UiO : **University of Oslo**

Elahe Fazeldehkordi

Security and Privacy Solutions in IoT and Distributed Systems Design

Thesis submitted for the degree of Philosophiae Doctor

Department of Technology Systems and
Department of Informatics
Faculty of Mathematics and Natural Sciences

2021

© Elahe Fazeldehkordi, 2021

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2375

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

Abstract

New distributed systems in computing technology are appearing on the market day by day. Consequently, new security and privacy challenges arise when designing these systems. These challenges demand a need to look for comprehensive and more precise approaches that can provide higher levels of security, privacy, and trust from the design phase in these systems.

In order to develop systems including open distributed systems, we need techniques and tools for specification, design, and code generation. For the initial ideas of such systems, it is desirable to use graphical notations, so that ideas can easily be understood. This is relevant for the early design phase. Once the ideas are agreed upon, it is then desirable to have a formal basis for the specifications of the desired system, in order to support rigorous reasoning about specifications and designs. This can be done in the late design phase.

Existing documents for security and privacy requirements and functionalities in IoT systems lack some of the security functionalities, and in particular, they do not focus on privacy issues or are presented only in textual form, without defining a framework. Structures of these documents are also complicated. Systems are often made without the help of security and privacy experts. So a comprehensive graphical representation framework is needed and should be easy to follow, even for non-experts.

Formal methods have come into wide use in the design and verification of safety, security, and privacy of systems in the industry, because they are very effective in verifying safety, security, and privacy requirements of systems, requirements for which testing is mostly ineffective. Formal verification techniques can guarantee that a design is free of specific flaws.

Provability of IoT and distributed systems depends on the language used to model the system, its semantics, and the kind of system properties to prove and the techniques used to verify them. One may take a bottom-up approach, starting with low level languages in use, or one

may take a top-down approach, starting with a more abstract language with an expressiveness suitable for IoT and distributed systems. Such a language can be used to model IoT and distributed systems, and if the language is executable, it can be used for simulation and prototyping of IoT and distributed systems, and if the language is imperative with standard mechanisms such as object orientation, IoT and distributed models made in the language can easily be translated to more low level languages. The latter approach also has the advantage that one can define the language and its semantics so that it is amenable to semantical analysis and verification. Additionally, this approach makes program reasoning simple and powerful.

The active object paradigm provides a natural way of modeling distributed systems in general, and IoT systems in particular, because it covers the fundamental aspects of IoT systems such as distribution of concurrent, autonomous units communicating by message passing, where each unit can run on a device with limited processing power and limited storage.

In this thesis, we focus on two aspects of the design phase to develop methodologies to improve security, privacy, and trust levels of IoT and distributed systems. Firstly, we consider the early design phase, where the architecture of a system is being developed, focusing on the setting of IoT systems. Secondly, we consider the late design phase, where models, in particular executable models and prototypes are designed, focusing on IoT systems and also the more general setting of distributed systems. For the modeling related to the late design phase, we limit our work to the active object paradigm. In particular, we find techniques for increasing the security level of the overall IoT systems using a security and privacy functionality framework, and a technique for detecting potential vulnerabilities that can cause distributed denial of service (DDoS) attacks using static analysis technique. In addition, we developed a language-based approach to provide trust, safety, security, and privacy for smart contracts.

Dedication

This thesis is dedicated to my beloved family and my fiancee for their endless support, encouragement, patience, and love in my life and their belief in my dreams.

Acknowledgements

This thesis would have been impossible without the support and mentoring of my main supervisor Olaf Owe. I would like to express my deep gratitude to him for his priceless time and help whenever I needed. I am grateful for his continuous encouragement, inspiration, patient guidance, fruitful discussions, constructive and constant cooperations, and optimism throughout this work, and for all the knowledge I have learned from him. I am also grateful for his great friendship and for teaching me various things of life. I would like to express my special thanks to my co-supervisor Josef Noll for his great help and support, critical suggestions, patient guidance, and active collaboration in various papers. I would also like to thank Martin Steffen for his constructive and useful feedbacks, comments, discussions, interesting lectures, and for all his help whenever I needed. I would like to thank my graduate committee members, Jüri Vain, Svetlana Boudko, and Paal Engelstad for their great feedbacks, help, and encouragement. I also thank Toktam Ramezanifarkhani and Christian Johansen for their feedbacks and guidance during this research.

I would like to thank everybody involved in the IOTSEC project, in particular I thank Habtamu Abie for his great help, support, encouragement, and constructive feedbacks. I am grateful for all the great assistance and facilities provided by the administration staff and the technical support at the Department of Technology Systems and the Department of Informatics, University of Oslo. My thanks also go to all researchers and professors who have crossed my path as friends and colleagues over the years, in particular Jüri Vain.

I am grateful to my family and my fiancee. Words cannot express how grateful I am to them. They have always encouraged and supported me in pursuing my dreams with their love and patience, and they have always been with me despite the distance. Finally, I would like to thank my friend Mona for her company during these years.

This work was supported by the IOTSEC project. Additional funding was provided by the Department of Technology Systems, SCOTT and ConSeRNS projects.

Contents

Abstract **i**

Dedication **iii**

Contents **vii**

List of Figures **xi**

List of Tables **xiii**

Part I Overview 1

1 Introduction **3**
- *1.1 Background and Motivation* 4
- *1.2 Research Questions* 12
- *1.3 Methodology* 13
- *1.4 Structure of the Thesis* 14

2 Theoretical Background **17**
- *2.1 Object-Oriented Systems* 17
- *2.2 Concurrent Object-Oriented Systems* 19
- *2.3 Creol/ABS Language* 27
- *2.4 Formal Verification and Specification* 29

3 Summary of Research Papers and Contributions **41**
- *3.1 Paper 1* 42
- *3.2 Paper 2* 43
- *3.3 Paper 3* 43
- *3.4 Paper 4* 44
- *3.5 Additional Publications* 47

4 Conclusions and Future Work **49**

4.1 Conclusions 49
4.2 Future Work 54

Bibliography **57**

Part II Scientific Contributions **67**

5 Paper 1: Security and Privacy Functionalities in IoT **69**
5.1 Introduction 69
5.2 IoT-Related Standards and Guidelines 72
5.3 Related Work 73
5.4 Framework Explanation 75
5.5 Security Classification 79
5.6 Pacemaker Case Study 81
5.7 Conclusion 92

Bibliography **95**

6 Paper 2: A Case Study of Healthcare Products **99**
6.1 Introduction 100
6.2 Related Work 102
6.3 Security Classification 103
6.4 Pacemaker Case Study 104
6.5 Discussion 113
6.6 Conclusion 115

Bibliography **117**

7 Paper 3: A Language-Based Approach to Prevent DDoS Attacks **121**
7.1 Introduction 121
7.2 Overview 124
7.3 Related Work 126
7.4 Our Framework for Active Object Systems 128
7.5 Static Analysis to Prevent Attacks 130
7.6 Examples of Possible DoS/DDoS Attacks 134
7.7 Conclusion 143

Bibliography **145**

8 Paper 4: An Approach to Smart Contracts Supporting Safety and Security **149**

8.1 Introduction 150
8.2 Smart Contracts and Blockchain 153
8.3 A High-Level Language for Active Object systems 156
8.4 The Proposed Framework 165
8.5 Specification of the Auction Example and Improvements 179
8.6 Verification 184
8.7 Evaluation 188
8.8 Related Work 193
8.9 Conclusion 196

Bibliography **199**

List of Figures

1.1 *An illustration of the system development life cycle* . . 5

5.1 *IoT security and privacy functionality framework* . . . 73
5.2 *Security mechanisms* 73
5.3 *Human resource security* 76
5.4 *Physical and environmental security* 76
5.5 *Privacy protection* 76
5.6 *Operations security* 77
5.7 *Development, maintenance, and audit* 77
5.8 *IoT security and privacy functionality framework – decommissioning* 77
5.9 *Basic inputs for defining a security class* 80
5.10 *Case study – scenario 1; with a security controller along with the pacemaker inside the body* 87
5.11 *Case study – scenario 2; with a security controller for the pacemaker out of the body for enhanced security* 87

6.1 *Case study – scenario 1; with a security controller along with the pacemaker inside the body* 109
6.2 *Case study – scenario 2; with a security controller for the pacemaker out of the body for enhanced security* 109

7.1 *Distributed communication (s-obj stands for server object and c-obj for consumer object)* 123
7.2 *Distributed object communication in DDoS* 124
7.3 *Control flow graph* 131
7.4 *Algorithm for detecting flooding by means of* $calls$ *and* $comps$ *sets in a given cycle* 132
7.5 *The interfaces of the units in the subscription example* . 135
7.6 *Classes providing an implementation of the subscription example* 136
7.7 *DoS attack by a variation of the subscription example* . 137

7.8 *The graph and call/comp sets for the original version of the program* 137
7.9 *The graph and call/comp sets for the modified version of the program* 137
7.10 *Flooding by unbounded creation of innocent clients targeting the same server* 139
7.11 *Static detection of flooding using unbounded creation* . 139

8.1 *Interfaces for the auction example* 158
8.2 *Examples of function definitions* 161
8.3 *The auction class* 166
8.4 *The interfaces of futures and history objects* 167
8.5 *History objects* 168
8.6 *Predefined types for transactions* 170
8.7 *A class implementation of futures and history objects* . 172
8.8 *Illustration of the future mechanism* 174
8.9 *An implementation of history objects where blacklisted bidders are blocked* 175
8.10 *The privacy-extended history class for auction* 176
8.11 *A history class implementation with safety check* 178
8.12 *The auction history interface and class* 180
8.13 *A trust-extended history class for auction* 183

List of Tables

5.1 *Security classes* . . . 80
5.2 *Exposure* . . . 80
5.3 *Security and privacy challenge comparison of scenario 1 and 2* . . . 89
5.4 *Security class of the sensor* . . . 91
5.5 *Security class of the pacemaker security controller* . . . 91
5.6 *Security class of the mobile phone* . . . 91

6.1 *Exposure* . . . 103
6.2 *Security classes* . . . 103
6.3 *Security and privacy challenge comparison of scenarios 1 and 2* . . . 109
6.4 *Security class of the sensor* . . . 113
6.5 *Security class of the pacemaker* . . . 113
6.6 *Security class of the mobile phone* . . . 113

Part I
Overview

CHAPTER 1

Introduction

The modern society is critically depending on information systems, in particular distributed systems and the Internet of Things (IoT). A distributed system is a system that consists of components placed on different networked computers which communicate concurrently by passing messages to each other. Examples of distributed systems include computer networks like the Internet, aircraft control systems, distributed information processing systems like banking systems, mobile phones, etc. The Internet of Things (IoT) consists of different distributed systems and is considered one of the most important areas of future technology, it has received massive attention from a wide variety of industries. IoT is made from a variety of *things* or *objects* – like sensors, mobile phones, television, refrigerators, actuators, etc. – that are able to interact with each other and cooperate with their neighbours through the Internet to reach a common goal using unique addressing schemes.

The aim of IoT is to provide an advanced mode of communication among the various systems and devices, and also to facilitate the interaction between humans and the virtual world. With this aim, IoT plays a significant role in the modern society and has applications in almost all fields including healthcare systems, automobile, industrial appliances, sports, homes, entertainments, environmental monitoring, etc. Services provided by IoT applications offer great benefits for human's life, but they can come with a huge price with respect to people's privacy and security protection. IoT devices have already outnumbered the number of people at a computerized workplaces. As a result of this expansion, and as these things are connecting to the Internet, and generate, process, and exchange huge amounts of security-critical and privacy-sensitive information, therefore they are attractive targets of various attacks [21, 29, 41, 47, 54, 55, 57, 61, 62, 63, 75, 76, 77, 83, 86]. The same attack may be used on a large number of (identical) devices.

What makes the IoT different from the traditional Internet is the absence of human role. IoT offers complex systems that can sense the external environment and make decisions with no need of human

intervention. Existing sensors in IoT capture almost every single piece of information from the environment such as image, video, sound, location, proximity, temperature, humidity, acceleration, pressure, and heartbeat. Hence, through these systems a lot of information about human life is going to be collected and processed with some environments like smart wearables, smart healthcare products, or smart homes that are able to sense and manage very sensitive and personal information. This huge amount of uncontrolled collection of sensitive information requires strong data protection and privacy controls.

Despite the advanced abilities provided by IoT and distributed systems in the data communication area, their vulnerability implications from a security and privacy standpoint are still of great concern. Cyberattacks on IoT and distributed systems such as healthcare systems, banking systems, and very recently digital coins are very critical since they can cause significant physical and economic damages, and even threaten human lives. It is therefore vital to make these systems more secure and data privacy-protected. This has made the investment in techniques for preventing security and privacy attacks very important. Appropriate steps should be taken in the initial phase of development and design of these systems.

1.1 Background and Motivation

1.1.1 System Development Life Cycle

System development life cycle (see Figure. 1.1), also referred to as the software/application development life cycle, is whole process of developing and maintaining an information system. system development life cycle comprised of the planning phase, the requirement analysis phase, the system design phase, the implementation phase, and the integration and testing phase.

System design phase defines the architecture, modules, interfaces, and data for a system to satisfy specific requirements. This phase is very important in the sense that, it bridges the gap between the problem domain and the system, supposedly in a manageable way. The focus of this phase is to find a solution of how to make a system that fulfills the desired purpose. This phase finds a solution to how the system can

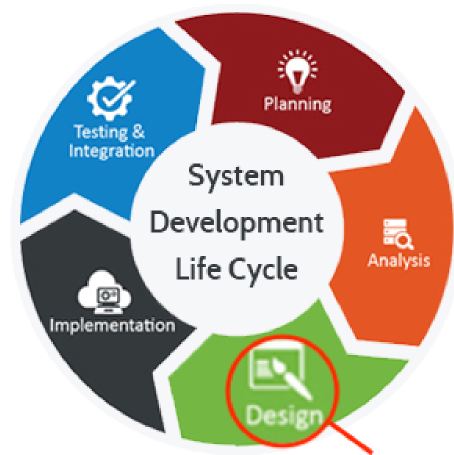

Figure 1.1: An illustration of the system development life cycle [90]

be implemented and operate in such a way that the specification can be satisfied. In the design phase, the complex activity of system development will be divided into several smaller sub-activities, which coordinate with each other to achieve the main objective of the system development.

In order to develop systems including open distributed systems, we need techniques and tools for specification, design, and code generation. For the initial ideas of such systems, it is desirable to use graphical notations, so that ideas can easily be understood. This is relevant for the early design phase. Once the ideas are agreed upon, it is then desirable to have a formal basis for the specifications of the desired system, in order to support rigorous reasoning about specifications and designs. This can be done in the late design phase.

1.1.2 Security and Privacy Taxonomy of IoT Systems

Having a comprehensive overview and *taxonomy* of security and privacy requirements and functionalities in IoT at the design phase is a prerequisite for architecting optimal security solutions, designing, and developing secure and privacy-aware IoT systems that can help both providers and consumers of IoT devices to have a better understanding of the security and privacy aspects. By functionality we mean: "The security and privacy-related features, functions, mechanisms, services, procedures, and architectures implemented within organizational information systems

or the environments in which those systems operate” [79].

General standards, guidelines, and frameworks [1, 2, 25, 32, 33, 68, 79, 81] for establishing, implementing, maintaining, and continually improving an information security management system, and protecting the confidentiality of Controlled Unclassified Information (CUI), exist. Existing IoT-related documents lack some of the security functionalities, and in particular, they do not focus on privacy issues or are presented only in textual form, without defining a framework. Layer structures of these documents are also complicated. Systems are often made without the help of security and privacy experts. So a comprehensive graphical representation framework is needed and should be easy to follow, even for non-experts. Such a representation should extract all the IoT-related security and privacy functionalities and unify them using a common vocabulary. The vocabulary should be categorized, and the guidelines and requirements should be integrated in a uniform style.

1.1.3 Privacy and GDPR

With respect to privacy, the European Union (EU) has passed the *General Data Protection Regulation* (GDPR) [84], the toughest privacy and security law in the world. It came into force on 24th May 2016, and since 25th May 2018 all the organizations and their software are required to be compliant. GDPR replaced the European Data Protection Directive which established minimum data privacy and security standards in 1995 and based on that every member state depends on its implementing law. The aim of GDPR is to protect and empower all EU citizens/residents’ data privacy and to reshape the way organizations approaching data privacy of such people. Even though GDPR is passed by the EU, it forces obligations on organizations anywhere if they collect and process data related to EU citizens/residents, or offer goods or services to these people. Penalties for violating the GDPR are very high and can be 4% of global revenue or €20 million (whichever is greater).

GDPR regulates the following principles:

> *Accountability; The purpose of limitation principle; Data minimization; Lawfulness, fairness and transparency; Accuracy; Data retention periods; and Data security.*

The key rights of Data Subjects, Controllers, and Processors are as follows:

Breach notification, right to access, right to rectification, right to be forgotten, right to restrict processing, right to data portability, right to object, rights in relation to automated decision making and profiling, consent, and data protection by design and by default.

The GDPR requirements relate to almost all modern computer systems and should be taken into consideration in the design phase to support privacy by design.

1.1.4 System Modeling

Modern computer systems comprise of complex hardware and software components, nevertheless ensuring the quality and the correctness of the software part has been often a more significant problem than the one in the underlying hardware. A documentation of this problem providing over 100 "software horror stories" can be found in [30]. The correctness of the software is, in particular, very important from the security and privacy point of view. Any single mistake in the software of a system can be misused by the attackers. We should ensure that the software is reliable and works correctly and does not have any vulnerability, ideally during the design process.

One way to promote correct and reliable programs is through *modeling* and *verification* techniques. *System modeling* develops abstract models of a system. Each model presents a different view of that system [10]. Models help to derive detailed requirements for a system: during the design process they describe the system to engineers, and after implementation document a system's structure and operation. System models simplify a system and do not represent a complete system. By leaving out details of the system and decreasing the amount of complexity, we will be able to understand the system and analyze specific properties related to the system more easily.

Models can be developed from different perspectives of a system [82]: (1) an external perspective models the context or environment of the system; (2) an interaction perspective models the interactions between a system and its environment, or between the components of the system; (3) a structural perspective models the organization of a system or the

structure of the data processed by the system, and (4) a behavioral perspective models the dynamic behavior of the system and how it responds to events.

1.1.5 Formal Methods

The use of *formal methods* technology in the industry has been an objective for researchers for several decades. Formal methods are mathematically-based approaches to software development that define a formal model of the software. This model can then be formally analyzed to look for errors and inconsistencies. Using mathematical approaches, users will be able to analyze and verify these models at any part of the program life-cycle (requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution).

Formal methods are important because they increase reliability of a system. Industrial applications of formal methods and verification technologies have been discussed in an excellent survey by Woodcock et al. [89]. Formal methods have come into wide use in the design and verification of safety-critical systems in the industry, because they are very effective in verifying safety and security requirements of systems, requirements for which *testing* is mostly ineffective. Testing and simulation are techniques that provide a variety of input conditions for a system and ensure that the output is as expected. The inputs can be provided by a designer or randomly generated. These techniques sample the response of the systems to the chosen inputs [74]. They can sample *functional properties* of a system – i.e., what a system is supposed to do, but they are not able to verify *safety and security properties* – i.e., what a system is not supposed to do. For the former properties, given a nominal input it produces the specified output. It is often assumed that the variety of designed-in behaviors are usually small enough to adequately test for compliance. This is only a small fraction of possible valid inputs. The latter properties are predicated on any path that could bring the system into an unsafe or insecure state; therefore, the input state space is vast and almost never accessible by testing. Mathematical analysis and verification of the program are needed to be sure that safety and security requirements are met and to guarantee that a design is free of specific flaws.

Formal verification techniques can guarantee that a design is free of specific flaws. Formal verification is the act of proving or disproving the

correctness of intended algorithms of a system concerning a certain formal specification or property, using formal methods. Formal verification techniques can check the behavior of a design for *all* input vectors.

In distributed systems like smart contracts, there is a need for formal verification in the most popular language, Solidity, that has been used for writing these contracts [4, 39, 94]. Also, in these systems we can achieve a high level of trust and privacy with less expensive (with respect to time, resources, and power consumption) approach using formal methods.

One of the formal verification techniques is program analysis. Program analysis offers static (compile-time) techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at runtime when executing a program on a computer [64].

In services like financial systems, static analysis techniques are very important. Even a single unfortunate mistake by a financial service provider could be enough to cause unintended attacks or be misused by attackers and influence customers and this would of course, destroy the customer's trust and confidence and result in financial loss and reputation damage. One should make sure that the software is not harmful for the customers before running it, and this makes the static detection very important in these systems. Most of the existing methods recommend using of Internet traffic monitoring techniques to detect and react to possible threats. But, this kind of runtime protection may slow down or temporarily shut down the websites and their services. Despite the importance of the static detection method as a valuable complement to runtime detection methods, it is underrepresented.

Provability of IoT and distributed systems depends on the language used to model the system, its semantics, and the kind of system properties to prove and the techniques used to verify them. One may take a bottom-up approach, starting with low level languages in use, or one may take a top-down approach, starting with a more abstract language with an expressiveness suitable for IoT and distributed systems. Such a language can be used to model IoT and distributed systems, and if the language is executable, it can be used for simulation and *prototyping* of IoT and distributed systems – i.e., the process of building a model of a system, and if the language is *imperative* – i.e., describes steps that change a program's state and tells *how* a program should operate, with standard mechanisms such as object orientation, IoT and distributed models made in the language can easily be translated to more low level IoT and

distributed languages. The latter approach also has the advantage that one can define the language and its semantics so that it is amenable to semantical analysis and verification. Additionally, this approach makes program *reasoning* (i.e., proof) simple and powerful.

1.1.6 Suitability of Modeling Paradigms for IoT and Distributed Systems

Object orientation is the leading paradigm used in the software industry today and is recommended for open distributed systems [78]. Program code in class-based object-oriented languages is organized in classes, encapsulating variables and methods, and classes can be specialized by subclassing, possibly with a redefinition of methods. Classes can be instantiated at runtime into objects. Many of the most popular programming languages are based on object orientation including Java, `C++`, `C#`, `Python`, etc. The object-oriented paradigm provides flexibility through polymorphism. A single function can shape-shift to adapt to whichever class it is in, we can create one function in the parent class and this one function would work with other functions. Reusable code through inheritance techniques saves time. Using inheritance, one can simply reuse existing code instead of duplicating code. Modularity is provided by the object-oriented paradigm, something which reduces complexity and makes it easier for troubleshooting. Objects may be seen as autonomous units. Object orientation provides good structuring mechanisms and flexibility in applying changes to artifacts produced during the software development process. This allows the structure to remain the same in the final system. By combining object orientation and *concurrency*, we can benefit from the increased responsiveness and throughput of distributed system.

Distributed systems are by nature concurrent, therefore modeling paradigms for distributed systems should support concurrency. Concurrent computing increases program throughput, parallel execution in concurrent programming allows the number of tasks completed in a given time to increase at best proportionally to the number of processors. Concurrent computing also increases responsiveness for input/output, it allows the time that would be spent mostly waiting for input or output operations to complete, to be used for other tasks. Concurrent computing

can also have more appropriate program structure, while some problems and problem domains are better-suited for representation as concurrent tasks or processes.

Standard models of object interaction do not address the specific challenges of distributed computation. Object interaction based on method calls is usually synchronous. However, synchronous communication in a distributed setting, causes undesired and uncontrolled waiting, and possibly deadlocks. Asynchronous message passing provides better control and efficiency, although it does not provide the structure and discipline inherent in method calls.

The *active object paradigm* [13, 91, 92] has evolved from the *actor model* [3] and object orientation. In this paradigm each object is concurrent and autonomous, with its virtual processor, and all interaction is through method invocations and replies, thus supporting two-way interaction rather than one-way interaction as in the case of the actor model. Therefore at most, one process is active on an object at a time, and method executions are decoupled from method invocations using underlying asynchronous message passing. Active object languages have mechanisms for efficient method interaction without active waiting, by means of the future concept or suspension.

The active object paradigm provides a natural way of modeling distributed systems in general, and IoT systems in particular, because it covers the fundamental aspects of IoT systems such as distribution of concurrent, autonomous units communicating by message passing, where each unit can run on a device with limited processing power and limited storage [48].

Each IoT device is modeled as one active object. If a single IoT device has several concurrent activities, such a device could be modeled by a group of active objects, using the concept of *object groups* as suggested and formalized in [12, 22, 50]. Such an object group can be seen as a single object for the environment. The modeling of different kinds of IoT networks, including wireless networks, can be modeled as in [52].

1.2 Research Questions

As discussed earlier, new distributed systems in computing technology are day by day appearing on the market, e.g., IoT systems. Consequently,

new security and privacy challenges arise when designing these systems. These challenges demand a need to look for comprehensive and more precise approaches that can provide higher levels of security, privacy, and trust from the design phase in these systems. For example, new taxonomy frameworks that organize all aspects of security and privacy baselines, guidelines, and recommendations, or the need to formally verify these systems at the software level. Therefore, this thesis is seeking solutions to the following main goal:

> ***Main objective:*** *To improve security and privacy of IoT and distributed systems during the design phase.*

The main objective is a broad area of research that requires a more defined focus. Therefore, we focus on two aspects of the design phase. First, the early design phase, where the architecture of a system is being developed, considering the setting of IoT systems. Second, the late phase of the design, where models, in particular executable models and prototypes are designed, considering IoT systems and also the more general setting of distributed systems. Having enough information at the architectural design phase and using graphical notations make it easier to understand the system components and their specifications for the modeling phase. These give two subgoals:

> ***Objective 1:*** *To develop a methodology to improve the overall security and privacy of IoT systems in the early design phase.*

> ***Objective 2:*** *To develop methodologies to improve the security, privacy, and trust level of distributed systems in the late design phase.*

To achieve the first subgoal, this thesis will address the following specific research questions:

RQ1: What are the security and privacy functionalities for IoT systems relevant at the early design phase?

RQ2: How can we build a security and privacy functionality framework for IoT systems relevant to the early design phase?

RQ3: How can the functionality framework help to improve security and privacy of IoT systems at the early design phase?

To achieve the second subgoal, this thesis will address the following specific research questions:

RQ4: How can we model distributed systems at a high level of abstraction?

RQ5: How can we analyze modeling frameworks of distributed systems to identify and detect possible vulnerabilities in the models in order to improve the security and trust level at the late design phase in these systems?

RQ6: How can we improve the security, privacy and trust levels of distributed systems using a methodology at the late design phase?

RQ7: How can we use formal verification and specification for the behavior of models of distributed systems?

1.3 Methodology

This thesis consists of several publications that describe our research results achieved in the direction of answering the above questions. Besides, the methodology we are looking for should meet the following requirements:

Model-based approach. A model is mainly a representation of some aspects of interest from a real-world system [10]. Models provide an abstraction of a real-world system. Therefore they contain less complexity than real-world systems. By decreasing the amount of complexity, we will be able to understand the interesting aspects better, and analyze specific properties related to them. Therefore it is desirable for the methodology presented in this thesis to be model-based. In addition, it is desirable that the model is executable in order to allow prototyping and verification of specification requirements of the system, and that it defines a program structure that can be a guideline for the implementation phase.

Formality. Formal methods are important since they seek to increase the reliability of systems by adding mathematical rigor to the design process [15]. For the methodology presented in this thesis, it is desirable to have a design notation that offers precision, unambiguity, and abstractions based on a formal syntax and a well-defined semantics.

The paradigm of concurrent and distributed active objects. We consider programming and specification languages suitable for modeling IoT and distributed systems. We define these languages, their semantics, and systems that are proof oriented towards reasoning support. In order to make reasoning as simple and powerful as possible, we focus on languages at a high level of abstraction rather than low level languages. We consider executable and imperative languages supporting object-oriented principles, and designed the languages so as to reduce the complexity of further translation to low level languages for IoT and distributed systems. The active object paradigm provides a natural way of modeling distributed systems in general, and IoT systems in particular, because it covers fundamental aspects of IoT systems, such as distribution of concurrent units communicating by message passing, where each unit can run on a device with limited processing power and limited storage [48]. Furthermore, the active object model can be seen as complementary to graphical modeling languages including the Unified Modeling Language (UML) [53], therefore is a suitable continuation of the early design phase.

For the modeling related to objective 2, we therefore limit our work to the active object paradigm.

1.4 Structure of the Thesis

The rest of this thesis is structured as follows: Chapter 2 introduces Creol/ABS language and discusses its underlying concepts. Chapter 3 gives a short summary of each of the papers collected in this thesis and contributions of this thesis by answering the research questions presented in Chapter 1. Chapter 4 concludes the thesis and suggests some directions for future research. Part II collects the research papers in the thesis.

CHAPTER 2

Theoretical Background

In this chapter we discuss some basic concepts of object-oriented and concurrent object-oriented modeling languages and then position the Creol/ABS modeling languages [48, 51] in this context. For this purpose, Sections 2.1 and 2.2 introduce object-oriented languages, and concurrent object-oriented languages, respectively. The presentation in Sections 2.1 and 2.2 is inspired by [6, 60]. Section 2.3 introduces specifically the Creol/ABS languages, which supports the active object modeling paradigm.

To build a reliable software system, it is important to develop techniques which facilitate verification about the behavior of the program code. For this purpose, we organize the discussion in Section 2.4 by first considering formal logical systems and programming logic, then discussing formal verification approaches including Hoare logic and static program analysis.

2.1 Object-Oriented Systems

Object orientation is claimed to be the most used programming techniques, in particular, for distributed systems [18]. The concept of objects was first introduced by Ole-Johan Dahl and Kristen Nygaard in Simula 67 [26, 28]. The ideas of the language which was system description as well as simulation programming originated by Kristen Nygaard at the Norwegian Computing Center in 1961 [66]. Along with Nygaard, Ole-Johan Dahl produced the initial ideas of object-oriented programming, which still is a leading approach for commercial and industrial applications today.

In the object-oriented programming style, a system is described as a collection of objects. An object is an integrated unit of *data* and *procedures* (called methods) that act on these data. We can think of object as a box that stores some data and is able to act on these data. The data in objects is stored in *variables*. A variable's content changes by executing an assignment statement. Variables in an object may be hidden from the environment, so called private variables. One may declare interfaces

to specify what methods can be called from the outside. By seeing an object solely through interfaces, object's variables will be strictly private and are not accessible to other objects. In class-based object-oriented languages, objects are described by *classes*, defining the common features for each kind of objects. Elements (*instances*) of a class have the same names and types for their variables and execute the same code for their methods. A class serves as a blueprint for building its instances. In the object-oriented model, objects exchange information with each other under the general label of *message passing* – a sender object sends a request to a receiver object to execute a certain method. The sender of the message may provide some parameters to the method and the method can return a result, which is passed to the receiver. Interaction between objects only happens based on the precisely determined message interface. Each object is responsible and is able to maintain its own local data in a consistent state.

At any time during the execution of a program, a new object can be generated, so there can be a large number of objects at some point. Objects are not destroyed explicitly. Although, if they certainly do not influence the correct execution of the program they can be removed through garbage collection.

The features listed below are common among languages considered to be strongly class- and object-oriented [60]:

- *Classes*: object-oriented languages are diverse, but the most popular ones are class-based, that means objects are instances of classes. A class is a description of a set of objects that are defined using a list of behavior (operations), properties (attributes), relationships, and semantics.

- *Encapsulation:* is a mechanism upon which certain features of a class will be unaccessible for user's calls. *Information hiding* sometimes used as a synonym for encapsulation. When writing a class, there are features that are part of the implementation of a class, but not of its interface. Other features of the class possibly available to users, may call the features if they need, but it is not possible for a user to call them directly. The encapsulation mechanism should be flexible in such a way that allows designers of a class to specify a feature available to all users, to no user,

or to some specific users; and an attribute that may need to be available for access only, access and restricted modification, or full modification.

- *Inheritance:* the basic idea is that when we want to define a new class it is often easier to start with variables and methods of existing class(s), then add more features to them in order to reach to the desired new class. In this case, we say that the new class inherits the variables and methods of the superclass(s). In other words, a class can be defined as an extension or restriction of another.

- *Polymorphism and dynamic late binding:* the ability of a program entity to be attached to objects of various possible types, and the guarantee that every feature on an entity will trigger the correct feature, corresponding to the attached runtime object's type.

- *Dynamic object creation instantiation:* at any point of the execution of a program a new object can be generated, so there might be a large number of objects.

Additional features of object-oriented languages are given as follows:

- *Multiple and repeated inheritance*: a class can have more than one parent and can inherit from another class in more than one way.

- *Abstraction*: is a mechanism that aims to representing a simplified model of the program without showing irrelevant details in order to handle complexity and enhance understanding [40, 56, 93]. Abstraction is provided by interfaces.

2.2 Concurrent Object-Oriented Systems

The concept of concurrent programming goes back to 1960s within the concept of operating systems. The idea was to build hardware units called *channels* or *device controllers* that operate independent of a controlling processor. They allow an I/O operation to work concurrently with continued execution of the central processor's program instructions. The program is called concurrent when it contains two or more processes that work together in order to do a task. Each of these processes is a sequential

program execution a sequence of statements one after another. In contrast to sequential programs that have a *single thread of control*, concurrent programs have *multiple threads of control*. The term *multithreaded* means a program has more processes or threads than processors to execute the threads. Concurrent programs are in general more complex than sequential programs. Execution in a concurrent program begins in some initial state and as processes execute independently, they examine and modify the program state.

2.2.1 Fundamental Concepts

Here, we look at five fundamental concepts of concurrent programs: program state, atomic actions, the concept of history, safety properties, and liveness properties [6]. The values of all program variables at a point in time are called the *state* of the program.

A sequence of statements executed in a process. A sequence of one or more *atomic actions* implement a statement. Atomic actions indivisibly examine or alter the program state. Therefore, the execution of a concurrent program results in an interleaving of the sequences of atomic actions that are executed by every process.

A *history* (also called a *trace* of the sequence of states) can be seen as a particular execution of a concurrent program: $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$, where s_0 is the initial state, s_1 ... s_n are subsequent states, and atomic actions make the transitions that change the state. In a concurrent program, every execution generates a history. In all but the most trivial programs, there is an enormous number of possible histories, since the next atomic action in each one of the processes could be the next one in a history, so there will be many paths that actions can be interleaved no matter a program always begins in the same initial state.

Synchronization aims to constrain the possible histories of a concurrent program to the desirable ones. There are two kinds of synchronization: *mutual exclusion* that ensures at most one process is executing its *critical section* at the same time, and *condition synchronization* that delays an action until the state holds a Boolean condition. Critical sections cannot be interleaved with actions in other processes when they reference the same variables.

In a program a *property* is an attribute that is true of every possible history of that program, so of all executions of the program. There are

two types of properties: safety and liveness. A *safety property* [6] is when the program never enters a bad state for instance a state that some variables have an undesirable value. A *liveness property* [6] is when the program eventually enters a good state for instance a state that variables have desirable values.

Partial correctness [6] is an example of a safety property. A program is called partially correct if it requires that if an answer is returned it will be correct. *Termination* is an example of a liveness property. If every loop and procedure call of a program terminates, then the program terminates, in this case, the length of every history of the program is finite. A *total correctness* property of a program combines partial correctness and termination. If a program always terminates with a correct answer, the program is called totally correct.

2.2.2 Communication in Concurrent Programming

In a concurrent program, the only non-trivial way that processes can work together in order to solve a problem is to *communicate* with each other in some way [6]. And this communication can only be done if one process writes into *something* that the other process can read. This something might be a shared variable or a shared communication channel that it depends on the hardware in which the program will execute on. Therefore, communication can be done either by writing and reading shared variables or by sending and receiving messages.

Shared variable communication is when one process writes into a variable that will be read by another process. Shared variables are mostly used in concurrent programs that execute on hardware where its processors share memory. In distributed-memory machines, if they are supported by a software implementation of so called distributed shared memory, the shared-variable programming model can also be used.

In a *distributed-memory architecture* where the processors have their own private memory, the communication is mostly through a communication network instead of shared memory. In this case, processes are not able to communicate through sharing variables, instead they communicate by exchanging messages with each other or by invoking remote operations. Concurrent programs that use message passing are named *distributed programs*, since processes can be distributed all over the distributed-memory architecture's processors. A distributed

program can also be executed on a shared memory multiprocessor, like any concurrent program that can be executed on a single, multiplexed processor. In this case, channels are implemented through shared memory rather than a communication network.

There are several mechanisms for distributed programming including asynchronous message passing, synchronous message passing, remote procedure call (RPC), and rendezvous. They are different in the way channels are named and used and the way communication is synchronized.

Message passing is when one process sends a message to a channel, another process can acquire the message through receiving it from the channel. Sending a message can be asynchronous (non-blocking) or synchronous(blocking). In asynchronous message passing, a channel is an unlimited queue of messages that have been sent but not received any answer yet. *Send* is a *non-blocking* primitive and never causes delay, but receive is a *blocking* primitive and it might cause delay until a message arrives to be received. In contrast, synchronous message passing causes the sender to *block* until the message is received, that means a process can send only one message on any channel at the same time and until that message receives and the receiver sends back a reply to the sender, the sending process cannot continue its execution and send another message. Synchronous message passing has one advantage that there is a limitation on the size of communication channels, therefore on buffer space. However, it has two disadvantages. First, concurrency is reduced because when two processes communicate, at least one of them will have to block. Another disadvantage of synchronous message passing is that this form of message passing is more prone to *deadlock* – when two or more processes are waiting for conditions that will never happen, therefore none of them could proceed. Therefore the difference between asynchronous and synchronous message passing is having more concurrency, less waiting time and having bounded communication buffer, respectively. Since memory is plentiful and compare with synchronous send, asynchronous message passing is less prone to deadlock, most programmers prefer it. In particular, for distributed environments considering that communication in a network takes time, asynchronous send is more suitable.

Remote procedure call (RPC) and rendezvous are ideally suitable for programming of client/server interactions. In RPC and rendezvous, an operation is a two-way communication channel from the caller to

the process that services the call and then back to the caller instead of sending information through one-way communication channels in message passing. One-way communication channels need a large number of channels for the two-way information flow between clients and servers that they need two explicit message interactions through two different message channels. In addition, every client needs a different reply channel. Also, in both RPC and rendezvous, the execution of call delays the caller until the called operation is executed and any result is returned as in synchronous message passing.

The difference between RPC and rendezvous is in the way of servicing the invocations of operations. RPC declares a procedure for every operation and provides a new process to handle every call, the caller and procedure body may be on different machines that's why it is called *remote procedure call*. In contrast, rendezvous uses an existing process to handle calls (rendezvous is sometimes called *extended rendezvous*).

2.2.3 Actor Model

The term *actor* was defined by Carl Hewitt at MIT [43] in 1973. The idea was to describe the concept of reasoning agents. However, over the years it has been refined into a model of concurrency.

An actor is a fundamental unit of computation that embodies all three essential elements of computation: processing, storage, communication. When an actor receives a message it can create some more actors, it can send messages to actors that it has addresses before and it can designates how to handle the next message that it receives. Actors are isolated from each other and they do not share memory, they have state and the only way to change the state is by receiving a message. Each actor has a unique mail address that is determined at the time of its creation. This address is used to specify the recipient of a message. Every actor has its own mailbox which is similar to message queue. Messages are stored in actors mailboxes until they are processed. Actors after they are created wait for the messages to arrive, actors can communicate with each other only through messages. Messages are sent to actor's mailboxes and processed in First In First Out (FIFO) order. Messages have immutable data structures that can be sent over the network. Conceptually an actor can only process one message at the time. Actors work asynchronously and they do not need to wait for a response from another actor. Actors

have addresses, it is possible for an actor to send a message to another actor by knowing its address, an actor can only communicate with actors whose addresses it has. An actor has addresses of the actors that it has created itself and also it can obtain other addresses from a message it receives. One actor can have many addresses.

The actor model supports maximum concurrency, however it includes excessive process creation overhead. With the actor model, developers do not have to worry about locking and race conditions. Properties of communication in the actor model are as follows: actors do not use a channel or intermediaries so there is nothing like type and semantics issues which exist in case of having a channel or things like buffer or guarantee delivery. Actors use best effort delivery. Actors have at-most-once-delivery. Messages can take arbitrarily long to be delivered. There is no message ordering guarantees. The running state of an actor is monitored and managed by another actor (called a *supervision*). The supervision can perform actions based on the state of the actor.

2.2.4 Active Object Model

The active object model was first proposed by Yonezawa et al. [92] in an object-oriented concurrent programming language called ABCL/1. Their computation model evolved from the actor model [9, 42], and it differs from actor computation model in many respects. Here, we explain the important differences between the two models.

Every object in the active object model of Yonezawa et al. has its own thread of control and it may have its local persistent memory. An object always has one of the three modes: *dormant*, *active*, or *waiting*. An object is initially in dormant mode. If an object receives a message that satisfies one of the specified patterns and constraints, it changes to the active mode. When the sequence of actions that are performed in response to an accepted message have been completed by an active object and there is no more subsequent messages have arrived, an active object changes to the dormant mode again. An active object sometimes needs to stop its current activity and wait for a message with a specified patterns to arrive. In this situation, the active object changes to the waiting mode and when it receives a required message it changes to the active mode again.

In the active object model, two distinct modes of message passing have been considered: *ordinary* and *express*. For each object T, there are two message queues: one for messages sent to T in the ordinary mode and the other for messages sent in the express mode. Messages are enqueued in arrival order.

Ordinary mode message passing [92]:

> Suppose a message M sen tin the ordinary mode arrives at an object T when the message queue associated with T is empty. If T is in the dormant mode, M is checked as to whether or not it is acceptable according to T's *script* (a description which satisfies its behavior: what messages it accepts and what actions it performs when it receives such messages). When M is acceptable,T becomes active and starts performing the actions specified for it. When M is not acceptable, it is discarded. If T is in the active mode, M is put at the end of the ordinary message queue associated with T. If T is in the waiting mode, M is checked to see if it satisfies one of the pattern-and-constraint pairs that T accepts in this waiting mode. When M is acceptable, T is reactivated and starts performing the specified actions. When M is not acceptable, it is put at the end of the message queue.

