SIMULA SPRINGER BRIEFS ON COMPUTING 4

Olav Lysne

The Huawei and Snowden Questions

Can Electronic Equipment from Untrusted Vendors be Verified? Can an Untrusted Vendor Build Trust into Electronic Equipment?

simula

Springer Open

Simula SpringerBriefs on Computing

Volume 4

Olav Lysne

The Huawei and Snowden Questions

Can Electronic Equipment from Untrusted Vendors be Verified? Can an Untrusted Vendor Build Trust into Electronic Equipment?

Springer Open

Olav Lysne
Simula Research Laboratory
Fornebu
Norway

Simula SpringerBriefs on Computing
ISBN 978-3-319-74949-5 ISBN 978-3-319-74950-1 (eBook)
https://doi.org/10.1007/978-3-319-74950-1

Library of Congress Control Number: 2018930108

Mathematics Subject Classification (2010): 97P99

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

The Cretans, always liars…

—Epimenides, Cretan

Foreword

Dear reader,

Our aim with the series *Simula SpringerBriefs on Computing* is to provide compact introductions to selected fields of computing. Entering a new field of research can be quite demanding for graduate students, postdocs and experienced researchers alike: the process often involves reading hundreds of papers, and the methods, results and notation styles used often vary considerably, which makes for a time-consuming and potentially frustrating experience. The briefs in this series are meant to ease the process by introducing and explaining important concepts and theories in a relatively narrow field, and by posing critical questions on the fundamentals of that field. A typical brief in this series should be around 100 pages and should be well suited as material for a research seminar in a well-defined and limited area of computing.

We have decided to publish all items in this series under the SpringerOpen framework, as this will allow authors to use the series to publish an initial version of their manuscript that could subsequently evolve into a full-scale book on a broader theme. Since the briefs are freely available online, the authors will not receive any direct income from the sales; however, remuneration is provided for every completed manuscript. Briefs are written on the basis of an invitation from a member of the editorial board. Suggestions for possible topics are most welcome and can be sent to aslak@simula.no.

Springer Heidelberg, Germany
January 2016

Prof. Aslak Tveito
CEO

Dr. Martin Peters
Executive Editor Mathematics

Preface

To date, devices deliberately designed to turn against their owners have been confined to fiction. The introduction of electronic devices into our daily lives and into the critical infrastructures of entire nations does, however, threaten to transform this troubling notion into a reality—firstly, because electronic equipment has the ability to act autonomously and, secondly, because its mode of operation is often beyond what its owners can reasonably be expected to understand.

This problem became the subject of international concern around 2010, when the use of equipment from Chinese manufacturers in critical infrastructures became more widespread in Western countries. Distrust in the design prompted several countries to ban equipment from companies such as Huawei and ZTE, based on the fear that it could have been deliberately designed to act against the interests of its owners.

My interest in this topic was spawned by these discussions and bans, which pose new challenges for buyers and vendors alike: buyers have to ask themselves whether electronic equipment from untrusted vendors can be verified. On the other hand, equipment manufacturers that experience loss of market share due to distrust have to ask themselves if it is possible to (re)build trust in their electronic equipment.

Surprisingly, there is no clear consensus on the answers to these questions. If you ask a random sampling of information technology experts, just as many will say yes as say no. This result stands in stark contrast to the questions' importance. Indeed, the discussions on Huawei and ZTE mentioned above were related to matters of national security.

The intention of this book is to provide answers to these questions by analysing the technical state of the art in all relevant fields of expertise. As such, writing it required a detailed study of several fields of technology in which I am hardly an expert. To help me avoid the most obvious mistakes, all parts of the book were discussed with people far more knowledgeable than myself. In particular, Kjell Jørgen Hole, Sabita Maharjan and Aslak Tveito contributed immensely to clarifying the ideas in an early precursor project to this book. Specific aspects of the book and portions of the text were also discussed with Bob Briscoe, Haakon Bryhni, José Duato, Kristian Gjøsteen, Janne Hagen, Magne Jørgensen, Olaf Owe, Øyvind Ytrehus and Toril Marie Øye. While I take responsibility for any remaining mistakes, all of the individuals listed above greatly contributed to the book's merits.

Fornebu, Norway Olav Lysne
November 2017

Contents

Chapter 1
Introduction

In September 2007, Israeli jets bombed what was suspected to be a nuclear installation in Syria. Apparently, the Syrian radar that was supposed to warn about the attacks malfunctioned in the critical time interval prior to the Israeli attacks. Eventually, an alleged leak from a US defence contractor suggested that a European chip maker had built a kill switch into its chips. The radar may thus have been remotely disabled just before the strike took place [1].

Whatever the real truth might be, the discussions around the bombing of the Syrian nuclear plant highlight a profound difference between electronic equipment and any other type of technology. When you buy an electronic device, you enter into a long-term relationship with the people and companies who developed and produced it. Their power over the equipment prevails long after you have bought it, even when you believe to be the only one who operates it and even while it is under your physical control. The relation between makers and buyers of information and communications technology equipment is thus very different from most other buyer–vendor relationships.

Three properties of electronics created this situation. First, the functionality that is there when the equipment is shipped is largely invisible to the customer. Second, a stream of software updates gives the manufacturer the ability to change the operation of the equipment long after it has been bought. Third, since most equipment is connected to the Internet, the manufacturer has the power to receive information from the equipment and even to operate it remotely.

This begs two important questions. First, for what could a manufacturer want to use its powers? Second, what means do we have to control the actions of the manufacturer? In this chapter, we give a high-level overview of these questions. We cite cases in which the position as the manufacturer of electronic equipment was misused to make the equipment work against the interests of its owner and we explain why the problem of protecting against untrusted vendors is different from that of protecting against third-party attacks. Finally, we relate the problem to questions of national security and international trade.

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_1

1.1 A New Situation

The trading of tools between tribes is probably as old as toolmaking and trade themselves. Early artefacts were perhaps arrowheads, axes, and knives made of stone and their exchange increased the hunting or fighting capability of the parties taking part in the transactions. The quality of these early tools was easy to verify and clearly no one feared that the tools themselves could turn against their owner.

As history evolved, the tools became more complex. Consequently, understanding the quality of a product became a more complex task. Still, as recently as 50 years ago, the fear that the tools you bought could be made to turn against you was practically nonexistent. Furthermore, if you feared such attacks, countermeasures were easily available. Equipment could be disassembled and understood and any exploitable weakness or malicious functionality ran a very high risk of being detected.

Today, the situation has changed. To a rapidly increasing degree, societies, organizations, and individuals base their lives and well-being on electronic tools and infrastructures. These electronic devices and tools are built with a complexity that surpasses human capacity for analysis. This is obvious from the fact that we are frequently surprised by what the devices we buy actually do. ‘Oh, I didn’t think it could do that!’ is a reaction most of us have had to our mobile phone.

It is not only as private consumers that we are losing our ability to check what our devices really do. The complexity of modern computer systems is so great that, even for the people developing them, their behaviour is impossible to fully control. For instance, we do not expect computer systems to be free of design or programming faults anymore, because we know that building fault-free systems is nearly impossible. This is accepted to the extent that no one will buy complex equipment without a support agreement that the vendor will provide software updates to correct programming mistakes as they are identified.

1.2 What Are We Afraid Of?

The reason for such support agreements is that buyers are not expected to find the bugs themselves. Neither are they assumed to be able to correct them, regardless of the resources their organization may have. However, if the buyers of equipment are no longer expected to even understand what the electronic system they have bought can do, this has serious implications. It means that the vendor of the equipment has the power to make it do things that are not in the interest of its owner. The vendor could make the equipment turn against its owner without the owner ever finding out.

This begs the question of what a dishonest vendor could possibly do. The exact answer will vary depending on the motivation of the dishonest vendor, but the actions we need to be concerned about are the same as those we fear from third-party cyberattackers. We fear that they will carry out espionage and surveillance to get hold of confidential information, from either companies, private persons, or nation states.

We fear that they will sabotage key equipment, either permanently or temporarily, to achieve some objective. Finally, we fear that they can commit fraud. Do we have stories of equipment manufacturers using their position as vendors for espionage, sabotage, or fraud? Yes, we do. Some examples are documented beyond doubt, some are made credible through circumstantial evidence, and some are just fear-based speculation.

One well-documented case of espionage is that the routers and servers manufactured by Cisco were manipulated by the National Security Agency (NSA) to send Internet traffic back to them. This case is well documented through documents made available to the press by Edward Snowden [3] and it is therefore a solid example of espionage done through an equipment provider. There is no evidence that Cisco was aware of the NSA's tampering, but the example is still relevant. When you buy equipment from a provider, you have to trust not only the provider, but anyone who is in power to change the equipment. To what extent Cisco itself contributed is therefore beside the point.

Another case of privacy-invading espionage is from November 2016. Kryptowire,[1] a US-based cybersecurity company, found that several models of Android mobile devices contained firmware that transmitted sensitive personal data to a server in China without disclosure or the users' consent [5, 7]. The code responsible for these actions was written by Shanghai Adups Technology Company, a Chinese company that allegedly provided code for 700 million phones, cars, and other smart devices. The phones were available through major online retailers and they sent user and device information to China. The leaked information included the full body of text messages, contact lists, and call history, with full telephone numbers. The intention of this surveillance is unclear, but Adups explained that the code was made for a Chinese manufacturer. One of their lawyers described its presence in phones sold in the United States as a mistake.

An example of fraud is the Volkswagen case, in which electronic circuits controlling a series of diesel engines reduced engine emissions once they detected they were being monitored [2]. This was deliberately done through the functionality of an electronic component put into cars by their manufacturer. Volkswagen misled American authorities into issuing approvals of the engine series, even if, under ordinary use, the engines emitted nitrogen oxide pollutants up to 40 times above allowable US limits. Volkswagen also misled buyers into thinking that the cars were environmentally friendly. This case was eventually detected by the authorities, but the important observation here is that the detection was not the result of analysis of the digital component; it was the result of analysis of the highly analogous behaviour of the diesel engine. The engineers who committed this fraud did so under the assumption that what they had done on the electronic chip itself would not be discovered.

Examples of sabotage from vendors of electronic equipment are fewer and less well documented. The event that we started the chapter with – when the Syrian radars malfunctioned just before an Israeli airstrike on a nuclear installation in 2007 – has

[1] Kryptowire is a contractor of Homeland Security. The discovery reported here was reportedly done outside of these contracts.

not been documented beyond doubt. In the aftermath of the event, there was intense speculation of what had happened, but no real proof has been put forward.

The merits of such speculation are, however, not important. The crucial observation is that we seldom have hard evidence to dismiss such arguments. The motivation of the vendor of the equipment to include unwanted functionality in the Syrian radar is clear. Its technical ability to go through with it is quite obvious and the chances the Syrians had to uncover the problem before it was too late appear to have been slim. Even in the absence of verified examples, the inclusion of kill switches that render equipment useless through a predefined external trigger is too powerful a concept to be ignored.

1.3 Huawei and ZTE

Discomforting as the above examples are, the most heated debate on trust in vendors of electronic equipment relates to Huawei and ZTE. These two Chinese companies are becoming increasingly dominant in the telecommunications market. Because of the vendors' Chinese origin, countries such as the United States, Australia, and Canada have repeatedly questioned their trustworthiness and objected to including their equipment in national infrastructures. The distrust is mainly motivated by Huawei's possible ties to the Chinese military and thus to the Chinese government.

Open political debate on how to handle the question of whether to trust Chinese vendors has taken place on three continents. In October 2012, the US government released a report by the US House of Representatives' Permanent Select Committee on Intelligence [10]. The report presented an analysis of the threats associated with remote equipment control, especially in the case of a cyberwar, and strongly urged that US firms stop doing business with both Huawei and ZTE. Australia has echoed the same concern. In 2011, the Australian government barred Huawei from bidding for the country's national broadband network project, based on security concerns associated with Huawei's hardware [12]. In Europe, Huawei, along with its smaller rival ZTE, hold almost a quarter of the European telecommunications equipment market. According to the European Commission, this strong market position poses a security risk because European industries ranging from healthcare to water utilities are becoming reliant on Chinese wireless technology [9]. In the United Kingdom, a parliamentary committee commented that it was "shocked" at the government's failure to monitor Huawei's activities and called its strategy for monitoring or reacting to cyberattacks "feeble at best" [11]. In France, telecommunications executives say the government generally discourages them from buying Chinese equipment for their core networks, but not for cell phone base stations and radio equipment [11].

It is important to note that there is no evidence or indication that Huawei, ZTE, or any other Chinese vendor, for that matter, has misused its position as a vendor of equipment against any of its customers; there are only suspicions, mainly based on Huawei's possible ties to the Chinese military and thus with the Chinese government. Therefore, some would categorize the suspicions as unsubstantiated fear. To us, however, the discussions themselves are important. From their mere existence, we can draw three very important conclusions, as follows:

1. It is critical for a modern society to be able to rely on its electronic equipment and infrastructures. All of society's critical functions, such as healthcare, financial stability, water supply, power supply, communication, transport, and ability to govern a state in a situation of national crisis, depend on them. The need to have an understood level of trust in critical infrastructure is the reason the discussions on Huawei and ZTE have reached the level of national security politics.
2. Investigating what electronic equipment does or can do is highly nontrivial. If it were easy, we would not need to discuss how much we can trust such equipment; checking and verifying it would suffice.
3. Few countries have the ability to design and produce the electronic equipment for their critical infrastructure entirely by themselves. In other areas, governmental regulatory bodies impose rules on systems of critical importance to national security. If designing and producing critical equipment nationally were a viable path, it would be an obvious solution to the problem.

The three observations summarize the motivation for this book. The trustworthiness of electronic equipment is paramount and we cannot choose to design and produce all aspects of this equipment ourselves. The question we then need to answer is how we can check and verify the equipment and how we can build well-founded trust.

1.4 Trust in Vendors

Trust in a vendor may have a varying foundation. We can choose to trust a vendor based on a long-term relationship or based on the vendor's reputation. In both cases, our trust is based on their previous actions. We could also base trust on our capacity to verify the contents of the vendor's equipment. In this case, the decision to trust rests on our belief that malicious intent by the provider will in some way be visible in the product and that we will be able to detect it. These bases for trust are often used in the discussions regarding Chinese vendors of telecommunications equipment. Unfortunately, as we shall document in this book, these arguments do not hold water.

When we discuss our fear of what equipment providers could do, our concerns will in all cases be related to what they could possibly do in the future. A key observation is that a provider could change tomorrow the properties of equipment we bought and installed today, since hardly any complex electronic equipment is sold without a support agreement. A steady stream of security updates will come from the provider of the equipment. For fear of being attacked by a third party exploiting the security holes closed by the updates, we will dutifully install them. These updates will have the power to change the operation of the equipment itself, even to the extent of performing actions against our interests. Consequently, the question of trust in a vendor has a timeless perspective. We may have every reason to trust a vendor based on experience with the company in the past. We may also be correct in that the vendor is currently to be trusted. Unfortunately, our need for software updates makes any change in the vendor's trustworthiness influence the trustworthiness of all the equipment ever bought from that vendor. Therefore, when we ask ourselves if a vendor can be trusted, we have to ask ourselves if we believe the vendor will remain trustworthy for the entire lifetime of the product we are buying.

1.5 Points of Attack

So far, we have somewhat inaccurately discussed the notion of a vendor as if it needed no further explanation. There are, however, many types of vendors and each type has a different potential point of attack and a different potential to cause harm [4]. Some provide pieces of intellectual property to be integrated as a small part of a silicon chip. Others may build the chip itself or the printed circuit board onto which the chip is welded. Some vendors provide the hardware abstraction layer or the operating system running on top of the hardware and a large group of vendors provide applications running on top of the operating system. There are system integrators synthesizing hardware, operating systems, and applications into a complete product and there are service providers providing communication or data centre services. Most vendors of electronic equipment are also buyers of software or electronic components needed to build what they eventually sell.

The most critical buyer–seller relationship is that between an operator of critical infrastructure – such as telecommunications, power grids, and cloud facilities – and the provider selling the equipment for this infrastructure. Throughout the book, we will often assume a vendor–buyer relationship in which the values at stake are high. Furthermore, the equipment bought is assumed to be a fully integrated physical product containing hardware, firmware, an operating system, drivers, and applications. Still, most of this book is of relevance to all buyer–seller relationships in the computer business, including buyers and sellers of components that go into finished products and buyers of electronic services provided by operators of infrastructures.

1.6 Trust in Vendors Is Different from Computer Security

Computer security has been a research field for decades. We should therefore ask ourselves whether the problem we address in this book raises new questions of computer security that have not previously been investigated. To answer this question, we have to extract and express some underlying assumptions in most work on computer security.

An often-cited definition of computer security is that of Peltier: [8]: *Information security encompasses the use of physical and logical data access controls to ensure* *the proper use of data and to prohibit unauthorized or accidental modification, destruction, disclosure, loss or access to automated or manual records and files as well as loss, damage or misuse of information assets.*

This definition or variants thereof has formed the basis of security research for decades. Its weakness – and thus the weakness of most security research – is that it focuses on "physical and logical data access controls". Although this phrase does not explicitly exclude the scenario in which the vendor of the equipment is the perpetrator, it has led to the almost ubiquitous and most often implicit assumption that the buyer and the producer of equipment are collaborating in defending against a third-party wrongdoer.

Out of this assumption comes a set of approaches that are not valid for our case, in particular, those for stopping malicious software, or malware, from entering your system. If the wrongdoer is the vendor, the malware may be in the system from the moment you buy it and stopping it from entering later therefore makes no sense. Another assumption is related to the mechanisms for detecting malware infections. These methods assume there is a noninfected golden sample for comparison. Since this golden sample has to be provided by the vendor, the notion of being noninfected is doubtful if you do not trust the vendor [6]. Yet another mechanism is based on building cryptographic security into hardware. A Trusted Platform Module is a piece of hardware that can be used to verify that the configuration of software and hardware on a system has not been tampered with. In our case, this will not help. We inevitably need to ask the question of who made the cryptographic module and to what extent the maker can be trusted. In summary, four aspects make our problem different and far more difficult to handle than the scenarios addressed in mainstream security research:

- When malware is already in the system at the time of purchase, stopping it from entering is futile.
- When there is no golden sample, it is not possible to detect tampering by comparing a system to a known healthy system.
- Built-in security mechanisms in system chips, operating systems, or compilers are in the hands of vendors we may not trust.
- The malicious actions of the system can be performed anywhere in the technology stack, from low-level hardware to software controlling the user interface.

The implications of these differences will be discussed in this book. We will go through the different computer science specialities and analyse the extent to which the current state of the art contains knowledge that can help us solve our problem.

1.7 Why the Problem Is Important

The importance of a problem should be measured by its consequences. Before we spend time delving into the finer details of controlling an untrusted vendor, we therefore need to understand the consequences of not being able to do so. It turns out that there are consequences for the buyer of the equipment, the vendor of the equipment, and for society at large.

The consequences for the buyer of the equipment are the most evident. If you need to invest in equipment for critical infrastructure in your country, you will be concerned with everything that can go wrong. You also need to think through scenarios of international crisis and even war. The discussions related to the Chinese companies Huawei and ZTE are examples of this challenge. Although no international consensus is reached on how to treat untrusted vendors, the discussions themselves reveal that the problem is considered important to the extent that it has reached the level of international politics.

The vendors, on the other hand, see the problem differently. Assume you are in the business of selling equipment and that you are completely trustworthy. Still, you lose contracts because there is no way you can prove beyond a doubt that your equipment is trustworthy. This is the situation in which the Chinese providers find themselves. No matter how understandable the worries of Western countries are, it is evident that it is extremely important for Huawei to be able to demonstrate the worries are groundless.

Finally, the difficulties we have in building a solid basis for trust between vendors and buyers of electronic equipment are a potential obstacle for international trade and cooperation. The internationalization of trade since the Second World War has fuelled financial growth in most societies around the world and this trade has been argued to be an important factor in the formation of trust between nation states. This trust has, however, never been blind, in the sense that the parties could not control the quality, properties, or actions of the goods that were traded. The lack of a solid basis for trust between buyers and sellers of electronic equipment therefore not only is a problem in the relation between the buyer and the seller, but can also be seen as a factor limiting the trust needed to grow international trade.

1.8 Advice for Readers

The topic of this book should be of interest to several groups of readers. One is security experts who need to understand the vulnerabilities implied by the vendor–buyer relationships they are a part of. Another is security researchers who need to understand the limitations of the state of the art, when a system must be defended against different adversaries from those previously considered.

An often overlooked but important group of stakeholders for this problem is the decision makers in nation states, public bodies, and large companies. Their need to understand this problem is huge, since they are the ones ultimately making the decisions that establish the type of customer–vendor relationships on which we focus. More often than not, these people are not educated technologists.

Therefore, We have made an effort to make this book readable and useful for all groups. Technologically savvy readers are advised to read the entire book from start to finish or to focus on the chapters of special interest between Chaps. 4 and 10.

Interested readers without a technology background should gain an informed view of the problem at hand by reading Chaps. 1–3, the conclusion of Chap. 4, and, finally, Chap. 12.

References

1. Adee, S.: The hunt for the kill switch. IEEE Spectr. **45**(5), 34–39 (2008)
2. BBC: http://www.bbc.com/news/business-34324772
3. Greenwald, G.: No Place to Hide: Edward Snowden, the NSA, and the US Surveillance State. Macmillan, Basingstoke (2014)
4. Jøsang, A.: Potential cyber warfare capabilities of major technology vendors. In: 13th European Conference on Cyber Warfare and Security ECCWS-2014, pp. 110–115 (2014)
5. Kryptowire: Kryptowire discovers mobile phone firmware that transmitted personally identifiable information (PII) without user consent or disclosure. http://www.kryptowire.com/adups_security_analysis.html
6. Lysne, O., Hole, K.J., Otterstad, C., Aarseth, R., et al.: Vendor malware: detection limits and mitigation. Computer **49**(8), 62–69 (2016)
7. New York Times: Secret back door in some U.S. phones sent data to China, analysts say. http://www.nytimes.com/2016/11/16/us/politics/china-phones-software-security.html?smprod=nytcore-iphone&smid=nytcore-iphone-share
8. Peltier, T.R.: Information Security Risk Analysis. CRC press, Bocca Raton (2005)
9. Reuters: Exclusive-EU threatens trade duties against China's Huawei, ZTE-sources. http://www.reuters.com/article/us-eu-china-huawei-idusbre94d0rx20130515
10. Rogers, M., Ruppersberger, D.: Investigative report on the U.S. national security issues posed by Chinese telecommunications companies Huawei and ZTE. http://intelligence.house.gov/sites/intelligence.house.gov/files/documents/huawei-zteinvestigativereport(final).pdf
11. The Wall Streeet Journal: U.K. raises cybersecurity concerns over Huawei. http://online.wsj.com/article/sb10001424127887323844804578529141741985244.html
12. ZDNet: Don't let Aus fall into US-China security fight: Huawei. http://www.zdnet.com/dont-let-aus-fall-into-us-china-security-fight-huawei-7000006282/

Chapter 2
Trust

A relationship between a buyer and a seller of electronic equipment is one of trust. The buyer of the equipment trusts the seller to deliver the equipment on time, with the right quality, and at the agreed price. Usually the buyer also has to trust the seller to provide support and security updates for the lifetime of the product. The focus of this book is somewhat unusual, since we are not concerned with price, quality, or technical support. Rather, we study the relationship between the seller and the buyer under the assumption that the seller might want to use its position as the equipment provider for purposes that are directly opposed to the interests of the buyer. From this position, the notion of trust between the equipment provider and the buyer of the equipment takes on a very different flavour.

Notions of trust have been heavily studied in modern philosophy from the 1980s onward. Much of this research is based on rational choice theory and most authors relate trust to an aspect of *risk taking*. The question that one tries to answer is how to capture the rationality of relationships between parties that are not transparent to each other. Framed by the problem we address in this book, the question is how to make rational decisions on buying equipment from a vendor when neither the interests of the vendor nor the current and future contents of the product are fully transparent.

2.1 Prisoner's Dilemma

One of the most celebrated examples that highlight the complexity of decisions based on trust is the prisoner's dilemma [12]. There are several versions of it, but the one most frequently cited is the following: two partners in crime are imprisoned. The crime they have committed will, in principle, give them a sentence of three years but the police have only evidence to sentence each of them to prison for one year. The police give each of the partners the offer that the sentence will be reduced by one year if they testify against the other. If only one defects, the other will serve three years in prison. However, if both partners defect, both of them are put away for two years.

What makes this situation interesting is the choice that each prisoner must make between defecting and not defecting. If you are one of the prisoners, you can go free if you defect and the other does not. You get one year in prison if none of you defects. If both of you defect, you get two years in prison and, if you do not defect but your partner does, you end up with a three-year sentence. Clearly, what would generate the least number of years in prison for the two prisoners in total is for neither to defect. This situation would require, however, that each one trust the other not to defect. Analysis based purely on self-interest makes it clear that whatever choice my partner makes, I will be better off defecting. If both prisoners follow this strategy, we will both receive a two-year sentence. Still, the best outcome is if my partner and I can trust each other not to defect. Then we would both receive a one-year sentence.

The winning strategy in the prisoner's dilemma is to defect, but the game changes when the same dilemma appears with the same pair of prisoners multiple times. It becomes even more complicated if there are multiple prisoners and if, for each iteration, any arbitrary one of them is involved in the game. In the latter case, the winning strategy depends on the average attitude of the population. If the population tends to collaborate, then tending towards collaboration is a winning strategy, whereas tending towards defection is a winning strategy if that is what most of the population does [2].

The prisoner's dilemma has been used to describe the appearance of trust-based collaboration among animals, in politics, and in economics, to mention just a few areas. It seems to capture the essence of trust-based decisions when there are risks of defection involved. For us, the risk we are concerned with is related to the fear that the vendor from which we buy our equipment will defect on us and the choice we have to make is whether we will buy from this vendor or not. For the sake of discussion here, we assume that the vendor has motivation for defecting, without going into detail what it might be.

The first question that arises is whether the singular or the iterated version of the prisoner's dilemma is involved. The answer depends heavily on what our fears are or, to put it in another way, what the defection of the vendor would amount to. If we buy a mobile phone and we fear that the vendor could steal the authentication details of our bank account, we have a case of the iterated prisoner's dilemma. It is unlikely that we would be fooled more than once and we would not buy equipment from the same vendor again. The vendor would most likely suffer a serious blow to its reputation and subsequent loss of market share, so the vendor would not be likely to defect.

At another extreme, we find nations that buy equipment for their critical infrastructure. If defection means to them that the infrastructure would be shut down as part of an attempt to overthrow the government, then the game is not likely to be played out more than once. In this case, the winning strategy for the nation would be to find another vendor and the aggressive vendor would not let the possible loss of a customer stop it from defecting.

