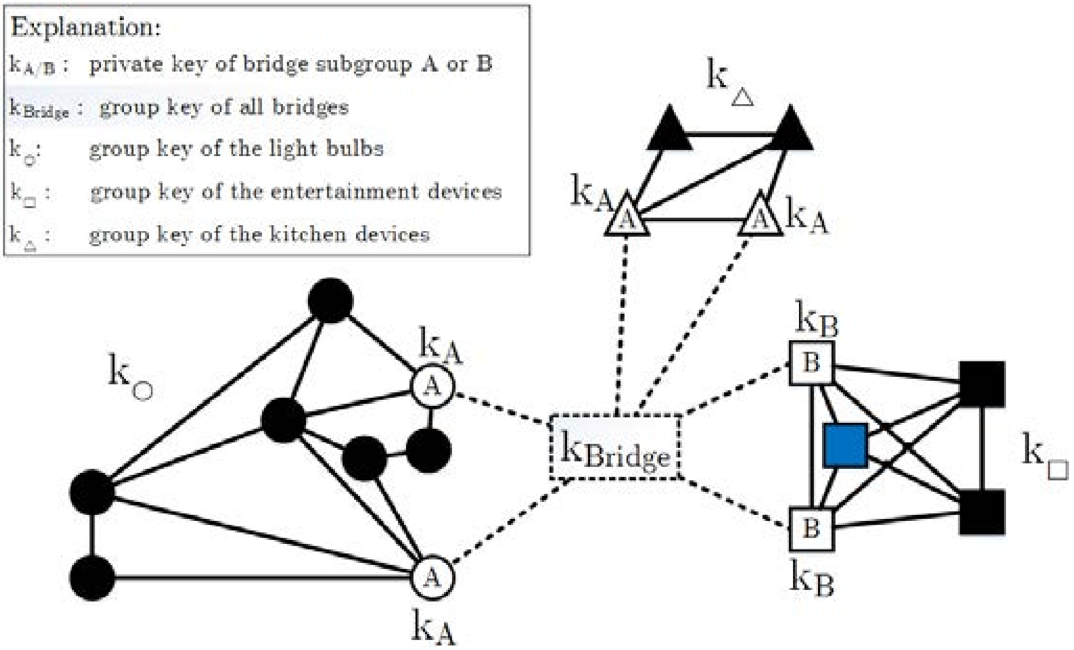
In order to react to attacks, the proposed architecture should be dynamic. Any device can *vote to promote* another device in its group to a bridge device and can *vote to exclude* any device in *any* group from the network. I do not restrict how to detect malicious behaviour; honest devices could e.g. notice ongoing DoS attacks or a malicious device which is dropping packages. An IDS, anti-virus, or honeypot could identify malware delivered from a certain source. In these cases the honest device would vote to exclude that particular dishonest device.

If there are a certain number of votes against a device (minimum two to avoid attacks), the distrusted device is isolated by placing it into a separate new group and revoking its permission to join any other group (see Fig. 6.1). If the isolated device was a bridge, it is also removed from the bridges (the keys evolve) and a new bridge is elected. To forbid quarantined devices to promote themselves, at least two group votes are needed to promote a device and devices are only allowed to vote to promote in their group (note that vote to exclude is possible against every device). Thus, a dynamic network architecture was realised by applying a moving target defence.

A potential adversary compromising one device in the network is, therefore, limited to the group of the device. He would have to compromise multiple bridge devices at the same time, also those of other groups which the attacker cannot access (yet). Additionally, bridge devices only expose a reduced interface.



(a) Geographic network layout showing actual data links


Explanation:
$k_{A/B}$: private key of bridge subgroup A or B
k_{Bridge} : group key of all bridges
$k_{\bigcirc}$: group key of the light bulbs
$k_{\square}$: group key of the entertainment devices
$k_{\triangle}$: group key of the kitchen devices
$k_{\triangle}$
k_A
$k_{\bigcirc}$
k_{Bridge}
k_B
$k_{\square}$
A
B

(b) Virtual network layout showing the links in each of the groups and the cryptographic keys. Additional to the displayed keys, each device has its own asymmetric key pair (identity key).

Figure 6.2: Example mesh network: The diagrams show the same mesh network in its geographic and virtual network layout.

The outlined network architecture focuses on access control, similar to a WiFi network where the router controls which devices can join the network and who can communicate with whom. The cryptography applied in the approach is mainly to create and supply the architecture, not to secure the requests themselves as this is already implemented by application protocols such as HTTPS. However, the architecture provides every device with an asymmetric key pair which could be used to set up a symmetric key via a Diffie-Hellman Key Exchange, if the application protocol does not already supply this.

The initial malware tolerance definition only states that resources are protected. For networks, the definition needs to be adjusted to incorporate more details about networks:

> *Malware tolerance (for networks) distributes trust over several independent components in a way that an individual component infected with malware cannot gain access (spreading), deny access, or control access (MITM) of devices on the network it did not control before. No part of the network is assumed invulnerable to attacks.*

To achieve this self-managing, malware-tolerant network architecture, six operations are necessary. At the beginning, the network has to be bootstrapped (1. Setup). Non-privileged devices can sign up and join a group (2. Get ticket / Join group), send messages to another group (3. Send), and vote to exclude malicious devices from the group (4. Vote to exclude / Leave group). Inside a group symmetric encryption is used to send messages and, hence, another protocol is not required. For bridges we have to define two operations, they can be promoted (5. Vote to promote / Bridge join) or excluded from the bridge group (6. Vote to exclude / Bridge leave).

The next sections introduce the underlying cryptography as well as the six operations.

6.1.2 Cryptography

The architecture utilises four different types of keys as already shown in Fig. 6.2 (b):

1. Each device has its own asymmetric key pair – the identity key (e.g. $[k^{pub}_{fridge}, k^{priv}_{fridge}]$).

2. Each device is part of one group and has this particular group key (e.g. the light bulb key $k_\circ$).

3. Bridge devices are additionally part of the bridge group with the symmetric bridge key k_{Bridge}.

4. All bridge devices of one group have exactly one of two private keys, k_A or k_B. E.g. both bridge devices of the kitchen group (Fig. 6.2 (b)) have key k_A, but not key k_B. The combination of k_A and k_B is used to certify the identity key of each device. For this, the Intrusion Resilient Signature (IRS) scheme of Itkis et al. [75, 74] is used. A second similar cryptosystem, Intrusion Resilient Encryption (IRE), was proposed by Dodis et al. [44, 45]

The basis of IRS (and IRE) is a static public key and an evolving private key. In IRS, messages can be verified with the public key and the time period in which the signature was created. The private key material consists of the evolving private key and an evolving base secret that is kept separate at a different device. To evolve the private key, a contribution of the base secret is necessary (see Fig. 6.3). There are two ways to modify the private key, called *update* and *refresh* by Itkis et al. An update changes to the next time period while a refresh only changes the private secrets and leaves public key and time period unaffected. If an attacker compromised the secret key, he can read messages of the corresponding time period but does not get further keys. The attacker would have to compromise both the base and the secret key, at the same time in order to succeed. For further reading, the interested reader is referred to the original papers [75, 74, 44, 45] and a summary by Franklin [56].

The presented architecture uses two bridge IRS keys (k_A and k_B) who both run an IRS scheme with the base secret being managed by the corresponding other bridges. Only certificates signed with both keys are considered trustworthy.

IRS is limited to $N = 2^t$ time periods with t being the bit length of the time variable. That means for a 32-bit time variable, we would have $N = 2^{32}$ or roughly 4 billion

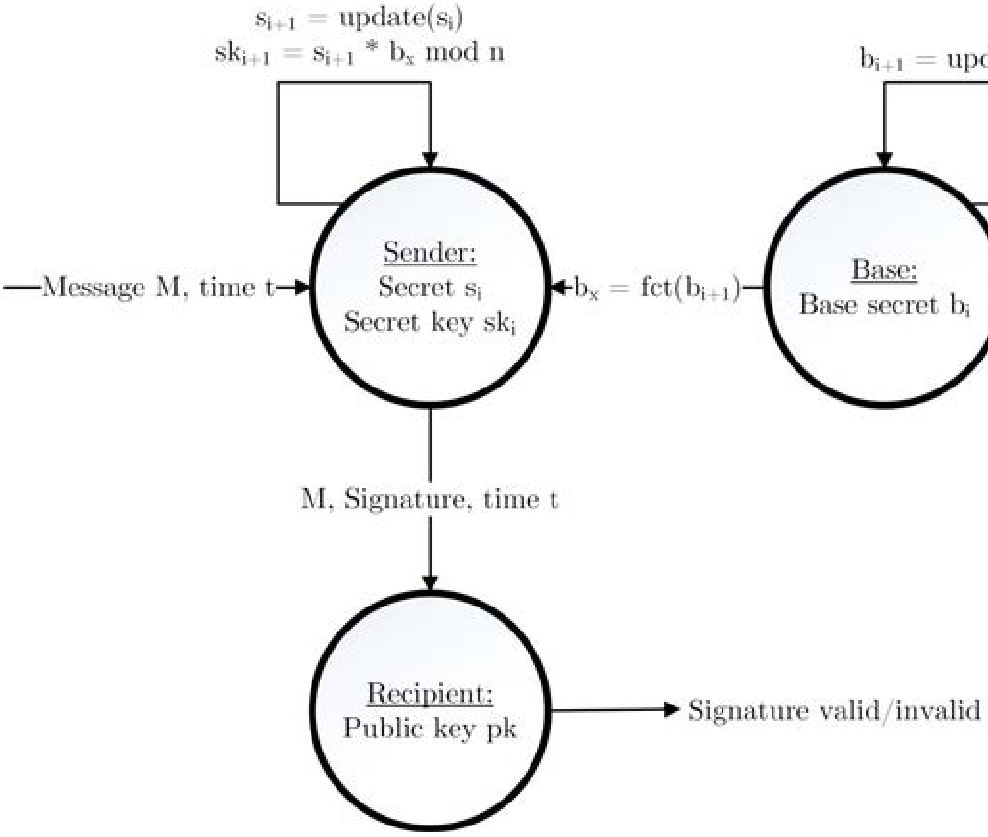


Figure 6.3: Intrusion Resilient Signatures: The scheme relies on an evolving secret key sk_i and a secret s_i at the sender as well as a base secret b_i at the base. Firstly, the base secret b evolves and a share b_x of the base secret is sent to the sender. Secondly, the sender's secret s evolves and s and bx form the new secret key sk_{i+1}. With the evolved secret key sk_{i+1}, the sender can now sign the message M. The recipient verifies the signature with the non-evolving public key pk and the time period t. Details in [75, 74, 44, 45, 56].

time periods. Assuming the network is used 100 years this corresponds to roughly 1 time period every second. Updating the secrets likely takes longer than this, limiting even DoS attacks. At the point where this might run out, a re-setup of the architecture is recommended. If a DoS manages to exhaust all these time periods regardless of the aforementioned restriction, human intervention is necessary.

This chapter abbreviates the Intrusion Resilient Signature scheme with $IRS(k^{priv}, m)$. Also, *time* refers to the time-periods of IRS.

6.1.3 Setup

Initially, the user selects four devices A_1, A_2, B_1, and B_2 to become bridges, each generating an asymmetric identity key-pair. Either the user sets these devices up with all the other public keys (a simple option to establish these channels would be QR codes or a USB stick), or the devices send the keys to each other. The second option assumes that the network is trustworthy at the beginning or the user verifies the fingerprints of the keys.

Afterwards, A_1 and B_1 generate k_A^{priv} stored at A_1 and $base_A$ stored at B_1 of IRS (see distributed key generation of Itkis et al. [75]). Similarly, A_2 and B_2 generate k_B^{priv} at B_2 and $base_B$ at A_2. A_1 and A_2 share their secrets with each other, i.e. k_A^{priv} and $base_B$, and similarly do B_1 and B_2. This establishes the initially needed keys. Two bridge IRS keys can support any number $n \geq 2$ of non-privileged groups. For additional security, three bridge IRS groups (A, B, and C) could be similarly set up. However, I suggest two bridge groups for performance and usability reasons.

6.1.4 GET TICKET / JOIN Protocol

Firstly, we have to define how devices can join the network. A base of four bridge devices was already set up, thus we can rely on them.