Express mode message passing [92]:

> Suppose a message M sent in the express mode arrives at an object T. If T has been previously activated by a message which was also sent to T in the express mode, M is put at the end of the express message queue associated with T. Otherwise, M is checked to see if it satisfies one of the pattern-and-constraint pairs that T accepts. If M is acceptable, T starts performing the actions specified for M even if T has been previously activated by a message sent to T in the ordinary mode. The actions specified for the previous message are suspended until the actions specified for M are completed. If so specified, the suspended actions are aborted. But, in default, they are resumed.

In the active object model, there are three types of message passing: *past*, *now*, and *future*.

Past type message passing:

> An active object O sends a message M to an object T and it proceeds its own task without waiting for a response to the requested task from the object T.

This type of message passing significantly increases the concurrency of activities in a system and corresponds to the standard way of message passing in the actor model.

Now type message passing:

> An active object O sends a message M to an object T and it waits for a response to the requested task from the object T.

This convention is useful in assigning a returned value to a variable in an assignment.

Future type message passing:

> An active object O sends a message M to an object T but it does not need the expected requested result immediately. Therefore, object O does not wait for object T to return the result and proceeds its tasks and computations. Meanwhile or later when object O needs the result, it checks its special *private* object called *future object* that was specified at the time of sending M. If the result is available in the future object, it can use it otherwise it continues its computations again until a result is available.

The future object behaves like a queue and the contents of the queue can be checked or removed by the object O. The concurrency of a system increases by future message passing.

One of the important difference between the active object model and actor model is that in the active object model, an object in the waiting mode can accept a message which is not necessarily at the head of the message queue, however an actor in the actor model can only accepts a message that is at the head of the message queue. Moreover, now type and future type message passing are not supported in the actor model. Hence, when an actor A sends a message to a target actor T and expects a result from T must terminate its current activity before being able to receive the response from actor T, like any other incoming messages arriving to A. In addition, this necessity of the termination of actor A's current activity

in order to receive actor T's response leads to an unnatural decomposition of A's task and requires additional procedures to identify a message that responds to the reply value of now type message passing.

2.3 Creol/ABS Language

The Creol language proposes programming constructs for distributed concurrent objects depending on asynchronous method calls and processor release points [48], building on the OUN language [70] and the language idea of [78]. Concurrent objects are active, meaning that an object encapsulates an execution thread. Method calls are represented by pairs of asynchronous messages. The language is class-based and supports inheritance. Objects have identity meaning that an object's "name" is unique. Communication is between named objects and object names may be exchanged between objects. In Creol object variables (references) are typed by interfaces. The language is extended with mechanisms for static type control in dynamically reconfigurable systems. The language is strongly typed, depending on a nominal type-system that makes sure invoked methods are supported by the callee (if not nil) and formal and actual parameters match [49]. Creol has a modular and compositional semantics meaning that each execution step in the semantic involves only one object. Its operational semantics is defined in rewriting logic [59] and is executable in the tool Maude [23].

ABS is an *abstract behavioral specification* language building on Creol, addressing *executable formal model* specifications for distributed object-oriented systems and has code generators for Java [37], Scala [67], Maude [24], and Erlang [7]. Both Creol and ABS allow a high level specification of a system including its concurrency and synchronization mechanisms, and also local state updates. Therefore, the concurrent control flow of object-oriented systems are captured by Creol/ABS models, while implementation details that are not desirable at the modeling level are abstracted away, like the concrete representation of internal data structures, which are captured by an applicative data type language, the scheduling of method activations, which is simply non-deterministic, and the properties of the communication environment. Like Creol, ABS uses interfaces as types for object variables, while the classes that implement the functionality of these objects are abstracted away

in the type system. ABS has a notion of object groups (cogs), which is similar to the one in JCoBox [80], it generalizes the concurrency model of Creol from single concurrent objects to concurrent object groups sharing the same processor.

In ABS, the communication mechanism of Creol [48] for remote communication has been used, it is based on asynchronous method calls using first-class futures [14]. Another feature of the ABS and Creol communication mechanism is their use of *cooperative scheduling* between asynchronous called methods. However, in ABS this is lifted from the level of objects to the level of concurrent object groups. Cooperative scheduling is a mechanism, allowing passive waiting which means there will be a process queue holding all suspended processes of a given object, the executing process of an object can be suspended so that a required result from another process of the queue of the object can be provided. The type system in ABS is similar to the one in Featherweight Java [46], a core calculus for Java. Featherweight Java is class-based with single inheritance, with subtyping as a reflexive and transitive closure of the subclass relation. Both Creol and ABS distinguishes between classes and types. In Creol asynchronous calls and interfaces are combined with multiple class inheritance, choice operators, and a notion of *cointerface* at the interface level in order to accommodate type-safe calls backs to callers [49]. The cointerface is the interface of the caller, through which the callee can make calls back to the caller. In this case the cointerface will ensure type correctness of these call-backs. Creol introduces a suspension mechanisms, which is also used in ABS. A process (or a method execution) may suspend while waiting for a result to arrive or condition to be satisfied. Suspension releases the processor of the object using a process queue to store suspended processes, allowing the object to continue with other (enabled) processes. Process release (suspension) points allow non-blocking waiting.

The language version that we consider in this thesis is inspired by both Creol and ABS and we refer to it by Creol/ABS. It ignores some of the features in ABS (like cogs) and supports some features that ABS does not support (inheritance, cointerfaces). Our work is mainly using local futures, rather than shared futures.

For the setting of IoT and distributed systems, local futures (in particular method-local ones) are suitable since they can be deleted by each object as soon as they are no longer needed so there is no future to

be garbage collected. However, if the active object paradigm supports *first-class futures (also called shared futures)* – i.e., futures that can be shared with other objects if they need them, it is not clear when none of the objects need the shared future(s) and when they can be deleted. In this case, garbage collection of futures will in general be needed. This is of special importance for the setting of IoT systems where garbage collection, in particular distributed garbage collection, is undesired due to resource limitations. For distributed systems, first-class futures may pose security and privacy issues, since they can be seen as shared variables with read-only access. Local futures are not shared between objects and security and privacy control is easier.

2.4 Formal Verification and Specification

Correctness proofs in programs are important in order to establish program properties. Results of the execution of processes when they are executed in parallel, can depend on the unpredictable order in which the actions from different processes are executed. This will result in a complexity that is very massive to handle informally. Also, program testing is not able to detect all mistakes since the particular interactions in which errors are visible may not happen. Therefore, it is important to have a proof system in order to remove some of this complexity, structure concurrent programs in a simple way, and to verify their correctness.

Below we will first look at verification methods (in Section 2.4.3) including semi-automatic ones, and methods that are fully automatic.

2.4.1 Formal Logical Systems

A formal logical system [27, 36] is used for inferring theorems from axioms based on a set of rules. These rules are defined in the form of: a set of *symbols*, a set of *formulas* built from the symbols, a set of *axioms*, and a set of *inference rules*. Formulas are a sequence of symbols. Axioms are formulas that are assumed to be true and are used as a starting point for further reasoning and arguments. Inference rules determine a way to derive additional true formulas from axioms and other true formulas.

Inference rules are in the form bellow:

$$\frac{H_1 H_2 ... H_n}{C}$$

where H_i is a *hypothesis*, and C is a conclusion. This means: if all the hypotheses are true, then conclusion is also true. Hypotheses and conclusion are formulas or schematic representations of formulas.

A *proof* in a formal logical system is a finite sequence of line, each line (any line in a formal proof is called *theorem*) can be an axiom or can be derived from previous lines by applying an inference rule.

A formal logical system is a mathematical abstraction (i.e., a collection of symbols and relations between them). In order to be able to know if the statements that the formulas represent are true or not, we need to provide an interpretation of the formulas.

An *interpretation* of a classical logic maps formulas to either true or false. In a logic, if all its axioms and inference rules are sound, the logic is called *sound*. Soundness of axioms and inference rules means all the axioms map to true, and all the inference rules conclusions map to true by the assumption that the hypotheses map to true.

The dual of soundness is *completeness*. In a logic, if each semantically true formula is provable, the logic is called *complete*.

2.4.2 Programming Logic

A *programming logic* is a formal logical system that allows us to state and prove properties of programs. A programming logic aims in facilitating proving properties of program execution. Like any formal logical system, programming logic also contains symbols, formulas, axioms, and inference rules. The symbols of programming logic are predicates, braces, and programming language statements.

One way to formally proof correctness of systems is Hoare logic. Hoare logic is a formal system with a set of logical rules for correctness proofs of programs including concurrent programs. Hoare [45] was the first to develop a formal logic for proving partial correctness properties of sequential programs, this work was inspired by the first proposed technique for proving correctness of programs by Floyd [34]. Later Susan Owicki in her dissertation supervised by David Gries, developed the first

complete logic for proving partial correctness properties of concurrent programs [71, 72, 73] based on Hoare logic for parallel programs [44].

Hoare introduced the concept of a *triple*, interpretation for triples, axioms, and inference rules for sequential statements. Since then logical systems that follow this style are called Hoare logic. Programming logic is an example of a Hoare logic. Formulas in programming logic and Hoare logic are triples of the following form:

$$\{P\}\ S\ \{Q\}$$

where the predicates P and Q (often called *assertions*) describe relations between the values of program variables, and S is a statement or statement list. The *interpretation* of a triple has the following meaning:

> Whenever execution of a program S starts in a state where assertion P holds, then the assertion Q will hold in a resulting state, assuming the execution of the program terminates.

This interpretation is named *partial correctness*. Predicates P and Q are called the *precondition* and *postcondition* of S, since they characterize conditions that must hold before and after the execution of S.

2.4.3 Formal Verification

The act of proving or disproving the correctness of the algorithms and programs in a system corresponding to certain *specifications* and properties of the system, by applying formal methods of mathematics, is called *formal verification*. *Formal specification* precisely describes system properties using mathematic-based languages.

There are many approaches to formally verify programs. The specification formalism that represent a specification, provides a good discrimination criterion between different approaches. Different formalisms are classifies to three groups [38]: logic, automata and language theory, and hybrid formalisms. In the first one, specifications are expressed as statements in a specific kind of logic. In the second one, a specification is represented in the form of a language (or an automaton). And, the last one uses a hybrid approach meaning that a specification first expressed using a logic statement then it is converted to an equivalent language/automaton

representation. Below we give a hint of approaches that use each of these formalisms. Due to extensive literature available in each area of formal verification, we cannot possibly do justice to the fine details of every approach and we ask the reader to refer to the references for more details.

2.4.3.1 Logic

Logic in general is the study of principles of reasoning and a subject of study in multiple disciplines like philosophy, mathematics, and computer science. We only consider *formal logic*, in which a *formal language* and a *formal system* represent logical truths and logical methods of reasoning. A formal language is a set of *well-formed-formulas* specified syntactically by a set of symbols and a set of rules for the formation of formulas. A formal system as explained earlier in this section consists of a formal language and a deductive apparatus. A deductive apparatus, also called a deductive system consists of a set of axioms and a set of inference rules that can be used to derive theorems of the system.

First-order predicate logic. An example of deductive system is first order predicate logic also known as first-order logic consists of a countable sets of symbols for constants, functions, and predicates, variables, and a set of standard Boolean connectives $(\wedge, \vee, \Rightarrow, \equiv)$ and quantifiers $(\exists, \forall)$. This distinguishes first-order logic from propositional logic, in which quantifiers or relations are not used. In Floyd-Hoare [34, 45] assertional methods also typically first-order predicate assertions are used. Most of the approaches that have used first-order logic actually draw upon the Hoare-style verification techniques for software.

Boyer-Moore computational logic. This logic is a restricted form of first-order logic. Boyer-Moore [16] introduced a quantifier-free first-order logic with equality. In this logic, terms composed of variables and function applications. Constants are demonstrated as functions without arguments. Logical connectives, like not, and, or are defined in terms of primitive logical constants $true$, $false$, and the primitive connective if. Equality is in the form of a function $equal$ and axioms that characterize it. Through the *Shell Principle*, a user can introduce inductively constructed object type (characterized by a set of axioms). According to the shell principle, all type definitions should be accompanied by specification of a recognizer function, a constructor function, and an accessor function

for that object type. A user can also introduce axioms defining new functions. These axioms have to satisfy the *Principle* of *Definition*, to avoid inconsistencies causing by their addition. Using this principle, one can make sure that all new functions are defined either non-recursively in case of pre-defined functions, or there exists a well-founded ordering on some measure of the arguments that reduces with every recursive call in case of recursive definitions. In this logic, standard rules of inference used for propositional logic and the principle of induction, which is based on the same well-founded ordering used by the definition axioms, have been used for reasoning.

The Boyer-Moore theorem prover equips an automated facility for generating proofs in the described logic above. The process of proof generation though, is not totally automatic, and the user might need to assist the theorem prover for setting up intermediate lemmas and helpful definitions. The basic theory used in the system (logic plus theorem prover) is including: axiomatize natural numbers, negative integers, lists and character strings, also commands are provided for adding shells, defining new functions and proving theorems. This system is an effective tool that has a number of applications in different areas [17], due to the strong mathematical foundation and heuristics built into the system.

Higher-order logic. This logic is a form of predicate logic that has additional quantifiers and sometimes stronger semantics. Higher-order logic is more expressive than first-order logic because of the ability to quantify over predicate symbols, however its model-theoretic properties are less well-behaved than in the first-order logic.

Modal logic. This is the logic of necessities and possibilities. Modal logic is considered the development of logic of increasing ability to express change [58]. Propositional logic is the logic of absolute truths, that means in a domain, propositions are either true or false. In predicate logic, the notion of truth has been extended by making it relative, that means the truth of a predicate may depend on the related actual arguments (variables), as well as the domain (since the arguments can change over the elements in the domain of discourse). Modal logic extends this notion further by providing additional variability, that means the meaning of a predicate (or a function) symbol may also vary depending on what point it is in (this point is often called a "world"). Therefore, modal formulas

are interpreted according to a state in a universe (the universe consists of a set of states), a domain of discourse that appropriate logic symbols are interpreted by every state, as well as an accessibility relation between states. There are two basic modal operators: the necessity operator that is represented by $\Box P$ and the possibility operator that is represented by $\Diamond P$. The meaning of modality operators is as follows:

- $\Box P$: a property is true in state s if the property P is true in all states accessible from s
- $\Diamond P$: a property is true in state s if the property P is true in at least one state accessible from s

Temporal logic. This logic is a specialized form of modal logic that places additional restrictions on the accessibility relation to represent passage of time. In temporal logic, one can talk about the past, present, and future. In addition to basic modalities $\Box P$ and $\Diamond P$, there are two other operators in temporal logic: $\bigcirc$ and U. In the temporal framework, these four operators are referred to as *Always* (*Henceforth*, G), *Sometimes* (*Eventually*, F), *Next-time* (X), and *Until* (U), respectively. The meaning of these operators is as follows:

- $\Box P$: is true in state s, if P is true in all future states accessible from s
- $\Diamond P$: is true in state s, if P is true in some future state accessible from s
- $\bigcirc P$: is true in state s, if P is true in the next state accessible from s
- $P\ U\ Q$: is true in state s, if either Q is true in s, or it is true in some future state of s, and until then P is true at every intermediate state

There are different kinds of temporal logic. If the truth of a formula is determined with respect to an interval between states, it is called *Interval Temporal Logic* [5]. If the truth of a formula is determined with respect to a state, depending on the difference in viewing the notion of time, there are other categorization. If time is categorized as a single linear sequence of events, it is called *Linear Time (Temporal) Logic (LTL)* [58], but if a branching view of time is considered, that means in any instant of

time there are a branching set of possibilities into the future, it is called *Branching Time (Temporal) Logic (BTTL)* [31].

Extended temporal logic (ETL). The original work on ETL was first suggested by Wolper [87]. He showed that in propositional version of LTL (called PTL), a property like $even(p)$ is not expressible. Therefore, he proposed to add new operators corresponding to right-linear grammars to temporal logic. To interpret grammars as operators, he established a correspondence between words generated by a grammar and sequences. The basic idea is that if there is a word w (finite or infinite) over a finite alphabet σ and an assignment of formulas to every letter of σ, a sequence satisfies the word w for the given assignment if for all i, the formula associated with the letter appearing in the ith position of the word w is true in the ith state of the sequence. Later, the original work on ETL was generalized by Wolper, Vardi, and Sistla [88].

Mu-Calculus. This is another extension framework of the temporal logic context for extending expressiveness that in addition to the usual predicate logic operators, provides operators for denoting fixed points of predicate transformers (functions from relations to relations) [20].

2.4.3.2 Automata and Language Theory

Machine equivalence. An approach that both the implementation and the specification are represented as finite-state machines. An equivalence between the corresponding machines should be approved in order to show that each behavior of the implementation satisfies the specification.

Language containment. In language containment approach a containment between the languages representing the implementation ($l(Imp)$) and the specification ($l(Spec)$) has been considered instead of an equivalence in the previous approach.

Trace conformation. In trace theory a system's behavior is modeled as a set of traces that each of them is a sequence of events.

2.4.3.3 Hybrid Formalisms

LTL and finite-state automata. This verification techniques were used by Fujita, Tanaka, and Moto-oka [35]. In this technique, a finite-

state automata (FSA) description of an implementation is given, then using traditional automata techniques can verify that it satisfies a LTL specification.

Temporal logics and Büchi automata. This techniques presented by Vardi and Wolper [85]. The basic idea of their technique is as follows: a PTL formula is interpreted according to a computation comprising of an infinite sequence of states. Every state can be entirely described by a finite set of propositions, therefore a computation can be expressed as an infinite word over the alphabet comprising of truth assignments to these propositions. A constraint on the computations can be directly translated to a constraint in the form of these infinite words. In this way, by given any PTL formula, a finite-state automaton on infinite words can be built that accepts the exact set of sequences that satisfy the formula. Temporal logic formulas here can be seen as *finite-state acceptors* of infinite words, which is in the form of a Büchi automaton (this is named after Büchi who first studied them [19]).

Approaches like *automated theorem proving*, *model checking*, and *static analysis* are automatic techniques. In automated theorem proving, the relationship between a specification of a system and an implementation is considered as a theorem in logic, that should be proved within the context of a proof calculus. By giving a description of the system, a set of logical axioms, and a set of inference rules, a system produces a formal proof from the scratch. By its very nature, theorem proving is a *deductive process* – reasoning from one or more statements to reach a logical conclusion. This raises theoretical and practical concerns regarding managing its complexity. The theorem proving approach is automatic to some degree like the form of rewriting rules, specialized tautology-checkers, etc. But most of the automated theorem provers available today are actually semi-automated and in order to guide the proof searching process they need some kind of human interaction. Theorem proving systems are very general in their applications. Logic allows representation of virtually all mathematical knowledge in the form of domain-specific theories. The ability of theorem proving that can define appropriate theories, and reason about them by a common set of inference rules, makes a unifying framework that can perform all kinds of verification tasks. This very generality makes theorem proving very complex.

Model checking [8] is an algorithmic method that determines whether

a finite-state model of a system satisfies a correctness specification (the specification or property of a system is in the form of a logical formula). This is done by means of an exhaustive search of all possible states that a system could enter during its execution. Attention in model checking is focused on a single model, and it does not require to encode incidental knowledge of the whole world, therefore the complexity of this task is more manageable compare with theorem proving. Mostly, model checking provides a clear algorithm that can be completely automatic. Typically, these algorithms also have counter-example mechanism that is useful for debugging purposes, and is important from a practical point of view. However, since these systems are not general like theorem provers, they can only work for the kind of logic and model that it is designed for. Static analysis technique will be explained later in this chapter.

Other approaches. Deductive verification, type and effect systems, confluent rewriting systems (such as SOS-style rules for defining operation semantics), relational calculus, refinement calculus that is a formal system (inspired from Hoare logic) that promoting program refinement (program refinement is the verifiable transformation of an abstract (high level) formal specification into a concrete (low level) executable program).

In this thesis, we have benefited from formal verification using a Hoare-style logic for partial correctness, and static analysis technique in order to prove our approaches.

2.4.4 Static Program Analysis

The availability of powerful tools is crucial for the design, implementation, and maintenance of large programs. Advanced programming environments provide many such tools like syntax-directed editors, optimizing compilers, and debuggers as well we tools for transforming programs and estimating their performance. Program analysis has a significant role in many of these tools. Although program analysis techniques are extensively used in compiler optimization, we focus on their use in program verification.

Program analyses [64, 65] are static compile-time techniques that give useful information for predicting safe and computable approximations about the dynamic behaviour of programs that will arise at runtime when

executing a program on a computer. The analyses are *static*, it means that to find the results of the analysis, the programs are *not* run at all. The analyses are *safe*, it means that the result of the analysis defines all possible runs of the program. A static analysis is said to be syntactic if it is only concerned with the grammatical structure of a program, and semantic if it involves the meaning of grammatically correct programs. Thus semantic analysis is, in general, more effective in recognizing potential problems in a program.

Traditionally, program analyses have been developed for *optimizing compilers*. They are used at different levels in the compiler including optimizations apply to the source program, optimizations apply to the various intermediate representations used inside the compiler, and optimizations that exploit the architecture of the target machine and directly improve the target code. The improvements and the associated transformations caused using these analyses can have dramatic reductions on the running time. A main application is allowing compilers to generate codes to avoid *redundant* computations for instance by reusing available results or by moving loop invariant computations out of loops, etc. One of the more recent applications is the validation of software in order to decrease the probability of malicious or unintended behaviour. Usually in these applications, information from different parts of the program should be combined.

Approaches to program analysis have many features in common. One common feature among the approaches is that to be able to remain computable they can only provide *approximate answers*. In general, program behaviour is *undecidable*. All non-trivial, and semantical properties of programs are undecidable. Thus, program analysis *efficiently* compute approximate but *sound* guarantees – guarantees that are true. Therefore, what is expected in the program analysis is producing a possibly *larger set* of possibilities than what will ever happen during the program execution. The challenge here is to not produce the safe approximation too often, otherwise the analysis might be useless. Even though program analysis does not give precise information, it gives useful information for example we can know that a value will be positive, or a variable will fit within 1 byte of storage, a variable will always hold a value of its declared type, etc.

Another common feature is that program analyses should be *semantic based*, meaning that the information that we get from the analysis can be

proved to be correct based on *semantics* of the programming language – i.e., mathematical definitions of how a program behaves.

Some of the main approaches to program analysis are: data flow analysis, constraint based analysis, abstract interpretation, and type and effect systems. In *data flow analysis* which is a traditional form of program analysis, it is common to think of a program as a graph in which nodes are elementary blocks and edges show how control might pass from one elementary block to another.

There are two kinds of analysis *intraprocedural analysis* and *interprocedural analysis*. Intraprocedural analysis only deals with simple languages without functions and procedures. In contrast in interprocedural analysis functions and procedures are also taken into account. Interprocedural analysis becomes complicated for instance when we want to ensure that calls and returns match one another.

The purpose of *control flow analysis* that is often expressed as *constraint based analysis* is to specify information concerning what "elementary blocks" may lead to what other "elementary blocks". Control flow analysis is usually expressed as a constraint based analysis. *Constraint based analysis* is in fact one way to specify the control flow analysis using a collection of constraints. Constraints are used to determine the control flow between functions. In this analysis, the labeling scheme has been used. All *subexpressions* will be labeled. The control flow analysis is specified using a pair of functions $(\widehat{C}, \widehat{\rho})$, where $\widehat{C}(\ell)$ contains the values that the subexpression labeled ℓ may evaluate to and $\widehat{\rho}(x)$ contains the values that the variable x may be bound to.

Abstract Interpretation is a framework for computing over-approximations of a behavioural property of the program based on a fixpoint computation over a lattice representing it. Abstract Interpretation cannot reason about the exact program behavior because of undecidability, rather a conservative over-approximation will be obtained; however, this can be enough to prove program correctness.

A simple *type and effect systems* can be described as follows: a statement S maps a state to a state (if it terminates), therefore can be considered to have type $\Sigma \rightarrow \Sigma$ where Σ denotes the type of states; this is written as the judgement:

$$S : \Sigma \rightarrow \Sigma$$

A type and effect system can often be viewed as a combination of an *Effect System* and an *Annotated Type System*. In an Effect System,

judgements are usually in the form: $S : \Sigma \xrightarrow{\varphi} \Sigma$ where φ is the *effect* and tell us something about what will happen when *S* is executed, for example this can be which error might happen, or which file might be changed. In an Annotated Type System, judgements are usually in the form: $S : \Sigma_1 \rightarrow \Sigma_2$ where the Σ_i describes certain *properties of states*, for instance it can tell that a variable is positive or certain invariant is maintained.

In this thesis we use control flow analysis.

CHAPTER 3

Summary of Research Papers and Contributions

This chapter presents research contributions of the thesis, summarizing four research papers. The chapter then returns to the research questions formulated in Section 1.2 and discusses them in connection with the contributions of this thesis. The full content of the papers appears in Part II, as in their original publications, but reformatted to fit the structure of this thesis.

We first recall the research questions documented in Section 1.2:

RQ1: What are the security and privacy functionalities for IoT systems relevant at the early design phase?

RQ2: How can we build a security and privacy functionality framework for IoT systems relevant to the early design phase?

RQ3: How can the functionality framework help to improve security and privacy of IoT systems at the early design phase?

RQ4: How can we model distributed systems at a high level of abstraction?

RQ5: How can we analyze modeling frameworks of distributed systems to identify and detect possible vulnerabilities in the models in order to improve the security and trust level at the late design phase in these systems?

RQ6: How can we improve the security, privacy and trust levels of distributed systems using a methodology at the late design phase?

RQ7: How can we use formal verification and specification for the behavior of models of distributed systems?

The following sections give a summary of each paper and explain how the paper answers above research question(s).

3.1 Paper 1

Title:	Security and Privacy Functionalities in IoT
Authors:	Elahe Fazeldehkordi, Olaf Owe, and Josef Noll
Publication:	Presented at 17th International Conference on Privacy, Security and Trust (PST), 26–28 August 2019 (pp. 1–12). Published in IEEE. DOI: https://doi.org/10.1109/PST47121.2019.8949054

Summary and contributions: This paper presents a new taxonomy framework that organizes all aspects of security and privacy baselines, guidelines, and recommendations by summarizing the most current standards and documents related to security and privacy functionalities in the setting of IoT. The paper then explores how the framework can facilitate the process of improving IoT security and privacy, in combination with a security classification method and demonstrates the application of the framework on a case study.

For the framework, this paper investigates the most important IoT-related security baselines and guidelines developed by ENISA [32, 33], OWASP [68], the Industrial Internet Consortium [81], the Cloud Security Alliance [25], and the Broadband Internet Technical Advisory Group [11], etc., as well as security and privacy guidelines from ISO [1, 2] and NIST [79], which could be relevant for securing IoT systems. The paper extracted the parts of these documents that deal with IoT and security/privacy, and unified them using a common vocabulary, and then categorized and integrated the resulting guidelines and requirements in a uniform style, and embedded them in a graphical representation by means of a tool based on diagrams called xmind.

By presenting the security and privacy functionality framework and explaining how we have developed it, we address RQ1 and RQ2. Also, this paper addresses RQ3 by combining the framework with a security classification method and demonstrating how the framework can help to improve the security and privacy of a system using a case study.

3.2 Paper 2

Title:	Security and Privacy in IoT Systems: A Case Study of Healthcare Products
Authors:	Elahe Fazeldehkordi, Olaf Owe, and Josef Noll
Publication:	Presented at 13th International Symposium on Medical Information and Communication Technology (ISMICT), 8–10 May, 2019 (pp. 1–8). Published in IEEE. DOI: https://doi.org/10.1109/ISMICT.2019.8743971

Summary and contributions: This paper delves deeper and demonstrates the application of the framework presented in Paper 1 by a case study of healthcare products, in combination with a recent security classification method. To give an understanding of how the framework can help to improve security and privacy in practice, this paper gives a comprehensive discussion on the details of two scenarios in the case study, compares security and privacy challenges of the two scenarios, and then shows how to soften these challenges using security classifications and functionalities of the framework in Paper 1. This paper addresses RQ3.

3.3 Paper 3

Title:	A Language-Based Approach to Prevent DDoS Attacks in Distributed Financial Agent Systems
Authors:	Elahe Fazeldehkordi, Olaf Owe, and Toktam Ramezanifarkhani
Publication:	Presented at Security for Financial Critical Infrastructures and Services (FINSEC), part of the 24th European Symposium on Research in Computer Security (ESORICS) Conference, 23–27 September 2019. Published in Lecture Notes in Computer Science, vol 11981 (pp. 258–277). Springer. DOI: https://doi.org/10.1007/978-3-030-42051-2_18

Summary and contributions: This paper presents an approach as an additional layer of defense in distributed agent systems to combat Denial

of Service(DoS) and Distributed DoS(DDoS) attacks. A high level concurrent object-oriented modeling framework for distributed systems, based on actor model with support of asynchronous and synchronous method interaction and futures has been considered. The language supports efficient interaction by features such as asynchronous and non-blocking method calls and first-class futures.

The approach in this paper uses static analysis to identify and prevent potential vulnerabilities caused by asynchronous communication that can directly or indirectly cause call-based flooding of agents. More precisely, a general algorithm has been adapted for detecting call flooding [69] to the setting of security analysis and for detection of distributed denial of service attacks by adding support for one-to-many and many-to-one attacks. The algorithm detects call cycles that might overflow the incoming queues of one or more communicating agents. Each cycle may involve any number of agents, possible involving the attacked agent(s).

The paper also illustrates the approach on examples of distributed systems in the financial sector, including versions of a one-to-many attack and a many-to-one attack and shows how static detection can solve the situation in each case.

By showing examples modeling distributed systems in the financial sector using object-oriented modeling frameworks based on the active object paradigm, this paper addresses RQ4. Also, by applying static analysis on these examples in order to detect and prevent potential vulnerabilities caused by asynchronous communication that can directly or indirectly cause call-based flooding of agents resulting DoS and DDoS attacks, this paper achieves aspects of RQ5.

3.4 Paper 4

Title:	A Language-Based Approach to Smart Contracts Supporting Safety and Security
Authors:	Elahe Fazeldehkordi and Olaf Owe
Publication:	Submitted to the Journal of Logic and Algebraic Programming (JLAMP), April 2020, 51 pages. (under revision)

Summary and contributions: This paper presents a new approach to define lightweight smart contracts with associated history objects, supporting similar trust, immutability, and transparency of smart contracts based on blockchain but at the software level, without use of blockchain. The approach can apply to a wide range of contracts, not only financial ones, and opens up for lightweight smart contracts without the resource and energy costs of blockchain. However, it can also be combined with the blockchain technology in order to improve the overall trust level on insecure platforms. Furthermore, this approach offers better privacy control and comes with a theory for formal specification and verification.

The framework integrates trust at the language level through the notion of history objects. A history object can be used to provide safety, security, and privacy, as well as runtime checking. For each contract, a history object will record all related transactions and generated future values. Because of these recorded transactions, a history object can be seen as a ledger, but local to a given contract. Contract partners may interact with the history objects through predefined interfaces. The history objects are specially protected objects by predefined interfaces and provide read-only access from the programmer side, and increment-only access by the underlying system. They are separated from the contract provider, and can be used by contract users to check the validity of the transactions made, in a way similar to blockchain. The approach protects against tampering and fraud, each history object is immutable and corruption-proof, and each user can observe the contract behavior through the history object to ensure validity.

Moreover, the paper gives a theory for formal specification and verification of smart contracts. In particular, the approach supports class-wise verification, which is essential in open distributed systems where the contracts interact in an unknown environment. The verification is based on sequential reasoning augmented with effects on the transaction history. These advantages have been achieved by defining a version of a high level language based on the active object paradigm and interface abstraction. The language supports multiple inheritance and allows subclasses and redefinitions without behavioral restrictions from the superclasses.

Furthermore, the paper shows how formal specifications can be checked automatically by the history objects at runtime, thereby protecting users of a faulty contract provider, as well as protecting the contract provider from illegal users. In addition, it shows how to achieve security

and privacy. The paper illustrates the approach on a typical smart contract example, namely an auction system. The contract implementation has been verified and various improvements with respect to added safety, security, and privacy have been shown.

The approach may be combined with the notion of dynamic and concurrent object groups [25]. This allows a contract provider to appear as a single object to the outside while internally consisting of a number of cooperating concurrent objects.

A history objects can provide runtime checking of specified behavioral properties of the contracts. The framework allows runtime roll-backs, since the transaction history gives sufficient information to rerun a contract provider and stop at the last state before the execution of method that results in error. Reasoning about simple error recovery has been considered here.

This paper addresses RQ4 by showing examples of modeling smart contracts using a high level language based on active object paradigm. Also, this paper achieves aspects of RQ6 by demonstrating that how this modeling framework of smart contracts can support the main advantages of smart contracts based on blockchain, including trust, immutability, and transparency at the programming level and without use of blockchain. Besides, this modeling framework offers better privacy control using the notion of history objects. So it would be less expensive to implement compared with smart contracts built on blockchain that are costly with respect to time, resources, and power consumption. Furthermore, the paper explains that the approach can be combined with the blockchain technology in order to improve the overall trust level on insecure platforms. Moreover, RQ7 has been achieved in this paper by developing an imperative language for contracts, an executable functional language for writing smart contract specifications, and a theory for class-wise verification, with support of privacy, security, and model checking of contract specifications.

3.5 Additional Publications

This section lists other publications to which this thesis has contributed during her PhD research, but which are indirectly included as part of this

thesis. These papers are related to the goal of this thesis or correspond to shorter and/or preliminary versions of the work reported in this thesis.

- **Title:** Static Detection of Distributed Denial of Service Attacks in Active Object Systems

  Authors: Elahe Fazeldehkordi, Olaf Owe, and Toktam Ramezanifarkhani

  Publication: Technical Report of the Western Norway University (pp. 1–3). (presented at the PhD Symposium at the 15th International Conference on integrated Formal Methods (iFM), December 2–6, 2019)

- **Title:** Futures, Histories, and Smart Contracts

  Authors: Elahe Fazeldehkordi and Olaf Owe

  Publication: Proceedings of 31st Nordic Workshop on Programming Theory NWPT 2019 (p. 25–28),
  DOI: https://digikogu.taltech.ee/et/Download/536ca083-81f0-4f6c-9e59-c6b90b35125a.
  (presented at the 31st Nordic Workshop on Programming Theory – NWPT'19, November 13–15, 2019) (Invited paper for JLAMP journal)

- **Title:** A Framework for Flexible Program Evolution and Verification of Distributed Systems

  Authors: Olaf Owe, Elahe Fazeldehkordi, and Jia-Chun Lin

  Publication: In International Conference on Model-Driven Engineering and Software Development (pp. 320–349), February 20–22, 2019, Springer.

- **Title:** A Flexible Framework for Program Evolution and Verification

  Authors: Olaf Owe, Jia-Chun Lin, and Elahe Fazeldehkordi

  Publication: In MODELSWARD (pp. 177–189), February 20–22, 2019.

- **Title:** A Language-Based Approach to Prevent DDoS Attacks in Distributed Object Systems

  Authors: Toktam Ramezanifarkhani, Elahe Fazeldehkordi, and Olaf Owe

  Publication: Proceedings of the 29th Nordic Workshop on Programming Theory (p. 19–21),
  DOI: https://research.it.abo.fi/nwpt17/proceedings/NWPT2017 proceedings.pdf
  (presented at the 29th Nordic Workshop on Programming Theory - NWPT'17, November 1–3, 2017)

- **Title:** Hoare-Style Reasoning from Multiple Contracts

  Authors: Olaf Owe, Toktam Ramezanifarkhani, and Elahe Fazeldehkordi

  Publication: In the International Conference on integrated Formal Methods (pp. 263–278), Septemeber 18–22, 2017, Springer.

- **Title:** Security Functionality of IoT Devices

  Authors: Elahe Fazeldehkordi, Olaf Owe, and Toktam Ramezanifarkhani

  Publication: Proceedings of the PhD Symposium at iFM'17 of Formal Methods: Algorithms, Tools, and Applications (pp. 44–47),
  DOI: https://www.duo.uio.no/handle/10852/57814
  (presented at PhD Symposium at the 13th International Conference on integrated Formal Methods (iFM), September 18–22, 2017)

CHAPTER 4

Conclusions and Future Work

4.1 Conclusions

The main goal of this thesis is to improve security and privacy of IoT and distributed systems during the design phase. From this goal, two subgoals are identified that resulted in seven research questions presented in Section 1.2. This thesis has contributed to presenting four research papers which address the research questions and the related subgoals.

The focus of this thesis is on two aspects of the design phase. First, the early design phase, where the architecture of a system is being developed, considering the setting of IoT systems. Second, the late design phase, where models, in particular executable models and prototypes are designed, considering IoT systems and also the more general setting of distributed systems. Having sufficient information at the architectural design phase and using graphical notations make it easier to understand the system components and their specifications for the modeling phase.

The methodology proposed in this thesis meets the requirements stated in Section 1.3. We now revisit these requirements and their rationale:

Model-based approach. The methodology proposed in this thesis uses a modeling language which is executable in order to allow prototyping and verification of specification requirements of the system, so that it defines a program structure that can be a guideline for the implementation phase.

Formality. The methodology proposed in this thesis uses a language formalized by means of a formal syntax and an operational semantics given in the structural operational semantics (SOS) style that supports rigorous, abstractions, and formal analysis.

The paradigm of concurrent and distributed active objects. The methodology proposed in this thesis builds on the Creol/ABS language, this methodology uses a concurrent and distributed

object-oriented framework. Object orientation is the leading paradigm used in the software industry today and is suitable for open distributed systems [78]. The active object paradigm provides a natural way of modeling distributed systems in general, and IoT systems in particular, because it covers fundamental aspects of IoT systems, such as distribution of concurrent units communicating by message passing, where each unit can run on a device with limited processing power and limited storage [48]. In order to make reasoning as simple and powerful as possible, we focus on languages at a high level of abstraction rather than low level languages. We consider executable and imperative languages supporting object-oriented principles, and have designed the languages to reduce the complexity of further translation to low level languages for IoT and distributed systems.

In the following, we outline how the methodologies presented in this thesis address security and privacy solutions during the design phase by answering the research questions.

***RQ1:* What are the security and privacy functionalities for IoT systems relevant at the early design phase?**

Paper 1 collects security and privacy functionalities in the setting of IoT by summarizing the most current related documents.

***RQ2:* How can we build a security and privacy functionality framework for IoT systems relevant to the early design phase?**

Paper 1 presents a new taxonomy framework that organizes all aspects of security and privacy baselines, guidelines, and recommendations. The functionalities are separated into two major parts, the *life cycle* aspects of a system and the *management* aspects of security and privacy. The life cycle in this paper relates to the different phases in the life of a system, while the management of security and privacy is the ability to put supporting functionality elements in the system.

***RQ3:* How can the functionality framework help to improve security and privacy of IoT systems at the early design phase?**

To give an understanding of how the framework can help to improve the security and privacy of IoT products and to facilitate developing and designing secure and privacy-aware IoT systems, Paper 1 combines the taxonomy framework with a security classification method and demonstrates the usefulness by a case study. Paper 2 delves deeper and demonstrates the application of the framework described in Paper 1 using a case study of healthcare products, in combination with a recent security classification method. Paper 2 gives a comprehensive discussion on the details of two scenarios in the case study, compares security and privacy challenges of the two scenarios, and then shows how to soften these challenges using the security classifications and functionalities of the framework. The analysis results in Paper 1 and 2 demonstrate that using the security and privacy functionality framework, the security of the overall system has improved.

***RQ4:* How can we model distributed systems at a high level of abstraction?**

Paper 3 considers a high level object-oriented modeling framework for distributed systems based on active objects. This setting is appealing in that it naturally supports the distribution of autonomous concurrent units, and efficient interaction, avoiding active waiting and low level synchronization primitives such as explicit signaling and lock operations. It is therefore useful as a framework for modeling and analysis of distributed systems. The language in this paper supports efficient interaction by features such as asynchronous and non-blocking method calls and first-class futures, which are popular features applied in many distributed systems today. However, the DDoS detection method works better for local futures, since then the association between call statements and get statements can be done with less over-approximation, and the analysis of this is modular, in contrast to the setting of first-class futures where get statements may associate to call statements in other objects, including objects not known or not part of the considered subsystem. The DDoS detection method is therefore well suited for IoT systems with local futures.

Paper 4 considers a high level object-oriented modeling language that allows reasoning support. The language gives fewer runtime errors and less need for roll-backs, which simplifies reasoning. In particular, the

approach in paper 4 supports class-wise verification. This is essential for scalability and open-ended program development, which are highly relevant factors for contracts. The language is based on the active object paradigm. This paradigm offers modular semantics, which is essential when we turn to specification and verification issues. Clients, contract, and history objects in this paper are then described by concurrent and distributed objects. The language builds on the principle of interface abstraction, i.e., remote field access is illegal, and an object can only be accessed through an interface. Each object has one or more interfaces, and the only possible way of object interaction is through the methods defined in the corresponding interfaces. The language combines first-class futures, which is often used in active object languages, and a restricted version of cooperative scheduling. This novel combination gives flexible method interaction, scheduling control, simplified reproducibility of executions, as well as simplified verification. In particular, the restricted cooperative scheduling implies that each object is deterministic, relative to its inputs (i.e., invocation messages), something which is essential for enabling roll-backs. Furthermore, the futures (related to a contract) are now collected in the history objects, for which we have suggested mechanisms for security and privacy control. These futures will not be garbage collected since they are used in the calculations of roll-backs and in the functionality of the history object interfaces.