It is therefore clear that decisions about which vendors to trust will depend heavily on the consequences of defection. Another equally important aspect of the discussion is the degree to which the vendor's actions are detectable. Assume that we, as buyers of equipment, fear that the vendor will use the equipment to steal information from us. The realism of this situation is highlighted by the fact that unauthorized eavesdropping is one of the greatest fears behind the discussions on whether to allow Chinese vendors to provide equipment for critical infrastructures in Western countries. In this case, it is unclear if we would ever know if the vendor defected. Therefore, even if we are in a situation of repeated decisions on whom to buy from, we might be no wiser the second time than we were the first. In terms of the prisoner's dilemma, this means that we can assume that the buyer will never find out if the vendor has defected or not and, then, the game can, for all practical purposes, be viewed as a single game. Still, seen from the perspective of the vendor, there is also a risk associated with the assumption that the vendor will never be caught. We return to this discussion in Sect. 2.7.

2.2 Trust and Game Theory

The prisoner's dilemma and variants that we presented above are examples of a problem from game theory; that is, the rules of the game are transparent and its outcome for each individual results from the decisions made by all the participants. An elaborate mathematical theory can be built around such games, given a solid foundation for what will be the rational choices of each participant in the game.

Basic notions of trust can be derived from such a theory of rational choice. Gambetta [6] defines trust (or, symmetrically, distrust) as

A particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such action (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action.

Seen from this point of view, a trust-based choice of a vendor for a particular piece of technology would be based on a subjective understanding of the chance that vendor fulfils the buyer's expectations, and does not use its position as equipment provider for purposes that are against the buyer's interests.

Rational choice as a basis for trust does have its weaknesses. Consider the iterated prisoner's dilemma again. It is easy to imagine a situation in which you do not trust your fellow prisoner. Still, knowing that establishing trust would be beneficial to the outcome of the game, it might be a rational choice to play as if you trusted the other prisoner, even if you do not. This point, taken up by Hardin [9], highlights a distinction between beliefs and strategies. It defines trust as a belief of what your opponent will actually do. When I trust you, it is because I believe it will be in your best interest to protect my interests. The implications and value of building a trust relation between a buyer and a vendor of electronic equipment is discussed further in Sect. 2.5.

2.3 Trust and Freedom of Choice

One major weakness in the game-theoretical definitions of trust is that trust is assumed to be something one can choose to have or not to have. In an essay on trust and antitrust, Baier [3] argues that this is not always the case. The most convincing example is the trust a small child must have towards his parents. He will not be in a position to choose other parents, so, regardless of what reasons he has been given to trust the ones he has, he actually has to place some trust in them.

The insight of Baier's work, for our purposes, is that you might be forced to entrust something that is of value to you to people or organizations you do not necessarily trust. There are many examples of this in the world of computers. In a modern society, one can hardly choose not to be dependent on the national communications network. Still, you may not trust it to be there when you need it or you might have reasons to believe that your communications are being intercepted by national authorities that you do not trust. You may not trust your computer to be virus free but you may still feel that you have to entrust it with the task of transferring money from your bank account.

When we choose which vendor of electronic equipment to use for critical infrastructure, we are in a situation in which our choices are very limited. There will generally be very few vendors to choose from and we will rarely be in a position to choose not to buy at all. If we decide, as some governments have, that equipment or components from a given country should not be trusted, we may find that there are no trustworthy options left. A very obvious example is that several Western countries have decided to prohibit Chinese equipment from central parts of their telecommunication infrastructure. In today's complex world, there is hardly any piece of modern electronic equipment without at least one part that was designed or fabricated in China. To a significant degree, the information carried by nations' telecommunication systems must be entrusted to components made in countries these nations do not trust.

2.4 Trust, Consequence, and Situation

Lack of real choices is not the only limitation to basing trust on game-theoretic approaches. Another shortcoming is the fact that the level of trust depends on factors that will typically not be part of the game. Most people would agree that the level of trust required to make a given choice will depend on the consequences of being let down. Govier [7] provides an example in which a stranger on the street volunteers to carry packages for you. Your willingness to entrust your packages to this stranger will depend on what the packets contain. If they contain things that are highly valuable to you, you would be less likely to entrust them to a stranger and you would require a higher level of trust to do so. The same considerations apply to your situation. If you have only a small, light packet that fits easily in your hand, the mere fact that some stranger is volunteering to help you will probably make you distrust that person.

To us, the most relevant learning point from Govier's work is that the answer to the question of whether to trust an equipment vendor will depend highly on what kind of equipment it is and on how you intend to use it. At one extreme is the private person who buys an alarm clock to help wake up in the morning. The level of trust the person has on the manufacturer of the clock should hardly be a criterion of choice. The worst thing that can happen is that the person will oversleep a day or two, requiring her to write off the costs when she buys a new clock. At the other extreme, we have investments in national infrastructures for electronic communications and control systems for power plants. The level of trust needed before one selects a provider of equipment for such structures is very high and cases in which lack of trust has barred named providers from competing for such contracts are well documented.

2.5 Trust and Security

The terms *trust* and *security* are usually seen as interrelated and it is common to assume that the presence of one will promote the formation of the other. If you place trust in a person, you expose yourself to harm from that person. This person knows that he can retaliate if you misbehave and thus has all the more reason to trust you by exposing himself to you in the same way. The fact that you are both exposed will reduce your inclination to harm each other and you therefore end up being more secure. Similarly, when one experiences a system as being secure, one starts to trust it. The fact that an online baking website is relative secure makes one trust it.

Unfortunately, the idea that trust and security are a consequence of each other is not always true. These two words have many interpretations and the claim that one follows from the other is valid only for some interpretations [11, 13]. In particular, placing trust in a computer system will not make it secure. A computer system will not (well, at least not yet) change simply because it is trusted not to do anything wrong. One objection to this argument could be that it is not the computer system itself with which you intend to build a secure relationship but, rather, with the people who developed it. Still, there is an inherent asymmetry in the exposure to harm in the relationship between a vendor and a buyer of equipment. Sometimes it is to the buyer's advantage, in the sense that the buyer can withhold payments or hurt the vendor's reputation. Other times it is to the vendor's advantage, particularly when the vendor is controlled by powers, for example, states, that are not exposed to the financial threats of single companies. If you trust a vendor, this trust will not automatically make a vendor safe to use; that will depend on the balance between the risks you expose to each other.

What, however, if we turn the question around? If I develop a highly secure system, would not the consequence be that my customers will tend to trust me? It would appear that the answer to this question is yes. Huawei is arguably the company that is the most challenged by customer distrust throughout the world. It has been banned from delivering products to the critical infrastructures of several countries and it has complained that it does not have a level playing ground when competing for contracts. One of their solutions to this challenge has been to focus heavily and visibly on making their products secure. Unfortunately, this strategy – even though it might be financially successful – misses the focal point of the discussion. It is entirely possible to make a system that is very secure against third-party attacks but where the maker of the equipment, that is, the second party, has full access to do whatever it wants. An analogy would be if you had a locksmith change all the locks of your house. Even if the locks were to keep all burglars out, the locksmith could still keep a copy of the key. Even if you made your house secure, this security is still built on the trustworthiness of the locksmith.

This section ends with two important observations. In the vendor–user relationship related to electronic equipment, we cannot assume that security comes with trust. We cannot assume that trust comes with security either.

2.6 Trusted Computing Base; Trust Between Components

A computer system is built by putting together a range of different pieces of technology. Multiple components constitute the hardware that an application is running on and an operating system will typically be composed of various parts. Adding to this, we have external devices and drivers, before we find an application on top of the technology stack. Finally, the application itself will be made of different components. More details on each of these components are given in Chap. 3.

These different parts can fail or default individually. It is thus clear that a notion of trust is also relevant between the constituent parts. A well-designed system will have an internal policy that makes it clear to what degree each part is supposed to be trusted, based on what tasks they are expected to perform. Such trust between components is described by Arbaugh et al. [1], with the eye-opening statement that *the integrity of lower layers is treated as axiomatic by higher layers*. There is every reason to question the basis for this confidence in the integrity of lower layers.

In their Orange Book from 1983, the US Department of Defense introduced the concept of the Trusted Computing Base (TCB). This is a minimal set of components of a system upon which the security of the entire system depends. Security breaches in any other component can have severe consequences, but the consequences should be confined to a subset of the system. Security breaches in the TCB, however, will compromise the entire system and all of its data. The Orange Book makes the case that, in a security-critical system, the TCB should be made as small as possible. Ideally, it should be restricted to sizes such that the security can be verified using formal methods.

Unfortunately for us, the notion of a TCB is of little help. The strengths and shortcomings of formal methods are discussed in Chap. 9, but the main reason why a TCB cannot help us is embedded within another fact. As long as you do not trust the manufacturer of the system, it is hard to limit the TCB to a tractable size. As discussed by Lysne et al. [10], the developer of electronic equipment has many possible points of attack that are unavailable to a third-party attacker. In particular, Thompson [15] has demonstrated that the compiler or any other development tool could be used as a point of attack. This makes the compiler as well as the synthesis tools used for hardware development part of the TCB. Furthermore, since the compiler itself is built by another compiler, the argument iterates backwards through the history of computing. Computer systems and compilers have for a long time evolved in generations, each generation being built using software from the preceding generation. For example, in the UNIX world, the origins of today's tools are found in the first design of the first system back in the 1970s [5]. Finding a small computing base that can truly be trusted given a dishonest equipment provider is therefore close to an impossible task.

2.7 Discussion

The need for trust between buyers and vendors of electronic equipment depends heavily on what is at stake. There are electronics in your egg timer, as well as in systems managing and monitoring the power networks of entire nations; however, whereas the need for trust in the former case is close to nonexistent, in the latter case, it is a question of deep concern to national security. This insight from Govier [7] helps us to focus the discussions in this book. We are not concerned with fine details on how the need for trust should be derived from a consequence analysis related to the equipment in question. Rather, we concentrate our discussion of trust on cases in which the damage potential is huge and thus the need for trust is paramount.

Long-term relationships with equipment providers are sometimes advocated as beneficial for building trust and confidence between customers and vendors. For many aspects of the customer–vendor relationships, we would agree. The quality and stability of a product, the availability of support, price, and time of delivery are all aspects of electronic equipment where trust can be built over time. It can also be argued that classic cybersecurity, where the vendor is supposed to help you to protect against third parties, is an area where trust can be built gradually. If, over several years, my equipment is not broken into, I will be inclined to trust my vendor. If I need to protect against the vendor itself, however, a long-term relationship is not likely to make me secure. Rather, a long-term relationship is likely to make me more exposed. An equipment vendor of complex equipment almost always has the opportunity to change the equipment it has already sold and deployed through software updates. A vendor that has proven to be trustworthy over a long period could become malicious through a change of regimes in its home country or through changes in ownership or management. A long-term relationship therefore increases the need for trust and confidence but it does not constitute a basis for trust in itself.

In the international discussions concerning Chinese vendors of telecommunication equipment, we have frequently heard arguments in the style of Hardin: we can trust Chinese vendors because it is in their best interest not to use their position against others. If they did, they would be out of business as soon as they were caught in the act. We should indeed take it as a fact that Chinese vendors of electronic equipment may be trustworthy. Still, regardless of the origin of the equipment, we should not make Hardin's kind of trust a basis for our security concerns about equipment in a country's critical infrastructure: First, because in questions of national security, the situation often has similarities to single games of the prisoner's dilemma. A hostile takeover of a country takes place once and there is no obvious sequence of games in which a dishonest vendor will pay for its actions. Second, there are examples of large companies that deliberately cheated with their electronic equipment, were caught in the act, and which are still in business. The most well-known example currently is the Volkswagen case, where electronic circuits controlling a series of diesel engines reduced engine emissions once they detected that they were being monitored [4]. A related example is the claim made by Edward Snowden that routers and servers manufactured by Cisco were manipulated by the National Security Agency (NSA) to send Internet traffic back to them [8]. There is no documentation showing that Cisco was aware of this manipulation. Still, the example highlights that the question of trust in an equipment vendor is far too complex to be based on our perception of what will be in the vendor's best interest. This argument is strengthened by Cisco's claim that its losses due to the event were minor [14].

An old Russian proverb, made famous by Ronald Reagan, states, 'Trust, but verify!'. This means that, in all trust relations, there is a limit to how far blind trust can go. At the end of the day, trust must be confirmed through observations; thus our need for trust depends heavily on our ability to observe. As Gambetta [6] points out, 'Our need of trust will increase with the decrease of our chances actually to coerce and monitor the opposite party'. For us, this raises the question of how to coerce and monitor the actions of an equipment vendor. The remainder of this book is devoted to exactly that question. We examine the state of the art in all relevant areas of computer science in an attempt to answer the question of how we can verify malicious actions in electronic equipment or prepare for them.

References

1. Arbaugh, W.A., Keromytis, A.D., Farber, D.J., Smith, J.M.: Automated Recovery In A Secure Bootstrap Process (1997)
2. Axelrod, R., Hamilton, W.D.: The Evolution Of Cooperation. Science (1981)
3. Baier, A.: Trust and antitrust. Ethics **96**(2), 231–260 (1986)
4. BBC: http://www.bbc.com/news/business-34324772
5. Danezis, G.: Trust as a methodological tool in security engineering. In: Trust, Computing, and Society, vol. 68 (2014)
6. Gambetta, D.: Trust: Making and breaking cooperative relations. In: Can We Trust, vol. 13, 213–237 (2000)
7. Govier, T.: Dilemmas of Trust. Cambridge University Press, Cambridge (1998)
8. Greenwald, G.: No Place To Hide: Edward Snowden, The NSA, and the US Surveillance State. Macmillan (2014)
9. Hardin, R.: Trust and Trustworthiness. Russell Sage Foundation (2002)
10. Lysne, O., Hole, K.J., Otterstad, C., Aarseth, R., et al.: Vendor malware: detection limits and mitigation. Computer **49**(8), 62–69 (2016)

11. Nissenbaum, H.: Securing trust online: wisdom or oxymoron? Boston University Law Review. **81**(3), 635–664 (2001)
12. Poundstone, W.: Prisoner's Dilemma. Anchor (2011)
13. Simpson, T.W.: Computing and the search for trust. In: Trust, Computing, and Society p. 95 (2014)
14. The New York Times: Revelations of N.S.A. spying cost U.S. tech companies. https://www.nytimes.com/2014/03/22/business/fallout-from-snowden-hurting-bottom-line-of-tech-companies.html?
15. Thompson, K.: Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)

Chapter 3
What Is an ICT System?

The full complexity of the information and communications technology (ICT) systems that we use every day is hard to fathom and it spans at least two dimensions. First, if I were to send an e-mail to my colleague in the office next door, the process easily involves more than a hundred devices over two continents. On its way from my computer in Norway to the mail server I use in the United States, it will traverse routers and switches in several countries. Each of these routers and switches will be dependent on several other components just to determine the next hop on the path towards the recipient of the e-mail.

The other dimension regards the complexity of every single one of these hundred devices. From the observable functionality of each device, there are multiple layers of technology all built on top of each other before we come down to the physical phenomena that allowed us to build the device in the first place. In this chapter, we describe the most important of these layers and conclude on how each layer can be exploited by a dishonest maker of ICT equipment.

3.1 Transistors and Integrated Circuits

In a discussion on the most important inventions of mankind, a strong argument can be made that the *transistor* [1] is on par with fire, the wheel, and agriculture. A few decades after its invention, it has changed the lives of billions of people and is now central to the way we work, the way we communicate and interact, and our consumption of cultural expressions.

However important, the transistor is conceptually a very simple thing. It can be explained as an electrically controlled light switch. The transistor is simply a device that lets the electric current through one conductor open and close a switch, and thereby start or stop electric current in another conductor. At a higher level of abstraction, we observe the transistor move information from one electric circuit to another and, from this observation, we can understand that the invention of the transistor was the seminal start of what came to be known as information technology. Circuits

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_3

consisting of transistors and resistors can be constructed so that they implement all the logic functions of Boolean algebra. Circuits implementing a Boolean function are generally called *logic gates* and, through clever combinations of such gates, all the computing functions of a modern computer can be constructed.

Of course, electrically controlled switches in the form of relays and vacuum tubes had been designed before the transistor was invented and they had been used for building computers as well [3, 10]. What the transistor did bring to the table was two crucial properties. First, it could switch much faster and far more reliably than the old technologies and, second, it could be miniaturized. These features have given rise to modern *integrated circuits* with a number of transistors that today (2017) can run in the multiple billions and operate at a switching speed of multiple gigahertz.[1]

Already at the level of the transistor we find places where a manufacturer can build in malicious functionality. The physical phenomena that allowed us to build transistors are the link between the analog continuous world and the discrete world of the computer, but this link is by no means trivial to engineer. There are a number of ways a transistor can become unstable or stop working as intended. Therefore, already here we find places where a manufacturer can plan to do harm. A kill switch can be implemented by overloading selected transistors based on a given input signal, thus leaving the integrated circuit unusable. Completely investigating a product for malicious functionality therefore goes all the way down to studying the countless transistors on each chip.

3.2 Memory and Communication

The three central elements of a computer are data processing, the storage of information, and communication. All of these elements can be implemented by transistors and logic gates.

Communication in the sense of moving information from one place to another can, in its simplest form, be done by transmitting high or low power on a conductor. Thus communication can easily be implemented by means of transistors. A great deal of development has gone into improving the speed and distance of communication links, as well as reducing the fault rate, and the most successful developments are based on using light waves in optic fibres rather than electric signals in metallic conductors. Apart from the added complexity that accompanies these developments, they are not directly relevant to the discussions we cover in this book. We therefore refrain from giving further details.

There are a number of ways in which one can store information using logic gates. The general idea is that to store a bit, you can design a circuit with a feedback loop, so

[1]In 1965, Gordon Moore observed that the number of transistors on an integrated circuit seemed to double every two years. The truth of the observation has been remarkably persistent ever since and has therefore been referred to as Moore's law. The switching speed of transistors grew as fast as the increase in the number of transistors for a long time, but it started to level off around 2010.

that the input to a logic gate depends on the output of the same gate. Such a feedback loop can create a perpetually unstable circuit: for example, when an output value of 1 fed back into the circuit will generate an output value of 0 and vice versa. If made correctly, however, the circuit will have exactly two stable states, corresponding to a stored 0 and a stored 1, respectively. Storage built from logic gates in this way are known as volatile memory, meaning that the information is retained only as long as the system is powered on. This setup is used for registers in the CPU and for RAM. There are a number of technologies available for storing information in a more durable form, such as hard disks and flash memory.[2] These are dependent on physical phenomena other than those of the transistor but they still depend on transistor-based logic circuits for their control logic.

Both memory circuits and communication circuits can be exploited by an untrusted vendor. Above we argued that a circuit can be constructed so that it self-destructs on a given input signal. For memory and communication circuits, it is clear how such input signals can be received. For a communication circuit, the signal can arrive from the outside world, whereas a memory circuit can be designed to self-destruct when a particular sequence of bits is either written to or read from memory. Self-destruction is, however, a very simple form of sabotage that can be controlled by an untrusted vendor. More advanced operations, such as espionage and fraud, are primarily made possible through the logic that controls the memory and communication circuits. We discuss such control logic and processors next.

3.3 Processors and Instruction Sets

What drives the actions of the ICT system forward are the circuits of logic gates that perform actions on the memory and the communication channels. Such circuits read bits from memory and/or a communication channel, let the logic gates react to the read input to compute some new value, and write the computed bits into (possibly other parts of) memory or send them on to a (possibly different) communication channel. Long sequences of such actions can be constructed to perform arbitrarily complex operations and, in principle, everything electronic equipment can possibly do can be realized through such sequences.

The flexibility of a modern computer does, however, not come from fixed sequences of operations. Early on, it became clear that the exact operation of a computer could be coded into memory itself. A very complex circuit of logic gates could read an instruction coded into memory, perform one very specific action based on that instruction, and then move on to read the next instruction [10]. The semantics of each instruction would be as simple as *read the contents of register A and add them to the contents of register B*. Other instructions could apply to communications between the circuit and neighbouring circuits in the same system or they could be a

[2]Non-volatile memory is generally slower in operation; therefore all computers presently uses volatile memory for registers and other memory close to the core of the architecture.

message to the system that it should start reading instructions from another part of memory, so-called *jump* instructions.

An electronic circuit reading instructions from memory in this way is called a *processor* and the set of different instructions it is able to understand is called the *instruction set* of the processor. Each processor model has its own unique instruction set, which can be quite large; the number of transistors needed to implement these instructions is therefore also quite high. An example that illustrates some of the complexity is the 22-core Intel Xeon Broadwell E5 [6]. This is a CPU chip containing 22 processors and cache memory implemented by 7.2 billion transistors and operating at a clock switching speed of up to 3.8 GHz.

The instruction sets implemented by a processor form the bridge between the tangible hardware and the software that controls its behaviour. The divide between hardware and software is very distinct in ICT, in that its respective designs require different skill sets from the engineers. Also pertinent to our problem, it represents a divide in the world of security experts. Most security experts are concerned with weaknesses and attack vectors in software, often assuming that the hardware itself can always be trusted. If we do not trust the maker of the hardware, however, this assumption is completely broken. A vendor can include undocumented computer instructions known only to itself and let their execution be triggered at a predefined input signal of the vendor's choice. Documented instructions can have undocumented side effects that leak information from memory and send it on to a communication channel.

In principle, all the malicious functionality one can think of could be implemented in the hardware of a CPU. In particular, one should be aware how this influences the security provided by cryptographic software. Cryptography is the main building block of computer security. If cryptographic software runs on untrusted hardware, there is no reason to believe that the system is secure.

3.4 Firmware

Firmware is often described as something residing in between hardware and software. In essence, it is a piece of code that is loaded onto non-volatile memory and is read by the processing unit at start-up [7]. The firmware is what ultimately defines the instruction set and functionality of a processor.

The huge benefit of firmware is the added flexibility of a chip after its manufacture. Firmware can be used to correct mistakes in the chip design and to add new functionality after the hardware has been made. For instance, updates to firmware are used to add new codec formats to portable music players and other firmware updates have improved the power consumption of mobile phones.

What firmware means for the problem we address is that it constitutes another possible point for the introduction of malicious functionality. Firmware that changes the semantics of a processor's instruction set can clearly be used by untrusted vendors to carry out espionage and fraud, as well as sabotage. In our journey up the technology stack, this is the first and lowest point where we can see that a vendor can introduce unwanted functionality after the product has been purchased and received. A firmware update has all the conceivable potential to render your product unsafe; therefore, if your vendor is not to be trusted, firmware updates should be regarded with considerable suspicion.

3.5 Operating Systems, Device Drivers, Hardware Adaptation Layers, and Hypervisors

Starting from the instruction sets of the underlying hardware, the *operating system* [8] is a piece of software that builds higher-level abstractions and concepts that are closer to what a user or programmer will need. From basic physics that allow you to store and read a bit of information, it builds the concept of *blocks* and *files*. From the instruction set of the processor, it builds the notion of *process*. Some equipment will allow for multiple users and therefore the concept of *user* must be formed. In systems that will have multiple simultaneous processes, provisions must be made for the time-sharing and virtualization of the computer's resources. In addition, security mechanisms must be built into the operating system such that each process is isolated from the other processes, ensuring that they cannot overwrite each other's portions of memory, and that processes and users do not have access to information on the system that should be hidden from them.

Ideally, once you have created an operating system, you would like it to be able to run on several different versions of hardware. To accommodate this, a thin layer of software often runs directly on the hardware, making the hardware appear as if it were of a predefined generic type. These thin layers of software are called *hardware adaptation layers* for the core part of a computer and *device drivers* for peripheral equipment. In particular, the notion of a driver should be well known to most: whenever you connect external equipment to your computer, a driver for that piece of equipment needs to be installed.

A *hypervisor* [2] is a more recent notion. It stems from the need to be able to virtualize a computer so that it appears as several computers, allowing the same piece of equipment to run multiple operating systems at the same time. This is particularly useful for cloud services that offer platforms as a service. Disconnecting the instances of operating systems from the hardware they run on allows them to offer a number of software platforms to its customers that is not fixed to the number of instances of hardware it has installed. A hypervisor can run directly on the hardware (native, or type 1 hypervisor) or it can itself run as a process on top of another operating system (hosted, or type 2 hypervisor).

An untrusted maker of operating systems, device drivers, hardware adaptation layers, and hypervisors has optimal working conditions. If any of these components does not work, the entire system would most likely stop; these components are therefore ideal places for introducing kill switches. Second, since most of a computer's security mechanisms are placed in these components, it is the ideal place to introduce code that steals and leaks information. The fact that the operating system is a very complex and large piece of software makes it extremely hard to examine all its parts fully. As an illustration, it is assumed that most versions of the Windows operating system have several tens of millions of code lines.

3.6 Bytecode Interpreters

In the same way that device drivers and hardware adaptation layers render operating systems less dependent on the specifics of the hardware they run on, the intention of bytecode interpreters is to make the application independent of the operating system and platform. In essence, bytecode is a program written in a low-level but generic instruction set. Once the bytecode has been created for an application, this application can run on any combination of hardware and operating system for which a bytecode interpreter exists that can run it.

A plethora of different bytecode definitions exist, but the one most of us have been directly in touch with is that realized by the Java virtual machine [4]. It allows the same executable Java bytecode to run on a vast set of different platforms, from huge servers down to handheld devices.

Not all applications come in the form of bytecode, since many applications run directly on the operating system/hardware platform itself. This is indeed the case for the bytecode interpreters themselves. The widespread use of bytecode still urges us to consider how a dishonest bytecode interpreter creator could harm us. Such ways are quite easy to find, since the bytecode interpreter is in full control of everything an application does. It can therefore implement kill switches based on given inputs and leak any information that is made available to the application itself.

3.7 The Application on Top

All throughout transistors, logic gates, integrated circuits, printed circuit boards, hardware adaptation layers, and finally operating systems and drivers, the intention of the different fields of engineering is to build generic platforms. The intention of the complete electronic device itself is determined by the application running on top of this platform. The set of possible applications is exceptionally diverse and spans all conceivable uses of electronic equipment.

Many known cyberattacks go through an application to compromise a system. The attack can appear in the form of a webpage and go through the web browser or it can come through an email attachment. Still, when what we fear is wrongdoing on the part of the maker of the equipment, we do have some defences against the maker of applications. Since security mechanisms against third party attacks will usually be built into the operating system, applications must exploit weaknesses in the operating system to cause harm to the entire installation. If the operating system is secure, applications will be limited to leaking information that is accessible by the user running the application. This does not mean that untrusted applications are not dangerous; it only means that, as opposed to makers of operating systems, we have at least some level of defence against malicious developers of applications.

3.8 Infrastructures and Distributed Systems

The days of standalone electronic devices are long gone. Most devices are connected to the Internet and the applications they run are part of a distributed system. Two obvious examples known to everyone are the World Wide Web and e-mail. The truth is that the applications that are currently standalone and running on only one device are quite few. If nothing else, most devices are part of a distributed system that provides software updates over the net.

A distributed system is as complex from the operating system and up to the user interface as it is from the operating system down to the physical phenomena of a transistor. As an example, your e-mail client is built on top of at least three different layers of path-finding systems before your machine is able to talk to your mail server. Typically, these are the Ethernet protocol that finds the path between your PC and the gateway of your building [11], the interior gateway protocol that finds the path within the network of your Internet provider [9], and the border gateway protocol that finds the way between different Internet providers [5]. On top of this, a separate distributed system, called the domain name service, is required that translates e-mail addresses to the IP addresses of the mail server and a distributed system for authentication is needed that can secure the integrity of the mail accounts. Only when all of these components are in place can the mail system start working and the key insight is that all of these underlying components are themselves complex distributed systems.

