Roma Tre University
Ph.D. in Computer Science and Engineering

Cybersecurity of Industrial Control System. Innovative solutions to enhance the security posture.

Federico Griscioli

Cybersecurity of Industrial Control System.
Innovative solutions to enhance the security posture.

A thesis presented by
Federico Griscioli
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Engineering

Roma Tre University
Dept. of Informatics and Automation

Autumn 2019

Committee:
Prof. Maurizio Pizzonia, Roma Tre University, Italy

Reviewers:
Prof. Tiago José dos Santos Martins da Cruz, University of Coimbra, Portugal
Prof. Erik Poll, Radboud University, The Netherlands

To my Family and my roots that let me fly
To light and fresh air.
To the gift for seeing far away with an open heart.

Contents

Chapter 1

Introduction

1.1 Industrial Control Systems

Operational technology (OT) is a generic term used to identify hardware and software used to control physical equipments and processes. One of the major segment of the OT is the *Industrial control systems* (ICSs). ICS is made for of several parts, like, Supervisory Control and Data Acquisition (SCADA) and Programmable Logic Controllers (PLC) largely used in the industrial sector and critical infrastructure [SPL+11].

Such a systems differ from regular *Information Technology* (IT) systems in several ways, and also for their security objective. IT systems have as their primary objective the confidentiality of the data. In contrast, ICS is mainly focused on the availability of the industrial process. Note that, both IT and ICS have the integrity as a second security goal.

IT and ICS world also different for requirements in term of latency. In the ICS it is common to have application requiring low latency, like the applications in charge of interacting with device at the process level.

The ICSs is generally segmented in different areas, each one with a specific role. According to the *PERA model* [Wil98] there should be five different zones. We consider a simplified architecture of an ICS (see Figure 1.1) composed by the following areas:

(I) *enterprise zone (management/corporate):* the zone where IT infrastructure systems and applications are placed,

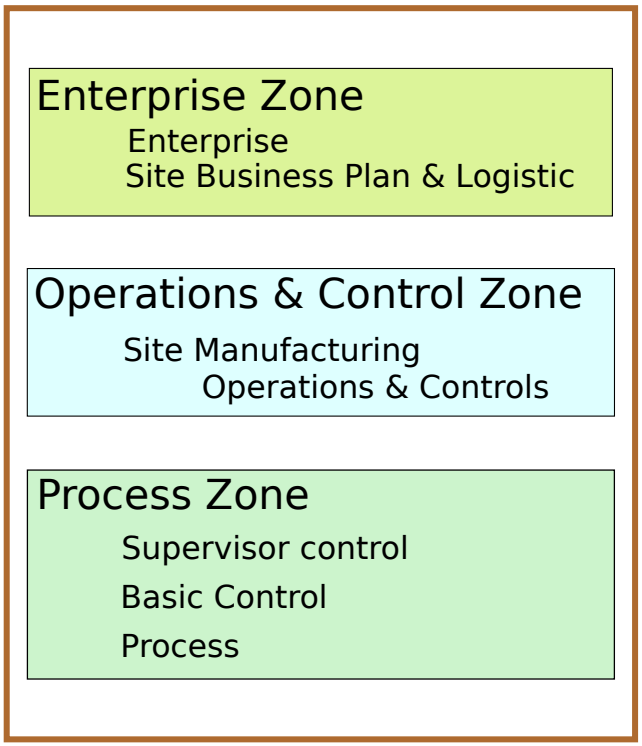

Figure 1.1: Simplified architecture of an Industrial Control System.

(II) *operations and control zone:*, the zone that is in charge of managing and controlling the operations needed to produce the end product, and

(III) *process zone:* the zone where are placed the devices that directly control the manufacturing process, i.e., sensors and actuators.

The architecture of ICSs has been changing in the last few years. Originally ICSs were built without cybersecurity in mind and having as a main concern the reliability. The old ICSs were protected by a physical isolation between the process zone and the rest of the world, called *air gap*. The air gap aims at eliminating network interfaces that can be potentially exploited by attackers to gain unauthorised access to the system. Many ICSs implement the isolation by firewalls instead of physical isolation arguing that a software isolation it is

enough to guarantee a proper protection.

The use of firewalls instead of the air gap is increasingly common. Recently, we are witnesses of a certain convergence between IT and OT driven by the idea that this integration can enable additional features and increase the flexibility and the efficiency of the systems. For instance, a direct connection between the enterprise and the process can enable data analysis and, especially, predictive analytics in order to facilitate real-time decision. This could be potentially a benefit for time-sensitive decisions for which the latency between events at process level and decisions, mostly taken at corporate level, is relevant. For instance, in this way, the performance of the system and the efficiency, in term of usage of the resources, can be enhanced. The integration of the high connectivity and all features typical of the IT world (e.g., mobile applications) with the OT world can potentially boost up the productivity of the system.

1.2 Cyber Security in Industrial Control System

The cyber security scenario is deeply changed in the last few years. Nowadays, most of the attacks are launched by wealthy organizations. It is common to recognise behind skilled attacks criminal organizations or countries (i.e. foreign intelligence) that combat silent wars by means of cyberweapons. Both of them are high motivated and with a lot of capital to invest in this kind of activities. That means they have the capability to tailor attacks for specific targets so that the possibility of failure is drastically reduced.

Critical infrastructures, and especially ICSs, are privileged targets of these attacks. Likely, it is due to the important role covered by these systems in the society. Indeed, an attack against an automation system, like an electrical utility, could be disastrous having an huge impact on common people. This is a possible case in which a cyber-attack can become a cyberweapon. In the last decade, the architecture of the ICSs is changed drastically. Also if the ICSs are still focused on the service, hence on the availability, they are moving fast to become a realisation of the system much more sophisticated. Air gap between the process zone and corporate zone seems to be unrealistic, as information exchanges result to be essential for process and business operations to function effectively. Further, innovative attacks as the *Advanced Persistent Threats* (APTs) [FMC11a] have largely demonstrated the air-gap is not totally effective as a security defence. Indeed, there are several pathways that can be travelled to reach an air gapped system like removable media such as thumb

drives, CDs, DVDs. Indeed, thumb drives can be considered an effective mean of infection [TDF+16].

The convergence between IT and OT exposes the ICSs to a set of vulnerabilities that are typical of the IT world. Leveraging the IT, on one hand enhances the usability of ICSs but, on the other hand, increases the attack surface since the system exposes an additional set of interfaces that can be exploited by an attacker. For instance, we can take into account the use of mobile application to have a view of the process level. For sure, it creates value in term of usability but it also turns out to be a potential direct entry point to infect remotely the sensors in charge of monitoring the manufacturing process.

We argue the integration of IT with OT exposes the latter at a set of additional risks that are challenging to face. Unfortunately, defences that are often effective in the IT are not easy applicable in the ICS due to the strict requirements of availability. A simple deployment of a patch or keeping the system up-to-date is most of the time unfeasible. This becomes more relevant in the ICS where is pretty normal to find legacy devices for which is likely to have exploitable vulnerabilities.

At the moment, one of the most relevant challenge regarding the security of ICSs is the speed at which threats change. It is fundamental to change the approach to the security and the mindset in order to face threats introduced by the new design of such systems and all recent innovations. A recent trend in the design of new ICSs is the so call *security by design* for which systems are designed by the beginning having the security in mind. The main focus is to make the developing system as free of vulnerabilities and robust to attacks as possible. Security by design is a guiding principle that leads to build the security in every part of the system so that the security becomes an implicit result. The long life-cycle of the equipments in a ICS suggests that it is common to have legacy devices in such environments. The downside is that assuming that a new part of the system has been just designed according to security by design approach and deployed, a cooperation of this part with another legacy could introduce weaknesses that could be hard to patch and make the new part somehow vulnerable.

1.3 Advanced Persistent Threats and Lesson Learned

The rapidity of the change of the ICS threats scenario and the new approach to the design of such systems force organizations to modify their approach

to security. While traditional threats continues to be an important concern, attacker are more and more skilled. Nowadays, we talk about *Advanced Persistent Threats* [Col13]. This term has evolved from its birth. Today it is used to stress the capability of this kind of malware to compromise effectively a system remaining hidden for long time.

APTs are so sophisticated to be able to bypass common protections and remain hidden for long time, also years. Once an APT has taken the control of the first machine of the target system, as a first step, it opens a communication channel with a machine placed outside the organization, called *master and control*, and starts gathering information about the internal structure of the system and its possible vulnerabilities. The master and control is in charge of harvesting all information sent by the worm and processing them in order to tailor the attack technique to the specific victim system. In this way, the attack has high probability to perform undetected lateral movements and silently exfiltrate sensitive information. An attack against the manufacturing process often represents the last stage of the APT since, with high probability, it is going to be detected by the the *Intrusion Detection System* (IDS) of the organization.

Stuxnet [FMC11a] is the first APT. Discovered in the 2010, it targeted an Iranian nuclear power plant physically isolated (i.e., without any connection outside the power plant). This worm is the demonstration of how the air-gap approach is an ineffective defence against such innovative attacks. Indeed, the initial infection vector was a thumb drive containing a malware, introduced in the target environment by a contractor or an insider. Once, the first computer was compromised, Stuxnet became to spread over the internal network in search of a specific device with Windows OS (e.g., Simatic Field PGs) designed to program *Programmable Logic Devices* (PLCs). Such devices was compromised exploiting a known Windows vulnerability and was used as a foothold to change the code in the PLC which were in charge of controlling the process. This approach suggests the attacker(s) had performed an information gathering before the attack and, hence, the attack has been created with a specific target in mind. Then, a master and control has been used to update the Stuxnet executable in order to perform lateral movements and reach the ultimate goal that was to sabotage the system. Indeed, the last stage performed by that worm was to alter the speed of power plant centrifuges and shut them down. Since the monitor and control system was already compromised, the disruption of the process was not detected till the effects were evident.

The APTs have pointed out a new innovative pattern of attack that mixes not known vulnerabilities, called *0-days*, to old vulnerabilities to achieve their final objective that can alter the manufacturing process, like Stuxnet, or exfiltrate information, like, for example, Duqu [BPBF11]. APTs are partially automated and partially involve humans intervention realised by means of the establishment of a connection with a master and control. The presence of a skilled attacker guarantees an effective ability to adapt the attack to the specific targeted system. Unexpected changes or defences deployed in real time can be circumvented thanks to human analysis and reasoning. USB thumb drives turned out to be an effective infection vector [TDF+16]. They can be weaponised to compromise also physically isolated systems (i.e., air-gapped systems) and protected by traditional defences.

Most of the time, these advanced threats enter into an organization as something legitimate, e.g., legitimate traffic or file. Once the system has been infected, an APT executable replicate itself in order to lateral-move over the system while remaining hidden as long as possible. So, the traditional defences result to be totally ineffective. A reactive approach to security seems to be no longer adequate. The idea of waiting for a visible sign before acting does not work with APT since, most of the time, when something becomes visible the attack has already reached the final target and every action is by then worthless. According to that, a change of the mindset to security is vital. It is important to start thinking as if the system were under attack every moment. The huge step forward done by APTs with respect to other type of threats is the persistence. Other malwares try to compromise a target using a limited list of techniques. If after a while they do not succeed, they jump to the next target. This is not true for APTs. Instead, these persistent malwares keep trying till they are successful. The organizations should switch from a reactive to a proactive approach and the attack surface has to be reduced to the minimum possible. The deployment of defences focused on blocking infections at the first stage is a must. These defences have to take also into account the use of USB thumb drives to circumvent the first line of protections placed to isolate properly the internal perimeter of the system from the outside. There is not a silver bullet to secure a system, especially when we talk about the malwares that nowadays are targeting ICSs: the *defence in depth* concept has to be applied. This concept is based on the idea that a single technology is not able to safeguard a systems but different levels of protections are needed. Detection techniques that properly integrate heterogeneous data collected from different parts of the system, e.g., from process and corporate, are valuable to increase

the probability to identify APT-like attacks.

1.4 Software Defined Networks and Industrial Control System

Software Defined Networking (SDN) [Fun12] is a relatively new paradigm initially used to describe the OpenFlow project [MAB+08] devised in Stanford University.