***RQ5:* How can we analyze modeling frameworks of distributed systems to identify and detect possible vulnerabilities in the models in order to improve the security and trust level at the late design phase in these systems?**

Paper 3 proposes an approach consisting of static analysis to identify and prevent potential vulnerabilities caused by asynchronous communication including call-based DoS or DDoS attacks, possibly involving a large number of distributed actors. The paper illustrates the approach on examples of distributed systems in the financial sector, including versions of a one-to-many attack and a many-to-one attack.

***RQ6:* How can we improve the security, privacy and trust levels of distributed systems using a methodology at the late design phase?**

Paper 4 proposes a new approach to define smart contracts, offering trust at the software level. This approach can apply to a wide range of contracts, not only financial ones, and opens up for lightweight smart contracts without the resource and energy costs of blockchain. The framework in paper 4 integrates trust at the language level through the notion of history objects. For each contract, a history object will record all related transactions. The history objects are specially protected objects, with read-only access at the programming language level. Contract partners may interact with the history objects through predefined interfaces. The paper shows that a history object can be used to provide safety, security, and privacy, as well as runtime checking. A history object can provide runtime checking of specified behavioral properties of the contracts. The paper demonstrates the approach on a smart contract example, namely an auction system.

RQ7: **How can we use formal verification and specification for the behavior of models of distributed systems?**

Paper 4 presents a framework integrating the notion of history objects giving rise to lightweight smart contracts, by developing an imperative language for contracts, an executable functional language for writing smart contract specifications by means of invariants referring to the transaction history of a contract, and a theory for class-wise verification, with support of privacy, security, and model checking of contract specifications.

4.1.1 Limitations

Our focus has been on the design phase of the system development life cycle, which means that planning, requirement analysis, implementation, and testing phases have not been considered. We have only considered two parts of the design phase, namely the early design phase, where the architecture of a system is being developed, and the late design phase – by means of executable modeling of systems using the active object concurrency model. However, the modeling paradigm considered allows high level implementation, prototyping, testing, as well as program analysis, at the abstraction level of the model.

We have considered only some approaches and some security problems. In the early design phase, we have only considered simple

forms of graphical architecture models in the setting of IoT systems. And at the late design phase we have only looked at the active object setting and have only considered DDoS attack and smart contracts. Security and privacy issues specifically for IoT models have not been considered. However, our analysis framework for DDoS attacks is specifically useful for IoT model where first-class future are avoided.

In the next section, we will look at the shortcomings of each research paper one by one and give some suggestions for possible extensions and future research directions.

4.2 Future Work

We organize the discussion in this section according to the research papers of the thesis and the shortcomings of each paper, this makes it easier to identify possible research directions, also makes it more clear to the reader how the suggested work relates to the papers in this thesis.

Security and Privacy Functionalities in IoT: The framework in this paper is a preliminary classification of privacy and security functionalities based on the available recommendations and standards for IoT systems. This should imply that all aspects are covered, but there is no guarantee for that. This can be seen as a limitation of the work. Secondly, the application of the methodology is ultimately depending on the judgements of software engineers or security experts, and is therefore not 100% precise. If their judgement is wrong, for instance if they choose the wrong connectivity or protection level, it will in general give a wrong estimate. One possible extension would be to allow the framework to be complemented with even more elements and look in more detail at GDPR for privacy functionalities.

Security and Privacy in IoT Systems: A Case Study of Healthcare Products: This paper only shows the application of our framework on one case study of healthcare products. The applications of our framework can be extended to other IoT domains and case studies with several kinds of IoT devices and sensors involved can be considered. This would be a valuable step toward validating the framework presented in this thesis.

A Language-Based Approach to Prevent DDoS Attacks in Distributed Financial Agent Systems: In this paper we used static analysis. Our static detection will overapproximate the set of (potentially) un-

completed calls. In future work, we suggest to complement the static checking with dynamic runtime checking since static detection methods give a degree of over-estimation. This could give a more precise combined detection strategy.

An Approach to Smart Contracts Supporting Safety and Security: In the current state, our framework can be used for modeling, prototyping, analysis, verification, and model checking of smart contracts. In future work, further mechanisms for error and exception handling can be added and efficient implementation of history objects and roll-backs can be considered.

Bibliography

[1] 27000, I. *Information technology — Security techniques — Information security management systems — Overview and vocabulary (fourth edition)*. ISO/IEC 2016. 2016.

[2] 27001, I. *INTERNATIONAL STANDARD ISO/IEC 27001 Information technology—Security techniques —Information security management systems —Requirements*. ISO/IEC 2013. 2013.

[3] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[4] Ahrendt, W., Pace, G. J., and Schneider, G. "Smart contracts: a killer application for deductive source code verification". In: *Principled Software Development*. Springer, 2018, pp. 1–18.

[5] Allen, J. F. and Ferguson, G. "Actions and events in interval temporal logic". In: *Journal of Logic and Computation* vol. 4, no. 5 (1994), pp. 531–579.

[6] Andrews, G. R. *Foundations of multithreaded, parallel, and distributed programming*. Vol. 11. Addison-Wesley Reading, 2000.

[7] Armstrong, J. et al. "Concurrent programming in ERLANG". In: (1993).

[8] Baier, C. and Katoen, J.-P. *Principles of model checking*. MIT press, 2008.

[9] Baker, H. and Hewitt, C. "Laws for communicating parallel processes". In: *MIT Artificial Intelligence Laboratory* (1977).

[10] Beydeda, S., Book, M., Gruhn, V., et al. *Model-driven software development*. Vol. 15. Heidelberg: Springer, 2005.

[11] BITAG. *Internet of Things (IoT) Security and Privacy Recommendations.* https://www.bitag.org/documents/BITAG_Report_-_Internet_of_Things_(IoT)_Security_and_Privacy_Recommendations.pdf. 2016.

[12] Bjørk, J. et al. "A type-safe model of adaptive object groups". In: *arXiv preprint arXiv:1208.4630* (2012).

[13] Boer, F. D. et al. "A Survey of Active Object Languages". In: *ACM Comput. Surv.* vol. 50, no. 5 (Oct. 2017), pp. 1–39.

[14] Boer, F. S. de, Clarke, D., and Johnsen, E. B. "A Complete Guide to the Future". In: *Programming Languages and Systems*. Ed. by De Nicola, R. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 316–330.

[15] Bowman, H. and Derrick, J. "Issues in Formal Methods (chapter 3)". In: *Formal Methods for Distributed Processing, A Survey of Object-oriented Approaches*. Ed. by Bowman, H. and Derrick, J. Cambridge, UK: Cambridge University Press, Sept. 2001, pp. 18–35.

[16] Boyer, R. S. and Moore, J. S. "A Computational Logic Handbook, volume 23 of". In: *Perspectives in computing* (1988).

[17] Boyer, R. S. and Moore, J. S. *Proof-Checking, Theorem Proving, and Program Verification.* Tech. rep. Texas Univ at Austin Inst for Computing Science and Computer Applications, 1983.

[18] Broy, M. "Distributed Concurrent Object-Oriented Software". In: *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*. Ed. by Owe, O., Krogdahl, S., and Lyche, T. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 83–95.

[19] Büchi, J. R. "On a Decision Method in Restricted Second Order Arithmetic". In: *The Collected Works of J. Richard Büchi*. Ed. by Mac Lane, S. and Siefkes, D. New York, NY: Springer New York, 1990, pp. 425–435.

[20] Burch, J. et al. "Symbolic model checking: 1020 States and beyond". In: *Information and Computation* vol. 98, no. 2 (1992), pp. 142–170.

[21] Byres, E. and Lowe, J. "The myths and facts behind cyber security risks for industrial control systems". In: *Proceedings of the VDE Kongress*. Vol. 116. 2004, pp. 213–218.

[22] Clarke, D., Johnsen, E. B., and Owe, O. “Concurrent Objects à la Carte”. In: *Concurrency, Compositionality, and Correctness: Essays in Honor of Willem-Paul de Roever*. Ed. by Dams, D., Hannemann, U., and Steffen, M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 185–206.

[23] Clavel, M. et al. “Maude: specification and programming in rewriting logic”. In: *Theoretical Computer Science* vol. 285, no. 2 (2002). Rewriting Logic and its Applications, pp. 187–243.

[24] Clavel, M. et al. *All About Maude – A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Vol. 4350. Springer, 2007.

[25] *Future-proofing the Connected World: 13 Steps to Developing Secure IoT Products*. https://downloads.cloudsecurityalliance.org/assets/research/internet-of-things/future-proofing-the-connected-world.pdf. 2016.

[26] Dahl, O.-J. “The Roots of Object Orientation: The Simula Language”. In: *Software Pioneers: Contributions to Software Engineering*. Ed. by Broy, M. and Denert, E. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 78–90.

[27] Dahl, O.-J. *Verifiable programming*. Prentice Hall PTR, 1992.

[28] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. *Common base language*. Norsk Regnesentral, 1970.

[29] Deogirikar, J. and Vidhate, A. “Security attacks in IoT: A survey”. In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE. 2017, pp. 32–37.

[30] Dershowitz, N. *Software horror stories*. URL: https://www.cs.tau.ac.il/~nachumd/horror.html.

[31] Emerson, E. A. and Halpern, J. Y. ““Sometimes” and “not never” revisited: on branching versus linear time temporal logic”. In: *Journal of the ACM (JACM)* vol. 33, no. 1 (1986), pp. 151–178.

[32] ENISA. *Baseline Security Recommendations for IoT in the context of Critical Information Infrastructures*. European Union Agency for Network and Inf. Sec. 2017.

[33] ENISA. *Guidelines for SMEs on the security of personal data processing*. https://www.enisa.europa.eu/publications/guidelines-for-smes-on-the-security-of-personal-data-processing. 2017.

[34] Floyd67, R. and Floyd, R. “Assigning meaning to programs”. In: *Mathematical Aspects of Computer Science* vol. 19 (), pp. 19–31.

[35] Fujita, M., Tanaka, H., and Moto-Oka, T. “Logic design assistance with temporal logic”. PhD thesis. University of Tokyo, 1984.

[36] Gallier, J. H. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.

[37] Gosling, J. et al. *The Java language specification*. Addison-Wesley Professional, 2000.

[38] Gupta, A. “Formal hardware verification methods: A survey”. In: *Formal Methods in System Design* vol. 1, no. 2-3 (1992), pp. 151–238.

[39] Hajdu, Á. and Jovanović, D. “SMT-Friendly Formalization of the Solidity Memory Model”. In: *arXiv preprint arXiv:2001.03256* (2020).

[40] Henderson-Sellers, B. *A book of object-oriented knowledge: an introduction to object-oriented software engineering*. Prentice-Hall, Inc., 1996.

[41] Hernandez, G. et al. “Smart nest thermostat: A smart spy in your home”. In: *Black Hat USA*, no. 2015 (2014).

[42] Hewitt, C. “Viewing control structures as patterns of passing messages”. In: *Artificial intelligence* vol. 8, no. 3 (1977), pp. 323–364.

[43] Hewitt, C., Bishop, P., and Steiger, R. “Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence”. In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[44] Hoare, C. A. R. “Towards a Theory of Parallel Programming”. In: *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Ed. by Hansen, P. B. New York, NY: Springer New York, 2002, pp. 231–244.

[45] Hoare, C. A. R. “An axiomatic basis for computer programming”. In: *Communications of the ACM* vol. 12, no. 10 (1969), pp. 576–580.

[46] Igarashi, A., Pierce, B. C., and Wadler, P. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 23, no. 3 (2001), pp. 396–450.

[47] Illera, A. G. and Vidal, J. V. “Lights off! The darkness of the smart meters”. In: *BlackHat Europe* (2014).

[48] Johnsen, E. B. and Owe, O. “An asynchronous communication model for distributed concurrent objects”. In: *Software & Systems Modeling* vol. 6, no. 1 (2007), pp. 39–58.

[49] Johnsen, E. B., Owe, O., and Yu, I. C. “Creol: A type-safe object-oriented model for distributed concurrent systems”. In: *Theoretical Computer Science* vol. 365, no. 1 (2006). Formal Methods for Components and Objects, pp. 23–66.

[50] Johnsen, E. B. et al. “A formal model of service-oriented dynamic object groups”. In: *Science of Computer Programming* vol. 115-116 (2016). Special Section on Foundations of Coordination Languages and Software (FOCLASA 2012) Special Section on Foundations of Coordination Languages and Software (FOCLASA 2013), pp. 3–22.

[51] Johnsen, E. B. et al. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Formal Methods for Components and Objects*. Ed. by Aichernig, B. K., Boer, F. S. de, and Bonsangue, M. M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 142–164.

[52] Johnsen, E. B. et al. “An Object-Oriented Component Model for Heterogeneous Nets”. In: *Formal Methods for Components and Objects*. Ed. by Boer, F. S. de et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 257–279.

[53] Johnsen, E. B. et al. “Specification of distributed systems with a combination of graphical and formal languages”. In: *Proceedings Eighth Asia-Pacific Software Engineering Conference*. IEEE. 2001, pp. 105–108.

[54] Kabay, M. “Attacks on Power Systems: Hackers, Malware”. In: *Norwich University* (2010).

[55] Koscher, K. et al. “Experimental security analysis of a modern automobile”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 447–462.

[56] Ledgard, H. F. *The little book of object-oriented programming*. Prentice-Hall, Inc., 1995.

[57] Levy, E. “Crossover: online pests plaguing the off line world”. In: *IEEE Security & Privacy* vol. 1, no. 6 (2003), pp. 71–73.

[58] Manna, Z. and Pnueli, A. *Verification of Concurrent Programs: The Temporal Framework; by Zohar Manna and Amir Pnueli*. Department of Computer Science, Stanford University, 1981.

[59] Meseguer, J. “Conditional rewriting logic as a unified model of concurrency”. In: *Theoretical computer science* vol. 96, no. 1 (1992), pp. 73–155.

[60] Meyer, B. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs, 1997.

[61] Miller, B. and Rowe, D. “A Survey SCADA of and Critical Infrastructure Incidents”. In: *Proceedings of the 1st Annual Conference on Research in Information Technology*. RIIT ’12. Calgary, Alberta, Canada: Association for Computing Machinery, 2012, pp. 51–56.

[62] Miller, C. and Valasek, C. “A survey of remote automotive attack surfaces”. In: *black hat USA* vol. 2014 (2014), p. 94.

[63] Nawir, M. et al. “Internet of Things (IoT): Taxonomy of security attacks”. In: *2016 3rd International Conference on Electronic Design (ICED)*. IEEE. 2016, pp. 321–326.

[64] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer, 2015.

[65] Nielson, H. R. and Nielson, F. *Semantics with applications: an appetizer*. Springer Science & Business Media, 2007.

[66] Nygaard, K. and Dahl, O.-J. “The development of the SIMULA languages”. In: *History of programming languages*. 1978, pp. 439–480.

[67] Odersky, M., Spoon, L., and Venners, B. *Programming in scala*. Artima Inc, 2008.

[68] OWASP. *IoT Security Guidance*.

[69] Owe, O. and McDowell, C. "On detecting over-eager concurrency in asynchronously communicating concurrent object systems". In: *Journal of Logical and Algebraic Methods in Programming* vol. 90 (2017), pp. 158–175.

[70] Owe, O. and Ryl, I. "On combining object orientation, openness and reliability". In: *Norwegian Informatics Conference*. 1999.

[71] Owicki, S. "Axiomatic proof techniques for parallel programs. Dept. of Computer Science, Cornell University". PhD thesis. 1975.

[72] Owicki, S. and Gries, D. "An axiomatic proof technique for parallel programs I". In: *Acta Informatica* vol. 6, no. 4 (1976), pp. 319–340.

[73] Owicki, S. and Gries, D. "Verifying properties of parallel programs: An axiomatic approach". In: *Communications of the ACM* vol. 19, no. 5 (1976), pp. 279–285.

[74] Pal, D. and Vain, J. "Model Based Test Framework for Communications-Critical Internet of Things Systems". In: *Databases and Information Systems X: Selected Papers from the Thirteenth International Baltic Conference, DB&IS 2018*. Vol. 315. IOS Press. 2019, p. 79.

[75] Pfleeger, C. P. *Security in computing*. Pearson Education India, 2009.

[76] Pollet, J. and Cummins, J. "Electricity for free? the dirty underbelly of scada and smart meters". In: *Proceedings of Black Hat USA* vol. 2010 (2010).

[77] Poulsen, K. "Slammer worm crashed Ohio nuke plant network". In: *http://www. securityfocus. com/news/6767* (2003).

[78] Raymond, K. "Reference Model of Open Distributed Processing (RM-ODP): Introduction". In: *Open Distributed Processing: Experiences with distributed environments. Proceedings of the third IFIP TC 6/WG 6.1 international conference on open distributed processing, 1994*. Ed. by Raymond, K. and Armstrong, L. Boston, MA: Springer US, 1995, pp. 3–14.

[79] Ross, R. S. et al. *Protecting Controlled Unclassified Information in Nonfederal Information Systems and Organizations [including updates as of 02-20-2018]*. Tech. rep. National Institute of Standards and Technology, 2018.

[80] Schäfer, J. and Poetzsch-Heffter, A. "JCoBox: Generalizing Active Objects to Concurrent Components". In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by D'Hondt, T. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 275–299.

[81] Schrecker, S. and al., et. *Industrial Internet of Things Volume G4: Security Framework*. Industrial Internet Consortium. https://www.iiconsortium.org/pdf/IIC_PUB_G4_V1.00_PB.pdf. 2016.

[82] Sommerville, I. "Software engineering 9th Edition". In: *ISBN-10* vol. 137035152 (2011), p. 18.

[83] Stallings, W. et al. *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA, 2012.

[84] Union, E. *The General Data Protection Regulation (GDPR)*. URL: https://ec.europa.eu/info/law/law-topic/data-protection_en (visited on 05/24/2016).

[85] Vardi, M. Y. and Wolper, P. "An automata-theoretic approach to automatic program verification". In: *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society. 1986, pp. 322–331.

[86] Vijayan, J. "Stuxnet renews power grid security concerns". In: *Computerworld* vol. 26 (2010).

[87] Wolper, P. "Temporal logic can be more expressive". In: *Information and Control* vol. 56, no. 1 (1983), pp. 72–99.

[88] Wolper, P., Vardi, M. Y., and Sistla, A. P. "Reasoning about infinite computation paths". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE. 1983, pp. 185–194.

[89] Woodcock, J. et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* vol. 41, no. 4 (2009), pp. 1–36.

[90] Yadav, N. *Know More About the System Development Life Cycle*. URL: https://darkbears.com/blog/know-more-about-the-system-development-life-cycle/ (visited on 03/29/2019).

[91] Yonezawa, A. *ABCL: An Object-Oriented Concurrent System*. Cambridge, MA, USA: MIT Press, 1990.

[92] Yonezawa, A., Briot, J.-P., and Shibayama, E. “Object-oriented concurrent programming in ABCL/1”. In: *ACM SIGPLAN Notices* vol. 21, no. 11 (1986), pp. 258–268.

[93] Yourdon, E. et al. *Mainstream objects: an analysis and design approach for business*. Yourdon Press, 1995.

[94] Zakrzewski, J. “Towards Verification of Ethereum Smart Contracts: A Formalization of Core of Solidity”. In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Piskac, R. and Rümmer, P. Cham: Springer International Publishing, 2018, pp. 229–247.

Part II
Scientific Contributions

CHAPTER **5**

Paper 1: Security and Privacy Functionalities in IoT

Authors: Elahe Fazeldehkordi, Olaf Owe, Josef Noll

Publication: Presented at 17th International Conference on Privacy, Security and Trust (PST), 26-28 August 2019 (pp. 1-12). Published in IEEE. DOI: https://doi.org/10.1109/PST47121.2019.8949054

Abstract

Internet of Things (IoT) offers a variety of technologies for connecting different kinds of heterogeneous devices. Security and privacy are becoming the main issue for IoT systems and their developers. Nevertheless, most works on IoT security and privacy requirements look at these requirements from a high-level view. Hence, the essential aspects of security and privacy functionalities will be disregarded, causing wrong design decisions. To combat this problem, this paper summarizes the most current documents related to security and privacy functionalities in the setting of IoT and provides a new taxonomy framework that organizes all aspects of security and privacy baselines, guidelines, and recommendations. To give an understanding of how the framework can help to improve security and privacy of IoT products, we combine it with a security classification method and demonstrate the usefulness by a case study of health products. Our framework can serve as a cornerstone towards the development of appropriate security solutions.

5.1 Introduction

Internet of Things (IoT) represents the concept of information flow among different kinds of embedded computing devices interconnected through the internet. The aim of IoT is to provide an advanced mode

of communication among the various systems and devices, and also to facilitate the interaction between humans and the virtual world. With this aim, IoT plays a significant role in the modern society and has applications in almost all fields including healthcare systems, automobile, industrial appliances, sports, homes, entertainments, environmental monitoring etc. IoT devices have already outnumbered the number of people at a computerized workplaces, and by 2020, connected “things" based on IoT will be around 212 billion [18, 22]. Those “things" will be daily used appliances like smart-phones, smart-watches, smart television, smart refrigerators, and others. As a result of this expansion, and as most things are connecting to the internet for exchanging information, IoT is vulnerable to various security issues and attacks (e.g., man in the middle attack, eavesdropping attack, denial of service attack, access attack, as well as major privacy concerns for the end users). Despite the advance abilities provided by IoT in the data communication area, its vulnerability implications from a security and privacy standpoint are still of great concern. Therefore appropriate steps in the initial phase of design and development of IoT systems should be taken.

In this paper, we focus on the comprehensive view of the state-of-the-art concerning security and privacy functionalities and requirements for IoT systems. Consequently, we suggest a complementary methodology for analyzing the functionalities in a comprehensive framework that can help both providers and consumers of IoT devices to have a better understanding of the security and privacy aspects. By functionality we mean: “The security and privacy-related features, functions, mechanisms, services, procedures, and architectures implemented within organizational information systems or the environments in which those systems operate” [24]. We explore how the framework can facilitate the process of improving IoT security and privacy, in combination with the security classification method suggested in [4, 26]. Security classification of a system will lead to a better understanding of the value of security and promote the extra cost of securing a system. An IoT system includes different devices, and protecting all of these devices at the same level is costly. It is economically impractical to employ all the security protection mechanisms for all the devices in a system. Dividing security into different classifications is necessary, to secure IoT systems to an appropriate level.

One attractive application area of IoT is health care [16, 21], where

IoT devices are becoming common. Medical applications like remote health monitoring, fitness programs, chronic diseases, elderly care, compliance with treatment and medication at home and by health-care providers are some of the important potential applications that can be facilitated by IoT. IoT-based health-care services can help to reduce costs, increase the quality of life, and enrich the users' experience. Therefore, we choose to demonstrate the framework on a case study concerning health products. Through the development of this framework together with security classification, extensive attention has been given to the requirements and limitations for securing IoT systems.

For this framework, we have investigated the most important IoT-related security baselines and guidelines developed by ENISA [10, 11], OWASP [19], Industrial Internet Consortium [25], Cloud Security Alliance[8], and Broadband Internet Technical Advisory Group [6], etc., as well as security and privacy guidelines from ISO [1, 2] and NIST [24], which could be relevant for securing IoT systems. With respect to privacy, the European Union (EU) has passed the general data protection regulation (GDPR), which regulates who can access private data, how and for what purpose, based on the consent of the data subject [11]. We have extracted the parts of these documents that deal with IoT and security/privacy, and then we have unified them using a common vocabulary, and have then categorized and integrated the resulting guidelines and requirements in a uniform style, and embedded them in a graphical representation by means of a tool based on diagrams.

Having a comprehensive view and taxonomy of security and privacy requirements and functionalities in IoT is a prerequisite for architecting optimal security solutions, designing, and developing secure and privacy-aware IoT systems. To give an understanding of how the framework can help to improve security and privacy in practice, we combine the framework with the security classification method of [4, 26], and demonstrate how the combined methodology can be used on a case study of health products.

The contribution of this paper is to present a new functionality framework for security and privacy of IoT systems, as outlined above, and show how it can be combined with the security classification method to analyse and evaluate the security and privacy weaknesses of IoT systems, and to reduce these weaknesses, as demonstrated in the case study. Systems are often made without the help of security and privacy

experts. Our framework is easy to follow, even for non-experts. The case study shows that by following our guidelines, one can detect security problems and get help in avoiding them.

The remainder of this paper is structured as follows: Section II explains IoT-related standards and guidelines. Section III provides related work. Section IV introduces the IoT security and privacy functionality framework. Section V explains the security classification method. Section VI describes the pacemaker case study, and Section VII concludes the paper.

5.2 IoT-Related Standards and Guidelines

Cloud Security Alliance [8] has provided considerations and guidance for designing and developing secure IoT devices. It aims to reduce some of the more common issues that can be found in the development of IoT devices. A number of activities that will enable a development organization to begin enhancing the security state of IoT devices have been outlined. This document has provided a graphical view of the steps needed in order to develop more secure IoT devices. Although IoT systems are complex, including devices, gateways, mobile applications, appliances, web services, datastores, analytics systems and more, the focus of this guidance is mainly on the “devices". In contrast, our work summarizes security and privacy functionalities considerations in the whole range of IoT system.

A security framework has been presented in Industrial Internet Consortium [25] which comprises of six interacting building blocks. These building blocks are organised into three layers. The top layer includes four core security functions, which are supported by a data protection layer and a system-wide security model and policy layer. The four core security functions are: endpoint protection, communication and connectivity protection, security monitoring and analysis, and security configuration management. And then they break down each layer into related key functions and explain the responsibility for each function. This document explains and positions security or related architectures, designs, and technologies. It also identifies procedures relevant to trustworthy Industrial Internet of Things (IIoT) systems. Security characteristics, technologies, and techniques that should be applied, and methods for

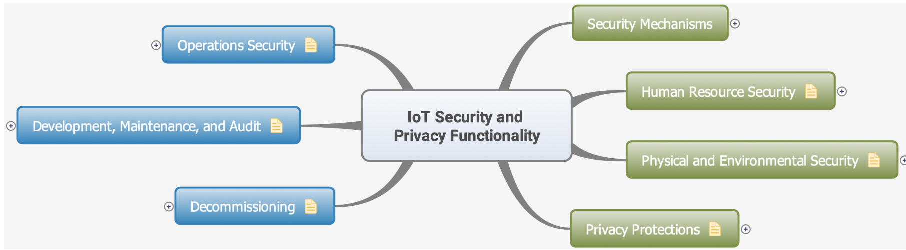

Figure 5.1: IoT security and privacy functionality framework

addressing security, have been described. However, it lacks some of the security functionalities, and in particular, it does not focus on privacy issues. The layer structure is a bit complicated, and we believe our framework has a more relaxed structure to use.

The ISO [1, 2] and NIST [24] standards are general requirements for establishing, implementing, maintaining and continually improving an information security management system, and protecting the confidentiality of Controlled Unclassified Information (CUI), respectively. We have extracted the IoT-related requirements from these standards and included these parts in our framework, while combining them with baselines and guidelines from other IoT-related documents, including OWASP and ENISA discussed below.

OWASP [19] presents guidance at a basic level, giving builders of IoT products a basic set of guidelines to consider from their perspective. The idea is that ensuring that these fundamentals are covered, will significantly improve the security of any IoT product. ENISA [10] elaborates baseline cybersecurity recommendations for IoT with a focus on Critical Information Infrastructures, which encompass facilities, networks, services, and physical and information technology equipment. Both of these guidelines, OWASP and ENISA, are addressing IoT, but both are presented in textual form, without defining a framework.

5.3 Related Work

In a work presented by Sicari et al. [27], the most relevant available solutions regarding security, privacy, and trust in IoT have been analyzed. Proposals related to security middleware, secure solutions for mobile devices, and ongoing international projects on this subject, have been

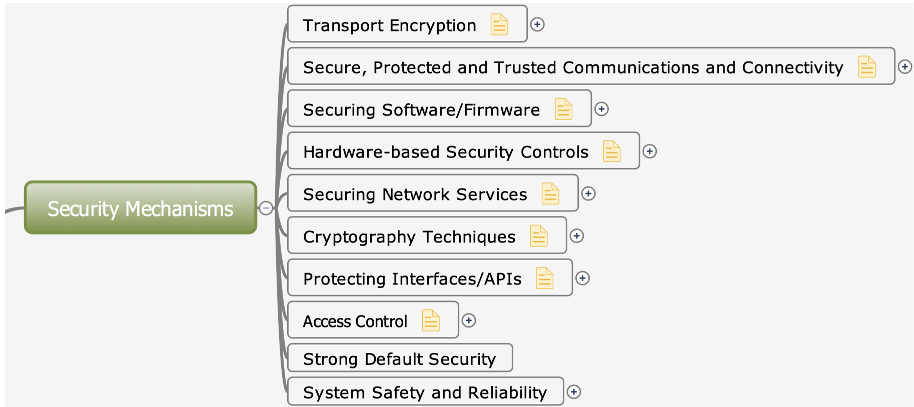

Figure 5.2: Security mechanisms

discussed in their work; however, the focus is more general, addressing authentication, confidentiality and access control, while we break down security and privacy requirements in more detail, using a framework of functionalities for all the baselines.

Main challenges and security threats in smart home networks have been analyzed by Lee et al. [17], and the fundamental requirements in order to provide secure and confidential operations in smart homes are explained from the results of their analysis. Although these requirements have been listed, they still lack practical solutions or recommendations in this matter. In [28], Suo et al. have deeply analyzed security architecture and features, and divided IoT systems into four key levels of architecture. According to this analysis, the security requirements for each level have been summarized. Furthermore, the research status of key technologies including encryption mechanism, communication security, protection of sensor data, and cryptography algorithms, have been discussed.

Roman et al. [23] have discussed threats faced by IoT, as well as security and privacy foundations based on objectives in a scenario involving a smart meter. However, they did not give any details about practical baselines and guidelines showing how to achieve these foundations.

Babar et al. [5] have presented a threat taxonomy and high-level security requirements for IoT, which like most of the other works only highlight these requirements without any practical recommendations for each category. And at the end, they introduced a security model

based on high-level requirements of security, privacy and trust. Related security requirements of IoT systems are discussed by Alqassem and Svetinovic [3], proposing a taxonomy of quality attributes, and some of the existing security mechanisms and policies in this matter have been reviewed, to reduce the identified security attacks and mitigate future vulnerabilities in these systems. They also have applied this taxonomy in a smart grid AMI as an IoT scenario. In contrast, our framework considers both security and privacy requirements and decomposes the related mechanisms, policies, and requirements with more details. In summary, our work provides a comprehensive view and a framework that covers all of the IoT security and privacy baselines, guidelines, and recommendations for every requirement.

5.4 Framework Explanation

In Figure. 5.1, we present an overview of the security and privacy functionality framework, including the top-level security and privacy concepts. The functionalities are separated into two major parts, the *life cycle* aspects of a system and the *management* aspects of security and privacy. The life cycle relates to the different phases in the life of a system, while the management of security and privacy is the ability to put supporting functionality elements in the system. We believe that awareness about where we are in the life cycle is essential, and makes it easier to apply the right functionalities and how to do the appropriate security and privacy management. We use blue color to distinguish subtopics related to the *life cycle* of a system from those associated with the *management* of security and privacy, colored in green. The coloring makes the division clear, and gives a better structure of the framework, separating the two primary concerns.

In the following, we describe each part of the framework and the related subtopics. For brevity, we do not expand the whole framework and just mention some of the aspects. For more details and complete expansion of the framework refer to the long version [12].

Security mechanisms - Different security mechanisms are illustrated in Figure. 5.2. Security mechanisms are processes designed to detect, prevent or recover from a security attack in IoT devices, including:

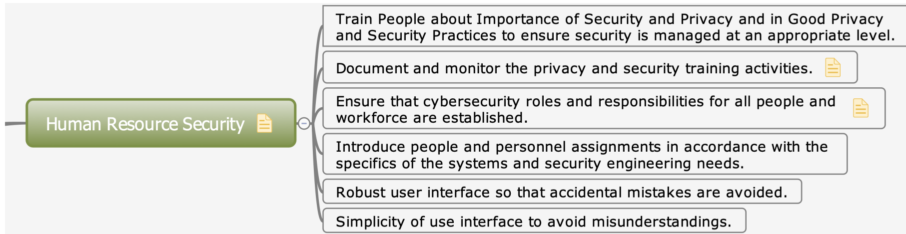

Figure 5.3: Human resource security

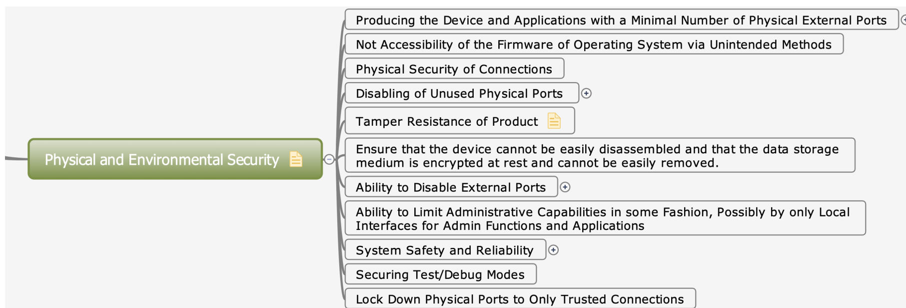

Figure 5.4: Physical and environmental security

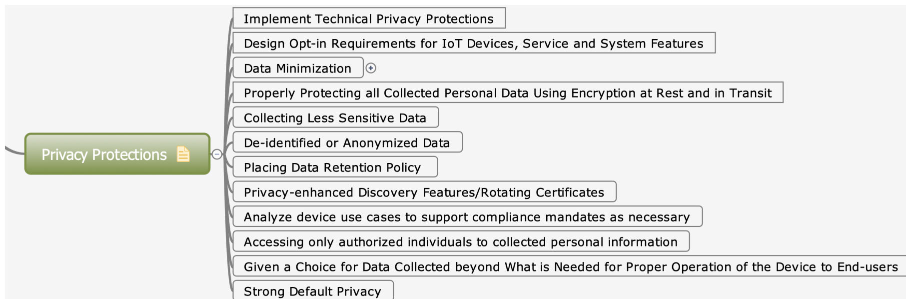

Figure 5.5: Privacy protection

- *Secure, protected and trusted communications and connectivity:* This includes information flow protection, standardising security protocols (i.e., Transport Layer Security (TLS) for encryption) guaranteeing data authenticity, signing data, disabling specific ports and/or network connections for selective connectivity, etc.
- *Hardware-based security controls:* product developers should evaluate and implement hardware protection mechanisms, including the use of Memory Protection Units (MPUs), considering a Trusted

Platform Module (TPM) into IoT Devices, securing physical interfaces, tamper protections, etc.

- *Protecting interfaces/application programming interface (APIs):* Interface security is one of the critical tasks when it comes to developing IoT devices. IoT products interact with so many cloud services, custom-developed smartphone apps and also peer IoT products. If APIs do not protect adequately, service providers might be exposed. APIs must protect adequately against misuse, by techniques like rate-limiting to protect against compromised IoT devices that attempts to flood the service, error handling, embedding time-stamps or counters into messaging to protect against replay attacks, certificate pinning to protect against sensitive data transmission into GET requests, etc.
- *Access control:* only authorized users should have access, applications and services and unauthorized accesses should be prevented, user accountability should be enabled to safeguard their authentication information.

Human resource security - People and contractors should understand the cybersecurity responsibilities suitable for their roles, and be trained about the importance of security and privacy. Further, to avoid misunderstanding, the user interfaces should be simple yet robust enough to avoid accidental mistakes. See Figure. 5.3.

Physical and environmental security - The objectives of this section include prevention of unauthorized physical access, damage, and interference with IoT's information and premises, as well as prevention of loss, damage, theft or compromise of assets and interruption to the activities and operations of IoT devices and systems. All the equipment and processing facilities should be placed in secure areas and protected from physical and environmental threats. The functional requirements of this matter are listed in Figure. 5.4.

Privacy protection - Personally identifiable information (PII) needs to be protected, according to the European General Data Protection Regulation (GDPR) regulations [30]. Privacy protection is also advisable to increase trust in the internet (see Figure. 5.5 for practical requirements).

Operations security - Information processing facilities should ensure correct and secure operation, including protection against attacks. Further, accountability auditing must be enabled for all events to ensure the

Figure 5.6: Operations security

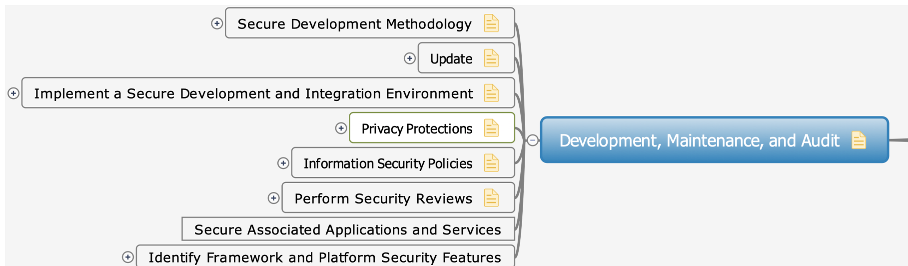

Figure 5.7: Development, maintenance, and audit

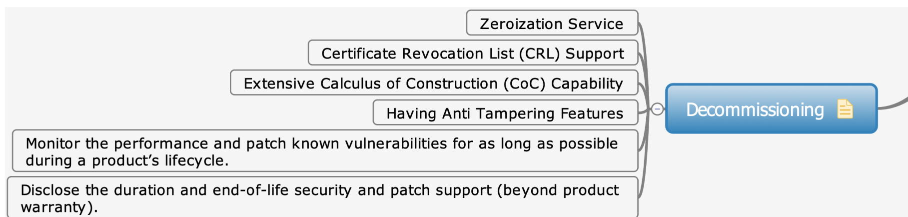

Figure 5.8: IoT security and privacy functionality framework – decommissioning

integrity of operational systems, and prevent against exploitation of technical vulnerabilities. See Figure. 5.6.

- *Logging and monitoring:* IoT products should have sufficient observations of occurrences happening on the device. For instance, connection requests, authentications (successful or failed), physical tamper, account updates, etc. It is essential to be able to monitor users' interaction with the system or a device, and in particular when they fail to login. Therefore, to detect possible security and integrity errors or potential threats, data should be captured on the entire state of the system from the endpoints, connectivity traffic and verifying the device behavior, in addition to analysing it.

- *Security configuration and management:* Includes the control of modifications to the operational functionality of the system (which covers reliability and safety behaviors) along with the security controls ensuring its protection,
- *Trust and integrity management:* Some of the practices in this matter include the following.
  1. Establishing trust in the boot environment before anything else since both the main hardware components and the operating system have been initialized by the boot process,
  2. Signing code cryptographically to prevent tampering,
  3. Controlling the installation of software on operating systems to prevent loading unauthorized software and files onto it, etc.
- *Management of security vulnerabilities and/or incidents:* To ensure a quick, effective and orderly response to information security incidents, management procedures should be established. To address identified vulnerabilities, disclosure of vulnerabilities should be coordinated. Participate in information sharing platforms, in order to report vulnerabilities and receive timely and critical information about current cyber threats and vulnerabilities from public and private partners, is recommended.

Development, maintenance, and audit - To ensure that security controls are efficient, audits and reviews for security controls should be organized periodically. Penetration tests also should be done regularly. Good practices in this area are shown in Figure. 5.7.

- *Secure development methodology:* Documentation, peer reviews, and incorporating security requirements into the product life cycle should be included, in addition to the technological checks. Additionally, essential feedback loops should be included in the engineering process to create more secure IoT products.
- *Update:* Ensuring a secured system update is probably one of the biggest challenges in an IoT life cycle. While initial systems are subject to secure testing on both the producers and the customers, a similar awareness is often missing for system updates. In the absence of sufficiently secured update, an intruder may change legitimate software and firmware, and put new malicious software and firmware into the device. Malicious software and firmware can disable security controls, apply new features or build

data exfiltration mechanisms. End-to-end protection of software and firmware is essential to the whole life cycle. And so are permissions regarding the update process, the integrity of updates, and authentication of update transactions of software and firmware.

- *Information security policies:* Regulatory, organizational and machine levels of security are covered here. The purpose of security in a system come as a security policy, and the security model represents security policies which should formally apply to the system. A security model and policy should state how to protect endpoints, communications and data, and specify what should be monitored, and analyzed, etc.
- *Perform security reviews:* Continuous feedback loops (i.e., the link connecting design, development, and test) and optimization during the life cycle of an IoT product are essential in this practice area. Identified faults/vulnerabilities have to go back to the design and threat modeling process, hardware and software baselines must be updated accordingly, and then be tested again to make sure that the patches do not identify new vulnerabilities. These vulnerabilities might be detected using IoT device security testing processes. Tests like Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), Interactive Application Security Testing (IAST) are a recommendation.
- *Secure associated applications and services:* Applications (Apps) and services connected to IoT devices must be developed securely. Configuring IoT devices, or interacting with IoT devices are usually being done using smartphone apps. These apps also create gateway functionality to transfer data from IoT devices to the cloud. So developers must use security credentials in order to provide authenticated and integrity protected communications to IoT devices.
- *Implement a secure development and integration environment:* A framework, addressing both physical and IT-security, is required to ensure a controlled environment for software and hardware development.

Decommissioning - To prevent exposing critical information to any possible attacker, the product must be disposed in a secure way at the end of life time. Therefore, secure devices should not be placed into the

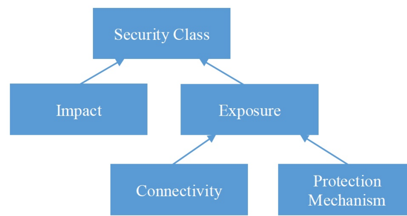

Figure 5.9: Basic inputs for defining a security class [26]

supply chain again. A low-cost and high-guaranty way to decommission can be provided by an automated decommissioning procedure. We show the practical considerations of decommissioning in Figure. 5.8.