In some cases, the same vendor will be responsible for making all the components of a distributed system. In other cases, the system is open, in the sense that new components can be added continuously and these new components can come from different vendors. Typical examples of the latter are e-mail systems and communication networks. In the former case, the harm that a malicious vendor could do is quite similar to what we described in the section above. All information trusted to the distributed system could leak and the system itself can be used to gain access to resources that should be secured by the operating system. In the latter case, we must assume that the product of the untrusted vendor is integrated into an already existing distributed system. This could be a new e-mail server or a new router that extends a country's critical network infrastructure. In the latter case, a new challenge arises, in that the new component could create chaos in the existing distributed system by behaving in unpredicted and malicious ways. A new router in a system could, for instance, sabotage the path-finding algorithm in the network and thus disconnect the entire network (Fig. 3.1).

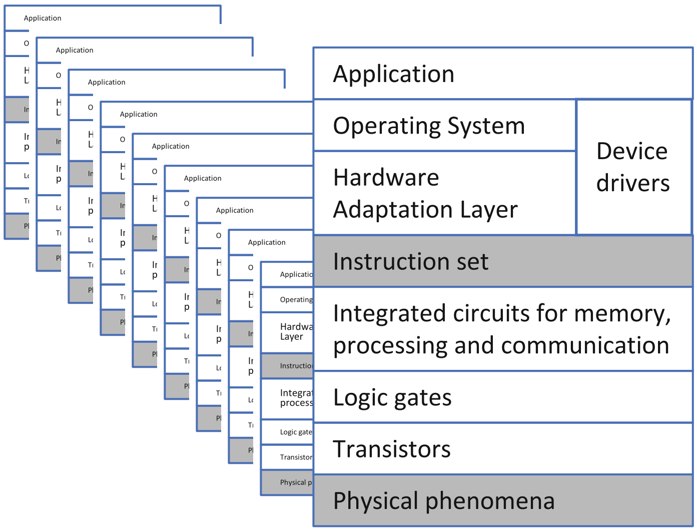

Fig. 3.1 Schematic overview of a distributed system. Each device builds instruction sets from physical phenomena through the hardware layers of transistors, logic gates, and integrated circuits. On top of the instruction sets, there are multiple layers of software before we reach the application. This technology stack can easily consist of billions of transistors and millions of lines of software code. Such a highly complex structure appears in all the – possibly hundreds of – devices that constitute the distributed system. Before a simple thing such as e-mail can work, several different distributed systems must be in place and work well. The domain name system, the interior gateway protocol, and the border gateway protocol are the three most obvious ones

3.9 Discussion

The takeaway from this chapter is that verifying an ICT system to ensure that it is not doing harm is a monumental task. In principle, it entails a careful study of all the software components of several distributed systems running on top of each other. Each of these systems can easily consist of tens of thousands to hundreds of thousands of lines of program code. For each of the possibly hundreds of devices running the code that is involved in these distributed systems, we would have to study the operating system, device drivers, and hardware adaptation layers that run on them. The operating system itself can consist of tens of millions of lines of code. If it runs on a virtualized platform, we would also need to include the hypervisor in the study.

All of the above would only touch the software side of the problem. Each device would consist of a number of chips integrated on a number of printed circuit boards. The CPU of the system alone will consist of tens of millions of logic gates and billions of transistors and, even if the CPU is probably be the most complex of the chips in the device, there will usually be more than a handful of other chips that need to be studied.

Most buyers of electronic equipment are only exposed to parts of this problem. The decision to buy electronic devices will rarely encompass all the infrastructure necessary to make the device work. If you do not trust the maker of a mobile phone, you will only need to investigate that device, because the network it connects to is not under your control. If, on the other hand, you are the network provider, you have control over what equipment you buy and use in your network but not over the equipment that your customers might connect to it or the equipment of the network provider you peer with to provide connectivity. Even under these limitations, the task of securing against dishonest makers of electronic equipment spans the physical phenomena underpinning the transistors of the system all the way up to the software running the user interface of the application. In large parts of this book, we will be concerned with ways of approaching this problem to understand if it is at all doable. Before that, in the next chapter, we will discuss the design process of ICT equipment to understand how a dishonest vendor could introduce malicious functionality into a product.

References

1. Bardeen, J., Brattain, W.H.: The transistor, a semi-conductor triode. Phys. Rev. **74**(2), 230 (1948)
2. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. ACM **29** (1995)
3. Copeland, B.J.: Colossus: The Secrets of Bletchley Park's Code-Breaking Computers. Oxford University Press, Oxford (2010)
4. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification: Java SE 8 Edition. Pearson Education, London (2014)
5. Lougheed, K., Rekhter, Y.: Border gateway protocol 3 (bgp-3). Technical report (1991)
6. Nalamalpu, A., Kurd, N., Deval, A., Mozak, C., Douglas, J., Khanna, A., Paillet, F., Schrom, G., Phelps, B.: Broadwell: a family of ia 14 nm processors. In: 2015 Symposium on VLSI Circuits (VLSI Circuits), pp. C314–C315. IEEE (2015)
7. Opler, A.: 4th generation software. Datamation **13**(1), 22–24 (1967)
8. Peterson, J.L., Silberschatz, A.: Operating System Concepts, vol. 2. Addison-Wesley, Reading (1985)
9. Rekhter, J.: NSFNET backbone SPF based interior gateway protocol. Database **4**, 5 (1988)
10. Rojas, R.: Konrad Zuse's legacy: the architecture of the Z1 and Z3. IEEE Ann. Hist. Comput. **19**(2), 5–16 (1997)
11. Seifert, R.: Gigabit Ethernet: Technology and Applications for High-Speed LANs. Addison-Wesley Longman Publishing Co. Inc., Reading (1998)

Chapter 4
Development of ICT Systems

An example from 2015 illustrates how compilers can be used to spread malware. Xcode is Apple's development tool for iOS applications. Attackers added infectious malware to Xcode and uploaded the modified version to a Chinese file-sharing service. Chinese iOS developers downloaded the malicious version of Xcode, compiled iOS applications with it and inadvertently created infected executables, and then distributed these infected executables through Apple's App Store [9]. This technique has allegedly long been known to the CIA [5], who has been claimed to have exploited Xcode to add malware to iOS applications.

In this chapter, we consider the processes behind the production and maintenance of information and communications technology (ICT) equipment. We discuss how hardware and executable software running on this hardware are produced and maintained. Our discussion is not related to the traditional fields of software or hardware engineering. Rather, we look at the process from a toolchain point of view, to understand from which vantage points a vendor can introduce hidden functionality into the equipment. Finally, we discuss the advantages, from the perpetrator's point of view, of using development and production line tools to include malicious functionality into a product and we clarify the implications of this for the equipment buyer.

4.1 Software Development

The code language running on a device is far from being easily understood by humans. We therefore distinguish between the *source code* and the *executable code*. The source code of a program is written in a *programming language* that is designed to be humanly understandable. The source code for the program needs to be translated into *executable code* before it is ready to be executed on a device. This translation is carried out by a *compiler*. The process of producing the code that actually runs on a device is illustrated by the three boxes at the bottom of Fig. 4.1.

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_4

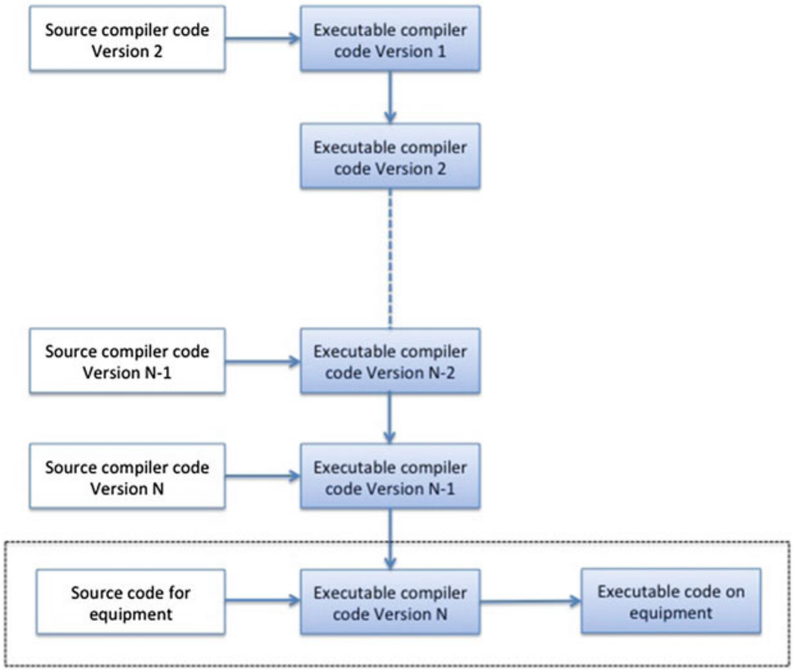
Source compiler code Version 2
Executable compiler code Version 1
Executable compiler code Version 2
Source compiler code Version N-1
Executable compiler code Version N-2
Source compiler code Version N
Executable compiler code Version N-1
Source code for equipment
Executable compiler code Version N
Executable code on equipment

Fig. 4.1 The structure of compiler code and equipment source code that eventually make up the executable running on the equipment. The first compiler is programmed directly in executable code. Version 2 of the compiler is programmed in a high-level language defined by the compiler. Version 1 of the compiler is then used to create an executable form of version 2 of the compiler. This iterative process is repeated for each new compiler generation. Thompson demonstrated how malware inserted into any historic version of the compiler can survive forever down the development chain and eventually result in backdoors inserted into present-day executable code by present-day compilers

The compiler that produces the executable code is itself a program. It was written in a programming language and was itself compiled by a compiler. That compiler was, in turn, compiled by another compiler and, thus, the full structure of programs, source code, and executable code that leads to the executable of a product is quite complex. A simplified picture of the dependencies is given in Fig. 4.1. Many aspects are left out of this figure: We have not shown how this structure allows new programming languages to emerge, we have not considered how compilers for programs running on new hardware architectures are built, and we have not considered how pre-compiled libraries or elements of an operating system would interact with the final executable. For our discussion, it suffices to know that, for most modern programs, the dependency chain backwards to older compilers, all the way back to the first compiler that was written in executable code, is quite long and, for some very common programming languages, can be traced back to the 1970s [4].

It is generally accepted that executable code is humanly understandable only with a tremendous amount of effort (see Chap. 6 for more details). The source code of a program is therefore at the centre of many discussions. It is usual for companies to release only the executable code of their products to ensure that their ideas are not copied by competitors. Open-source discussions and initiatives are other examples illustrating the absence of transparency in executable code. Equipment vendors have therefore occasionally countered customer worries by making the source code of their products available.

Given the above, the following question presents itself: Where in the process of Fig. 4.1 can malicious code be inserted so that the intended malicious functionality is part of the final executable? It is obvious that such code can be part of the source code of the product itself, but can it be placed in the compiler without altering the source code of the product? The answer is given by Ken Thompson [11] in his Turing Award lecture in 1983. In it, he gave a very simple example of how a Trojan horse can be inserted into the end executable code of a product through the source code of any single one of the compiler versions that, at some time in the past, played a part in producing the final executable.

What Thomson demonstrated is how one could alter the C compiler so that it introduces a backdoor into the UNIX operating system whenever the operating system is compiled. Furthermore, he showed how one could code this property into a ‘gene’ so that any later version of the C compiler would inherit this capability. Finally, he showed how to remove any trace that this had happened from the source code of both the compiler and the operating system. The backdoor would be inserted into any future version of UNIX by any future version of the C compiler and neither the UNIX developers nor future developers of the C compilers would ever know.

The insights of Thompson are well known in the scientific community. Still, as pointed out by [4], such methods are not considered a real threat to the security of computer systems. The danger of someone using a compiler to inject backdoors to break into some system yet to be made has been considered unlikely. If we put ourselves in the position of a company that wishes to include backdoors into its products without being caught, the discoveries of Thompson can be viewed from a different angle. Such a company would like to leave as few traces as possible of this ever having happened. Using the compiler to introduce backdoors would leave no trace in the source code, so the code can safely be given to any suspicious customer. Furthermore, such a company would benefit from keeping knowledge of the secret confined to as small a group of people as possible to minimize the risk of information on their actions leaking out. Altering compiler tools means that the development team itself need not know that the backdoors are being introduced.

From the above discussion, we now draw the following conclusions:

- The absence of malicious elements from the source code of a software product does not prove that such elements do not exist in the executable code.
- If a vendor wants to install malicious code in a product, it is not necessary for the development team to be aware of this. The malicious code can be installed in the compiler tools that the developers are instructed to use.

4.2 Hardware Development

Putting malicious functionality into hardware can be very effective in gaining control over an entire system. Furthermore, it could require very little space on the integrated circuit. As an example, it has been demonstrated that a backdoor can be inserted into an entire system by adding as few as 1,341 additional gates to a chip [7]. These additional gates are used to check the checksum of an IP packet and, given the right checksum, it would install the packet's payload as new firmware on the processor. The firmware could, in principle, do anything, but what is described in the paper by King [7] is an attack where the installed firmware gives a particular username login root access to the system.

The process for the development of integrated circuits is somewhat different from that of software. Still, there are similarities, in the sense that hardware can be defined through a high-level language, where the details of gates and transistors are abstracted away. The transition from descriptions understandable by humans into a physical chip coming off a production line will go through a process that can roughly be described as follows:

1. An algorithmic description of the desired behaviour (usually specified in a dialect of the C programming language) is synthesized into a register-transfer level (RTL) hardware design language by a *high-level synthesis tool* [8].[1]
2. The RTL is translated into a gate-level description of the chip by a *logic synthesis tool* [10].
3. The gate-level description is used by production lines to produce the actual chip.

Again, we agree that our description is vastly simplified. We have not considered difficulties related to, for example, the layout or thermal issues. Still, the model is sufficient for the conclusions we need to draw.

In the case of integrated circuits, malicious functionality can, of course, be placed directly into the RTL or the algorithmic description of the chip. Learning from Sect. 4.1 above, we need to also consider the synthesis tools that translate the hardware descriptions from languages that are humanly understandable and down to gate-level descriptions. The important observation in this case is, however, that all of the synthesis tools are themselves pieces of software. They are therefore subject to the same considerations made at the end of the previous section. Knowing that as few as 1,341 extra gates on a chip can leave an entire system wide open, it is easy to see that any tool involved in the making of a chip can easily insert serious malicious functionality into it. Such functionality could even be inserted by the production line that produces the final chip from the gate-level descriptions [1, 3, 6]. We are therefore forced to make similar conclusions for hardware as we did for software:

- The absence of malicious elements from the source code of a hardware product does not prove that such elements do not exist in the chip.

[1]Sometimes the circuit is described directly in RTL. In those cases, step 1 is omitted.

- If a vendor wants to install malicious code in a chip, it is not necessary for the development team to be aware of this. The malicious functionality can be installed in the synthesis tools that the developers use.

4.3 Security Updates and Maintenance

An electronic device consists of two parts: one is the hardware that the customer can touch and feel and the other is the software that guides the device to behave as intended.[2] Whereas the hardware of a device can be considered fixed at the time of purchase, the software of the device is generally updated and changed several times during the lifetime of the device.

The reasons for updating the software running on network equipment can be the following:

1. There could be a bug in the device that needs to be fixed.
2. New and optimized code could have been developed that increases the performance of the device.
3. New functionality that was defined after the purchase of the device, for example, new protocol standards, needs to be supported.
4. New security threats have emerged, creating the need for new protection mechanisms in the equipment.

In particular, points 3 and 4 in the above list make software updates inevitable. The requirement that equipment support new protocols that will be defined after the time of purchase will prevail for the foreseeable future. Furthermore, the security threats that the equipment must handle will continuously take on new forms.

Software updates come in different sizes, depending on what part of the software is updated. Device driver updates will have different implications than an update of the operating system of the entire device. For our discussion, we do not need to analyse these differences in more depth. It suffices to observe that changes to the operating system and the device drivers will be required from time to time.

Obviously, since software updates are made in software, they inherit the same conclusions as those we drew in Sect. 4.1. In addition to these conclusions, we can draw another one from the discussion in this section:

- Malicious elements in the software code on a device can be introduced through software updates at any time in the life cycle of the device.
- If a vendor, at some point in time, wants to install malicious code in a sold device through a software update, it is not necessary that the development team of the update be aware of this. The malicious code can be installed in the compiler tools that developers use.

[2]We have chosen not to include the notion of firmware. In our discussions, we classify firmware as a particular kind of software because the firmware of a device can be replaced and modified without replacing any physical component.

4.4 Discussion

Protecting an ICT infrastructure from attacks and data leakage is a daunting task. We are currently witnessing an arms race between the developers of equipment and intruders in which ever more sophisticated attacks are designed and need to be countered with ever more sophisticated defence mechanisms. When we assume that the perpetrators could be the designers of the equipment themselves, the issue looks entirely different. The problem is new and no real arms race has started. The only technical discussion of the matter we are aware of is that where Huawei suggested making its software source code available to its customers [2] in response to doubts over whether it could be trusted.

There is every reason to applaud the effort made by Huawei to build trust through openness. Still, there are two separate and independent reasons why giving access to the software source code is nowhere near giving insight into the true ability of an electronic system. First, it has been demonstrated beyond any doubt that full backdoors into a system can be created with a microscopic number of additional gates on a chip. A backdoor existing in the hardware will clearly not be visible in the software running on top of it. Second, the code running on a machine is not the source code, but instead some executable code that was generated from the source code by a compiler. Again, it has been demonstrated beyond any doubt that backdoors can be introduced into the executable code by the compiler and, thus, not even a software-based backdoor need be visible in the source code.

Figure 4.2 shows a schematic of an integrated electronic product and parts of the production line leading up to it. The yellow boxes represent the product itself, coarsely divided up into one hardware layer and three software layers. The green boxes are representations of parts of the final product that are visible to the engineers building it. These are the source code of the software and the high-level representations of the hardware. The blue boxes represent the tools that transform the source code representations into the finished product. As we have seen above, a backdoor can be

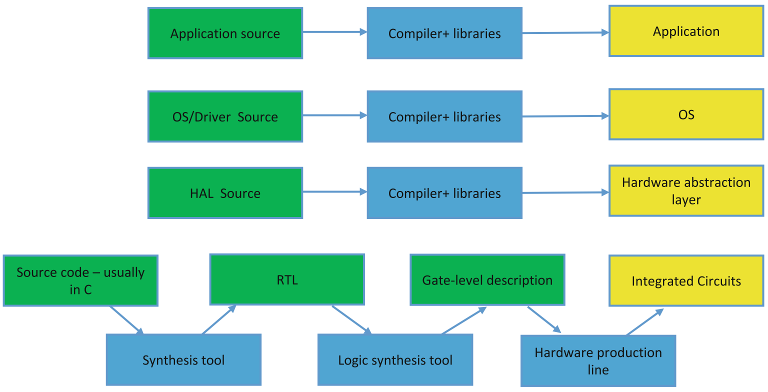

Fig. 4.2 Schematic overview of the elements that contribute to the final integrated product

inserted by any tool used in the process, so any blue box is a potential point of attack. To fathom the full complexity of the design process, we also need to understand that all of the tools in this chain – all of the blue boxes – are themselves electronic products that are built in exactly the same way, with source code, compilers, synthesis tools, logic synthesis tools, and a hardware production line. The recursion of tools that is implied by this observation is illustrated for compilers alone in Fig. 4.1.

Defence against an untrusted equipment maker should therefore focus on the actual hardware that has been purchased and on the actual machine code running on the system. It is depressingly simple for a dishonest equipment provider to introduce unwanted functionality through the development tools and for the dishonest provider this approach has the clear advantage that very few people in the company need to know about it. Looking at source code and verifying the toolchain itself is most likely futile; we will probably end up in a recursive pit consisting of tools building tools. For compilers alone, we are easily faced with a sequence of tools going all the way back to the early days of computing.

References

1. Adee, S.: The hunt for the kill switch. IEEE Spectr. **45**(5), 34–39 (2008)
2. BBC: Huawei offers access to source code and equipment. http://www.bbc.com/news/business-20053511
3. Bhunia, S., Hsiao, M.S., Banga, M., Narasimhan, S.: Hardware trojan attacks: threat analysis and countermeasures. Proc. IEEE **102**(8), 1229–1247 (2014)
4. Danezis, G.: Trust as a methodological tool in security engineering. Trust, Computing, and Society, vol. 68 (2014)
5. The Intercept: CIA campaign steal apples secrets. https://theintercept.com/2015/03/10/ispy-cia-campaign-steal-apples-secrets/
6. Karri, R., Rajendran, J., Rosenfeld, K.: Trojan taxonomy. Introduction to Hardware Security and Trust, pp. 325–338. Springer, Berlin (2012)
7. King, S.T., Tucek, J., Cozzie, A., Grier, C., Jiang, W., Zhou, Y.: Designing and implementing malicious hardware
8. McFarland, M.C., Parker, A.C., Camposano, R.: The high-level synthesis of digital systems. Proc. IEEE **78**(2), 301–318 (1990)
9. Reuters: Apple's iOS app store suffers first major attack. http://www.reuters.com/article/us-apple-china-malware-iduskcn0rk0zb20150920
10. Riesgo, T., Torroja, Y., de la Torre, E.: Design methodologies based on hardware description languages. IEEE Trans. Ind. Electron. **46**(1), 3–12 (1999)
11. Thompson, K.: Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)

Chapter 5
Theoretical Foundation

What computers can and cannot do has been a long-standing topic in the foundation of computer science. Some of the pioneers of the field had a strong background in mathematics and, in the early days of computing, worked on the mathematical formulation of the limits of computation. The work led to the notion of decidability. Informally speaking, a question that can be answered by either yes or no is decidable if a computer can compute the correct answer in a finite amount of time.

The relation that the notion of decidability has to our problem of vendor trust should be obvious. If the question of whether an executable program performs malicious acts is decidable, we can hope to devise a program to check the code made by an untrusted vendor. If it is known to be undecidable, this conclusion should impact on where to invest our efforts. In this chapter, we review and explain some of the key results on decidability and explain how these results impact the problem of untrusted equipment vendors.

5.1 Gödel and the Liar's Paradox

The easiest accessible pathway into decidability is through the *liar's paradox*. Although we intuitively think that a statement is either true or false, it is possible to make an expression that is inconsistent if it is true and equally absurd if it is false. The liar's paradox is such an example: consider the statement, 'This statement is false.' If it is true, then it has to be false and, if it is false, then it has to be true; thus it can be neither true nor false.

The liar's paradox has been subject to long philosophical discussions throughout history. Its first application of relevance to our case was by the logician and mathematician Kurt Gödel [6]. Gödel used a slightly modified version of the liar's paradox to prove his first incompleteness theorem. This theorem states that no theory with a countable number of theorems is able to prove all truths about the relation of natural numbers. Roughly, what Gödel did was replace the statement ‘This statement is false’ with ‘This statement is not provable’. Clearly, if the latter statement is false, it has to be provable, meaning that it has to be true. This situation again implies that the statement has to be true but not provably so.

The incompleteness theorem of Gödel is not of direct interest to us, but his proof technique is. At the core of it lies the observation that there are limits to what a statement can say about itself without becoming an absurdity. Our interest in this question is as follows: we would like to understand what a program can say about a program or, more precisely, if a program can decide whether another program will behave maliciously.

5.2 Turing and the Halting Problem

Before we dive into the limits of what a computer can do, we need to have a firm understanding of what a computer is. The most common way to model the concept of a computer is through an abstract device described by Alan Turing [10].

There are several manifestations of this device. What they have in common is that they consist of a finite state machine, a read/writable tape, and a read/write head that is located over a position on the tape. The device operates by reading the position on the tape, changing the state of the state machine depending on what it read, writing a symbol to the tape depending on the state and what it read, and moving the tape forward or backward depending on the state and what it read. A more formal definition, similar to the definition given by Cohen [5], is as follows:

- Σ is a finite set of states for the Turing machine.
- Γ is a finite set of symbols that can be read from and written to the tape.
- Ω is a function $\Sigma \times \Gamma \rightarrow \Gamma$ that decides the symbol to be written to tape in each computation step.
- Δ is a function $\Sigma \times \Gamma \rightarrow \{-1, 0, 1\}$ that decides in which direction the tape should move after having written a symbol to the tape.
- Π is a function $\Sigma \times \Gamma \rightarrow \Sigma$ that decides the state that the Turing machine enters after having written a symbol to the tape.

The Turing machine starts with a tape with symbols on it and executes its operation by performing the functions defined by Ω, Δ, and Π in each execution step. In this definition, the computation of the machine halts when a computation step changes neither the symbol on the tape, the position of the tape, nor the state of the state machine (Fig. 5.1).

A famous postulate is that every function that is computable by any machine is computable by a Turing machine. Although this postulate has not been proven, it is widely believed to be true. In the years since its formulation, there has been no reasonable description of a machine that has been proven able to do computations that cannot be simulated by a Turing machine. Consequently, the strength of programming languages and computational concepts is often measured by their ability to simulate a

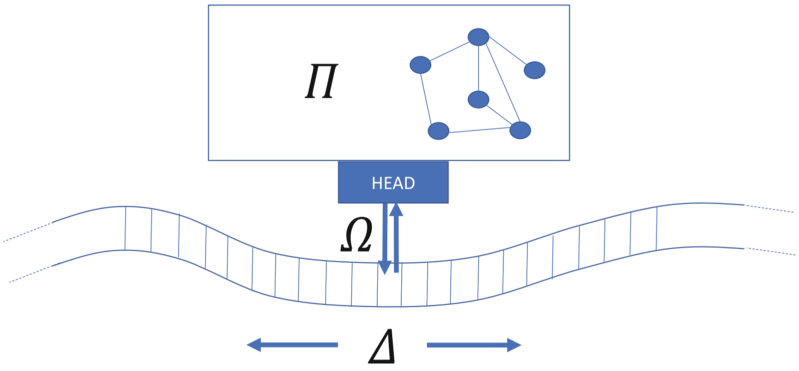

Fig. 5.1 A Turing machine. Based on the state of the state machine and the symbol on the tape, Ω decides the symbol to be written, Δ decides how the tape should be moved, and Π decides the new state of the machine

Turing machine. If they are able to, then we can assume that anything programmable can be programmed by the language or the concept.

The definition of a Turing machine and the postulation that it can simulate any other computing machine is interesting in itself. Still, its most important property, from our point of view, is that it can help us understand the limits of what a computer can do. Turing himself was the first to consider the limitations of computations, in that he proved that a computer is unable to decide if a given Turing machine terminates by reaching a halt. This is famously known as the halting problem and, to prove it, Turing used the liar's paradox in much the same way as Gödel did.

In a modernized form, we can state the proof as follows: assume that we have a programmed function P that, for any program U, is able to answer if U halts for all inputs. This would mean that $P(U)$ returns a value of *true* if U halts for all inputs and *false* if there is an input for which U does not halt. Then we could write the following program Q:

$$Q : \text{if } P(Q) \text{ then loop forever; else exit}$$

This program is a manifestation of the liar's paradox. If Q halts, then it will loop forever and, if it does not halt, then it halts. The only possible explanation for this is that our assumption that P exists was wrong. Therefore, no P can exist that is able to decide the halting problem.