The SDN technology has been changing how networks are designed simplifying the network management and enabling innovation. The main change with respect to the traditional approach to the network management is the separation of the *control plane* from the *data plane* and the possibility to set up the behaviour of the network devices (e.g., routers and switches) by software. The control plane represents the "mind" that is in charge of taking decision how to move packets. The data plane represents the "arm" that is in charge of moving the packets according to the instructions given by the control plane. The decision-making process is centralised in a *controller* that, having an overview of the entire network, can control multiple elements of the data plane like, for example, switches and routers. The behaviour of the data plane for a specific *traffic flow* (also simply called *flow*) is characterised by *rules* that define the actions to perform.

Two control models are possible: *reactive* and *proactive*. We have a reactive control model when the switch (or router) consults the controller to know how a new flow has to be handled. The downside of this approach is that for each flow there is an additional latency induced by consulting the controller. On the other hand, the benefit of this approach is that it makes network management particularly flexible since each flow is routed according to the status of the network at that specific instant of time. We have a proactive control model when the controller populates the flow tables of the switches (or routers) in advance with respect to the flows arrival. The benefit of this approach is that it eliminates the latency due to the request sent to the controller to know how to handle flows. The downside is that the routing policies have to be defined ahead in time and, hence, the system could result to be a little bit less flexible with respect to the reactive control model.

The architecture of ICS are growing increasingly complex (e.g., smart grids).

The demand of additional functionalities, like for example the data analytics, requires more connectivity within the system. In a static environment such as the industrial system, where the traffic patterns are pretty limited and the equipments have a long life-cycle, changes in the network architecture can be not so easy to do. The SDN and its flexibility seems to suit well the needs of the ICSs.

The centralised role of the controller enables the implementation of the network security policies and their update in an easy way and almost in real-time. So, using SDN to implement dynamic security policies is possible, too. The possibility to set up data-plane elements (e.g., routers and switches) via software, along with the capability of the controller to have "the big picture", allows the deployment of new functionalities without modifying the legacy part of the system. It is possible to implement security defences as well. SDN could make possible to use different detection and effective incident-response techniques. Once an alarm of a possible attack is raised, the reaction time interval can be incredibly small. The capability to reroute easily and quickly all traffic (i.e., it can be done automatically by a software) makes the system robust to *Denial of Service Attack* (DoS). Furthermore, if a malicious traffic is detected, the attack can be investigated routing that specific flow toward an ICS honeypot. Under specific conditions (e.g., an attack is ongoing), it is also possible to dynamic change the system segmentation scheme in order to isolate a part of the organization that could have been eventually compromised.

In literature there are several works that investigate the adoption of SDN in Industrial Automation. Kalman [Kál16] analyses the different aspects of SDN in a industrial scenario including security. It also shows the possible enhancements to mitigate the challenges related to network segmentation. Khandakar et al. [ABGS18] propose a network SDN-based architecture, called Software Defined Industrial Automation Network, that aim at improving network scalability and efficiency. Piedrahita et al. [PGG+18] present an architecture based on SDN able to automatically respond to an attack against a water treatment process.

1.5 Cloud Computing and Industrial Control System

The *Cloud Computing*, in the rest called also *cloud*, as defined by the NIST, is a model for enabling ubiquitous, convenient, on-demand network access to a

shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [MG+11]. The cloud provides three different services models:

Software as a Service (SaaS). In case of SaaS, an application (software) runs in the cloud infrastructure. The users do not have any visibility to underlying infrastructure that in completely transparent to them. They do not have to control or manage the network, servers, the operating system or the storage.

Platform as a Service (PaaS). In case of PaaS, the capability to deploy software into the infrastructure is provided to the users. PaaS provides platform layer resources like, for example, a software development framework or deployment components. As for the SaaS, the end users do not have to manage the infrastructure but they could have control of the configuration settings for the application-hosting environment.

Infrastructure as a Service (IaaS). In case of IaaS, computing resources in form of hardware, storage, and networking are provided to the clients. A virtualization technology needed to manage these resourced could be provided as well. Also for IaaS the underlying cloud infrastructure is transparent to the users who, instead, have to control and manage the operating systems, storage, and the deployed applications.

According to the scenario mentioned in the Section 1.1, the ICSs have been evolving over the years to facilitate the information exchange with the system [WSJ17]. The automation systems has been embracing the new technologies faster than in the past. This trend seems to suggest that in the next future the cloud could be fully integrated in ICS environments. For instance, the cloud potentially well fits a scenario in which agility and flexibility in production plants are needed.

The sensors and actuators are becoming more intelligent by embedding new functionalities. The cloud can help these functionalities and allows real-time data sharing among plants distributed in various locations as, for instance, in case of oil, gas utilities and smart grids. Just mentioning the cloud to security people working in a ICS environment stresses them out. Most of them believes there is no way to adopt solution based on the cloud computing and guarantee, at the same time, a good level of security. We argue this can be possible by

means of a deep understanding of the cloud and of its potential security weaknesses. Indeed, a sufficient awareness is the only way to integrate the cloud with an ICS architecture without introducing new vulnerabilities exploitable by an attacker. Changes of the existing architecture, that could be needed, have to be designed carefully to avoid the growth of the attack surface.

A *private cloud* is provisioned for exclusive use by a single organization and, in general, can be owned and managed by the organization itself or by an external provider. If a cloud provider is involved, it has to be considered potentially an untrusted third party. In case of *hybrid cloud*, the infrastructure is a composition of two or more distinct cloud infrastructures that remain unique entities. We talk about *public cloud* when the cloud services are offered by the Internet. In this case different organizations and users can use the resources from the same infrastructure at the same time.

In the ICS context there are strict requirements in term of availability and the data exchange, especially at process level, are critical for the correct functioning of the system. According to that, a deployment of a cloud service based on the private model seems to be a better solution in term of security. Another possible approach could be the use of an hybrid model on which the services that are considered critical are provided by a private cloud, while services less critical (or not critical at all) can be provided by a public cloud.

A real case in which the cloud is used within an ICS can be, for example, the adoption of the SaaS for the process historians. This data stored in cloud can be integrated with the real-time process data, picked from different plants remotely connected, to have efficient data analytics process. Due to the diversity of the data (i.e., different manufacturing processes) and the huge amount of data in place (i.e., several remote plants), leveraging the cloud represents a feasible, cheap, and easy way. It is worthy of note that, in this case, the cloud represents the enabler of an additional feature that would have been hard to realise differently. Goldin et al. [GFG+17] presents a possible example of cloud computing infrastructure for big data analytics in the process control industry.

1.6 Thesis Outline

The purpose of this thesis is to provide techniques that, along with traditional defences, can enhance the improvement of the cyber security posture of indus-

trial control systems.

The research undertaken in this work has been motivated mainly by the needs of innovative defences able to mitigate risks concerning the new threat scenario of industrial control systems. Indeed, the common protection adopted by ICSs seems not to be effective against innovative attacks (e.g., APT) any more. Moreover, ICSs are changing rapidly to satisfy requests of higher interconnectivity (e.g., connection between enterprise and process zone) and introduction of additional features that can boost up the governance (e.g., IT-typical components and data analytics process). This revolution exposes these systems to a new infection vectors from which is challenging to protect considering the complexity of deploying new components, especially in the process zone.

We started with an overview of the current cyber security scenario and how it has been changing over the last decade. We introduced also the cloud computing and the software defined networking. These technologies, typically belonging to the IT world, can create value in the ICSs in the next future.

As mentioned in Section 1.3, to deal with APT a change of the mindset to security is needed. A single defence technique does not seem to be enough any more. Te high-profile attacker often penetrate into the system with a traffic that looks like legitimate.

An analysis of techniques used by such innovative attacks targeting ICS suggested the USB thumb drives are effective infection vector that can be used to bypass the first perimeter of defence and jump directly into the critical part of the system (i.e., critical machines) that has to be carefully protected. Leveraging a USB thumb drive allows attackers to compromised also system that are strongly isolated by means of air-gap.

In Chapter 2 we show a method that adopts cryptographic techniques to inhibit critical machines from reading possibly malicious files coming from regular machines (i.e., machines that are potentially a sources of attacks against critical machines) on untrusted USB thumb drives. The proposed approach acts in a preventive way and exposes limited attack surface for any malware, even those based on zero-days.

The more a malware appears as something legitimate and the higher is the probability it passes undetected. The BadUSB attack [NK16] techniques allows an attacker to exploit a legitimate functionality (e.g., keyboard typing) to infect a system. In this situation, traditional defences result to be totally

ineffective. The idea behind BadUSB is to modify the firmware of USB devices in order to impersonate a different USB peripheral like, for instance, a mouse of a keyboard. It allows an attacker to send malicious commands to the host the USB device is plugged into and provides a way to eavesdrop, replay, modify, fabricate, or exfiltrate data.

In the Chapter 3 we address this security issue focusing on mice and keyboards. We propose a new approach that, before allowing the device to be used, forces the user to interact with it physically, to ensure that a real human-interface device is attached. Our implementation is hardware-based and, hence, can be used with any host, comprising embedded devices, and also during boot, i.e., before any operating system is running. Considering that our approach does not require any special feature from USB devices neither, it well suits the ICS environment and can be easily integrated with legacy system as well.

In Chapter 4, we present an integration of the works shows in Chapter 2 and Chapter 3. The result is a solution that addresses the security issues due to threats coming both from malware hidden in files stored in the thumb drives and from BadUSB attacks. Our solution allows a promiscuous use of USB thumbs drives while guaranteeing an high level of defence. The main component of the architecture we propose is an hardware, called *USBCaptchaIn*, intended to be in the middle between a critical machine and all USB devices. We do not require users to change the way they use thumb drives and to avoid human-errors, we do not require users to take any decision. The proposed approach is highly compatible with already deployed products of a ICS environment and proactively blocks malware before they reach their targets.

In critical infrastructures, communication networks are used to exchange vital data among elements of Industrial Control Systems (ICSs). Due to the criticality of such systems and the increase of the cybersecurity risks in these contexts (see Section 1.2), best practices recommend the adoption of Intrusion Detection Systems (IDSes) as monitoring facilities. The choice of the positions of IDSes is crucial to monitor as many streams of data traffic as possible. This is especially true for the traffic patterns of ICS networks, mostly confined in many subnetworks, which are geographically distributed and largely autonomous.

In Chapter 5, we introduce a methodology and a software architecture that allow an ICS operator to use the spare bandwidth that might be available in over-provisioned networks to forward replicas of traffic streams towards a single IDS placed at an arbitrary location. We leverage certain characteristics

of ICS networks, like stability of topology and bandwidth needs predictability, and make use of the Software-Defined Networking (SDN) paradigm. We fulfil strict requirements about packet loss, for both functional and security aspects. Finally, we evaluate our approach on network topologies derived from real networks.

In Chapter 6, we present an overview of a solution devised to improve the cyber security of ICSs adopting an innovative approach. This solution, developed within the context of Preemptive European Project [Par], encompasses several detection and prevention tools. Each of them aims at addressing a specific security aspect and use data collected in different part of the system, i.e., heterogeneous data from host, process, and corporate network. All data are integrated and correlate in order to decrease false positives and increase the chance to detect also APT-like attacks. The alarm are displayed to user by means of a *Human-Machine Interface* (HMI) that allows the human decision-making process.

As mentioned in the Section 1.1, the ICS environments have been evolving over the years picking up (relatively) new technologies that are emerging in other fields, most of the time in the IT world. In the Section 1.5, we point out the importance of the cloud computing also in the context of automation systems and how it represents a paradigm that is a good candidate to be integrated into the next ICS architectures.

Public cloud storage services are widely adopted for their scalability and low cost. However, delegating the management of the storage has serious implications from the security point of view. We focus on integrity verification of query results based on the use of Authenticated Data Structures (ADS). An ADS enables efficient updates of a cryptographic digest, when data changes, and efficient query verification against this digest. Since, the digest can be updated (and usually signed) exclusively with the intervention of a trusted party, the adoption of this approach is source of a serious performance degradation, in particular when the trusted party is far from the server that stores the ADS.