5.5 Security Classification

Concepts of security classes have been suggested and defined in [4, 26]. In this section, we briefly touch on these concepts to give an overview. There are six classes of security, from A to F, with A representing the best security and F representing the least security. Further, we based these security classes on the *exposure* and the *impact* factors of the possible attacks on the system (see Figure. 5.9 and Table 6.2). A lower exposure level means a lower attack surface. Therefore, attacks that have low exposure are relatively safe and vice-versa. The high impact of attacks on a system affects the security class of the system, and necessary precaution should be considered to protect the system. Consequently the security class of the system will increase. A system with low exposure and low impact is relatively safe.

Impact is a consequence of the possible attacks on a system. When a system is compromised, it can have an impact on several sectors beyond the system itself, including business, government, or society. We divided this *impact* into five levels, namely, Insignificant, Minor, Moderate, Major, and Catastrophic. Defining each of these levels depends on the system under evaluation, the type of impacts it can have (e.g. financial, social), or the application area. Therefore, we base the security classification level assigned to the system on the security/risk analyst responsible for that particular application or system discretion.

Exposure (see Table 6.1) is a consequence of connectivity and the

protection level of a system. According to the connectivity of the system, an appropriate set of security and privacy functionalities is identified to protect the system, while the strength of the identified security and privacy functionalities determines the protection levels (i.e., for authentication we can use passwords or PINs or two/multi-factor authentication). The definition of the protection levels (P1 to P5) is according to the ISA99 standard. However, the protection level evaluation depends on the expert and the particular scenario considered. The connectivity is divided into five levels, C1 to C5, according to ANSSI [4]:

- (C1): a closed and isolated Information & Communication System (ICS)
- (C2): an ICS connected to a corporate Management Information System (MIS) for which operations from outside the network are not allowed
- (C3): an ICS connected to wireless technology.
- (C4): an ICS with *private* infrastructure permitting operations from outside (VPN, APN, etc.)
- (C5): a distributed ICS with *public* infrastructure.

Increasing the protection level or reducing the connectivity level can reduce the exposure (see Table 6.1). From Tables 6.2 and 6.1 we observe that by keeping the protection level in the highest level (P1 is the lowest

Table 5.1: Security classes (from [26])

Impact					
Catastrophic	Class A	Class D	Class E	Class F	Class F
Major	Class A	Class B	Class D	Class E	Class F
Moderate	Class A	Class B	Class C	Class E	Class E
Minor	Class A	Class B	Class B	Class C	Class D
Insignificant	Class A	Class A	Class A	Class B	Class C
	E1	E2	E3	E4	E5
	Exposure				

Table 5.2: Exposure (from [26])

Protection					
P1	E4	E4	E5	E5	E5
P2	E3	E3	E4	E4	E4
P3	E2	E2	E3	E3	E3
P4	E1	E1	E2	E2	E2
P5	E1	E1	E1	E1	E1
	C1	C2	C3	C4	C5
	Connectivity				

and P5 the highest protection level), exposure will be in the lowest level (E1), therefore resulting in the highest security class. And the way we can provide the highest protection level is by applying an effective and appropriate set of security and privacy functionality criteria in a particular device, for instance, use of multi-factor authentication instead of just passwords or PINs for authentication. Adequate and proper sets of security and privacy functionality criteria used in the case study have been taken from our framework.

5.6 Pacemaker Case Study

We have built a methodology for looking at the functionalities from both security and privacy points of view. In order to understand how we can use the functionality framework to improve security and privacy of IoT systems in practice, we here use a case study of health products involving a pacemaker and related control units such as a mobile phone and a heart rate sensor. In the domain of health services, the highest security class is recommended for devices like pacemakers that directly control life functions of a patient.

A pacemaker is a medical device that is implanted under the skin to help with abnormal behaviors of heartbeats [7, 15]. It consists of a battery, a computerized generator, and wires with sensors at their tips [14]. The generator is powered by the battery, and both are surrounded by a thin metal box. The generator is connected to the heart by the wires. The pacemaker helps to monitor and control the heartbeat. The sensors detect the heart's electrical activity and send data through the wires to the computer in the generator. If the heart rhythm is abnormal, the generator will be directed by the computer to send electrical pulses to the heart, and the pulses travel through the wires to reach the heart. Embedded microprocessors in modern pacemakers have enabled them to do additional tasks like monitoring heart activity and providing a record for the patients and their healthcare providers, as well as collecting data on heart functions to help doctors to identify and diagnose patient conditions, and send required shock signals when needed. Doctors can monitor the patient's heart activities and control the pacemaker using a mobile phone or a computer device, or send required shock signals in case of observation of abnormal behavior.

Beside threatening patients' lives, malicious attackers can get access to patients' medical records through a pacemaker [9, 29], or track a patient's location. In addition, malicious software can be run on pacemakers and cause security and privacy breaks. Therefore, any security or functional weakness can result in a security failure. Like any device that uses remote technology, pacemakers are also vulnerable against cyber attacks, and hackers can break into the pacemaker itself, the back-end systems or the communication between the pacemaker and its surrounding. By breaking into a pacemaker, an attacker can send strong shock signals, disturb the pacemaker setting or heart functions, or disturb them from working properly. One simple example of hacking into a pacemaker is to change the setting from battery-saving “sleep" mode to “standby" mode, and this can quickly drain the battery, which is normally supposed to last for years. Furthermore, an attacker can steal private and personal information of a patient from the device that for instance can later track the patient's location. Hence, security in this kind of device is crucial and should have the highest (best) security class, meaning security class A.

We will below discuss the security and privacy challenges for each of the three devices. In each case we discuss connectivity, protection level and relevant functionality criteria. We use the general criteria given in the functionality framework, select and discuss the parts relevant for each device.

Pacemaker security controller. We consider Connectivity 2 (explained in Section 6.3) for the pacemaker security controller, since it is only connected to the pacemaker. We define below the protection levels which are relevant for the pacemaker security controller:

- Protection level 1 (P1): includes secure authentication, securing software/firmware, secure communication, and human interface security. For authentication we consider: requiring passwords, option to change the default username and password. For communication we consider: data authenticity to enable reliable exchanges from data emission to data reception. For securing software/firmware we consider: update capability for *some* of the system devices and applications, transmitting the files using encryption.

- Protection level 2 (P2): includes P1 and in addition, for authenti-

cation: requiring strong passwords, securing password recovery mechanisms, making sure that default passwords and even default usernames are changed during the initial setup, and that weak, null or blank passwords are not allowed. For communication: verifying any interconnections, discover, identify and verify/authenticate the devices connected to the network before trust can be established, and preserve their integrity for reliable solutions and services, prevent unauthorised connections to it or other devices the product is connected to, at all protocol levels, providing communication security using state-of-the-art mechanisms, standardising security protocols, such as TLS for encryption. For securing software/-firmware: encrypting update files for *some* of the applications.

- Protection level 3 (P3): includes P2 and in addition, for authentication: having options to force password expiration after a specific period, and to change the default username and password, making sure that the password recovery or reset mechanism are robust and do not supply an attacker with information indicating a valid account. The same should apply to key update and recovery mechanisms. For communication: data authenticity to enable reliable exchanges from data emission to data reception.
- Protection level 4 (P4): includes P3 and in addition, for authentication: implementing two-Factor Authentication (2FA), making sure that default passwords and even default usernames are changed during the initial setup. For communication: signing the data whenever and wherever it is captured and stored, making intentional connections, disabling specific ports and/or network connections for selective connectivity. For securing software/firmware: capability of quick updates when vulnerabilities are discovered for *some* of the system devices and applications, and offering an automatic firmware update.
- Protection level 5 (P5): includes P4. In addition, for authentication: using Multi-Factor Authentication (MFA) (considering biometrics for authentication), considering Certificate-Less Authenticated Encryption (CLAE), and User Managed Access (UMA). For communication: rate limiting – controlling the traffic sent or received by a network to reduce the risk of automated attacks. For securing

software/firmware: update capability for *all* system devices and applications, capability of quick updates when vulnerabilities are discovered for *all* system devices and applications, encrypting update files for *all* applications, signing update files and validating by the device before installing, securing update servers, having ability to implement scheduled updates, having backward compatibility of firmware updates.

According to Tables 6.2 and 6.1, we might have protection levels 2, 3, 4, or 5 to obtain security class A; however we choose to use protection level 3 since it is a realistic level in this case. A higher level would be costly, and P2 would give poor protection. We therefore consider how to obtain that protection level, and select the following set of relevant functionality criteria from the functionality framework, adapted to the challenges of pacemakers. In this selection we have used the guidelines for securing pacemakers from [13]. This gives a certain guarantee that we cover all relevant aspects.

- *Secure authentication/authorization*: Access to the pacemaker security controller and mobile phone connected to the doctor, should be limited using the authentication of users (for example user ID and password, Personal Identification Numbers (PINs), biometric authentications). Authorization refers to checking necessary permissions of an identified individual to do an action. Authentication and authorization are completely related to each other. Authorization checks should immediately be followed by authentication of a request. In order to have secure authorization, the roles and permissions of the authenticated user should only be verified through information in backend systems, not through roles or permission information coming from the device. Any incoming identifiers with a request alongside should be verified by backend code independently. Failure in a secure authentication/authorization would give access to unauthorized people and could lead to reputational damages, fraud, unauthorized access to information, information theft, and modification of data.
- *Securing software/firmware*: Before any software or firmware update, user authentication or other suitable controls should be required. Software/firmware updates should be restricted to authorised code. Manufacturers may consider code signature

verification as an authentication method.

- *Secure communication*: Data transmission between the pacemaker security controller and the mobile phone connected to the doctor must be secure enough so that a third party cannot listen to their communication. The communication should not be vulnerable to eavesdropping or interception. Failure in having a secure communication can cause identity theft, fraud, data modification, privacy information leakage. One should consider strong handshaking, correct SSL versions, no clear-text communication of sensitive assets, etc.

- *Human interface security*: Patients should be trained about the importance of security and privacy and how to use the pacemaker properly to ensure security is managed at an appropriate level. For instance, if we consider all the security protections in the highest level in the security controller, but the patient does not know how to deal with error notifications (restart, turn off or low battery errors) in the security controller, all the security considerations in the controller will be useless.

- *Data privacy*: Measures to avoid risk of breaches in connection with long term storage of private information, handling of encryption of private information, and GDPR compliance including consent, purpose, and access rights.

The final discussion of the security class of the pacemaker system depends on the scenario chosen and the other components involved. We next consider the mobile phone.

Mobile phone. In our case study, a mobile phone is used in the communication between the pacemaker and the healthcare provider/doctor. Security in this mobile phone is then important in order to send correct data to/from the pacemaker controller. Hacking or tampering into this mobile phone can result in sending wrong data from the pacemaker to the doctor, something that could result in wrong decisions from the doctor, or possibly sending inappropriate shocks to the patient's heart.

Therefore, it is important to secure the mobile phone properly. However, the security class of the mobile phone is only considered class B, since mobile phones are inherently not of the highest security class

and have a number of possibilities for attacks due to high exposure. For instance, because of all the applications and browsers on the device, as well as software developed by third parties. Therefore, both the impact of attack as well as the connectivity are in a higher level for the mobile phone compared to the pacemaker. Hence the security class of the mobile phone is lower than the security class of the pacemaker security controller.

We consider C4 in the mobile phone connected to the doctor. According to Tables 6.2 and 6.1, we might have protection levels 2, 3, or 4 to obtain security class B, however since mobile phones naturally have moderate impact of attacks as discussed above, we should use protection level 4 in order to reach security class B. So based on that the following set of relevant security and privacy functionality criteria are selected to ensure that the mobile phone has a sufficiently high protection level, following the functionality framework and the OWASP guidelines for securing mobile phones [20]:

- *Secure authentication/authorization* and *secure communication*: are the same as what we discussed for the pacemaker security controller.

- *Sufficient cryptography techniques*: Appropriate cryptography techniques should be considered for instance using AES instead of DES.

- *Code tampering*: When an application is on the device, the code and data resources are also available there. An attacker modify the code through either malicious apps in third party app stores or trick the user via phishing attacks and install the app on the device. Code tampering could result in revenue loss due to piracy, reputational damage, unauthorized new features, identity theft or fraud.

- *Secure data storage*: Failure in having secure data storage can result in data loss, extraction of sensitive data using malware, modified apps or forensic tools, identity theft, fraud, reputation damage, material loss or external policy violations.
- *Proper platform usage*: Misuse of a platform feature or failure to use platform security controls fall under this category. Android intents, platform permissions, misuse of TouchID, Keychains, or other security controls which are part of the operating system could

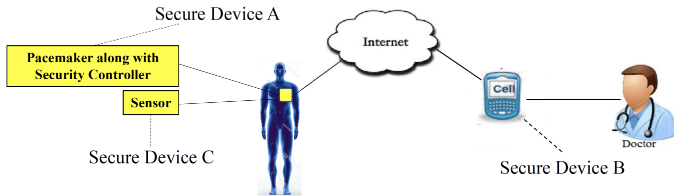

Figure 5.10: Scenario 1: pacemaker along with security controller

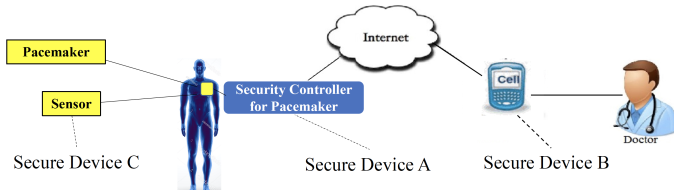

Figure 5.11: Scenario 2: pacemaker with separate security controller

be included. To prevent the attacks in this category, secure coding and configuration practices must be applied on the server side of the application.

- *Data privacy*: are as discussed above for the pacemaker security controller.

We next consider the protection levels of ISA99, and select the relevant criteria from the functionality framework, and adjust them for the case of the mobile phone connected to the doctor. These are defined below:

- Protection level 1 (P1): includes secure authentication, secure authorization, securing software/firmware, sufficient cryptography techniques, code tampering, secure data storage, secure communication, and proper platform usage. Secure authentication, securing software/firmware, and secure communication are the same as the considerations for protection level 1 for the pacemaker security controller.
- Protection level 2 (P2): includes P1. Secure authentication, securing software/firmware, and secure communication are the same as the considerations for protection level 2 for the pacemaker security controller. In addition, for cryptography techniques we should consider: ensuring proper selection of standard encryption algorithms and keys.
- Protection level 3 (P3): includes P2. Secure authentication,

securing software/firmware, and secure communication are the same as the considerations for protection level 3 for the pacemaker security controller. In addition, for cryptography techniques we should consider: strong encryption algorithms, strong keys.

- Protection level 4 (P4): includes P3. Secure authentication, securing software/firmware, and secure communication are the same as the considerations for protection level 4 for the pacemaker security controller. In addition, for cryptography techniques we should consider: verifying the robustness of the implementation, establishing secure and scalable key management. And cryptographic keys must be securely managed.
- Protection level 5 (P5): includes P4. Secure authentication, securing software/firmware, and secure communication are the same as the considerations for protection level 4 for the pacemaker security controller. In addition, for cryptography techniques we should consider: disabling insecure protocols.

Finally, the security class of the heart rate sensor in our case study is considered below.

Heart rate sensor. The heart rate sensor does not have any external connectivity; it has only connections to the heart and computerized generator inside the pacemaker in order to transfer the captured heart rate from the heart to the pacemaker generator. So the connectivity in the sensor is C1. Furthermore it has only a chip set from the company provider. Hence, the sensor has the least chance of attack. However, there is still a possible vulnerability if the wired connection is not sufficiently shielded and allows eavesdropping by inductive sensors or senders. But this requires very short physical distance. Therefore we may assume only minor impact, and do not need a high protection level for this device, and may use P1. Exposure is then E4 (see Table 6.4). This gives security class C.

Pacemaker scenarios. The challenge in the case study is how to implement the security controls like decryption, authentication, etc. For instance, whether to do data decryption in the mobile phone, and then send the decrypted data to the pacemaker. If the mobile phone is compromised, wrong data and signals could then go to the doctor and pacemaker, since mobile phones can get compromised, as we discussed before. Whereas, if we consider security controls in the pacemaker itself, we only have software provided by one company. Thus it would be much

Table 5.3: Security and privacy challenge comparison of scenario 1 and 2

	Security and Privacy Challenges	Scenario 1	Scenario 2
In Sensor	Draining battery by changing battery saving controls	Major	Insignificant
	Interrupting into heartbeat capturing	Catastrophic	Minor
	Data and private information breaches	Minor	Minor
In Pacemaker	Sending/Receiving wrong data to/from mobile phone connected to the doctor	Major	Insignificant
	Changing pacemaker shock settings	Major	Insignificant
	Draining battery by changing battery saving controls	Major	Insignificant
	Higher battery usage	Catastrophic	Insignificant
	Data and private information breaches	Major	Insignificant
	Transparency of GDPR compliance	Major	Insignificant
	Risk of long-term storage of private information	Major	Insignificant
In Mobile Phone Connected to the Doctor	Sending/Receiving wrong data to/from pacemaker	Minor	Minor
	Sending wrong unnecessary/necessary shocks	Minor	Minor
	Changing pacemaker shock settings	Minor	Minor
	Draining pacemaker battery by changing battery saving controls	Minor	Minor
	Data and private information breaches	Minor	Minor
	Transparency of GDPR compliance	Minor	Minor
	Risk of long-term storage of private information	Minor	Minor

Severity of Challenge: Insignificant: (green) Minor: (yellow) Moderate: (orange) Major: (brown) Catastrophic: (red)

more secure to do the security controls directly in the pacemaker, and use the mobile phone just as a gateway to transfer the information. However, this will require additional computational power and battery capacity. Another issue is whether we can do all the security controls inside the computerized generator of the pacemaker - or consider a security controller as a separate unit out of the body with a close and secure connection to the pacemaker.

We therefore define two scenarios: In scenario 1 (see Figure. 6.1), the security controls for the pacemaker are done inside the computerized generator of the pacemaker, and in scenario 2 (see Figure. 6.2), a separate security controller unit makes the security controls of the pacemaker, such that this unit is outside the body with a close and secure connection to the pacemaker. In scenario 1, we would need computational power inside the body, needing more storage, stronger CPU, much more battery capacity and so on. This is not desirable since it might increase the potential necessary surgeries in order to change the battery, maintain, or update the pacemaker. Moreover, we might not be able to consider some of the security and privacy functionalities in order to avoid increasing CPU usage, which results in even more battery usage.

Therefore, we might want to have a pacemaker with a simple sensor

inside the body and a security controller out of the body, say in the pocket or at home very close to the pacemaker. Here, it is essential that the security controller be close to the pacemaker, since the pacemaker must have very weak signals, limited interactions and computations, to avoid using too much battery power and resulting battery changes. So, all the security and battery-intensive controls would be done in the external security controller, which can have a stronger and easily rechargeable battery. The controller can maintain the device in case of any problem, update or troubleshoot its computer system, or even increase the level of protection by adding more security and privacy functionalities or appropriate software at any time because of easy access.

According to all the security considerations in scenario 2, we can then have a better security class in the pacemaker security controller: In Table 6.3 we summarize all the security and privacy challenges in the sensor, pacemaker, and mobile phone connected to the doctor, and compare the severity of these security and privacy challenges in scenarios 1 and 2. This table determines the impact. The challenges considered in the table are found by following the functionality framework, for each device, in a top-down manner and in each case determining the problems that may occur. The further we break down the problems according to the functionality framework, the easier it is to find the specific challenges for the given device.

In the following, we see that the severity of the security and privacy challenges in the sensor and pacemaker has been reduced from very high in the worst case to low. Hence, the impact of attacks is reduced from catastrophic to minor.

Discussion. The pacemaker in scenario 1 is a complex device because all security controls are done inside the computerized generator of the pacemaker. The security controller is very close to the sensor, the signals received by the sensor from the pacemaker are very frequent. Interference is possible and that can have negative effect on correct heartbeat capturing. However, by transferring the security controller outside of the body this effect would be low (still there are some frequencies from devices close by such as mobile phones that can affect the sensor) resulting in more precise heartbeat capturing. As mentioned earlier in the heart rate sensor, there are low possibilities of data and private information breaches in both scenarios.

In the pacemaker in scenario 1, because of the complexity of the sys-

tem and having all the security controls inside the computerized generator of the pacemaker, we need higher CPU and memory consumption and then battery usage would be very high, while in scenario 2 this problem would decrease to very low. In scenario 1, because of difficulties in accessing the pacemaker and its security controller on time (in case of maintenance, update, troubleshooting or installing new software/equipment, also not being able to apply all the necessary criteria with high protection level in order to avoid high battery usage), the level of the security and privacy functionality criteria as well as the level of the protection is low.

Therefore the vulnerability of the pacemaker and sensor against attacks compromising the device (that can cause changing battery saving controls so draining the battery more quickly, changing pacemaker shock settings, tampering of transferred information from/to the pacemaker, data and private information breaches, transparency of GDPR compliance, risk of long-term storage of private information) is high; however, this problems decrease to very low when we change to scenario 2, due to easier access to the security controller out of the body.

By compromising the mobile phone connected to the doctor, the problems listed in the last part of Table 6.3 may occur. In both scenarios, we have considered high level of protection for the mobile phone by

Table 5.4: Security class of the sensor

Protection	Connectivity C1
P1	E4
P2	E3
P3	E2
P4	E1
P5	E1

Impact	Exposure E4
Catastrophic	Class F (In Scenario 1)
Major	Class E
Moderate	Class E
Minor	Class C (In Scenario 2)
Insignificant	Class B

Table 5.5: Security class of the pacemaker security controller

Protection	Connectivity C2
P1	E4
P2	E3
P3	E2
P4	E1
P5	E1

Impact	Exposure E2
Catastrophic	Class D (In Scenario 1)
Major	Class B
Moderate	Class B
Minor	Class B
Insignificant	Class A (In Scenario 2)

Table 5.6: Security class of the mobile phone

Protection	Connectivity: C4
P1	E5
P2	E4
P3	E3
P4	E2
P5	E1

Impact	Exposure: E2
Catastrophic	Class D
Major	Class B
Moderate	Class B
Minor	Class B
Insignificant	Class A

In Scenarios 1 & 2

using appropriate set of security and privacy functionality criteria with high protection level, therefore we have reduced the vulnerability of the device to low, but as discussed earlier, these devices still have a number of vulnerabilities for attacks, and therefore the vulnerability is not in a very low level.Furthermore, softening the problems in scenario 1 results in obtaining higher protection level and security class. For instance, we can easily recharge the battery of the security controller, maintain the device in case of any problem to avoid shutting down all the security considerations, and increase the level of protection by updating or troubleshooting its computer system..

In the sensor, as explained above, we have connectivity C1, and protection level P1, therefore exposure is E4 (see Table 6.4), and because of very high problem severity, the impact of attacks is Catastrophic, resulting in security class F in this device. By changing from scenario 1 to 2, the severity of the security and privacy challenges has reduced from very high to low, therefore the impact of attacks is reduced from Catastrophic to Minor, and consequently, the security class has improved from class F to class C.

In the pacemaker, we have connectivity C2, and protection level P5, however in scenario 1, the severity of security and privacy challenges have affect on the protection level, which falls to level 3, therefore exposure is E2 (see Table 6.5), and because of very high security and privacy challenge severity, the impact of attacks is Catastrophic, then we have security class D in this device. By changing from scenario 1 to 2, the severity of the security and privacy challenges has reduced from very high to very low, therefore the impact of attacks has reduced from Catastrophic to Insignificant, and consequently, the security class has improved from class D to A.

And, in the mobile phone connected to the doctor, we have connectivity C4, and protection level P4, therefore the exposure is E2 (see

Table 6.6), and the security is class B in both scenarios. By considering security and privacy challenges, as well as the effect of these challenges on the protection level of each device and the whole system, and also their affect on the impact of attacks, we have changed scenario 1 to 2, and were able to improve the security class in the sensor and pacemaker security controller from class F to C, and class D to A, respectively. Hence the security of the overall system has improved significantly.

In this case study, we started out with security and privacy design requirements to each device: class A for the pacemaker security controller, class B for the mobile phone, and class C for the sensor. We have seen that this is not realistically possible to achieve for the design of scenario 1, while it was possible to satisfy these requirements for scenario 2. The framework was most useful in these design evaluations. Indeed, the case study shows that by following the guidelines given by our framework, one can achieve security easily and decrease the impact of a possible attack.

5.7 Conclusion

The expansion of IoT in the last decade has resulted in several security and privacy issues and attacks against things and people. Unfortunately, the security and privacy functionalities to combat these attacks are not well-recognized in the domain of IoT. This paper summarizes and categorizes IoT security and privacy functionalities, and as the main contribution, the paper presents a new taxonomy framework that organizes the related standards in this area. The proposed framework is oriented towards practical application. We have demonstrated the application of this framework in combination with the security classification method using a case study about pacemakers. Our case study is quite generic and reveals general issues that can be found in other case studies.

The security class of a system is based on two factors: exposure and impact of possible attacks. The exposure is a consequence of the protection level of the system and its connectivity. A lower exposure level means a lower attack surface. Therefore, by reducing the exposure level of a system we can have a more secure system. A higher protection level in a system or lower connectivity can result in lower exposure. At the same time, lower impact of attacks on a system raises the security class

of the system. By applying the appropriate set of security and privacy functionality criteria from our framework, the protection level of a system can increase, while exposure can decrease, as demonstrated by the case study and discussed at the end of Section 6.4.

The main objective of this paper is to give security developers, designers, and end-users an opportunity to understand and explore what the IoT security and privacy functionalities are, and how these functionalities can help to improve the security and privacy of IoT systems. Our approach combines detailed information about security and privacy functionalities with a security classification method. The approach is systematic and structured; it is easy to use and is oriented towards practical engineering. The framework is based on the available recommendations and standards for IoT systems. This should imply that all aspects are covered, but there is no guarantee for that. This can be seen as a limitation of the work. Secondly, the application of the methodology is ultimately depending on the judgements of software engineers or security experts, and is therefore not 100% precise. If their judgement is wrong, for instance if they choose the wrong connectivity or protection level, it will in general give a wrong estimate.

Future work will consider case studies with several kinds of IoT devices and sensors involved. This will be a valuable step toward validating our framework, and possibly allowing the framework to be complemented with even more elements.

Bibliography

[1] 27000, I. *Information technology — Security techniques — Information security management systems — Overview and vocabulary (fourth edition)*. ISO/IEC 2016. 2016.

[2] 27001, I. *INTERNATIONAL STANDARD ISO/IEC 27001 Information technology—Security techniques —Information security management systems —Requirements*. ISO/IEC 2013. 2013.

[3] Alqassem, I. and Svetinovic, D. "A taxonomy of security and privacy requirements for the Internet of Things (IoT)". In: *2014 IEEE Intern. Conf. on Industrial Engineering and Engineering Management (IEEM)*. IEEE. 2014, pp. 1244–1248.

[4] ANSSI. *Classification Method and Key Measures*. 2014.

[5] Babar, S. et al. "Proposed security model and threat taxonomy for the Internet of Things (IoT)". In: *Intern. Conf. on Network Security and Applications*. Springer. 2010, pp. 420–429.

[6] BITAG. *Internet of Things (IoT) Security and Privacy Recommendations*. https://www.bitag.org/documents/BITAG_Report_-_Internet_of_Things_(IoT)_Security_and_Privacy_Recommendations.pdf. 2016.

[7] Boulos, P. et al. "Pacemakers: a survey on development history, cyber-security threats and countermeasures". In: *Int. J. Innov. Stud. Sci. Eng. Technol* vol. 2, no. 8 (2016).

[8] *Future-proofing the Connected World: 13 Steps to Developing Secure IoT Products*. https://downloads.cloudsecurityalliance.org/assets/research/internet-of-things/future-proofing-the-connected-world.pdf. 2016.

[9] Dadourian, M. *Heart alert: Pacemakers can be hacked*. 2018.

[10] ENISA. *Baseline Security Recommendations for IoT in the context of Critical Information Infrastructures*. European Union Agency for Network and Inf. Sec. 2017.

[11] ENISA. *Guidelines for SMEs on the security of personal data processing*. https://www.enisa.europa.eu/publications/guidelines-for-smes-on-the-security-of-personal-data-processing. 2017.

[12] Fazeldehkordi, E., Owe, O., and Noll, J. *Security and Privacy Functionalities in the Internet of Things (long version)*. Tech. rep. University of Oslo, 2019.

[13] FDA. *Content of Premarket Submissions for Management of Cybersecurity in Medical Devices Guidance for Industry and Food and Drug Administration Staff*. U.S. Food and Drug Administration (FDA). 2014.

[14] Health & Human Services, U. D. of. *Pacemaker*. 2019.

[15] Healthline. *Heart Pacemaker*. 2019.

[16] Islam, S. R. et al. "The Internet of Things for health care: a comprehensive survey". In: *IEEE Access* vol. 3 (2015), pp. 678–708.

[17] Lee, C. et al. "Securing smart home: Technologies, security challenges, and security requirements". In: *2014 IEEE Conference on Communications and Network Security*. IEEE. 2014, pp. 67–72.

[18] MacGillivray, C., Turner, V., and Lund, D. *Worldwide Internet of Things (IoT) 2013–2020 Forecast: Billions of Things, Trillions of Dollars.* International Data Corporation. http://www.idc.com/getdoc.jsp?containerId=243661f. 2014.

[19] OWASP. *IoT Security Guidance*.

[20] OWASP. *OWASP Mobile Security Project*.

[21] Pang, Z. "Technologies and Architectures of the Internet-of-Things (IoT) for Health and Well-being". PhD thesis. KTH Royal Institute of Technology, 2013.

[22] Pishva, D. "Internet of Things: Security and privacy issues and possible solution". In: *Advanced Communication Technology (ICACT), 2017 19th Intern. Conf. on*. IEEE. 2017, pp. 797–808.

[23] Roman, R., Najera, P., and Lopez, J. "Securing the Internet of Things". In: *Computer* vol. 44, no. 9 (2011), pp. 51–58.

[24] Ross, R. et al. “Protecting controlled unclassified information in nonfederal information systems and organizations”. In: *NIST Special Publication* vol. 800 (2015), p. 171.

[25] Schrecker, S. and al., et. *Industrial Internet of Things Volume G4: Security Framework*. Industrial Internet Consortium. https://www.iiconsortium.org/pdf/IIC_PUB_G4_V1.00_PB.pdf. 2016.

[26] Shrestha, M., Johansen, C., and Noll, J. “Security Classification for Smart Grid Infra structures (long version)”. In: (2017).

[27] Sicari, S. et al. “Security, privacy and trust in Internet of Things: The road ahead”. In: *Computer networks* vol. 76 (2015), pp. 146–164.

[28] Suo, H. et al. “Security in the internet of things: a review”. In: *2012 international conference on computer science and electronics engineering*. Vol. 3. IEEE. 2012, pp. 648–651.

[29] Thehackernews.com. *Over 8,600 Vulnerabilities Found in Pacemakers*. https://thehackernews.com/2017/06/pacemaker-vulnerability.html.

[30] Union, E. *REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL*. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

CHAPTER 6

Paper 2: Security and Privacy in IoT Systems: A Case Study of Healthcare Products

Authors: Elahe Fazeldehkordi, Olaf Owe, Josef Noll

Publication: Presented at 13th International Symposium on Medical Information and Communication Technology (ISMICT), 8-10 May, 2019 (pp. 1-8). Published in IEEE. DOI: https://doi.org/10.1109/ISMICT.2019.8743971

Abstract

Internet of Things (IoT) is facilitated by heterogeneous technologies, which contribute to the providing of innovative services in a large number of application domains. The satisfaction of security and privacy requirements in this scenario are becoming a main challenge for IoT systems and their developers. Nevertheless, most works on IoT security and privacy requirements look at these requirements from a high level view. Hence, important aspects of security and privacy functionalities will be disregarded, causing wrong design decisions. To combat this problem, in our previous work, we summarized the most current documents related to security and privacy functionalities in the setting of IoT and provided a new taxonomy framework that organizes all aspects of security and privacy baselines, guidelines and recommendations. To give an understanding of how the framework can help to improve security and privacy of IoT products, and help to facilitate developing and designing secure and privacy-aware IoT systems, in this paper we delve deeper and demonstrate the usefulness of the framework by a case study of healthcare products, in combination with a recent security classification method.

6.1 Introduction

Internet of Things (IoT) involves information flow between different kinds of embedded computing devices interconnected through a network. The aim of IoT is to enable an advanced mode of communication among the various systems and devices, and also to facilitate the interaction between humans and the virtual world. With this aim, IoT plays a significant role in the modern society and has applications in almost all fields of the modern society including healthcare systems, automobile, industrial appliances, sports, homes, entertainments, environmental monitoring etc. IoT devices have already outnumbered the number of people at computerized workplaces, and by 2020, connected "things" based on IoT will be around 212 billion [18, 22]. Those "things" include daily used appliances like smart-phones, smart-watches, smart television, smart refrigerators and others. As a result of this expansion, and as most things are connecting to the internet for exchanging information, IoT is vulnerable to various security issues and attacks, e.g., man in the middle attack, eavesdropping attack, denial of service attack, access attack, as well as major privacy concerns for the end users. Even though IoT provides advanced abilities in the data communication area, it is vulnerable from the security and privacy points of view. Therefore appropriate steps in the initial phase of design and development of IoT systems should be taken.

In our previous work [12], we gave a comprehensive view of the state of the art with respect to security and privacy functionalities and requirements for IoT systems, and suggested a complementary methodology for analyzing the functionalities in a comprehensive framework that can help both providers and consumers of IoT devices to have a better understanding of the security and privacy aspects [12]. By functionality we mean: "The security and privacy-related features, functions, mechanisms, services, procedures, and architectures implemented within organizational information systems or the environments in which those systems operate" [24]. For this framework we investigated IoT-related security baselines and guidelines developed by ENISA [10], OWASP [19], Industrial Internet Consortium [25], Cloud Security Alliance [8], and Broadband Internet Technical Advisory Group [6], as well as IoT related security guidelines from ISO [1, 2] and NIST [24]. With respect to privacy, EU has passed

the general data protection regulation (GDPR), which regulates who can access private data, how and for what purpose, based on consent of the data subject [11]. We then categorized and integrated these guidelines and requirements in a uniform style, and embedded them in a graphical representation by means of diagrams. Having a comprehensive view and taxonomy of security and privacy requirements and functionalities in IoT is a prerequisite for architecting optimal security solutions, designing, and developing secure and privacy-aware IoT systems.

In this paper, we explore how the framework can facilitate the process of improving IoT security and privacy, in combination with the security classification method suggested in [4, 26]. Through the development of this framework, together with the security classes, extensive attention has been given to the requirements and limitations for securing IoT systems. The security classification of a system will lead to better understanding of the value of security and promote the extra cost of securing a system. An IoT system includes different kinds of devices, and protecting all of these devices at a same level is costly. It is economically impossible to employ all the security protection mechanisms for all the devices in a system. Dividing security into different classifications is necessary in order to secure IoT systems to an appropriate level.

To give an understanding of how the framework can help to improve security and privacy in practice, we give a comprehensive discussion on the details of two scenarios in a case study of healthcare products, compare security and privacy challenges of the two scenarios, and then show how to soften these challenges using security classifications and functionalities of our previous framework. One of the most attractive application areas of IoT is health care [16, 21]. Medical applications like remote health monitoring, fitness programs, chronic diseases, elderly care, compliance with treatment and medication at home and by healthcare providers, are some of the important potential applications that IoT can bring. IoT-based healthcare services can help to reduce costs, increase the quality of life, and enrich the user's experience. Therefore, in this paper we demonstrate the framework on a case study concerning healthcare products.

The remainder of this paper is structured as follows: Section 2 provides related work. Section 3 gives a background about the security classification method. Section 4 describes the pacemaker case study, and Section 5 concludes the paper.

6.2 Related Work

In a work presented by Sicari et al. [27], the most relevant available solutions regarding security, privacy, and trust in IoT have been analyzed. Proposals related to security middleware, secure solutions for mobile devices and ongoing international projects on this subject have also been discussed in their work; however, the focus is more general, addressing authentication, confidentiality and access control, while we break down security and privacy requirements in more detail, using the framework of functionalities for all the baselines.

Major challenges and security threats in smart home networks have been analyzed by Lee et al. [17], and fundamental requirements in order to provide secure and confidential operations in smart homes are explained from the results of their analysis. However, these requirements are just listed up and there is no practical solutions or recommendations in this matter. In [28], Suo et al. have deeply analyzed security architectures and features, and divided IoT systems into four key levels of architecture. According to this analysis, the security requirements for each level have been summarized. Furthermore, the research status of key technologies including encryption mechanism, communication security, protection of sensor data, and cryptography algorithms, have been discussed in this paper.

Roman et al. [23] have discussed threats faced by IoT, as well as security and privacy foundations based on objectives in a scenario involving a smart meter. However, they did not give any details about practical baselines and guidelines showing how to achieve these foundations.

Babar et al. [5] have presented a threat taxonomy and high level security requirements for IoT, which like most of the other works just number these requirements without any practical recommendations for each category. And at the end, they introduced a security model based on high-level requirements of security, privacy and trust. Related security requirements of IoT systems are discussed by Alqassem and Svetinovic [3], proposing a taxonomy of quality attributes, and some of the existing security mechanisms and policies in this matter have been reviewed, in order to reduce the identified security attacks and mitigate future vulnerabilities in these systems. They also have applied

this taxonomy in a smart grid AMI as an IoT scenario. In contrast, the framework [12] considers both security and privacy requirements and decomposes the related mechanisms, policies, and requirements with more details.

6.3 Security Classification

Concepts of security classes have been suggested and defined in [4, 26]. Here, we briefly touch on these concepts to give an overview. There are six classes of security, from A to F, with A representing the best security and F representing the least security. Security classes are based on two factors: *exposure* and *impact* of the possible attacks on the system (see Table 6.2). A lower exposure level means a lower attack surface, therefore, attacks that have low exposure are relatively safe. And vice versa, high impact of attacks on a system affects the security class of the system and necessary precaution should be considered to protect the system. Consequently the security class of the system will be raised. A system with low exposure and low impact is relatively safe.

Table 6.1: Exposure (from [26])

Protection	C1	C2	C3	C4	C5
P1	E4	E4	E5	E5	E5
P2	E3	E3	E4	E4	E4
P3	E2	E2	E3	E3	E3
P4	E1	E1	E2	E2	E2
P5	E1	E1	E1	E1	E1

Connectivity

Table 6.2: Security classes (from [26])

Impact	E1	E2	E3	E4	E5
Catastrophic	Class A	Class D	Class E	Class F	Class F
Major	Class A	Class B	Class D	Class E	Class F
Moderate	Class A	Class B	Class C	Class E	Class E
Minor	Class A	Class B	Class B	Class C	Class D
Insignificant	Class A	Class A	Class A	Class B	Class C

Exposure

Impact is a consequence of the possible attacks on a system. When a system is compromised, it can have impact on several sectors beyond the system itself, including business, government, or society. The *impact* is divided into 5 levels: Insignificant, Minor, Moderate, Major, and Catastrophic. Defining each of these levels depends on the system under evaluation, the type of impacts it can have (e.g. financial, social), or the application area. So, it is based on the judgment of the security/risk analysts in that special application or system.

Exposure (see Table 6.1) is a consequence of the connectivity and the protection level of a system. According to the connectivity of the system, an appropriate set of security functionalities is identified to protect the system, and the strength of the identified security functionalities determines the protection levels (for instance for authentication we can use passwords or PINs or two/multi-factor authentication). The definition of the protection levels (P1 to P5) is according to the ISA99 standard. However, the protection level evaluation is depending on the expert and the particular scenario considered.

The connectivity is divided into five levels, C1 to C5, according to ANSSI [4]:

- (C1): a closed and isolated Information & Communication System (ICS)
- (C2): an ICS connected to a corporate Management Information System (MIS) for which operations from outside the network are not allowed
- (C3): an ICS connected to wireless technology.
- (C4): an ICS with *private* infrastructure permitting operations from outside
- (C5): a distributed ICS with *public* infrastructure.

Increasing the protection level or reducing the connectivity level can reduce the exposure (see Table 6.1). As we can observe from Tables 6.1 and 6.2, by keeping the protection level in the highest level (P1 is the lowest and P5 the highest protection level), exposure will be in the lowest level (E1), therefore resulting in the highest security class. And the way we can provide the highest protection level is by applying an effective and appropriate set of security functionality criteria in a particular device, for instance, use of multi-factor authentication instead of just passwords or PINs in authentication. Effective and appropriate sets of security functionality criteria used in the case study have been

taken from our framework in [12], which is based on the most relevant standards: ENISA [10], OWASP [19], Industrial Internet Consortium [25], Cloud Security Alliance[8], and Broadband Internet Technical Advisory Group [6], as well as security and privacy guidelines from ISO [1, 2], ENISA [11], and NIST [24].

6.4 Pacemaker Case Study

In previous work [12], we have built a methodology for looking at the functionalities from both security and privacy points of view. In order to understand how we can use the functionality framework to improve security and privacy of IoT systems in practice, we here use a case study of health products involving a pacemaker and related control units such as a mobile phone and a heart rate sensor. In the domain of health services, the highest security class is recommended for devices like pacemakers that directly control life functions of a patient.

A pacemaker is a medical device that is implanted under the skin to help with abnormal behaviors of heartbeats [7, 15]. It consists of a battery, a computerized generator, and wires with sensors at their tips [14]. The generator is powered by the battery, and both are surrounded by a thin metal box. The generator is connected to the heart by the wires. The pacemaker helps to monitor and control the heartbeat. The sensors detect the heart's electrical activity and send data through the wires to the computer in the generator. If the heart rhythm is abnormal, the generator will be directed by the computer to send electrical pulses to the heart, and the pulses travel through the wires to reach the heart. Embedded microprocessors in modern pacemakers have enabled them to do additional tasks like monitoring heart activity and providing a record for the patients and their healthcare providers, as well as collecting data on heart functions to help doctors to identify and diagnose patient conditions, and send required shock signals when needed. Doctors can monitor the patient's heart activities and control the pacemaker using a mobile phone or a computer device, or send required shock signals in case of observation of abnormal behavior.