5.3 Decidability of Malicious Behaviour

The importance of understanding the work of Gödel and Turing is that it forms the basis for a series of results on the analysis of what a piece of code actually does. This basis was exploited by Cohen's [5] Ph.D. thesis from 1985. Cohen first defined the main characteristic of a computer virus to be its ability to spread. Then the author assumed the existence of a programmed function P with the ability to decide for any program U whether it spreads. Using the liar's paradox in the same way as Gödel and Turing, Cohen found that the following example code constituted an absurdity:

$$Q : \text{if } P(Q) \text{ then exit; else spread}$$

The reasoning is the same. If Q spreads, then it exits without spreading. If Q does not spread, then it spreads. Again, the only consistent explanation is that P does not exist.

Cohen's work was a breakthrough in the understanding of the limits of looking for malicious code. Subsequently, many developments extended undecidability into other areas of computer security. The most important basis for this development was the insight that the automatic detection of malicious machine code requires code with the ability to analyse code. By using the liar's paradox, we can easily generate absurdities similar to those described above. A general variant would be the following: Assume that P is a program that detects *any* specific behaviour B specified in any program code. Then we can write the following program:

$$Q : \ \text{if } P(Q) \text{ then exit; else behave according to } B$$

In the same way as seen before, this is a manifestation of the liar's paradox and the only conclusion we can draw is that P does not exist.

Note that the latter example is entirely general, in that it does not specify what behaviour B is. This means that, by using different instantiations of B, we can conclude the following:

- It is undecidable if computer code contains a Trojan [4] (B defines the behaviour of a Trojan).
- It is undecidable if computer code contains the unpack–execute functionality often found in Trojans [8] (B defines the behaviour of an unpacker).
- It is undecidable if computer code contains trigger-based malware, that is, malicious computer code that will execute only through an external stimulus [2] (B defines the behaviour of trigger-based code).
- It is undecidable if computer code is an obfuscated version of some well-known malware behaviour [1] (B contains a specification of how the well-known malware behaves).

We will return to these bullet points in Chaps. 7 and 8, where the craft of malware detection is discussed in more detail.

5.4 Is There Still Hope?

The theoretical observations that we have elaborated on above appear to extinguish all hope that we can solve security problems in computer code. This is, of course, not true, since there is a lively and vibrant computer security industry out there accomplishing valuable work. In the remainder of this chapter, we take a look at where the rays of light are that are exploited by this industry. First, however, we make things even darker by correcting a common misunderstanding regarding the undecidability proofs above.

All the proofs above were based on the liar's paradox, meaning that a central part of the proof is that it is impossible to decide which branch of the following program will actually be followed:

$$Q: \text{ if } P(Q) \text{ then exit; else do something bad}$$

This situation has led to the very common misunderstanding that the only undecidable question is whether the code that does something bad is actually executed. Clearly, any detection method that found badly behaving code – regardless of whether it would be executed – would be of great help.

Unfortunately, identifying badly behaving subparts of code is also undecidable and the proof is very similar to those we cited above. A common misunderstanding stems from mixing up two proof concepts: proof by counterexample and *reductio ad absurdum*. The program Q above is not a counterexample in the sense that it is a program for which all P will have to give the wrong answer. Rather, Q is an absurdity that is implied by the existence of P and, consequently, P cannot exist. The proof itself sheds no light on for what subparts or subsets of programs P would actually exist.

A ray of light is to be found in the definition of the Turing machine itself. This is a theoretically defined machine that is not constrained by the realities of the physical world. Where a real-world machine can only work on a limited amount of memory, a Turing machine can have an infinitely long tape. Based on this insight, we have seen some limited results that identify decidable versions of the problems studied above. The existence of certain malware unpacking behaviour in code is shown to be decidable and NP-complete [3]. A similar result exists, where, under some restrictions, the decision of whether some code is an obfuscated version of another is proven to be NP-complete [1]. Additionally, for the detection of viruses, a restricted version of the problem is NP-complete [9].

However, NP-completeness is still a major problem. It does imply the existence of a system that, in finite time, can produce the right answer, but the amount of time needed rises very steeply with the size of the investigated program. For real-life program sizes, 'finite time' means 'finite but not less than a thousand years'. The practical difference between undecidable and NP-complete for our problem is therefore not significant. Still, the future may have further developments along this axis that will help. Work on identifying decidable formulations of our problem therefore remains important.

A more promising possibility is to be found in the absoluteness in the definition of decidability. The proofs based on the liar's paradox hold as long as we require that *P* have no false negatives or false positives. In particular, when looking for malware injected by a vendor, we may be able to accept a relative high ratio of false positives, as long as there are few false negatives. Most would be happy to require a purchased system to be designed so that it tested negatively, if one could assume that deliberately building a system for a false negative was hard.

5.5 Where Does This Lead Us?

It has been stated [7] that there will never be a general test to decide whether a piece of software performs malicious acts. Above we have gone through the reasoning that substantiates this claim, which urges us to ask what options remain. The answer lies in the observation at the end of the previous section. We must aim to organize processes, mechanisms, and human expertise for investigating equipment such that deliberately building equipment that would generate a false negative in the investigation is hard. In other words, any vendor that deliberately inserts unwanted malicious functionality into its products should run a high risk of being caught.

This places our problem in the same category as most other sciences related to security. It is usually impossible to guarantee that security will never be breached, but one can make it difficult to the extent that it rarely happens. This is the case in aviation, in finance, and in traditional computer security. We must therefore understand what it means to make it difficult to build malicious functionality into a system without being caught.

Several fields of research have the potential to help. First and perhaps most obvious, we have all the research that has been done in malware detection. Although most of the work in that field is based on the assumption that the perpetrator is a third party and not the vendor, the problem it addresses is close to ours. We study the applicability of malware detection techniques to our problem in Chap. 7. Then we study how developments in formal methods can help us. The aim of formal methods is to build formal proofs of the properties of computer systems and we thus consider how that can help us in Chap. 9. Adding to this, Chap. 8 examines how the systematic testing of a computer system can help us make it difficult to include malicious behaviour into a system without being caught. Looking for machine code that can solve the problem once and for all is, unfortunately, futile.

References

1. Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008)
2. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. Botnet Detection, pp. 65–88. Springer, New York (2008)
3. Bueno, D., Compton, K.J., Sakallah, K.A., Bailey, M.: Detecting traditional packers, decisively. Research in Attacks, Intrusions, and Defenses, pp. 184–203. Springer, Berlin (2013)
4. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy, pp. 32–46. IEEE (2005)
5. Cohen, F.: Computer viruses. Ph.D. thesis, University of Southern California (1985)
6. Gödel, K.: Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für mathematik und physik **38**(1), 173–198 (1931)
7. Oppliger, R., Rytz, R.: Does trusted computing remedy computer security problems? IEEE Secur. Priv. **2**, 16–19 (2005)
8. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: automating the hidden-code extraction of unpack-executing malware. In: Null, pp. 289–300. IEEE (2006)
9. Spinellis, D.: Reliable identification of bounded-length viruses is np-complete. IEEE Trans. Inf. Theory **49**(1), 280–284 (2003)
10. Turing, A.: On computable numbers, with an application to the entscheidungs problem. Proc. Lond. Math. Soc. **42**, 230–265 (1936–1937)

Chapter 6
Reverse Engineering of Code

The ability to reverse engineer a product has been important for as long as technology has existed. A vital activity in most branches of industrial design and production has been to acquire samples of the products sold by competing companies and pick them apart. Understanding the engineering done by your competing opponents can shed insight into the strengths and weaknesses of their products, reveal the engineering ideas behind their products' features, and fertilize and further improve the innovation that goes on in one's own company.

Within information and communications technology (ICT), reverse engineering has played a lesser role than it has in traditional industries. The main reason for this is that reverse engineering a product for the sake of copying it as your own is often considered too costly and time-consuming to be worth the effort. Still, reverse engineering is used and studied for a handful of select purposes in ICT. In this chapter, we provide an overview of the state of the art in this area and conclude how it relates to our ability to verify the contents of a product from an untrusted vendor.

6.1 Application of Reverse Engineering in ICT

The most common form of reverse engineering in computer programming and engineering is deeply rooted in the everyday tasks of the engineer. The making of a program or a piece of electronic equipment largely consists of interactions with libraries and components that one has not made oneself. Everybody who has worked as an engineer in ICT will know that the interfaces of such libraries and components are hard to understand and often lack the necessary documentation. Reverse engineering the interfaces of the components you need is therefore something that every ICT engineer will have spent time on [6].

Reverse engineering for the purpose of understanding interfaces is largely uncontroversial. The reasons for doing it are clear and they are generally compatible with the interests of society. An exception to this is when a company intentionally keeps its user interface secret for commercial reasons or for reasons related to security.

Whether reverse engineering that interface is acceptable then becomes a legal as well as a moral question. In 1990, Sega Enterprises released a gaming console called Genesis. Its strategy was to let Sega and its licensed affiliates be the only developers of games for it. A California-based company called Accolade reverse engineered the interface of the Genesis gaming platform and successfully developed and sold games for it. In 1991, Accolade was sued by Sega for copyright infringement. The court ruled in Accolade's favour because it had not copied any of Sega's code and because of the public benefit of the additional competition in the market that Accolade represented [2]. Today – several decades later – secrecy regarding interfaces and challenges of such secrecy through reverse engineering still take place. The jailbreaking of mobile phones bears witness to this phenomenon [12].

Another area where reverse engineering is applied is in the search for security holes. The reason for wanting to do this is much the same as for needing to reverse the interface of a component you require. When you include a component in your product, you also expose yourself to the component's security vulnerabilities. Reverse engineering the component will therefore be necessary to understand the extent to which such vulnerabilities exist and to assess the degree to which they are inherited in your own product [4]. This is particularly important for components that implement cryptographic security protocols. These protocols are usually mathematically sound but the security they provide is very sensitive to subtle mistakes or shortcuts in implementation.

The emergence of software reverse engineering as a field in its own right largely came about as the result of the need to analyse malware. Whenever a system is infected, there is a need to identify the malware involved, how it infects the system, how it spreads, and what damage it can do or has already done. There is therefore a high demand for expertise in reverse engineering different types of malware and conducting forensic analyses of infected systems [10].

The most controversial use of reverse engineering is related to digital rights management (DRM). The age of digital equipment has turned copyrighted material such as recorded music, books, and movies into digital information. On one hand, it has become trivially easy to copy, distribute, and share such material and, on the other hand, it has become increasingly hard for copyright owners to protect their property. The term DRM is used to denote technologies invented for the purpose of protecting copyright holders while still making copyrighted material easily accessible to those consumers who paid for the right to enjoy it [8]. Reverse engineering of DRM protection schemes will allow the cracker to gain unprotected access to the material. It is therefore regularly done by software pirates, who, in many cases, will make cracked material available on file-sharing sites. The material then becomes downloadable for free for anyone who chooses to do so and the value of the copyright diminishes accordingly.

Even though reverse engineering is more complex and therefore less used in ICT than in other fields of engineering, it has developed into an area in which a computer scientist can specialize. In the upcoming sections, we discuss some of the tools that have been developed in this area. Understanding the tools of the trade will help us understand the state of the art. Ultimately, this will help us assess the extent to which it is possible to fully investigate a product bought from an untrusted vender.

6.2 Static Code Analysis

In Chap. 3 we discussed the different layers of technology that constitute an ICT-system. These layers span from the user interface of a system and all the way down to the physical phenomena that allow us to build them in the first place. Then, in Chap. 4 we illustrated that many of these layers are hidden from the system developers themselves. Tools for hardware synthesis, compilers and assemblers makes the development process more efficient by relieving the engineers from having to relate to many of the technology layers.

A reverse engineer looking for malicious functionality inserted by an untrusted vendor will have to study all of the technology layers, as well as the interaction between them. The software-part of this technology stack starts from machine-code at the bottom. Here, none of the high-level notions known to programmers are present. Concepts such as variables, arrays, structures, objects, sets, list, trees, graphs, methods and procedures are not present. Rather, there are memory locations with a fixed number of bits in them, a clear distinction between registers close to the CPU and memory-locations in caches and off-chip, and no obvious distinction between data, pointers to memory-locations and instructions.

Since reverse engineering of software largely consists of recreating the intentions and thoughts of the initial programmer, the reverse engineer will have to backtrack from the machine code and towards the high-level program code that was originally written. This is a huge challenge. In its pure form, machine code loaded into memory can easily consist of millions of memory locations, all containing 32 or 64 bits of information. This has to be translated back – first into assembly code and then, possibly, into an interpreter of byte-level code – before the high-level concepts used by the original programmer can be recreated.

To most programmers inexperienced in reverse engineering, this sounds undoable. All software engineers have experienced not understanding high-level code written by their colleagues and most will admit to cases in which they did not understand code they wrote themselves two months ago. Understanding code starting from machine-level instructions seems like an impossible task. Still, reverse engineering has celebrated significant successes in deciphering programmer interfaces, malware analysis, as well as cracking DRM schemes. This is largely due to the tool sets available. In the upcoming sections, we review the most important classes of reverse engineering tools.

6.3 Disassemblers

A disassembler is a relatively simple piece of software. It takes a sequence of machine code instructions encoded in binary format readable by the machine and translates the instructions one by one into the more humanly readable textual form of assembly code. Since the instruction set differs from platform to platform, a disassembler is generally platform specific [5].

Although disassemblers vary in strength, it is generally agreed that automatically recreating readable assembly code from machine code is doable [6]. In our case, however, it should be noted that we do not necessarily trust the platform either. This means that the hardware and firmware could implement undocumented side effects and undocumented instructions that will not be correctly interpreted by the disassembler.

6.4 Decompilers

A decompiler is a piece of software that does the opposite of what a compiler does. This means that it tries to recreate the original source code by analysing the executable binary file. This is a very difficult problem. In all but a very few platforms, actual recovery of the full original source code with comments and variable names is impossible. Most decompilers are complete in the sense that they construct source code that, if recompiled, will be functionally equivalent to the original program. Still, the results they produce may be extremely hard to understand and that is where the limitation of decompilers lies with respect to reverse engineering.

There are several reasons why understanding decompiled code is far harder than understanding the original source code. First, in the compilation process, much of the information that the programmer writes into the code to make it readable is removed. Clearly, all comments to the code are removed. Furthermore, no variable names will survive, since they are translated into memory locations in machine code. High-level concepts such as classes, objects, arrays, lists, and sets will not be readily recreated. The programmer's structuring of the code into methods and procedures may be removed by the compiler and calls for these procedures may have been replaced by copying the code in-line. Furthermore, the compiler may create new procedures through the observation of repeated code and the flow graph of the machine code may be very different from that of the original source code. In addition, all assignments containing arithmetical expressions will be replaced by the compiler by a highly optimized sequence of operations that renders the original assignment statement impossible to recreate [1].

These difficulties are such that some compare decompilation with the process of trying to bring back eggs from an omelette or the cow from a hamburger. This is true to the extent that the output from the decompiler will, in most cases, be far less readable than the original source code. On the other hand, it is important to note that all the information that makes the program do what it does will be present in the decompiled code as well. Decompilers are therefore valuable tools in most reverse engineering processes [6].

6.5 Debuggers

Whereas disassemblers and decompilers are tools that work on static program code, debuggers operate on code that is actively running. As the name suggests, the first debuggers were not intended for reverse engineering. Rather, they were tools intended to help programmers find programming mistakes.

A debugger allows a programmer to observe all actions of a program while it is running. Most of the actions of programs running outside of a debugger will be unobservable to the human eye. Usually, only user interactions are actually visible. The task of a debugger is to make all internal states and all internal actions of a program observable. A debugger can be instructed to stop the execution of a program at a given code line. When the specified code line is reached, the content of specific memory locations can be probed. The internal state of a program can thus be revealed at any point of the execution. Another useful feature of a debugger is that it will allow stepwise execution of the machine code. It is therefore possible to follow the flow of a program at a speed compatible with the speed of the human brain.

Debuggers are indispensable tools in most reverse engineering processes. They allow the reverse engineer to understand the control flow of a program, as well as how complex data structures are actually built and used. A debugger can therefore help fill the semantic void left by disassemblers and decompilers.

6.6 Anti-reversing

All instances of reverse engineering for the purposes of analysing malware, understanding undocumented interfaces, and cracking DRM schemes have one important thing in common: to reveal something that the original programmer intended to keep secret. Therefore, the development of reverse engineering schemes has run in parallel with the development of schemes trying to prevent reverse engineering.

Such anti-reversing schemes generally come in two flavours, where one is known as code obfuscation. The purpose of obfuscation techniques is to change the code into a representation that is semantically equivalent but where the structure of the code and data of the program are difficult to reconstruct. This can be achieved through the rearrangement of instructions, the insertion of irrelevant code, or the encryption of parts of the code. The arms race between the reverse engineers and obfuscators of code follows the same lines as that between malware makers and malware detectors and is largely being fought by the same people. Rather than giving a separate account for the race here, we refer to Chap. 7 and particularly to Sect. 7.4.

The second type of anti-reversing scheme consists of those that intend to render the tools of the reverse engineer useless. For all three tools discussed above – disassemblers, decompilers, and debuggers – there exist ways to confuse them. Disassemblers can be confused into interpreting data as instructions and instructions as data. This is particularly useful for architectures with variable-length instructions. Decompilers can be confused by machine code that cannot be reconstructed in the high-level language. One very simple example of this is the insertion of arbitrary unconditional jump statements into Java bytecode. Since Java does not have a goto statement, arbitrary unconditional jump statements are hard to decompile [3]. A more challenging way to confuse a decompiler is to exploit the lack of division between instructions and data that exist on most hardware platforms. In machine code, sequences of binary values can be computed in an arbitrarily complex way and, after they have been computed, they can be used as instructions. A decompiler will not be able to handle such situations because of the strong division between code and data that is assumed in most high-level languages.

Debuggers can be beat by having the program code detect that it is being run in debug mode and simply terminate if it finds that it is. The debugger will thus not be able to observe the dynamic behaviour of the code. The key to this approach is that it is generally impossible for a debugger to completely hide its presence. It will most often be visible in the set of processes running on the machine. Furthermore, to stop the execution of a program at arbitrary points, the debugger will have to change the program by inserting an interrupt instruction. Such changes can be detected by the program itself by calculating checksums on portions of its code. The arms race between debuggers and anti-debuggers has a counterpart in the dynamic detection of malware. This topic is discussed in Chap. 8 and the ways in which malware can detect that it is being observed are discussed in Sect. 8.5.

A combination of anti-reversing techniques can make the reverse engineering of code arbitrarily complex, even for the simplest program functionality. On the positive side, these anti-reversing techniques all come at a cost: they either make the code less efficient, longer, or both. Unfortunately for us, short and efficient code may not be important criteria for a dishonest vendor of digital equipment. The cost of implementing anti-reversing techniques is therefore not likely to help us.

6.7 Hardware

Reverse engineering a computer chip is, in many ways, an art form. A plethora of techniques are available for removing thin layers of material and then identifying the structure of logic gates that constitute the chip. An overview of some of the techniques is given by Torrance and James [11]. Based on these techniques, it is often stated that any integrated circuit can be reverse engineered, given sufficient resources. The key to understanding this statement lies in quantifying what is meant by the term *sufficient resources*.

Nowadays a chip can consist of hundreds of millions of logic gates spread over a number of metal layers that runs in the two digits. Each logic gate performs an extremely simple operation; thus, the complex operation of a chip is a product of the interactions between these gates. There is currently no mature methodology that can produce high-level concepts from such a set of gate-level designs. Actually, finding a word-level structure from bit-level gates is still considered a difficult problem and, even when that problem is solved, we are very far from having understood a complete chip [7]. Fully reverse engineering a modern complex chip to the extent that all details of its operation are understood, down to the impact of every bit-level gate, is practically impossible, given the amount of effort it would require.

However, this may change in the future. As discussed above, strong tool sets are readily available for building high-level structures from low-level software and it is reasonable to assume that similar advances can be achieved for hardware. On the other hand, the development of such tools for hardware will clearly lead to the development of hardware obfuscation techniques as well. The possible future balance of power between hardware reverse engineers and hardware obfuscators is hard to predict. Still, a reasonable guess is that it will converge to a state similar to the balance of power found in the software domain. If this happens, hardware obfuscation techniques will reach a state where reverse engineering can be made arbitrarily complex but not theoretically impossible.

6.8 Discussion

Reverse engineering has played vital roles in most areas of engineering. It is used to understand the technical ideas behind competing products and the ease with which a product can be reverse engineered has been a driver behind such legal institutions as patents. In most fields, it is considered nearly impossible to include an idea in a product without revealing that idea to anyone who picks the product apart. Stating that ships, cars, buildings, and bridges have not been built according to specifications has also been the basis of legal claims.

In ICT, however, things are different. Implementing an idea into a product can often be done without disclosing the idea itself and, to support this, obfuscation techniques have been developed that make it even more difficult to extract engineering ideas from analysing a product. One consequence is that patents have played a lesser role in ICT equipment than could be expected from the technical complexity involved. Finding out whether a patented idea has been copied into a competing product is often a highly non-trivial task in itself.

The importance of reverse engineering in ICT is nevertheless evident. Major successes have been celebrated[1] by reverse engineering teams in all important appli-

[1]It should be noted that we use the words *success* and *celebrated* in a very subjective manner here and as seen from the perspective of the reverse engineer. In many cases, the task of a reverse engineer is to extract information that others try to hide. Depending on the situation, the reverse engineer may or may not have the right to claim the moral high ground.

cation areas. Programmer interfaces have been reversed to allow for the correct use and inclusion of components, the reverse engineering of malware has allowed us to better protect ourselves, the reverse engineering of DRM schemes has changed the course of entire industries, and the reverse engineering of cryptographic protocols has revealed weaknesses to be exploited or removed.

All of the successes do have one thing in common. They relate to relatively small pieces of code or the reverse engineering team was able to narrow the focus of the effort down to a sufficiently limited code area to make it tractable. For our case, this will not suffice. Depending on the intent of the dishonest vendor, the unwanted functionality can be placed anywhere in the product. Kill switches can be placed anywhere in the hundreds of millions of transistors on a given chip. They can be placed anywhere in firmware of a CPU so that it is rendered useless when a given combination of machine instructions are executed. As argued in Chap. 3, a kill switch can be placed anywhere in the operating system – in the device drivers, the hypervisors, the bytecode interpreters, the dynamic link libraries, or in an application itself – and, as explained in Chap. 4, can be introduced by any development tool used by the developers.

Morrison and colleagues and colleagues estimated that a full analysis of the Windows code base should take between 35 and 350 person–years, even if the source code is available to the reverse engineers [9]. Knowing the Windows operating system is only a small part of the total technology stack and that the expected lifetime of this code base is only a handful of years, it becomes evident that the state of the art in reverse engineering falls far short of being a satisfactory answer to the problem of untrusted vendors. It is, however, unlikely that reverse engineering will not play a central role in the future of this problem. Reverse engineering is and will remain the field that most directly addresses the core of our problem.

References

1. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Softw. Pract. Exp. **25**(7), 811–829 (1995)
2. Coats, W.S., Rafter, H.D.: The games people play: Sega v. Accolade and the right to reverse engineer software. Hastings Commun. Entertain. Law J. **15**, 557 (1992)
3. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Department of Computer Science, The University of Auckland, New Zealand, Technical report (1997)
4. Down, M., McDonald, J., Schuh, J.: The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Pearson Education, London (2006)
5. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, San Francisco (2011)
6. Eilam, E.: Reversing: Secrets of Reverse Engineering. Wiley, New York (2011)
7. Li, W., Gascon, A., Subramanyan, P., Tan, W.Y., Tiwari, A., Malik, S., Shankar, N., Seshia, S.A.: Wordrev: finding word-level structures in a sea of bit-level gates. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 67–74. IEEE (2013)
8. Liu, Q., Safavi-Naini, R., Sheppard, N.P.: Digital rights management for content distribution. In: Proceedings of the Australasian Information Security Workshop Conference on ACSW Frontiers 2003, vol 21, pp. 49–58. Australian Computer Society, Inc. (2003)

9. Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security. ACM–Association for Computing Machinery (2015)
10. Sikorski, M., Honig, A.: Practical Malware Analysis: The Hands-on Guide To Dissecting Malicious Software. No Starch Press, San Francisco (2012)
11. Torrance, R., James, D.: The state-of-the-art in IC reverse engineering. In: Cryptographic Hardware and Embedded Systems-CHES 2009, pp. 363–381. Springer (2009)
12. Zdziarski, J.: Hacking and Securing iOS Applications: Stealing Data, Hijacking Software, and How to Prevent It. O'Reilly Media, Inc., Sebastopol (2012)

Chapter 7
Static Detection of Malware

In the search for research fields that can shed light on our issue of checking a piece of equipment for unwanted functionality, static malware detection stands out as the most obvious candidate. Malware detection is as old as malware itself and its main goal is to discover if maliciously behaving code has been introduced into an otherwise clean system by a third party. In this chapter, we consider techniques that are static, in the sense that they are based on investigating the code rather than a running system. We will return to dynamic methods in a later chapter.

Our point of view is somewhat different from that of classic malware detection, in that we do not assume that we have a clean system to begin with. Our intention is to shed light on the implications of this difference to understand the extent to which the successes of static malware detection can transfer to our case.

7.1 Malware Classes

The most frequently cited definition of malware is *software that fulfils the deliberately harmful intent of an attacker*. This definition was presented by Moser et al. [11]. The term *malware* is usually understood to be an abbreviation of the two words *malicious software*. As we shall see in a later section, this definition is overly restrictive, since some types of malware threaten hardware development. A more general definition and one more relevant to our topic is, therefore, that malware is malicious code, regardless of whether this code defines hardware or if it is to be run as software. A reasonable definition of the term is therefore ‘code that deliberately fulfils the harmful intent of an attacker’.

An abundance of taxonomies of malware are to be found in the literature, but they generally agree on the most important terms. We do not intend to give a complete overview here; rather, we concentrate on the notions that are most relevant to our topic. A more complete set of definitions can be found in [17]. Here, we proceed with the three notions that define different ways that malware can spread and reside on a machine.

Virus	A computer virus shares the property of its biological counterpart that it cannot live on its own. Rather, it is a piece of code that inserts itself into an existing program and is executed whenever the host program is executed. Computer viruses spread by inserting themselves into other executables. The initial infection of the system can be accomplished through a program that only needs to be run once. The infecting program could, for example, reside on a memory stick.
Worm	A worm is a complete program in its own right and can execute independently of any other program. Its primary distinction from a virus is that it does not need a host program. This also means that its strategies for spreading will be different, since it does not need to alter existing executables to spread. It can spread through a network by exploiting vulnerabilities in operating systems.
Trojan	While viruses and worms spread in stealth mode, a Trojan horse is malware embedded into a seemingly innocent application that is explicitly and knowingly downloaded and run by the user. This application can be a screensaver, a small widget that displays the local weather, or a file received as a seemingly harmless attachment in e-mail. Infections embedded in malicious webpages are also categorised as Trojans.