In Chapter 7, we show a protocol for a key-value storage service that provides ADS-enabled integrity-protected queries and updates without impairing scalability, even in the presence of large network latencies between trusted clients and an untrusted server. Our solution complies with the principle of the cloud paradigm in which services should be able to arbitrarily scale with

respect to number of clients, requests rates, and data size keeping response time limited. We formally prove that our approach is able to detect server misbehaviour in a setting whose consistency rules are only slightly weaker than those considered by previous literature. Our solution is valuable in a industrial system, for instance, in case there are many unintelligent devices (e.g., sensors) that store data in a remote private cloud. In this case, the integrity of data store in the cloud is guaranteed while maintaining the possibility to achieve high throughput keeping limited latency. A remote server can use these data as input of a data analytics process.

Chapter 2

Enabling Promiscuous Use of Thumb Drives

In the past decade, a growth of cyber-attacks directed toward Industrial Control Systems (*ICS*) has been observed [ICS11]. Specifically crafted malware can be used by attackers to alter an industrial process or gather industrial secrets, and, in the end, gain some market or political advantage. Cyber-attacks to critical infrastsructures constitutes a serious risk for society [Lew14]. Historically, SCADA systems, PLCs, and other elements of ICSs are built to provide high levels of safety and reliability but are not prepared to contrast software attacks effectively. Further, attacks to ICSs can be quite advanced, exploiting zero-day vulnerabilities and knowledge of regular antiviruses to evade their detection [VGA13]. Due to the inherent criticality of ICSs, best practices [SFS11] suggest to isolate the most critical parts of the system from other IT components, either physically or by means of firewalls. To overcome the limitation of a poorly connected environment, file transfers are usually performed by means of USB thumb drives and other *removable storage devices* (*RSD*s). The use of RSDs turned out to be an important vector of malware spread [Rau11] making isolation efforts to protect ICSs from the rest of the IT systems largely ineffective.

In this Chapter, we propose an architecture that enables the promiscuous use of RSDs in critical infrastructures, while preserving security. In our approach, machines are either critical (SCADA, embedded devices, etc.) or regular (pesonal notebooks, company PC, etc.). We consider regular machines and RSDs as possible sources and vectors of attacks against critical machines.

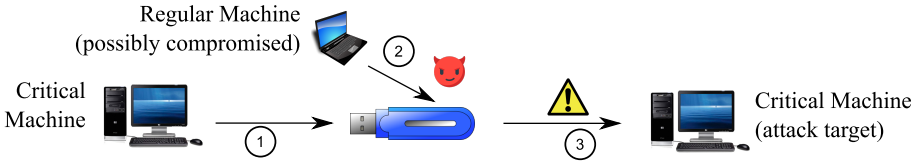

Figure 2.1: A promiscuous use of a removable storage device.

Consider the file copy scenario with promiscuous use of a RSD that is depicted in Fig. 2.1: (1) a critical machine (e.g. a development workstation) writes some data (e.g. a new logic) into the RSD, (2) the RSD is plugged into a, possibly compromised, regular machine, which can infect the logic or add other malicious files in the RSD, and (3) the RSD is plugged into a critical machine (e.g. a SCADA server) that is the destination of the file copy and also the target of the attack. Our goal is to allow this kind of use while preventing (potentially malicious) data or code originated from regular machines to spread into critical ones. We do that by introducing a form of cryptography-based access control solely in critical machines, which are the only trusted part in our architecture. Exceptional data flows from regular machines to critical machines are completely mediated by a special critical machine called *gatekeeper*. Our approach does not rely on malware signatures and is a strong obstacle to the spread of zero-day attacks, even if RSDs are used promiscuously in critical and regular machines. Our architecture requires just small additional software to be included in critical machines, and hence it is easily deployable in real ICS environments. Furthermore, the complete mediation approach enables security policies that may also involve human decisions and complex workflows, and hence can support arbitrarily high security levels with a cost that does not depend on the number of machines to protect.

The rest of this Chapter is organized as follows. In Section 2.1, we review the state of the art and provide some background. Section 2.2 formalizes the requirements we intend to meet. In sections 2.3, we describe the security model and the threat model on which we base our work. Section 2.4 shows the architecture of the proposed solution. Section 2.5 provides an example of use of our architecture. Section 2.6 provides a security analysis. Section 2.7 discusses the applicability of our approach in ICS environments.

2.1 State of the Art and Background

Our problem fits the well known Biba integrity model [Bib77], which describes a set of access control rules that can be used to protect the integrity of certain data. In the Biba model, each element is associated to an integrity level. The rules of this model deny any flow of information from lower levels to higher levels and can be summarized with the statement "no read down, no write up". This model is implemented in recent versions of the Windows operating system [RS09] and, in principle, can be adopted also in ICS environments. However, any form of access control on a filesystem must be performed by a trusted operating system, while we want an USB thumb drive to be usable even on untrusted machines.

There are a number of products on the market that specifically address security for removable storage devices (e.g., see [bit]) and USB thumb drives (e.g., see [top]). These are mostly focused on confidentiality, which, however, is not our primary objective. In these cases, support to integrity is on a file basis or on a block basis, and there is no integrity protection for the whole storage: an attacker can delete selected portions of data and also revert part of them to a previously saved version. Further, all solutions imply some form of authentication, usually password-based, but once the user is authenticated, full access to data is allowed, and a malware can easily infect the stored files.

To mitigate the risk for critical systems to be infected by a malware, an antivirus can be adopted and properly configured to scan the data stored in the USB thumb drive before any access. Most commercial antiviruses perform detection based on a database of known malware signatures. This approach has some drawbacks: it cannot detect zero-days attacks, it needs regular signatures updates to keep its effectiveness, its performances depend on the size of the data to be protected, and it cannot protect from generic tampering, since tampered data in general does not contain any recognizable malware.

Concerning techniques for checking the integrity of data, a large body of work is known in literature. Many rely on robust cryptographic hash functions [RS04]. When the dataset to be protected is large, using hash functions is inefficient. In fact, for each change, even small ones, the hash of the whole dataset have to be re-computed. Also, to check the authenticity of a small part of the dataset, the hash of the whole dataset should be checked. *Authenticated data structures* (ADS) allow a user to efficiently update a cryptographic hash of a large dataset when just a small part of the dataset is changed. For an ADS, the hash of the whole dataset is called *root hash* or *basis*. They also allow a user to efficiently check the integrity of a small subset of data by only

comparing against the root hash an *integrity proof* of size $O(\log n)$ with n the size of the data. Supposing that only the root hash is known to be genuine, it is possible to check the integrity of a small subset of data, efficiently. Widely-known ADSs are *Merkle Hash Trees* (MHT) [Mer88] and authenticated skip lists [GT00]. For these data structures, updates and checks are performed in logarithmic time with respect to the size of the dataset, which is comparable to the efficiency of many indexes for databases and filesystems. For this reason, MHTes or other ADSs have been used in commercial, free, or research products. For example, MHTes were used for securing filesystems (see for example, [LKMS04, SvDJO12]). Authenticated data structures were also adopted to authenticate relational database operations [DGMS03]. The problem of efficiently using ADSs with regular DBMS was studied in [MS05, DBP07, PPP10a].

2.2 Requirements

In this section, we list the requirements that, in our opinion, should lead the design and development of a *solution* for our problem. For each requirement, we provide a brief description. When needed, we also provide some motivations or point out criticalities.

Discernment. The solution should prevent critical machines from reading data whose source is not a critical machine. To realize this, the solution should be able to distinguish data (and meta-data) written by critical machines from those written by other, possibly malicious, machines. This should be possible even if read and write operations are performed on a RSD by distinct machines, and even if those machines do not share any other common knowledge beyond that stored into the RSD.

Full Integrity. The solution should be able to detect a vast range of integrity violations, comprising deletions or restoration of previous versions of files or parts of them. This will enable the detection of attacks vectored by a RSD regardless of the kind of the attack and of the attacker. Restoration of previous backup of an entire volume is not considered an integrity violation.

Timeliness. Violations should be detected before tampered data or code is used or run inside a critical machine. For our purposes, it is essential to adopt a pure proactive approach. Indeed, malicious data or code that is used or run in ICS might immediately (and seriously) impair it.

Interoperability. The solution should be usable in conjunction with the existing systems and software suites (SCADA, HMI, harsh laptops, developement environments, inventory management, etc.), without requiring any invasive change to those products.

Usability. The solution should preserve the convenience and high usability perceived by users when using RSDs. It should enable the *promiscuous* use of RSDs, i.e., the user should be able to use a RSD on both critical and non-critcal systems and even on systems that are not under the control of the organization that runs the ICS.

Efficiency. The solution should not introduce asymptotic complexity overhead on read and write operations, i.e., all operations on the filesystem of an RSD should run in at most $O(\log n)$ time (where n is the amount of data stored) as with regular non-protected storage technologies.

This should be true also for read and write operations on small parts of large files in order to enable the use of RSDs for storing databases or virtual environments

	Disc.	Full Int.	Timel.	Inter.	Usab.	Effic.
Access Control	-	-	+	+	+	+
Antivirus	-	-	=	+	+	-
Encryption	-	=	+	=	-	+(-)
Integrity System	+	+	+	+	+	+

+ yes, = limited, - no

Table 2.1: Comparison of security solutions for RSDs with respect to the requirements described in Section 2.2.

Table 2.1 summarizes the effectiveness of currently available solutions in meeting the requirements described above, while the last line refers to the approach that is the subject of this chapter and is described in Sections 2.4.

Access control relies on meta-data stored on RSDs, hence, it does not guarantee discernment. Indeed, a malicious machine can easily circumvent access control ignoring meta-data during writing operations. Access control does not encompass integrity checks.

Antiviruses cannot discern data written by malicious software from data that comes from critical machines. They aims at recognizing malicious data inspecting the entire device and matching data with malware signatures stored in its database. The complexity of this approach is proportional to the amount of data saved on a RSD. Antiviruses perform real integrity checks only on systems files, which however are not stored on RSDs.

Encryption solutions change the way the user interacts with RSDs (they need passwords or new drivers) and hence have low usability. User authentication can be used to discern different users but it is not suitable for distinguishing if the source of the data is a critical or regular machine. Furthermore, encryption performed on large files suffers of a penalty, in term of efficiency, when carried out at file-level and not at block-level. Most encryption solutions support integrity only at block-level or at file-level.

2.3 Security and Threat Models

We model an ICS as a set of *machines* that exchange data only by means of RSDs. Machines in our model are workstations, notebooks, SCADA systems, etc. Machines are either *critical* or *regular*. Critical machines are intended to be the parts of the ICS in which an "infection" can have a big impact on the physical process. Critical machines require special protection, while regular machines do not. In the following, we call *critical (regular) realm* the set of critical (regular) machines.

In our threat model, the threats originate in the regular realm and spread into the critical realm by means of malicious or accidental writing on RSDs. The objective of the attack is to let malicious code or data to be read by at least one critical machine when the RSD is plugged in it. The objective of our defense is to avoid that this can happen. In this setting, to protect the critical realm, it is enough to forbid information flows from the regular realm to the critical realm. All other information flows can be allowed. This ideal setting conforms to the Biba integrity model [Bib77] with just two integrity

levels, where the rule "no read down, no write up" applies and the critical and regular realms are the higher and lower integrity level respectively. However, in real ICS environments, certain data or software have to be transferred from the regular realm to the critical realm (e.g., software or documentation provided by a vendor). In our security model, this is possible only by means of a special machine called *gatekeeper*. The gatekeeper realizes the "complete mediation" security principle [SS75], which is a well-accepted approach when a security boundary should be traversed, playing the same role of firewalls in networking and of the security reference monitor for operating systems.

In our security model, we consider only information flows that are realized by means of read and write operations on RSDs that are plugged into, or unplugged from, machines. RSDs are not machines. They should be considered as a mere medium for data and they do not have capabilities to carry out any form of access control. We allow the same RSD to be *promiscuously* used in both critical and regular machines. Regular read operations performed on RSDs do not allow the reader to distinguish data written by critical machines from data written by regular machines and malicious regular machines can perform any kind of write operation. This means that any constraint on information flow should be enforced during read operations performed by critical machines.