If the user wants to connect a new device d (e.g. a smart light bulb) to group g, he logs in at two devices d_1 and d_2 (e.g. phone and PC) both different from d. Then, the

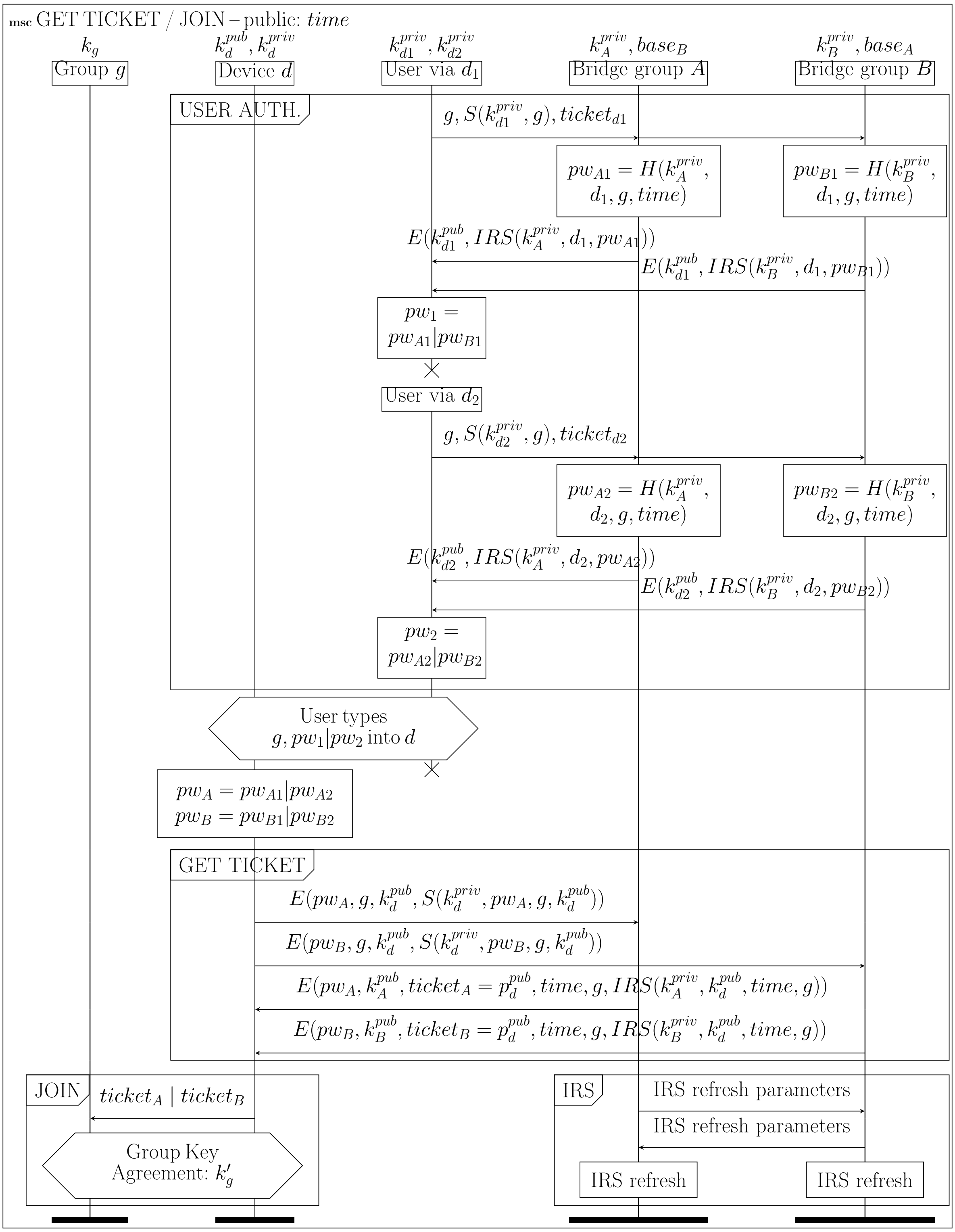

msc GET TICKET / JOIN – public: $time$
k_g
Group g
k_d^{pub}, k_d^{priv}
Device d
$k_{d1}^{priv}, k_{d2}^{priv}$
User via d_1
$k_A^{priv}, base_B$
Bridge group A
$k_B^{priv}, base_A$
Bridge group B
USER AUTH.
$g, S(k_{d1}^{priv}, g), ticket_{d1}$
$pw_{A1} = H(k_A^{priv}, d_1, g, time)$
$pw_{B1} = H(k_B^{priv}, d_1, g, time)$
$E(k_{d1}^{pub}, IRS(k_A^{priv}, d_1, pw_{A1}))$
$E(k_{d1}^{pub}, IRS(k_B^{priv}, d_1, pw_{B1}))$
$pw_1 = pw_{A1}|pw_{B1}$
User via d_2
$g, S(k_{d2}^{priv}, g), ticket_{d2}$
$pw_{A2} = H(k_A^{priv}, d_2, g, time)$
$pw_{B2} = H(k_B^{priv}, d_2, g, time)$
$E(k_{d2}^{pub}, IRS(k_A^{priv}, d_2, pw_{A2}))$
$E(k_{d2}^{pub}, IRS(k_B^{priv}, d_2, pw_{B2}))$
$pw_2 = pw_{A2}|pw_{B2}$
User types $g, pw_1|pw_2$ into d
$pw_A = pw_{A1}|pw_{A2}$
$pw_B = pw_{B1}|pw_{B2}$
GET TICKET
$E(pw_A, g, k_d^{pub}, S(k_d^{priv}, pw_A, g, k_d^{pub}))$
$E(pw_B, g, k_d^{pub}, S(k_d^{priv}, pw_B, g, k_d^{pub}))$
$E(pw_A, k_A^{pub}, ticket_A = p_d^{pub}, time, g, IRS(k_A^{priv}, k_d^{pub}, time, g))$
$E(pw_B, k_B^{pub}, ticket_B = p_d^{pub}, time, g, IRS(k_B^{priv}, k_d^{pub}, time, g))$
JOIN
$ticket_A \mid ticket_B$
Group Key Agreement: k_g'
IRS
IRS refresh parameters
IRS refresh parameters
IRS refresh
IRS refresh

Figure 6.4: GET TICKET / JOIN protocol: The user logs in with two devices d_1 and d_2 to get a password for device d. With this password, d can request a ticket and join group g.

user would request a new one-time password via d_1 and d_2 to sign up the new device d to group g. Half of the password is displayed on d_1 and the other half on d_2.

The total password consists of four shares $(pw_{A1}, pw_{A2}, pw_{B1}, pw_{B2})$. Two of the shares identify to one bridge group, e.g. half-password $pw_A = pw_{A1}|pw_{A2}$ is needed to authenticate to group A (similar for B). Each bridge IRS group generates the shares by hashing k_A^{priv} or k_B^{priv}, the requesting device (d_1 or d_2), g, and the time. This way, any bridge device in the particular bridge IRS group can re-calculate the shares.

To avoid attacks of d_1 or d_2, the half-passwords are sent interleaved, e.g. d_1 would receive $pw_1 = pw_{A1}|pw_{B1}$ and d_2 receives $pw_2 = pw_{A2}|pw_{B2}$. As a result,

- only the user has the full password;
- no device in the entire network has the full password (not even the bridges);
- only A and B have the half-password needed for authentication; and
- d_1 has shares of the password which, alone, are worthless (similar d_2). Thus, neither d_1 nor d_2 can sign up.

The two devices d_1 and d_2 each show their shares to the user who then has the full password. The user concatenates all shares (displayed to the user as two parts) and types the full password into d. With this combined one-time password the new device d can identify itself to both bridge IRS groups and receives a certificate for its public key k_d^{pub}. I named this certificate a *ticket*. Tickets include the time-period(s) in which they are valid as well as the group to which d belongs (here: g). This ticket must be signed by both IRS keys, i.e. by two different bridges, to be valid.

With the ticket, the device d can authenticate to group g and take part in a Group Key Agreement to establish a shared symmetric key with the group. Applicable are e.g. Group Diffie-Hellman, Tree-Based Group Diffie-Hellman, Burmester-Desmedt Group Key Agreement Protocol, and Skinny Tree Key Agreement Protocol. Centralised Key Distribution relies on a trusted central system and can therefore not be used [5]. I suggest using the system of Burmester-Desmedt as it is stateless but the other Group Key Agreement protocols are also possible.

The full GET TICKET / JOIN protocol is also shown in Fig. 6.4. It signs before encrypting because d has to prove that its key k_d^{pub} is connected to pw_A or pw_B, and that d is in possession of k_d^{priv}. Since the one-time password identifies the receiver as well, attacks are prevented through a naming repair [37].

6.1.5 SEND Protocol

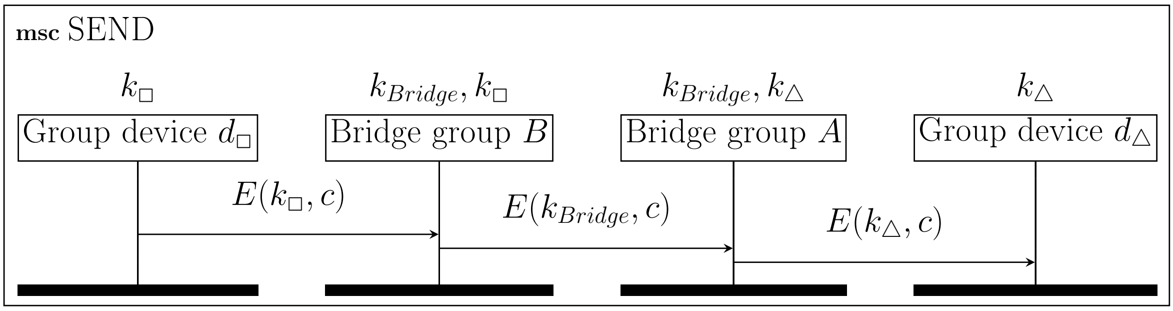


Figure 6.5: SEND protocol

Each group device can communicate via the group key with devices in its group. However, there might be exceptional cases where two devices have to communicate over two groups, e.g. the user wants to check cooking recipes on a tablet (entertainment group) for the content of the fridge (kitchen group).

Sending messages to another group requires bridges to re-encrypt the packet from one group key ($k_\square$) to another group key ($k_\triangle$). The sending device $d_\square$ (e.g. the tablet) would send the packet to a bridge of group $\square$ which re-encrypts the message with the bridge key k_{Bridge} and sends it to a bridge of group $\triangle$. The second bridge re-encrypts the message from k_{Bridge} to $k_\triangle$ and sends it to the recipient (e.g. the fridge). Figure 6.5 displays this as a message sequence chart.

This re-encryption realises filtered communication between the groups. To set up a more efficient connection after the initial SEND protocol, an application protocol (e.g. HTTPS) could provide a session key. Alternatively, such a tunnel could be set up via the certified identity keys of the two devices, i.e. running an authenticated Diffie-Hellman Key Exchange via SEND.

6.1.6 VOTE TO EXCLUDE / LEAVE Protocol

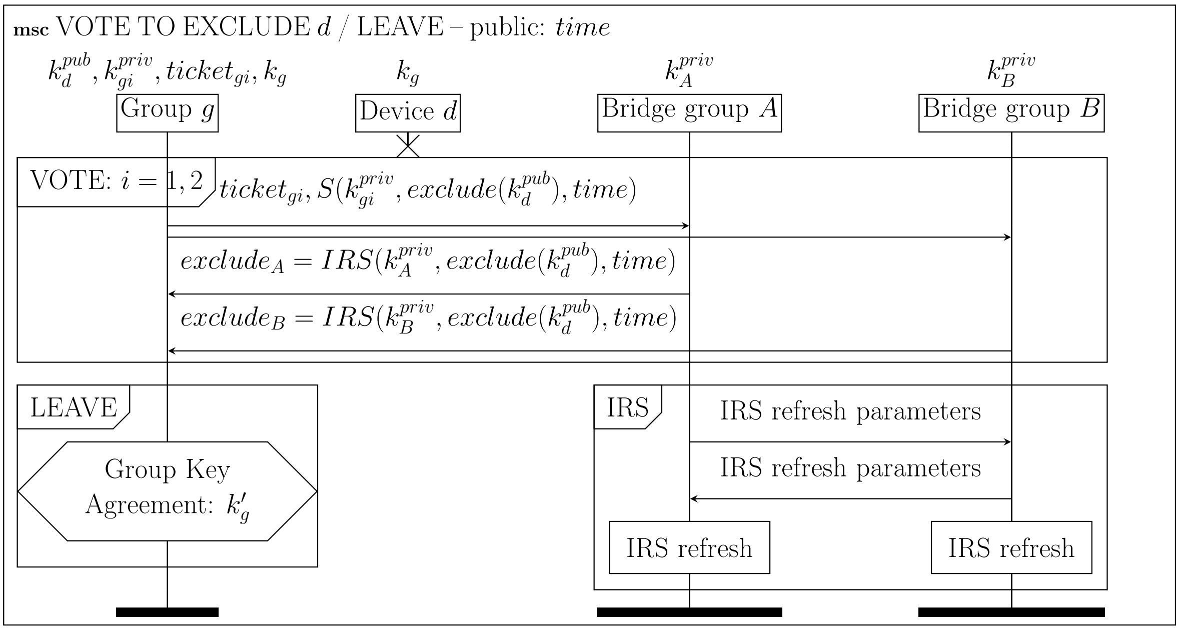

msc VOTE TO EXCLUDE d / LEAVE – public: $time$
$k_d^{pub}, k_{gi}^{priv}, ticket_{gi}, k_g$
k_g
k_A^{priv}
k_B^{priv}
Group g
Device d
Bridge group A
Bridge group B
VOTE: $i = 1, 2$
$ticket_{gi}, S(k_{gi}^{priv}, exclude(k_d^{pub}), time)$
$exclude_A = IRS(k_A^{priv}, exclude(k_d^{pub}), time)$
$exclude_B = IRS(k_B^{priv}, exclude(k_d^{pub}), time)$
LEAVE
Group Key Agreement: k_g'
IRS
IRS refresh parameters
IRS refresh parameters
IRS refresh
IRS refresh

Figure 6.6: VOTE TO EXCLUDE / LEAVE protocol

If devices are misbehaving, bridges must be able to isolate these devices. To force misbehaving devices out of a group, the group key is re-established – so-called leave event in Group Key Agreement protocols.