For each device we will below discuss the security and privacy challenges for each of the three devices. In each case we discuss connectivity, protection level and relevant functionality criteria. We

use the general criteria given in the functionality framework, select and discuss the parts relevant for each device.

Pacemaker security controller. Beside threatening patients' lives, malicious attackers can get access to patients' medical records through a pacemakers [9, 29], or track a patient's location. In addition, malicious software can be run on pacemakers and cause security and privacy breaks. Therefore, any security or functional weakness can result in a security failure. Like any device that uses remote technology, pacemakers are also vulnerable against cyber attacks, and hackers can break into the pacemaker itself, the back-end systems or the communication between the pacemaker and its surrounding. By breaking into a pacemaker, an attacker can send strong shock signals, disturb the pacemaker setting or heart functions, or disturb them from working properly. One simple example of hacking into a pacemaker is to change the setting from battery-saving "sleep" mode to "standby" mode, and this can quickly drain the battery, which is normally supposed to last for years. Hence, security in this kind of device is crucial and should have the highest (best) security class, meaning security class A.

We consider Connectivity 2 (explained in Section. 6.3) in the pacemaker security controller, since it is only connected to the pacemaker. We define below the protection levels which are relevant <for the pacemaker security controller:

- Protection level 1 (P1): includes Secure authentication, Securing Software/Firmware, Secure Communication, and Human Interface Security. For authentication we consider: requiring passwords, option to change the default username and password. For communication we consider: data authenticity to enable reliable exchanges from data emission to data reception. For securing software/firmware we consider: update capability for *some* of the system devices and applications, transmitting the files using encryption.
- Protection level 2 (P2): includes P1 and in addition, for authentication: requiring strong passwords, securing password recovery mechanisms, making sure that default passwords and even default usernames are changed during the initial setup, and that weak, null or blank passwords are not allowed. For communication: verifying any interconnections, discover, identify and verify/authenticate the devices connected to the network before trust can be established,

and preserve their integrity for reliable solutions and services, prevent unauthorised connections to it or other devices the product is connected to, at all protocol levels, providing communication security using state-of-the-art mechanisms, standardising security protocols, such as TLS for encryption. For securing software/-firmware: encrypting update files for *some* of the applications.

- Protection level 3 (P3): includes P2 and in addition, for authentication: having options to force password expiration after a specific period, and to change the default username and password, making sure that the password recovery or reset mechanism are robust and do not supply an attacker with information indicating a valid account. The same should apply to key update and recovery mechanisms. For communication: data authenticity to enable reliable exchanges from data emission to data reception.
- Protection level 4 (P4): includes P3 and in addition, for authentication: implementing two-Factor Authentication (2FA), making sure that default passwords and even default usernames are changed during the initial setup. For communication: signing the data whenever and wherever it is captured and stored, making intentional connections, disabling specific ports and/or network connections for selective connectivity. For securing software/firmware: capability of quick updates when vulnerabilities are discovered for *some* of the system devices and applications, and offering an automatic firmware update.
- Protection level 5 (P5): includes P4. In addition, for authentication: using Multi-Factor Authentication (MFA) (considering biometrics for authentication), considering Certificate-Less Authenticated Encryption (CLAE) and User Managed Access (UMA). For communication: rate limiting – controlling the traffic sent or received by a network to reduce the risk of automated attacks. For securing software/firmware: update capability for *all* system devices and applications, capability of quick updates when vulnerabilities are discovered for *all* system devices and applications, encrypting update files for *all* applications, signing update files and validating by the device before installing, securing update servers, having ability to implement scheduled updates, having backward compatibility of firmware updates.

According to Tables 6.1 and 6.2, protection level 4 or 5 are needed

to obtain security class A. We therefore consider how to obtain that protection level, and select the following set of relevant functionality criteria from the functionality framework given in [12], adapted to the challenges of pacemakers. In this selection we have used the guidelines for securing pacemakers from [13]. This gives a certain guarantee that we cover all relevant aspects.

- *Secure authentication/authorization*: Access to the pacemaker security controller and mobile phone connected to the doctor, should be limited using authentication of users (for example user ID and password, Personal Identification Numbers (PINs), biometric authentications). Authorization refers to checking necessary permissions of an identified individual to do an action. Authentication and authorization are completely related to each other. Authorization checks should immediately be followed by authentication of a request. In order to have secure authorization, the roles and permissions of the authenticated user should only be verified through information in back-end systems, not through roles or permission information coming from the device. Any incoming identifiers with a request alongside should be verified by the back-end code independently. Failure in a secure authentication/authorization would give access to unauthorized people and could lead to reputational damages, fraud, unauthorized access to information, information theft, and modification of data.
- *Securing software/firmware*: Before any software or firmware update, user authentication or other suitable controls should be required. Software/firmware updates should be restricted to authorised code. Manufacturers may consider code signature verification as an authentication method.
- *Secure communication*: Data transmission between the pacemaker security controller and the mobile phone connected to the doctor must be secure enough so that a third party cannot listen to their communication. The communication should not be vulnerable to eavesdropping or interception. Failure in having a secure communication can cause identity theft, fraud, data modification, or privacy information leakage. One should consider strong handshaking, correct SSL versions, no clear-text communication of sensitive assets, etc.

- *Human interface security*: Patients should be trained about the importance of security and privacy and how to use the pacemaker properly to ensure security is managed at an appropriate level. For instance, if we consider all the security protections in the highest level in the security controller, but the patient does not know how to deal with error notifications (restart, turn off or low battery errors) in the security controller, all the security considerations in the controller will be useless.
- *Data privacy*: Measures to avoid risk of breaches in connection with long term storage of private information, handling of encryption of private information, and GDPR compliance including consent, purpose, and access rights.

The final discussion of the security class of the pacemaker system depends on the scenario chosen and the other components involved. We next consider the mobile phone.

Mobile phone. In our case study, a mobile phone is used in the communication between the pacemaker and the healthcare provider/doctor. Security in this mobile phone is then important in order to send correct data to/from the pacemaker controller. Hacking or tampering into this mobile phone can result in sending wrong data from the pacemaker to the doctor, something that could result in wrong decisions from the doctor, or possibly sending inappropriate shocks to the patient's heart.

Therefore, it is important to secure the mobile phone properly. However, the security class of the mobile phone/computer device is only considered class B, since mobile phones are inherently not of the highest security class and have a number of possibilities for attacks due to high exposure. For instance, because of all the applications and browsers on the device, as well as software developed by third parties. Therefore, both the impact of attack as well as the connectivity are in a higher level for the mobile phone compared to the pacemaker. Hence the security class of the mobile phone is lower than the security class of the pacemaker security controller. We consider C4 in the mobile phone connected to the doctor, and based on that the following set of relevant security functionality criteria are selected to ensure that the mobile phone has a high protection level, following the functionality framework of [12] and the OWASP guidelines for securing mobile phones [20]:

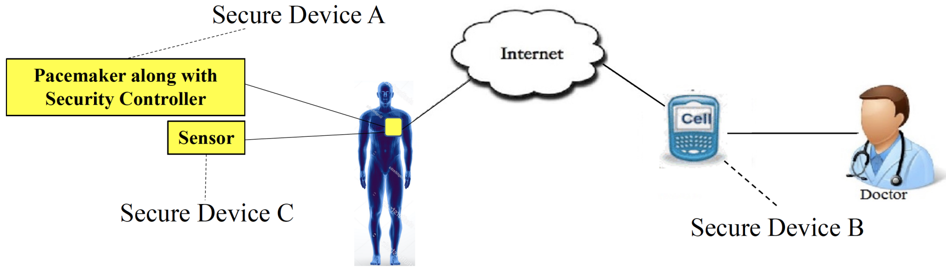

Figure 6.1: Scenario 1: pacemaker along with security controller

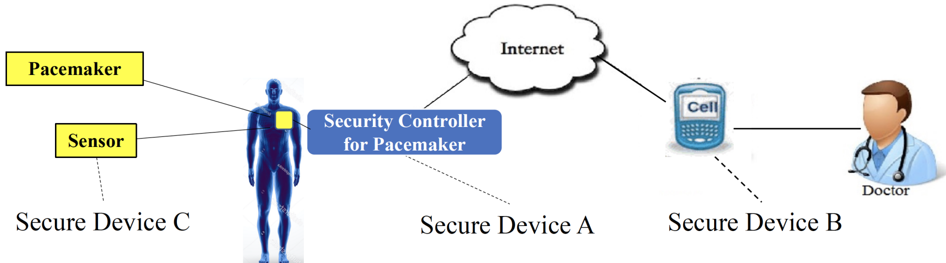

Figure 6.2: Scenario 2: pacemaker with separate security controller

Table 6.3: Security and privacy challenge comparison of scenarios 1 and 2

	Security and Privacy Challenges	Scenario 1	Scenario 2
In Sensor	Draining battery by changing battery saving controls		
	Interrupting into heartbeat capturing		
	Data and private information breaches		
In Pacemaker	Sending/Receiving wrong data to/from mobile phone connected to the doctor		
	Changing pacemaker shock settings		
	Draining battery by changing battery saving controls		
	Higher battery usage		
	Data and private information breaches		
	Transparency of GDPR compliance		
	Risk of long-term storage of private information		
In Mobile Phone Connected to the Doctor	Sending/Receiving wrong data to/from pacemaker		
	Sending wrong unnecessary/necessary shocks		
	Changing pacemaker shock settings		
	Draining pacemaker battery by changing battery saving controls		
	Data and private information breaches		
	Transparency of GDPR compliance		
	Risk of long-term storage of private information		

Severity of Challenge: Insignificant: Minor: Moderate: Major: Catastrophic:

- *Secure authentication/authorization* and *secure communication* are the same as what we discussed for the pacemaker.
- *Sufficient cryptography techniques*: Appropriate cryptography techniques should be considered for instance using AES instead of

DES.

- *Code tampering*: When an application is on the device, the code and data resources are also available there. An attacker can modify the code through either malicious apps in third party app stores or trick the user via phishing attacks and install the app on the device. Code tampering could result in revenue loss due to piracy, reputational damage, unauthorized new features, identity theft or fraud.
- *Secure data storage*: Failure in having secure data storage can result in data loss, extraction of sensitive data using malware, modified apps or forensic tools, identity theft, fraud, reputation damage, material loss or external policy violations.
- *Proper platform usage*: Misuse of a platform feature or failure to use platform security controls fall under this category. Android intents, platform permissions, misuse of TouchID, Keychains, or other security controls which are part of the operating system could be included. To prevent the attacks in this category, secure coding and configuration practices must be applied on the server side of the application.
- *Data privacy*: are as discussed above for the pacemaker.

We next consider the protection levels of ISA99, and select the relevant criteria from the functionality framework, and adjust them for the case of the mobile phone connected to the doctor. These are defined below:

- Protection level 1 (P1): includes Secure Authentication, Secure Authorization, Securing software/firmware, Sufficient Cryptography Techniques, Code Tampering, Secure Data Storage, Secure Communication, and Proper Platform Usage. Secure authentication, Securing software/firmware, and Secure Communication are the same as the considerations for protection level 1 for the pacemaker security controller.
- Protection level 2 (P2): includes P1. Secure authentication, Securing software/firmware, and Secure Communication are the same as the considerations for protection level 2 for the pacemaker security controller. In addition, for cryptography techniques we should consider: ensuring proper selection of standard encryption algorithms and keys.

- Protection level 3 (P3): includes P2. Secure authentication, Securing software/firmware, and Secure Communication are the same as the considerations for protection level 3 for the pacemaker security controller. In addition, for cryptography techniques we should consider: strong encryption algorithms, strong keys.
- Protection level 4 (P4): includes P3. Secure authentication, Securing software/firmware, and Secure Communication are the same as the considerations for protection level 4 for the pacemaker security controller. In addition, for cryptography techniques we should consider: verifying the robustness of the implementation, establishing secure and scalable key management. And cryptographic keys must be securely managed.
- Protection level 5 (P5): includes P4. Secure authentication, Securing software/firmware, and Secure Communication are same as the considerations for protection level 4 for the pacemaker security controller. In addition, for cryptography techniques we should consider: disabling insecure protocols.

Finally, the security class of the heart rate sensor in our case study is considered below.

Heart rate sensor. The heart rate sensor does not have any external connectivity; it has only connections to the heart and computerized generator inside the pacemaker in order to transfer the captured heart rate from the heart to the pacemaker generator. So the connectivity in the sensor is C1. Furthermore it has a chip set from the company provider. Hence, the sensor has the least chance of attack. However, there is still a possible vulnerability if the wired connection is not sufficiently shielded and allows eavesdropping by inductive sensors or senders. But this requires very short physical distance. Therefore we may assume only minor impact, and do not need a high protection level for this device, and may use P1. Exposure is then E4 (see Table 6.4). This gives security class C.

Pacemaker scenarios. The challenge in the case study is how to implement the security controls like decryption, authentication, etc. For instance, whether to do data decryption in the mobile phone, and then send the decrypted data to the pacemaker. If the mobile phone is compromised, wrong data and signals could then go to the doctor and pacemaker, since mobile phones can get compromised, as we discussed before. Whereas, if we consider security controls in the pacemaker itself,

we only have software provided by one company. Thus it would be much more secure to do the security controls directly in the pacemaker, and use the mobile phone just as a gateway to transfer the information. However, this will require additional computational power and battery capacity. Another issue is whether we can do all the security controls inside the computerized generator of the pacemaker – or consider a security controller as a separate unit out of the body with a close and secure connection to the pacemaker.

We therefore define two scenarios: In scenario 1 (see Figure. 6.1), the security controls for the pacemaker are done inside the computerized generator of the pacemaker, and in scenario 2 (see Figure. 6.2), a separate security controller unit makes the security controls of the pacemaker, such that this unit is outside the body with a close and secure connection to the pacemaker.

In scenario 1, we would need computational power inside the body, needing more storage, stronger CPU, much more battery capacity and so on. This is not desirable since it might increase the potential necessary surgeries in order to change the battery, maintain, or update the pacemaker. Moreover, we might not be able to consider some of the security functionalities in order to avoid increasing CPU usage, which results in even more battery usage. Therefore, we might want to have a pacemaker with a simple sensor inside the body and a security controller out of the body, say in the pocket or at home very close to the pacemaker. Here, it is essential that the security controller is close to the pacemaker, since the pacemaker must have very weak signals, limited interactions and computations, to avoid using too much battery power and resulting battery changes. So, all the security and battery-intensive controls would be done in the external security controller, which can have a stronger and easily rechargeable battery. The controller can maintain the device in case of any problem, update or troubleshoot its computer system, or even increase the level of protection by adding more security and privacy functionalities or appropriate software at any time because of easy access. According to all the security considerations in scenario 2, we can then have a better security class in the pacemaker security controller: In Table 6.3 we summarize all the security and privacy challenges in the sensor, pacemaker, and mobile phone connected to the doctor, and compare the severity of these security and privacy challenges in scenarios 1 and 2. In the next section we see that the severity of the security and

privacy challenges in the sensor and pacemaker has been reduced from very high in the worst case to low. Hence, the impact of attacks is reduced from catastrophic to minor.

6.5 Discussion

The pacemaker in scenario 1 is a complex device because all security controls are done inside the computerized generator of the pacemaker. The security controller is very close to the sensor, the signals received by the sensor from the pacemaker are very frequent. Interference is possible and that can have negative effect on correct heartbeat capturing. However, by transferring the security controller outside of the body this effect would be low (still there are some frequencies from devices close by such as mobile phones that can affect the sensor) resulting in more precise heartbeat capturing. As mentioned earlier in the heart rate sensor, there are low possibilities of data and private information breaches in both scenarios.

In the pacemaker in scenario 1, because of the complexity of the system and having all the security controls inside the computerized generator

Table 6.4: Security class of the sensor

Protection	Connectivity: C1
P1	E4
P2	E3
P3	E2
P4	E1
P5	E1

Impact	Exposure: E4
Catastrophic	Class F
Major	Class E
Moderate	Class E
Minor	Class C
Insignificant	Class B

In Scenario 1: P1 / E4, Catastrophic / Class F. In Scenario 2: Minor / Class C.

Table 6.5: Security class of the pacemaker

Protection	Connectivity: C2
P1	E4
P2	E3
P3	E2
P4	E1
P5	E1

Impact	Exposure: E2
Catastrophic	Class D
Major	Class B
Moderate	Class B
Minor	Class B
Insignificant	Class A

In Scenario 1: Catastrophic / Class D. In Scenario 2: Insignificant / Class A.

Table 6.6: Security class of the mobile phone

Protection		
P1	E5	
P2	E4	
P3	E3	
P4	E2	
P5	E1	
	C4	
	Connectivity	

Impact		
Catastrophic	Class D	
Major	Class B	
Moderate	Class B	
Minor	Class B	In Scenarios 1 & 2
Insignificant	Class A	
	E2	
	Exposure	

of the pacemaker, we need higher CPU and memory consumption and then battery usage would be very high, while in scenario 2 this problem would decrease to very low. In scenario 1, because of difficulties in accessing the pacemaker and its security controller on time (in case of maintenance, update, troubleshooting or installing new software/equipment, also not being able to apply all the necessary criteria with high protection level in order to avoid high battery usage), the level of the security and privacy functionality criteria as well as the level of the protection is low. Therefore the vulnerability of the pacemaker and sensor against attacks compromising the device (that can cause tampering of transferred information from/to the pacemaker, changing pacemaker shock settings, changing battery saving controls, data and private information breaches, transparency of GDPR compliance, risk of long-term storage of private information) is high; however, this problems decreases to very low when we change to scenario 2, due to easier access to the security controller out of the body.

By compromising the mobile phone connected to the doctor, the problems listed in the last part of Table 6.3 may occur. In both scenarios, we have considered high level of protection for the mobile phone by using appropriate set of security and privacy functionality criteria with high protection level, therefore we have reduced the vulnerability of the device to low, but as discussed earlier, these devices still have a number of vulnerabilities for attacks, and therefore the vulnerability is not in a very low level.

Furthermore, softening the problems in scenario 1 results in obtaining higher protection level and security class. For instance, we can easily recharge the battery of the security controller, maintain the device in case of any problem to avoid shutting down all the security considerations, and increase the level of protection by updating or troubleshooting its

computer system.

In the sensor, as explained above, we have connectivity C1, and protection level P1, therefore exposure is E4 (see Table 6.4), and because of very high problem severity, the impact of attacks is Catastrophic, resulting in security Class F in this device.By changing from scenario 1 to 2, the severity of the security and privacy challenges has reduced from very high to low, therefore the impact of attacks is reduced from Catastrophic to Minor, and consequently, the security class has improved from Class F to Class C.

In the pacemaker, we have connectivity C2, and protection level P5, however in scenario 1, the severity of security and privacy challenges have affect on the protection level, which falls to level 3, therefore exposure is E2 (see Table 6.5), and because of very high security and privacy challenge severity, the impact of attacks is Catastrophic, then we have security Class D in this device. By changing from scenario 1 to 2, the severity of the security and privacy challenges has reduced from very high to very low, therefore the impact of attacks has reduced from Catastrophic to Insignificant, and consequently, the security class has improved from Class D to A.

And, in the mobile phone connected to the doctor, we have connectivity C4, and protection level P4, therefore the exposure is E2 (see Table 6.6), and the security is Class B in both scenarios.

By considering security and privacy challenges, as well as the effect of these challenges on the protection level of each device and the whole system, and also their affect on the impact of attacks, we have changed scenario 1 to 2, and were able to improve the security class in the sensor and pacemaker security controller from class F to C, and class D to A, respectively. Hence the security of the overall system has improved significantly.

6.6 Conclusion

The expansion of IoT in the last decade has resulted in several security and privacy vulnerabilities and attacks against IoT devices and people. Unfortunately, the security and privacy functionalities to combat these attacks are not well-recognized in the domain of IoT. We previously presented a new taxonomy framework that addresses all of the IoT

security and privacy functionalities. In this paper, we demonstrate the application of this framework in combination with the security classification method, using a case study of pacemaker as medical device communicating with the surrounding, and discuss the various security and privacy challenges associated with two scenarios.

Our analysis results demonstrate that using our security functionality framework, the security class in the sensor and pacemaker has been improved from class F to C, and class D to A, respectively. Therefore, the security of the overall system has improved. The main objective of this paper is to give security developers, designers, and end-users an opportunity to explore what the IoT security and privacy functionalities are, and how these functionalities can help to improve security and privacy of IoT systems. In future work, we will investigate the applications of our framework to other IoT domains.

Acknowledgment

This work has been supported by the research project *IoTSec*, Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Research Council of Norway. It is also supported by the *SCOTT* project (www.scott-project.eu), funded by the Electronic Component Systems of the European Leadership Joint Undertaking under grant agreement No 737422, with support from the EU H2020 research and innovation programme.

Bibliography

[1] 27000, I. *Information technology — Security techniques — Information security management systems — Overview and vocabulary (fourth edition)*. ISO/IEC 2016. 2016.

[2] 27001, I. *INTERNATIONAL STANDARD ISO/IEC 27001 Information technology—Security techniques —Information security management systems —Requirements*. ISO/IEC 2013. 2013.

[3] Alqassem, I. and Svetinovic, D. “A taxonomy of security and privacy requirements for the Internet of Things (IoT)”. In: *2014 IEEE Intern. Conf. on Industrial Engineering and Engineering Management (IEEM)*. IEEE. 2014, pp. 1244–1248.

[4] ANSSI. *Classification Method and Key Measures*. 2014.

[5] Babar, S. et al. “Proposed security model and threat taxonomy for the Internet of Things (IoT)”. In: *Intern. Conf. on Network Security and Applications*. Springer. 2010, pp. 420–429.

[6] BITAG. *Internet of Things (IoT) Security and Privacy Recommendations.* https://www.bitag.org/documents/BITAG_Report_-_Internet_of_Things_(IoT)_Security_and_Privacy_Recommendations.pdf. 2016.

[7] Boulos, P. et al. “Pacemakers: a survey on development history, cyber-security threats and countermeasures”. In: *Int. J. Innov. Stud. Sci. Eng. Technol* vol. 2, no. 8 (2016).

[8] *Future-proofing the Connected World: 13 Steps to Developing Secure IoT Products*. https://downloads.cloudsecurityalliance.org/assets/research/internet-of-things/future-proofing-the-connected-world.pdf. 2016.

[9] Dadourian, M. *Heart alert: Pacemakers can be hacked*. 2018.

[10] ENISA. *Baseline Security Recommendations for IoT in the context of Critical Information Infrastructures*. European Union Agency for Network and Inf. Sec. 2017.

[11] ENISA. *Guidelines for SMEs on the security of personal data processing*. https://www.enisa.europa.eu/publications/guidelines-for-smes-on-the-security-of-personal-data-processing. 2017.

[12] Fazeldehkordi, E., Owe, O., and Noll, J. *Security and Privacy Functionalities in the Internet of Things (long version)*. Tech. rep. University of Oslo, 2019.

[13] FDA. *Content of Premarket Submissions for Management of Cybersecurity in Medical Devices Guidance for Industry and Food and Drug Administration Staff*. U.S. Food and Drug Administration (FDA). 2014.

[14] Health & Human Services, U. D. of. *Pacemaker*. 2019.

[15] Healthline. *Heart Pacemaker*. 2019.

[16] Islam, S. R. et al. "The Internet of Things for health care: a comprehensive survey". In: *IEEE Access* vol. 3 (2015), pp. 678–708.

[17] Lee, C. et al. "Securing smart home: Technologies, security challenges, and security requirements". In: *Communications and Network Security (CNS), 2014 IEEE Conf. on*. IEEE. 2014, pp. 67–72.

[18] MacGillivray, C., Turner, V., and Lund, D. *Worldwide Internet of Things (IoT) 2013–2020 Forecast: Billions of Things, Trillions of Dollars*. International Data Corporation. http://www.idc.com/getdoc.jsp?containerId=243661f. 2014.

[19] OWASP. *IoT Security Guidance*.

[20] OWASP. *OWASP Mobile Security Project*.

[21] Pang, Z. "Technologies and Architectures of the Internet-of-Things (IoT) for Health and Well-being". PhD thesis. KTH Royal Institute of Technology, 2013.

[22] Pishva, D. "Internet of Things: Security and privacy issues and possible solution". In: *Advanced Communication Technology (ICACT), 2017 19th Intern. Conf. on*. IEEE. 2017, pp. 797–808.

[23] Roman, R., Najera, P., and Lopez, J. "Securing the Internet of Things". In: *Computer* vol. 44, no. 9 (2011), pp. 51–58.

[24] Ross, R. et al. "Protecting controlled unclassified information in nonfederal information systems and organizations". In: *NIST Special Publication* vol. 800 (2015), p. 171.

[25] Schrecker, S. and al., et. *Industrial Internet of Things Volume G4: Security Framework*. Industrial Internet Consortium. https://www.iiconsortium.org/pdf/IIC_PUB_G4_V1.00_PB.pdf. 2016.

[26] Shrestha, M., Johansen, C., and Noll, J. "Security Classification for Smart Grid Infra structures (long version)". In: (2017).

[27] Sicari, S. et al. "Security, privacy and trust in Internet of Things: The road ahead". In: *Computer networks* vol. 76 (2015), pp. 146–164.

[28] Suo, H. et al. "Security in the Internet of Things: a review". In: *Computer Science and Electronics Engineering (ICCSEE), 2012 Intern. Conf. on*. Vol. 3. IEEE. 2012, pp. 648–651.

[29] Thehackernews.com. *Over 8,600 Vulnerabilities Found in Pacemakers*. https://thehackernews.com/2017/06/pacemaker-vulnerability.html.

CHAPTER 7

Paper 3: A Language-Based Approach to Prevent DDoS Attacks in Distributed Financial Agent Systems

Authors: Elahe Fazeldehkordi, Olaf Owe, Toktam Ramezanifarkhani

Publication: Presented at Security for Financial Critical Infrastructures and Services (FINSEC), part of the 24th European Symposium on Research in Computer Security (ESORICS) Conference, 23-27 September 2019. Published in Lecture Notes in Computer Science, vol 11981 (pp. 258-277). Springer. DOI: https://doi.org/10.1007/978-3-030-42051-2_18

Abstract

Denial of Service (DoS) and Distributed DoS (DDoS) attacks, with even higher severity, are among the major security threats for distributed systems, and in particular in the financial sector where trust is essential.

In this paper, our aim is to develop an additional layer of defense in distributed agent systems to combat such threats. We consider a high-level object-oriented modeling framework for distributed systems, based on the actor model with support of asynchronous and synchronous method interaction and futures, which are sophisticated and popular communication mechanisms applied in many systems today. Our approach uses static detection to identify and prevent potential vulnerabilities caused by asynchronous communication including call-based DoS or DDoS attacks, possibly involving a large number of distributed actors.

7.1 Introduction

Today distributed and service-oriented systems form critical parts of

infrastructures of the modern society, including financial services. In the financial sector security and trust are essential for users of financial services [19]. Security breaches may lead to significant loss of assets, such as physical or virtual money, including cryptocurrencies and bitcoins. In addition, successful attacks on services of a financial institution may damage the trust of customers, which indirectly may hurt the institution [18]. According to [1, 12, 17], a main threat on financial institutions is Distributed Denial of Service (DDoS) attacks. Protection against DoS/DDoS attacks is therefore crucial for financial institutions. Slow website responses caused by targeted attacks, can imply that customers cannot access their online banking and trading websites during such attacks. Both network layer and application layer DDoS attacks continue to be more and more persistent according to a report from the Global DDoS Threat Landscape Q4 2017. Based on this report, it becomes easier and easier to launch DDoS attacks, and one may even purchase botnet-for-hire services that provide the basis for starting a hazardous DDoS attack. Financial institutions are recommended to monitor the internet traffic to their websites in order to detect and react to possible threats. However, this kind of run-time protection may slow down or temporarily shut down the websites.

Unintended attacks on customers from a financial institution may easily destroy the customer's trust and confidence and result in reputation damage. If customers cannot trust an institution, they may quickly shift to a different institution, due to competition between the many different financial institutions and service providers. Even a single unfortunate incident of a financial service provider could be enough to influence customers. One should make sure that the software is not harmful for the customers before running it, and this makes static (compile-time) detection more important than in other areas. Thus, in the financial sector static detection is a valuable complement to run-time detection methods, and seems underrepresented.

Call-based *flooding* is commonly seen in the form of application-based DDoS attacks [6]. To prevent DoS/DDoS flooding attacks in a manner complementary to existing approaches, we propose an additional layer of defense, based on language-based security analysis. We focus on DDoS attacks that try to force a (sub)system out of order by flooding applications running on the target system, or by using such applications to drain the resources of their victim.

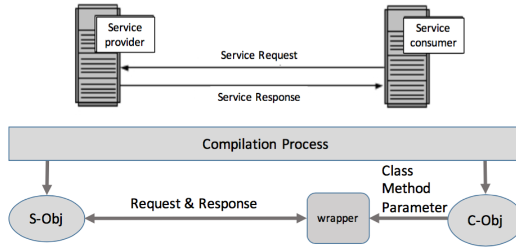

Figure 7.1: Distributed communication (s-obj stands for server object and c-obj for consumer object)

In this paper, we consider a high-level imperative and object-oriented language for distributed systems, based on the actor model with support of asynchronous and synchronous method interaction. This setting is appealing in that it naturally supports the distribution of autonomous concurrent units, and efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling and lock operations. It is therefore useful as a framework for modeling and analysis of distributed service-oriented systems. Our language supports efficient interaction by features such as asynchronous and non-blocking method calls and first-class futures, which are popular features applied in many distributed systems today. However, these mechanisms make it even easier for an attacker to launch a DDoS attack, because undesirable waiting by the attacker can be avoided with these mechanisms.

We propose an approach consisting of static analysis. We identify and prevent potential vulnerabilities in asynchronous communication that directly or indirectly can cause call-based flooding of agents. More precisely, we adapt a general algorithm for detecting call flooding [14] to the setting of security analysis and for detection of distributed denial of service attacks adding support for many-to-one attacks. The algorithm detects call cycles that might overflow the incoming queues of one or more communicating agents. Each cycle may involve any number of agents, possible involving the attacked agent(s).

The high-level framework considered here is relevant for a large class of programming languages and service-oriented systems.

Outline. Section. 7.2 describes the background of the problem. Related work is discussed in Section. 7.3. The active object framework is

explained in Section. 7.4. Our static analysis to prevent attacks is described in Section. 7.5. Examples of possible DoS/DDoS attacks are given in Section. 7.6. The final section concludes and suggests future work.

7.2 Overview

In distributed system communication there is an underlying distributed object system as shown in Figure. 7.1. In such a distributed system, classes such as server or client classes would be instantiated by objects, and communication is established in the form of method calls, usually wrapped in XML or other forms. Therefore, communication in a distributed system is implemented by method calls between objects. If there is a possibility of flood of requests to the service provider (S-Obj) from the consumer object(s) (C-Obj) in this figure, a DoS attack is probable.

Call-based flooding attacks. To launch a DoS attack, the attacker may try to submerge the target server under many requests to saturate its computing resources. To do so, flooding attacks [6, 20] by method calls are effective, especially when the server allocates a lot of resources in response to a single request. Therefore, we detect:

- call-flooding: flooding from one object to another.
- parametric-call-flooding: flooding from one object to another when the target object allocates resources or consumes resources for each call.

In the case of call-flooding, communications are just simple requests like a simple call without parameters or parameters that do not lead to resource consumption. Parametric-call-flooding is when requests usually include parameters in a non-trivial manner. Such requests usually trigger relatively complex processing on the server such as access to a database. Parametric-call-flooding is more effective than call-flooding because it takes fewer requests to drown the target system. However, call-flooding are more common and easier for attackers to exploit.

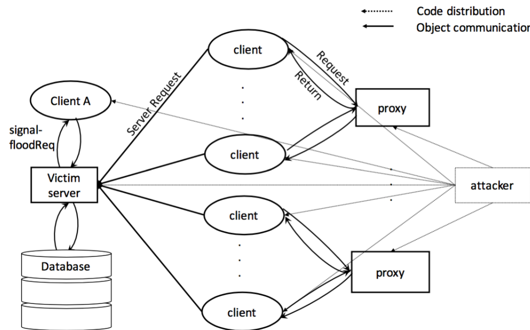

Figure 7.2: Distributed object communication in DDoS

Categories of call-based flooding attacks: DoS or DDoS possibilities.

One-to-one (OTO): If thousands of requests every single second come from one source object to a target object, then it is a *one-to-one (OTO) DoS attack*. The intent of the flooding might be malicious, or even undeliberate call cycles. Communication between Client A and the victim server in Figure. 7.2 is an example of this attack.

Many-to-one (MTO): If the incoming flooding traffic originates from many distinguishable different sources, then it is a *many-to-one (MTO) DDoS attack*. Figure. 7.2 shows a distribution of code between clients and a server, and proxies.

One-to-many (OTM): A *one-to-many attack* appears if a system makes an unlimited number of requests to many objects simultaneously. Such an attack can be serious since many target objects are attacked at the same time.

Static attack detection and prevention. For any set of methods that call the same target method, a call cycle could be harmful. The methods might belong to the same or different objects with the same or different interfaces. In the case of normal blocking calls, where the caller is blocking while waiting for the response, making a flood of requests also means receiving a flood of responses. And thus in the case of OTO, it may cause a self DoS for the attacker. With the possibility of non-blocking calls in a distributed setting, it is more cost-beneficial for an OTO attacker to launch a DoS, because then undesirable blocking by the attacker is

avoided. By means of futures and asynchronous calls, a caller process can make non-blocking method calls.

The possibility of unbounded object creation, referred to as *instantiation flooding* [8], could cause resource consumption and DoS that could be detected statically, especially if those objects and their communication can cause flooding requests from the bots, such as the customers in our example. It is even worse if there is instantiation flooding on the target side of the distributed code. This can be detected by static analysis of the target. (See the example in Figure. 7.10.) Our static analysis detects explicit or implicit call-flooding. Static detection is accomplished by static analysis at compile time and informs the programmer about the possibility of program exploitation at runtime.

7.3 Related Work

A DoS attack, or its distributed version, happens when access to a computer or network resource is intentionally blocked. Considering the exploited vulnerabilities, these attacks might be classified by resource consumption attacks or flooding attacks, of which the latter category is the most common [6]. In this paper, we aim to prevent distributed code to be exploited by attackers to launch a DoS attack by detection of possible call-based flooding in both of the target and zombie sides. To do so, we analyze the distributed code to make an additional layer of defense against DoS or DDoS attacks.

In the following, we discuss related works for preventing application-based DDOS attacks using static detection. In the paper presented by Chang et al. [3], a novel static analysis approach was introduced in order to detect semantic vulnerabilities in networked software that might cause denial of service attacks because of resource exhaustion. Their approach is implemented in a tool named SAFER: Static Analysis Framework for Exhaustion of Resources. SAFER integrates taint analysis (in order to compute the group of program values that are data-dependent on network inputs) and control dependency analysis (for computing the group of program statements whose execution can affect the execution of a given statement) toward detecting high complexity control structures that can be caused by untrusted network inputs. The tool applies the CIL static analysis framework and combines different heuristics for recognizing

loops and recursive calls. Compared to our work the SAFER approach is oriented toward detecting server attacks from within the server code, whereas our approach is mainly targeting server attacks from an external attacker, or a combination of external agents. An attacker needs to understand the code of the server in order to find weaknesses that can be triggered by specific inputs. In contrast, our approach is detecting attacks caused by coordination of several agents and/or servers in a distributed setting.

Another work that detects resource attacks from within the server code is presented by Qie et al. [15]. In their toolkit, they check for possible "rule" violations at runtime. This work is complementary to ours, since our work is oriented toward static detection. Gulavani and Gulwani [7] describe a precise numerical abstract domain. This domain can be used to prove the termination of a large class of programs and also to estimate valuable information such as timing bounds. In order to make linear numerical abstract domains more precise, they make use of two domain lifting operations: One operation depends on the principle of *expression abstraction*. This describes a set of expressions and determines their semantics by use of a selection of directed inference rules. It works by picking up an abstract domain and a group of expressions, such that their semantics are described by a group of rewrite rules, in order to construct a more precise abstract domain. The second domain constructor operation picks up a linear arithmetic abstract domain and constructs a new arithmetic domain that is able to represent linear relations through introduction of *max* expressions. Another approach to estimate worst-case complexity is presented by Colon and Sipma [4]. These approaches [4, 7], in which the complexity of loops and recursive calls has been estimated using structural analysis, are widely complementary to our work.

Zheng and Myers [21] propose a framework for using static information flow analysis in order to specify and enforce end-to-end availability policies in programs. They extend the decentralized label model to include security policies for availability. This work presents a simple language with fine-grained information security policies described by type annotations. In addition, this language has a security type system to reason about end-to-end availability policies. Various examples have been discussed, in which abuse of an availability policy can represent denial of service attacks.

In a work by Meadows [13], a formal analysis has been developed in

order to apply the maximum benefit of tools and approaches that have already been used to strengthen protocols against denial of service attack. This analysis has been done at the protocol specification level. Also, different ways in which existing cryptographic protocol analysis tools can be modified for the purpose of operating in this formal framework, have been demonstrated. In contrast, we do a detailed static analysis of source code both inside and outside a server. The class of software vulnerabilities that we can detect is more complicated than what appears just at the network-protocol specifications level. Moreover, vulnerable sections of the source code have been identified in our work.

The current work shows how the static analysis method for detection of flooding can be used for detection of DoS and DDoS attacks. This general idea was also outlined by the same authors in an extended workshop abstract [16]. Moreover we here discuss why this is particularly harmful in the financial sector, where both economic assets and customer trust are at risk. Furthermore we simplify and adapt the static analysis method of [14] to the setting of financial service systems, extending it to detect many-to-one attacks involving unbounded creation of objects (as demonstrated in example 7.10), as well as hidden attacks, neither of which were detected by the original method of [14].

7.4 Our Framework for Active Object Systems

The setting of concurrent objects communicating by asynchronous method calls combines the Actor model and object-orientation, and is referred to as *active objects*. Active object languages are suitable for modeling and implementing distributed applications, letting a distributed system be modeled by a number of active objects that interact via asynchronous method calls. The active object model provides natural description of autonomous agents in a distributed system, and the *future mechanism* provides an efficient communication primitives [2, 11], allowing results computed in a distributed setting to be referred to and shared. Moreover, the addition of *cooperative scheduling*, as suggested in the Creol language [9], allows further communication efficiency, by adding process scheduling control in the programming

language, and passive waiting. This is achieved by including statements for suspension control, and letting each object have a process queue for holding suspended processes. We consider a core language for active objects with future-based communication primitives, inspired by Creol and ABS [10]. The objects are concurrent units distributed over a network, and their identity is globally unique. An object has a process queue, as well as a queue for incoming method call requests, and can perform at most one process (i.e., remaining part of method call) at a time. A process can be suspended by an **await** statement, allowing other (enabled) processes to continue. When a process is ended or suspended, the object may continue with an incoming call request or other enabled process from the process queue (if any). The **await** statement allows a process to wait for a Boolean condition to be satisfied, or for a future value to be available. The statement is enabled when the waiting condition/future is satisfied/available. The **await** statement enables high-level process control, instead of low-level process synchronization statements such as signaling and lock operations.

Our core language is a typed, imperative language. An assignment has the form $x := e$ where the expression e is without side-effects. All object variables (i.e., object references) are typed by an interface, and an interface specifies the set of methods that are visible through that interface. The interfaces of a class protect and limit the object communication, and in particular shared variable interaction is forbidden. Local data structure is made by data type declarations, indicated by **data**, and a functional data type sublanguage is used to create and manipulate data values. Data values are passed by value, while object variables are passed by reference. The language supports first-class futures. The basic interaction mechanisms (by method calls/futures) are as follows:

- $f := o!m(\overline{e})$ – the current object calls method m on object o with actual parameters $\overline{e}$. A globally unique identity u identifying the call is assigned to the future variable f. A message is then sent over the network from the current object to object o. When object o eventually performs the method and the method gives a result defined by a **return** statement, that result is placed in a (globally accessible) future with identity u, and the future u is then said to be *resolved*. Any process of any object that knows u may access the future value or wait for it to be available.

- $x :=$ **get** f – this statement blocks until the value of the future f is available, and then that value is assigned to the variable x. (Here f may be an expression resulting in a future identity.)

- **await** c – this statement suspends if the Boolean condition c is not satisfied, and is enabled when c is satisfied,

- **await** $x :=$ **get** f – this statement suspends if the value of f is not yet available, and is enabled when the future is available. Then the future value is assigned to x.

The statement sequence $f{:=}\, o!m(\overline{e}); x{:=}$**get** f corresponds to a traditional blocking call, and is abbreviated $x := o.m(\overline{e})$ using the conventional dot-notation. The statement sequence $f := o!m(\overline{e});$ **await** $f; x :=$ **get** f is abbreviated **await** $x := o.m(\overline{e})$ and corresponds to a non-blocking call, since the **await**-statement ensures that the future is available before the **get**-statement is performed. If the result value is not needed, we may simplify the syntax to $o.m(\overline{e})$ for blocking calls and **await** $o.m(\overline{e})$ for non-blocking calls. And if the future is not needed, $f := o!m(\overline{e})$ may be abbreviated to $o!m(\overline{e})$, in which case the future cannot be accessed (since it is not stored in a future variable).

Object creation has the syntax $x :=$ **new** $C(\overline{e})$ **at** o, where the class parameters behave like fields (initialized to the values of $\overline{e}$) except that they are read-only, and the new object is created locally at the site of object o. With the syntax $x :=$ **new** $C(\overline{e})$ the new object is located anywhere in the distributed system.