The main objective of our approach is to equip critical machines with security features so that each critical machine can, without any doubt, assess if the data returned by a read operation on a RSD is exactly what was written by other critical machines.

In our threat model, an attack can involve the injection of malicious code in any file stored in a RSD or the tampering of any data stored in it. In ICSs, several kinds of files can be attacked: control logic, firmware files to be installed on embedded devices, user documents, technical documentation, etc. We do not distinguish among them. For the purpose of our analysis, attaching malicious code to any file or tampering with any kind of data are changes that are equally illegal. We also consider an attack any change performed by regular machines to metadata or directory structures. In our threat model, restoring a previous version of a file is a *freshness attack*, but restoring the whole content of a RSD at a previous version is not considered an attack. We consider this as a restoration of a previous backup (see Requirement Full Integrity in Section 2.2).

Since in our model RSDs are passive, we explicitly do not consider attacks that illegally change the firmware of the RSDs like, for example, BadUSB [NL14].

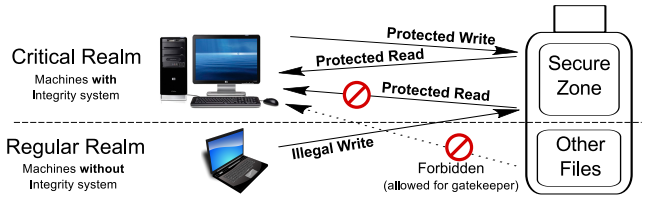

Figure 2.2: Relationships between (regular or critical) machines and removable storage devices in our approach. Critical machines can read only from secure zones, and protected read operations fail on data written by illegal write operations.

2.4 Architecture

In this section, we describe the architecture of our solution that we call *integrity system*. In our architecture, critical machines are equipped with the integrity system that forbids read access when data-tampering is detected.

The main actors in our architecture are machines and RSDs. Figure 2.2 summarizes their relationships and the operations that machines can perform on RSDs. For the sake of simplicity, in the following, we refer to RSDs as having only one volume. In our approach, each RSD contains one or more special directories, which we call *secure zones*. Critical or regular machines can read or write a secure zone. For critical machines, these operations are performed under the protection of the integrity system while for regular machines they are not. For critical machines, it is *strictly forbidden* to perform read operations on parts of an RSD that do not belong to a secure zone.

The integrity system, installed in critical machines, redefines the semantic of usual read and write operations at the system call level, hence, applications are automatically and transparently protected at each read or write operation issued by standard means. This is also true for any piece of ICS-specific software like, for example, a SCADA suite.

In the following, we describe each kind of operation involved by our approach.

Protected-Read. Protected-reads are the read operations performed by critical machines on secure zones. The integrity system changes the semantic in the following way: each protected-read checks the integrity of the read data, if data is recognized as *authentic* (we will also say *genuine*) the data is reported to the application as in the regular read semantic, if data is

recognized as *not authentic*, an error is reported. Special care is taken in order to link the read data with their check to avoid time-of-check to time-of-use attacks.

Protected-Write. Protected-writes are the write operations performed by critical machines on secure zones. The integrity system changes the semantic in the following way: each protected-write operation writes the data on the secure zone and stores additional data in the RSD to enable integrity checks during protected-read operations. Since write operations can involve read operations (e.g., for metadata), if integrity problems are detected an error is reported.

Illegal-Write. An illegal-write is a write operation performed on a secure zone by a regular machine or by any other mean. A regular machine is not equipped with the integrity system, hence, it cannot update the additional information that allows subsequent protected-read operations to recognize the data as authentic: the data changed by an illegal-write are always recognized by the following protected-reads as not authentic.

Plain-Read. Plain-reads are normal read operations performed by regular machines when reading any part of an RSD.

The above rules forbid any data flow from the regular realm to the critical realm. The special gatekeeper machine exceptionally allows ICS operators to perform transfers that are normally forbidden. To do that, the gatekeeper behaves partially as a critical machine and partially as a regular one: (i) it performs plain-read operations from outside the secure zone of an RSD, and then (ii) it performs protected-write operations on the secure zone of the RSD of the data just read. In our approach, the gatekeeper is the single point where data flow policies must be configured and enforced. While the actual policies may depend on the specific industrial context, they should cover (a) *authentication* of the operator that requires the transfer, (b) *authorization* of the transfer according to the policy of the organization, and (c) *logging* of the details of the operations performed to allow subsequent *auditing*. Our gatekeeping approach enables to *scale-up* the security level that on organization can achieve enabling the implementation of arbitrarily deep analysis of the data to be transferred. This analysis can possibly involve automatic antivirus-like checks for known malwares, human-based analysis by specialized security personnel, and human-based authorization from management to check the motivation of the transfer. Since the gatekeeper is a single enforcement point, the cost of the

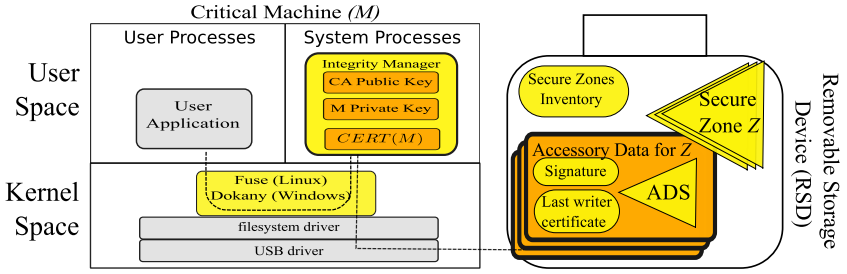

Figure 2.3: The elements of the integrity system and their positions within a critical machine and a removable storage device (RSD).

security realized by the gatekeeper depends only on the strength of the policy and not on the number of critical machines to protect.

The elements of the realization of the integrity system are shown in Figure 2.3. In our solution, system calls related to read and write operations on RSDs are intercepted and their semantic is changed in the way we described above. A possible way to do this is to implement the integrity system functionalities as a patch to the operating system kernel. Even if this is surely the most efficient way to realize our ideas, it requires considerable effort to handle all the technicalities that a kernel space development encompasses. For our prototype, we opt for a simpler approach. We leverage available open source software that allows a developer to realize a filesystem solely developing code that runs in user space. Dokany [Dok], for the Windows operating system, and FUSE [Fus], for the Linux operating system, are kernel drivers that enable this. The main module of the integrity system is the *integrity manager* that runs in user space as a system process. User processes that perform system calls that encompass read and write operations are intercepted by the Dokany (or FUSE) driver, which, in turn, triggers proper callbacks of the integrity manager. To realize the protected-read and protected-write operations, the integrity manager needs to access the RSD. This is accomplished by using the regular read and write primitives provided by the kernel.

When the integrity system is running and a RSD is plugged in, the RSD is automatically mounted (as usually happens in current operating systems) but the operating system is configured so that only the integrity manager can access it. The integrity manager, by means of Dokany or FUSE, shows to user processes a distinct virtual volume that has the exact content of the real RSD. All read and write operations performed by user processes on the virtual

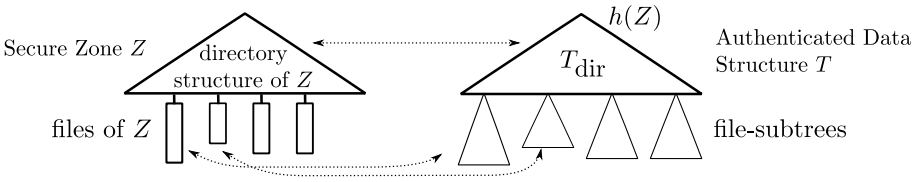

Figure 2.4: Correspondence among parts of the secure zone Z and the authenticated data structure T.

volume are replicated by the integrity manager on the real RSD, with the semantic change explained above, unless they infringe the rules of our security model.

Each critical machine M stores its *private key*, a corresponding *certificate*, denoted by CERT(M) and signed by a unique *Certification Authority (CA)*. Private key and certificate are generated during the installation of the integrity manager on M. Machine M also stores the public key of the CA. Given a certain state of a secure zone Z, a hash of its current content is denoted by $h(Z)$ and its signature is denoted by $\text{sign}_M(h(Z))$. For each secure zone Z, the RSD stores accessory data comprising a *signature* and a *last writer certificate*. When the content of Z is updated by M, M computes $\text{sign}_M(h(Z))$ and stores it in the RSD as the signature of Z along with CERT(M) as last writer certificate of Z. Another critical machine M' can check the integrity of Z relying only on the content of the RSD and on its locally stored public key of the CA. A detailed security analysis is provided in Section 2.6.

For performance reasons, each RSD contains an inventory of the secure zones that are available in that RSD. In this way, the integrity manager can efficiently find secure zones when a RSD is mounted. A critical aspect for the efficiency of the whole integrity system is the time complexity of protected read and protected write operations. We adopt a specifically tailored ADS, stored on the same RSD for each secure zone, to speed up both the computation of $h(Z)$ after each protected write and the check of the integrity of small parts of the content of Z for each protected read. Fig. 2.4 shows the structure of the ADS for a generic secure zone Z. The ADS is denoted T and it supports integrity checks for the directory structure of Z, metadata, and contents of all files. As for MHT (see Section 2.1), T is a tree whose nodes store hashes of the composition of the hashes of their children, and whose leaves store the hash of the corresponding data. The value $h(Z)$ introduced above is the root hash of T. Each file in Z corresponds to a subtree in T, we call it a *file-subtree*. The directory tree of Z corresponds to a part of T, where all file-subtrees were

pruned, we call it T_{dir}. T is represented in a hidden directory in the RSD. T_{dir} is represented as a directory tree that mimics the directory structure of Z. File-subtrees are a variation of MHTs inspired by *history trees* [CW09] and adapted to support generic file content. Each file-subtree is stored in a single file, using only append and overwrite operations, which are the only ones that are supported by operating systems on files. The remaining technical aspects of the representations of T_{dir} and file-subtrees are trivial and are omitted.

We developed a first prototypical implementation of the integrity manager under Linux in Java. We carefully designed the management of the ADS using a concurrent approach based on the actor model and realized it using the Akka [Typ] framework. Our intent was to limit as much as possible the impact of the ADS on the performances experienced by the user. Our implementation keeps recently used parts of the ADS cached in memory and allows several protected-read/protected-write operations to be executed in parallel. The integrity manager is interfaced with Fuse by means of the fuse-jna [Per] library. One can argue that introducing such a complex architecture for managing a filesystem can badly impact performances. Our preliminary experiments show that for thumb drives the additional overhead is negligible with respect to timings involved in normal user interactions, like working on documents and opening multimedia files.

Our integrity system complies to all requirements introduced in Section 2.2 (see also see Table 2.1). **Discernment**: if the data in the secure zone is recognized to be genuine, the integrity system is sure that it comes from a critical machine (see Section 2.6). **Full Integrity**: the signature is the signed hash of the whole content of the secure zone and all integrity checks are performed against it. **Timeliness**: the checks of the integrity system are performed as part of each protected read operation. If any tampered data is detected, the protected-read operation returns an error. **Interoperability**: any RSD can be used to store a secure zone. Also, no changes to ICS-specific software is required to adopt the integrity system. **Usability**: the user can use his/her RSD in any machine without providing any password, and the integrity system is required only for critical machines. **Efficiency**: the adoption of ADSs allows any operation on the filesystem to be performed without introducing any significant asymptotic overhead with respect to common filesystem implementations.