Misbehaving devices can be identified by any other device which then casts a vote to quarantine the misbehaving device(s) to all bridges. If a certain threshold of votes is received (Fig. 6.6 indicates this by i votes), the misbehaving device is forced to leave the group and the user is informed about the attack. Devices without group are quarantined and cannot communicate with any other device. This way, malware cannot spread further and is automatically contained. Optionally, one could allow quarantined devices to communicate with bridge devices in order to access the internet.

6.1.7 VOTE TO PROMOTE / BRIDGE JOIN Protocol

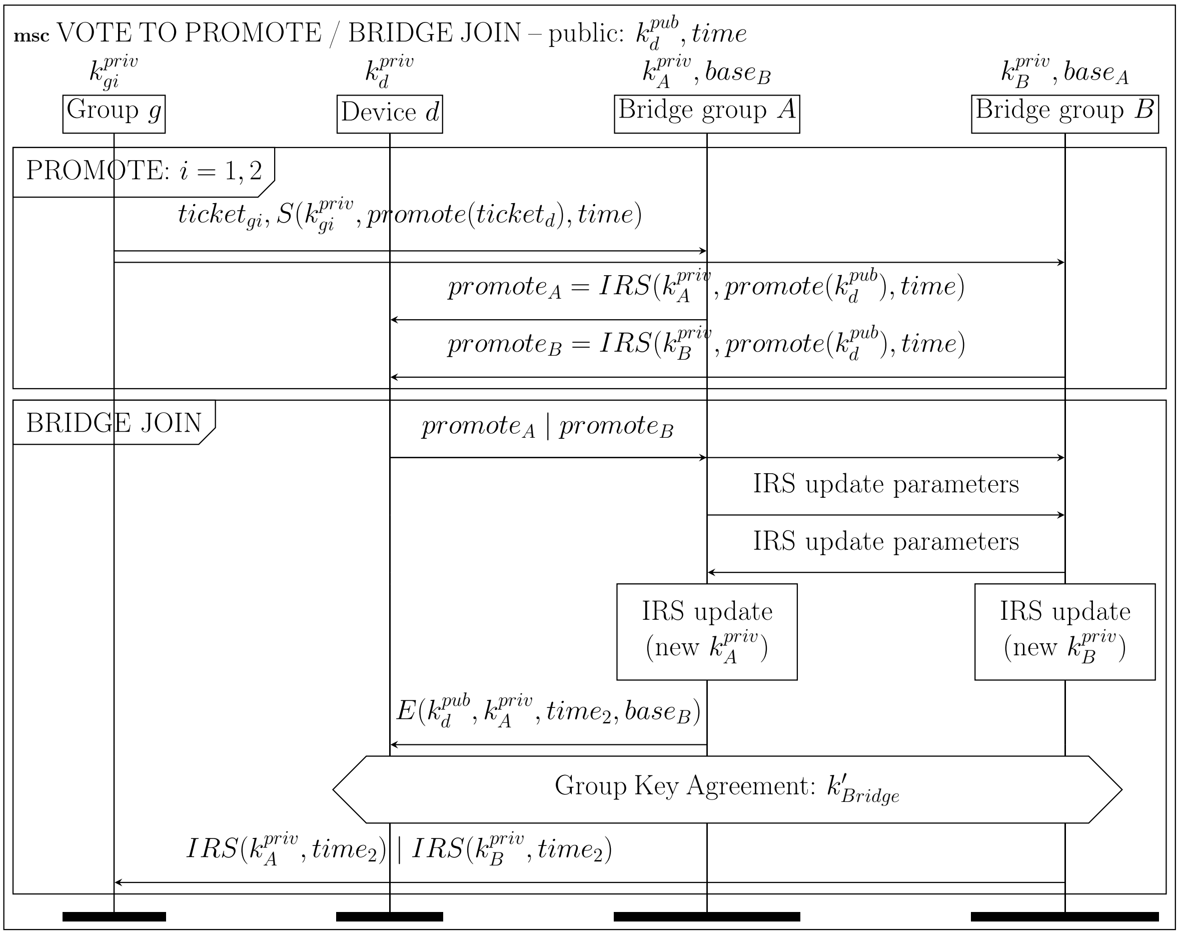

msc VOTE TO PROMOTE / BRIDGE JOIN – public: $k_d^{pub}, time$
k_{gi}^{priv}
Group g
k_d^{priv}
Device d
$k_A^{priv}, base_B$
Bridge group A
$k_B^{priv}, base_A$
Bridge group B
PROMOTE: $i = 1, 2$
$ticket_{gi}, S(k_{gi}^{priv}, promote(ticket_d), time)$
$promote_A = IRS(k_A^{priv}, promote(k_d^{pub}), time)$
$promote_B = IRS(k_B^{priv}, promote(k_d^{pub}), time)$
BRIDGE JOIN
$promote_A \mid promote_B$
IRS update parameters
IRS update parameters
IRS update (new k_A^{priv})
IRS update (new k_B^{priv})
$E(k_d^{pub}, k_A^{priv}, time_2, base_B)$
Group Key Agreement: k'_{Bridge}
$IRS(k_A^{priv}, time_2) \mid IRS(k_B^{priv}, time_2)$

Figure 6.7: VOTE TO PROMOTE / BRIDGE JOIN protocol

When a group is introduced or updated, new bridge devices are necessary. The group members directly elect these devices which are picked depending on their resources and trustworthiness. This work proposes that every device sends a signed message with a device for promotion but this can be replaced by any other voting algorithm tolerating malicious nodes. This *promote* message is signed with the private key of the voter and sent to both bridge IRS groups. Bridge devices can verify the signature and record the vote. Devices lacking the capabilities (e.g. battery) to become a bridge, can opt-out by adding this to their vote. Self-election is possible but at least two votes are necessary to elect a bridge. At the end of the election, both bridge groups issue a promotion ticket and send it to the promoted device to verify that it agrees to being elected. Promotion tickets must expire in the next time period to avoid devices joining twice. The bridge group which the new device should join (A or B, not both), must be fixed and reproducible. I suggest to use a hash like $H(k_d^{pub}, time, k_{Bridge})$.

Promoted devices can join the bridge group similar to the JOIN operation. The new device d sends the promotion ticket to the bridge group and participates in a Group Key Agreement to get the bridge key k_{Bridge}. The difference to JOIN is that bridge devices also share the asymmetric key and base secret of IRS in order to certify other keys in the GET TICKET protocol. This key material needs to be updated and shared with the new bridge device; i.e. to join bridge group A, d gets k_A^{priv} and $base_B$ (see Fig. 6.7). Optionally, this message could be signed with the key of the device sharing the secrets k_{A1}^{priv} and include k_d^{pub} as naming repair (see also [37]). This detects attacks of bridge device A_1.

6.1.8 VOTE TO EXCLUDE / BRIDGE LEAVE Protocol

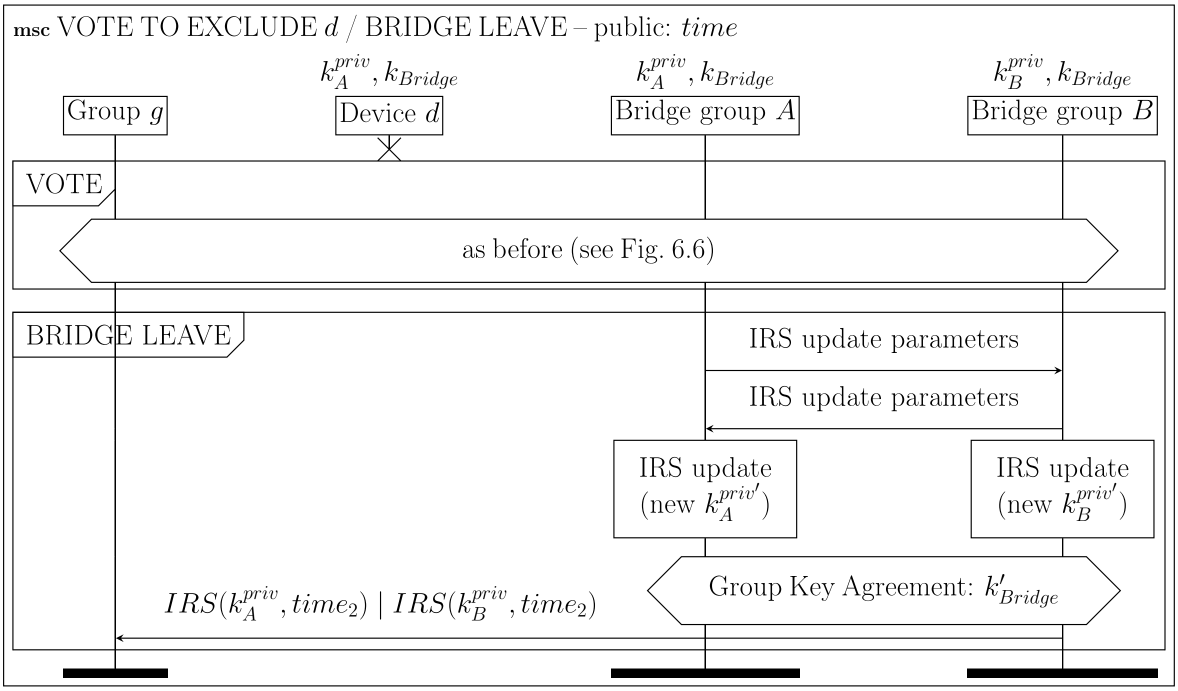


Figure 6.8: VOTE TO EXCLUDE / BRIDGE LEAVE protocol

Analogous to VOTE TO EXCLUDE / LEAVE, devices can also vote against a bridge device. Whereas the vote phase works exactly as before (see Fig. 6.6), BRIDGE LEAVE requires that bridge secrets (for bridge group A: k_A^{priv} and $base_B$) and bridge group key k_{Bridge} are also updated. Additionally, a new bridge needs to be promoted. Fig. 6.8 shows the details of BRIDGE LEAVE. Depending on the used Group Key Agreement, it might be more efficient to run a Group Key Agreement leave protocol first and use the renewed bridge group key to run IRS update.

6.2 Security Analysis

As in previous chapters, the mesh architecture is evaluated using ProVerif and the proof scripts are available online[1]. Each proof one by one assumes every combination of devices compromised and analyses four properties:

1. if the main purpose of the protocol is guaranteed,
2. if a second group cannot interfere with the protocol,
3. if at least one bridge IRS key is secure,
4. if the proof script ran through.

Property 3 and 4 are the same for all proofs. They are introduced first and the other properties are explained below for each protocol.

Property 3: Since there are two IRS bridge groups, it is sufficient when one of them stays secure. With bridge devices changing from time to time, the compromised bridge group can recover from the attack. This is impossible if both IRS groups are compromised at the same time. Thus, the proofs test for each protocol if the confidentiality of at least one of the secret bridge keys is guaranteed.

Property 4: ProVerif does not indicate whether a proof script ran entirely through or aborted early. To test that the proof finished, it is common to leak an artificial secret at

[1] https://github.com/mdenzel/malware-tolerant_mesh_network_proofs

the end of the protocol. If the attacker is in possession of this secret, one knows that the proof script reached the end of the protocol. This was tested as Property 4.