Although the above categorization gives the impression that an attack falls into exactly one of these categories, this is not generally true. A sophisticated operation could take advantage of all three strategies above.

Orthogonal to the infection methods above is a set of notions related to what the malware is trying to achieve.

Spyware	The task of spyware is to collect sensitive information from the system it resides on and transfer this information to the attacker. The information can be gathered by logging keystrokes on a keyboard, analysing the contents of documents on the system, or analysing the system itself in preparation for future attacks.
Ransomware	As the name suggests, this is malware that puts the attacker in a position to require a ransom from the owner of the system. The most frequent way to do this is by rendering the system useless through encrypting vital information and requiring compensation for making it available again.
Bot	A bot is a piece of software that gives the attacker—or botmaster—the ability to remotely control a system. Usually a botmaster has infected a large number of systems and has a set of machines—a botnet—under his or her control. Botnets are typically used to perform attacks on other computers or to send out spam emails.
Rootkit	A rootkit is a set of techniques are used to mask the presence of malware on a computer, usually through privileged access—root or administrator access—to the system. Rootkits are not bad per se, but they are central parts of most sophisticated attacks. They are

also typically hard to detect and remove, since they can subvert any anti-malware program trying to detect it.

This list of actions that could be performed by malware covers the most frequent motivations for infecting a system. Still, we emphasize that the list is not exhaustive. Other motivations not only are conceivable but also have inspired some of the most spectacular digital attacks known to date. The most widely known of these is Stuxnet, whose prime motivation was to cause physical harm to centrifuges used in the enrichment of uranium in Iran [10]. Another example is Flame [3], which can misuse the microphone and camera of an infected device to record audio and video from the room where the infected system is physically located.

7.2 Signatures and Static Code Analysis

Checking for malicious intent in program code is usually done through *signatures*. In its simplest and earliest form, a signature is a sequence of assembly instructions that is known to perform a malicious act. Two decades of arms race between makers and detectors of malware have led to the development of malware that is hard to detect and advanced static signatures with complex structures. The utilization of such signatures is, in principle, quite straightforward: we need a repository of known sequences of instructions sampled from all known malware. Checking code against this repository, a malware detection system would be able to raise the alarm when a matching sequence is found.

There are basically three challenges to finding malware this way. First, the signature has to be generated and this is usually done manually [8]. Second, before the signature can be generated, the malware must have been analysed. This will not happen until its existence is known. There are examples of malware that were active for several years before they were found [19]. Finally, the repository of signatures is ever growing and new signatures have to be distributed continuously.

These challenges notwithstanding, the detection of malware through static signatures has historically been one of the most successful countermeasures against malware infections. The arms race between detectors and developers of malware is, however, still ongoing and, in the upcoming sections, we give an overview of how the race has played out.

7.3 Encrypted and Oligomorphic Malware

The response of malware developers to signatures was quite predictable. The developers needed to make malware that had the same functionality as malware for which a signature existed but where the signature itself would not produce a match. This was important for them for two reasons. First, in writing new malware, it is important that it is not caught by existing signatures. Second, as one's malware spreads and infects more and more machines, one would like it to automatically develop into different strands. This way, whenever new signatures can fight some instances of your malware, there are others that are immune.

An early attempt at making a virus develop into different versions as it spread involved encrypting the part of the code that performed the malicious actions. Using different encryption keys, the virus could morph into seemingly unrelated versions every other generation. For this to work, the virus had to consist of two parts, one part being a decryptor that decrypts the active parts of the malware and the other the malicious code itself. Although this made the static analysis of the actions of the malware somewhat harder, finding the malware by using signatures was not made any more difficult. The decryption loop itself could not be encrypted and it turned out that finding a signature that matched a known decryption loop was no more difficult than finding a signature for a non-evolving virus.

A second approach was to embed several versions of the decryption loop into the encrypted part of the malware. For each new generation of the virus, an arbitrary decryption loop is chosen so that one single signature will not be able to detect all generations of the malware. Viruses that use this concealment strategy are called oligomorphic [15] and they present a somewhat greater challenge for virus analysers, which will have to develop signatures for each version of the decryption loop. Still, for virus detection software, only the analysis time is increased. Oligomorphic viruses are therefore currently considered tractable.

7.4 Obfuscation Techniques

From the point of view of a malware developer, one would want to overcome oligomorphic viruses' weakness of using only a limited number of different decryption loops. The natural next step in the evolution of viruses was to find ways to make the code develop into an unlimited number of different versions.

In searching for ways to do this, malware developers had strong allies. Parts of the software industry had for some time already been developing ways to make code hard to reverse engineer, so that they could better protect their intellectual property. Rewriting code to have the same functionality but with a vastly different appearance was therefore researched in full openness. Some of the methods developed naturally found their way into malware development. Many techniques could be mentioned [20], but here we only consider the most common ones.

The most obvious thing to do when a signature contains a sequence of instructions to be performed one after the other is to insert extra insignificant code. This obfuscation method is called dead code insertion and consists of arbitrarily introducing instructions that do not alter the result of the program's execution. There are several ways of doing this. One can, for instance, insert instructions that do nothing at all—so-called nooperations—and these are present in the instruction sets of most processors. Another method is to insert two or more operations that cancel each other out. An example of the latter is two instructions that push and pop the same variable on a stack. Another obfuscation technique is to exchange the usage of variables or registers between instances of the same malware. The semantics of the malware would be the same, but a signature that detects one instance will not necessarily detect the other.

More advanced methods will make more profound changes to the code. A key observation is that, in many situations, multiple instructions will have the same effect. An example is when you want to initialize a register to zeros only: you could do so by explicitly assigning a value to it or by XOR-ing it with itself. In addition, one can also alter the malware by scattering code around and maintaining the control flow through jump instructions.

The most advanced obfuscations techniques are the so-called *virtualization obfuscators* [16]. Malware using this technique programs malicious actions in a randomly chosen programming language. The malware contains an interpreter for this language and thus performs the malicious acts through the interpreter.

In parallel with the development of obfuscation techniques, we have seen an abundance of suggestions for deobfuscators. These are tasked with transforming the obfuscated code into a representation that is recognizable to either humans or a malware detector equipped with a signature. For some of the obfuscation techniques above, deobfuscators are easy to create and efficient to use. The successes of these techniques unfortunately diminish when obfuscators replace instructions with semantically identical instructions where the semantic identity is dependent on the actual program state or when the control flow of the program is manipulated with conditional branches that are also dependent on the program state. This should, however, not come as a surprise. We learned in Chap. 5 that whether two programs are behaviourally identical is undecidable. Perfect deobfuscators are therefore impossible to design.

The hardest challenge in deobfuscation is to extract the meaning of code that has been through virtualization obfuscation. The first step in doing this would have to be to reverse engineer the virtual machine, to get hold of the programming language that was used in the writing of the malicious code. The complexity of this task becomes clear when we consider the following two facts. First, the virtual machine may itself have been obfuscated through any or all of the mechanisms mentioned above. Second, many different programming paradigms have strength of expression equal to that of a Turing machine. Logic programming, functional programming, and imperative programming are all considered in Sect. 9.3—but, in addition, we have algebraic programming [6] and Petri nets [13], to mention two of the more important. All of these paradigms can be implemented in a programming language in many different ways. Analysing the virtual machine itself is a task that can be made arbitrarily complex and the analysis must be completed before one can start analysing the operational part of the malware. This is a clear indication that we have a long way to go before the static analysis of programming code can help us against a malicious equipment vendor.

7.5 Polymorphic and Metamorphic Malware

Given the weakness of oligomorphic malware and the obfuscation techniques described above, the next step in the development of advanced viruses should be obvious. A polymorphic virus is an encrypted virus that uses obfuscation techniques to generate an unlimited number of versions of its decryption loop. A well-designed polymorphic virus can thus not be fought by finding signatures for the decryptor. These viruses are therefore fought through deep analysis of one version of the decryptor so that the decryption key can be extracted. Thereafter, the body of the virus is decrypted and matched with an ordinary signature. Although polymorphic viruses require a great deal of human effort in their analysis, their automatic detection need not be too computationally heavy once analysed.

Metamorphic viruses are the most challenging. They are not necessarily based on encryption and, instead, use obfuscation techniques throughout the entire body of the virus. This means that each new copy of the virus may have a different code sequence, structure, and length and may use a different part of the instruction set. Since obfuscation techniques have to be executed automatically from one generation of the virus to the next, a metamorphic virus must carry out the following sequence of operations to mutate successfully:

1. Identify its own location in storage media.
2. Disassemble itself to prepare for analysis of the code.
3. Analyse its own code, with little generic information passed along, since this information could be used in signature matching.
4. Use obfuscation techniques to transform its own code based on the analysis above.
5. Assemble the transformed code to create an executable for the new generation.

Efficient static methods for fighting metamorphic virus have yet to be developed [14]. The fact that no two versions of them need share any syntactic similarities makes the task hard and it is made even harder by the fact that some of the viruses morph into different versions every time they run, even on the same computer.

7.6 Heuristic Approaches

Looking for malicious code through signatures has the obvious drawback that, for a signature to exist, the malicious code has to be analysed in advance [2]. This also means that the malware has to be known in advance. In the problem we are studying, this is rarely the case. If the malware were already known, we would know it had been inserted; thus, we would already know that the vendor in question was not to be trusted. We need to search for unknown code with malicious functionality and we therefore need to approach malware detection differently. Heuristic malware detection tries to do so by identifying features of the code where one can expect there to be differences in the occurrence of that feature, depending on whether the code is malicious or benign. The code in question is analysed for the features in question and a classification algorithm is used to classify the code as either malicious or benign.

The first classes of features that were considered were N-grams [1]. An N-gram is a code sequence of length N, where N is a given number. Although an N-gram, at first glance, looks exactly like a very simple signature, there are crucial differences. First, N is often a very low number, so the N-gram is very short in comparison with a signature. Second, unlike for signatures, we are not interested in the mere question of whether there is a match or not; rather, we are interested in *how many* matches there are. Heuristic methods based on N-grams extract a profile of how a set of N-grams occurs in the code under investigation. This profile is classified as either benign or malicious by a classifier. The complexity of classifiers varies greatly, from simple counts of the occurrence of features to advanced machine learning techniques.

Other heuristic approaches use so-called opcodes instead of N-grams [4]. An opcode is the part of an assembly instruction that identifies the operation itself but without the part that identifies the data on which it operates. The techniques that can be used for classifiers are more or less the same as those used for N-grams.

A final class of features worth mentioning is that based on *control flow graphs* [5]. A control flow graph in its simplest form is a directed graph whose nodes represent the statements of the program and the edges the flow of program control. From this graph, several features can be extracted, such as nodes, edges, subgraphs, and simplified subgraphs with collapsed nodes.

Heuristic approaches have had some significant success. Still, static versions of these have one major limitation when applied to our problem: since we can assume that a dishonest equipment vendor is well aware of the state of the art in heuristic analysis, we can also assume that the vendor has made an effort to develop code that will be wrongly classified. Given the flexibility of the code obfuscation techniques described above, this is unfortunately not very difficult to do [12]. For this reason, present research and commercial anti-malware products favour dynamic heuristics [7]. We return to this topic in the next chapter.

7.7 Malicious Hardware

The intense study of malicious software that has taken place over several decades has been mirrored in the hardware domain only to a limited extent. For a long time, this situation was a reasonable reflection of the state of threats. The development and manufacture of hardware components were assumed to be completely controlled by one company and it was not suspected that any development team would deliberately insert unwanted functionality in the chips.

Both of these assumptions have now become irrelevant. Indeed, one topic of this book is exactly that of hardware vendors inserting unwanted functionality. Furthermore, the process of developing integrated circuits now involves many development teams from different companies. Putting together a reasonably advanced application-specific integrated circuit (ASIC) now largely consists of exactly that:

putting together blocks of logic from different suppliers. These blocks can be simple microprocessors, microcontrollers, digital signal processors, or network processors. Furthermore, as we saw in Chap. 4, Trojans can be inserted through the design tools and in the fabrication as well [18].

Static analysis of ASICs is conducted in industry for a variety of reasons. The state of the art in the field is discussed in Sect. 6.7 to the extent that it is relevant to our discussions. In addition to full static analysis of the chip, several approaches require the execution of hardware functionality. For these methods, we refer the reader to Sect. 8.7.

7.8 Specification-Based Techniques

The most intuitively appealing approach to detecting malware inserted by an equipment vendor is to start with a specification of what the system should do. Thereafter, one analyses whether the system does only this or if it does something else in addition. This approach is very close to what specification-based malware detection takes. Specification-based malware detection comprises a learning phase, where a set of rules defining valid behaviour is obtained. The code is then examined to assess if it does only what is specified.

The main limitation of specification-based techniques is that a complete and accurate specification of all valid behaviours of a system is extremely work intensive to develop, even for moderately complex systems [9]. The amount of results in this area is therefore limited.

7.9 Discussion

The static detection of malware has had many success stories. In particular, early virus detection software was based almost exclusively on static detection. As the arms race between malware writers and malware detectors has progressed, we have unfortunately reached a situation in which static detection is no longer effective on its own. Obfuscation techniques have significantly reduced the value of signatures and static heuristic approaches have not been able to close this gap.

The problem becomes even worse when we focus on dishonest equipment vendors rather than third-party attackers. All static methods require a baseline of non-infected systems for comparison. The whole idea behind signature-based malware detection is that it detects a previously known and analysed piece of malware and this malware is not present in non-infected systems. If you want to check whether a vendor inserted malware into a system before you buy it, the malware will not be known and analysed, and there will not be a non-infected system for comparison. This means that the analysis will have to encompass the entire system. We return to a discussion of the tractability of this task in Sect. 10.10. Heuristic methods will suffer from the same shortcoming: there is no malware-free baseline with which heuristic methods can train their classifier.

Even after having painted this bleak picture, there is still hope in the further development of static approaches. We have argued that full deobfuscation is very hard and often an impossible task. Still, it is possible to detect the existence of obfuscated code to some extent. One approach is therefore to agree with the vendor that the code used in your equipment will never be obfuscated. The problem with this is that obfuscation is used for many benign purposes as well. In particular, it is used for the protection of intellectual property. The balance between the benefits and drawbacks of obfuscation in the formation of trust between customers and vendors of ICT equipment needs further investigation before one can conclude whether banning obfuscation is a feasible way forward.

What appears to be the most promising way forward for static approaches is a combination of specification-based techniques and proof-carrying code, which we will elaborate upon further in Sect. 9.8. Specification-based techniques have not been subject to the same amount of attention as the other techniques. Still, for our problem, it has one big advantage over the other methods: it does not require the existence of a clean system and it does not require the malware to have been identified and analysed beforehand. Proof-carrying code has the drawback of being costly to produce. Still, efforts in this area so far have been to provide proof that the code is correct. Our purpose will be somewhat different, in that we want to make sure that the code does not contain unwanted security-related functionality. Although this is not likely to make all problems go away, the combination of controlling the use of obfuscation, applying specification-based techniques, and requiring proof-carrying code on critical components has the potential to reduce the degrees of freedom for a supposedly dishonest equipment vendor.

In recent years, malware detection has been based on a combination of static methods such as those discussed in this chapter and dynamic methods based on observing the actions of executing code. Such dynamic methods are discussed in the next chapter.

References

1. Abou-Assaleh, T., Cercone, N., Kešelj, V., Sweidan, R.: N-gram-based detection of new malicious code. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004, vol. 2, pp. 41–42. IEEE (2004)
2. Bazrafshan, Z., Hashemi, H., Fard, S.M.H., Hamzeh, A.: A survey on heuristic malware detection techniques. In: 2013 5th Conference on Information and Knowledge Technology (IKT), pp. 113–120. IEEE (2013)
3. Bencsáth, B., Pék, G., Buttyán, L., Felegyhazi, M.: The cousins of stuxnet: Duqu, flame, and gauss. Fut. Internet **4**(4), 971–1003 (2012)
4. Bilar, D.: Opcodes as predictor for malware. Int. J. Electron. Secur. Digit. Forensics **1**(2), 156–168 (2007)

5. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Detection of Intrusions and Malware & Vulnerability Assessment, pp. 129–143. Springer (2006)
6. Didrich, K., Fett, A., Gerke, C., Grieskamp, W., Pepper, P.: Opal: Design and implementation of an algebraic programming language. In: Programming Languages and System Architectures, pp. 228–244. Springer (1994)
7. Dube, T., Raines, R., Peterson, G., Bauer, K., Grimaila, M., Rogers, S.: Malware target recognition via static heuristics. Comput. Secur. **31**(1), 137–147 (2012)
8. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 1–42 (2012)
9. Idika, N., Mathur, A.P.: A survey of malware detection techniques, vol. 48. Purdue University (2007)
10. Langner, R.: Stuxnet: dissecting a cyberwarfare weapon. IEEE Secur. Privacy **9**(3), 49–51 (2011)
11. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: IEEE Symposium on Security and Privacy, 2007. SP'07, pp. 231–245. IEEE (2007)
12. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-Third Annual on Computer Security Applications Conference, 2007. ACSAC 2007, pp. 421–430. IEEE (2007)
13. Murata, T.: Petri nets: properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
14. O'Kane, P., Sezer, S., McLaughlin, K.: Obfuscation: the hidden malware. IEEE Secur. Privacy **9**(5), 41–47 (2011)
15. Rad, B.B., Masrom, M., Ibrahim, S.: Camouflage in malware: from encryption to metamorphism. Int. J. Comput. Sci. Netw. Secur. **12**(8), 74–83 (2012)
16. Rolles, R.: Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies.(WOOT) (2009)
17. Szor, P.: The Art of Computer Virus Research and Defense. Pearson Education (2005)
18. Tehranipoor, M., Koushanfar, F.: A Survey of Hardware Trojan Taxonomy and Detection (2010)
19. Virvilis, N., Gritzalis, D., Apostolopoulos, T.: Trusted computing vs. advanced persistent threats: Can a defender win this game? In: 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC) Ubiquitous Intelligence and Computing, pp. 396–403. IEEE (2013)
20. You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300. IEEE (2010)

Chapter 8
Dynamic Detection Methods

The static detection of malware has celebrated successes over the years, but obfuscation techniques have deprived static methods of many of their advantages. The Achilles heel of obfuscated code is that, however difficult to read and understand, it has to display its actions when executed. Dynamic methods for malware detection exploit this fact. They execute the code and study its behaviour.

In this chapter, we give an overview of dynamic malware detection. We discuss the different methods used and their strengths and weaknesses compared to static analysis. Again, the goal of the chapter is to shed light on the extent to which dynamic malware detection techniques can help us in detecting the actions of a dishonest equipment provider.

8.1 Dynamic Properties

When we want to observe the behaviour of a system, we first need to decide what aspects of the execution we are going to monitor. Egele et al. [6] provide a good overview of the different alternatives, the most important of which we will review here.

Tracing the function calls of a system under inspection is a natural approach to dynamic malware detection [9]. In programming, functions usually represent higher-level abstractions related to the task that the code should perform. They are also the building blocks of the application programming interface (API) and constitute the system calls that an operating system provides to the programmer to make use of system resources. The semantics of the functions and the sequence in which they are called are therefore very suitable for analysis of what a program actually does. The interception of function calls is carried out by inserting *hook functions* into the code under inspection or by exploiting the functionality of a debugger. These hook functions are invoked whenever a function subject to analysis is called.

In addition to obtaining the actual sequence of function calls carried out by a program, the dynamic collection of function calls also reveals the parameters used in each call. This information is inherently difficult to obtain through static code analysis, since the value of each variable passed on as a parameter could be the result of arbitrarily complex computation.

Another useful feature that can be extracted through dynamic analysis is how the system processes data. In its simplest form, this consists of recording how the contents of given memory locations influence what is later stored in other memory locations. As an example, we could assume that one memory location is initially tainted. Whenever an assignment statement is executed, the taint progresses to the memory location that is assigned a new value, if and only if the new value is computed based on the content of the already tainted memory location. Using this information, we would, for example, be able to detect cases in which tainted information is leaked over the network.

The lowest level of properties that can be captured by dynamic analysis is the sequence of machine instructions that is performed by the system. For any given program, such a trace will easily turn out to be immense and therefore extremely costly to analyse in full. Still, such traces will contain information that may not be represented in the high level of abstractions of API-, function-, and system calls.

8.2 Unrestricted Execution

Capturing the dynamic properties of suspicious code by running it in an unrestricted environment has some clear advantages. First, the test environment is easy to set up and, second, it is hard for the malware to detect that it is being observed and thus choose to suppress its malicious actions. API calls and system calls can be collected by inserting hooks into the called functions. The hooks are inserted in the same manner as in rootkits [10].

This approach has two significant weaknesses. First, since the code under inspection runs directly on hardware, it is complicated to collect instruction traces and internal function calls. Second—and this is the most serious drawback—the malicious actions of the malware will be executed in an unrestricted way. This means that the harm intended by the malware will actually be committed. The second drawback can, to some extent, be mitigated by recording the actions of the code under inspection and taking snapshots of the system state. This would allow the tester to restore the system to a non-damaged state. Unfortunately, it is not always possible to roll back the effects of a system. Information leaked out of a system cannot be called back and attacks on other systems can cause harm that cannot easily be undone in the test environment alone.

8.3 Emulator-Based Analysis

Running the code without allowing it to cause harm requires the use of a controlled execution environment. An emulator—often referred to as a *sandbox* in the context of malware analysis—is a software system that replicates the actions of hardware. Using an emulator for a given hardware architecture, one can load an operating system on top of it. In this operating system, the code under inspection can be started and the consequences of its actions confined to the sandbox.

Emulator-based analysis has an additional advantage. Since the hardware is emulated by software, it is easy to insert hooks to extract features such as instruction traces, which are usually only observable at the hardware level. On the other hand, such analysis introduces a semantic gap in comparison with unrestricted execution with API hooks [6]. A hardware emulator cannot immediately identify API or system calls, since these will appear to the emulator as a sequence of instructions that together implement the call in question. These sequences will need to be identified and recognized before the higher-level semantic information of API calls can be obtained.

The most important limitation of sandboxes is, however, their speed of operation. Hardware emulated in software inevitably represents a slowdown in itself. More significantly, however, every resource used for analysis represents the removal of resources for running the code under inspection. The detailed insight provided by the sandbox comes at a high cost of overhead. This approach is therefore usually applied only to small suspicious portions of the code.

8.4 Virtual Machines

In an emulator, all the actions of the hardware are simulated by means of software. Therefore, the emulator itself can, in principle, run on hardware that is different from the hardware it emulates. This level of indirection contributes to the significant slowdown induced by emulation-based analysis in sandboxes.

Virtual machines are similar to emulators, in that they encapsulate execution in a software environment to strongly isolate the execution under inspection. They are, however, different, in that they run on hardware that is similar to the hardware they pretend to be. This means that non-privileged instruction sequences in the code under inspection can be run directly on hardware, a feature that could speed up the execution considerably.

The strength of virtual machines in relation to emulators is that they run faster. On the other hand, it is hard to generate instruction traces on non-privileged sequences of instructions that run directly on hardware. The semantic gap between the sequences of instructions and API and system calls is similar between the two approaches.

8.5 Evasion Techniques

A malware programmer's first duty is to prevent the malware from being detected. When facing dynamic detection methods, malware should therefore be expected to use the same strategy as a human wrongdoer, namely, to behave well as long as you are being observed. Code whose intention is to hide its malicious capability will therefore try to detect whether it is under observation and will only carry out its malicious actions when it has reason to believe that it can do so without being detected [13].

Several different ways of detecting observation have been found in malware. We have seen samples that will not exhibit their malicious behaviour when run in debug mode. Furthermore, we have seen malware that behaves well whenever it detects specific patterns of users or simultaneously running processes. Furthermore, there is an arms race between the makers of emulators and virtual machines and malware developers when it comes to detection and evasion techniques. One particularly interesting aspect of this race is the suggestion that having your system appear as if it were observing the executing code reduces the likelihood of the malware in your system revealing its malicious effects [4]. This will not prevent one from becoming infected but it will, in some cases, make one immune to the consequences.

The bad news is that it is very hard to completely hide emulators or virtualization techniques from the code that is running on top of them. This is particularly the case if the code under inspection must have access to the Internet and can use this to query a remote time source [8]. The good news regarding evasion techniques is that the emulators and virtual machines that some malware are trying so hard to detect are rapidly becoming common platforms upon which most software, including operating systems, are running. The virtualization of resources is nowadays ubiquitous in computing and, thus, malware that exhibits its malicious behaviour only when run on non-virtualized systems is losing its strength. In our case, however, the problem is of a different nature. We are not only concerned with a piece of code. A typical mode of interest to us would be determining if an entire system consisting of hardware, virtualization support from a virtual machine monitor (VMM), operating systems, and application code does not have malicious intent coded into it. It is not important that the malware hide its behaviour from the VMM if it is the VMM itself we do not trust.

8.6 Analysis

In the sections above, we discussed the problem of collecting information from the execution of some code under inspection. Analysis of this information to conclude whether malicious behaviour is observed is, however, a challenge of its own.

The analysis of execution traces is associated with many of the same problems and solutions that we discussed for heuristic static analysis in Sect. 7.6. This means that many of the methods we described in the previous chapter also apply to dynamic analysis. N-Grams have been used to analyse dynamic instruction traces, as well as static instructions found in the code [2]. Although there are differences between instruction traces captured dynamically from those that are captured statically, the methods for their analysis have close similarities.

Other features, such as the sequences of system and API calls, are more easily captured through dynamic than static detection. Still, the problem of making sense of such sequences bears similarities with the problem of making sense of the control flow information that is used in static analysis (see Sect. 7.6). Therefore, the problem is approached in much the same way and with techniques from machine learning and artificial intelligence [7, 11].

Our question is whether the dynamic analysis of software for malware identification can help us identify malicious intent in integrated products offered by a vendor. The weakness of static analysis in our case is that the same malicious behaviour can manifest itself in countless different ways in static code. During execution, however, similar malicious actions will express themselves similarly. We have therefore seen dynamic methods for detecting malware gain momentum, and few modern products for malware detection still based on only static detection [5]. There are still two weaknesses that static and dynamic analysis based on machine learning and artificial intelligence have in common: First, they must have access to a learning set that is free of malicious actions. In our case, such sets are hard to define accurately. Second, when the methods for detecting malicious actions become publicly available, they are generally easy to evade.

8.7 Hardware

The literature on malicious software has focused almost exclusively on scenarios where the attacker is a third party. The software user and manufacturer are implicitly assumed to be in collaboration in deterring unknown wrongdoers from inserting malicious code into a piece of equipment. The situation is different for hardware. A wrongdoer is not expected to be able to change the hardware in a finished product. Therefore, hardware security has consisted of techniques to ensure that the design does not inadvertently build security holes into a system.

The thought that a wrongdoer could insert malicious logic gate structures into an integrated circuit was first taken seriously when integrated circuit design and manufacturing practices started increasingly relying on intellectual property cores supplied by third-party vendors. In addition to this development, modern integrated circuits are currently manufactured in third-party facilities and are designed using third-party software for design automation [3]. Distrust in third parties in the hardware domain therefore points the finger at the production phase rather than the post-deployment phase.