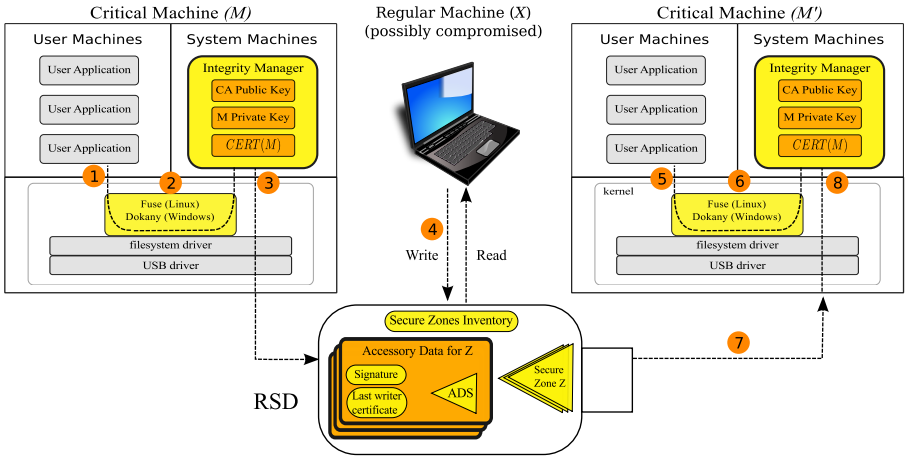

Figure 2.5: Example of use of a RSD for a data transfer between two critical machines M and M' equipped with the integrity system. The RSD is also promiscuously used in a, possibly compromised, regular machine X.

2.5 Example of Use

In this section, we show an example of use of a RSD to transfer data between two critical machines M and M' equipped with our integrity system. We also provide a detailed description of the operations performed by the integrity manager during protected-read and protected-write operations. In our example, a file is copied from M to M', but the RSD is plugged into a possibly compromised regular machine X before being plugged into M'. In the following, we show the most important steps performed by the elements of our architecture (see Figure 2.5).

1. A user adopts a common file manager application to copy a file F from the local hard disk of critical machine M to a secure zone Z of the RSD. This operation encompasses at least one *write* system call.

2. The *write* system call is intercepted by the FUSE/Dokany driver, which performs a corresponding callback for the integrity manager.

3. The integrity manager performs the following updates on the RSD (*protected-write*):

a) F is added to Z,

b) the ADS for Z is updated to account for the presence of F and for its content, the root hash $h(Z)$ of the ADS is also updated,

c) the signature is updated to $\text{sign}_M(h(Z))$,

d) the last writer certificate is updated to $\text{cert}(M)$

4. The user plugs the RSD into a, possibly compromised, regular machine X. If X is compromised, it may tamper with file F in Z. Machine X can only perform illegal-write operations on Z and cannot update the signature to match the different content of Z.

5. The user plugs the RSD into critical machine M' and use the file manager to copy F form Z into the local hard disk of M'. This operation encompasses at least one *read* system call.

6. The *read* system call is intercepted by the FUSE/Dokany driver, which performs a corresponding callback for the integrity manager.

7. The integrity manager performs the following operations (*protected-read*):

   a) it fetches the data required to fulfill the read operation,

   b) it feteches the corresponding ADS proof (see Sections 2.1) comprising its root hash $h(Z)$,

   c) it checks the consistency of the proof, both internally and with the read data,

   d) it checks the root hash $h(Z)$ of the proof against the signature for Z stored in the RSD,

   e) it checks the signature against the the public key that is read from the last writer certificate. The authenticity of that certificate is verified using the CA public key locally stored in M'.

8. If all checks are successful, the requested data are returned to the file manager application. If any of the checks fail, an error is returned.

2.6 Security Analysis

As detailed in Sections 2.3 and 2.4, the objective of the integrity system is to protect critical machines by malware or other forms of attack coming from

regular machines. We do that by enforcing that critical machines can only read (from RSDs) only genuine data coming from other critical machines and by detecting violation of this rule before tampered data can reach user processes.

Our approach encompasses a certain number of assumptions: (I) critical machines are not compromised when the integrity system is installed, (II) gatekeeper effectively checks (and possibly blocks) data flowing from the regular realm toward the critical realm, (III) critical machines cannot communicate with regular machines with means other then RSDs and console operators are trusted, (IV) attackers cannot know private keys and cannot force the CA to sign a new certificate, (V) a limited number of software modules have no security flaws, namely, the integrity manager, the FUSE/Dokany driver, the filesystem driver, and the USB driver (see Figure 2.3), (VI) the adopted cryptographic primitives have no security flaws, and (VII) RSDs are passive (see Section 2.3). By Assumptions I and III, the only vector of attack are RSDs plugged into critical machines. By Assumption VII, any attack conveyed by RSD should leverage some form of data or code stored in it (see also Section 2.3). By the rules stated in Section 2.3, critical machines can read only data stored in secure zones. By Assumption V, read operations are supposed to involve only code that have no security flaws, hence, malformed filesystem data structures cannot be used to attack a critical machine. By Assumption II, no malicious file from the regular realm is admitted in the critical realm. Hence, the only remaining possibility for an attack is trying to make user processes to read tampered data or meta-data from a secure zone stored in a well-formed filesystem.

Since RSDs are completely untrusted, the attacker, for example a compromised regular machine, can freely tamper with any data stored in the RSD (see Section 2.3). This includes data stored in the secure zone, and all accessory data stored in the RSD by the integrity system, namely, the signature, the certificate of the last writer, and the secure zones inventory. Let us consider a tampering of the secure zone. Since the result of any read operation is checked against the signed hash (by means of the ADS), the tampering of secure zone data is easily detected. The attacker can try to avoid detection by tampering also the ADS. An attack to the ADS, that does not change the root hash, requires to find a collision for the hash function on which the ADS is based, which is against Assumption VI. On the other hand, to change the root hash the attacker should be able to violate the signature. However, cryptographic attacks are ruled out by Assumption VI and, by Assumption IV, we assume that private keys and the CA are adequately protected.

Tampering only with the signature, the last writer certificate, the ADS, or

the inventory ends up in a false positive. In fact, in those cases, while the secure zone may be genuine, the integrity system has no way to prove it, hence, it behaves as if the secure zone was corrupted denying any access to it. However, ensuring data availability is not within the objective of the integrity system.

Accidentally, we point out that an attacker that can intercept, and possibly change, the communication between the host and the RSD does not gain any particular advantage. Noteworthy, in this context, Assumption VII can be relaxed. In fact, if a malicious firmware is used to realize a man-in-the-middle between storage and host, with the intent to show tampered data to the host, our approach is still effective in preventing the attack. Essentially, Assumption VII is only needed to rule out from our analysis attacks that comes from a RSD and are not related to data storage, like those that end up in keystrokes injection [NL14].

If a protected-write operation is partially executed, for example because the RSD is unplugged abruptly, either part of the data in the secure zone or part of the accessory data (e.g. the signed hash) is not written. In this case, a protected-read, detects a tampering.

In case of disclosure of a private key, critical machines cannot be considered secure anymore. To ease the recovery from such abnormal situation, our architecture can easily be extended to handle certificate revocation lists that, however, may require manual distribution on the critical machines since, in our approach, they may lack connectivity (see also Section 2.7).

2.7 Applicability Considerations

The advantage of adopting the solution described in this chapter in an ICS context is twofold: (i) regular USB thumb drives can be used for exchanging files among critical machines in a safe manner and (ii) the organization can enforce any security policy, possibly integrated with human-based processes, to restrict the data that are transferred into the critical realm.

In the following, we discuss the hypothesis of our model, as well as the usability and deployment impact of the integrity system.

Model vs. Reality

In describing our approach, we stated several hypothesis, we now review the most important ones from the point of view of the applicability.

- We have supposed that there is no connection between critical machines and regular machines. Isolation between critical machines and the rest of the IT machines is the best practice for ICS security [SFS11], even if this is often obtained by firewalling. The model introduced in Section 2.3 also assumes that all ICS components interact solely by means of RSDs. However, this assumption can be relaxed. Actually, interconnection among critical machines and interconnection among regular machines do not compromise the effectiveness of our approach. Furthermore, interconnecting critical machines may ease the distribution of certificate revocation lists as proposed in Section 2.6

- Concerning the assumptions of trusted operators, non-compromised fresh-installed critical machines, private keys, and protected CA, we expect them to be enforceable by the application of appropriate policies. These policies are likely to be already adopted in a typical environment that can take advantage of our integrity system.

- We assumed the absence of security flaws in specific drivers or parts of the kernel and in the integrity manager (see Assumption V). This attack surface is quite small with respect to other security measures on which ICS usually relies. For example, for the access control enforced by the operating system to be effective, the whole kernel should be flawless. Also, software certification procedures restricted to the parts identified by Assumption V can be applied.

- We have assumed that RSDs are passive devices in the sense they do not have computational power. In our opinion, this is the most critical assumption. Indeed, RSDs are not passive and they run a firmware that can host a malware. If that malware shows malicious data to a critical machine, our integrity system can still detect the attack. However, the integrity system cannot detect attacks in which the malicious firmware impersonates a different kind of device (e.g., a keyboard), like in the BadUSB attack [NL14]. Other research works deal with defending from these kinds of attacks and can in principle be adopted along with our approach (see for example [TBB15a]).

Usability

From the point of view of the user, our approach is highly usable. Contrary to current suggested best practices [SFS11], with our approach, users are allowed

to use the same RSD with any machine without relaying on passwords and without the need to install specific software. In fact, this is not needed for regular machines, while installation of the integrity manager on critical machine is expected to be performed by the organization in a strictly controlled manner. Furthermore, data coming from critical machines can be read anywhere.

The user may notice that additional data are stored in the RSD for ADSs, signatures, last writer certificates, and the inventory. We think that this has a small impact on the usability of the system.

Obviously, a user should be aware that any write operation in the secure zone, performed by a regular machine is actually detected as a tampering when read by critical machines. A possible usability problem is due to users that tamper with a secure zone by mistake. Users should be trained to create a new empty secure zone by proper utilities to easily recover from this situation.

Concerning the need to transfer data or code from a regular machine to a critical machine, the gatekeeper approach (see Section 2.3 and 2.4) addresses the problem. It also enables customization of the security checks so that a good trade-off between usability and security can be achieved for each specific applicative context.

Since the integrity system relies on efficient ADSs that have sublinear time execution for all kinds of read and write operations, no relevant performance penalties should be observed by the user. This was also recognized in preliminary experiments (see Section 2.4).

Deployment Impact

Our approach allows on organization to use regular RSDs, it does not need special software on regular systems, and it is easy to deploy on critical machines. In fact, it does not need any change to existing operating systems, comprising those commonly adopted in ICS, like MS Windows. Also, no networking change is needed since our integrity system does not require any communication channel among machines, beyond that provided by RSDs.

The integrity system is required only inside critical machines where the protection is crucial. Since critical machines are supposed to be in a limited number, slightly higher management standards are supposed to be affordable.

The key management is quite simple. The deployment of the integrity system in an ICS environment requires only the presence of an off-line CA, well protected and managed, (which may already be present in the organization for other purposes) and a proper installation procedure that involves the cre-

ation of a certificate for each critical machine. Aspects related with certificate revocations have been dealt with in Section 2.6.

To be effective, our approach requires the gatekeeper and the certification authority to be properly secured by hardening, control of physical access, etc. The gatekeeper provides the organization with great flexibility about the security policies, but this means that they should be carefully designed and possibly integrated with business or decision processes. For strict security policies, traversing the gatekeeper may be costly, hence, it is advisable to deploy a critical machine realizing a repository of commonly used files ready to be used in the critical realm.

Special care should be taken when dealing with critical machines realized using a virtualization technology. Actually, in that context, new attacks that involve the *hypervisor* are possible. In particular, bugs in the hypervisor can enable unexpected communication among machines in the same host. Also, the host itself can be target of an attack. Consequently, a good security practice is to consider the host and the guests either all critical or all regular. Further, guest access to RSDs is mediated by the hypervisor, so the hypothesis that this mediation is bug-free must also be taken into account to preserve the validity of the analysis developed in Section 2.6.

Chapter 3

BadUSB Attacks: Hardware-based Protection

Traditionally, USB security mostly deals with thumb drives and their role as infection vector or as vehicle for confidential information leakage. Recently, the presentation of the BadUSB [NK16] class of attacks disclosed new threats that involve USB. These attacks are based on a modification of the device firmware, that forces the infected device, typically a thumb drive, to behave as a different kind of device, for example as a keyboard. A malicious "virtual" keyboard can inject commands that end up in a malware infecting the host. These malicious commands could download a malware from the Internet, but can also create it on-the-fly, for example, by "typing" the content of a malicious script and running it. In this context, regular antiviruses are largely ineffective, since the malicious USB device is exploiting basic capabilities (e.g., typing operating system commands) that the user is normally allowed to use.