The protocol specific queries are as follows:

GET TICKET / JOIN proves that (1) authentication via the user and the two devices is correct, and that (2) only the right group can be joined with a ticket. Table 6.1 shows the results of the proofs with d_1 and d_2 being the two devices. Important are cases 1 to 6 where the attacker controls up to one device. The protocol is even much stronger than this requirement and only fails if either both bridge IRS groups A and B, or d_1 and d_2 are compromised.

No	Compromised Devices	Authentication	Group 2 join	k_A/k_B secure	End reached
1	None	✓	✓	✓	✓
2	d	✓	✓	✓	✓
3	d_1	✓	✓	✓	✓
4	d_2	✓	✓	✓	✓
5	A	✓	✓	✓	✓
6	B	✓	✓	✓	✓
7	d, A	✓	✓	✓	✓
8	d, B	✓	✓	✓	✓
9	d, d_1	✓	✓	✓	✓
10	d_1, A	✓	✓	✓	✓
11	d_1, B	✓	✓	✓	✓
12	d_1, d_2			✓	✓
13	A, B				✓
14	d, d_1, A	✓	✓	✓	✓
15	d, d_2, A	✓	✓	✓	✓
16	d, A, B				✓
17	d, d_1, d_2			✓	✓
18	d_1, d_2, A			✓	✓
19	d_1, A, B				✓
20	All				✓

Table 6.1: ProVerif results: GET TICKET / JOIN

For VOTE TO EXCLUDE / LEAVE, the proofs verify that (1) a minimum of two votes is needed and that (2) the exclusion only takes place in the correct group. This is true for all cases apart from the ones where both bridge IRS groups are compromised (see Table 6.2).

No	Compromised Devices	Vote	Group 2 leave	k_A/k_B secure	End reached
1	None	✓	✓	✓	✓
2	d	✓	✓	✓	✓
3	A	✓	✓	✓	✓
4	B	✓	✓	✓	✓
5	d, A	✓	✓	✓	✓
6	d, B	✓	✓	✓	✓
7	A, B				✓
8	All				✓

Table 6.2: ProVerif results: VOTE TO EXCLUDE / LEAVE

The SEND protocol is trivial and therefore no proof was formulated in ProVerif. If device $d_{\square}$ only accepts messages signed with $k_{\square}$ and device $d_{\triangle}$ is not in possession of the key, isolation is achieved.

VOTE TO PROMOTE / BRIDGE JOIN proves that (1) one honest device voted for d and at least two promote messages were received. All devices are allowed to vote, i.e. also potentially malicious devices cast votes. However, there must be at least one honest device agreeing in order to prove the absence of attacks. Furthermore, the second property tests if (2) devices of another group cannot promote devices of the original group. The protocol only holds for the basic requirement (one compromised device) as displayed in Table 6.3. The reason is that a compromised A can always leak its secrets to d (case 5) and also a compromised d correctly receives the secrets of A at the end of the protocol. It can then leak them to compromised B (case 6). However, malware tolerance is still guaranteed because the protocol holds for one compromised device (case 1 to 4).

No	Compromised Devices	Promote	Group 2 vote	k_A/k_B secure	End reached
1	None	✓	✓	✓	✓
2	d	✓	✓	✓	✓
3	A	✓	✓	✓	✓
4	B	✓	✓	✓	✓
5	d, A		✓	✓	✓
6	d, B	✓			✓
7	A, B				✓
8	All				✓

Table 6.3: ProVerif results: VOTE TO PROMOTE / BRIDGE JOIN

Since Group Key Agreement and IRS cryptography scheme are already proven by the original papers, VOTE TO EXCLUDE / BRIDGE LEAVE is equivalent to VOTE TO EXCLUDE / LEAVE and, therefore, does not need additional proofs.

All in all, the mesh architecture is secure for at least one compromised device. As expected, none of the protocols holds if both bridge IRS groups are compromised.

6.3 Discussion

The security analysis proves that the architecture is tolerant against attacks (malware-tolerant) and can even recover from infected bridge devices. Devices are isolated in their groups and can only communicate to another group if bridge devices allow this – similar to a firewall. This also enables the architecture to proxy vulnerable or outdated devices by simply moving them into an empty group. Services to the network of the isolated device can be made available or turned off per connection.

In order to control the whole network, an attacker would have to compromise the two bridge IRS groups A and B in the same time-period. That means for n groups, a maximum of n compromised devices can be tolerated (one in each group) and $n/2$ of them can be bridge devices (if they are in the same IRS group). If the threshold for elections is greater than two, even more compromised (non-privileged) devices can be tolerated. Also, votes could be encrypted for additional security as an adversary at a central location – if such a central location exists in the network layout – could block unwanted votes.

The presented mesh network does not automatically defend against stealthy attacks, however, an adversary is restricted to his own group. A compromised bridge device would give access to two groups (one group and the bridge group) but cannot interfere with other groups. I estimate that the system will converge to a trusted state in the long run, because as soon as any attack becomes visible the device will be automatically isolated. The infrastructure becomes a moving target for the adversary which is harder to compromise.

By incorporating the user into the GET TICKET protocol, clone attacks are prevented. Isolating the internet gateway hardens the architecture against exfiltration but this also imposes an overhead on internet traffic.

DoS attacks targeting the cryptographic routines are unavoidable but they are easily identified, resulting in the misbehaving device being quickly removed from the network.

6.3.1 Performance and Adoption

A major concern for all types of network is performance. The suggested inter-group communication requires three symmetric encryptions until the session key is set up. Hence, devices communicating a lot with each other should be organised in the same group, such that connections between the groups is an exception. With the expensive operations JOIN and LEAVE occurring seldomly – during setup, failures, and attacks – the performance is approximately similar to symmetric key encryption. Adding devices to multiple groups would improve performance, but bypasses the bridge devices and undermines the distributed firewall. Non-privileged devices only need public key cryptography during setup and for voting. However, resource-constrained devices could rely on other devices to a limited degree. Bridges utilise public key cryptography (the IRS scheme) to add or remove devices but only one bridge of each IRS group needs to answer requests. Hence, these operations can be distributed.

Another concern is that one bridge device of group A and B has to stay online. Since fridge, smartphone, router, smart-meters, and smart thermostats are already online around-the-clock, this is less of an issue when smart devices are widespread.

Lastly, also old devices which do not support the protocol could be assigned to a (possibly static) group. GET TICKET could be adjusted to support legacy devices if A and B share their one-time passwords. To the legacy device, the group g appears as a network name and the full password is only a long password. Old devices can also communicate with the rest of the network without supporting the protocol because the SEND operation is transparent to the sender and receiver. While these old devices do not gain all the benefits, they can rely on other devices to verify and run the network.

6.3.2 Other Networks

After having seen how to modify mesh networks to be malware-tolerant, this section briefly evaluates other infrastructures.

6.3.2.1 By Topology

- Bus: Since broadcasts are cheap in bus networks and there are no single points-of-failure, the proposed isolation overlay would be more efficient. Bridge devices could be at any point in the network and only one of them has to answer since all others can identify the reply.

- Star: Star networks have a single point-of-failure in the middle which is a concern for security and reliability. Removing devices would partition the network, forcing the network to still rely on them to forward traffic. Replicating the centre in a wireless network would solve this issue. However, the resulting network is a hybrid network.

- Daisy chain (linear, ring): In linear networks every device is a single point-of-failure, thus these networks should be avoided from a security perspective. Ring networks always provide two paths from a device to another. Here, the introduced mesh architecture provides the additional benefit of hiding intra-group traffic from the other groups. Outvoting a device does not partition the network immediately, but one still needs to rely on it later to forward traffic.

- Hybrid: For hybrid networks, it depends on the specific layout if the malware-tolerant approach is useful. As those networks are similar to mesh networks, it is likely applicable.

6.3.2.2 By Network Types

- Home networks: These networks are usually star networks with the before mentioned points. But, newer developments like Google WiFi and smart-homes indicate that these network move to different topologies – especially mesh networks – for better connection.

- Business networks: Companies who split their networks into (hardware-based) zones have isolation already in place and, thus, do not need additional protection. Flat company networks on the other hand benefit from the introduced virtual network isolation mechanism.

- Embedded networks: Networks like the CAN bus are a bus network and are hence suited for the architecture.

- Internet: Even though creating a model of the infrastructure of the internet is difficult, it is a hybrid network. Despite this, the presented mesh architecture does not fit since some countries have a much higher number of devices than others. As a result, choosing bridge devices would be biased and would not work efficiently. Groups of people could also collude to vote against devices, enabling DoS. The assumption that the attacker only controls a small amount of devices is therefore false.

- Overlays: Virtual overlays like peer-to-peer networks are possible in combination with the proposed protocols but impose an additional overhead. Inside of the groups, some overlays might make sense but I doubt this would be practical.

I estimate that the mesh architecture is beneficial for interconnected, flat networks. It also improves networks where the virtual layout does not match the geographic one.

The architecture does not work for less connected networks with a lot of single points-of-failure. Subdivided networks, where virtual and geographic setup are equal, should employ isolation in hardware.

6.3.3 Improvements

Future work should focus on two areas. Firstly, the more expensive cryptography routines for GET TICKET, BRIDGE JOIN, and BRIDGE LEAVE could be improved. I imagine a combination of IRS and mRSA [19, 20]. In mRSA the private key is split into two additive shares, which could be split between bridge group A and B removing the need for two public keys. But, a new cryptographic scheme was beyond the scope of this research.

Secondly, the usability of the GET TICKET procedure needs improvements. Instead of having a two device authentication method employing two different devices, we could allow GET TICKET from one device (e.g. a smartphone) and store or re-compute old one-time passwords. If a password is used twice, either the setup device (the smartphone) or the new device (e.g. the new smart light bulb) misbehaved. At the moment, this would require to store all one-time passwords.

6.4 Related Work and Comparison

In the literature, mesh networks are usually analysed in their applied forms, i.e. Wireless Sensor Networks, Wireless Ad-Hoc Networks, Mobile Ad-Hoc Networks, and Vehicular Ad-Hoc Networks.

Diop et al. [43] deployed isolation to WSNs by utilising secret sharing and a network-wide symmetric key. The base station uses secret sharing to set up keys for clusters. Each head of a cluster derives the keys for the sensor nodes. Through this key infrastructure, sensor data is delivered via the cluster heads to the base station and isolation is achieved between the clusters. Diop et al. assume a trusted base station with an IDS, unlimited resources, and table of all nodes in the network. The base station is a single point-of-failure and the architecture is, thus, not malware-tolerant.

With SASHA, Bokareva et al. [18] propose a self-healing technique for sensor networks that is inspired by the immune system. The objectives of this work are automatic fault recognition, adaptive network monitoring, and a coordinated response. To detect faults, the network has a definition of itself in the form of a neural network. Base station, Thymus system, and Lymph system are single points-of-failure making the architecture vulnerable to targeted attacks.

Posh [41] is a proactive self-healing mechanism for WSNs. The sink periodically re-initialises sensor nodes with a new key and a secret seed. The nodes then also share some randomness with their neighbours to make it more difficult for the adversary to derive keys if nodes are compromised. Since the sink is in possession of all secrets, it can recompute the keys and read the data. However, the sink is a trusted entity and a single point-of-failure.

The authors [42] later improved their architecture with a moving target defence system. Data is moved either once or continuously, and then re-encrypted with one of three cryptographic schemes: symmetric encryption, symmetric encryption with key evolution, and asymmetric encryption. This way an adversary cannot easily delete data as the location of the data is unknown. But in this architecture, the sink is assumed trustworthy which is not malware-tolerant.

On the side of intrusion-tolerant [137] architectures, Sousa et al. [121] designed a distributed replication system built on top of *critical utility infrastructural resilience information switches*, a firewall device which they previously developed. The approach uses proactive and reactive recovery to self-heal failed replicas. The devices are rejuvenated periodically and upon detection of malicious or faulty behaviour. To agree on firewall decisions, the replicas communicate through a synchronous, trusted channel between the replicas called wormhole. If a majority of replicas approve a network message, it is signed with a key unknown to the replicas inside the wormhole subsystem. Local machines behind the firewall can verify the signature to identify trusted packages. The approach of Sousa et al. is similar to the one of this chapter but they deploy protection mechanisms in front of the network, instead of distributing the firewall over the network enforced through cryptography as we do. However, their technique applies self-healing while our malware-tolerant mesh architecture only automatically quarantines devices. Self-healing would be optionally possible since it is orthogonal to the infrastructure but there is likely no general purpose solution to securely rejuvenate consumer devices of different manufacturers.