Discussions on hardware Trojans have taken particular foothold in the military sector. In September 2007, Israeli jets bombed what was suspected to be a nuclear installation in Syria. During the attack, the Syrian radar system apparently failed to warn the Syrian army of the incoming attack. Intense speculation and an alleged leak from a US defence contractor point to a European chipmaker that built a kill switch into its chips. The speculation is therefore that the radars were remotely disabled just before the strike [1].

There are two classes of dynamic approaches for finding such Trojans in integrated circuits. One is the activation of the Trojan by executing test vectors and comparing the responses with expected responses. If the Trojan is designed to reveal itself only in exceptionally rare cases, such as after a specifically defined sequence of a thousand instructions, such an approach is unfortunately practically impossible, due to the combinatoric explosion in the number of possible stimuli [16].

The other method is the analysis of side-channels through measuring irregularities in power consumption, electromagnetic emission, and timing analysis. This is doable when there are chips without Trojans with which the measurements can be compared [14]. When these Trojans are deliberately introduced by the manufacturer, there may not be any Trojan-free chips for comparison. In our case, side-channel analysis can therefore be easily duped by the perpetrator.

With hardware, unlike software, a great deal of effort goes into handling the problem of untrusted producers of code that goes into the product. It is therefore disconcerting that wrongdoers still seem to operate without substantial risk of being caught. There is yet no 'silver bullet' available that can be applied to detect all classes of hardware Trojans with high confidence [15]. This observation should, however, not lead to criticism of the researchers in the area; rather, it should make us realize the complexity of the problem at hand. Continued research is the only real hope we have to be able to address it.

8.8 Discussion

After obfuscation techniques have rendered many previously successful static approaches useless, the industry has turned, to an increasing extent, towards dynamic techniques for detecting malware. The reasoning behind this development is sound. Whereas it is possible to camouflage malicious behaviour in the code, it is far more difficult to hide malicious behaviour in the actions that are performed.

Dynamic detection has two significant weaknesses. One is that, in any run through the code, only one execution path will be followed. Consequently, a dishonest vendor can make sure that the malicious actions of the code are not executed in the time available to the analyser. One way to do this is to start the malicious behaviour only after the system has run for a given amount of time. As discussed above, the observation of code behaviour will, in most cases, mean that the code will run quite slowly and, as a result, there is a high probability that the limit of a given delay will never be exceeded. Still, the safest way for a malicious equipment producer to steer the execution path away from the malicious code is to make it dependent on a predefined external stimulus. Having this external stimulus encoded in only 512 bits would yield 13.4×10^{153} combinations. For comparison, the universe has existed for approximately 4×10^{17} seconds. If the strongest computer on Earth had started computing at the beginning of the universe, it would still not have made any visible progress on the problem of testing these combinations [1]. The other weakness of dynamic detection is that one has to choose between execution in a real environment, where the malicious actions will actually take place, and execution in an environment where the system can detect that it is being observed.

Modern malware detection products combine static and dynamic detection in an attempt to cancel out the respective weaknesses of the two approaches [7]. Unfortunately, this is not likely to deter a dishonest equipment vendor. In software, the combination of obfuscating the malicious code in the product and letting it hide its functionality until a complex external trigger occurs suffices to reduce the chance of detection below any reasonable threshold. Symbolic execution has made some inroads in that area [17], but the state of the art is still far from changing the game. In hardware, the same trigger approach can be used to avoid dynamic detection and malicious actions can be hidden in a few hundred gates within a chip containing several billions of them [12], so that static analysis would be prohibitively hard, even if destructive analysis of the chip resulted in a perfect map of the chip's logic. In addition, a vendor that produces both the hardware and the software can make the malicious acts be a consequence of their interaction. The malicious acts would thus invisible in either the hardware or software when studied separately.

When we expect a malicious equipment vendor to be able to put malicious functionality into any part of a technology stack, it is hard to extract traces of activities that we can be reasonably sure have not been tampered with. Still, insight from the field of dynamic analysis is of interest to us. In particular, there is reason to hope that malicious functionality can be detected through observation of external communication channels. Unfortunately, with kill switches and, to some extent, the leakage of information, the damage will already have been done when the activity is detected.

References

1. Adee, S.: The hunt for the kill switch. IEEE Spectrum **45**(5), 34–39 (2008)
2. Anderson, B., Storlie, C., Lane, T.: Improving malware classification: bridging the static/dynamic gap. In: Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence, pp. 3–14. ACM (2012)
3. Bhunia, S., Hsiao, M.S., Banga, M., Narasimhan, S.: Hardware trojan attacks: threat analysis and countermeasures. Proc. IEEE **102**(8), 1229–1247 (2014)
4. Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), pp. 177–186. IEEE (2008)
5. Dube, T., Raines, R., Peterson, G., Bauer, K., Grimaila, M., Rogers, S.: Malware target recognition via static heuristics. Comput. Secur. **31**(1), 137–147 (2012)
6. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 1–42 (2012)

7. Gandotra, E., Bansal, D., Sofat, S.: Malware analysis and classification: a survey. J. Inf. Secur. (2014)
8. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: Vmm detection myths and realities. In: HotOS (2007)
9. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. J. Comput. Secur. **6**(3), 151–180 (1998)
10. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows kernel. Addison-Wesley Professional (2006)
11. Jacob, G., Debar, H., Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. J. Comput. Virol. **4**(3), 251–266 (2008)
12. King, S.T., Tucek, J., Cozzie, A., Grier, C., Jiang, W., Zhou, Y.: Designing and implementing malicious hardware
13. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting environment-sensitive malware. In: Recent Advances in Intrusion Detection, pp. 338–357. Springer (2011)
14. Rad, R., Plusquellic, J., Tehranipoor, M.: A sensitivity analysis of power signal methods for detecting hardware trojans under real process and environmental conditions. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **18**(12), 1735–1744 (2010)
15. Tehranipoor, M., Wang, C.: Introduction to Hardware Security and Trust. Springer Science & Business Media (2011)
16. Wang, X., Tehranipoor, M., Plusquellic, J.: Detecting malicious inclusions in secure hardware: challenges and solutions. In: IEEE International Workshop on Hardware-Oriented Security and Trust, 2008. HOST 2008, pp. 15–19. IEEE (2008)
17. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 732–744. ACM (2015)

Chapter 9
Formal Methods

Mathematical reasoning is the foundation of most engineering disciplines. It would be unthinkable to construct a ship, bridge, or building without first making a mathematical model of the design and calculating that the design satisfies relevant requirements. Such models are used in the exploration of the design space, in quality assurance processes during construction, and in certification processes.

Computers and systems' counterpart to traditional engineering mathematics is called *formal methods*. The goal of this discipline is to capture the essence of computer code in formal mathematical theory and then draw conclusions about the code through mathematical analysis. In this chapter, we give an overview of the field, with the aim of understanding how formal methods can help us defend against untrusted equipment vendors.

9.1 Overview

The ultimate goal of a formal method is to prove the properties of the behaviour of a piece of executable code or electronic circuit. In our case, we are interested in proving that the piece of equipment at hand does what it is supposed to do and nothing else. Five basic notions are important to grasp on how such a proof is actually possible:

Specification If we want to prove a property of some code, we first have to state what this property is. Such statements are written in a formal language that is often an enriched version of some mathematical theory.

Implementation This is the object whose properties are to be proved. It can consist of code in some programming language or a specification of an entire integrated circuit or parts of one.

Semantic translation Given the implementation and specification of the wanted properties, one must synthesize the proof obligations, that is, determine which mathematical proofs need to be performed before we can conclude that the implementation fulfils the specification.

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_9

Model In some cases, the detailed complexity of a complete implementation is such that a full formal proof is practically impossible. In that case, one can make a simplified model of what an implementation of the specification could be, prove that the model fulfils the specification, and finally generate the implementation automatically from the model. The idea is still to prove that the implementation fulfils the specification but this is now done by a detour through a model. In this case, it is the specification and the model that are passed on to the semantic translation.

Logic propositions The result of semantic translation is a set of propositions expressed in some mathematical logic. If all of the propositions are true in the given logic, we can conclude that the implementation fulfils the specification.

Theorem proving This is the process of proving the logic propositions above.

Figure 9.1 illustrates the structure of the full formal proof process of a piece of code. It is important to note that this figure intends to capture ways of finding a complete and watertight proof of correctness. Formal methods have been developed to help minimize the number of unintended programming faults. If that were our goal, there would be many variations of Fig. 9.1 that would be more enlightening. We are, however, not interested in improving the quality of programming practice. Rather, we are looking for ways to ensure that no malicious code has deliberately been inserted by the software makers. For that purpose, our description is appropriate.

In a perfect world, all the phases of a full formal proof would be performed automatically. If that were doable, we would only need to provide the specification and the correctness of the implementation could be checked by a machine. Unfortunately, as we saw in Chap. 5, there are very tight limits to what can be decided automatically. This means that all complete sets of formal methods that are able to prove interesting properties of a piece of code will require the heavy use of human expertise, in either semantic translation, theorem proofs, or both.

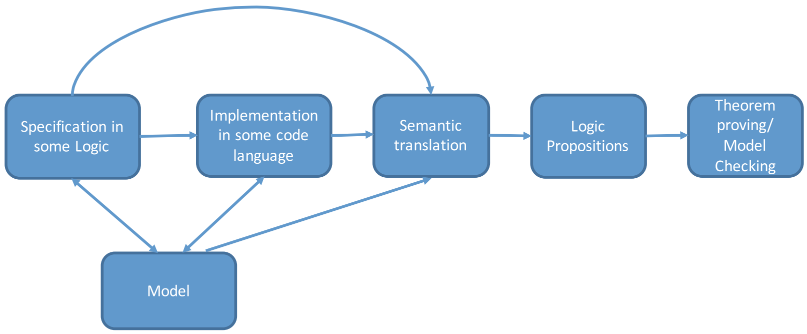
Specification in some Logic
Implementation in some code language
Semantic translation
Logic Propositions
Theorem proving/ Model Checking
Model

Fig. 9.1 These are the elements that constitute proof of a property using formal methods

9.2 Specification

An early attempt to build a logic for reasoning on the behaviour of a program was made via the notion of *assertions*. An assertion is a statement that is to be evaluated as **true** at a given place in a program [14]. Assume, for example, that we have the statement

```
x := y+1
```

Then, clearly, after the statement's execution, the assertion that `x>y` would be true.

This argument can be extended to the notions of *preconditions* and *postconditions*, where a precondition is an assertion that is supposed to be true before the start of a program and a postcondition is supposed to be true at the end of a program's execution. Preconditions and postconditions can be used to specify a program's behaviour. For instance, if we want to specify that a program is to sort an array `A` of `n` distinct integers in increasing order, we would require a postcondition that stated that, for all integers `i` between `1` and `n-1`, `A(i-1) < A(i)` should be true.[1] Furthermore, the postcondition would state that all integers present in `A` before the execution should also be present after the execution. The precondition should simply state that no integers are represented in `A` more than once.

Pre- and postconditions are important because they can be used to specify what a program should do without specifying how it should be done. Note that all of the different sorting algorithms one has ever learned would be valid according to the pre- and postconditions above, even though they differ vastly in their approach.

Pre- and postconditions were involved in an early approach to specify the outcome of a program's execution. It had several shortcomings, though, in that several important aspects of programming could not be easily captured. Some, such as input/output and user interaction, can be overcome by introducing sequences of inputs and outputs, respectively, as different variables into the machine. Others, such as the fact that, for instance, an operating system is assumed to run indefinitely and is therefore not supposed to have a final state, are more of a stretch. Several important notions emerged from these observations. One is the notion of an *invariant*, which is an assertion that is supposed to be true at a point of the code that can be reached infinitely many times. Another is the development of the *temporal logic of programs* [23], which allows one to express properties about time, such as *when A happens, then eventually B will happen*. Many other notions could be mentioned, but these two stand out as the most important and they form the basis of most of the others.

Significant amounts of training are required to master the rigour and precise formal semantics of provable specifications. This has been a barrier for the uptake of formal methods. Because of this, the specification of software is generally done in languages that are not based in firm formal semantics. An example of such a language is the *Unified Modeling Language* (UML), which is widely used for the modelling and

[1]We follow the informatics convention of starting the enumeration at zero. An array of `n` integers is thus enumerated from `0` to `n-1`.

specification of systems [25]. Bridging the gap between formal rigour and intuitively appealing specification methods has therefore been one of the key challenges to making fully formal specification more widespread [11].

9.3 Programming Languages

Historically, programming developed from bit-level programming to assembly programming, to structured programming with loops and procedure calls, and then to object-oriented programming. All of these programming paradigms are said to be *imperative*, in that they require the programmer to specify sequences of actions that change the state of memory. Imperative programming languages, including C, C++, and Java, form the basis of the overwhelming majority of software development today. They have therefore also been the subject of a great deal of research trying to capture programs' properties through formal methods.

There are, however, many different paradigms that have the strength of expression needed to form the basis for software development. In the formal methods community, some are of particular interest, since their mode of operation closely parallels existing mathematical theories. Two such programming paradigms that have gained industrial importance are *functional programming* and *logic programming*. Functional programming languages use the recursive definition of mathematical functions as their main building block. Execution of a program corresponds to evaluation of the mathematical function. In its pure form, a functional program has no explicit internal state; thus, there is no concept of state change. Most functional languages in industrial use can, however, be deemed to have a hybrid nature, where some notion of program state is supported. Examples of functional languages that have industrial use are Lisp [26], Scheme [27], and Erlang [2].

Logic programming languages are based on formal logic and a programs in these languages can therefore be viewed as logical clauses. Execution of a program consists of finding solutions to a logical clause, in the sense of finding instantiations of the variables for which the clause is evaluated as being true. The industrial impact of logic programming languages has been relatively limited, but Prolog [8] and Datalog [5] are worth mentioning.

The definition of integrated circuits was for a very long time quite distinct from that of programming. These days, the distinctions have diminished, in that the definition of circuits is mostly carried out in high-level languages such as those described above. Synthesis tools transform these high-level descriptions into descriptions in terms of logic gates (see Sect. 4.2).

9.4 Hybrid Programming and Specification Languages

In Sect. 9.2, we concentrated on the pure specification of programs, in the sense that we only considered methods that did not say anything about how the execution should be carried out. Then, in Sect. 9.3, we turned our attention to programming languages that specify the execution in itself. There is, however, a class of languages that are somewhere in the middle, in that they are mostly used for specification purposes but where the specifications are, to some extent, executable. For efficiency reasons, these languages are most often used for modelling only. Programs or specifications in these languages are translated, manually or automatically, into other languages before they are executed.

The earliest example of this involves state machines [12]. A state machine consists of a (usually finite) set of states that a system can be in, together with rules for what events make the system move from one state to the other. Recall that, in Chap. 5, we learned that a Turing machine is believed to be able to simulate any computing device. When we study a Turing machine, we see that it can easily be represented by a set of states and some rules for transitioning between the states. Therefore, most reasonable definitions of what a state machine is will be Turing equivalent. Even though this means that a state machine can be seen as a complete programming language, state machines are rarely used to model a computer system in full detail. A computer with, for example, 1 GB of memory will have $2^{10^{12}}$ potential states, which is far beyond what can be used in a formal method proof chain. The states are therefore usually abstracted and aggregated into logical *extended* states.

One particular string of hybrid languages came out of the observation that the distinction between specification and programming caused problems when the systems became parallel and distributed. Whereas a sequential program will execute the same way for the same input every time, a parallel program can behave differently for the same input, depending to how race conditions play out. This issue made it clear that, when proving the properties of a parallel program, it would be highly beneficial if the specification itself contained notions of the different parallel processes in the program, as well as how they were supposed to interact. This led to the development of *process algebras*, which, in the context of specification, have the commonalities that they describe the processes of a parallel program and that these processes communicate through explicit message passing rather than through the manipulation of shared variables. The first two developments that fall into the category of process algebras are communicating sequential processes (CSP) [15] and the calculus of communicating systems (CCS) [17]. These appeared more or less at the same time and were developed in parallel for a long time. Later, several variants of process algebras were introduced. The most acknowledged of these is the π-calculus [18] that still forms the basis for an active area of research.

A third strand worth mentioning consists of languages that specify programs as algebraic transformations. The elegance of this approach is that it unifies a mathematical theory of algebraic equivalence with the operational aspect provided by algebraic simplifications [13]. This strand of research had its early days in the late 1970 s and is still vibrant today, through the executable specification language Maude [7].

9.5 Semantic Translation

When we have a program and a specification of what the program should do, the task of semantic translation is to extract from the two a set of *proof obligations*. These proof obligations are a set of mathematical expressions that, if proven to be true, imply that the program behaves according to the specification. Our entry point into this issue is *Hoare logic* [14], which is an axiomatic basis for reasoning about imperative programs.

The core of Hoare logic is the Hoare triple, denoted $\{P\}S\{Q\}$. In this triple, S is a program and P and Q are logic expressions on the state of the program. Intuitively, the triple means that if P is true before the execution of S, then Q will be true after the execution. An example of a valid triple is

$$\{x + 1 < y\}\ x := x + 1;\ \ \{x < y\}$$

Hoare logic devises deduction rules for reasoning on such triplets over constructs that occur in imperative languages, such as assignments, if statements, loops, and statement composition. Using these deduction rules, it is possible to establish a triple $\{P\}S'\{Q\}$ for programs of any complexity. Finally, instantiation of the consequence rule below relates this to the specified precondition and postcondition (see Sect. 9.2):

$$\frac{Precondition \Rightarrow P \quad \{P\}S\{Q\} \quad Q \Rightarrow Postcondition}{\{Precondition\}S\{Postcondition\}}$$

This rule states that, if a program is specified by a precondition and a postcondition and Hoare logic is used to prove the triple $\{P\}S\{Q\}$, then the resulting mathematical proof obligations are that the precondition implies that P holds and that Q implies that the postcondition holds.

Unfortunately, proving a Hoare triple by the deduction rules given by Hoare was soon shown to be highly nontrivial. Later developments therefore attempted to reformulate the deduction rules to form the basis of algorithmic support for building this part of the proof. This approach led to the notion of *weakest precondition*, introduced by Dijkstra [9], which reformulated the ideas of Hoare logic into a system where the formation of a weakest possible precondition from a given postcondition could partly be determined automatically. Later, the work of Hoare and Dijkstra was extended to parallel programs [16] and object-oriented programs, to mention just two [22].

As stated in the previous section, process algebra was borne out of the observation that the distinction between specification and implementation seen in Hoare logic creates difficulties when it comes to parallel and distributed systems. Since a specification in process algebra is partially executable in its own right, the proof obligation for these formalisms is to establish equivalence between the execution of the specification and its implementation. Such equivalence is defined as capturing a situation where the two simulate each other. *Bisimulation* is thus a relation between two executables where, intuitively, an external observer is not able to identify which one is executing. Execution time is usually not deemed to be observable in these definitions. The reason is that improved execution time is the reason for replacing the specification with the implementation in the first place.

9.6 Logics

Several preexisting mathematical theories have been used as the fixed frame for proving the properties of programs. Initial approaches were based on some predicate logic – usually first-order logic – enriched with algebraic laws for integers or rational numbers. Others were developed specifically for the purpose. Two examples are temporal logic [23], which was specifically developed to capture the notion of time, which is important in infinite processes and real-time systems, and logics to handle the finite resources available in a physical computer [24].

9.7 Theorem Proving and Model Checking

After the relation between the specification and the program has been transformed into propositions in formal logic, these propositions need to be proven. The implication is that most processes for proving theorems are interactive. A great deal of research has gone into the development of interactive theorem provers and a few of the most important ones are the Prototype Verification System (PVS) [20], Isabelle [21], and Coq [3]. Going into the details of their different properties is outside the scope of this book, so we refer to Boldo et al. [4] for a good overview.

Ideally, one would hope that these theorem provers could work automatically but, unfortunately, all the relevant problems are NP-hard. Furthermore, a majority of them are super-exponential or even undecidable. This should come as no surprise, because we saw in Chap. 5 that whether a piece of code performs malicious actions is generally undecidable. This undecidability would have to show up in the case of formal proofs as well, either in semantic translation or in theorem proving.

The undecidability problem did limit the applicability of formal methods to prove the correctness of more than very small systems for a long time. The full complexity of the proof obligation for large systems simply became intractable. *Model checking* then came to the rescue and showed its value through the 1990s. The idea in model checking is to abstract a simplified model of the state space of the system and then check the entire state space or parts of it for execution traces that would be a counterexample to the specification.

Model checking has proven itself to be very useful for both finding bugs and proving the properties of protocols [6]. Still, for the problem we study in this book, model checking comes at a high price, because building a model of a system creates a level of indirection in the proof systems. Even if we can prove a model's properties and can generate the code directly from it, the generation of code is in itself a possible point of insertion of unwanted behaviour. This fact is discussed in detail in Chap. 4.

9.8 Proof-Carrying Code

We have seen that it is very hard to find a full formal proof for a finished piece of code. Semantic translation of the specification and the code is highly nontrivial and proving the logic propositions that come out of this translation is an NP-complete problem at best and most often undecidable. One of the main obstacles is that it is generally very hard to analyse code that one has not participated in writing. We return to this topic in Sect. 10.10.

Providing proof of correctness while one develops the code should, however, be easier. Clearly, when one programs, one has a mental understanding of why the code will work. Constructing proof of correctness while one programs should therefore amount to formalizing an understanding one already has. Furthermore, while finding a proof for a piece of code is very hard, checking the correctness of an existing proof is trivially simple. The idea behind proof-carrying code [19] is to exploit these two facts. The burden of providing proof is moved to the producer of the equipment and the buyer only has to check that the proofs hold water. Proof-carrying code was first described in application to software but the idea was later transferred to hardware [10]. The concept should therefore apply to the entire technology stack.

Although the idea of proof-carrying code holds great promise, it has not seen much practical use. The reasons for this are that, even though it is easier for a programmer to provide proof while programming, proof-carrying code still requires a great deal of effort and expertise. The general sentiment is that the extra cost that proof-carrying code would incur in a development project will seldom be covered by increased sales. Still, this remains one of the promising paths to follow when looking for solutions to the problem of trust in untrusted vendors.

9.9 Conclusion

Much effort has been expended for many decades in finding the foundations of programming and developing formal methods for reasoning about programs and integrated circuits. This effort has had a profound impact on the development of programming languages and language constructs. Furthermore, it has contributed notions such as *invariant* and *precondition* to the everyday language of most programmers. Finally, formal methods has been successfully applied to finding mistakes in specifications and rooting out bugs in computer code.

Unfortunately, hopes that formal methods will one day provide the full mathematical proof of the total correctness of an entire system, as described in Chap. 3, have dwindled. The scalability of the approaches has in no way kept up with the rapid increase in size and complexity of the systems that one needs to consider. There are some noteworthy successes of full formal proofs of software, but they are confined to select application areas and most often to critical subparts of the system itself. For example, in a survey on the engineering of security into distributed systems, Uzunov and his co-authors [28] stated that 'the application of formal methods is almost invariably localised to critical aspects of a system: no single formal technique is capable of formally verifying a complete complex system'. The picture is somewhat different for integrated circuits, where formal methods in many development projects have become a natural and integral part of the development process [1].

If we look at formal methods from our viewpoint of wanting to build trust in equipment from untrusted vendors, we could choose to conclude that there is no help to be obtained from this particular research area. As long as we cannot build complete formal proofs of an entire system, we cannot guarantee that inserted malicious behaviour will be detected. We should, however, not be discouraged. At this point in the book, it is clear that a guarantee cannot be given by any methodology. This means that we should look for approaches that increase the risk of the acts of a malicious vendor being detected. Clearly, if the risk of being detected is either high, unpredictable, or both, this would be strong deterrent against inserting hidden malicious functionality into a system.

While whether the application of formal methods to a system will result in a high probability of finding unwanted functionality is debatable, formal methods do have properties that make them unpredictable. Formal methods have shown their worth in finding very subtle bugs in computer code, bugs that were exceptionally challenging to detect just through testing or code review. This property is interesting for our problem. It partly means that techniques that can be used to hide functionality, such as code obfuscation, are of limited value in the face of formal methods. Using computer-aided means of formal reasoning on the correctness of computer code therefore has the potential to make the risk of being detected impossible to estimate.

The focus of the research in formal methods has been to provide formal verification, find bugs, or generate cases for testing. Finding deliberately inserted but possibly obfuscated functionality has not been addressed specifically. Still, in the quest for a solution to our problem, further research in this direction stands out as one of the most promising paths forward. It is up to the research community to rise to the challenge.

References

1. Alur, R., Henzinger, T.A., Vardi, M.Y.: Theory in practice for system design and verification. ACM Siglog News **2**(1), 46–51 (2015)
2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall, Englewood Cliffs (1993)

3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer Science and Business Media, Berlin (2013)
4. Boldo, S., Lelay, C., Melquiond, G.: Formalization of real analysis: a survey of proof assistants and libraries. Math. Struct. Comput. Sci. 1–38 (2014)
5. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (1989)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
7. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Talcott, C.: Maude manual (version 2.7) (2015)
8. Clocksin, W., Mellish, C.S.: Programming in Prolog. Springer Science and Business Media, Springer (2003)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
10. Drzevitzky, S., Kastens, U., Platzner, M.: Proof-carrying hardware: towards runtime verification of reconfigurable modules. In: 2009 International Conference on Reconfigurable Computing and FPGAs, pp. 189–194. IEEE (2009)
11. Evans, A., France, R., Lano, K., Rumpe, B.: The UML as a formal modeling notation. The Unified Modeling Language. UML'98: Beyond the Notation, pp. 336–348. Springer, Berlin (1998)
12. Gill, A., et al.: Introduction to the Theory of Finite-State Machines. McGraw-Hill, Maidenheach (1962)
13. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta informatica **10**(1), 27–52 (1978)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
15. Hoare, C.A.R.: Communicating Sequential Processes. Springer, Berlin (1978)
16. Lamport, L.: The 'hoare logic' of concurrent programs. Acta Informatica **14**(1), 21–37 (1980)
17. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1980)
18. Milner, R.: Communicating and Mobile Systems: The Pi Calculus. Cambridge university press, Cambridge (1999)
19. Necula, G., Lee, P.: Proof-carrying code. Technical report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University (1996)
20. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. Automated Deduction—CADE-11, pp. 748–752. Springer, Berlin (1992)
21. Paulson, L.C.: Isabelle: A Generic Theorem Prover, vol. 828. Springer Science and Business Media, Berlin (1994)
22. Pierik, C., De Boer, F.S.: A syntax-directed hoare logic for object-oriented programming concepts. Formal Methods for Open Object-Based Distributed Systems, pp. 64–78. Springer, Berlin (2003)
23. Pnueli, A.: The temporal logic of programs. In: Proccedings of 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 46–57. IEEE (1977)
24. Pym, D., Tofts, C.: A calculus and logic of resources and processes. Form. Asp. Comput. **18**(4), 495–517 (2006)
25. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual. The, Pearson Higher Education (2004)
26. Steele, G.: Common LISP: The Language. Elsevier, London (1990)
27. Steele Jr., G.L., Sussman, G.J.: The revised report on scheme: a dialect of lisp. Technical report, DTIC Document (1978)
28. Uzunov, A.V., Fernandez, E.B., Falkner, K.: Engineering security into distributed systems: a survey of methodologies. J. UCS **18**(20), 2920–3006 (2012)

Chapter 10
Software Quality and Quality Management

All engineering disciplines have notions of product quality. Along with these notions come mechanisms and best practices ensuring that, for a given product, each item of the product has a specified quality. Furthermore, we are used to thinking that the most critical of these quality metrics are absolute. If the product fails to meet these absolute quality metrics, the customer might have legal claims on the producer. Such quality breaches are therefore expected to be relatively rare in most engineering disciplines.