The primary countermeasure proposed against BadUSB was to *protect* USB devices by a firmware authentication feature that limits the firmwares that can be uploaded into a device to those signed by the device vendor (see for example [IRO16]). However, this approach protects devices but not hosts, which are secured only if they are forcibly limited to interact with just protected devices. This strongly limits the usability of USB devices and it is insecure, if the limitation is delegated to error-prone user behaviour.

GoodUSB [TBB15b] is a software solution that aims at protecting the host against BadUSB attacks. When a new USB device is attached, a message is shown to the user, which must declare his/her expectation about the function-

alities of the device.

In this chapter we present USBCheckIn, an hardware solution that is able to protect any kind of USB host against attacks from devices that claim to be *human interface devices* but are not. The basic idea is that the authenticity of a real human interface device can be easily checked by asking the user to use it. To *authorize* a human intreface device to connect to the host, USBCheckIn instructs the user to perform certain actions on the human interface device by showing messages on its own display. This makes it completely independent from host capabilities and, hence, compatible with any kind of host. While devising USBCheckIn, we focused on verification of keyboards and mice, and we designed the human-machine interaction for high usability (e.g., just 3 gestures to verify a mouse), while preserving security and keeping the probability of a successful brute-force attack negligible.

USBCheckIn has several advantages with respect to other state-of-the-art proposals. (1) It requires a human to provide a proof that the device is, indeed, a human interface device. This ensures that even naive or malicious users cannot behave in a way that the attack is successful. (2) It is compatible with any kind of USB hosts, USB devices, operating systems, BIOS, and boot loaders. (3) It does not rely on signatures, heuristics, or parameters that are provided by the device.

Our contributions are the following: (1) we describe the architecture of USBCheckIn (Section 3.2), (2) we show the interaction between USBCheckIn, the user, and the device that leads USBCheckIn to determine if the device is indeed a real human interface device (Section 3.3), (3) we provide a security analysis of our approach (Section 3.4).

A *companion video*, showing a prototypical realization of USBCheckIn, can be downloaded from the Internet [vid].

3.1 Background

In this section we briefly recall some of the basic concepts about the USB protocol.

An actor in the USB communication can have one of two roles: *device* or *host*. When a device is plugged into the USB port of an host system, the host performs an *enumeration*. During this stage, the host discovers the *class* of the device and its *type*. This chapter mainly deals with devices belonging to the *Human Interface Device* (HID) class as defined by the USB standard, in

particular, of *type* mouse and keyboard[1]. Other device classes refer, for example, to mass storage, printing, and audio/video streaming. HIDs are used to submit human input to the system and hence are fundamental in the BadUSB attack. In this chapter, we focus on the authorization of mice and keyboards while inhibiting the use of other HIDs, like joysticks. After enumeration has been performed by the host, normally the device can be used until it is disconnected. The device can simulate a physical disconnection from the host by sending a *detach* signal. This allows the device to trigger a new enumeration and present a class and a type different from the previous enumeration. Indeed, a physical USB device can be composite and can show to the host several *interfaces*, each of them with a distinct $\langle class, type \rangle$ pair. For the purpose of this chapter, they can be considered distinct devices, since, the communication channels from the host toward each interface of the device are completely independent, hence, any filtering can be selectively performed. A device never speaks autonomously (beside the detach signal case). The protocol states that the host periodically polls all the devices (interfaces), which must start replying within a very short time, compared to frame length. Timing is so strict that a malicious reply from a non-polled device gets mixed with the genuine answer and discarded. The host does not poll the next device until either receive a reply or a timeout expires. Reply packets have no source field, but since they can only follow a poll from the host, their source is easily deduced.

3.2 Architecture

In this section we present the architecture of USBCheckIn, as shown in Figure 3.1. USBCheckIn is equipped with two USB ports: one *upstream port* and one *downstream port* (*up/down ports*). The up port of USBCheckIn is connected to one of the USB ports of the host system we intend to protect. For full protection, we assume that all other USB ports of the host are not used (e.g., they might be physically disabled) and all devices the user intends to attach to the host are actually attached, possibly through a hub, to a down port. For the sake of simplicity, we present the architecture of USBCheckIn with only one down port. The handling of additional down ports is a straightforward extension of the approach proposed in this chapter. Near the up port, there is an orange LED that indicates that the host is powering USBCheckIn and it is correctly working (e.g., it is not suspended). Near the down port, there is

[1]Actually, with the word "type" we refer to the value of the bInterfaceProtocol field in the USB standard.

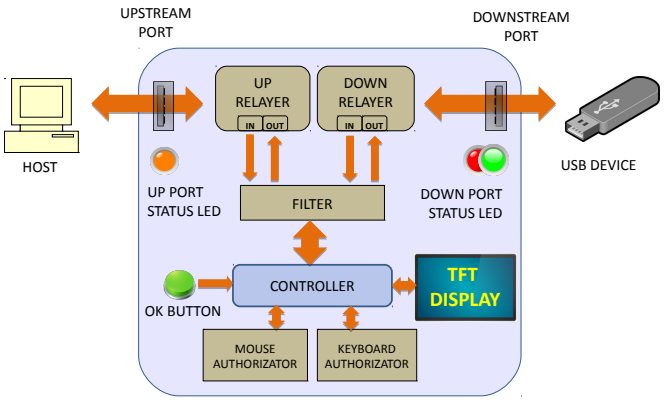

Figure 3.1: The architecture of USBCheckIn.

a status red/green LED. The LED blinks green when one of the attached devices, possibly through a hub, is undergoing the authorization procedure, turns fixed green when all devices are successfully authorized and can interact with the host, and turns blinking red if the authorization procedure of one of the attached devices failed too many times and the device must be disconnected. Messages required for user interaction are presented to the user on a 2.1" TFT display which is embedded in USBCheckIn. An "ok" button is also present, since a direct interaction is needed for certain special cases (see Section 3.3).

USBCheckIn is based on the Beaglebone Black board [bea16] on which a Linux kernel is running. In USBCheckIn, a software, running in user space, intercepts, inspects, redirects, and injects USB traffic between up and down ports, depending on the state of the authorization procedure, as described in Section 3.3. The USBCheckIn software is made up of several components.

Relayers. On one side, up/down relayers are responsible of sending and receiving packets to and from the USB ports. On the other side, packets are forwarded to and received from the filter.

Filter. The filter is in charge of performing standard USB initialization steps (like enumeration, see Section 3.1) and passes USB packets to and from the relayers and the controller depending on the authorization status of the device as explained in Section 3.3.

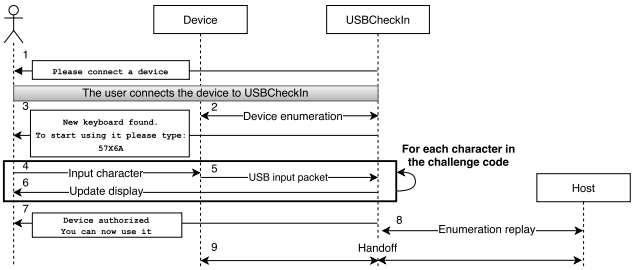

Figure 3.2: Interactions and messages among user, device, USBCheckIn, and host. The diagram shows messages for the keyboard authorization procedure.

Controller. The controller orchestrates the filter, the display, the status leds and the authorizator to realize the authorization procedure and human-machine interaction described in Section 3.3.

Authorizators. Authorizators manage the authorization process of HIDs connected a down port, by generating challenge codes, keeping track of the number of attempts, and checking the correctness of the submitted code (see Section 3.3). Each authorizator corresponds to a specific type of HID. The correct authorizator is chosen by the controller according to the type declared by the device.

The software running on USBCheckIn is a customised version of USBProxy [Spi16]. The filter and the authorizators are USBProxy plugins. The communication between the device connected to a down port and the down relayer is realized by means of libUSB [lib16], a library that supports the interaction with generic USB devices, while the communication between the host and the up relayer is realized by means of gadgetFS, a linux kernel module that allows the Beaglebone to act as a client towards the host.

3.3 Interactions

In this section, we describe the interactions between four actors: a human, USBCheckIn, the USB device just plugged, and the host. In the absence of USBCheckIn, when a device is directly attached to a host, the host enumerates the device (see Section 3.1). When USBCheckIn is attached to the host, it prompts the user to connect a device by showing a proper message on the display (see Figure 3.2, step 1, and the companion video [vid]). When the user

attaches the device to USBCheckIn, the latter performs the enumeration (step 2). In this phase, USBCheckIn recognizes the capabilities of the device and records all provided data.

If the device is an HID, USBCheckIn starts the authorization process asking to the user to input, **by means of the HID itself**, a randomly-generated challenge code (Step 3).

In case of a keyboard the challenge code is a 5-characters alphanumeric string, which is communicated to the user by showing on the display a message like the following

```
New keyboard found
To start using it please type:
57X6A
```

The user types/clicks the *elements* of the challenge code (Step 4). For each of them, the element (for a keyboard is a key code, for a mouse is a click event) is sent by the device to USBCheckIn according to HID normal behaviour and the regular USB protocol (Step 5). The authorizator checks that each received element matches with the corresponding element of the challenge code. If it matches, USBCheckIn provides a visual feedback to the user (Step 6). For example, for a keyboard, by coloring green the corresponding element on the display. Then, USBCheckIn loops expecting further elements and checking them until the challenge code is finished.

If the user correctly inputs all challenge elements, USBCheckIn notifies her/him that the new device has been authorized and that it can be used (step 7). At the same time, USBCheckIn impersonates the device toward the host by performing a "replay" of the enumeration that was previously recorded (step 8). Then USBCheckIn performs an *handoff* by configuring the filter to short-circuit the up relayer and the down relayer (step 9). USBCheckIn keeps monitoring the exchanged packets between the host and the device in order to recognize hardware or logic *detach/re-attach* operations and then trigger a new authorization procedure, when needed.

If any of the received elements does not match the corresponding element of the challenge code, the authorization is started over again showing a message that, for a keyboard, are like the following.

```
Wrong code - try again.
To start using the keyboard pls type:
7E5N3
```

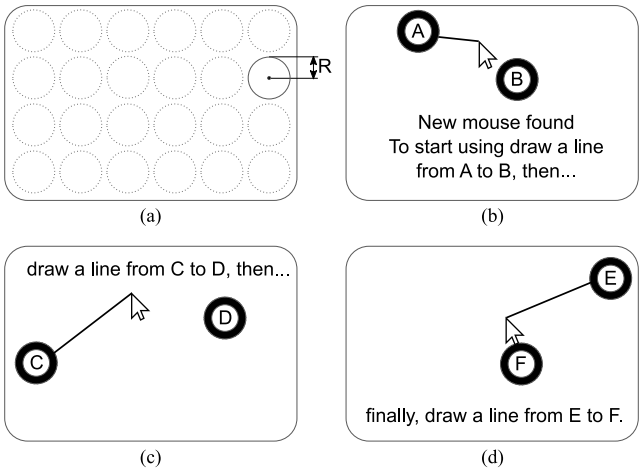

Figure 3.3: The mouse authorization method. (a) Allowed target positions and radius R. (b,c,d) The three steps of the authorization: in (a) just two rows are available for second target positioning, in (b) and (c) three rows are available.

The authorization process can be tried (steps 3-6) for a maximum of three times. After three wrong attempts, USBCheckIn definitely blocks the device, ignoring any more input from it, and shows a message on the display, as follows:

```
    *** Authorization failed ***
 Device claims to be a [device type].
 Is it true? Is the device malicious?
To check it again, press the ok button.
```

In this message, "[device type]" can be "keyboard" or "mouse". To try again, the user has to push the "ok" button on the USBCheckIn hardware, then the authorization procedure starts over from step 2. This physical interaction makes a completely automatic brute-force attack impossible. This message is displayed also when a malicious device tries a guess attack.