One may refer to the current object by this and to the caller object by (the implicit method parameter) caller. Self calls are possible by making calls to this, and recursion is allowed. *Active behavior* is possible by making a recursive self call in the constructor method (given as a nameless method). By means of suspension, the active self behavior may be interleaved with execution of incoming calls from other objects, thereby combining active behavior and passive behavior. If- and while-statements are as usual.

We assume all class parameters and method parameters (including this and caller) are read-only. This helps the static analysis by reducing the set of false positives. For methods that return no information we use a predefined type $Void$ with only one value, $void$. For simplicity we omit **return** $void$ at the end of $Void$ methods in the examples.

7.5 Static Analysis to Prevent Attacks

We base our approach on the static analysis of flooding presented in [14] for detection of flooding of requests, formalized for the Creol/ABS setting with futures. We adapt this notion of flooding to deal with detection of DDoS attacks, which have a similar nature. The static analysis will search for flooding cycles in the code, possibly involving several classes. According to [14] (unbounded) flooding is defined as follows:

Definition 7.5.1 (Flooding)**.** An execution is *flooding with respect to a method* m if there is an execution cycle C containing a call statement to a method m at a given program location, such that this statement may produce an unbounded number of uncompleted calls to method m, in which case we say that the call is *flooding with respect to* C in the given execution.

Like in [14], we distinguish between weak flooding and strong flooding. Strong flooding is flooding under the assumption of so-called *favorable* process scheduling, i.e., enabled processes are executed in a fair manner.

Definition 7.5.2 (Strong and weak flooding)**.** A call is *weakly flooding* with respect to a cycle C if there is an execution where the call is flooding with respect to C. And a call is *strongly flooding* with respect to a cycle C if there is an execution with fair scheduling of enabled processes where the call is flooding with respect to C.

Strong flooding reflects the more serious flooding situations that persist regardless of the underlying scheduling policy. In the detection of strong flooding, a *statically enabled* node is considered strongly reachable if each of its predecessor flow nodes are strongly reachable. All statements are statically enabled, apart from **get/await** statements. A **get** statement or an **await** on a future/call is statically enabled if the corresponding future/result is available, detected statically if the corresponding return statement is strongly reachable or another **get/await** statement on the same future is strongly reachable. We rely on a static under-detection of the correspondence between return statements and futures. In the examples this detection is straight forward. With respect to DDoS, weak flooding of a server is in general harmless unless the flooding is caused by a large enough number of objects. Strong flooding

1. Make separate *control flow graphs* (CFGs) for each method. Include a node for each *call*, *get*, *await*, *new* (for object creation), *if* and *while* statement, as well as an initial *starting* and a final *return* node.
2. Add call edges from call nodes to the start node of a *copy* of the called method. In case the call is recursive, simply add a call edge to the existing start node.
3. Identify any cycles in the resulting graph (including all copies of the CFGs).
4. Assign a unique label to each call node, and assign this label to the *start* and *return* node of the corresponding copy of the method CFG.
5. Make *put* edges from the *return* nodes to the corresponding *get*/*await* nodes. This requires static flow analysis, possibly with over-approximation of *put* edges.

Figure 7.3: Control flow graph

is dangerous even from a single attacker.

Following [14], flooding is detected by building the control flow graph (CFG) of the program, locating control flow cycles as outlined in Figure. 7.3, and then analyzing the sets of weakly reachable calls, denoted *calls*, and the set of strongly reachable call completions, denoted *comps*, in each cycle. Flooding is reported for each cycle with a nonempty difference between *calls* and *comps*, as explained in Figure. 7.4. Note that the abbreviated notations for synchronous calls and suspending calls are expanded to the more basic call primitives, as explained above. We assume that assignments (other than calls) will terminate efficiently and therefore ignore them in the CFG. For each method the CFG begins with a start node and ends with a return node (even for void methods) – the latter helps in the analysis of method completion. We next define *weakly and strongly reachable nodes*. The detection of strongly reachable nodes uses a combination of forward and backward analysis, and is simplified compared to [14]:

Definition 7.5.3 (Weakly and strongly reachable nodes)**.** Consider a

Consider a cycle C in the control flow graph G resulting from Figure. 7.3:

1. Mark all nodes in C as*strongly-reachable (SR)*, and the rest as (initially) *not reachable*.
2. From the entry point to the cycle, follow all flow and call edges in a depth-first traversal of G and mark the nodes as *weakly-reachable (WR)*, *strongly-reachable (SR)*, or neither, as defined in Def. 7.5.3.
3. If the previous step results in any changes to the *SR* or *WR* node sets, go to step 2.
4. Report flooding of call n if $n \in (calls - comps)$ where $calls = \{n \mid call_n \in WR\}$ and $comps = \{n \mid return_n \in SR \vee get_n \in SR\}$.

Figure 7.4: Algorithm for detecting flooding by means of $calls$ and $comps$ sets in a given cycle

given cycle C.
Weakly reachable (WR) nodes are those that are on the cycle or reachable from the cycle by following a flow edge or a call edge.

A node is *strongly reachable (SR)* if it is in the given cycle or is reachable from an SR node without entering an if/while node nor passing a wait node (**get/await**) outside the cycle, unless the return node of the corresponding call is strongly reachable. A return node is SR if there is a SR **get/await** node on the same future. And a node is SR if all its predecessor nodes are SR.

We consider two versions of SR, the *optimistic*, where we follow call edges (as indicated above), and the *pessimistic*, where we follow a call edge n only when the call is known to complete, i.e., when $n \in SR$ before following the call edge. (As above we follow flow and put edges without restrictions.)

The optimistic version is used to find unbounded flooding under the assumption of favorable scheduling, i.e., *strong flooding*. The pessimistic version is used to detect unbounded flooding without this assumption, i.e., *weak flooding*. Detection of strong flooding implies detection of weak flooding, but with less precise details about which calls that possibly may cause flooding. If there is a call causing weak flooding wrt. a given cycle,

pessimistic detection will report this call or a call leading to this call. If there is a call causing strong flooding for a given cycle C, optimistic detection will report this call. Our notions of optimistic and pessimistic reachability cover a wider class of nodes than in [14]. The soundness of [14] can be generalized to our setting.

7.6 Examples of Possible DoS/DDoS Attacks

An example of flooding cycles. We consider here an example of a possible DoS attack on customers caused by a financial institution. The attack may be unintended by the institution, but may result from an update supposed to give better efficiency, by use of the future mechanism to reduce the amount of data communicated over the network.

We imagine that the financial institution has a subscription *service* for customers, such that customers can register and receive the latest information about shares and funds, through data of type “newsletter”, here simply defined as a product type consisting of a *content* and a *date*. The financial institution uses a method `signal` to notify the customers about new information about shares. In the first version, each call to `signal` has the newsletter as a parameter. This may result in heavy network traffic and many of the newsletters may not be read by the customers. In the “improved” solution, each call to `signal` contains a reference (by means of a future) to the newsletter rather than the newsletter itself. However, this allows the subscription service system to send `signal` calls even before the newsletter is available, and as we will see, this can cause a DoS attack on the subscribing customers.

In order to handle many customers, a (dynamic) number of proxies are used by the service object, and an underlying newsletter producer is used for the sake of getting newsletters, using suspension when waiting for news. The proxies are organized in a list (*myCustomers*), growing upon need. In both solutions, futures are used by the service object to avoid delays while waiting for a newsletter to be available. In this way the service object can continuously respond to customers. The interfaces are shown in Figure. 7.5. We abbreviate “Newsletter” to “News”. Figure. 7.6 represents a high-level implementation of the publish/subscribe model, adapted from [5]. A multi-cast to each object in the *myCustomers* list is made by the statement *myCustomers!signal(ns)* in line 13. If we shift

```
data News == (String content, Int date) // a product data type
interface ServiceI{
    Void subscribe(CustomerI cl) // called by Clients
    Void produce() // called by Proxies
}
interface ProxyI{
    ProxyI add(CustomerI cl) // called by Service
    Void publish(Fut[News]fut) // called by Service
}
interface ProducerI{
    News detectNews() // called by Service
}
interface NewsProducerI{
    Void add(News ns) // called when news arrives
    News getNews() // called by Producers
}
interface CustomerI{
    Void signal(News ns) // called by Proxies
}
```

Figure 7.5: The interfaces of the units in the subscription example

requiring the actual newsletter to have arrived, from the Proxy (as shown by the statements *ns:=get fut; myCustomers!signal(ns)* in the original *publish* method) to the Customer (i.e., *news:=**get** fut* in the modified *Customer.signal* method). This change in the program causes flooding of customers.

`Service.produce` asynchronously calls `Producer.detectNews` (Pd), line 5 of Figure. 7.6

`Service.produce` asynchronously calls `Proxy.publish` (Xb), line 5 of Figure. 7.6

`Proxy.publish` asynchronously calls `Customer.signal` (Cs), line 6 of Figure. 7.7

`Proxy.publish` asynchronously calls `Service.produce` (Sp) line 6 of Figure. 7.7

Each iteration of this cycle generates an asynchronous call to *Proxy.publish*, which again produces an asynchronous call to *Producer.detectNews*, which is not processed as part of this cycle, nor is its processing synchronized call with the cycle. An unbounded number of suspended calls to *Producer.detectNews* can be produced by this cycle.

```
class Service(Int limit, NewsProducerI np) implements ServiceI {
   ProducerI prod; ProxyI proxy; ProxyI lastProxy; // declaration of fields
   { // constructor
     prod := new Producer(np); proxy:= new Proxy(limit,this);
     lastProxy:=proxy; this!produce() }
   Void subscribe(CustomerI cl) { lastProxy:=lastProxy.add(cl) }
   Void produce() { Fut[News] fut :=prod!detectNews();
        proxy!publish(fut) } // sends future
}
class Proxy(Int limit,ServiceI s) implements ProxyI {
   ProxyI nextProxy;
   List[CustomerI] myCustomers:=empty; // fields
   ProxyI add(CustomerI cl) { ProxyI lastProxy:=this;
     if length(myCustomers)<limit
       then myCustomers:=append(myCustomers,cl)
       else if nextProxy=null
            then nextProxy:= new Proxy(limit,s) fi;
         lastProxy:=nextProxy.add(cl) fi;
     return lastProxy }
   Void publish(Fut[News]fut){ News ns := get fut; // wait for the future
     myCustomers!signal(ns); // multi-cast the result
     if nextProxy=null then s!produce() else
     nextProxy!publish(fut) fi }
}
class Producer(NewsProducerI np) implements ProducerI {
  // Wrapper for NewsProducer
   News detectNews(){ News news; news:=np.getNews(); return news }
}
class NewsProducer()implements NewsProducerI{ List[News] nl;
   Void add(News ns){nl:=append(nl,ns)}
   News getNews(){News n;
     await nl /= empty; n:=first(nl); nl:=rest(nl); return n }
}
class Customer implements CustomerI{ // Consumer of news items:
   News news; // the latest news
   Void signal(News ns){ news:=ns }
}
```

Figure 7.6: Classes providing an implementation of the subscription example

```
class Proxy(Int limit,ServiceI s) implements ProxyI{
   ProxyI nextProxy; List[CustomerI] myCustomers:=empty;
   ProxyI add(CustomerI cl){ ... }
   Void publish(Fut[News]fut){
     myCustomers!signal(fut); // send future, no waiting
     if nextProxy=null then s!produce() else nextProxy!publish(fut) fi}
   }

class Customer implements CustomerI{ News news; ...
   Void signal(Fut[News] fut) { news:= get fut } // blocking wait
   }
```

Figure 7.7: DoS attack by a variation of the subscription example

We then say that the cycle is flooding. The flooding cycle identified above is harmless provided the customers are able to process their signal calls as fast as the cycle iterations. The programmer will be warned by our algorithm about each possible flooding, and should determine whether it is a real problem.

In contrast, the modified program version (Figure. 7.7) does not wait in *Proxy.publish* (doing *ns*:=***get*** *fut*) until the newsletter is produced. Instead the future is directly passed to another asynchronous call ($myCustomers!signal(fut)$) in line 6 of Figure. 7.7 through the method *Proxy.publish*, this removes any progress dependency between the cycle producing the *Producer.detectNews* and *Customer.signal* calls and the processing of those calls. The completion of the *Producer.detectNews* and `Customer.signal` calls does not only depend on the speed of code execution, but depend on the rate of newsletter items arrivals. Practically, this flooding cycle generates a number of unprocessed calls that quickly grows to system limits.

Applying the algorithm to the example. Following [14], the *call* and *comps* sets for the two publish/subscribe versions are shown in Figures 7.8 and 7.9. Method names are abbreviated with two letters as indicated above, letting *Ng* abbreviate method *getNews* of interface *NewsProducerI*. There are two cycles in Figure. 7.8, i.e., cycle A and B. We have a flooding on the call to *Customer.signal* (Cs) in both cycles. However, this flooding does not reflect an actual flooding since the Customer objects

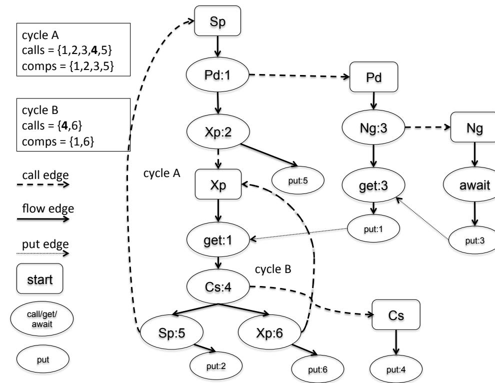

Figure 7.8: The graph and *call*/*comp* sets for the original version of the program (Figure. 7.6)

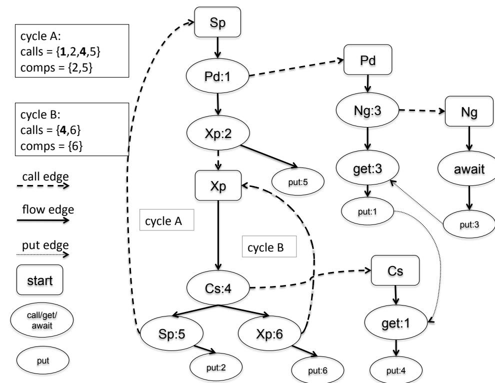

Figure 7.9: The graph and *call*/*comp* sets for the modified version of the program (Figure. 7.7)

easily keep up with the calls since the amount of work required by the Customer to complete a signal call is trivial. The execution rate is restricted with respect to the actual arrival of new items from the *NewsProducer* (by the blocking call in the proxies), and therefore, the rate of produced asynchronous calls to *Customer.signal* by this cycle is limited. Thus this is an example of *weak flooding* that is harmless.

```
class Attacker(ServerI s) {
  {this!run(); } // initialization
  Void run() { ClientI c := new Client(); c!connect(s);
    this!run() } // terminate and make recursive call
}
class Client() implements ClientI{
  Nat connect(ServerI s){
    Nat n := s.register();
     // blocking call, so a single client will not cause flooding
    return n }
}
class Server(DataBase db) implements ServerI{ {...} // Initialization
  Nat register(){Nat n :=0;
    if okcheck(caller) then Bool ok := db.open();
      if ok then n:=db.add(caller);
        db.query(...); db.close() fi fi; return n }
  // register requires time and resources
... }
```

Figure 7.10: Flooding by unbounded creation of innocent clients targeting the same server

Furthermore, cycle *B* is not infinite since it goes through the chain of Proxies. The modified version of the program is shown in Figure. 7.9. This version is displaying *strong flooding*. The flooding-cycle of *Pd* (Producer.detectNews) through both cycles is dangerous and will cause flooding of the system instantly. In version 1, there is a *get* in cycle A that regulates the speed of this cycle, whereas in the modified version there is no *get* in cycle A.

An example of instantiation flooding. The example in Figure. 7.10 shows how a *ClientDistribution* object can cause an attack by using an unbounded number of clients to flood the same server s, due to an unbounded recursion of the *run* method. The initialization of the attacker object of class ClientDistribution connects to a client, and the client do the registration of the server object. The attacker may start such a communication with lots of clients to register at the same server. (For simplicity, interfaces are omitted here.) Each client is innocent in the sense that it does not cause any attack by itself. By finding such a

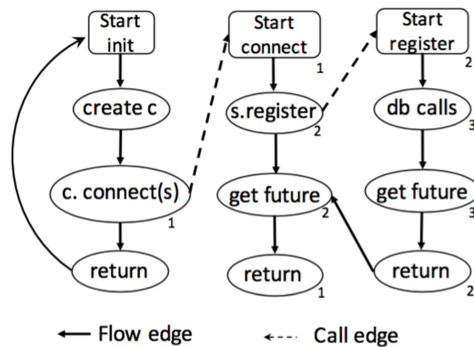

Figure 7.11: Static detection of flooding using unbounded creation

vulnerability in the ClientDistribution, an attacker can cause the flooding attack by calling `run()`. In addition, the non-blocking call in this method helps the attacker because the method does not wait for the `connect` calls to complete, therefore it is able to create more and more workload for the server s in almost no time. The execution of `f:=c!connect(s)` causes an asynchronous call and assigns a future to the call. Thus no waiting is involved. The run method recursively creates more and more objects, located somewhere in the distributed network. Therefore, the attacker creates flooding by rapidly creating clients that each performs a resource-demanding operation on the same server. Static analysis detects such attacks by finding a call loop (in this case inside `run`) which is also targeting the same server.

In this example, if the object creation in `run` had happened locally, an explicit instantiation flooding that consumes all the resources in an object will happen, which is a self DoS attack. However, since the object creation is distributed, the example in Figure. 7.10 shows an implicit attack because of targeting the same server by different clients.

Static analysis of the instantiation example. Consider the example in Figure. 7.10. For the run method of class ClientDistribution, the following cycle is detected:

the initialization of the attacker calls *run*
run creates a client object c
run calls `c!connect(s)`

run terminates and calls itself recursively in an asynchronous call.

The *run* call has a call edge to the flow graph of *connect* (call 1), and *connect* has a call edge to the flow graph of *register* (call 2). The call to *register* waits for completion of *register* since it is a blocking call, and the database calls (call 3) made by *register* wait for the completion of these database calls. The code for the database is not given, and therefore the analysis will be worst-case by considering the termination of such calls non-reachable (unless indirectly found strongly reachable). The control flow graph is given in Figure. 7.11. The set of weakly reachable call nodes of the cycle, i.e., *calls*, are $\{1, 2, 3\}$ with optimistic detection, and $\{1\}$ with pessimistic detection. And the set of strongly reachable calls, i.e., *comps*, is empty in both cases. This gives that the set of potentially flooding calls, given by $calls - comps$, is $\{1\}$ (c.connect) with pessimistic detection and $\{1, 2, 3\}$ with optimistic detection. However, in this case, call 1 does not reflect a real flooding since each call is on a separate object, but call 2 (s.register) and call 3 (the db calls) do. We detect strong flooding. The example may be improved by using suspending calls (using **await**) on the database operations.

7.6.1 Modification of the Static Detection to DDoS

As seen in Figure. 7.2, a DDoS attack on a server is often made through many innocent clients. This is hard to detect from the server side at runtime since each client may behave in an acceptable manner, and since the real attacker is hidden behind the clients. The original detection method [14] is not oriented towards such attacks, since it is not aware of the number of generated objects of a class. Moreover, the approach is using as an assumption that *an execution has a bounded number of objects*. Nevertheless, if applied to the example in Figure. 7.10, it will report a possible attack on the clients (treating all customers as one object), but not an attack on the Service server, which is the real attack. A draw-back is that there could be reported more false positives due to over-approximation.

A weakness with pessimistic detection is that the *connect* call, but not the *register* call, would be reported. Although the former call leads to the second, the detection result is not appropriate since a harmless

call is reported an not the harmful one. Another weakness is that the attack would not be discovered when removing the `connect` call from the attacker class and instead letting the `register` call be caused by the `init` method of class Client. (In this case the method parameter s should be transferred as a class parameter.) The reason for this is that indirect calls due to object generation are ignored (since by assumption there cannot be unboundedly many such calls). To compensate these weaknesses, we make two modifications wrt. [14], described below:

First, we modify the static analysis by viewing a **new** C statement as a special kind of a call statement with its own associated call number and a call edge to a copy of the *init* code of class C, which again may have further calls, treated as usual. More precisely, we treat **new** C as a *simple* call statement (like $new!C(classparameters)$ except that the new object is not known before the call) since the **new** statement does not wait for the *init* to complete. This allows us to see the generation of objects and to follow all implicit calls from the initialization code. Thus we can detect instantiation flooding attacks depending on call indirectly caused by object initialization. We may assume that an initialization cannot generate flooding in itself since each initialization is on a new object. Thus the call numbers associated with object creation can be included in *comps*. Furthermore, one more call on a new object cannot generate flooding on this object (unless in a cycle after the object creation). The same goes for a finite number of calls on a new object, if it can be detected statically that all these calls have the same new object as callee. These calls can also be included in *comps*. In the example of Figure. 7.10, we detect that the call `c!connect` is to the new Client object, and this call will then not be reported with the improved static detection.

Secondly, since implicit calls are important in DDoS attacks, we follow all call edges in the calculation of WR nodes, even in the pessimistic version. The resulting improved static detection method can then also detect the hidden attacker in all versions of the instantiation example, as shown below. Since the improved static detection method depends on static detection of same callee, we briefly discuss how to incorporate this: Two calls in the same method activation have the same callee if the callee is the same variable, and it is either

- a read-only variable (such as a parameter),

- a local variable and there are no updates on this variable between the calls, or
- a field variable and there are no updates on it nor suspension between the calls.

The first of these calls may be an object creation $x := \textbf{new}\ C(\ldots)$, and the second a call with x as callee (provided one of the conditions above are satisfied for x). This suffices for the example with the two calls $c := \textbf{new}\ Client()$ and $f := c.connect(s)$. This detection could be improved in several ways. In particular, we may detect that the actual parameter s in the latter call refers to the same object for all activations of *run* since s is a read-only class parameter and the recursive *run* call is on the same object (since `this` is read-only). This could be used in the detection algorithm to see that all *s.register* calls refer to the same server s, which gives a clear indication of a DDoS attack.

Instantiation example revisited. We reconsider the example in Figure. 7.10, using the improved static detection algorithm. Now the *create c* node of Figure. 7.11 is represented as a call, say call 0. For the original version of the example, the initialization is empty so call 0 is considered terminating ($0 \in comps$). We get $calls = \{0, 1, 2, 3\}$ where 0 corresponds to the creation of the new C object. But call 0 (`c:= new Client`) and call 1 (`c!connect`) do not generate flooding since they are on a new object. Thus $comps = \{0, 1\}$. This shows that there is a possibility of call flooding through call 2 (`s.register`) and call 3 (the `db` calls); and these correspond to actual attacks. For the modified version of Figure. 7.10, where the `register` call is caused by the Client initialization, we get a similar analysis except that there is no call 1 (`c!connect`) since this is incorporated in the Client initialization. Thus we get that $calls = \{0, 2, 3\}$ and $comps = \{0\}$. Here the presence of call 0 enables us to detect call 2 in the (modified) initialization code and thereby also call 3 (and both correspond to possible flooding).

7.7 Conclusion

In this paper we have considered denial of service attacks, formulated in a high-level imperative language based on concurrent objects communicat-

ing by asynchronous calls and futures, thereby supporting asynchronous as well as synchronous communication. The language includes mechanisms for process control allowing non-trivial process synchronization by means of cooperative scheduling. We adapt a static detection algorithm developed for analysis of flooding to this setting, in order to detect possible denial of service attacks. This kind of static analysis is useful in the financial sector, because the aspect of trust between customers and service providers is essential, perhaps more so than in other application areas, and therefore static detection is valuable.

We have illustrated the approach on examples of distributed systems in the financial sector, including versions of a one-to-many attack and a many-to-one attack. In the first example a financial institution notifies a number of subscribing customers. We have seen that a revision of the basic notification software used by the financial institution, intended to be more efficient, actually implies a one-to-many attack on the subscribing customers. In this example, the financial institution was responsible for the attack, which could lead to loss of reputation and of customers. Static detection solved the situation here since the detection is made before the program is run. The underlying detection algorithm is sound for call-based coordinated attacks, provided the source code of the objects involved in the coordinated attack is available. In the many-to-one example, an attacker object causes an attack by using an unbounded number of clients, each innocent, to flood the same server s, letting a new client be created in each cycle.

In this paper we have adapted a general algorithm for detecting flooding [14] to the setting of DDoS and improved it to deal with unbounded object generation and to better reveal hidden attacks. Our framework can deal with advanced programming mechanisms including suspension and first-class futures considering distributed systems at a high-level of abstraction. It is therefore relevant for high-level modeling and prototyping of distributed software solutions. In future work, we suggest to complement the static checking with dynamic runtime checking since static detection methods give a degree of over-estimation. This could give a more precise combined detection strategy.

Acknowledgments. We thank the reviewers for significant feedback. This work is supported by the *IoTSec* project, the Norwegian Research

Council (No. 248113/O70), and by the *SCOTT* project, the European Leadership Joint Undertaking under EU H2020 (No. 737422).

Bibliography

[1] Ashford, W. *DDoS is most common cyber attack on financial institutions*. https://www.computerweekly.com/news/4500272230/DDoS-is-most-common-cyber-attack-on-financial-institutions/. 2016.

[2] Boer, F. D. et al. “A Survey of Active Object Languages”. In: *ACM Comput. Surv.* vol. 50, no. 5 (Oct. 2017), pp. 1–39.

[3] Chang, R. et al. “Inputs of coma: Static detection of denial-of-service vulnerabilities”. In: *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE. 2009, pp. 186–199.

[4] Colón, M. and Sipma, H. “Synthesis of Linear Ranking Functions”. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2001. London, UK, UK: Springer-Verlag, 2001, pp. 67–81.

[5] Din, C. C. and Owe, O. “A sound and complete reasoning system for asynchronous communication with shared futures”. In: *J. Logical and Alg. Methods in Prog.* vol. 83, no. 5 (2014), pp. 360–383.

[6] Douligeris, C. and Mitrokotsa, A. “DDoS Attacks and Defense Mechanisms: Classification and State-Of-The-Art”. In: *Computer Networks* vol. 44, no. 5 (2004), pp. 643–666.

[7] Gulavani, B. S. and Gulwani, S. “A numerical abstract domain based on expression abstraction and max operator with application in timing analysis”. In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 370–384.

[8] Jensen, M., Gruschka, N., and Herkenhöner, R. “A survey of attacks on web services”. In: *Computer Science-Research and Development* vol. 24, no. 4 (2009), p. 185.

[9] Johnsen, E. B. and Owe, O. “An asynchronous communication model for distributed concurrent objects”. In: *Software & Systems Modeling* vol. 6, no. 1 (2007), pp. 39–58.

[10] Johnsen, E. B. et al. "ABS: A Core Language for Abstract Behavioral Specification". In: *Formal Methods for Components and Objects*. Ed. by Aichernig, B. K., Boer, F. S. de, and Bonsangue, M. M. Springer, 2012, pp. 142–164.

[11] Karami, F., Owe, O., and Ramezanifarkhani, T. "An evaluation of interaction paradigms for active objects". In: *J. Logical and Alg. Methods in Prog.* vol. 103 (2019), pp. 154–183.

[12] Lambert, K. *Protecting Financial Institutions from DDoS Attacks*. https://www.imperva.com/blog/protecting-financial-institutions-from-ddos-attacks. 2018.

[13] Meadows, C. "A formal framework and evaluation method for network denial of service". In: *Computer Security Foundations Workshop. Proceedings of the 12th IEEE* (1999), pp. 4–13.

[14] Owe, O. and McDowell, C. "On detecting over-eager concurrency in asynchronously communicating concurrent object systems". In: *Journal of Logical and Algebraic Methods in Programming* vol. 90 (2017), pp. 158–175.

[15] Qie, X. and Larry Peterson, R. P. and. "Defensive programming: Using an annotation toolkit to build DoS-resistant software". In: *CM SIGOPS Operating Systems Review, 36(SI)* (2002), pp. 45–60.

[16] Ramezanifarkhani, T., Fazeldehkordi, E., and Owe, O. "A Language-Based Approach to Prevent DDoS Attacks in Distributed Object Systems". In: *29th Nordic Workshop on Programming Theory*. (extended abstract, 3 pages). Turku Centre for Computer Science, Nov. 2017.

[17] Urrico, R. *Denial of Service Attacks Overwhelmingly Target Financial Services: Verisign*. https://www.cutimes.com/2018/07/03/denial-of-service-attacks-overwhelmingly-target-fi/?slreturn=20190713065814/. 2018.

[18] Wilczek, M. *Why Banks Shouldn't Be in Denial About DDoS Attacks*. https://www.globalbankingandfinance.com/why-banks-shouldnt-be-in-denial-about-ddos-attacks/. 2018.

[19] Zahoor, Z., Ud-din, M., and Sunami, K. "Challenges in Privacy and Security in Banking Sector and Related Countermeasures". In: *International Journal of Computer Applications* vol. 144, no. 3 (2016), pp. 24–35.

[20] Zargar, S. T., Joshi, J., and Tipper, D. "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks". In: *IEEE Comm. Surveys Tutorials* vol. 15, no. 4 (2013), pp. 2046–2069.

[21] Zheng, L. and Myers, A. C. "End-to-end availability policies and noninterference". In: *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop* (2005), pp. 272–286.

CHAPTER **8**

Paper 4: A Language-Based Approach to Smart Contracts Supporting Safety and Security

Authors: Elahe Fazeldehkordi, Olaf Owe

Publication: Submitted to the Journal of Logic and Algebraic Programming (JLAMP), April 2020, 51 pages. (under revision)

Abstract

The concept of smart contract represents one of the most attractive uses of blockchain technology and has the advantage of being transparent, immutable, and corruption-free. We propose a new approach to define smart contracts, offering a similar trust at the software level, even without use of blockchain. This approach can apply to a wide range of contracts, not only financial ones, and opens up for lightweight smart contracts without the resource and energy costs of blockchain. Our framework integrates trust at the language level through the notion of history objects. For each contract, a history object will record all related transactions. The history objects are specially protected objects, with read-only access at the programming level. Contract partners may interact with the history objects through predefined interfaces. We show that a history object can be used to provide safety, security, and privacy, as well as runtime checking. A history objects can provide runtime checking of specified behavioral properties of the contracts.

We present a framework for lightweight smart contracts with associated history objects. Our framework consists of an executable and imperative language for writing smart contracts and specifications by means of invariants referring to the transaction history of a contract, as well as a verification system. The framework is oriented towards simple verification, and we are

able to do sequential style reasoning in a class-wise manner. We demonstrate the approach on an auction system.

8.1 Introduction

Blockchain technology started initially with a new currency called Bitcoin that was based on automated consensus between networked users, not required to trust each other. Financial industries related to cryptocurrency have been seen as primary users of this technology and have resulted in the most widespread applications of blockchain, but its applications go far beyond financial ones. The concept of smart contracts represents one of the most attractive uses of blockchain technology that has appeared recently.

A *smart contract* is a program that executes the terms and conditions of an agreement that are predefined by mutually distrusting participants of the agreement. These programs are stored on blockchain, and their correct execution is enforced by the consensus mechanism of the blockchain without depending on a trusted authority. Compared to traditional contracts, smart contracts provide trust with low legal and traditional costs, no risk of tampering and fraud, no interference or trust issues of a third party. Apart from these advantages, smart contracts are automatic, fast, and transparent. The number of applications of smart contracts in industry and everyday life is countless. Most applications include digital identity, banking, tax records, insurance, real estate and land title recording, supply chains, IoT, gaming and gambling, auctions, authorship and intellectual property rights, life science, and health care.

Despite many advantages, there are also some drawbacks in smart contract technology. The concept of smart contracts is built on blockchain; therefore, it is expensive with respect to time, resources, and power consumption. Besides that, since it has the consensus mechanism of the blockchain at the bottom, privacy is discarded. Our ambition is to suggest an approach that avoids these disadvantages.

In this work, we propose a new construct at the programming language level that supports the main advantages of smart contracts based on blockchain, including trust, immutability, and transparency, but is less expensive to implement since it does not need to use blockchain. It gives trust at the application layer without use of blockchain. However, it can

also be combined with the blockchain technology in order to improve the overall trust level on insecure platforms. Furthermore, our approach offers better privacy control and comes with a theory for formal specification and verification.

More specifically, we propose a “container box” for recording all calls and *futures* (see Section 8.4.4), related to the interactions involving a given contract service provider. This “container box” will then hold all transactions such as calls to/from the contract and future values generated by the contract, including present and past communications. For this reason, we will call it a *history object*. It can also be seen as a “ledger” since it records all the transactions involving communication with the service provider. The transaction history is generated by the underlying system, and programmers have only read access to history objects. We associate one history object to each contract. We predefine classes and interfaces for these history objects, restricting write access. These interfaces and classes may be extended through inheritance in the same way as in object-oriented languages, something which allows addition of safety, security, and privacy aspects. The redefined history classes must adhere to the restriction of no write access.

The need for formal verification of smart contracts is pointed out in several papers [2, 17]. Solidity is the most dominant language used for writing smart contracts; however, a main drawback with Solidity is that it is not well suited for specification and reasoning since it lacks a formal semantics and is not oriented towards program reasoning [40]. For instance, class-wise verification is not supported, and even soundness can be a problem. In case of errors, Solidity uses roll-backs to return to a previous state, reverting all the modifications made until the last safe state. Reasoning about roll-backs is challenging since the cause of a roll-back can be implicit.

We therefore consider a high-level language that allows better reasoning support than Solidity. Our approach avoids the mentioned problems with Solidity. Furthermore, our language gives fewer runtime errors and less need for roll-backs, which simplifies reasoning. In particular, our approach supports class-wise verification, i.e., we can verify a class invariant by looking at the class itself (and inherited code from superclasses), without looking at other classes nor prior knowledge of the environment. This is essential for scalability and open-ended program development, which are highly relevant factors for contracts.

In order to define history objects and contracts as autonomous distributed objects, our language is based on the active object paradigm [39]. This paradigm offers a natural and high-level understanding of service-oriented systems, and with a modular semantics, which is essential when we turn to specification and verification issues. Clients, contract, and history objects are then described by concurrent and distributed objects. The language builds on the principle of interface abstraction, i.e., remote field access is illegal, and an object can only be accessed through an interface. Each object has one or more interfaces, and the only possible way of object interaction is through the methods defined in the corresponding interfaces. Our language combines first-class futures [5], which is often used in active object languages, and a restricted version of cooperative scheduling [10]. This novel combination gives flexible method interaction, scheduling control, simplified reproducability of executions, as well as simplified verification. Furthermore, we show that the history objects provide the functionality of the future mechanism, and may play the role of futures. Our solution avoids the need for garbage collection of futures and improves the privacy control of future values.

We demonstrate our approach on a smart contract example, namely the auction example used by Ahrendt, Pace, and Schneider in [2]. We exemplify an active object defining an auction and an associated history object, and demonstrate how our approach can be used to provide security, privacy, safety, as well as high-level functional specifications that can be checked at runtime. We show how the verification of the auction contract specification can be done in a simple manner, by means of sequential-style, class-wise reasoning.

The main contributions of this paper are the notion of history objects giving rise to lightweight smart contracts and a framework integrating this notion, by developing an imperative language for contracts, an executable functional language for writing smart contract specifications, and a theory for class-wise verification, with support of privacy, security and model checking of contract specifications.

Outline. Section 8.2 describes relevant background on smart contracts and blockchain, and Section 8.3 introduces an underlying programming and specification language. Section 8.4 presents our suggested framework based on this language, and Section 8.5 presents our formalization on an auction system. Section 8.6 discusses how to verify contracts, including an application to the example. Section 8.7 gives a comparison with

Solidity and blockchain. The two last sections (8.8 and 8.9) present related work and a conclusion, respectively.

8.2 Smart Contracts and Blockchain

In systems operating with a centralised model, parties who wish to trade with each other have to do this via the central system, which the parties should trust. Therefore, all the trust should be placed on the central system, and all the business transactions depend on this third party. This dependency could be costly for both parties of a transaction. Smart contracts came to solve these problems. The term smart contract was introduced by Szabo in 1997 [36]. It refers to simple programs that store rules for negotiation and terms of a contract. These terms will then be checked by smart contract. Using smart contracts, untrusted parties can trade directly with each other. Smart contracts are stored on the blockchain, and each party has a copy of it.

A blockchain is a distributed *ledger* that is open to anyone; it stores the information across a network of computers. In general, a ledger is a list of records that can be in any form, like a notebook, or an excel file. In blockchain, a ledger is given by the complete information about all transactions of some kind, typically transactions with financial aspects. A distributed ledger is distributed across many locations instead of placing it in a fixed location. Blockchain is a chain of blocks. Each block holds some data together with the hash of the block, which is unique just like a fingerprint, and also the previous block's hash. The stored data inside the block depends on the type of blockchain; for instance, the Bitcoin blockchain stores details about the transactions, like the sender, the receiver, and the number of coins. Immediately upon creating a block, its hash is calculated. Any tampering inside a block will cause its hash to change, and therefore it makes the next block and all the following blocks invalid since they no longer store a valid hash of the previous block.

The use of hashes is not enough to prevent tampering, since computers nowadays can calculate hundreds of thousands of hashes per second, which means that someone can tamper with a block and calculate all the hashes of the other blocks again to make the blockchain valid. In order to mitigate the risk of tampering, a blockchain also uses a mechanism that is called *proof-of-work*, a mathematical computation that slows down

the creation of the new blocks. This mechanism makes it harder to tamper with the blocks because if someone tampers with one block, he/she must calculate the proof-of-work for all the following blocks again. Therefore hashing and the proof-of-work mechanism provide trust in blockchain. Nevertheless, there is one more way that blockchains can secure themselves, namely by being distributed. Blockchain uses a peer-to-peer network, and everyone can join. When someone joins the network, he/she receives a full copy of the blockchain. The node uses this to verify that everything is in order. When someone creates a new block, that block will be sent to everyone on the network, each node can verify the block to make sure that it has not been tampered with, and if validity is accepted by the majority of the nodes, each node adds this block to their blockchain. All the nodes in the network create consensus, agreeing about which blocks are valid and which are not. This consensus promotes transparency and makes blockchains corruption-proof. Other nodes in the network reject those blocks that are tampered with. Therefore, without the consent of the majority of the nodes, no one is allowed to add a transaction block to the ledger. Besides, once a transaction block is added to the ledger, nobody can change it. So, no single user in the network can modify, delete, or update the blocks. This characteristic promotes immutability (i.e., something that cannot be changed) and makes sure that the blocks remain unchanged. All the fundamental characteristics of the blockchain technology are also shared with smart contracts since smart contracts are based on the blockchain technology.

Blockchain, coupled with smart contracts technologies, removes the dependence on central trusted systems between trading parties. If any party tries to change a transaction on the blockchain, it can be detected and prevented by all the other parties on the network. Since smart contracts run on the blockchain, they behave like a single massive secure self-operating computer program that executes automatically under certain conditions, but without the risks of tampering or fraud, extra costs, distrust or interference issues of a third party.

Smart contracts can be applied to many different areas. Smart contracts allow not only exchange of money, but also property, stock, or anything else without having to go through a lawyer, a notary or other centralised service providers. They entirely cut out the need for a middle man. Banks, for instance, can use smart contract to issue their loans or to offer automatic payments, insurance companies can use it to

process specific claims, or postal companies can use it for payment on delivery. Other examples of smart contracts deal with escrow agreements, employment agreements, auctions, and voting systems.

Ethereum, the most prominent smart contract platform today, was first proposed in late 2013 by Vitalik Buterin [9]. Ethereum is an open software platform and is based on blockchain technology that enables developers to build and deploy decentralised applications. It focuses on running code for decentralised applications that deploy on its network. When we write such decentralised applications, we write them in the form of smart contracts. Ethereum's blockchain not only allows currencies to reside on it but also software code. Parties or smart contracts in Ethereum can communicate with each other via transactions, in order to distribute assets between each other.

In Ethereum, a smart contract is like an object in object-oriented languages such as C++ or Java. Objects in Ethereum are parties. Each party can have its own state and logic, similar to objects having variables and methods in object-oriented languages, especially in the active object paradigm where the objects are distributed and autonomous and can have active behavior.

Several programming languages have been used for writing smart contracts on Ethereum like Solidity, Serpent, and Lisp Like Language (LLL). LLL is similar to the Lisp language and was used mostly in the very early history of Ethereum and is probably the hardest to write in. Serpent is similar to the Python language and was popular in the early days of Ethereum, but the most popular and functional one currently is Solidity that is very similar to JavaScript.

Solidity is a contract-oriented, high-level programming language (high-level programming languages refer to highly abstracted languages that are far easier to use for humans). It is statically typed; it supports inheritance, libraries, and complex user-defined types, among other features. Solidity builds on Ethereum Virtual Machine (EVM), which executes an associated low-level bytecode language. This is similar to bytecode as used in the Java JVM or C# CLR. There are several compilers (e.g., SolC, a browser-based compiler) that compile smart contracts written in Solidity into EVM bytecode, which can then be deployed into the Ethereum blockchain and will be ready to receive transactions. So the entire lifecycle of a smart contract in Ethereum is as

follows:

$$Solidity \longrightarrow EVM\ bytecode \longrightarrow Deployment$$

In the EVM machine code, there are several operations. In Ethereum, each operation has a cost, in order to execute the smart contract, all the operations need to be paid. Ethereum has its own currency, which is named *ether*. Every transaction and execution of bytecode costs ether.