In the research field of networks, there are also Software Defined Networks (SDNs) which are virtual overlays that can be used to guarantee security properties. However, usually the programming interface of SDN components is a single point-of-failure making the network vulnerable to targeted attacks. Shin et al. [117] created a robust and secure network operating system called Rosemary. It separates the SDN controller software from the network applications. Despite improving the resiliency of the network, it can only defend against certain malicious attacks and only isolates kernel from applications.

The *Splendid Isolation* approach of Gutz et al. [66] splits the SDN into network slices. To isolate slices from each other, they developed a special compiler mapping the formulated properties to Virtual Local Area Network (VLAN) tags. Additionally, the authors created a verification tool which checks the isolation properties using logic formulas.

Lastly, Jafarian et al. [76] used SDNs for moving target defence. They proactively changed the IP address of devices to defend against network scanners and worms. Each device gets a real IP and a virtual one. While the real IP stays the same, the virtual IP is transparently updated by SDN controllers. As a result, network scanners cannot reach hosts by IP reliably and worms cannot cooperatively scan the network to find new targets. In contrast to Jafarian et al. who alter IP addresses, I “move” the cryptographic key (i.e. the bridge keys) physically between devices. If an adversary manages to access real IP addresses of Jafarian et al., he could still scan effectively.

All three SDN approaches are vulnerable to targeted attacks on the programming interface of controllers and on network program or network operating system.

6.5 Summary

All in all, the presented architecture can provide isolation for flat, interconnected networks and networks where geographic and virtual layout differ. It enables them to automatically contain compromised devices while distributing trust over the entire architecture, i.e. malware tolerance.

This chapter demonstrated that the concept of malware tolerance is also suitable for consumer scenarios and general purpose networks. The principle is arguably more universally applicable than the initially examined scenarios of trusted input and ICSes.

CHAPTER 7

DISCUSSION OF THE MALWARE TOLERANCE CONCEPT

In the last chapters, I illustrated three techniques which give security guarantees even if an adversary controls some of the utilised devices. One or more of the devices can be compromised without bringing down the entire system.

Smart-guard demonstrated how to deliver input securely to a trusted receiver which can also be an enclave in a computer architecture like Intel SGX. The trusted input system is secure for up to two compromised devices of the total three devices. Its main advantage over existent techniques is that it also defends against hardware attacks but it only provides trusted input and no output.

The presented malware-tolerant ICS architecture remains operational if one of three PLCs is compromised. Additional security is provided by the hardware-based self-healing mechanism which can recover one compromised PLC at the cost of a restart. To complement this, I developed a software self-healing algorithm built on top of ARM TrustZone to overcome user-level attacks without a restart. While this approach provides strong security properties and can recover devices which broke through updates, it comes at the potential cost of additional hardware (if not already existent).

Lastly, we saw how an entire network, in this case a mesh network, can tolerate attacks by using a dynamic moving-target approach. Devices were clustered into groups and each group elected its representatives (bridges) for inter-group communication. Misbehaving devices can be quarantined upon detection. The network tolerates up to one compromised device per group and half of them can also be privileged bridge devices. The cryptographic implementation imposes an overhead but no additional hardware is necessary.

7.1 Evaluation of Malware Tolerance

Research Questions Revisited: The aim of this research was to show the possibility to build systems malware-tolerant, i.e. in a way that no single component can tamper with the resources. Three scenarios demonstrated malware tolerance with each of them allowing (at least) one device being compromised. Even if the adversary has full control over this one device, the rest of the architecture continues to work securely. In several cases, this was also possible for more compromised devices.

Essential to achieve malware tolerance by distribution are independent components. In multi device scenarios like networks this is already the case. Single points-of-failure are a limiting factor, if they cannot be replaced by multiple components.

To evaluate architectures towards their potential to tolerate malware, I developed the multiple TCB model. The model is compatible with protocol verifiers like ProVerif and enables semi-automatic proofs for malware-tolerant architectures.

I do not have the capabilities to produce a full working product at the university and, hence, only provided a proof-of-concept implementation and ProVerif proof scripts. However, this work shows that systems can indeed be designed in a way without a single component or device being able to successfully tamper with the resources. This was achieved by distributing trust over multiple devices and proved the hypothesis possible.

During the research on malware tolerance, it became evident that in some scenarios it is even possible to automatically contain and recover compromised devices. It is sometimes called *self-healing.* Although this thesis did not formulate any research questions regarding the field of self-healing, it is the logical next step to malware tolerance.

Comparison to Previous Research: The findings of this research support the upper bound of multi-party computation [36] where in general 33% of the total architecture can be malicious. The presented trusted input system and the malware-tolerant ICS, both, are secure for one of three or 33% of the devices being malicious. The mesh network could in a special case (groups of only 2 devices) reach up to 50%, but three to ten devices per group are more accurate, corresponding to 10-33%.

It was unexpected that in certain cases the 33% boundary can be exceeded. However, these were special cases and the general rule of a maximum of 33% malicious devices still applies. The practical results of this work, thus, support the theoretical model as well as the practical evaluations of distributed systems [139, 6] (f malicious replicas of $3f + 1$) and intrusion tolerance [136, 121] (f malicious replicas of $3f + 1$ or $2f + k + 1$ with k systems recovering).

Multiple Trusted Computing Bases Model: Modelling architectures on basis of multiple additive TCBs proved to be an effective representation. It can be mapped to formal proof systems, like Horn clauses, without effort and made it possible to represent the approaches in ProVerif.

As first step, I informally analysed the TCBs of the techniques and expressed them with the model. It was unexpected that evaluating single points-of-failure would accurately identify vulnerabilities. When the architectures were later translated to ProVerif proofs, the protocols passed formal verification without major adjustments and showed that the representation through TCBs is a suitable estimation.

7.2 Limitations and Future Work

While it is the primary advantage of malware and intrusion tolerance to overcome attacks, it is similarly the main limitation. The upper bound of malicious devices which can potentially be tolerated only exceeds 33% in special cases.

Another restricting factor is performance and cost. While it is known that cryptography lowers performance, many vendors are not willing to pay for additional devices. However, only high risk infrastructures need this level of security justifying the costs. In other scenarios a virtualised malware-tolerant architecture might be applicable. With cryptographic co-processors becoming more widespread, the impact on performance is feasible and expected to decrease further.

Future work should improve performance and cost of redundancy for malware tolerance. E.g. the redundancy of having two cryptographic IRS keys for the malware-tolerant mesh network (Chapter 6) could be replaced by developing a cryptographic scheme similar to mRSA with evolving keys. Employed in the proposed mesh network architecture, this would combine two keys while maintaining the same security level. However, a new cryptographic scheme (including its proof) was beyond this work.

Furthermore, I would like to explore virtualised malware tolerance. Especially in consumer scenarios, redundancy in hardware might not be necessary. Software techniques could make malware tolerance more feasible for vendors.

All in all, the principle of malware (and intrusion) tolerance as well as self-healing can vastly improve security when building systems. We should abandon the idea of invulnerable components in security and incorporate security into the design.

Analysing approaches from a (no) single point-of-failure perspective reveals weak points which can then be removed before production. I recommend applying this way of thinking even if malware tolerance is not used.

CHAPTER 8

CONCLUSION

The aim of this work was to examine if we can build systems tolerant to malware by distributing trust over several independent components. That means one (or sometimes more) components can be compromised without bringing down the system. This thesis calls this new concept *malware tolerance*. The goal of a malware-tolerant architecture is to remove all single points-of-failure where necessary.

The idea was demonstrated through three techniques: a trusted input system, an ICS, and a mesh network architecture. All presented techniques are secure even if the adversary controls one device and in certain cases even more. This is also confirmed through the research of Verissimo et al. on intrusion tolerance [137, 138, 16, 121].

Multi-party computation established that in general 66% of an architecture has to be trustworthy to tolerate malicious attacks. This is the main limitation of malware- and intrusion-tolerance.

It could be shown and proven that removing single points-of-failure at vulnerable intersection points improves security. The security could even be expressed in numbers: The adversary has to compromise twice as many systems as before to still succeed.

For general security related work I, therefore, recommend to *actively* identify single points-of-failure in an architecture and consider distributing their assets securely among several devices – malware tolerance. The illustrated concept highlights security deficits and enables better system design.

BIBLIOGRAPHY

[1] Marshall Abrams and Joe Weiss. Malicious Control System Cyber Security Attack Case Study - Maroochy Water Services, Australia. Technical report, NIST, July, 2008. URL `http://csrc.nist.gov/groups/SMA/fisma/ics/documents/Maroochy-Water-Services-Case-Study_report.pdf`. accessed: 2016-07-27.

[2] Manal Adham, Amir Azodi, Yvo Desmedt, and Ioannis Karaolis. How to attack two-factor authentication internet banking. In *Financial Cryptography and Data Security*, pages 322–328. Springer, 2013. URL `http://link.springer.com/chapter/10.1007/978-3-642-39884-1_27`. accessed: 2014-06-26.

[3] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium (CSF)*, volume 25, pages 171–185. IEEE, 2012. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6266159`. accessed: 2014-07-24.

[4] Bas Alberts. DR Linux 2.6 rootkit released. Online, September 2008. URL `http://seclists.org/dailydave/2008/q3/215`. accessed: 2015-02-16.

[5] Yair Amir, Yongdae Kim, Cristina Nita-Rotaru, and Gene Tsudik. On the performance of group key agreement protocols. *ACM Transactions on Information and System Security (TISSEC)*, 7(3):457–488, 2004. URL `http://dl.acm.org/citation.cfm?id=1015045`. accessed: 2017-04-04.

[6] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):

564–577, 2011. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5654509&tag=1. accessed: 2016-04-25.

[7] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for CPU based attestation and sealing. In *2nd Intl. Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013. URL https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing. accessed: 2014-04-17.

[8] Anonymous. How the coffee-machine took down a factories control room. online, Jul 2017. URL https://amp.reddit.com/r/talesfromtechsupport/comments/6ovy0h/how_the_coffeemachine_took_down_a_factories/. accessed: 2017-08-03.

[9] ARM. Building a Secure System using TrustZone Technology. Technical report, ARM Security Technology, April 2009. URL http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. accessed: 2015-02-26.

[10] ARM. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.* ARM Limited, 110 Fulbourn Road, Cambridge, England CB1 9NJ, v8 edition, December 2013. URL https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet3/DDI0487A_b_armv8_arm.pdf. accessed: 2014-06-24.

[11] ARM. ARM TrustZone Website. Online, 2014. URL http://www.arm.com/products/processors/technologies/trustzone/index.php?tab=Hardware+Architecture. accessed: 2014-06-24.

[12] ARM. ARM Information Center. online, 2015. URL http://infocenter.arm.com. accessed: 2017-12-03.

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell,

et al. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, pages 689–703, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf. accessed: 2017-11-23.

[14] Eric Baize. Developing secure products in the age of advanced persistent threats. *IEEE Security & Privacy*, 10(3):88–92, 2012. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6226545. accessed: 2015-02-13.

[15] Gal Beniamini. Trust Issues: Exploiting TrustZone TEEs. Technical report, Google Project Zero, July 2017. URL https://googleprojectzero.blogspot.co.uk/2017/07/trust-issues-exploiting-trustzone-tees.html. accessed: 2018-05-24.

[16] Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy*, 6(6):44–51, 2008. URL http://ieeexplore.ieee.org/abstract/document/4753673/. accessed: 2017-03-03.

[17] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343. Springer, 2009. URL http://eprint.iacr.org/2008/068.pdf. accessed: 2015-08-17.

[18] Tatiana Bokareva, Nirupama Bulusu, and Sanjay Jha. Sasha: Toward a self-healing hybrid sensor network architecture. In *The Second IEEE Workshop on Embedded Networked Sensors*, pages 71–78. Citeseer, 2005. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.8588&rep=rep1&type=pdf. accessed: 2016-03-14.

[19] Dan Boneh, Xuhua Ding, Gene Tsudik, and Chi-Ming Wong. A Method for Fast Revocation of Public Key Certificates and Security Capabilities. In *USENIX Secu-*

rity Symposium, 2001. URL http://static.usenix.org/publications/library/proceedings/sec01/full_papers/boneh/boneh.ps. accessed: 2015-02-17.

[20] Dan Boneh, Xuhua Ding, and Gene Tsudik. Identity-based mediated RSA. In *3rd Workshop on Information Security Application* Boneh et al. [19]. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.1676&rep=rep1&type=pdf. accessed: 2015-08-27.

[21] Danny Bradbury. Shadows in the cloud: Chinese involvement in advanced persistent threats. *Network Security, Elsevier*, 2010(5):16–19, 2010. URL http://www.sciencedirect.com/science/article/pii/S1353485810700581. accessed: 2015-02-13.