If the device is a mouse, USBCheckIn asks the user to move a pointer on the display to draw a line between two randomly-placed targets, for 3 times. The procedure was selected to obtain a good compromise between security and usability. In this case, each element of the challenge is a pair of points of the display, and the input of the user matches the element of the challenge if and

only if the distance between the points of the challenge and the clicked ones is below a certain *radius*. The radius is chosen so that 24 non-overlapping targets can fit in the display (see Figure 3.3). Instruction messages for the user are adaptively placed on the display on an unused zone to allow higher freedom in target positioning. See section 3.4 for a security analysis. If the class of the device is not HID, the display shows to the user the capabilities declared by the device with a message like the following:

```
The connected device claims to be:
   a webcam
To authorize it, hold down OK for 2 secs.
```

Once the message is displayed, the user has to authorize the device explicitly, by means of long-pressing the button located on USBCheckIn. If the user allows the device, USBCheckIn replays the enumeration toward the host and the filter is programmed to short-circuit the relayers for that USB traffic. After that, the device can directly communicate with the host. To deny the device, the user just disconnects it.

3.4 Security analysis

In this section, we discuss the effectiveness of USBCheckIn in protecting any host from BadUSB attacks that mimic HID behaviour, for example performing keystroke injection.

Our approach is not vulnerable to human mistakes. In fact, we do not ask to the user to take any decision, like accepting device features or choosing in a list of allowed device classes (but for the case of non-HID devices, which are not the primary target of this work). Actually, not even a malicious user can force USBCheckIn to authorize a device that claims to be an HID but offers no means to the user to provide the requested challenge code.

Our approach is extremely well protected against guessing/brute-force attacks. For keyboard authorization, the challenge code consists of 5 characters each ranging within 26 letters plus 10 digits. The probability for a malicious device to correctly guess a random challenge in three attempts is 3 over 36^5 (i.e., 1 over about 20 millions). For mouse authorization, each challenge code consists of 3 pairs of points. The first point of each pair ranges within 24 possible positions. The second point ranges within the unused positions that remain after the text message is placed: 11 for the first elements (Figure 3.3b) and 17 for the second and third element (Figures 3.3c and 3.3d). The probability of

a successful attack in 3 attempts is 3 over $24^3 \cdot 11 \cdot 17^2$ (i.e., 1 over about 14 millions).

It is worth noting that a malicious device has no clue about the success or failure of each attempt and after three failed attempts the human intervention is required to gain more attempts. Non-HID devices cannot maliciously mimic HID behaviour: once a device is authorized as non-HID, any attempt to mimic HID behaviour requires a re-enumeration that can be triggered by a detach signal, which in turn triggers a new authorization process by USBCheckIn.

The above analysis assumes that USBCheckIn has no security bugs and it is not compromised. While this is a demanding assumption, honoring it for a dedicated hardware is surely easier than honoring it for a software which runs on the host. In fact, the security of USBCheckIn is based on the security of a reasonably small and stable amount of software (the software we developed, USBProxy, libUSB, and the kernel), while for a host-based solution all software running on the machine should be trusted unless proper mandatory access control configurations are in place, which may not be feasible in many environments. Also, certification procedures are much easier for an isolated system.

Finally, we note that our approach has an obvious limit: it cannot prevent a *malicious HID* to actually allow the user to enter the challenge code. However, other proposals, like for example [TBB15b], have similar limitations and, to our knowledge, this is still an open problem.

Chapter 4

USBCaptchaIn: Integrated USB Attacks Protection

Cyber-attacks to critical infrastructures constitute a serious risk for society [Lew14]. Specifically crafted malware can be used by attackers to alter an industrial process or gather industrial secrets, and, in the end, gain some market or political advantage. Due to the inherent criticality of ICSs, best practices [SFS11] suggest to isolate the most critical parts of the system from other IT components, either physically or by means of firewalls. To overcome the limitation of a poorly connected environment, file transfers are usually performed by means of USB thumb drives and other *Removable Storage Devices* (*RSD*s). The use of RSDs turned out to be an important vector of malware spread [Rau11] making isolation efforts to protect ICSs from the rest of the IT systems largely ineffective. Further, the recent class of attacks called BadUSB [NK16] disclosed new threats that involve USB devices. These attacks are based on a modification of the device firmware, that forces the infected device, typically a thumb drive, to behave as a different kind of device, for example as a keyboard. In this way, a *malicious firmware* can inject commands that end up in a malware infecting the host. These malicious commands could download a malware from the Internet, but can also create it on-the-fly, for example, by "typing" the content of a malicious script and running it.

Regular antiviruses are largely ineffective against innovative malware, i.e. malware that exploit zero-day vulnerabilities, and, especially against BadUSB attacks since they exploit basic capabilities (e.g., typing operating system commands) that the user is normally allowed to use.

In this chapter, we present an architecture based on a dedicated hardware, called *USBCheckIn* that is an integration of the solutions presented in Chapter 2 and 3. The idea is to have an hardware that enables promiscuous use of RSDs in critical infrastructures, while preventing the spread of both conventional and firmware-based malware into the critical machines.

In our approach, machines are either *critical* (SCADA, embedded devices, etc.) or *regular* (personal notebooks, company PC, etc.). We consider regular machines and RSDs as possible sources and vectors of attacks against critical machines.

Our goals are to allow this kind of use while preventing (potentially malicious) data or code originated from regular machines to spread into critical ones as well as prevent the damage of critical machines by malicious firmware. To achieve the first goal, we rely on cryptographic integrity protection along with the use of authenticated data structures for efficiency. Our approach does not rely on malware signatures and is a strong obstacle to the spread of zero-day attacks, even if RSDs are used promiscuously in critical and regular machines. To achieve the second goal, we rely on an hardware-based "captcha". The basic idea is that the authenticity of a real *Human Interface Device* (HID) can be easily checked by asking the user to use it. Our hardware-based solution does not require any change to host systems and, hence, it is easily deployable in real ICS environments.

The rest of this chapter is organized as follows. In Section 4.1, we review the state of the art. Section 4.2 introduces actors involved in our solution and formalises the requirements we intend to meet. In Sections 4.3, we describe the security model and the threat model on which we base our work. Section 4.4 shows the architecture of the proposed solution. Section 4.5 provides a security analysis. In Section 4.6, we modify our solution to provide additional security features. Section 4.7 discusses the applicability of our approach in ICS environments. In Section 4.8, we present a prototype of the proposed solution and report informal feedback from experts.

4.1 State of the Art

To mitigate the risk for critical systems to be infected by a malware, an antivirus can be adopted and properly configured to scan the data stored in the USB thumb drive before any access.

Most commercial antiviruses perform detection based on a database of known malware signatures. This approach has some drawbacks: it cannot

detect zero-days attacks, it needs regular signatures updates to keep its effectiveness, its performances depend on the size of the data to be protected, and it cannot protect from generic tampering. Further, the class of attacks that leverage on the modification of the firmware (BadUSB attacks) makes regular antivirus largely ineffective since they use capability this kind of devices are allowed to use. However, it may detect or block further malware actions occurring after the BadUSB attack.

The primary countermeasure proposed against BadUSB was to *protect* USB devices by a firmware authentication feature that limits the firmwares that can be uploaded into a device to those signed by the device vendor (see for example [IRO16]). However, this approach protects devices but not hosts, which are secured only if they are forcibly limited to interact with just protected devices. Further, this strongly limits the choice of USB storage devices and it is insecure, if the limitation is delegated to error-prone user behaviour.

GoodUSB [TBB15b] is a software solution that aims at protecting the host against BadUSB attacks. When a new USB device is attached, a message is shown to the user, which must declare his/her expectation about the functionalities of the device. The user has to make a decision, that means there is the possibility to incur in a human mistake or a deliberate malicious human behaviour. A similar approach is adopted by [KS17, LHK+16]. USBlock [MW18] consider the timing of USB traffic similarly to certain the intrusion detection approaches for IP networks.

There are a number of products on the market that specifically address security for RSD (e.g., see [bit]) and USB thumb drives (e.g., see [top]). These are mostly focused on confidentiality, which, however, is not our primary objective. In these cases, support to integrity is on a file basis or on a block basis, and there is no integrity protection for the whole storage: an attacker can delete selected portions of data and also revert part of them to a previously saved version. Further, all solutions imply some form of authentication, usually password-based, but once the user is authenticated, full access to data is allowed, and a malware can easily infect the stored files. Further, the adoption of a password as a protection impacts the usability in term of ease to allow different people to use the device, i.e., the possibility to pass the device from hand to hand.

Integrity

Concerning techniques for checking the integrity of data, a large body of work is known in literature.

Protecting information by means of integrity in a scenario where exist different type of machines (i.e. regular and critical) recalls the well known Biba integrity model [Bib77], which describes a set of access control rules that can be used to protect the integrity of certain data. In the Biba model, each element is associated with an integrity level. The rules of this model deny any flow of information from lower levels to higher levels and can be summarised with the statement "no read down, no write up". This model is implemented in recent versions of the Windows operating system [RS09]. However, any form of access control on a filesystem must be performed by a trusted operating system, while we want an USB thumb drive to be promiscuously usable even on untrusted machines.

Many integrity approaches rely on robust cryptographic hash functions [RS04]. When the dataset to be protected is large, using hash functions is inefficient. In fact, for each change, even small ones, the hash of the whole dataset have to be re-computed. Also, to check the authenticity of a small part of the dataset, the hash of the whole dataset should be checked. *Authenticated Data Structures* (ADS) allow a user to efficiently update a cryptographic hash of a large dataset when just a small part of the dataset is changed. For an ADS, the hash of the whole dataset is called *root hash* or *basis*. They also allow a user to efficiently check the integrity of a small subset of data by only comparing against the root hash an *integrity proof* of size $O(\log n)$ with n the size of the data. Supposing that only the root hash is known to be genuine, it is possible to check the integrity of a small subset of data, efficiently.

Widely-known ADSs are *Merkle Hash Trees* (MHT) [Mer88] and authenticated skip lists [GT00]. MHTs are balanced search trees where leaves contains a cryptographic hash of the data and each internal node contains a cryptographic hash of a concatenation of the hashes stored in the children. For MHTs, the proof for a leaf l is made of the hashes stored in the sibling of the nodes that are in the path from l to the root.

For these data structures, updates and checks are performed in logarithmic time with respect to the size of the dataset, which is comparable to the efficiency of many indexes for databases and filesystems. For this reason, MHTes or other ADSs have been used in commercial, free, or research products. For example, MHTes were used for securing filesystems (see for example, [LKMS04, SvDJO12]). Authenticated data structures were also adopted to authenticate relational database operations [DGMS03]. The problem of efficiently using ADSs with regular DBMS was studied in [MS05, DBP07, PPP10a].

4.2 Actors and Requirements

The actors of our solution are machines, humans, and USB devices. USB devices can be USB HIDes (for simplicity we consider only mice and keyboards) or RSDs. Machines are divided in *critical* and *non-critical* (or *regular*). The set of critical (non-critical) machines is the critical (non-critical/regular) *realm*. We consider critical realm as the part of the system that requires special protection against malware generated in the non-critical realm and spread by means of RSDs, which include thumb drives.

In the USB protocol, the newly attached device declares its *type* to the host (i.e., if it is a keyboard, mouse, or RSD). As for the protocol, a device is allowed to act only according to the type it stated.

We consider two different possible ways to interact with machines: *data-flows* and *inputs*. We define data-flows as data transferred from machine to machine realised by means of read and write operations on RSDs. We define inputs as data generated by device that allege to be HIDes. We have a *malicious data-flow* when the flow is from a regular machine to a critical machine. We do not consider a data-flow as malicious when data pass through a regular machine to a critical machine with the mediation of a special component of the architecture called Gatekeeper (see Section 4.4). We have a *malicious input* when the input is generated by a RSD that states to be an HID and, hence, the input itself was not generated by an interaction between a human and the HID.

Most of requirements considered in the design of this solution are the same we took into account in Chapter 2. In the following, we remind the requirements already defined and list the new ones thought for this solution, namely, *Resiliency to Human Misbehaviour* and *Determinism*.