Our expectations of the quality of software are different. For software running on standard consumer equipment, we expect a stream of security updates. These updates fix security holes that clearly must have been in the product that was bought prior to the installation of the update. This state of affairs is not the result of the neglect of the software engineering community or the result of a lack of investment in quality assurance by the manufacturers. Instead, it is the result of inherent challenges in the concept of software development. The support of quality through firm notions and mathematical rigour existing in many other engineering fields has been shown to be hard to replicate in a scalable manner in software engineering (see Chap. 9).

Still, the quality assurance of software has received a massive amount of attention. In this chapter, we provide an overview of the most important developments, aiming to see if methods have been developed that can help address our question.

10.1 What is Software Quality Management?

Early in the development of computers, it became apparent that the instructions that guide the operation of machines could be stored as data. This naturally pointed towards the formation of a new notion, *software*, that encompassed the executable files in a computer system. The flexibility of separation between the physical machine itself and its operating instructions was shown to be immensely powerful and software soon developed into an engineering discipline in its own right.

Soon, however, it became evident that the engineering of software presented its own very specific challenges. Unintentional errors in the code appeared to be hard to avoid and, when created, equally hard to find and correct. Throughout the decades, a great deal of effort has been invested into finding ways to ensure the quality of developed software. These investments have provided insights into some key properties that make software quality assurance different from what we find in other engineering disciplines [14].

Complexity One way of measuring the complexity of a product is to count the number of distinct operating modes in which it is supposed to function. A piece of software may be required to work on top of different platforms, work together with different sets of other software running alongside, and be guided by a set of parameters and settings that can be freely chosen by the user. The combinatoric explosion of possible configuration settings for a software package easily runs into the millions. Traditional engineering artefacts rarely see them run into just the thousands.

Visibility Software is invisible. This means that the opportunity to find mistakes without observing their consequences is limited to very few people. In most physical products, many mistakes are impossible simply because they would be obvious to the eye of anyone looking at them.

Manufacturing The production phase is very different in software engineering, since it simply consists of making some files available, either through download or through the standardized process of producing storage media with the files on them. There is no production planning phase in which mistakes can be discovered and the manufacturing is carried out without anyone seeing the product.

Assuring the quality of software is therefore different from assuring the quality of most other products. Because of the general invisibility of software and the fact that its quality is not influenced by the process of manufacturing copies of it, software quality assurance is aimed at the production and maintenance phases. The main elements of software quality assurance are the development process, the quality model, and the implemented quality management scheme. These three notions are described in the upcoming sections.

10.2 Software Development Process

The initial *waterfall* model for software development was described in the early 1970s. Many variations of the model exist but they generally consisted of a series of stages, with the following stages in common: a requirements phase, a specification phase, an implementation phase, a testing phase, and a maintenance phase. Although the model has been ridiculed and used as an example to avoid, there is a general consensus that the tasks described in these phases remain important in any software development effort. Most of the models we have seen since, such as those in iterative, incremental, and spiral approaches, relate to the tasks identified in the original waterfall method.

One particular trend that has been influential in recent years is the inclusion of insights on human psychology and interaction patterns into the software development process. This has led to the development of so-called *agile* approaches [12], among which *Scrum* [25] is particularly popular and successful.

10.3 Software Quality Models

The goal of any software development process is to promote the quality of the software under development. The notion of *software quality* is, however, highly nontrivial and it has therefore been subject to significant research and standardization efforts. These efforts have a common starting point in the efforts of McCall et al. [8, 22] and Boehm et al. [6], where quality aspects are ordered into a hierarchy. Figure 10.1 depicts the hierarchy as defined by Boehm.

Later efforts refined and updated the underlying ideas of McCall and Boehm in several ways. In particular, the quality attributes highlighted in these early models have been argued to be heavily based on the designer's point of view. There are, however, other stakeholders, such as the user/customer and the company that develops the software as part of a business model. Depending on the role, the importance given each attribute may differ [4]. Security as a quality metric has also grown considerably

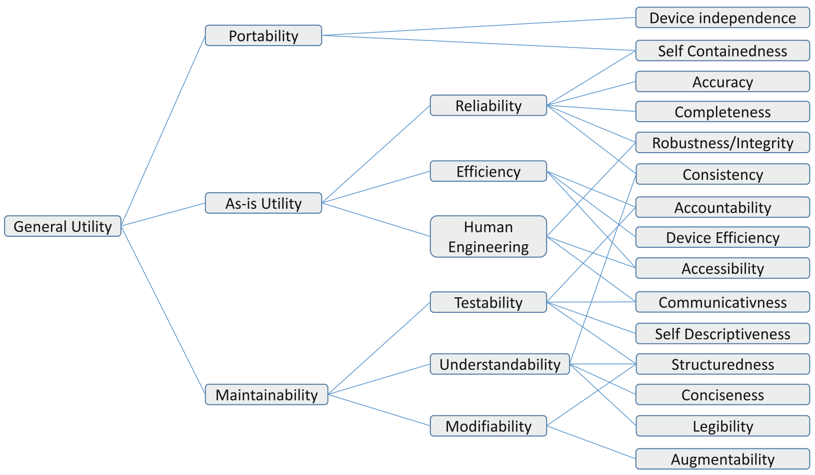

Fig. 10.1 The hierarchy of software quality characteristics of Boehm et al. [6]. The edges in the hierarchy denote necessary conditions. For example, testability, understandability, and modifiability are necessary conditions for maintainability

in importance over the decades. It is thus included in all modern quality models and has been the subject of significant academic scrutiny [15].

10.4 Software Quality Management

Quality management has a long history in production and development. Most of the concepts developed have been shown to be transferrable to software development, although with some adaptations. Alan Gillies [16] lists four principal aspects of quality management for software development:

Development procedures These include the tools used by the development team, in testing procedures, and in training staff.

Quality control This includes planning, progress meetings, documentation control, code walkthroughs, and so forth.

Quality improvement Organized staff activity aiming at establishing a quality culture amongst the staff.

Quality assurance Monitoring activity to ensure that all aspects of the quality system are carried out correctly.

There exist many different manifestations of these aspects in organizations, standards, and the academic literature. What they all have in common is that they specify the quality of work processes. This will, of course, have implications on the quality of the finished product, but hard requirements of the product itself are to be found elsewhere. We therefore move on to the metrics of software quality.

10.5 Software Quality Metrics

Whereas a quality model as described in Sect. 10.3 gives us a structure for the quality concept, the model has to be supported by metrics to be useful in a quality assurance setting. Unfortunately, there is a divide between the qualitative notions in a model and our ability to accurately capture these notions in numbers. It is a sad fact that the quality of software cannot be measured in terms of absolute and unambiguous scales [16].

Depending on the quality notions under scrutiny, the definitions and collection of metrics differ significantly in complexity. Whereas portability is a quality characteristic that can largely be captured by the frequency and number of platform-dependent constructs in the code, the notion of user friendliness is typically captured with metrics involving questionnaires and correctness/robustness can be captured by logging the number of post-release bugs corrected.

The main interest of this book concerns the metrics that relate to security. Since security has been included as a separate quality characteristic in most quality models, the world has witnessed many attempts to capture security in a quantitative way. Still, as pointed out by Jansen [21], we have a long way to go and it is not likely that all aspects of the problem are resolvable. One reason for this is that our understanding of security does not degrade gracefully with the number of security flaws in the code. With one flaw, all security could be lost and the second flaw thus adds very little to the situation. Although this is not true in all scenarios, it is clearly true in ours. If a system vendor has included a hidden backdoor in the code, the number of backdoors included in total is of little significance.

10.6 Standards

The notion of quality management is intrinsically elusive, since it has different meanings for different people, in different roles, in different situations, and at different times. This issue presents problems in any industry that needs to be able to communicate with some degree of accuracy and efficiency how quality is ensured in an organization or for a product. A series of standards are being developed to address these problems and some of the ones used in the development of software are listed below [15].

ISO 9001 This is a general standard applicable to any organization in any line of business. It defines a set of operation processes and proposes designing, documenting, implementing, monitoring, and continuously improving these operation processes [17].

ISO/IEC 9126 This standard contains a quality model for software and a set of metrics to support the model. It was first issued in 1991 [20].

ISO/IEC 25010 In 2011, this standard replaced ISO 9126. It contains a modernized set of quality attributes and, in particular, attributes related to security were given more attention [18].

ISO/IEC 15504 This standard, which is sometimes referred to as SPICE, covers a broad set of processes involved in software acquisition, development, operation, supply, maintenance, and support.

CMM/CMMI The term *CMM* stands for Capability Maturity Model and specifies software development processes. It is structured such that a development organization can be defined as belonging to one of a set of *maturity levels*, depending on the processes it has in place. The term *CMMI* stands for Capability Maturity Model Integration, which superseded CMM [10].

ISO 27001 This model was specifically designed for the certification of information security processes and is part of a series of standards related to information security [19]. Like ISO/IEC 25010, ISO/IEC 15504, and CMM/CMMI, ISO 27001 is process centric. This means that the models are based on the underlying assumption that if the process is correct, the outcome will be satisfactory [16].

The use of such standards and their certification arrangements is likely to have had positive effects on the quality of software. It is also reasonable to assume that the standards that focus on security aspects, such as the ISO-27000 series and Common

Criteria, have contributed to the development of secure software and to the safe operation of information and communications technology installations. Still, it is important to note that the contribution of these standards lies in the structured use of methods that have been described and developed elsewhere. Therefore, if our problem – namely, verifying that an equipment vendor is indeed performing malicious acts against us – cannot be solved by existing technical approaches, then structuring these approaches into a standardized framework is not likely to help. Neither is the certification of an organization or a piece of software according to the same standards. In our quest for a solution, we must therefore look at the strengths of our technical methods. Only when they are strong enough can we benefit from making them into a standard.

10.7 Common Criteria (ISO/IEC 15408)

The *Common Criteria* [1] is the result of a joint effort by many nations. They comprise a standard that requires our particular attention for two reasons. First, like ISO 27001, it is aimed specifically at security. Second, unlike ISO 27001, it contains requirements for products, not only processes.

The key concepts in this standard are the *protection profiles* by which a user specifies security requirements. A vendor claiming to have a product meeting these requirements can have the product tested for compliance in independent accredited testing laboratories. The rigour of the testing can be adapted to the criticality of the component in question and, for this purpose, seven Evaluation Assurance Levels (EALs) have been defined. The lowest of these, EAL1, requires only functional testing for correct operation, whereas EAL7 mandates full formal verification of the source code:

- EAL1: Functionally tested.
- EAL2: Structurally tested.
- EAL3: Methodically tested and checked.
- EAL4: Methodically designed, tested, and reviewed.
- EAL5: Semiformally designed and tested.
- EAL6: Design semiformally verified and tested.
- EAL7: Design formally verified and tested.

The wide adoption of Common Criteria has been a major step towards more secure information technology systems. Still, nothing in this effort addresses cases in which the company developing the system injects unwanted functionality. The highest assurance level, EAL7, is, at the time of writing, considered prohibitively costly for most development projects for reasons discussed in Chap. 9. Furthermore, evaluation focuses primarily on assessing documentation, which ultimately has to be provided by the company that is not to be trusted, and secondarily on functional testing, which we concluded above can be easily fooled. So, at the end of the day, we have to draw the same conclusion for Common Criteria as we did for the other standards. Structuring existing methods into a framework will not solve our problem unless there is a solution in the methods themselves. In the following sections, we cover the basic methods that make up the assurance levels of Common Criteria.

10.8 Software Testing

Testing remains the most frequently used technique for the validation of the functional quality of a piece of software. It has been claimed that, in a typical development project, more than 50% of the total cost is expended in testing and this figure has remained more or less stable since the early days of programming [24]. The importance of testing in the development of stable and trustworthy software has led to a large body of research in the field. Still, we have not been able to turn testing into an exact science in the sense that there exist test methodologies with the guaranteed ability to find all faults. There are several reasons for this.

First, for most software of any complexity, the combination of stimuli that the software should be able to handle is subject to a combinatorial explosion that prevents the tester from exhausting all possible test scenarios. This means that an important part of the testing effort is to choose the set of tests the software should undergo [2]. The goal of this test set could be to cover all branches of the code or – for some definition of input equivalence – all the equivalence classes of input parameters.

Most serious testing situations involve such a large set of test cases that they will have to be carried out automatically. This brings us to the second problem of testing, namely, that of automatically recognizing incorrect behaviour. In the field of software testing, this is often referred to as the *test oracle problem* [3]. In complex applications, this problem is frequently a major obstacle in the testing process.

Some software systems are actually deemed untestable. These are systems where exact test oracles cannot be defined or where the system interacts with the physical environment in such complex ways that it is impossible to adequately cover all the test cases. For these cases, building a model of the system and then performing tests on the model instead of the software itself has been proposed [7]. This concept is quite similar to model checking, which is discussed in Sect. 10.9.

Although software testing is and remains the main vehicle for the validation of software functionality, the state of the art falls short of helping us when facing a potentially dishonest equipment vendor. Triggering unwanted software can be accomplished by arbitrarily complex sets and sequences of stimuli, meaning that the likelihood of being detected in a test case can be made arbitrarily small. Furthermore, model testing is not likely to help us, since the malicious functionality will clearly not be communicated in a way that will let it become part of the model. For testing to be of help to us, we will need a scientific breakthrough that is not on the horizon at the time of this writing.

10.9 Verification Through Formal Methods

One of the methods referred to in the Common Criteria is code verification. This consists of writing an accurate specification of how the software should behave and performing a rigorous mathematical proof that the piece of software actually conforms to the specification. The correctness of this proof can in turn be checked automatically by a relatively simple program. In Chap. 9, we discussed formal methods in some detail and concluded that both hope and despair are associated with this approach.

The hope is related to the fact that the approach itself carries the promise of being watertight. If there is a formal and verifiable proof that there are no malicious actions in the software code, then a mathematical certainty has been established. The despair is due to two facts. First, it can be argued that formal methods only move the complexity from understanding the software to understanding and proofreading the specifications. The second – and this is the more serious one – full formal specification and verification are considered prohibitively costly for larger systems [11]. This is further supported by the fact that very few pieces of software have been reported to have undergone full formal verification as specified in the highest assurance level, EAL7, of Common Criteria.

One particular development that seems to address both the complexity of specification and the cost of formal verification is model checking [9]. This consists of building an abstract model of the software that is sufficiently small so that its state space can be made the subject of an exhaustive search. Although this is a promising approach, our problem pops up again in the question of how to build the model of the software. This task is intrinsically difficult and has to be carried out by human experts [5]. The likelihood of an intentional backdoor in the code showing up in the model is therefore quite similar to the likelihood of it being detected by reviewing the code itself. Model checking is therefore not a silver bullet to our problem, so we move on to seeing if code review can help.

10.10 Code Review

Code reviews have been performed in security engineering for a long time. The goal of code reviews in security is to look for known classes of vulnerabilities that typically result from programming mistakes. These reviews can be performed as part of the development process or take place entirely after the system has been developed. There are several types of reviews, depending on which code is available. Down et al. [11] mention the following alternatives, varying in the information available to the tester:

- Source only.
- Binary only.
- Both source and binary.

- Binary with debugging information.
- Black box, where only the running application is available.

As argued in Chap. 4, a review of the source code alone gives a dishonest equipment vendor full opportunity to include malicious code through the compiler or other development tools. In the case of the black box, we are, in principle, left with testing, a situation we discussed in Sect. 10.8. This leaves us with the testing situations in which the binaries are available.

Code reviews of binaries have many the same strengths and weaknesses as formal methods. On one hand, they hold the promise that, when carried out flawlessly and on a complete binary representation of a piece of software, all hidden malicious code can be found. Unfortunately, there are two major shortcomings: one is that finding all hidden malicious code is generally impossible, as shown in Chap. 5. Even finding some instances will be extremely challenging if the malicious code has been obfuscated, as described in Chap. 7. The second shortcoming is one of scale. Down et al. [11] estimate that an experienced code reviewer can analyse between 100 and 1,000 lines of code per day. Morrison et al. [23] estimate that a full analysis of the Windows code base should take between 35 and 350 person–years. To make things worse, these figures are based on the assumption that it is the source code that is to be analysed, not the binaries; the real costs are therefore likely to be higher.

One way to handle this problem is to perform a code review on only parts of the software product. Which parts are to be selected for scrutiny could be chosen by experts or by a model that helps choose the parts that are more likely to contain malicious code. Such *defect prediction models* have been used for some time in software engineering to look for unintended faults and a range of similar models have been proposed for vulnerability prediction. The value of the vulnerability prediction models is currently under debate [23]. We do not need to take a stand in that discussion. Rather, we observe that the models try to identify modules that are more likely to contain unintended vulnerabilities. If the vulnerabilities are intended, however, and we can assume that they are hidden with some intelligence, then a model is not likely to be able to identify the hiding places. Even if such a model existed, it would have to be kept secret to remain effective.

It is thus a sad fact that rigorous code reviews and vulnerability detection models also leave ample opportunity for a dishonest equipment producer to include malicious code in its products.

10.11 Discussion

Fuggetta and Di Nitto [13] took a historical view of what was perceived as the most important trends in software process research in 2000 and compared this to what was arguably most important in 2014. One of the key observations is that security has gained significantly in importance and, consequently, the community should ‘identify and assess the major issues, propose ways to monitor and manage threats, and assess the impact that these problems can have on software development activities.' Although the statement was intended for scenarios with unintended security holes and where the perpetrator is a third party, it does shed light on the extent to which software development processes and quality assurance are promising places to look for solutions to the problem of untrusted equipment vendors and system integrators. If these tools and methods are inadequate to help the developers themselves find unintended security holes, they are probably be a far cry from helping people outside of the development team to find intentional and obfuscated security holes left there by a dishonest product developer.

The findings in this chapter confirm the above observation. We have covered the concepts of development processes, quality models, and quality management and have found that they relate too indirectly to the code that runs on the machine to qualify as adequate entry points in the verification of absence from deliberately inserted malware. The field of software quality metrics in some aspects directly relates to the source code but, unfortunately, it is doubtful that all aspects of security are measurable. In addition, counting the number of bugs found has still not given us bug-free software; thus, it is not likely that counting security flaws will provide any guidance in finding deliberately inserted malware.

Methods that go deep into the code running on the machine include testing, code reviews, and formal methods. In the present state of the art, formal methods do not scale to the program sizes we require, code review is error prone and too time-consuming to be watertight, and testing has limited value because of the problem of hitting the right test vector that triggers the malicious behaviour. Even when triggered, recognizing the malicious behaviour may be highly nontrivial. It is clear that these methods still leave ample space for dishonest equipment providers to insert unwanted functionality without any real danger of being exposed.

Although these approaches still fall far short of solving our problem, they remain a viable starting point for research on the topic. In particular, we hope that, at some time in the future, the combination of formal methods, careful binary code review, and testing will increase to a significant level the risk of being exposed if you include malware in your products. However, we have a long way to go and a great deal of research must be done before that point is reached.

References

1. http://www.commoncriteriaportal.org
2. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. **36**(6), 742–762 (2010)
3. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
4. Berander, P., Damm, L.O., Eriksson, J., Gorschek, T., Henningsson, K., Jönsson, P., Kågström, S., Milicic, D., Mårtensson, F., Rönkkö, K., et al.: Software quality attributes and trade-offs. Blekinge Institute of Technology (2005)

5. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer Science and Business Media, Berlin (2013)
6. Boehm, B.W., Brown, J.R., Kaspar, H.: Characteristics of Software Quality. North-Holland, Amsterdam (1978)
7. Briand, L., Nejati, S., Sabetzadeh, M., Bianculli, D.: Testing the untestable: model testing of complex software-intensive systems. In: Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), ACM (2016)
8. Cavano, J.P., McCall, J.A.: A framework for the measurement of software quality. In: ACM SIGMETRICS Performance Evaluation Review, vol. 7, pp. 133–139. ACM (1978)
9. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
10. CMMI Product Team: CMMI for development, version 1.2 (2006)
11. Down, M., McDonald, J., Schuh, J.: The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Pearson Education, Boston (2006)
12. Fowler, M., Highsmith, J.: The agile manifesto, Softw. Dev. **9**(8), 28–35 (2001)
13. Fuggetta, A., Di Nitto, E.: Software process. In: Proceedings of the on Future of Software Engineering, pp. 1–12. ACM (2014)
14. Galin, D.: Software Quality Assurance: From Theory to Implementation. Pearson education, Boston (2004)
15. García-Mireles, G.A., Moraga, M.Á., García, F., Piattini, M.: Approaches to promote product quality within software process improvement initiatives: a mapping study. J. Syst. Softw. **103**, 150–166 (2015)
16. Gillies, A.: Software Quality: Theory and Management (2011). https://www.lulu.com
17. ISO 9001:2015 quality management systems - requirements
18. ISO/IEC 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models
19. ISO/IEC 27001:2013 information technology – security techniques – information security management systems – requirements
20. ISO/IEC 9126-1:2001 software engineering–product quality–part 1: quality model
21. Jansen, W.: Directions in Security Metrics Research (2009)
22. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in software quality. volume I. Concepts and definitions of software quality. Technical report, DTIC Document (1977)
23. Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security. ACM–Association for Computing Machinery (2015)
24. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, New York (2011)
25. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice-Hall, Englewood Cliffs (2001)

Chapter 11
Containment of Untrusted Modules

In previous chapters, we established that the problem of fully verifying information and communications technology (ICT) equipment from an untrusted vendor is currently not feasible. As long as full and unconditional trust does not prevail in the world, we will have to build and maintain digital infrastructures consisting of equipment we do not fully trust and equipment consisting of modules we do not fully trust.

In the real world, we handle persons and organizations we do not trust by trying to contain them. First, we make efforts to detect if they defect on us and, second, if they defect on us, we work to limit the impact of the defection and then strive to manage without the defectors in the future. The counterparts to such strategies in the digital world are discussed in this chapter.

11.1 Overview

Many of the chapters in this book are concerned with fields of research that are motivated by problems besides ours but which still have some bearing on the problem of untrusted vendors. This chapter is different, in that it is motivated by an envisaged solution to the problem rather than a research field.

This envisaged solution – containment of the equipment that we do not trust – consists of two parts. The first part is the detection of misbehaviour. This problem area is discussed thoroughly throughout the book. In separate chapters, we have discussed detection through formal methods, reverse engineering, the static analysis of code and hardware, the dynamic analysis of systems, and software quality management. The other part of the solution is that of replacing or somehow handling misbehaving equipment. For this part of the solution, we are in the fortunate situation in which the mechanisms we need have been studied and developed over decades. This work has been conducted under the headline of fault tolerance rather than security, but the developments work well in our situation nevertheless. In the upcoming sections, we touch upon some of this work. The field is rich and solutions have been developed for so many different areas that this chapter cannot do justice to all of them. Our ambition is therefore to give a high-level overview but still make it sufficiently detailed to assess whether the containment of ICT equipment from untrusted vendors is a viable path forward.

11.2 Partial Failures and Fault Models

In the early days of computer programming, a program was a monolithic entity that either worked or completely collapsed. Whenever one part of the program ran into a problem – for example, division of a number by zero – the entire program stopped executing. Later, when computer architectures allowed for several parallel processes and computer networks allowed for distributed systems, this situation changed. Systems where one component or process failed while the others continued to run became something we had to deal with. This gave birth to the notion of *partial failures*.

One basic problem in the area of partial failures is that there is a wide range of different ways in which a component can fail. It can fail suddenly and immediately by simply halting its execution. It can fail slowly by performing inconsistent actions before it halts and these actions may or may not leave the entire system in an inconsistent state. Furthermore, the faults may or may not be detectable by the other components in the system or they may be detectable long after the fault occurred. The different ways by which a component can fail span out a set of *fault models*, each of which captures a specific class of ways to malfunction [8].

In Chap. 1, we discussed the different deeds we fear that a dishonest vendor might commit. In particular, we mentioned the introduction of kill switches that would allow an adversary to render the equipment nonfunctional at a time of choosing. In the theory of fault models, this would be close to the *fail-stop* model [18]. The fail-stop model captures cases in which a system component crashes, the crash is detectable by the other components of the system, and the component works in a benign and correct manner until it crashes. Other scenarios we were worried about are where the vendor uses the equipment it has sold for either espionage or fraud. In those cases, the malevolent components will appear to function correctly while they are actually not. In addition to performing the tasks expected of them, they leak information to the outside world or behave in a fraudulent manner. In the world of fault models, this would be categorized as a *Byzantine* fault [14].

Theoretical solutions for both fault models are easily derived through capacity and functionality replication. For a fail-stop failure, there is need for one additional module with the same functionality that can take over the job of the failing one. For Byzantine faults, the situation is more complex. Detection of the fault in itself is a challenge and the situation usually comes down to having multiple modules doing the same job, with a comparison of the output. The number of replicated modules needed will depend on the number of defecting modules one wants to be able to defend against and how the tasks are distributed among the modules. One easily understandable result is that if all the modules have identical information and identical tasks, $2N + 1$ modules are needed in total to successfully detect and isolate a coordinated Byzantine fault in N of the modules. This number will guarantee that trustworthy replicas will be able to outvote the replicas that have defected. More complex situations can be described, and they will lead to different ratios between the trustworthy and defected modules.

Fault models, in general, and fail-stop and Byzantine faults, in particular, can easily be described in a simple, idealized way and, indeed, this is what we have done above. Both models are, however, deeply problematic from a practical point of view. For example, it is very hard to implement a module so that it guarantees adherence to the fail-stop model. This would require it to never do anything wrong. In particular, this would require that the module itself detect that it is starting to malfunction, for example, because a bit has flipped in memory, and then stops all operations before it does anything wrong. This is a highly nontrivial task to implement. In real life, a module will most likely undergo a period with a Byzantine fault before it stops [19]. This is especially likely to be the case if the fault itself has been deliberately introduced to cause harm. Indeed, most fault models can themselves be fooled by an intelligent adversary. The obvious answer to this problem would be to treat all faults as Byzantine.

Byzantine faults do, however, have problems of their own. The algorithms that handle Byzantine faults generally build on the notion of an *atomic broadcast*. An atomic broadcast is a message-passing service that allows a module to send the same message to all the other modules and with a guarantee that the messages arrive in the same order at every module. Unfortunately, atomic broadcasts, like fail-stop modules, are easy to describe but generally impossible to implement in a real system.

In spite of these shortcomings, the study of fault models and the approximative implementations of the protocols that handle them have resulted in remarkably complex, robust, and flexible systems in the real world. Our methods for handling failing modules in complex systems is perhaps the most promising of the existing technologies when it comes to controlling the effects of buying equipment from untrusted vendors.

