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)