Gas, on the other hand, is a unit that translates into the ether; for instance, if there are several instructions, the first instruction might cost two gas, and two gas get translated into some number of ethers. The reason for separating gas and ether is to decouple the price of an operation with the market price of an ether. The gas price for an operation is constant, and it cannot easily be changed; however, we can change how much each gas costs in terms of ether. In this way, we decouple the market price of an ether with how much we actually pay for each operation. A list of operation codes and how much each operation costs in terms of gas can be found in the Ethereum yellow paper (with the formal definition of the Ethereum protocol) [38]. In order to distribute assets between different parties in Ethereum smart contracts, each party sends ether via the transactions.

8.3 A High-Level Language for Active Object systems

Before defining history objects, we need an underlying language for defining transactions, histories, and behaviour. We present a high-level language supporting active objects, based on the Creol/ABS language family [21, 25].

In particular, we define a functional language for defining interfaces (Section 8.3.1), data types and functions (Section 8.3.2), and then an imperative language for defining classes supporting concurrent active objects (Section 8.3.3). Our language is strongly typed, and object variables are typed by an interface (not a class). In order to enable verification and specification of behavior, we build on language constructs for specification and reasoning, supporting class-wise reasoning [29, 30]. This means that the clients of a contract can rely on the abstract

specification of the interface of the associated history object rather than program code. Furthermore, they may interact with the history objects if they do not trust the contract objects. The interfaces define visible methods and their specifications. A class may have several interfaces. Methods of a class that are not exported through an interface are considered private.

8.3.1 Interface Definitions

An interface defines a view of a class object, in terms of methods, together with an invariant describing behavior and related data types and functions. We define the syntax of interfaces with an augmented Backus-Naur Form (BNF) regular expression:

```
interface I [[ I+ ]] [ extends I+ ] {
  [ [ with I ] [ T m([T x]*) ]+ ]*
  [ type and function definitions ]*
  [ invar A ]*
}
```

The superscripts * and $^+$ are used to denote repeated parts (* for zero or more and $^+$ for one or more repetitions), and the meta-symbols $[\,]$ (without a superscript) are used to indicate optional parts. In order to allow generalized interface definitions, an interface may by parameterized by a number of types or interfaces using the syntax $[\mathrm{I}^+]$ (where the brackets are part of the syntax). For example, **interface** $History[I]$ is generalized over I.

We let I denote an interface name, T a type name, m a method name, x a formal parameter, and A an invariant assertion, which may refer to the local transaction history h, and user-defined functions on the history. An invariant is an assertion that must be true before and after each method execution. Data types and function definitions are explained in the next subsection.

The BNF interface definition shows how to define an interface I by extending a number of already defined interfaces (I^+), defining a number of methods, types, and functions. A (directly or indirectly) extended interface is said to be a superinterface of I, and interface I is said to be a subinterface of its superinterfaces. Note that a method may have a

```
interface Bidder {
  Void newBid(Nat x) // inform a bidder that x now is the highest bid
  Void youwon(Nat x) // inform the bidder that he/she won with bid x
  Void winner(Bidder o) // method to inform a bidder/owner that o won
}

interface Auction {
  Nat highest() // gives the highest bid in the current auction
  with Bidder // only bidders may call the methods below
    Void open() // to open a new auction
    Bool close() // to close the current auction
    Bool makeBid(Nat x) // to place a bid in the current auction
}
```

Figure 8.1: Interfaces for the auction example

cointerface, given by a **with** clause, which defines the (minimal) interface of the caller object. An object calling the method must be typed by a subinterface of the cointerface.

Examples of interfaces for an auction system are given in Figure 8.1, and a cointerface is used. Interface Bidder defines the methods of bidder objects, and interface Auction defines the methods of the auction service provider. The cointerface used in interface Auction restricts clients calling *open*, *close*, and *makeBid* to Bidder objects (i.e., objects of classes implementing the Bidder interface).

A *cointerface* is needed when a method body is making calls back to the caller. In order to make these call-backs type-correct, we need to declare an interface for the caller (by a **with** clause). For instance, a method defined after the clause **with** $Bidder$, knows that the caller supports interface $Bidder$ and is therefore allowed to make (type-correct) calls to methods of that interface. In the example below, Bidder is such a cointerface.

Our language uses interfaces to describe distributed objects, which may have active behavior when desirable, and uses data types to define data structure local to an object. Since an interface declares no state variables, an interface can only talk about the transaction history h. Moreover, by means of functions defined over the history, one may express essential aspects of objects of the interface and define an abstract

state. In a distributed, open-ended, and unpredictable environment, reasoning by means of history invariants is more suitable than reasoning based on pre- and post-conditions, since pre/post-conditions on the callee side will in general need to refer to variables not found on the caller side (when these conditions need to talk about more than just the input/output). We will restrict ourselves to executable history invariants, and include runtime checking of invariants by means of history objects.

In the next section, we define a functional language for defining data types and functions, and then use this to define transactions, histories, and contract specifications.

8.3.2 Data Type and Function Definitions

For the definition of data types and functions, we use a syntax typical for strongly typed functional programming languages with pattern matching. We consider an executable functional language for data types and functions. As this language also is used for specifications, we obtain an executable specification language. A user-defined data type is defined by (named) disjoint unions (... | ...) and products (... $*$ ...), allowing recursive definitions. A disjoint union may look like $c_1 : E_1 \mid c_2 : E_2 \mid ...$ where E_i is a type expression, often a product and c_i is implicitly defining a *constructor function* used to name an alternative in a disjoint union. Thus, a data type T is defined by a number of constructor functions for constructing T values. Each constructor function may have a number of input parameters. For instance, the list type could be defined by **type List**[T] = nil: | append: **List**[T] $*$ T where *nil* and *append* are constructor functions. As *nil* has no input parameters, it is called a constant constructor. For convenience, we let the *append* constructor function be denoted by the infix symbol ";" and we omit writing an empty product. Generalized data type definitions are parameterized by one or more types or interfaces using the syntax $[\mathrm{I}^+]$ as before. The list type can then be declared by the syntax:

```
type List[T] = nil: | _;_ : List[T]*T
```

where the underline indicates argument positions of functions with infix notation (and if needed, more generally for mixfix notations). The transaction history a, b, c can then be expressed as $nil; a; b; c$. The constructor functions define the set of possible values (as variable-free

constructor terms) and are implicitly defined. Standard types such as Void, Bool, Nat, String are predefined with standard functions. The type Void is used to represent no information (i.e., a unit type), and $void$ is the only value of this type.

User-defined functions (other than constructors) are defined by a **func** declaration stating the input and output types of each function and by a number of equations where the left- and right-hand sides are expressions consisting of functions, constructor functions, and variables. The types of these variables are declared in a **var** clause. We restrict ourselves to executable functions (using mulitset-path ordering to avoid non-termination recursion). We use the BNF syntax:

```
[ func f : T* → T ]*  // function declaration
[ var [T x]*]  // variable declaration
[ lhs = rhs [ if cond ] ]*  // function definition
```

Here f is a function name, lhs (left-hand side) and rhs (right-hand side) are expressions, and $cond$ is a boolean condition. A left-hand side defines a pattern and may contain the special symbols _ and **others** representing an arbitrary pattern and cases not covered, respectively. Any variable in a right-hand side must occur in the left-hand side.

Examples are given in Figure 8.2. The last $length$ equation can also be written as `length( q ; _ ) = length(q) + 1`. We define the ends-with operator (**ew**), with infix notation using _ to indicate the argument positions as before, expressing that a list q ends with a given element x. A left-hand side may use *others* to match any other case not covered by the equations above. For instance, the last equation for **ew** could be replaced by the two equations $(q; x)$ **ew** $x = true$ and $(q;$ **others**$)$ **ew** $x = false$.

In order to deal with partial functions, we allow **error** in the right-hand side, for instance as in the definition of the above *last* function over lists. By static checking, one can ensure that a function definition properly covers all cases of possible inputs (without conflicting overlap). For instance, if the equation $last(nil) =$ **error** is omitted, the static checking would detect that $last(nil)$ is not defined.

8.3.3 Imperative Class Language for Active Objects

We now show a way of defining classes. For the purpose of defining classes, we introduce a high-level language combining the active object

```
func length : List[T] → Nat // list length
func _ew_ : List[T] ∗ T → Bool // ends-with test
func last : List[T] → T // finds the last element, if any

var List[T] q, T x
length(nil) = 0
length(q;x) = length(q) + 1

nil ew x = false
(q;x) ew y = if x=y then true else false

last(nil) = error
last(q;x) = x
```

Figure 8.2: Examples of function definitions

paradigm, first-class futures, i.e., allowing futures to be passed like parameters to other objects, and guarded methods, i.e., identifying the conditions under which the methods are accepted [10]. We define classes with the BNF syntax below:

```
class C [[ I⁺]] [([T x]⁺)] [implements I⁺] [extends C⁺] {
  [T w]* // fields
  [body] // class constructor
  [ [ with I] [ T m([T x]*) {body} ]* ]* // method definitions
  [ type and function definitions]* // as defined above
  [ invar A]* // invariant specification
}
```

In order to allow generalized class definitions, a class may by parameterized by a number of types, interfaces, and classes using the syntax $[\,\mathrm{I}^{+}]$ as before. A class C may implement a number of interfaces and extend a number of *superclasses* (A derived class is called a subclass, and the classes from which it is derived are superclasses). Thus we support multiple inheritance, but single inheritance suffices for our example. The body of the class definition consists of a number of field declarations and method definitions, possibly a class constructor method as well as additional function, type, and invariant definitions. Any type or function

defined in an interface of a class is available in the class. Class parameters, fields, methods, as well as type and function definitions, are inherited in a leftmost, depth-first traversal of the inheritance tree. A class must implement the methods of its interfaces and respect their invariant(s). For each method in an interface, the type of the parameters and cointerface (if any) must be a subtype of the types in the corresponding method definition in the class, and the result type in the class must be a subtype of that in the interface. A class invariant A may refer to class parameters and fields as well as the transaction history.

We allow an inherited name to be qualified by the superclass name in order to deal with inheritance of multiple definitions of the same name. For instance, if a class extends A, B, and both superclasses have a method m, then the subclass may refer to these as $m@A$ (or just m) and $m@B$. Invariants and implements clauses are not inherited, which allows a subclass to freely redefine methods and invariants without the semantic constraints of the superclass. When desired, an invariant OK of a superclass C can be inherited by stating **invar** $OK@C$. Similarly, we may reuse an invariant from an interface I by saying $OK@I$. We refer to [30] for more details.

We consider a syntax similar to that of the Creol/ABS language family. Let v denote a variable, o an object variable (an object reference), or f a future variable (a reference to a future object), e an expression (assumed to be pure), and m a method. A variable is either a field w, a local variable y, or a formal parameter x (assumed to be read-only). We let this denote the current object, and a method has fid and caller as implicit parameters, giving the future identity of the call and the caller object, respectively. We use capitalized words for types and interfaces, while variable and method names start with a lower-case character. Class names are written in upper-case characters.

A method body has the form:

when $guard$; $\overline{T\ y}$; *statements*; **return** e

The body may have an (optional) initial guard, written **when** $guard$, in order to make sure the starting state is appropriate, otherwise the execution is delayed. The rest of the body consists of a number of typed local variables ($\overline{T\ y}$), a statement list, and a final **return** e statement, where the value of the expression e is the resulting value, which must

be of the method result type (the return statement **return** $void$ may be omitted).

The *guard* is either a boolean condition, which must be satisfied when the method starts, or the special construct $f?$, which checks that the future f is resolved. A when clause can be compared to the notion of method modifiers in Solidity, but rather than resulting in a runtime error as in the case of method modifiers, a when clause will delay the method execution to a state where the guard is satisfied.

Methods with initial guards are expressive enough to avoid blocking calls and blocking get statements. The resulting value of a future generated by one method execution can be picked up in a guard by another method execution, either on the same object or on another object (which knows the future). Methods with an initial guard are semantically simpler than methods with internal guards, as suggested in [10]. The combination of first-class futures and initial guards has not been used before.

We use the syntax **if** cond **then** s [**else** s] **fi** for if-statements (with optional then-part) and **while** cond **do** s **end** for while-statements. Assignments have the syntax $v := e$ where v is a variable and e a (pure) expression. We may abbreviate $T\ v; v := e$ to $T\ v := e$.

The statement syntax $v :++\ e$ is permitted when v is a list, to express that an element e is appended at the tail of the list v (this can be seen as an abbreviation of the assignment $v := (v; e)$). This statement adheres to the *write-once* discipline since it cannot change previously written elements. This discipline gives a protection comparable to blockchain. We, therefore, allow the $:++$ statement (and only by the underlying system), but not assignment on transaction lists.

We consider five kinds of method calls. In order to avoid or control blocking calls, we combine one-way asynchronous calls, guards, and the future mechanism, which is popular in active object languages.

- A *synchronous (or blocking) call* has the syntax $v := o.m(\overline{e})$, where o is the callee, m the called method, and $\overline{e}$ the list of parameters. The caller is blocked until the result is available. The method result will be assigned to v (when the callee is the same object as the caller, the call reduces to an ordinary stack-based local call).

- A *simple asynchronous call* has the syntax $o!m(\overline{e})$. The caller is

not blocked and cannot access the result value. It is used when the caller does not need the result value.

- A *broadcast* has the syntax $list!m(\bar{e})$ where $list$ is a list of objects. The caller is not blocked, and the result values are not communicated to the caller.

- An *asynchronous call* has the syntax $f := o!m(\bar{e})$ where f is a future variable declared with type $Fut[T]$ where T is the return type of m. The future variable f is assigned a new call identity (a reference to the future object) uniquely identifying the call. This identity may be communicated to other objects. The caller is not blocked. In order to obtain the value returned from the call, the caller object (or another object that knows the future identity) may perform $v :=$ **get** f where v is a program variable of type T. This statement will block if the future is not resolved, otherwise the result value is copied into v.

- An *error handler* can be appended to an assignment-like statement, including assignments, incremental assignments, synchronous calls, and get statements, using the syntax $< s >$ where s is the list of statements to be performed when the execution of the statement results in an error. And if there is no handler, the current method returns an error. For instance $list : ++ last(l) < skip >$ will have no effect when the $last$ function returns an error (when l is empty). Without the handler, the method would result in an error. More advanced forms of exception handling would be beyond the scope of this paper.

The combination of futures and guards allows a programming style without blocking calls. For instance, the blocking call to m in the code fragment $x := o.m(..); s$ where s is the rest of the body, can be replaced by the non-blocking call to m in the fragment $f := o!m(..); this!n(f)$ where n is defined as the guarded method $Void\, n(Fut[T]\ x)\{$**when** $f?; T\ x :=$ **get** $f; s\}$. The guard ensures that the get command will success immediately. We here assume s is without local variables and return (local variables can be transmitted as parameters, and returns can be handled by delegation, using the mechanism of [29]). For simplicity, we will not allow local synchronous calls of guarded

methods. This ensures that the history will determine the execution order since it determines all external inputs, and there is no other internal source of non-determinism.

The implementation of guarded methods involves cooperative scheduling, letting each object have a *process queue*. When the guard is not satisfied, the executing process (the method invocation) is placed on the object's process queue, and the object becomes *idle*. When idle, an object may continue with an enabled process from its queue or start an incoming call (say, taking the oldest enabled option). A suspended process is *enabled* when the guard is satisfied. An operational semantics for our language can be defined from the set of operational semantics outlined in [26] by selecting the rules for our set of language features, including first-class futures and suspension (the restriction to initial guards is ensured by the syntax and need not be reflected in the operational rules). Rules for exception handling and the recording of transactions (by put calls) must be added.

An example of a class defined in this language is given in Figure 8.3, showing a class implementing the Auction interface given in Figure 8.1. It defines class AUCTION with a number of fields and invariants. The invariants restrict the values of the fields of Auction objects when idle. The example illustrates the use of the implicit parameter `caller`, and the methods *open*, *close*, and *makeBid* use a cointerface Bidder to restrict the callers to Bidder objects. This is needed for type correctness since the caller of *open* is assigned to *owner*, which is typed by Bidder, and method *close* sends a message to the owner object through the Bidder interface. A similar discussion applies to makeBid as well, whereas method *highest* must not have a cointerface since none is given in the interface. An implementation of a method must have the same (or wider) cointerface as in the interface.

8.4 The Proposed Framework

We consider the setting of distributed concurrent objects. A smart contract in this setting is reflected by an object providing a certain service to the environment. Such an object is called a *contract object*. We assume a smart contract supports a predefined interface Contract (i.e., the class of the object implements interface Contract). Consider a contract to be used

```
class AUCTION implements Auction {
Bool isopen:=false; // tells if the auction is open or closed
Nat highBid:=0; // the current high bid
Bidder highBidder; // the current high bidder
Bidder owner; // the auction owner
List[Bidders] bidders:=nil;

invar isopen=(owner≠null)
invar owner=null ⇒ highBidder=null
invar highBidder=null ⇒ highBid=0

Nat highest() {return highBid }

with Bidder
  Void open() {if not isopen then owner:=caller; isopen:=true fi }

  Bool close() { Bool ok:=(caller=owner);
   if ok then // only owner may close the auction
    if highBid > 0 then // isopen follows by the invariant
      owner!winner(highBidder); // or: bidders!winner(highBidder);
      highBidder!youwon(highBid); // to notify the winner
      highBid:=0 fi;
    owner:=null;
    highBidder:= null; bidders:= nil;
    isopen:=false fi; return ok }

  Bool makeBid(Nat x){ bidders :++ caller;
    if open and x>highBid then
      highBid:=x; bidders!newBid(x);
      highBidder:= caller; return true
    else return false fi }
}
```

Figure 8.3: The auction class

by a number of clients. We suggest to add an additional object, called the *history object*, associated with the contract, storing the history of transactions related to the contract. This object is provided automatically

```
interface Future[I] { // generalized interface definition, with I as parameter
  type Trans = Transaction[I]
// Void put(Trans t)  // used by the runtime system
  T get(Fut[T] f) // for each T appearing as a method result in I
   // note: it may give error if the value is error
   // and suspend when the future is not resolved
}

interface History[I] extends Future[I] {
  type Hist = List[Trans] // type Trans is inherited
  Trans lastTrans() // return last transaction
  Hist getTrans() // return all transactions
  Hist transOf(Any o) // return all transactions involving o
  ... other functions
}
```

Figure 8.4: The interfaces of futures and history objects

for each contract object and there is no write access to it from the code, i.e., neither from the clients nor the contract object. Contracts as well as history objects are defined by object-oriented classes. By default, everybody has read access to a history object; however, this may be restricted by use of inheritance as discussed in more detail in Section 8.4.5. This gives a way to enforce privacy restrictions.

A history object can be seen through the *History* interface or through the more limited *Future* interface, providing the functionality of the future mechanism. These interfaces are given in Figure 8.4, using a type parameter *I* reflecting the interface of the associated contract. The Future interface has standard *put* and *get* methods (to store and retrieve information), but such that the *put* method is not visible to programmers. Thus when seeing a history object through the Future interface one may use it as a future. By seeing it through the History interface one may access the whole transaction history or use some predefined/user-defined functions to extract parts of it.

Specifications of interfaces may be given by means of invariants over the transaction history h, involving the history and user-defined functions. The behavior of a contract is specified by a number of *executable invariants*, defining a boolean condition over the transaction

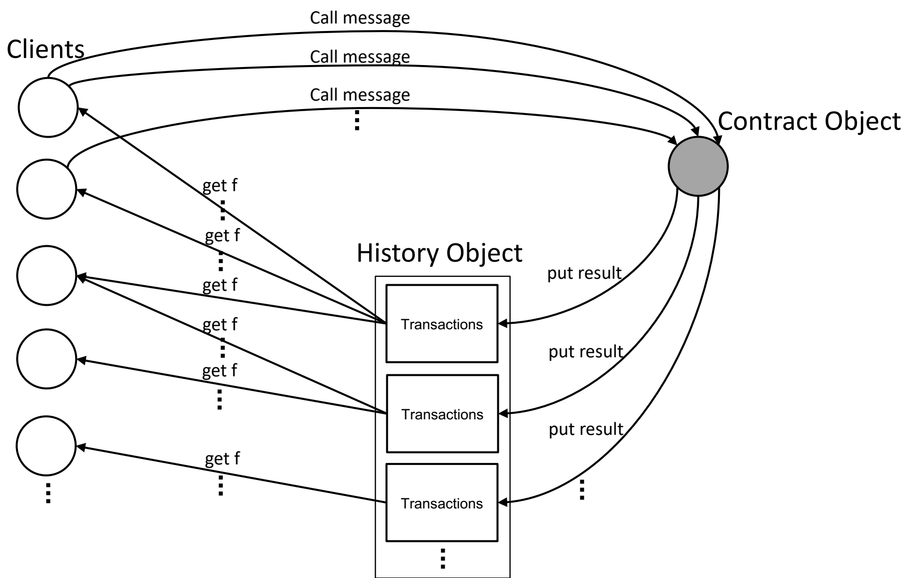

Figure 8.5: History objects

history h. This invariant will be checked at runtime. This is further explained in Section 8.5 and an example is given in Figure 8.12.

8.4.1 History Objects

For each contract object o we associate a history object, $history(o)$. The history object will keep track of all transactions involving the contract object. A contract participant may use a history object to check and verify that the interaction with the corresponding contract object is appropriate. Furthermore, a history object is write-protected. Therefore no object may manipulate the state of the history objects. From the outside, a history object is treated like a normal object, which means that any external object may communicate with a history object when desired, using simple or blocking calls. This is following the spirit of [34]. Moreover, the behavior of the history objects is given by predefined classes and interfaces, and these may be extended upon need. In particular, there is a *get* method to retrieve information, something which enables a history object to take the role as a future (see Figure 8.5). By means of inheritance, these class declarations can be extended and modified in the same way as other classes (but not by ordinary programmers). In particular, one may

add functionality by implementing new interfaces and methods, and one may add protection mechanisms in redefined *get* methods to implement security and privacy restrictions, for instance, by requiring that the caller satisfies some restrictions. This means that calls to the history objects may distinguish their behavior depending on the caller (using the implicit parameter caller and its interface).

Notice that the history objects are provided by the underlying software/network platform. As long as this underlying software platform is protected from manipulation by attackers, the history objects can be trusted since they are not accessible through the source code and therefore cannot be modified neither by attackers nor through intentional or unintentional programming "mistakes" in the contract object. Thus history objects are protected against application-level attacks. To improve trust, one can place the history object on a different physical location than the corresponding contract object, and one may use a trusted platform if possible. In case the underlying platform cannot be trusted, one can use blockchain technology to ensure trust at the underlying platform level.

8.4.2 The Transaction Type Corresponding to an Interface

In order to define the transaction history, we consider the relevant kinds of transactions/events including *invocation*s and *completion*s, representing method call messages and return from a method, respectively, as well as *get events* reflecting the transfer of a future value in connection with a get statement or blocking call. These transactions are recorded in the history object. In addition, we may consider *object generation* events. In our framework, we do not need to consider reception of invocation messages (on the callee side) since these can be derived from the history. For our language, we only need to consider *invocation*, and *completion*, and *get* transactions.

An invocation transaction corresponds to a call to a method m with actual parameter list $\overline{e}$ and is represented by a four-tuple of form:

$$call(\mathit{fid}, caller, callee, m(\overline{e}))$$

where *fid* is a unique identity generated for the call, a so-called *future* identity, $caller$ is the caller object identity, and $callee$ is the callee object.

Note that the second and third arguments indicate the direction of the message (from-to). This transaction is generated when the corresponding invocation message is sent over the network. When the call has completed normally or abnormally (i.e., resulted in an error), the completion of the method by the callee generates the transaction:

$$comp(\mathit{fid}, result)$$

where $result$ is the value resulting from the call, possibly **error**. As discussed in Section 8.4.4, this is a generalization of the future mechanism, in that all future values generated by the same object are stored in the same object.

A *get* event reflects the transmission of a future value and has the form:

$$get(\mathit{fid}, o)$$

where o is the object requesting the future value (through a synchronous call or get statement). In order to keep the transaction history small, the future value is not part of the event since it is given by the history (by the completion event for *fid*).

For any interface or class I with methods m_i (for $i \in \{1, 2, , ...\}$), we define the corresponding type $Call[I]$ by one constructor function m_i for each method m_i, and with input types as given by the parameters of method m_i, and result type as given by the return type of method m_i, see Figure 8.6. In case a method of the interface has a cointerface, we also add constructor functions for each method of the cointerface. For a class C, then $Call[C]$ includes constructor functions for all local methods, exported methods and cointerface methods. The type $Transaction[I]$ is the union of such calls and completions, adding implicit parameters. For calls the implicit parameters are the future identity, the caller and the callee. For completions the call identity suffices. Here T_k is the type of the kth parameter of method m_i, and the different constructor functions are separated by a bar $|$ (disjoint union).

For interface Auction we have that the type $Call[Auction]$ consists of:

```
open: | close: | makeBid: Nat,            for methods of Auction, plus
newBid: Nat | youwon: Nat | winner: Bidder | pay: Nat,
Bidder methods
```

```
type Call[I] = ... | m_i: .. *T_k* .. | ...  // encoding calls to methods of I/cointerf
type Comp[I] = ... | m_i | ...     // encoding the corresponding call completions
type Transaction[I] = call: Fid*Any*Any*Call[I]
                    | comp: Fid*Comp[I]
                    | get: Fid*Any
```

Figure 8.6: Predefined types for transactions

(since Bidder is a cointerface). The transactions defined for Auction objects, defined by type $Transaction[Auction]$, consist of the following call events made by $bidder$ objects (left column) or by the $auction$ object (right column):

$$
\begin{array}{ll}
call(\mathit{fid}, bidder, auction, open()), & call(\mathit{fid}, auction, bidder, newBid(n)) \\
call(\mathit{fid}, bidder, auction, close()), & call(\mathit{fid}, auction, bidder, youwon(n)) \\
call(\mathit{fid}, bidder, auction, makeBid(n)), & call(\mathit{fid}, auction, bidder, winner(b))
\end{array}
$$

The completion events have the form $comp(\mathit{fid}, void)$ and $comp(\mathit{fid}, bool)$ in this case, and the get events have the form $get(\mathit{fid}, o)$.

Notation on transactions We use dot-notation to extract the different components of a transaction. For instance, the caller of a transaction t is given by $t.caller$. Similarly, we write $t.callee$, $t.fid$, and $t.result$. This syntax is lifted to transaction lists. For instance, the set of callers in a transaction list $trans$ is given by $trans.caller$.

The projection operator "/" is defined for lists such that a list projected by a set gives the sublist containing all elements in the set. In particular, $trans/\{.\mathit{fid} = f\}$ is the sublist of transactions where the future identity is f, and $trans/\{.caller = o\}$ is the sublist of transactions where o is the caller.

Furthermore, $trans/Call$ and $trans/Comp$ give the sublists of invocation and completion transactions, respectively. The sublist of completion transactions without error is $trans/Comp/\{.result \neq error\}$. The notation $last(trans/Comp/.\mathit{fid} = f).result$ means that we take the sublist of completions in $trans$ that have a fid equal to f, and then take the value of the last transaction in the sublist. For an interface I we let the projection $trans/I$ denote the subsequence of $trans$ restricted to call and completion transactions of methods in the interface I. For instance, we can state that a subclass satisfies the invariant $OK@I$ of

an interface I by requiring $OK@I(trans/I)$, thereby considering the relevant part of the transaction history.

8.4.3 The Implementation of History Objects

Finally, we can describe how history objects are implemented. The history object of a contract of class C contains a (private) transaction list called *trans*:

```
List[Transaction[C]] trans := nil // restricted by read-only access
```

which is updated by the underlying runtime system by appending each new message from or to the contract object, see Figure 8.7.

This list variable is read-only in the class and is only visible through the defined methods. It is updated by the runtime system through a predefined *put* method:

```
Void put(Transaction[C] t) { trans :++ t } // appending t to trans
```

When a return statement is executed at runtime by a contract object, a completion message is generated. The runtime system will then add this transaction to the history object by doing $history(\text{this}).put(comp(\mathit{fid}, r))$ where r is the result value. The *put* method is not available to the programmer. Abnormal termination results in a *put* call with result *error*.

Similarly, a history object includes a public *get* method for each method result of type T:

```
T get(Fut[T] f){when <f is resolved>; return <the future value>}
```

In order to check that f is resolved, we project the transaction history taking the completions events such that *fid* is f, and then checking if this sublist is empty. If not, it will have exactly one element since all future values are unique, and we can extract the future value by $.result$.

A get statement is therefore possible in our setting as a blocking call to *get* on the appropriate history object, or as a non-blocking call to *get* using a guard on the corresponding future. Class *HISTORY* adds definitions

```
class FUTURE[Contract,ContractClass](Contract contract)
implements Future[Contract] {
// contract is a class parameter providing a reference to the contract object
// Contract is the interface of the contract object
// ContractClass is the class of the contract object

  type Trans = Transaction[Contract] // transactions of interface Contract
  type AllTrans = Transaction[ContractClass] // trans. of the Contract class
  type Hist = List[Trans] // the history seen through the Contract interface
  type FullHist = List[AllTrans] // the full history, including local events

  FullHist trans := nil // the history field, restricted by read-only access

// Void put(AllTrans t){trans :++ t}  // used by the runtime system

  T get(Fut[T] f) { // to get a future value when resolved
    when (trans/Comp/{.fid=f})≠nil; // suspending when unresolved
    return last(trans/Comp/{.fid=f}).result }
}

class HISTORY[Contract,ContractClass] implements History[Contract]
extends FUTURE[Contract,ContractClass] {
  // inherits the class parameter "Contract contract"
  Trans lastTrans() {return last(trans/Trans)} // error when no last Trans
  Hist getTrans() {return trans/Trans} // returns all visible transactions
  FullHist getAllTrans() {return trans} // returns all transactions
  Hist transOf(Any o) {return trans/{.caller=o} } // all transactions of o
  ...
}
```

Figure 8.7: A class implementation of futures and history objects

of methods for extracting the whole or parts of the history, again by means of projection, see explanations in Figure 8.7.

As mentioned, a history object is generated when a contract object is generated. For instance, a new contract created by v := **new** $C(...)$ gives (implicitly) a new history object of the largest class supporting the interface of v, with v as a parameter, which ensures that the history

object is aware of the associated contract object and class. The new class is instantiated with C, which ensures that the type of the $trans$ variable includes the relevant transactions. For instance, if v is of interface $Future[I]$ the object will be of class $\mathrm{FUTURE[I, C]}$, if v is of interface $History[I]$ (with $I \leq Contact$) the object will be of class $\mathrm{HISTORY[I, C]}$, and if v is of interface AuctionHist the object will be of class $\mathrm{AUCTIONHIST}$. This ensures that the chosen history class is required to respect the invariant of the interface of v (one could consider syntax for more fine-grained selection of classes for implicit the history objects).

We observe that the transaction history of an active object, as given by its history object, is sufficient to define the state of the object at the end of a method execution. The state of an active object at its last method completion can be reconstructed from the transaction history, and also the pre-state of method execution resulting in error. Note that suspension in the middle of a method would require more events to be recorded in the *trans* variables. We also observe that the history objects define the transaction history in a faithful way due to the language restriction on the *trans* variables by means of read-only access for programmers and increment-only write access for the underlying operating system. This restriction is also imposed on subclasses of the history classes. Thus one may rely on the transaction histories with write-once/multiple-read access guaranteed at the software level, in a way similar to smart contracts. This is checked statically, i.e., it is checked statically that the *trans* variable cannot be changed from the program. If the runtime system also offers protection of unauthorized write access to these variables by means of a trusted execution environment, one does not need blockchain technology to guarantee write-once/multiple-read access. If not, one may use blockchains to obtain full trust.

Our approach can be related to the future concept, which has become popular in the setting of active object languages and is supported by several languages [7]. Remote method calls are handled by message passing and the result of a method invocation is placed in a future object, at which time the future is said to resolved. The caller generates a reference to the future object and this reference may be passed to other objects in the case of first-class futures. Any object with a reference to the future object may ask for the value, typically via a *get* statement, which will block when the future is not yet resolved. Some languages allow

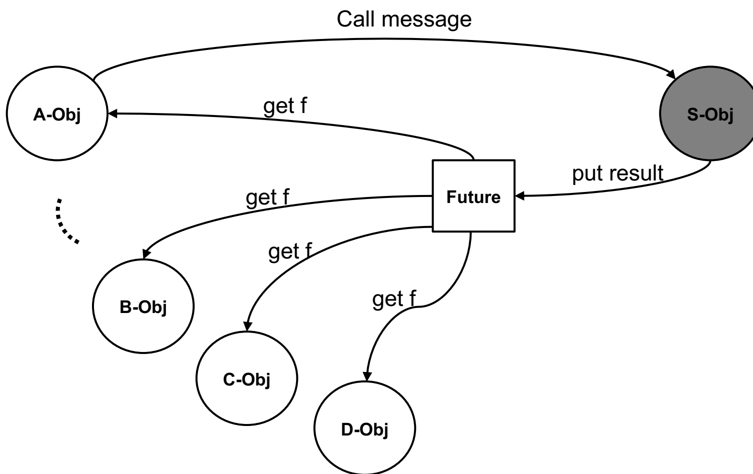

Figure 8.8: Illustration of the future mechanism

polling (i.e., testing repeatedly until a condition becomes true) to check if a future is resolved, to avoid blocking. Figure 8.8 illustrates this use of futures.

8.4.4 Histories as a Generalization of Futures

With the standard future mechanism, it is not trivial to detect when a future can be discarded; and as many futures may be generated, garbage collection is in general needed. In the active object paradigm, this is a clear disadvantage since the active objects themselves have a long life time. When local data inside objects is defined by data types, using a functional programming language to express and manipulate values of the data type (such as in the Creol and ABS languages), there is no need for general garbage collection of these values, assuming storage for values of the data types can be retrieved efficiently. In our proposed solution, the same history object will contain all future values generated by a given callee object, including those of the past as well as future ones. This history object is therefore long-lived and need not be garbage-collected. However, as the storage need is growing dynamically, the history objects could be placed in a cloud, possibly split over several storage media letting the latest part be most accessible.

Another disadvantage of the future mechanism is that the future value is unprotected, and an object getting the value may not know where the future came from and what it represents. In particular, privacy aspects are unknown and the information can easily be misused [26]. In our approach, we can add protection mechanisms in redefined *get* methods to implement security and privacy restrictions (see Section 8.4.5).

```
// Subclass with build-in security
class SECHISTORY[Contract,ContractClass]implements History[Contract]
extends HISTORY[Contract,ContractClass]{
  Bool isOk() {return not caller in blacklist(trans)
   // this requires a definition of blacklist, for example
  return not caller in (trans/{.result != error}).caller
   // blacklisting callers that make illegal calls
}

  T get(Fut[T] f) {return if this.isOk()
                  then (last(trans/Comp/{.fid=f})).result else error()}

   // similar for the other retrieval methods ...
}
```

Figure 8.9: An implementation of history objects where blacklisted bidders are blocked

8.4.5 Adding Security and Privacy Aspects

Security additions can be defined by means of redefinitions and inheritance. For instance, one may add a function (or a data structure) in a history object to include a *blacklist* (bL) of objects whose behavior have violated some property. This is shown in Figure 8.9 for the Auction example in Section 8.5. The *get* function is redefined such that blacklisted callers cannot get any information. For instance, we could blacklist callers that have made calls resulting in error (as indicated in Figure 8.9). The definition of who is blacklisted can be further redefined to accommodate more sophisticated checking.

Similarly, we can handle privacy by restricting access to information in the smart contracts to specific groups or persons. Restrictions may be

```
class AUCTIONHIST2 implements AuctionHist
  extends AUCTIONHIST{

  Nat get(Fut[Nat] f){
    if caller is Bidder // restricting callers by interface
    then return super.get(f)
    else return error() fi}

  Nat highbid(){
    if caller is Bidder // restricting callers by interface
    then return bid(red(trans))
    else return error() fi}

  Bidder highbidder(){
    if caller in bidders(trans) // restricting callers by condition
    then return bid(red(trans))
    else return error() fi}
}
```

Figure 8.10: The privacy-extended history class for auction

given by means of interfaces or semantical constraints. This is shown in the auction example in Figure 8.10 where we use an interface to restrict the callers of method get and $highbid$, and restrict the callers of method $highbidder$ to caller objects that have had the highest bid at some point in the ongoing auction. We could restrict access further by requiring that the caller is a bidder that has made successful bids. Alternatively, we could require that the caller of $highbid$ and $highbidder$ is a registered bidder, i.e., checking that $caller \in bidders(trans)$.

In Figure 8.10, the notation "$o\ is\ I$" checks if (the class of) an object o has I as an interface. This subclass allows a $Bidder$ object to check the current highest bid and who has placed it, to reflect that this information is considered private to $Bidder$ objects. Thus the subclass has the built-in protection that non-bidder objects will not get any information. Rather than defining these redefined methods by means of subclasses, one may consider dynamic class upgrades [22], allowing a class to be replaced by a new version of the class, which means that the class in question is replaced at runtime, updating both existing and future history objects of

```
class SAFEHISTORY[Contract,ContractClass]
implements History[Contract]
extends HISTORY[Contract,ContractClass] {
  Void put(Trans t) { // redefined to check contract violations
     trans:++ t;
     if not OK(trans;t) then
       if t in Call then contract!reportviolation(t.caller)
       else trans:++ comp(t.fid,error) fi fi }
  ...
  func OK: Hist → Bool
  // definition of OK
  invar OK(trans)
}
```

Figure 8.11: A history class implementation with safety check

the class.

8.4.6 Adding Safety Aspects

Our specification language gives rise to executable invariants. This may be exploited in dynamic checking to ensure that there is no contract violation. This gives an extra level of safety and trust between the contract object and its users. This is, of course, valuable if the contract class has not been formally verified against the contract specification, and even if the contract has been successfully verified, a user may not know if the code was changed after verification time.

An implementation of a safety check is shown in Figure 8.11, by defining a new version of class HISTORY. Class SAFEHISTORY checks the contract invariant ($OK(trans)$) for each new transaction being recorded, by redefining the put method so that it checks output from the contract as well as input to the contract against the contract specification. If the contract does not behave according to the specification, an additional error message is recorded in the transaction history to make the violation observable. Contract users will then observe that the output from the contract is not safe when they use the get operation.

In order to handle safety violations caused by contract users, we may let the Contract interface provide a method *reportviolation* that can be used to inform a contract about problems with a contract user, namely that the user does not

respect the contract specification. This means that if certain user requirements are not checked by the contract itself, it is possible to detect it in the history object and report the problem to the contract. However, when contracts are programmed to be used in any environment, calls to *reportviolation* could be used to reflect serious problems such as security violations.

Thus the added safety check are made by redefining the *put* method. Since the put method is not available to the programmer, we require that such redefinitions of put are done at a properly authorized level.

8.4.7 Simplified Specifications

As the *comp* and *get* events contain little information in themselves, specification of functions and invariants over the history is easier when redundant information is included in these events. For instance, to relate a return value to the input parameters of the corresponding call-event, it is convenient to have the input parameters easily available. For this reason, we define a type for extended completion events, CallComp, which is like type Call but with the final value-added as a fifth parameter, i.e., resulting in five-tuples of the form:

$$cc(fid, caller, callee, method(parameters), result)$$

and we define extended *get* events, called *read* events, which are like *get* events but with the future value-added, $read(fid, o, r)$ where r is the result of the corresponding completion event (the one with the same *fid*). For this reason, we define a general function:

$$red : List[Transaction[I]] \rightarrow List[ExtendedTransaction[I]]$$

where ExtendedTransaction$[I]$ is defined by the disjoint union of call events (Call), extended completion events (CallComp), and extended get events (read). The red function shows the redundant arguments of comp and get transactions. It converts a history (q) in terms of call-, comp- and get-transactions to a history of calls and extended completion- and get-transactions, defined as follows:

$$\begin{array}{rcl}
red(nil) & = & nil \\
red(q; call(f,o,o',m(\overline{e}))) & = & red(q); call(f, o, o', m(\overline{e})) \\
red(q; call(f,o,o',m(\overline{e})); & & \\
q'; comp(f, r)) & = & red(q; call(f,o,o',m(\overline{e})); q'); cc(f, o, o', m(\overline{e}), r) \\
red(q; comp(f, r)); & & \\
q'; get(fid, o) & = & red(q; comp(f, r); q'); read(fid, o, r)
\end{array}$$

In the verification of a class, we will deal with histories over *local events*, i.e., those generated by this object, which amounts to *call* events with this as caller,

cc events generated by this, and $read$ events received by this. These histories must refer to extended transactions, since neither the call event corresponding to a local completion event, nor the get event corresponding to a local read event, are local, and these cannot be detected from the code in the class. Each asynchronous event is local to exactly one object.

For instance, a call event is local to the caller and not to the callee. This way, verification of a class can be done locally without considering events generated by other objects (apart from synchronous ones). This is why the verification of a contract refers to its local history. With extended transactions we are able to express invariants where inputs from other objects are visible in the extended completion and extended get events (i.e., cc and read events). The local history h of a contract object o is given by $red(trans)/o$ where $trans$ is the transaction history of the contract, and the projection "$/o$" gives the local events of o. The verification system is presented in Section 8.6.

8.5 Specification of the Auction Example and Improvements

We reconsider the auction example, using the interface $\mathrm{ActionHist}$ and class $\mathrm{AUCTIONHIST}$ given in Figure 8.12. Note that the functions defined in an interface are available in any class supporting the interface, and may occur in executable code. In this example, we use simplified specifications, which make specification and verification easier. The functions and invariant can be expressed in terms of the history of extended comp-transactions ($red(trans)$), as defined above. In fact, we only need to consider extended method completions.

The *contract specification* $check(red(trans))$ expressing the main property of the Auction, says that $o = bidder(red(trans)) \wedge n = bid(red(trans))$ holds when $youwon$ is called. This ensures that the $youwon$ call is sent to the right actor, i.e., the one winning the auction according to the transaction history, and that the winning amount is the correct highest bid according to the (reduced) transaction history.