11.3 Erlang: A Programming Language Supporting Containment

In our discussion, the significance of fault models relates to what a dishonest component might do and how we could discover that it is not acting in the interest of the system's owner. After the detection of dishonest actions, the next step would be to actually proceed with the containment of the malicious components. This could, in most settings, imply the replacement of the dishonest components with components that can presumably be trusted.

Erlang is a programming language that was developed at Ericsson in the 1980s. One of its motivating intentions was the need to develop highly reliable software systems – particularly for telephony applications – taking into account that neither hardware nor software will ever be perfect. Erlang has many distinctive features, but the most important ones to us are the following [1, 2]:

- The core component in an Erlang system is a process. This process is strongly isolated from other processes; thus, its existence or performance is not, per se, dependent on the perfect operation of the other processes.
- Processes can only interact through message passing and there is no shared memory. This means that the memory state of one process cannot be corrupted by another process.
- The addition and removal of processes are lightweight operations.

These mechanisms have proven to be very powerful in the creation of robust parallel and distributed systems and Erlang is still an important language in the development of such systems [4]. For us, this means that one of the core mechanisms for the containment of untrusted modules – the replacement of misbehaving pieces of software at runtime – has been studied for a long time and that we can consider this problem solved to a significant extent.

11.4 Microservices: An Architecture Model Supporting Containment

There have also been developments within software architecture that are – at least partly – motivated by the need for the containment of malfunctioning components. A recent development is called *microservices* [17]. Microservices are sometimes referred to as a dialect of service-oriented architecture [12], but they have important differences. We will return to these below.

A microservice is the name of the architecture model and its components alike. A software system based on this model consists of a set of microservices, where each microservice has the following set of features:

- It is a relatively small piece of software that performs a well-defined service. Exactly what is meant by the term *small* is unclear, but a statement that is repeated in many places should be small enough so that it can be programmed from scratch in two weeks. This means that it must have limited and focused functionality that can be observed and understood.
- It is autonomous in the sense that its existence is not dependent on other microservices. Such autonomy can be implemented by letting it be its own operating system process or an isolated service on a ‘platform as a service’.
- It can be independently developed and deployed and is not restricted to a given programming language or communication protocol.

- It communicates with other microservices through messages. It exposes an application programming interface (API) and collaborating microservices use this API for interaction.
- It can be changed or even replaced without affecting the services using it.

These features are a good match for the problem we are grappling with in this book. The emphasis on the size of microservices sets them apart from mainstream service-oriented architectures. For us, this is an interesting distinction. A small service that performs a relatively simple task is easy to observe and monitor while it is working. If its complexity is sufficiently low for it to be reprogrammed in two weeks, the task of reverse engineering it also falls within the limits of tractability (see Chap. 6). This means that this architecture model is well suited for the detection of components written with malicious intent [15].

Another interesting aspect of microservices is that they can be independently developed and deployed. Since the development of each microservice is relatively low cost, it is possible to develop several versions of the same service independently of each other. These can use different programming languages and be based on distinct libraries. For us, this means that microservices open up the possibility of heterogeneity in the set of components. Carefully designed, such heterogeneity could be used to detect malicious components at runtime. Last, but not least, the autonomy and replaceability of a microservice allow us to contain and later replace a malfunctioning microservice [17].

Microservices are an important development when it comes to understanding the problem of verifying equipment bought from an untrusted vendor. Although designed for another purpose, it is an architecture model that captures all of our challenges if we want to encapsulate untrusted equipment. It addresses the detection of malicious behaviour through the size and observability of the modules, it handles the need to have alternatives that can be trusted through heterogeneity, and, like Erlang, it allows for the easy replacement of untrusted modules.

It is important to state, however, that microservices are not yet a full solution to our problem. The apparent simplicity of the microservice model hides the fact that the size of a complex system reappears in the number of microservices needed to implement it [6]. The complexity of the entire system will thus reappear in a complex web of collaborating microservices. The challenge of analysing a system down to the bottom is therefore no simpler in a microservice architecture than it is in any other system. In addition, when we buy a piece of equipment, it consists of both software and hardware and the microservice approach is part of software development philosophy only. Finally, we generally have no control over the architecture model used by the vendor.

Nevertheless, future research into microservices and their properties in terms of fault tolerance stands out as one of the few promising avenues when it comes to tackling the problem we address. Insights from this field of research hold the promise of forcefully addressing the question of trust between buyers and vendors of electronic equipment.

11.5 Hardware Containment

Handling untrusted hardware consists of the same two tasks as for software: First, the fact that a given piece of hardware is not to be trusted needs to be detected and, second, the distrusted components will need to be contained. In hardware as well, it is the detection task that is the most difficult. We have elaborated on methods of detection in Chaps. 7 and 8 and concluded that this is a highly challenging task [23]. In terms of hardware, some are even labelling it the problem from hell, deeming it generally impossible [21]; therefore, the problem of untrusted hardware is often overlooked [20].

On the other hand, the actual containment of pieces of hardware that are no longer to be trusted is a topic that has been successfully studied since a long time. Solutions for disk arrays that allow for the removal and hot swapping of disks are one example [7]. Other examples are routing techniques that route around faulty routers [13], switching techniques that handle faulty switching components [9], techniques that handle misbehaving portions of disks or memory [5], and load-balancing techniques that keep misbehaving CPUs out of the equation [16]. There also exist methods that let redundant pieces of logic on a chip take over when a limited area of the chip has failed [22].

The replacement of failing components or graceful degradation by shutting down failing parts is regularly carried out in all complex ICT systems. In tightly coupled hardware systems, the possibility of hot swapping, that is, replacing components while the system is running, is a research field in itself since decades. Unlike for software, however, the fact that hardware is a physical thing requires that provisions for hot swapping be made in advance. The replacement of a chip's functionality with the same functionality on another part of the chip is limited by what is on the chip in the first place. Routing around components in a network or the replacement of a disk in an array have to be carried out within the framework of what disjoint paths exist in the network or what other trusted disks are present in the array, respectively. Still, for hardware as well, we can conclude that one part of the problem of containment, that of isolating untrusted parts, is largely covered by existing mechanisms and technology for fault tolerance.

11.6 Discussion

The scheme for the containment of untrusted ICT systems or components thereof consists of two parts. First, mechanisms must be in place to detect that components are misbehaving and, second, mechanisms must be in place that isolate the misbehaving components. In this chapter, we have demonstrated that the latter of these challenges is tractable. Fault-tolerant computer systems have been studied for decades and mechanisms for making the system independent of components that no longer work have celebrated many successes. We therefore have a good grasp of how to handle the actual containment. The detection part is, however, far more involved. In Chaps. 7 and 8, we discussed static analysis methods and methods that detect malicious behaviour at runtime, respectively. Our conclusion from these chapters is that the developer or manufacturer of products has all the freedom needed to include stealthy malicious functionality into a product.

This could lead us to conclude that containment, as an approach to handling untrusted electronic equipment, is futile. Our conclusion is, however, completely the opposite. The containment of untrusted modules is the most promising approach we have to our problem, first, because all the mechanisms that have been developed for fault-tolerant systems in the last decades fit nicely with our need to perform the containment and, second, because the detection of misbehaving components need not come from ICT technology itself. Indeed, the most communicated incidents of intentional fraud we know of were discovered in ways that had nothing to do with ICT. The Volkswagen case, where electronic circuits controlling diesel engines reduced the engine emissions once it detected that it was being monitored, was discovered by analysing the car's behaviour [3] and an insider disclosed that routers and servers manufactured by Cisco were manipulated by the National Security Agency to send Internet traffic back to them [10]. Containment therefore remains an important mechanism, regardless of our lack of ability to analyse the behaviour of ICT equipment.

All mechanisms for containment we have seen come at a cost. If your untrusted piece of equipment consists of software alone, then alternative software with equivalent functionality must exist and it must be ready at hand. For some pieces of standardized high-volume products, alternatives are likely to exist. In addition, in a microservice system, several generations of each service are likely to be available. For tailor-made monolithic software for specialized tasks, the development of completely independent alternatives will probably double the price of development. For hardware, one will have to buy equipment from different vendors. This means that, apart from the cost of having redundant components, one will have to cope with the extra cost of running and integrating different product lines. This will clearly not always be worth the cost. Still, for highly critical systems underpinning critical infrastructures supporting large populations, we can consider this price low compared to the potential consequences.

Based on the above, we conclude that further research on containment is one of the most promising ways forward to tackle the problem. In this future research path, it is important to keep in mind that the containment of modules from an untrusted vendor is, in some respects, very different from the containment of faulty modules. When working only on fault tolerance, one can assume that ICT equipment fails according to a statistical distribution. Each component will work well for some time and then, for some reason, it will start to malfunction independently of all the other components. Then, when you fix whatever the configuration problem is, restart a piece of equipment, or replace a piece of hardware, the system is good to go again. When you detect that a vendor of ICT equipment is not to be trusted, however, then no statistical distribution can be assumed. Furthermore, all the components made by this vendor become untrusted. This can affect a large portion of a system and, if the equipment is part of an infrastructure one depends on, one has to be able to cope with this situation for some time.

The above brings to mind two lines of research that should be pursued with more intensity than is currently the case: First, we must develop hardware and software architectures that allow a system to run with malicious components in its midst without letting them cause substantial harm. Recent developments point to the notion of anti-fragility as a particularly promising avenue to explore [11]. Second, for critical infrastructures, more effort must be directed to handling large-scale disasters as opposed to single faults. The large number of components of an infrastructure that can suddenly become untrustworthy as a result of a vendor being deemed untrustworthy makes this case different from what is generally studied under the label of fault tolerance.

References

1. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, The Royal Institute of Technology Stockholm, Sweden (2003)
2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall, New Jersey (1993)
3. BBC: http://www.bbc.com/news/business-34324772
4. Cesarini, F., Thompson, S.: Erlang Programming. "O'Reilly Media, Inc.", Massachusetts (2009)
5. Deyring, K.P.: Management of defect areas in recording media. US Patent 5,075,804, (1991)
6. Fowler, S.J.: Production-Ready Microservices (2016)
7. Ganger, G.R., Worthington, B.L., Hou, R.Y., Patt, Y.N.: Disk arrays: high-performance, high-reliability storage subsystems. Computer **27**(3), 30–36 (1994)
8. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv. (CSUR) **31**(1), 1–26 (1999)
9. Gomez, M.E., Nordbotten, N.A., Flich, J., Lopez, P., Robles, A., Duato, J., Skeie, T., Lysne, O.: A routing methodology for achieving fault tolerance in direct networks. IEEE Trans. Comput. **55**(4), 400–415 (2006)
10. Greenwald, G.: No place to Hide: Edward Snowden, the NSA, and the US Surveillance State. Macmillan (2014)
11. Hole, J.K.: Anti-Fragile ICT Systems. Springer, Verlag GmbH (2016)
12. Josuttis, N.M.: SOA in Practice: The Art of Distributed System Design. "O'Reilly Media, Inc.", Massachusetts (2007)
13. Kvalbein, A., Hansen, A.F., Čičic, T., Gjessing, S., Lysne, O.: Multiple routing configurations for fast ip network recovery. IEEE/ACM Trans. Netw. (TON) **17**(2), 473–486 (2009)
14. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. (TOPLAS) **4**(3), 382–401 (1982)
15. Lysne, O., Hole, K.J., Otterstad, C., Aarseth, R., et al.: Vendor malware: detection limits and mitigation. Computer **49**(8), 62–69 (2016)
16. Nelson, M., Lim, B.H., Hutchins, G., et al.: Fast transparent migration for virtual machines. In: USENIX Annual Technical Conference, General Track, pp. 391–394 (2005)
17. Newman, S.: Building Microservices. "O'Reilly Media, Inc.", Massachusetts (2015)
18. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: an approach to designing fault-tolerant computing systems. ACM Trans. Comput. Syst. (TOCS) **1**(3), 222–238 (1983)
19. Schneider, F.B.: Byzantine generals in action: implementing fail-stop processors. ACM Trans. Comput. Syst. (TOCS) **2**(2), 145–154 (1984)

20. Sethumadhavan, S., Waksman, A., Suozzo, M., Huang, Y., Eum, J.: Trustworthy hardware from untrusted components. Commun. ACM **58**(9), 60–71 (2015)
21. Simonite, T.: NSA's own hardware backdoors may still be a "problem from hell" (2013)
22. Srinivasan, J., Adve, S.V., Bose, P., Rivers, J.A.: Exploiting structural duplication for lifetime reliability enhancement. In: ACM SIGARCH Computer Architecture News, vol. 33, pp. 520–531. IEEE Computer Society (2005)
23. Tehranipoor, M., Koushanfar, F.: A survey of hardware trojan taxonomy and detection. IEEE Des. Test Comput. **27**(1), 10–12 (2010)

Chapter 12
Summary and Way Forward

In this book, we have asked the following question: What if one or more of the providers of the core components of an information and communication technology (ICT) system are dishonest? This question has been actualized by recent discussions and events, such as the Snowden revelations, the discussions that have taken place in many Western countries on the inclusion of equipment from Chinese providers into telecommunications infrastructures, and the case of Volkswagen cars having electronics recognizing that they were being tested for emissions.

Our problem is widely distinct from the traditional problem of the prevention and detection of malware in a system, first, because the malware is already present at the time of purchase, so preventing it from entering into the system is meaningless, and, second, because there is no clean malware-free sample for comparison, something that is the basis for most effective malware detection techniques. In this book, we have wandered through the landscape of ICT knowledge. Throughout the journey, we have tried to determine how one might verify if a producer of an artefact consisting of electronic hardware and software has included unwanted functionality in the product.

12.1 Summary of Findings

The notion of trust has been heavily studied in modern philosophy since the 1980s and onwards. In most aspects of life, our relationships to the people and institutions we interact with are influenced by the amount of trust we have in the people and institutions in question. Our need to verify their actions depends on our earlier history with them, their reputation among other people we trust, and the consequences for ourselves and the other party if they defect on us. In Chap. 2, we discuss the trust relationship between a buyer and a vendor of electronic equipment. Ultimately, we find that meaningful trust in this relationship can only be attained through an ability to verify the functionality of the equipment that is being sold. Most of this book is therefore devoted to the technical verification of the properties of electronic equipment.

In Chap. 3, we embarked on the notion of a system, pointing out that it spans many dimensions. The most obvious dimension is that going from the user interface of a given system – through many layers of software, firmware, and hardware – down to the physical phenomena that allow us to build the machines in the first place. Another dimension emerges from the distributed nature of most complex systems. These are typically constituted by a series of subsystems, many of which physically reside on different machines. In addition, these subsystems may be owned and run by different organizations in different countries. Since the outsourcing of services and the usage of cloud facilities are becoming commonplace, the diversity of the ownership of subsystems is likely to increase.

A usual assumption when looking for weaknesses in computer systems is that the weakness, or, in our case, the security hole, will be visible from the source code. Several companies have therefore made efforts to let their customers study the source code of their systems. This is a strong act of goodwill but, unfortunately, we cannot assume that the absence of malicious actions in the source code guarantees freedom from malicious behaviour in the equipment itself. In Chap. 4, we use a result from the 1980s to show that malicious executable code and electronic circuits can be introduced by the tools used by developers. In these cases, not even the developers of the product need be aware that malicious code is being embedded into the product they are building. Therefore, to look for malicious code or malicious circuits, we have to study gate-level electronics and the executable code.

One reason reverse engineering and the search for malware are challenging is that there are theoretical limits to what can be achieved through code analysis. This is studied in Chap. 5. We showed proof that decision procedures for detecting malicious functionality are a mathematical impossibility.

Understanding and making sense of executable code and gate-level definitions of hardware in life-size systems is a massive task. There is, however, an active community with well-developed tools for the reverse engineering of systems. The expertise of this community is used in analysing the effects of known malware, as well as in reverse engineering programmer interfaces for software development. We provided an overview of the classes of tools used for the reverse engineering of hardware and software in Chap. 6. Although these tools have shown their worth in analysing limited problems, they are a far cry from bringing us close to analysing entire systems in the full complexity illustrated in Chap. 3.

The detection of intrusion is perhaps the most developed field in computer security. Such detection consists of scanning a system in the search for a computer virus, Trojan, or unwanted active processes placed there by third parties. Chapter 7 gave a high-level overview of the different known methods. The success of this field is one of the finest stories in the tale of computer security. Still, its methods fall short of solving our problem. The most efficient modus for intrusion detection is to scan a system for the signature of a piece of code that is not supposed to exist in a healthy system. When malware is inserted into a system at the development phase, no system will be healthy. The distinction between code that is supposed to be there and code that is not disappears.

An alternative to analysing an electronic product before it is powered on is to start it up, let it run, and then study its behaviour. In Chap. 8, we explained that this can be done in one of two ways. One is to build a controlled environment around the piece of equipment, a so-called sandbox This has the advantage of containing the malicious acts the equipment could carry out so that no real harm is done. Another benefit of this approach is that the sandbox itself can conduct a systematic test of how the piece of equipment acts under different stimuli. The main weaknesses of sandboxes are, first, that they only run for a limited time and, second, that malicious pieces of code may try to check if it is running in a sandbox before it exposes its malicious behaviour. These weaknesses form separate battlefields where malware creators try to hide malicious functionality from the sandbox and the sandboxes creators work on tricking the malware into exposing itself.

The second approach to dynamic analysis consists of putting the device into ordinary production outside of a sandbox and then observing its behaviour. By doing so, we avoid sandboxes' drawbacks of time limitations and test detection. On the other hand, malicious acts performed by the device will have real-life effects. The severity of these effects will differ, depending on what the effects were. If the malicious effect is to render a piece of equipment unusable, then it will be too late by the time it is detected. If it is to leak confidential information, it may not be too late but, depending on the criticality of the function of the equipment and alternative ways of providing this function, it may be hard to stop.

The key to understanding if dynamic testing – either in a sandbox or in production – can help solve the problem lies in the question of whether the malicious acts can actually be detected. As stated, acts that render the equipment useless are easily detected when they occur. When they occur in real production, however, it will be too late and, since they can be triggered by arbitrarily complex external stimuli, one cannot expect to be able to trigger them in a sandbox. As for the leakage of information, there is more bad news. It has been shown through information theory that undetectable leakage is possible, although with limited bandwidth.

Although we have presented a series of undecidability results that seem to end our analysis, several theoretical developments have promise. In the field of formal methods, which we discuss in Chap. 9, we have a plethora of logic systems for programs and deduction systems that are *sound and complete*. This means that, for every property of a program that can be expressed in the logic system, there exists a proof for this property. Although the undecidability results mean there is no systematic way the absence of malicious behaviour can be proven, it is clear that an honest developer will have a clear mental picture of this during the development phase. We can therefore envision that the programmer provides proof of absence of malicious behaviour as part of the development process. Checking the correctness of such proofs is a trivial task that can be done by a machine.

However, even formal methods have limitations. In this case, the limitations are related to the fact that the developer mainly works on source code. This source code is automatically transformed into binary code before it is executed on a machine or into gate-level structures before it is implemented in hardware. For the proofs to be watertight, they will therefore have to refer to code on another level of abstraction than what is in the developer's mind. Another difficulty is that an ICT system is generally exceptionally complex, as discussed in Chap. 3. This complexity overwhelms the current state of the art in formal methods. Finally and most importantly, the notion of sound and complete is deceptive. It leaves the impression that every property of a program can be proven. What it really means is that every property *expressible in the logic system* can be proven. A logic system that is sound and complete therefore has the limitation that properties that are undecidable in the Turing sense cannot be expressed in the system. As mentioned above, this will be the case for most of the malicious properties we are interested in finding.

We discussed in Chap. 10 the existence of various systematic approaches to the assurance of software quality. The most common one of these for computer security is called the Common Criteria and these are formed around seven quality assurance levels. Although the gist of these assurance levels is that the developer of the equipment collaborates with the customer to secure the system against third parties, they can be made relevant for our discussion as well. They do, however, encounter the same limitations as those discussed in the various chapters of this book, with regards to both the static analysis of code and dynamic testing.

If our system has only a few components that we do not trust, we could try to execute them in a controlled manner. They can be put in a sandbox environment where they cannot send information to the outside and where the information going into the components is controlled and even encrypted to ensure that all communications from the outside are scrambled before reaching the component in question. We study these mechanisms in Chap. 11. Unfortunately, they are usable only for a limited set of cases.

12.2 The Way Forward

The conclusion on the current state of the art with respect to our problem is rather bleak. If we do not trust the makers or vendors of our electronic equipment, there seems to be no framework of knowledge that adequately addresses the full complexity of the question. However, some approaches appear to be more promising than others and they should receive the additional attention that the seriousness of the problem seems to justify. We will mention some of them in the following sections.

12.2.1 Encryption

Distrust in the equipment you communicate through and collaborate with is one of basic assumptions in encryption algorithms and protocols. This is therefore an example – and possibly the only example – of a field of ICT security and robustness where the premise of the field need not be changed as a result of distrusting the equipment vendor instead of accidents or a third party. The research field has therefore resulted in numerous methods that provide a solid basis for trust in our case as well, even when communication can be leaked and even in the presence of attempted man-in-the-middle attacks.

The development, harnessing, and proliferation of strong encryption techniques should therefore be encouraged and will be an important component in the solution to the problem of untrusted equipment. Still, encryption does not solve the problem altogether, for two reasons: First, the equipment used for encryption and decryption will also consist of electronic devices. The question of who built the encryption equipment, from the transistors up to the application, will be important. If one cannot place complete trust in the people who built the encryption equipment, in all the complexity we have described in Chap. 4, then the encryption methods offer nothing. Second, while encryption promises to build defences against threats such as eavesdropping and man-in-the-middle attacks, it does not touch the problem of remotely controlled kill switches. For that problem, we must look for solutions elsewhere.

12.2.2 Formal Methods

We have argued above that formal methods currently fall short of solving the problem of fully verifying electronic equipment. In spite of this, we choose to list it as one of the research fields that should be pursued with greater intensity in the search for solutions to our problem. Our reason for this is that, of all the existing approaches to quality and the scrutiny of code – such as software quality management, reverse engineering, code review, and static and dynamic analysis – the field of formal methods is the only one that holds the promise of being able to issue *guarantees* of the absence of unwanted functionality.

One particularly promising approach lies in the combination of specification-based techniques in defining proper benign system behaviour, discussed in Sect. 7.8, and proof-carrying code, discussed in Sect. 9.8. Specification-based techniques have the advantage over other static detection methods of not requiring a clean, uninfected sample of the system. Proof-carrying code has the advantage that the computational challenges of finding the proof of a system's correctness are transferred from the system's customer to its provider. The customer of the system will therefore only have to carry out proof-checking, usually a fairly simple computational task.

Several problems, however, need to be addressed through research. First, much research in the field has focused on providing specifications in which a piece of code performs some specified predefined actions. Our problem is somewhat different, in that we need to be able to prove that a system is free of unwanted side effects, such as the leakage of information or the halting of operations. The latter problem is particularly challenging, since it is well known that the halting problem is generally undecidable (see Chap. 5). A second challenge is that many of the existing results of formal methods relate to source code. As made clear in Chap. 4, it is the machine code that needs to be verified. We will therefore need, not only scalable tools to help programmers construct correctness proofs of the source code, but also compilers that create machine code and translate proofs from source code to machine code. Finally, the production of proof-carrying code is highly resource intensive. The approach is therefore most likely only applicable to a relatively small trusted computing base (see Sect. 2.6).

The verification of hardware chips is, however, challenging in a different way. Whereas the development chain from an algorithmic description of the hardware down to gate-level descriptions can be framed in by formal methods, the actual production of the chip cannot. Furthermore, as described in Sect. 6.7, determining what logic was actually placed on a chip is no easy task. Whereas formal methods seem to be an interesting prospect in solving the software side of our problem, it is hard to see how it can help us prove that malicious functionality is not being included into hardware in the production phase.

12.2.3 Heterogeneity and Containment

The most potent approach to handling untrusted electronic equipment is through containment and heterogeneity. This basically means that systems must be constructed to the extent possible so that no code and no piece of hardware is universally trusted. Unfortunately, this field has received limited attention, as discussed in Chap. 11.

The strength of the approach is that it holds the promise to handle all sides of the problem. Disjoint pieces of equipment can handle disjoint pieces of information, so that damage through information leakage is limited. Whenever a kill switch is triggered, rendering a piece of equipment out of service, other pieces of equipment can take over. Once it has become clear that parts of an infrastructure can no longer be trusted, software can be replaced and hardware isolated for later replacement.

Another strong aspect of this approach is that it can be applied to the entire technology stack. Intellectual property integrated onto a chip can be controlled and contained by other pieces of intellectual property, pieces of software can be controlled and contained by other pieces of software, and entire devices can be controlled and contained by other devices.

Still, there are problematic aspects to the state of the art in this area. The first is that most methods currently applicable were developed for the purpose of fault tolerance. This means that much of the work assumes that faults strike in an arbitrary fashion and that, at a given time, there will be few faulty components to handle. These assumptions will not necessarily hold when there is an intelligent adversary and this adversary controls a large number of components in your system. Another problematic aspect is that of cost. Duplication of functionality is costly at all levels, and heterogeneity itself will significantly increase the cost of maintenance. Finally, the containment of malicious actions is not a trivial feature to build into a system. It will require a great deal of research to understand how to do so and most likely its implementation will come at the cost of loss of performance.

Even if many problems are involved in handling untrusted vendors through heterogeneity and containment, we still see these as the most promising way forward. Unlike other conceivable approaches, there are no important parts of the problem that we can identify upfront as impossible to handle. In particular, when we are challenged by possible kill switches implemented in hardware, it seems that heterogeneity and containment are the only viable path that can lead to a solution at all.

12.3 Concluding Remarks

Industrialized nation states are currently facing an almost impossible dilemma. On one hand, the critical functions of their societies, such as the water supply, the power supply, transportation, healthcare, and phone and messaging services, are built on top of a huge distributed digital infrastructure. On the other hand, equipment for the same infrastructure is made of components constructed in countries or by companies that are inherently not trusted. In this book, we have demonstrated that verifying the functionality of these components is not feasible given the current state of the art.

The security implications of this are enormous. The critical functions of society mentioned above are so instrumental to our well-being that threats to their integrity also threaten the integrity of entire nations. The procurement of electronic equipment for national infrastructures therefore represents serious exposure to risk and decisions on whom to buy equipment from should be treated accordingly. The problem also has an industrial dimension, in that companies fearing industrial espionage or sabotage should be cautious in choosing from whom to buy electronic components and equipment.

Honest providers of equipment and components see this problem from another angle. Large international companies have been shut out of entire markets because of allegations that their equipment cannot be trusted. For them, the problem is stated differently: How can they prove that the equipment they sell does not have hidden malicious functionality? We have seen throughout the chapters of this book that we are currently far from being able to solve the problem from that angle as well. This observation implies that our problem is not only a question of security but also a question of impediments to free trade.

Although difficult, the question of how to build verifiable trust in electronic equipment remains important and its importance shows every sign of growing. The problem should therefore receive considerably more attention from the research community as well as from decision makers than is currently the case. The question has implications for national security as well as for trade and is therefore simply too important to ignore.