[22] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Middleware*, 2016. URL https://lsds.doc.ic.ac.uk/sites/default/files/securekeeper-middleware16.pdf. accessed: 2017-11-24.

[23] BSI. Die Lage der IT-Sicherheit in Deutschland 2014. Technical report, Bundesamt für Sicherheit in der Informationstechnik, 2014. URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2014.pdf. accessed: 2016-07-27.

[24] Xia Cai, Michael R Lyu, and Mladen A Vouk. An experimental evaluation on reliability features of N-version programming. In *16th IEEE Intl. Symposium on Software Reliability Engineering (ISSRE'05)*. IEEE, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1544731. accessed: 2016-09-26.

[25] Seyit A Camtepe and Bülent Yener. Key distribution mechanisms for wireless sensor networks: a survey. *Rensselaer Polytechnic Institute, Troy, New York, Technical Report*, pages 5–7, 2005. URL http://www.cs.rpi.edu/research/pdf/05-07.pdf. accessed: 2017-02-24.

[26] Alvaro A Cárdenas, Saurabh Amin, and Shankar Sastry. Research Challenges for the Security of Control Systems. In *HotSec*, 2008. URL `https://www.usenix.org/legacy/events/hotsec08/tech/full_papers/cardenas/cardenas_html/hotsecHTML.html`. accessed: 2015-08-21.

[27] Alvaro A Cardenas, Saurabh Amin, and Shankar Sastry. Secure control: Towards survivable cyber-physical systems. In *The 28th Intl. Conf. on Distributed Computing Systems Workshops* Cárdenas et al. [26], pages 495–500. URL `http://feihu.eng.ua.edu/NSF_CPS/year1/w8_1.pdf`. accessed: 2015-12-01.

[28] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *16th Intl. Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5416657`. accessed: 2014-07-24.

[29] Yueqiang Cheng, Xuhua Ding, and Robert H Deng. DriverGuard: A fine-grained protection on I/O flows. In *Computer Security–ESORICS*, pages 227–244. Springer, Berlin, Heidelberg, 2011. URL `http://link.springer.com/chapter/10.1007/978-3-642-23822-2_13`. accessed: 2014-11-24.

[30] Joris Claessens, Valentin Dem, Danny De Cock, Bart Preneel, and Joos Vandewalle. On the security of today's online electronic banking systems. *Computers & Security - Elsevier*, 21(3):253–265, 2002. URL `http://www.sciencedirect.com/science/article/pii/S0167404802003127`. accessed: 2014-06-25.

[31] Gregory Conti, Tom Cross, and David Raymond. Pen Testing a City. In *Black Hat USA*, 2015. URL `https://www.blackhat.com/docs/us-15/materials/us-15-Conti-Pen-Testing-A-City-wp.pdf`. accessed: 2015-08-12.

[32] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive: Report 2016/086, Jan 2016. URL `https://eprint.iacr.org/2016/086.pdf`. accessed: 2016-02-02.

[33] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Usenix Security*, volume 6, pages 105–120, 2006. URL `http://static.usenix.org/event/sec06/tech/full_papers/cox/cox.pdf`. accessed: 2016-09-26.

[34] Ronald Cramer, Ivan Damgard, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing - An Information Theoretic Appoach (Book Draft)*. Cambridge University Press, 1 edition, July 2015. URL `http://www.cs.au.dk/~ivan/MPCbook.pdf`. accessed: 2015-09-07.

[35] CRoCS MU. JCAlgTest Comparative table. online, 2018. URL `https://www.fi.muni.cz/%7Exsvenda/jcalgtest/comparative-table.html`. accessed: 2018-05-29.

[36] I. Damgard. Security of Multi Party Computation. Invited talk at FOSAD 2015, August 2015. URL `http://www.sti.uniurb.it/events/fosad15/damgard.ppt`. accessed: 2017-12-08.

[37] Don Davis. Defective Sign & Encrypt in S/MIME, PKCS# 7, MOSS, PEM, PGP, and XML. In *USENIX Annual Technical Conf., General Track*, pages 65–78, 2001. URL `http://static.usenix.org/publications/library/proceedings/usenix01/full_papers/davis/davis_html/`. accessed: 2017-07-26.

[38] M. Denzel, A. Bruni, and M. D. Ryan. Smart-Guard: Defending User Input from Malware. In *Intl IEEE Conf. on Advanced and Trusted Computing*, pages 502–509. IEEE, 2016. doi: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.97. URL `http://ieeexplore.ieee.org/abstract/document/7816885/`. accessed: 2017-09-15.

[39] M. Denzel, M. Ryan, and E. Ritter. A Malware-Tolerant, Self-Healing Industrial Control System Framework. In *IFIP Advances in Information and Communication Technology ICT Systems Security and Privacy Protection*, volume 502. Springer,

2017. doi: https://doi.org/10.1007/978-3-319-58469-0_4. URL https://link.springer.com/chapter/10.1007%2F978-3-319-58469-0_4. accessed: 2017-07-23.

[40] Yves Deswarte, Laurent Blain, and J-C Fabre. Intrusion tolerance in distributed computing systems. In *Computer Society Symposium on Research in Security and Privacy*, pages 110–121. IEEE, 1991. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=130780. accessed: 2014-04-08.

[41] Roberto Di Pietro, Di Ma, Claudio Soriente, and Gene Tsudik. Posh: Proactive co-operative self-healing in unattended wireless sensor networks. In *IEEE Symposium on Reliable Distributed Systems*, pages 185–194. IEEE, 2008. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4690813. accessed: 2017-01-11.

[42] Roberto Di Pietro, Luigi V Mancini, Claudio Soriente, Angelo Spognardi, and Gene Tsudik. Playing hide-and-seek with a focused mobile adversary in unattended wireless sensor networks. *Ad Hoc Networks (Elsevier)*, 7(8):1463–1475, 2009. URL http://www.sciencedirect.com/science/article/pii/S1570870509000341. accessed: 2017-01-11.

[43] Abdoulaye Diop, Yue Qi, and Qin Wang. Efficient group key management using symmetric key and threshold cryptography for cluster based wireless sensor networks. *Intl. Journal of Computer Network and Information Security*, 6(8):9, 2014. URL http://www.mecs-press.org/ijcnis/ijcnis-v6-n8/IJCNIS-V6-N8-2.pdf. accessed: 2017-02-17.

[44] Yevgeniy Dodis, Matt Franklin, Jonathan Katz, Atsuko Miyaji, and Moti Yung. Intrusion-resilient public-key encryption. In *Cryptographers' Track at the RSA Conf.*, pages 19–32. Springer, 2003. URL http://link.springer.com/chapter/10.1007/3-540-36563-X_2. accessed: 2017-04-11.

[45] Yevgeniy Dodis, Matt Franklin, Jonathan Katz, Atsuko Miyaji, and Moti Yung. A generic construction for intrusion-resilient public-key encryption. In *Cryptographers'*

Track at the RSA Conf. Dodis et al. [44], pages 81–98. URL http://link.springer.com/chapter/10.1007/978-3-540-24660-2_7. accessed: 2017-04-08.

[46] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Transactions on Information Theory, IEEE*, 29(2):198–208, 1983. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1056650. accessed: 2014-11-24.

[47] David Drummond. A new approach to China. online (official Google blog), January 2010. URL https://googleblog.blogspot.co.uk/2010/01/new-approach-to-china.html. accessed: 2016-07-27.

[48] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM rootkit: A new breed of os independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013. URL http://onlinelibrary.wiley.com/doi/10.1002/sec.166/full. accessed: 2017-11-22.

[49] Ariel J Feldman, J Alex Halderman, and Edward W Felten. Security Analysis of the Diebold AccuVote-TS Voting Machine. In *USENIX Electronic Voting Technology Workshop*, 2006. URL https://www.usenix.org/event/evt07/tech/full_papers/feldman/feldman_html/.

[50] Norman Feske. *GENODE Operating System Framework 15.05.* Genode Labs, 2015. URL http://genode.org/documentation/genode-foundations-15-05.pdf. accessed: 2015-12-04.

[51] Norman Feske and Christian Helmuth. An Exploration of ARM TrustZone Technology. Online, 2014. URL http://genode.org/documentation/articles/trustzone. accessed: 2015-02-25.

[52] Andrew Fielder, Tingting Li, and Chris Hankin. Defense-in-depth vs. Critical Component Defense for Industrial Control Systems. In *ICS-CSR*, 2016. URL https://www.researchgate.net/profile/Tingting_Li20/publication/306056457_Defen

se-in-depth_vs_Critical_Component_Defense_for_Industrial_Control_Systems/links/57d1434c08ae5f03b48a74cc/Defense-in-depth-vs-Critical-Component-Defense-for-Industrial-Control-Systems.pdf. accessed: 2017-12-02.

[53] Atanas Filyanov, Jonathan M McCune, A-R Sadeghiz, and Marcel Winandy. Unidirectional trusted path: Transaction confirmation on just one device. In *IFIP 41st Intl. Conf. on Dependable Systems & Networks (DSN)*, pages 1–12. IEEE, 2011. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958202. accessed: 2014-11-25.

[54] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. The ANDIX Research OS-ARM TrustZone Meets Industrial Control Systems Security. *IEEE Intl. Conf. on Industrial Informatics (INDIN)*, April 2015. URL http://www.arrowhead.eu/wp-content/uploads/2013/03/TUG_Fitzek_INDIN2015.pdf. accessed: 2015-11-18.

[55] Igor Nai Fovino, Andrea Carcano, Marcelo Masera, and Alberto Trombetta. Design and implementation of a secure modbus protocol. In *Intl. Conf. on Critical Infrastructure Protection*, pages 83–96. Springer, 2009. URL http://link.springer.com/chapter/10.1007/978-3-642-04798-5_6. accessed: 2016-07-27.

[56] Matt Franklin. A survey of key evolving cryptosystems. *Intl. Journal of Security and Networks*, 1(1-2):46–53, 2006. URL http://web.cs.ucdavis.edu/~franklin/ecs228/pubs/extra_pubs/key_evolve_survey.pdf. accessed: 2017-04-08.

[57] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *IFIP 41st Intl. Conf. on Dependable Systems & Networks (DSN)*, pages 383–394. IEEE, 2011. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958251. accessed: 2014-04-08.

[58] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro.

Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44:735–770, January 2013. doi: 10.1002/spe.2180. URL http://onlinelibrary.wiley.com/doi/10.1002/spe.2180/full. accessed: 2014-04-08.

[59] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003. URL http://dl.acm.org/citation.cfm?doid=1165389.945464. accessed: 2014-04-17.

[60] Genode Labs. Genode Operating System Framework. Online, 2015. URL http://genode.org. accessed: 2015-08-27.

[61] Debanjan Ghosh, Raj Sharman, H Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4): 2164–2185, 2007. URL http://www.sciencedirect.com/science/article/pii/S0167923606000807. accessed: 2016-01-12.

[62] Giesecke And Devrient. StarSign Crypto USB Token. Technical report, Giesecke And Devrient, July 2014. URL http://www.gi-de.com/gd_media/media/en/documents/brochures/mobile_security_2/nb/StarSign_Crypto_USB_Token.pdf. accessed: 2014-08-21.

[63] Giesecke And Devrient. StarSign Mobile Security Card SE 1.2. Technical report, Giesecke And Devrient, April 2014. URL http://www.gi-de.com/gd_media/media/documents/brochures/mobile_security_2/nb/StarSign_Mobile_Security_Card_SE_12.pdf. accessed: 2014-08-21.

[64] Jamie Grierson and Samuel Gibbs. NHS cyber-attack causing disruption one week after breach. *The Guardian*, May 2017. URL https://www.theguardian.com/society/2017/may/19/nhs-cyber-attack-ransomware-disruption-breach. accessed: 2017-11-22.

[65] Julian Bennett Grizzard. *Towards self-healing systems: re-establishing trust in compromised systems.* PhD thesis, Georgia Institute of Technology, 2006. URL `https://pdfs.semanticscholar.org/7ee0/f3f20b2f7820e404c668e847f2b1eb700c52.pdf`. accessed: 2017-02-14.

[66] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 79–84. ACM, 2012. URL `http://dl.acm.org/citation.cfm?id=2342458`. accessed: 2017-04-19.

[67] John Heasman. Implementing and Detecting an ACPI BIOS Rootkit. *Black Hat Federal*, 368, 2006. URL `https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf`. accessed: 2017-11-22.