This property can be verified for a class implementation of AuctionHist as given in Figure 8.12, and can be checked independently by a Bidder agent since he/she may communicate with the history object, and get the information based on the transaction history. The verification is shown in Section 8.6. Note that:

- bid and $bidder$ are calculated from successful $makeBid$ transactions, i.e., with return value $true$.

```
interface AuctionHist extends History[Auction]{
  type Trans = Transaction[Auction] // abbreviated data type definition
  //type Hist = List[Trans]; Hist trans --  inherited
  type EHist = List[ETrans]; // history with extended comp transactions
  // Functions defined (inductively) over the transaction history
  func bid: EHist → Nat // calculates the highest bid from trans
  func bidder: EHist → Bidder // calculates the highest bidder
  func check: EHist → Bool // checks that the winner is the real winner
  var EHist q, Nat n, Bidder b
  bid(nil) = 0
  bid(q; cc(_,_,contract,makeBid(n),true)) = max(n,bid(q))
  bid(q; cc(_,_,contract,close(),true)) = 0
   // auction closed, high bid reset
  bid(q; others) = bid(q)

  bidder(nil) = null
  bidder(q; cc(_,b,contract,makeBid(n),true)) =
                                    if n>bid(q) then b else bidder(q)
  bidder(q; cc(_,_,contract,close(),true)) = null
   // auction closed, highbidder reset
  bidder(q; others) = bidder(q)

  check(nil) = true
  check(q; call(_,contract,b, youwon(n))) = (b=bidder(q)) and (n=bid(q))
  check(q; others) = check(q)
   // Finally, the contract specification:
  invar check(h)
}
```

```
class AUCTIONHIST(Auction a)
  implements AuctionHist extends HISTORY[Auction, AUCTION] {
   // inherits trans
  invar check(red(trans)) // Note that the local history h=red(trans)
        and highBid=bid(red(trans)) and highBidder=bidder(red(trans))
}
```

Figure 8.12: The auction history interface and class

- The *contract specification* expressing the main property of the Auction says that $o = bidder(red(trans)) \wedge n = bid(red(trans))$ holds when $youwon$ is called.

Below we consider improvements on the example concerning security, privacy, functionality, and trust.

8.5.1 Adding security by means of a blacklist

Let us now consider the restriction that an auction owner may not make a bid to his/her own auction (instead, we could give the owner a chance to set a minimal value as a parameter to $open$). An auction owner is then blacklisted if he/she makes a bid on his/her own auction. This could be specified by:

```
func owner: EHist → List[Bidders]
func bL: EHist → List[Bidders]

owner(nil) = null
owner(q; cc(_, b, _, open(), _) = b
owner(q; cc(_, b, _, close(), _) = null
owner(q; others) = owner(q)

bL(nil) = nil
bL(q; cc(_, b, _, bid(n), _)) = if b=owner(q) then bL(q);b else bL(q)
bL(q; others) = bL(q)
```

Another restriction could be related to correct payment. We introduce a method for payment from the winner of an auction to the auction owner. In this case, the winner needs to be informed about who is the owner, so we use the following Bidder interface:

```
interface BidderPay {
 with Auction
  Void youwon(Bidder owner, Nat bid) // inform the winner about owner and bid
  ...
 with Bidder
  Void pay(Nat x) // the winner pays to the auction owner
}
```

We may then add a specification of a blacklist (bL):

```
func bL: EHist → List[Bidders]

bL(nil) = nil
bL(q; t) = if q/b ew cc(_,_,b,youwon(o,n),_) and t in cc(_, b, o, pay(n), _)
           then bL(q) else (bL(q);b)
```

A bidder is blacklisted after completing a $youwon(owner, amount)$ call, if he makes another Auction call than a pay call to the owner with the given amount. The different definitions of blacklist could well be combined.

8.5.2 Functionality improvements: Method open

A weakness with the simple version of the Auction example showed is that a Bidder agent calling $open$ does not know if he/she succeeds in opening an auction. To solve this, one may use a **require**-clause as in the Solidity language to ensure that a condition holds at the start of the method body. In this case we want to ensure that:

- $isopen = false$ at the beginning of method $open$.
- $caller = owner$ at the beginning of method $close$.

However this may lead to transaction failure, and the issue of recreating an appropriate state becomes essential.

Another solution is to let the $open$ return a Boolean method result (as the $close$ method) expressing whether the open was successful. The methods are then modified as follows:

- A variable $Bool\ result := (isopen = false)$ is created at the beginning of the $open$ method.
- The method ends with **return** $result$.

This solution has the advantage over the approach with **require** that no runtime error is created and the method has no effect when they are not successful. Thus no roll-back of the state is needed.

8.5.3 Adding trust

It may be useful to inform bidders about the correctness of the information from the contract provider, directly from the associated history object, thereby providing reliable information from a third party. In the Auction example, we

```
class AUCTIONTRUST extends AUCTIONHIST {
 {this!inform()} // the class constructor starts a call to inform

 Void inform() { // a local method called recursively
   when trans ew call(_, contract,o,youwon(n));
   bidder(red(trans))!youwon(bid(red(trans))); // true information to winner
   this!inform() }
// the information sent to winner is calculated by the history object.
}
```

Figure 8.13: A trust-extended history class for auction

could let a winner know about the correct high value when he/she wins and also that he/she is the correct winner. To achieve this, we use inheritance and let the subclass objects have an active behavior providing this information flow to the winner by means of a background activity using an initial guard. We show a subclass providing this background activity in Figure 8.13.

Note that here the $youwon$ call by the history object is added to the transaction list and therefore the guard is not satisfied in the next execution of *inform* since $trans$ will end with `call(_,this,o,youwon(n))`; thus the *inform* message is not repeated. We could also add a test:

```
if bid(red(trans))=n and bidder(red(trans))=o then <send ok message>
else bidder(red(trans))!youwon(bid(red(trans))) fi
```

and let the bidder know the correct values only in the else branch where the information from the Auction provider is incorrect. This means that as long as a bidder is not informed from the history object he/she can trust the information from the Auction provider. A similar treatment can be done for all other output to Bidders from the Auction provider.

8.6 Verification

In this section we discuss how to verify contract specifications by a Hoare-style logic for partial correctness. We show how to verify a class invariant in a class-wise manner and demonstrate the verification technique on the auction example. A class invariant R may talk about fields, class parameters, and its local history h, i.e., the history of extended events ($call$, cc, and $read$ events) generated by that object. We have that the local history of o is the same as the extended history

of the contract restricted to the events generated by o, i.e., $h = red(trans)/o$. In general, one may need to consider creation events (of new objects), which we omit here for simplicity. We need not consider events reflecting start of execution of a method in the class since these can be derived from the history. The reasoning system here is simpler than in [11, 29, 30], since we may consider a smaller event set for method interaction due to our notion of initial guards. Due to the presence of futures, we have a more expressive language than in [10]. Reasoning about an object taking part in a smart contract can only verify the role of that object. To verify the whole contract, as stated in the history object, we need in general to combine the invariants of the objects taking part in the contract, using a rule for composition.

In order to verify a subsystem of several concurrent objects such as a smart contract, we use the composition rule of [29, 30]. Adapted to our case, the composition rule generates an invariant $Inv(trans)$ of the contract (where $trans$ is the history of all events seen by the history object) by forming the conjunction of all the local invariants $R\,^{h,\ \mathsf{this}}_{red(trans)/o,\ o}$ of each object o in the subsystem together with a *wellformedness predicate* stating that each call transaction has a unique future identity, and that a call comes before the corresponding completion transaction (the one with the same future identity), and that a result read (in a $read$ event) is the same as the generated result (in the preceding cc event with the same future identity). The notation R^v_e denotes the substitution of (free) occurrences of v in Q by e, and $R^{\overline{v}}_{\overline{e}}$ simultaneous substitutions. The replacement of this by o is needed to formalize the fact that what is called this in the class invariant is the object o in the contract.

To verify that R is an invariant of a given class C, we must prove that the invariant is satisfied initially, i.e., that R holds for an empty history and initial field values, and that each method of the class maintains R. The verification of a method inside a class is done by sequential Hoare-style reasoning. The Hoare triple $[P]\ s\ [Q]$ expresses that if P holds in the pre-state of s then Q holds in the post-state, provided s terminates normally. If P implies Q, then Q is said to be an invariant of s.

For a method m with parameters $\overline{x}$ and body **when** guard; s; **return** e, we need to verify:

$$[R \wedge guard] \quad \mathsf{s;\ h\!:\!++}\ cc(\mathit{fid}, \mathsf{caller}, \mathsf{this}, m(\overline{x}), e) \quad [R]$$

For simplicity we assume that method parameters are read-only. This reduces to:

$$[R \wedge guard] \quad \mathsf{s} \quad [R\,^{h}_{h;cc(\mathit{fid},\mathsf{caller},\mathsf{this},m(\overline{x}),e)}]$$

This triple can be verified locally in the class by ordinary sequential Hoare logic, and the special syntax $h :++\ t$ is semantically the same as the assignment

$h := (h; t)$, and the axiom for assignment is given by $[Q^v_e]$ v:=e $[Q]$. This axiom holds since there is no remote field access (and therefore no semantical aliasing problems) and since expressions are pure.

Furthermore, an asynchronous call $o!m(\overline{e})$ in the body is treated as the two assignments $f :=$ **new** $Fut;\ h :++\ call(f, this, o, m(\overline{e}))$, where the first assignment represents a non-deterministic assignment to f resulting in a fresh future identity (like an object creation statement). The Hoare rule for this assignment is $[\forall f\,.\, f \notin h \Rightarrow Q]$ f := **new** Fut $[Q]$ where the universal quantifier corresponds to non-determinism, and the condition ensures freshness of f, i.e., that f has not already occurred in a transaction in h. Thus we derive the following rule for asynchronous calls:

$$[\forall f\,.\, f \notin h \Rightarrow Q^h_{(h;call(f,this,o,m(\overline{e})))}]\ \ \text{o!m}(\overline{e})\ \ [Q]$$

The rule for the get statement is given by:

$$[\forall v\,.\, Q^{\,h}_{\,h;read(f,this,v})]\ \ \text{v:=}\textbf{get}\text{ f}\ \ [Q]$$

where the universal quantifier on v' corresponds to a non-deterministic assignment to v, which reflects that the read value is locally unknown. In the compositional rule, this value is resolved through wellformedness of the contract history (using the corresponding completion event and the wellformedness predicate).

Combining these two rules, we obtain a similar rule for synchronous calls:

$$[\forall f, v'\,.\, f \notin h \Rightarrow Q^{\,v,\ h}_{\,v',(h;call(f,this,o,m(\overline{e}));cc(f,this,o,m(\overline{e}),v'))}]\ \ \text{v:=o.m}(\overline{e})\ \ [Q]$$

since a synchronous call $v := o.m(\overline{e})$ is semantically the same as the statement sequence $f :=$ **new** $Fut;\ h :++\ call(f, this, o, m(\overline{e})); v :=$ **get**f. The prime on v' is needed in case v occurs in $\overline{e}$ (which is the old v and should not be quantified).

In the setting of partial correctness, $[P]$ s $[Q]$ assumes normal termination of s (and no errors). Thus it does not ensure error-free execution. However, reasoning about errors is needed in the presence of error handlers, because a handler may turn an error into a normal value and then the assumption of normal termination is satisfied. Reasoning about error handlers can be done as indicated below for handlers on assignments and calls.

One may treat an assignment with a handler, $v := e < s >$, as the assignment $v := e$ if e does not result in an error, otherwise $< s >$. To distinguish the two cases, one may use a a predicate WD expressing welldefinedness, letting $WD(e)$ be true if the evaluation of e results in a normal

value and false if it results in an error. For instance $WD(last(q))$ is $(q = nil)$, which can be obtained from the definition of $last$ in Figure 8.2. The rule is then:

$$\frac{[P]\ s\ [Q]}{[\textbf{if } WD(e) \textbf{ then } Q_{e}^{v} \textbf{ else } P]\ v := e < s >\ [Q]}$$

Reasoning about a synchronous call with a handler, $v := o.m(\overline{e}) < s >$, can be done by the rule:

$$\frac{[P]\ s\ [Q]}{[\forall f, v' \,.\, f \notin h \Rightarrow Q' \vee P']\ v := o.m(\overline{e}) < s >\ [Q]}$$

where Q' is Q with the substitutions given in the rule for synchronous calls, and P' is $P^{\,h}_{h;call(f,this,o,m(\overline{e}));cc(f,this,o,m(\overline{e}),error)}$. The two extensions of the history in the precondition are disjoint since quantified variables range over defined values. A key point here is that the handler can be connected to a particular transaction in the history.

One could consider other forms of error handlers (for instance at the end of a method body) in a similar manner, and reasoning about raising exceptions can be done as usual. We do not discuss reasoning about roll-backs, which can be non-trivial, especially if it is not clear at verification time how far the roll-back should go.

8.6.1 Verification of the Example

We show how to verify that the implementation of the auction system given in Figure 8.12 satisfies the invariant (check) given by interface AuctionHist. For our example, it suffices to consider extended completion events since only these are used in the contract specification in AuctionHist.

We prove that class AUCTIONHIST satisfies its invariant R, namely:

$$check(h) \wedge highBid = bid(h) \wedge highBidder = bidder(h)$$

where $check$ is defined in interface AuctionHist. The invariant must hold initially and be maintained by each method visible to the environment. Initially the history is empty, $highBid$ is 0, and $highBidder$ is $null$ (the initial value of object references). Thus we need to prove $check(nil) \wedge 0 = bidder(nil) \wedge null = bidder(nil)$, which is trivial.

Method $highest()$ does not change any fields and the completion of $highest$ has no effect on the invariant, so verification is also trivial (since R implies R).

For method *open* we need to verify:

$$[R] \quad \textsf{\textbf{if} not isopen \textbf{then} owner:=caller; isopen:=true \textbf{fi}} \quad [R^{\,h}_{h;t}]$$

where t is $cc(\mathit{fid}, \textsf{caller}, \textsf{this}, open(), void)$. This gives the following two verification conditions:

$$R \wedge isopen \Rightarrow R^{\,h}_{h;t}$$

$$R \wedge \ \textbf{not}\ isopen \Rightarrow R^{\,h,owner,isopen}_{(h;t),caller,true}$$

with t as above. Both are verified without problems.

For method $makeBid$ we need to verify the following two triples:

$$[R \wedge not(open \wedge x > highBid)] \quad \textsf{bidders :++ caller} \quad [R^{\,h}_{h;t}]$$

where t is $cc(\mathit{fid}, \textsf{caller}, \textsf{this}, makeBid(x), \mathit{false})$, and:

$$[R \wedge open \wedge x{>}highBid] \ \textsf{highBid:=x; bidders!newBid(x); highBidder:=caller}[R^{\,h}_{h;t}]$$

where t is $cc(\mathit{fid}, \textsf{caller}, \textsf{this}, makeBid(x), true)$. We here ignore the assignment bidders:++caller since the variable bidders does not occur in the invariant. The first triple is trivial since $R^{\,h}_{h;t}$ reduces to R in this case. The second verification condition reduces to:

$$R \wedge open \wedge x{>}highBid \Rightarrow R^{\,h,highBidder,highBid}_{(h;t;t'),caller,x}$$

by sequential Hoare analysis, replacing the asynchronous call by h:++ t′ where t′ is the call event reflecting the newBid call, which does not influence the invariant. This gives the verification condition:

$$check(h) \wedge x = bid(h;t) \wedge caller = bidder(h;t)$$

provided $check(h) \wedge highBid = bid(h) \wedge highBidder = bidder(h) \wedge open \wedge x{>}highBid$, which reduces to true by the function definitions of bid, $bidder$, and $check$.

Finally, we consider method $close$: ignoring assignments and calls not affecting the verification, the most challenging branch reduces to the triple:

$[R \wedge caller{=}owner \wedge highBid{>}0]$
highBidder!youwon(highBid); highBidder:=null $[R^{\,h}_{h;t'}]$

which gives the verification condition:

$$R \wedge caller{=}owner \wedge highBid{>}0 \Rightarrow (\forall f \,.\, f \notin h \Rightarrow \forall f \,.\, R^{\,h,\ highBidder}_{(h;t';t),\ null})$$

where t is $call(f, this, highBidder, youwon(highBid))$ reflecting the $youwon$ call transaction, and t' is $cc(fid, caller, this, close(), true)$. This verification condition follows from the definitions of the invariant and the $check$, bid, and $bidder$ functions. This completes the invariance of the invariant.

By the rule for composition we may conclude $check(red(trans))$, which ensures the contract specification of the history object. The fact that the contract class invariant alone ensures the contract specification reflects that the contract specification does not put any restrictions on the contract users. For the version of the contract with payment form the highest bidder to the auction owner, a specification of the history object would involve requirements to the highest bidder as well as the contract object. In this case, one would need to use compositional reasoning of the contract object and the bidders.

8.7 Evaluation

8.7.1 Difference between our language and Solidity

In general, our language is more high-level and abstract than Solidity [35]. Our language is oriented towards a compositional semantics and class-wise reasoning. In contrast to our approach, Solidity does not support reasoning about contract specifications by a logic or verification system. Thus the focus of the languages is different, nevertheless we may try to compare the expressiveness.

The communication mechanisms are similar, but are following more closely the object-oriented style in our case. We allow one-way and two-way message passing, as well as broadcasting and first-class futures. The Solidity notion of $msg.sender$ corresponds to $caller$ in our language.

Every contract in Solidity can include declarations of *State Variables* that contain persistent data, similar to class fields in our language, *Functions* that can modify variables, like methods in our language, *Function Modifiers*, comparable to guards in our language, *Events*, similar to transactions in our setting, *Struct Types* (record type) and Enum Types, which can be seen as special cases or implementations of user-defined data types in our language. Besides, contracts support encapsulation (visibility attributes) and may inherit from other contracts – in our setting this follows from our use of interface abstraction and inheritance.

Through *fallback functions*, custom handling of messages that do not specify a concrete function to call is supported in Solidity. Values are returned from functions by use of return statements/variables. In our setting, there are no fallback functions since all call messages will be understood due to static typing as explained below. The queue order in our language is defined by taking the

oldest enabled call first (priority calls could be added). The use of guards allows calls to be delayed and thereby control the scheduling of calls.

Strong typing. Like Solidity, our language supports strong typing of variables and expressions. Calls are strongly typed, and for a call $o.m(\overline{e})$ it is checked that the type of o is an interface that supports a method m such that the actual parameters respect the parameter types of that method, and one can guarantee that the caller o cannot be nulll by static over-approximation. Due to the cointerface mechanism, the caller has a type, and thus even call-backs of form $caller.m(\overline{e})$ can be strongly typed [23], in contrast to Solidity. The primitives *call* and *delegated* in Solidity give rise to untyped calls.

Visibility of attributes. Solidity uses a number of primitives, including *private*, *public*, *internal*, and *external* to declare visibility of methods/functions from the outside, and even state variables marked private can be visible. Our language is based on interface abstractions, i.e., we use interfaces to define visible parts of a class. No field is visible, and only methods declared in an interface are visible. It is statically checked that a class defines (either explicitly or implicitly through inheritance) each method declared in an interface implemented by the class. Thus "method not understood" errors are not possible, as explained above, which explains why fallback functions are not needed in our setting.

Reentrance. Reentrancy happens when an object is interrupted during execution, and can be called again before its previous invocations complete execution. Reetrance is a well-known source of undesired behaviour in Solidity programs. Reetrance is also possible in our setting, but only in invariant states and when the guard is satisfied. The guard makes it possible to delay calls that would break the invariant, and thereby ensure desired ordering restrictions on the transactions. This mechanism is valuable in avoiding undesired behaviour.

Inheritance. Multiple inheritance and polymorphism are supported in Solidity. This is controlled by means of the keywords virtual and override. Using the syntax *ContractName.functionName()*, and *super.functionName()*, one can call functions further higher up in the hierarchy, and one level up in the flattened inheritance hierarchy, respectively. Our language supports multiple inheritance as well and has a similar way of selecting methods from particular superclasses. For simplicity, we do not discuss overloading here, but in our setting of strong typing, overloading with respect to both method parameters and method result could be done as in [22].

Function modifiers. By using modifiers in Solidity, the behaviour of functions can be modified in a declarative way. For instance, modifiers can automatically check a condition prior to executing the function. Function modifiers execute before entering the actual function body and cause abnormal

termination and roll-back of the state. In our approach, the guards are also checked at the start of a method execution, but do not cause abnormal termination, instead, they may cause the execution of the current method to be delayed until the guard is satisfied. For instance, if the guard is a future check $f?$, this gives a non-blocking way of waiting for a future value. Reasoning about guards is straight forward, as discussed in section 8.6, whereas reasoning about roll-backs is non-trivial since it depends on dynamic information about which calls have been made. Our setting still supports runtime roll-backs since the state of a contract at the point of a specific call in the past can be calculated from the transaction history in the history object. For simplicity, the price/cost aspect of transactions are not considered in our approach, since we consider a wider concept of contracts.

Predefined data types and struct definitions. Solidity has a large number of predefined numeric types. We have a small number of predefined types and can mimic other types by inductive data type definitions. Struct definitions are composed of a list of pairs of type names and field names. These can be defined in our setting by data type definitions using disjoint union.

Expressions. Function calls in Solidity can be of several types: internal, external, delegate, and calls to certain builtin functions. Internal function calls are simply jumps in the code of the current account. External calls cause a message to be sent over the Ethereum network, executing code on another account. Delegate calls exist to provide a functionality akin to shared libraries, by allowing code from another account to directly operate on the storage of the calling account. The semantics of external and delegate calls are notorious sources of bugs in contracts [3], notably the DAO [8] and Parity multi-sig contract [31] incidents. In our setting, local method and function calls execute on the local object and its store, while all external (and delegated calls if handled as in [29]) calls execute on other objects. Our language supports a modular semantics, and the mentioned bugs do not apply. As mentioned, we may add a reasoning-friendly delegation mechanism [29].

Error types in Solidity. The main categories of errors in Solidity are runtime errors and logic errors, apart from syntax errors, which are detected at compile time. Runtime errors are more difficult to troubleshot than syntax errors (like in many other languages including our language) because we do not have the help of the Solidity compiler to tell us when this happens and usually they are more serious because at this stage we have already deployed the smart contract to the blockchain and we cannot just change the code or update it to fix the problem.

A runtime error happens once the smart contract is deployed to the Ethereum blockchain and the Solidity code has been compiled to a bytecode that can be

understood by the Ethereum virtual machine. A runtime error happens when the Ethereum virtual machine detects something wrong with the smart contract or is making transactions against the logic of the code. All the state changes that are caused by a transaction resulting in error are cancelled, and the transaction is reverted, and depending on the kind of error, all the gas of the transaction is consumed or some of it is refunded. Common runtime errors are: out-of-gas error, revert error, invalid opcode error, invalid jump error, stack overflow, and stack underflow. An out-of-gas error happens when there is insufficient gas to complete a transaction. If we try to execute a transaction that is not according to the logic of the smart contract, then EVM returns an error called *revert error* and the transaction will be reverted. Invalid opcode happens when we try to execute a code that does not exist. Invalid jump occurs when we try to execute a function that does not exist, for instance, if we try to call a function in another smart contract at an address that does not exist. It can also happen if we use assembly in Solidity and we point to a wrong location in the code. Stack overflow happens when there is a function that calls itself recursively and there is no stopping condition that is triggered early enough. In Solidity, the stack can be at most 1024 frame deep, which means a function can only call itself 1024 times. This is an error that is not specific to Solidity and may also happen in many other programming languages. Stack underflow happens when we try to read an item on an empty stack.

The mentioned runtime errors do not occur in our language, but errors may result from execution of partial functions and methods raising errors. This is due to a stronger type system and a simpler and more abstract semantics. For instance, the runtime stack is considered unbounded. But there may be calls made to object expressions that are null, which again may result in non-terminating **get** statements. This issue can be avoided by static type checking, over-approximating the type of non-null variables and requiring that any callee and actual object parameters are of non-null types. Object generation, this, and caller can be assumed to be of non-null types.

And the last kind of error is logic errors. A logic error happens once the smart contract is deployed to the blockchain, reflecting a problem in the logic of the smart contract. It is important to note that it is not like a runtime error in that the Ethereum virtual machine does not consider it to be an error from its perspective; from its perspective, everything is working fine and the code can be run easily. Logic errors happen when the developer has made a mistake, or there are open loopholes in a smart contract that can be exploited by attackers. Basically, this kind of error makes the smart contract either dangerous or makes it produce false result. These kinds of bugs are the most serious and also the hardest to fix because there are no tools that can automatically run through the

smart contract and tell us whether or not there are some logic errors. Contrary to syntax or runtime errors where we can use respectively the Solidity compiler and the Ethereum virtual machine. Formal verifications can be useful in analysis of logic errors in smart contracts. An example of a logic error is the famous 2016 reentrancy attack of the Distributed Autonomous Organizations (DAO) smart contract, where a code that connects a set of smart contracts together and functions as a governance mechanism, was a result of a logic error. A reentrancy attack can happen when we make an external call to another untrusted contract before it resolves an error. If the untrusted contract is malicious, it can take over the control flow of the original smart contract's code; for instance it may repeatedly withdraw Ether from the smart contract. In contrast, our approach has a modular semantics, which makes it harder to attack a contract provider, and it gives support of behavioral specification by means of contract invariants that can be checked at runtime. Furthermore, our approach supports class-wise verification method, which can guarantee absence of certain logical errors at the cost of interactive theorem proving.

8.7.2 Difference between our framework and blockchain

Blockchains have the advantage of being transparent, immutable, and corruption-free. Our history objects offer similar advantages through the programming language level:

- There is read access to the history objects, possibly limited by privacy restriction.
- The history objects are immutable from the programmer's perspective and only incremental updates are allowed at the operating system level.
- The history object specifications guarantee corruption-freedom, as these can be available to the involved partners, and may, therefore, be rechecked when desired.

These advantages are obtained without use of blockchains. But as mentioned, blockchain implementations may be used when underlying middleware or hardware is not reliable, or when the application is financially critical.

The consensus mechanism provided by blockchain is handled differently in our setting. For each contract, the history object is immutable and defines the transactions in an object manner since it is outside the contract provider control.

In contrast to the blockchain framework, our framework gives support of security, privacy, and safety:

- Security measures such as a blacklist can be added by specifying how to blacklist objects in the history objects. Several specifications of blacklists have been shown in Sections 8.4 and 8.5.

- Privacy can be added by restricting the access to information in the transaction history.

- Contract specifications can be specified for history objects and checked at runtime by these history objects.

This support of security, privacy, and safety is built into the history objects and is therefore useful in a setting where the contract objects and/or its users could be malicious.

8.8 Related Work

There are two different approaches for verifying smart contracts: dynamic analysis/runtime verification, and static analysis/compile-time techniques. Runtime verification deals with various techniques, including those in [12, 14, 15, 20, 33, 37] for checking whether a running system satisfies or violates certain correctness properties, whereas compile-time techniques analyse programs before runtime, either by a full automatic technique (such as a type and effect system) or by a semi-automatic technique (such as a verification system). In general, compile-time techniques have the advantage that they provide a guarantee since they check or deduce program properties before runtime. The disadvantage is that fully automatic techniques often involve over-approximation, and formal verification may require non-trivial human assistance. In general, runtime verification introduces additional runtime overhead. In smart contract platforms like Ethereum, these overheads not only cost time, but also money since they cause additional gas consumption, therefore the cost of smart contract execution increases. As mentioned, our approach with runtime checking of contract invariants by the added history objects, will not slow down the contract providers (unless they run on the same platform).

A static analysis is said to be *syntactic* if it is only concerned with the grammatical structure of a program, and *semantic* if it involves the meaning of grammatically correct programs. Thus semantic analysis is, in general, more effective in recognizing potential problems in a program. There are many syntactic analysis tools for smart contracts as well as smart contracts

written in Solidity (e.g., Solcheck [1], Solhint [2], Solint [3], Solium [4]). Different approaches have been presented that try to identify and fix different types of security vulnerabilities and design patterns [28] in Ethereum smart contracts using semantic analysis techniques. These are summarized below.

Luu et al. [27] provided a symbolic execution static analysis tool called OYENTE that analyses Ethereum smart contracts in order to detect bugs. Their tool works directly with Ethereum virtual machine (EVM) bytecode without access to the high-level representation (like Solidity). Also, in [13], a heuristic indicator of control flow immutability has been defined by Fröwis and Böhme for the sake of qualifying loophole risks resulting from modified control flow. They applied this to a number of smart contracts on Ethereum, and found that two out of five smart contracts are not trustworthy.

Mavridou and Laszka [28] introduced a formal semantic model called *FSolidM* for creating secure smart contracts, that is based on Finite State Machines (FSM). They provided a graphical editor to help simplifying the design of smart contracts in FSolidM. They translate FSMs into Solidity code to support Ethereum smart contracts. They also provided extensions and design patterns to improve the security and functionality of contracts. These extensions and patterns are implemented as plugins that developers can add to a contract automatically.

Many of these semantic approaches are at the bytecode level; therefore they allow the verification of compiled contracts. Grishchenko et al. [16], for instance, developed a translation from Solidity to the functional language F*, for which verification tools exist. They have also presented a formalization of a small-step semantics for EVM bytecode in F*. Thus they can compare the F* code resulting directly from Solidity with that resulting via EVM. KEVM has been presented by Hildenbrandt et al. [18]. KEVM is an executable formal specification of the EVM's bytecode and stack-based language within $\mathbb{K}$ Framework. $\mathbb{K}$ Framework is a framework based on rewriting logic to define executable semantic specifications of programming languages. Hirai [19] defined a formalization of EVM and used it to prove safety properties of some smart contracts in the interactive theorem prover Isabelle/HOL. Bhargavan et al. [6] presented a framework to analyze and formally verify Ethereum smart contracts using F*. Their smart contract verification combines two approaches: They start from the translation of Solidity source code into F*, and then using decompilation techniques, they go from EVM bytecode into F* code. An automatic verifier

[1] See https://github.com/federicobond/solcheck
[2] See https://github.com/protofire/solhint
[3] See https://github.com/SilentCicero/solint
[4] See https://github.com/duaraghav8/Ethlint

to prove temporal safety properties of Ethereum smart contracts called VERX has been presented by Permenev et al. [32]. VERX is based on reduction of temporal property verification to reachability checking, a symbolic execution engine for a fragment of EVM, and a form of predicate abstraction.

Bai et al. [4] propose formal modeling and verification of smart contracts using the SPIN model checking tool. Abdellatif et al. [1] use the BIP (Behavior Interaction Priorities) framework to model smart contracts implementation and verify the correctness. This framework uses timed automata to implement the contract functions. In order to deal with external attacks, they have also modeled the blockchain execution environment. They use the Statistical Model Checking (SMC).

Zakrzewski looks at a formalization of the Solidity language [40]. He points out the Solidity language has no formal semantics and questions the appropriateness of several of the language constructs, from a verification perspective. In particular, he claims that the modifiers are not complicating verification. The paper ends with a proposal for a big-step operation semantics for parts of Solidity, proposing a clarification/revision of unclear semantical issues.

Our work does not follow the approach of translating from Solidity or EVM to a language or formalism with verification support. Instead, we provide a more abstract language for smart contracts with support of simple verification, as encouraged in [2, 40]. The programming and specification language, its semantics, and verification system are oriented towards simple verification, and we are able to formulate a system for sequential style reasoning in a class-wise manner. We consider safety rather than temporal properties and do not consider time nor probabilistic methods. Our framework integrates trust at the language level through the notion of history objects. Moreover, the history objects can be used to provide security and privacy, as well as runtime checking of specified safety properties.

Compared to earlier work on Creol/ABS [11, 21, 25], our language is novel in the combination of initial guards for methods and first-class futures. This combination gives an expressiveness that allows ordering control with respect to method scheduling, thereby extending the expressiveness of [29]. With respect to reasoning, it gives a simpler event structure than in [11], and as a result, the reasoning system is simpler than the Creol/ABS reasoning systems for programs allowing suspension in the middle of a method.

8.9 Conclusion

We have presented a new approach to lightweight smart contracts where transparency, imuatbility, and protection against corruption are guaranteed at the source code level, by adding a special kind of protected objects called *history objects* that record all the calls and future values generated for each contract. Because of these recorded transactions, a history object can be seen as a ledger, but local to a given contract. The history objects are protected by predefined interfaces and provide read-only access from the programmer side, and increment-only access by the underlying system. They are separated from the contract provider, and can be used by contract users to check the validity of the transactions made, in a way similar to blockchain. Therefore, trust is provided at the software level even without blockchain technology, and our approach supports the main advantages of smart contracts in that it gives trust and transparency without centralized control. The approach protects against tampering and fraud, each history object is immutable and corruption-proof, and each user can observe the contract behavior through the history object to ensure validity.

Moreover, we give a theory for formal specification and verification of smart contracts, something which Solidity lacks. In particular, our approach supports class-wise verification, which is essential in open distributed systems where the contracts interact in an unknown environment. The verification is based on sequential reasoning augmented with effects on the transaction history. These advantages have been achieved by defining a version of a high-level language based on the active object paradigm and interface abstraction. We support multiple inheritance and allow subclasses and redefinitions without behavioral restrictions from the superclasses. The language has an expressiveness that can be compared to Solidity, but is more high-level, with more abstract communications mechanisms and a more abstract data type language. Method guards can be compared to the require mechanism in Solidity, but give control over the ordering of operations rather than forcing errors. Guards are useful for avoiding undesired contract behavior and avoiding attacks by an adversary. A major difference is that our language has a modular semantics and comes with a verification theory. A contract class can be verified with respect to an external invariant specified in the history object.

Furthermore, we have shown how formal specifications can be checked automatically by the history objects at runtime, thereby protecting users of a faulty contract provider, as well as protecting the contract provider from illegal users. In addition, we have shown how to achieve security and privacy, something which has been seen as a weakness of traditional smart contracts

based on blockchain. We have illustrated the approach on a typical smart contract example, namely an auction system. We have verified the contract implementation and shown various improvements with respect to added safety, security, and privacy.

The approach may be combined with the notion of dynamic and concurrent object groups [24]. This allows a contract provider to appear as a single object to the outside while internally consisting of a number of cooperating concurrent objects. From the environment, an object group is treated like a normal object, with the benefit that it can serve many contract users in parallel.

Our framework allows runtime roll-backs, since the transaction history gives sufficient information to rerun a contract provider and stop at the last state before the execution of method that results in error. We have not considered assets like gas and ether. At compile time, one cannot in general ensure sufficient assets, but one can consider the possibility of errors due to insufficient assets. We have considered reasoning about simple error recovery. In future work, we may add further mechanisms for error and exception handling and consider efficient implementation of history objects and roll-backs. In the current state, our framework can be used for modeling, prototyping, analysis, verification, and model checking of smart contracts.

Acknowledgment

This work was supported by the project IoTSec - Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the project SCOTT (www.scottproject.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422.

Bibliography

[1] Abdellatif, T. and Brousmiche, K.-L. “Formal verification of smart contracts based on users and blockchain behaviors models”. In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE. 2018, pp. 1–5.

[2] Ahrendt, W., Pace, G. J., and Schneider, G. “Smart contracts: A killer application for deductive source code verification”. In: *Principled Software Development*. Springer, 2018, pp. 1–18.

[3] Atzei, N., Bartoletti, M., and Cimoli, T. “A survey of attacks on ethereum smart contracts (sok)”. In: *International conference on principles of security and trust*. Springer. 2017, pp. 164–186.

[4] Bai, X. et al. “Formal modeling and verification of smart contracts”. In: *Proceedings of the 2018 7th International Conference on Software and Computer Applications*. ACM. 2018, pp. 322–326.

[5] Baker, H. C. and Hewitt, C. “The Incremental Garbage Collection of Processes”. In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1977, 55?59.

[6] Bhargavan, K. et al. “Formal verification of smart contracts: Short paper”. In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM. 2016, pp. 91–96.

[7] Boer, F. D. et al. “A Survey of Active Object Languages”. In: *ACM Comput. Surv.* vol. 50, no. 5 (Oct. 2017), pp. 1–39.

[8] Buterin, V. “Critical update re: DAO vulnerability”. In: *Ethereum Blog, June* (2016).

[9] Buterin, V. et al. “Ethereum white paper”. In: *GitHub repository* (2013), pp. 22–23.

[10] Dahl, O.-J. and Owe, O. “Formal Methods and the RM-ODP”. In: *NWPT’98: Nordic Workshop on Programming Theory, Turku, Finland 1998*. Available as Research Report 261, Dept. of Informatics, Univ. of Oslo (18 pages). 1998.

[11] Din, C. and Owe, O. “Compositional reasoning about active objects with shared futures”. In: *Formal Aspects of Computing* vol. 27 (2015), pp. 551–572.

[12] Ellul, J. and Pace, G. J. “Runtime Verification of Ethereum Smart Contracts”. In: *2018 14th European Dependable Computing Conference (EDCC)*. Sept. 2018, pp. 158–163.

[13] Fröwis, M. and Böhme, R. “In code we trust?” In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2017, pp. 357–372.

[14] García-Bañuelos, L. et al. “Optimized Execution of Business Processes on Blockchain”. In: *Business Process Management*. Ed. by Carmona, J., Engels, G., and Kumar, A. Cham: Springer International Publishing, 2017, pp. 130–146.

[15] Governatori, G. et al. “On legal contracts, imperative and declarative smart contracts, and blockchain systems”. In: *Artificial Intelligence and Law* vol. 26, no. 4 (Dec. 2018), pp. 377–409.

[16] Grishchenko, I., Maffei, M., and Schneidewind, C. “A semantic framework for the security analysis of Ethereum smart contracts”. In: *International Conference on Principles of Security and Trust*. Springer. 2018, pp. 243–269.

[17] Hajdu, Á. and Jovanović, D. “SMT-Friendly Formalization of the Solidity Memory Model”. In: *arXiv preprint arXiv:2001.03256* (2020).

[18] Hildenbrandt, E. et al. “KEVM: A complete formal semantics of the Ethereum virtual machine”. In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 204–217.

[19] Hirai, Y. “Defining the Ethereum virtual machine for interactive theorem provers”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 520–535.

[20] Idelberger, F. et al. “Evaluation of Logic-Based Smart Contracts for Blockchain Systems”. In: *Rule Technologies. Research, Tools, and Applications*. Ed. by Alferes, J. J. et al. Cham: Springer International Publishing, 2016, pp. 167–183.

[21] Johnsen, E. B. and Owe, O. “An asynchronous communication model for distributed concurrent objects”. In: *Software & Systems Modeling* vol. 6, no. 1 (2007), pp. 39–58.

[22] Johnsen, E. B., Owe, O., and Simplot-Ryl, I. “A Dynamic Class Construct for Asynchronous Concurrent Objects”. In: *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005, Proceedings*. Ed. by Steffen, M. and Zavattaro, G. Vol. 3535. Lecture Notes in Computer Science. Springer, 2005, pp. 15–30.

[23] Johnsen, E. B., Owe, O., and Yu, I. C. “Creol: A type-safe object-oriented model for distributed concurrent systems”. In: *Theoretical Computer Science* vol. 365, no. 1 (2006). Formal Methods for Components and Objects, pp. 23–66.

[24] Johnsen, E. B. et al. “A formal model of service-oriented dynamic object groups”. In: *Sci. Comput. Program.* vol. 115-116 (2016), pp. 3–22.

[25] Johnsen, E. B. et al. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Formal Methods for Components and Objects*. Ed. by Aichernig, B. K., Boer, F. S. de, and Bonsangue, M. M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 142–164.

[26] Karami, F., Owe, O., and Ramezanifarkhani, T. “An evaluation of interaction paradigms for active objects”. In: *J. Log. Algebr. Meth. Program.* vol. 103 (2019), pp. 154–183.

[27] Luu, L. et al. “Making smart contracts smarter”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM. 2016, pp. 254–269.

[28] Mavridou, A. and Laszka, A. “Designing secure Ethereum smart contracts: A finite state machine based approach”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2018, pp. 523–540.

[29] Owe, O. *Verifiable Programming of Object-Oriented and Distributed Systems*. Ed. by Petre, L. and Sekerinski, E. 2016.

[30] Owe, O., Fazeldehkordi, E., and Lin, J.-C. “A Framework for Flexible Program Evolution and Verification of Distributed Systems”. In: *Model-Driven Engineering and Software Development*. Ed. by Hammoudi, S., Pires, L. F., and Selić, B. Cham: Springer International Publishing, 2020, pp. 320–349.

[31] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ . 2018.

[32] Permenev, A. et al. “Verx: Safety verification of smart contracts”. In: *2020 IEEE Symposium on Security and Privacy, SP*. 2020, pp. 18–20.

[33] Prybila, C. et al. “Runtime verification for business processes utilizing the Bitcoin blockchain”. In: *Future Generation Computer Systems* (2017).

[34] Sergey, I. and Hobor, A. “A Concurrent Perspective on Smart Contracts”. In: *CoRR* vol. abs/1702.05511 (2017). arXiv: 1702.05511.

[35] *Solidity documentation*. https://solidity.readthedocs.io/ . 2019.

[36] Szabo, N. “Formalizing and securing relationships on public networks”. In: *First Monday* vol. 2, no. 9 (1997).

[37] Weber, I. et al. “Untrusted Business Process Monitoring and Execution Using Blockchain”. In: *Business Process Management*. Ed. by La Rosa, M., Loos, P., and Pastor, O. Cham: Springer International Publishing, 2016, pp. 329–347.

[38] Wood, G. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* (2019), pp. 1–39.

[39] Yonezawa, A. *ABCL: An Object-Oriented Concurrent System*. Cambridge, MA, USA: MIT Press, 1990.

[40] Zakrzewski, J. “Towards Verification of Ethereum Smart Contracts: A Formalization of Core of Solidity”. In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Piskac, R. and Rümmer, P. Cham: Springer International Publishing, 2018, pp. 229–247.