[68] John Heasman. Implementing and Detecting a PCI Rootkit. *Black Hat Federal*, 2006. URL `https://packetstorm.foofus.com/papers/general/Implementing_And_Detecting_A_PCI_Rootkit.pdf`. accessed: 2017-11-22.

[69] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *2nd Intl. Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013. URL `https://software.intel.com/sites/default/files/article/413938/hasp-2013-innovative-instructions-for-trusted-solutions.pdf`. accessed: 2014-04-17.

[70] Homeland Security. Recommended Practice: Improving Industrial Control Systems Cybersecurity with Defense-In-Depth Strategies. Technical report, U.S. Homeland Security, October 2009. URL `https://ics-cert.us-cert.gov/sites/default/files/recommended_practices/Defense_in_Depth_Oct09.pdf`. accessed: 2015-12-10.

[71] Yiguang Hong, Dong Chen, Lei Li, and Kishor S. Trivedi. Closed loop design for software rejuvenation. In *Workshop on Self-Healing, Adaptive, and Self-Managed Systems*, 2002. URL `http://shannon.ee.duke.edu/Rejuv/shaman-aging.ps`. accessed: 2016-02-05.

[72] Yih-Chun Hu and Adrian Perrig. A survey of secure wireless ad hoc routing. *IEEE Security & Privacy*, 2(3):28–39, 2004. URL `http://robotics.eecs.berkeley.edu/~sastry/pubs/PDFs%20of%20Pubs2000-2005/Pdfs%20of%20Misc.Others/Hu,Jih%20Chun/YihChunHU.ISnP.pdf`. accessed: 2016-10-18.

[73] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1:80, 2011. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.301.2845&rep=rep1&type=pdf#page=123`. accessed: 2015-08-11.

[74] Gene Itkis. Intrusion-resilient signatures: Generic constructions, or defeating strong adversary with minimal assumptions. In *Intl. Conf. on Security in Communication Networks* Itkis and Reyzin [75], pages 102–118. URL `http://link.springer.com/chapter/10.1007/3-540-36413-7_8`. Accessed: 2017-04-10.

[75] Gene Itkis and Leonid Reyzin. SiBIR: Signer-base intrusion-resilient signatures. *Advances in Cryptology–Crypto 2002*, pages 101–116, 2002. URL `http://www.springerlink.com/index/JD2TTH28DR6J6YY3.pdf`. Accessed: 2017-04-10.

[76] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *roceedings of the first workshop on Hot topics in software defined networks*, pages 127–132. ACM, 2012. URL `http://dl.acm.org/citation.cfm?id=2342467`. accessed: 2017-07-26.

[77] James E Just, James C Reynolds, Larry A Clough, Melissa Danforth, Karl N Levitt,

Ryan Maglich, and Jeff Rowe. Learning unknown attacks - A start. In *Recent Advances in Intrusion Detection*, pages 158–176. Springer, 2002. URL http://link.springer.com/chapter/10.1007/3-540-36084-0_9. accessed: 2014-04-08.

[78] Kaspersky. Teamwork: How the ZitMo Trojan Bypasses Online Banking Security. Online, October 2011. URL http://www.kaspersky.com/about/news/virus/2011/Teamwork_How_the_ZitMo_Trojan_Bypasses_Online_Banking_Security. accessed: 2015-02-16.

[79] Kaspersky Lab. Empowering Industrial Cyber Security. Technical report, Kaspersky Lab, 2015. URL http://media.kaspersky.com/en/business-security/Empowering%20Industrial%20Cyber%20Security_web.pdf. accessed: 2016-02-10.

[80] Michael Kassner. KeyScrambler: How keystroke encryption works to thwart keylogging threats. Online, October 2010. URL http://www.techrepublic.com/blog/it-security/keyscrambler-how-keystroke-encryption-works-to-thwart-keylogging-threats/. accessed: 2015-05-25.

[81] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. In *USENIX Security Symposium*, volume 24, page 173, 2007. URL http://static.usenix.org/event/sec07/tech/full_papers/kauer/kauer_html/. accessed: 2014-04-17.

[82] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*, volume 2. Springer Science & Business Media, 2011.

[83] Liwei Kuang and Mohammad Zulkernine. An Intrusion-Tolerant Mechanism for Intrusion Detection Systems. In *Availability, Reliability and Security*, pages 319–326. IEEE, 2008. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4529353. accessed: 2014-09-03.

[84] Christopher Kugler and Tilo Müller. SCADS Separated Control- and Data-Stacks. In *Intl. Conf. on Security and Privacy in Communication Systems*, pages 323–340.

Springer, 2014. URL https://www1.cs.fau.de/filepool/scads/scads-securecomm2014.pdf. accessed: 2016-10-14.

[85] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. In *Security & Privacy, IEEE* Langner [86], pages 49–51. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5772960. accessed: 2014-08-06.

[86] Ralph Langner. To Kill a Centrifuge. Technical report, Langner Group, Nov 2013. URL http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf. accessed: 2016-03-21.

[87] Ruby B Lee, Peter CS Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *32nd Intl. Symposium on Computer Architecture (ISCA)*, pages 2–13. IEEE, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1431541. accessed: 2014-07-24.

[88] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Technical Conf.*, 2014. URL https://www.usenix.org/system/files/conference/atc14/atc14-paper-li_yanlin.pdf. accessed: 2018-02-19.

[89] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown and Spectre. online, Jan 2018. URL https://meltdownattack.com/. accessed: 2018-05-24.

[90] Jonathan M McCune, Adrian Perrig, and Michael K Reiter. Bump in the ether: A framework for securing sensitive user input. In *USENIX Annual Technical Conf.*, pages 17–17, 2006. URL http://dl.acm.org/citation.cfm?id=1267376. accessed: 2014-08-12.

[91] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008. URL `http://dl.acm.org/citation.cfm?id=1352625`. accessed: 2014-04-30.

[92] Jonathan M McCune, Adrian Perrig, and Michael K Reiter. Safe Passage for Passwords and Other Sensitive Data. In *16th Annual Network and Distributed System Security Symposium (NDSS)*, February 2009. URL `http://www.cs.unc.edu/~reiter/papers/2009/NDSS.pdf`. accessed: 2014-11-26.

[93] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Symposium on Security and Privacy (SP)*, pages 143–158. IEEE, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5504713`. accessed: 2014-08-12.

[94] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP*, 133:10, 2013. URL `http://css.csail.mit.edu/6.858/2013/readings/intel-sgx.pdf`. accessed: 2014-04-17.

[95] Alexander Meduna, Lukas Vrabel, and Petr Zemek. Converting Finite Automata to Regular Expressions. online, 2012. URL `http://www.fit.vutbr.cz/~izemek/grants.php.cs?file=%2Fproj%2F589%2FPresentations%2FPB05-Converting-FAs-To-REs.pdf&id=589`. accessed: 2016-03-14.

[96] Steven J Murdoch. Introduction to Trusted Execution Environments (TEE). Online, 2014. URL `https://www.cl.cam.ac.uk/~sjm217/talks/rhul14tee.pdf`. accessed: 2014-04-17.

[97] National Institute of Standards and Technology. Framework for Improving Critical Infrastructure Cybersecurity. *NIST Special Publication*, pages 1–61, January

2017. URL https://www.nist.gov/sites/default/files/documents////draft-cybersecurity-framework-v1.11.pdf. accessed: 2017-08-20.

[98] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX Security*, pages 479–494, 2013. URL https://www.usenix.org/sites/default/files/conference/protected-files/noorman_sec13_slides.pdf. accessed: 2014-07-24.

[99] Job Noorman, Jo van Bulck, Jan Tobias Muehlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Goetzfried, Tilo Mueller, and Felix Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. online, 2017. URL https://www.esat.kuleuven.be/cosic/publications/article-2785.pdf. accessed: 2017-11-15.

[100] Daniel Paillet. Defending against cyber threats to building management systems. online (FM Magazine), April 2016. URL https://www.fmmagazine.com.au/sectors/defending-against-cyber-threats-to-building-management-systems/. accessed: 2016-08-05.

[101] Bryan Parno, Adrian Perrig, and Virgil Gligor. Distributed detection of node replication attacks in sensor networks. In *IEEE Symposium on Security and Privacy*, pages 49–63. IEEE, 2005. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1425058. accessed: 2017-01-11.

[102] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Symposium on Security and Privacy (SP)*, pages 379–394. IEEE, 2011. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958041. accessed: 2014-08-12.

[103] Marco Patrignani, Dave Clarke, and Frank Piessens. Secure compilation of object-

oriented components to protected module architectures. In *Programming Languages and Systems*, pages 176–191. Springer, 2013. URL http://link.springer.com/chapter/10.1007/978-3-319-03542-0_13. accessed: 2014-07-24.

[104] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. In *ACM Transactions on Programming Languages and Systems (TOPLAS)* Patrignani et al. [103], page 6. URL http://dl.acm.org/citation.cfm?id=2699503. accessed: 2017-11-15.

[105] Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, 2004. URL http://dl.acm.org/citation.cfm?id=990707. accessed: 2016-11-24.

[106] Martin Pirker and Daniel Slamanig. A framework for privacy-preserving mobile payment on security enhanced ARM TrustZone platforms. In *11th Intl. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1155–1160. IEEE, 2012. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6296107. accessed: 2015-02-17.

[107] Marco Platania, Daniel Obenshain, Thomas Tantillo, Ritu Sharma, and Yair Amir. Towards a practical survivable intrusion tolerant replication system. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd Intl. Symposium on* Amir et al. [6], pages 242–252. URL http://www.cnds.jhu.edu/pub/papers/cnds-2014-1.pdf. accessed: 2016-04-25.

[108] QFX software. How KeyScrambler Works. Online, 2015. URL https://www.qfxsoftware.com/ks-windows/how-it-works.htm. accessed: 2015-05-25.

[109] Joanna Rutkowska, Marek Marczykowski, and Wojciech Porczyk. Welcome to the Qubes OS Project. Online, October 2015. URL https://www.qubes-os.org/. accessed: 2015-10-12.

[110] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, volume 13, pages 16–33, 2004. URL `https://www.usenix.org/legacy/events/sec04/tech/full_papers/sailer/sailer.pdf`. accessed: 2014-04-17.

[111] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2008* Cox et al. [33], pages 843–848. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4606777`. accessed: 2016-09-28.

[112] Falk Salewski, Dirk Wilking, and Stefan Kowalewski. The effect of diverse hardware platforms on n-version programming in embedded systems-an empirical evaluation. In *Proc. of the 3rd. Workshop on Dependable Embedded Sytems (WDES)*, volume 105, pages 61–66. Citeseer, 2006. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.7705&rep=rep1&type=pdf#page=61`. accessed: 2016-09-26.

[113] Craig Schmugar. More Details on "Operation Aurora". online (McAfee), January 2010. URL `https://blogs.mcafee.com/mcafee-labs/more-details-on-operation-aurora/`. accessed: 2016-07-27.

[114] Seal One AG. Seal One USB - Offering simplicity and elegance. Online, 2013. URL `http://www.seal-one.com/products-list.en-UK.html`. accessed: 2014-08-21.

[115] R Sekar, Ajay Gupta, James Frullo, Tushar Shanbhag, Abhishek Tiwari, Henglin Yang, and Sheng Zhou. Specification-based anomaly detection: A new approach for detecting network intrusions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 265–274. ACM, 2002. URL `http://dl.acm.org/citation.cfm?id=586146`. accessed: 2016-08-14.

[116] Di Shen. Exploiting TrustZone on Android. In *Black Hat USA*, 2015.

URL https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf. accessed: 2015-08-12.

[117] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A Robust, Secure, and High-Performance Network Operating System. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 78–89. ACM, 2014. URL http://dl.acm.org/citation.cfm?id=2660353. accessed: 2018-01-04.

[118] Atul Singh, Nishant Sinha, and Nitin Agrawal. Avatars for pennies: Cheap n-version programming for replication. In *6th Workshop on Hot Topics in System Dependability*, 2010. URL https://www.usenix.org/events/hotdep10/tech/full_papers/Singh.pdf. accessed: 2016-09-26.

[119] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Dependable Computing*, pages 373–380. IEEE, 2007. URL http://ieeexplore.ieee.org/abstract/document/4459685/. accessed: 2018-02-19.

[120] Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. Hidden problems of asynchronous proactive recovery. In *Proc. of the Workshop on Hot Topics in System Dependability (HotDep07)*, 2007. URL http://www.academia.edu/download/30840447/paper10.pdf. accessed: 2017-03-03.

[121] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010. URL http://ieeexplore.ieee.org/abstract/document/5010435/. accessed: 2017-03-03.

[122] Keith Stouffer, Suzanne Lightman, Victoria Pillitteri, Mashrall Abrams, and Adam Hahn. Guide to Industrial Control Systems (ICS) security. *NIST Special Publication*, 800(82):1–255, May 2014. URL http://www.gocs.com.de/pages/fachberichte/archiv/164-sp800_82_r2_draft.pdf. accessed: 2015-08-21.

[123] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Conf. on Computer and Communications Security*, pages 2–13. ACM, 2012. URL http://dl.acm.org/citation.cfm?id=2382200. accessed: 2014-07-24.

[124] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003. URL http://dl.acm.org/citation.cfm?id=782838. accessed: 2014-07-24.

[125] Petr et al. Svenda. JCAlgTest. online, 2018. URL https://github.com/crocs-muni/JCAlgTest. accessed: 2018-05-29.

[126] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. In *ACM SIGARCH Computer Architecture News* Lee et al. [87], pages 437–450. URL http://dl.acm.org/citation.cfm?id=2151022. accessed: 2014-07-24.

[127] Colin Tankard. Advanced Persistent Threats and how to monitor and deter them. *Network security, Elsvevier*, 2011(8):16–19, 2011. URL http://www.sciencedirect.com/science/article/pii/S1353485811700861. accessed: 2015-02-13.

[128] Gordon Thomson. APTs: A poorly understood challenge. *Network Security, Elsevier*, 2011(11):9–11, 2011. URL http://www.sciencedirect.com/science/article/pii/S1353485811701180. accessed: 2015-02-13.

[129] Eric Totel, Frédéric Majorczyk, and Ludovic Mé. COTS diversity based intrusion detection and application to web servers. In *Intl. Workshop on Recent Advances in*

Intrusion Detection, pages 43–62. Springer, 2005. URL http://link.springer.com/chapter/10.1007/11663812_3. accessed: 2017-03-06.

[130] Huan-yu Tu. Comparisons of self-healing fault-tolerant computing schemes. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 1. Citeseer, 2010. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.445.3617&rep=rep1&type=pdf. accessed: 2016-03-08.

[131] Tyfone. Secure Transaction Platform. Online, 2004. URL http://tyfone.com/Platform.php. accessed: 2014-08-21.

[132] Roland van Rijswijk-Deij and Erik Poll. Using Trusted Execution Environments in Two-factor Authentication: Comparing Approaches. In *Open Identity Summit, OID*, volume P-223 of *Lecture notes in informatics*, pages 20–31, Bonn, Germany, September 2013. URL http://doc.utwente.nl/91957/. accessed: 2015-02-26.

[133] Vasco. Card Readers. Online, 2016. URL https://www.vasco.com/products/two-factor-authenticators/hardware/card-readers/index.html. accessed: 2017-04-19.

[134] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M McCune. *Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?*, volume 7344 of *Trust and Trustworthy Computing, CyLab, Carnegie Mellon University, USA*. Springer, November 2012. doi: 10.1007/978-3-642-30921-2_10. URL https://www.cylab.cmu.edu/files/pdfs/tech_reports/CMUCyLab11023.pdf. accessed: 2014-06-23.

[135] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Symposium on Security and Privacy (SP)*, pages 430–444. IEEE, 2013. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6547125. accessed: 2014-08-12.

[136] Paulo Verıssimo. Intrusion Tolerance: Concepts and Design Principles. Technical report, Department of Informatics, University of Lisbon, July 2002. URL http://repositorio.ul.pt/bitstream/10451/14089/1/02-6.pdf. accessed: 2017-09-14.

[137] Paulo Veríssimo, Nuno Neves, and Miguel Correia. Intrusion-tolerant architectures: Concepts and design. *Architecting Dependable Systems*, pages 3–36, 2003. URL http://link.springer.com/chapter/10.1007/3-540-45177-3_1. accessed: 2017-03-03.

[138] Paulo E Veríssimo. Travelling through wormholes: A new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006. URL http://dl.acm.org/citation.cfm?id=1122497. accessed: 2017-03-03.

[139] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6081855. accessed: 2014-04-08.

[140] Nikos Virvilis, Dimitris Gritzalis, and Theodoros Apostolopoulos. Trusted Computing vs. Advanced Persistent Threats: Can a defender win this game? In *10th Intl. Conf. on Autonomic and Trusted Computing (ATC)*, pages 396–403. IEEE, 2013. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6726235. accessed: 2015-02-13.

[141] Feiyi Wang, Fengmin Gong, Chandramouli Sargor, Katerina Goseva-Popstojanova, Kishor Trivedi, and Frank Jou. SITAR: A scalable intrusion-tolerant architecture for distributed services. In *Workshop on Information Assurance and Security*. IEEE, 2001. URL http://users.nccs.gov/~fwang2/papers/T1B3_35.pdf. accessed: 2017-10-13.

[142] Eric Weatherwax, John Knight, and Anh Nguyen-Tuong. A model of secretless security in N-variant systems. In *Workshop on Compiler and Architectural Tech-*

niques for Application Reliability and Security (CATARS-2), 2009. URL `http://core.ac.uk/download/pdf/21748638.pdf`. accessed: 2016-09-28.

[143] Johannes Winter. Trusted computing building blocks for embedded Linux-based ARM TrustZone platforms. In *3rd workshop on Scalable trusted computing*, pages 21–30. ACM, 2008. URL `https://dl.acm.org/citation.cfm?id=1456460`. accessed: 2015-02-26.

[144] Johannes Winter. Experimenting with ARM TrustZone–Or: How I Met Friendly Piece of Trusted Hardware. In *11th Intl. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)* Winter [143], pages 1161–1166. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6296108`. accessed: 2015-02-17.

[145] Zongwei Zhou, Virgil D Gligor, James Newsome, and Jonathan M McCune. Building verifiable trusted path on commodity x86 computers. In *Symposium on Security and Privacy (SP)*, pages 616–630. IEEE, 2012. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6234440`. accessed: 2014-11-13.

[146] Bonnie Zhu, Anthony Joseph, and Shankar Sastry. A taxonomy of cyber attacks on SCADA systems. In *Intl. Conf. on Internet of things (iThings/CPSCom)*, pages 380–388. IEEE, 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6142258`. accessed: 2016-02-29.

APPENDIX A

PICTURES

A.1 Genode Microkernel

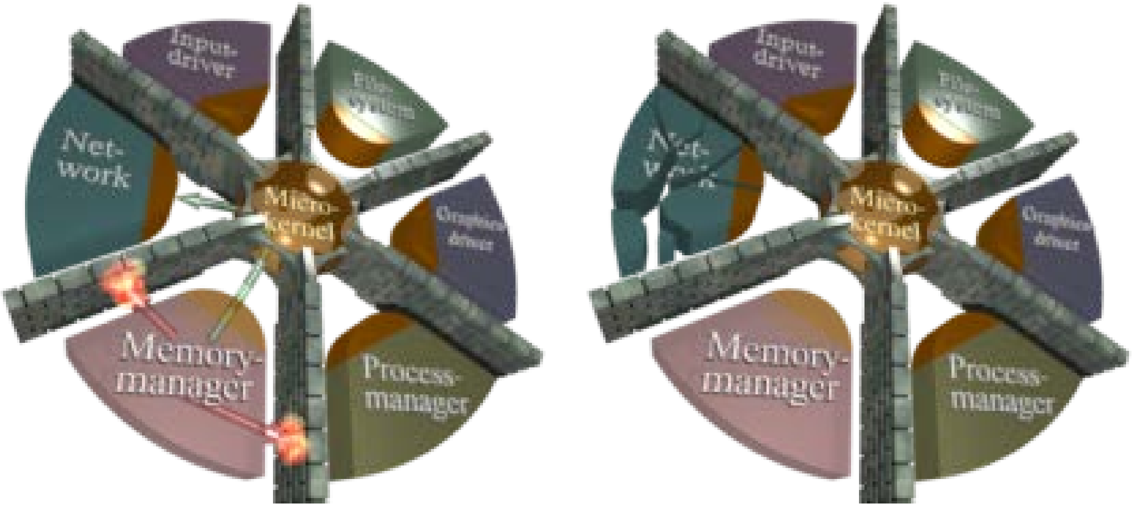
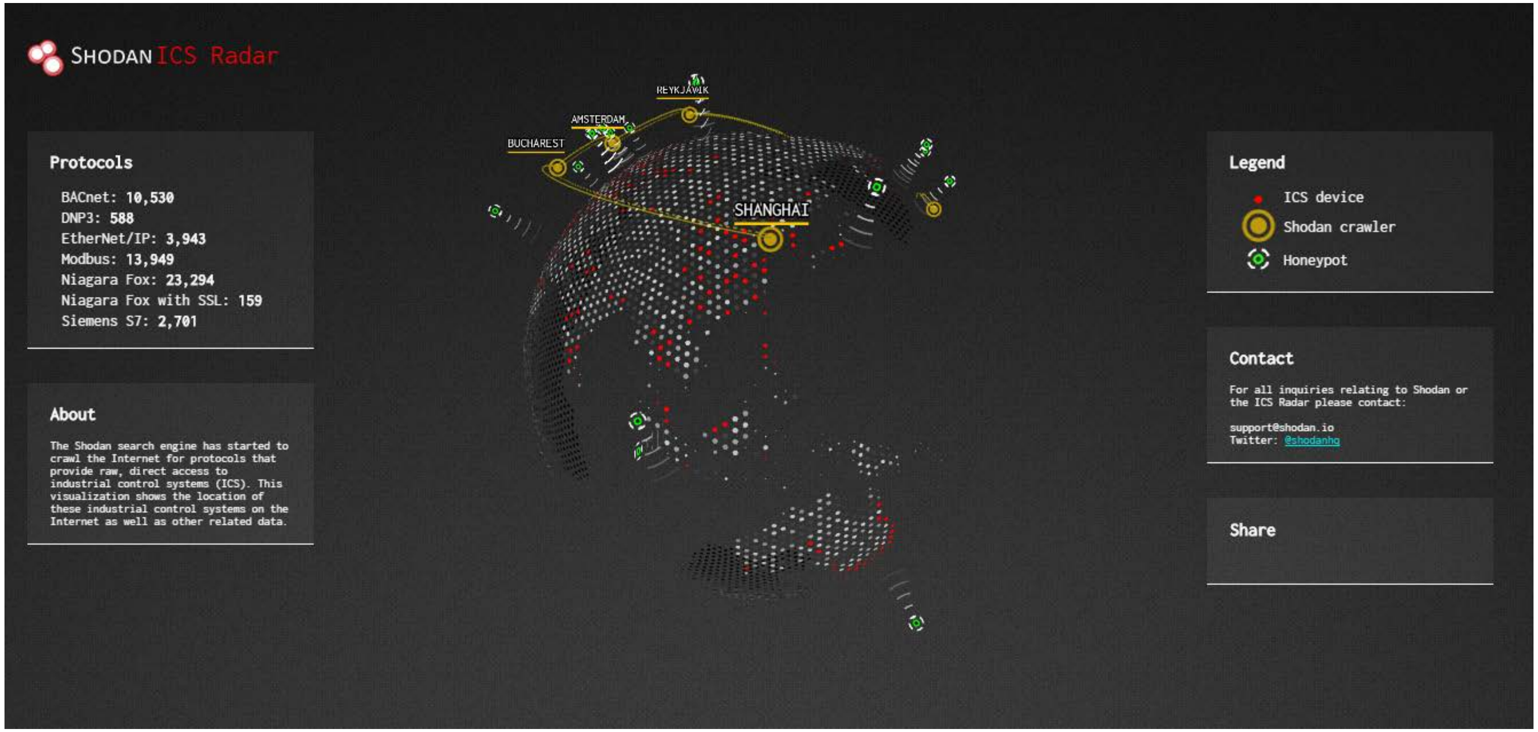


Figure A.1: Sketch of the Genode operating system [60] showing the microkernel architecture.
Source: `https://genode.org/documentation/general-overview/index`

A.2 Shodan Industrial Control System Scan



SHODAN ICS Radar

Protocols

BACnet: **10,530**
DNP3: **588**
EtherNet/IP: **3,943**
Modbus: **13,949**
Niagara Fox: **23,294**
Niagara Fox with SSL: **159**
Siemens S7: **2,701**

About

The Shodan search engine has started to crawl the Internet for protocols that provide raw, direct access to industrial control systems (ICS). This visualization shows the location of these industrial control systems on the Internet as well as other related data.

Legend

● ICS device
◎ Shodan crawler
⊙ Honeypot

Contact

For all inquiries relating to Shodan or the ICS Radar please contact:

support@shodan.io
Twitter: @shodanhq

Share

Figure A.2: Shodan ICS scan website: `https://ics-radar.shodan.io/`