Discernment (Ds). The solution should prevent malicious data-flows and malicious inputs from reaching while allowing non-malicious ones.

Full Integrity (FI). The solution should be able to detect malicious data-flows. In other words, the solution has to detect all kinds of integrity violations, comprising deletions and restoration of previous versions of files or parts of them.

Timeliness (T). Malicious data-flows and malicious inputs should be detected before they reach critical machines.

Interoperability (I). The solution should be usable in conjunction with the existing systems and software suites (SCADA, HMI, harsh laptops, de-

velopment environments, inventory management, etc.), without requiring any invasive change to those products.

Usability (U). The solution should preserve the convenience and high usability perceived by users when using RSDs and USB HIDes.

Efficiency (E). The solution should not introduce asymptotic complexity overhead. Since operations for regular non-protected storage technologies run in at most $O(\log n)$ time, where n is the amount of data stored, we mandate our solution cannot increase this complexity. For HID, we accept only constant time operations.

Resiliency to Human Misbehaviour (R). The solution should be not vulnerable to human mistakes or intentional misbehaviour, hence, it has not to be based on any decision made by humans.

Determinism (Dt). The solution should be regarded deterministic for any practical purpose, that is, the probability that each single attack to be successful should be so small to make any brute-force attack not viable.

4.3 Security and Threat Model

We model an ICS as a set of machines (e.g. notebooks, workstations, SCADA, embedded systems, etc) equipped with USB ports. In our model we assume that only RSDs and HIDes can be used as USB devices. Different USB devices are not allowed. RSDs and HIDes can be used promiscuously in both critical and non-critical realm. We define a RSD as *malicious* when it states to be an HID.

All data generated in the critical realm are considered trusted. Data flows from regular to critical realm are forbidden and allowed only using a *gatekeeper* (see Section 4.4). All other information flows should be allowed. This ideal setting conforms to the Biba integrity model [Bib77] with just two integrity levels, where the rule "no read down, no write up" applies and the critical and regular realms are the higher and lower integrity level respectively.

Physical keyboards and mice are considered trusted and cannot be source of infections. RSD devices are considered non-trusted due to the possibility to change the firmware so that the device can act in a malicious way, i.e in a way that aim at damaging the system.

In our model, we consider an attack to be (1) any write operation performed by a regular machine on something that is supposed to be read by a

critical machine comprising addition, deletion and changes to data, metadata and directory structure, and (2) any input generated by malicious RSDs that reaches a critical machine.

4.4 Architecture

In this section, we describe the architecture of our solution. We equip each critical machine with an hardware, that we call *USBCaptchaIn*, intended to be connected to one of the USB ports of the host. USBCaptchaIn is itself provided with a USB port where other USB devices can be connected. The idea is that USBCaptchaIn should always be in the middle between any USB device that is intend to be connected with a critical machine and the critical machine itself. For this reason, we assume that USBCaptchaIn cannot be unplugged from the machine and each critical machine has all available USB ports either protected by USBCaptchaIn or disabled.

In our approach, regular machines are not equipped with any specific hardware or software. The general architecture is depicted in Figure 4.1. We also consider a special machine called *gatekeeper* that allows exceptional data transfer from regular to critical machines and whose details are described in Section 4.4. We now focus on the internal architecture of USBCaptchaIn.

USBCaptchaIn

The internal details of USBCaptchaIn are shown in Figure 4.2. USBCaptchaIn is equipped with two USB ports: one *upstream port* and one *downstream port* (also called *up/down ports*). The up port of USBCaptchaIn is connected to one of the USB ports of a critical machine that we intend to protect. All USB devices that a user intends to attach to the critical machine should be attached to a down port. For the sake of simplicity, we present the architecture of USBCaptchaIn with only one down port. The introduction of additional down ports is a straightforward extension of the approach proposed in this chapter.

Near the up port, there is an orange LED that indicates that the host is powering USBCaptchaIn and it is correctly working. Near the down port, there is a status red/green LED. The LED blinks green when one of the attached devices, possibly through a hub, is undergoing the authorisation procedure, turns fixed green when all devices are successfully authorized and can interact with the host, and turns blinking red if the authorization procedure of one of the attached devices failed too many times and the device must be disconnected.

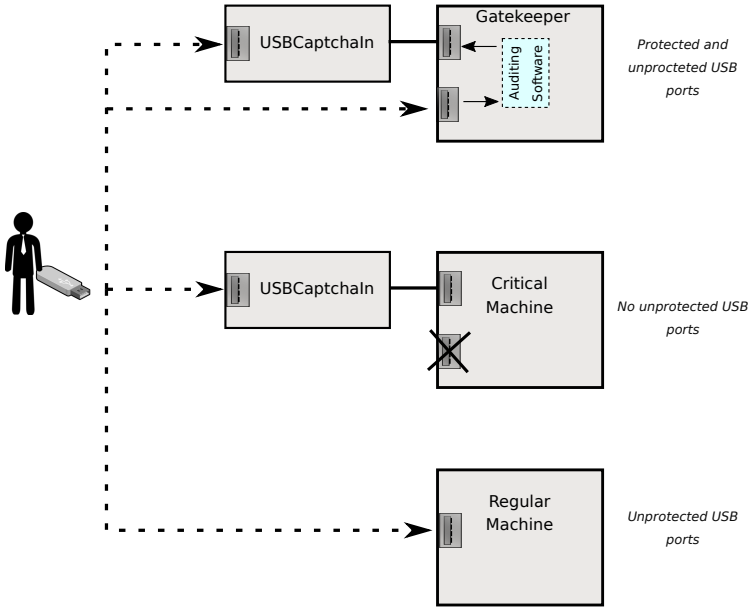

Figure 4.1: Components of the proposed architecture.

Messages required for user interaction are presented to the user on a 2.1" TFT display which is embedded in USBCaptchaIn.

In USBCaptchaIn, a software intercepts, inspects, redirects, and injects USB traffic between up and down ports. When a user attaches the device to USBCaptchaIn, the interaction between device and USBCaptchaIn follows the USB protocol: an *enumeration* phase is performed, during which the device is supposed to provide specific information about itself. USBCaptchaIn listens the capabilities declared by the device and records all information provided during enumeration. If the device declares to be a RSD, USBCaptchaIn behaves as described in Section 4.4. If it declares to be a HID, USBCaptchaIn behaves as described in Section 4.4. In both cases, the software realises the rules described in our security model. The actual operation depends on the state of the *authorisation procedure* described in the above mentioned sections.

USBCaptchaIn encompasses the following components.

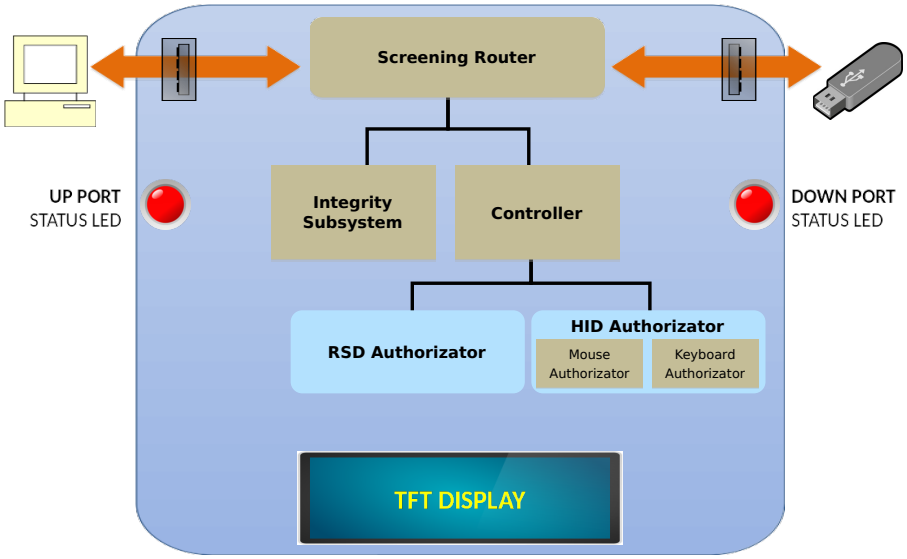

Figure 4.2: Components of the USBCaptchaIn device.

Screening Router. The *screening router* is in charge to route, USB packets to and from the ports, the controller, and the integrity subsystem. Its action depends on the authorization status of the device as explained in Sections 4.4 and 4.4, which may end up in filtering out certain USB traffic. It also performs standard USB enumeration, when a new device is connected.

Integrity subsystem. The *integrity subsystem* handles read and write operations related to RSDs and executes additional actions to guarantee that the data flows constraints of our security model are met (see Section 4.3). The integrity techniques adopted are detailed in Section 4.4.

Controller. The controller orchestrates the display, the status leds, and the authorizators to realise the authorization procedure and human-machine interaction described in Section 4.4 and 4.4. It also reconfigures the screening router depending on the result of the authorisation procedure.

Authorizators. The *authorizators* manage the authorisation process of RSDs and supported HIDes types. The correct authorizator is chosen by the controller according to what is declared by the device. The RSD Authorizator performs some consistency checks (see Section 4.4), while each HID Authorizator generates challenge codes for the user, keep track of the number of attempts, and check the correctness of the submitted code (see Section 4.4).

Integrity Subsystem and RSD Authorization

The integrity subsystem is in charge to ensure that data flows through RSDs comply to the constraints described in Section 4.3. Essentially, any data transfer from regular machines to critical machines should be blocked (exceptions are handled as described in Section 4.4). To efficiently perform this task and fulfil the Requirement E, we base our solution on ADSs (see Section 4.1).

Each RSD has a *secure partition* and an *ADS partition*. In the secure partition, we store data to be protected. In the ADS partition, we store an ADS over the data do be protected plus some additional cryptographic information. The state of the ADS is tightly coupled with the content of the secure partition.

RSDs store sequentially-numbered equal-sized blocks of bytes. In the USB protocol read and write operations issued by the host are at block level. The integrity subsystem intercepts these operations and inhibits those targeting blocks outside the secure partition. Operations that target the secure partition are performed along with additional tasks: reading encompasses integrity checks based on the ADS and writing encompasses update of the ADS. The root-hash of the ADS is kept signed in the ADS partition and used during the integrity checks. In the following, we provide the details of our solution.

In our approach, the identifiable operations are the same of the Chapter 2 that, for convenience, we remind in the following.

Protected-Read. Protected-reads are read operations performed by the critical machine on data stored in the secure partitions of a RSD. Protected-reads are mediated by USBCaptchaIn, which checks the integrity of the read data. If data is recognised as *genuine*, the data is reported to the critical machine as result of the read. If data is recognised as *tampered*, the read operation is blocked and an error is communicated to the critical machine. As detailed below, this fulfil Requirements Ds, T, R and Dt and partially FI.

Protected-Write. Protected-writes are the write operations performed by the critical machine on data stored in the secure partitions of a RSD. Protected-writes are mediated by USBCaptchaIn, which additionally updates the ADS and the signature of the root hash. During the ADS update some integrity checks are performed. If they fail, an error is communicated to the critical machine. This contributes to fulfilment of Requirements Ds, T, R, and partially FI.

Illegal-Write. An illegal-write is a write operation performed on a secure partition by a regular machine. The data changed by an illegal-write are always recognised by the subsequent protected-reads as tampered.

Plain-Read. Plain-reads are normal read operations performed by regular machines when reading any part of a RSD comprising the secure partition.

Secure and ADS partitions are realised as regular partitions on the RSD. Before the Integrity Subsystem starts to mediate the interaction between host and RSD, the RSD authorizator performs the following checks that guarantee the safety of read/write operations performed by the Integrity Subsystem.

1. It reads the partition table and check its compliance with its standard format to exclude attacks at this level.
2. It identifies secure and ADS partitions. If they are not present, this procedure is aborted and the RSD is not authorised.
3. It checks the correctness of format and size of the ADS partition.
4. If the above actions are successful, it configures the screening router to pass all host read/write requests to the Integrity Subsystem. Further, it asks the Integrity Subsystem to initialise itself for handling the identified secure and ADS partitions (see below).

Before describing the Integrity Subsystem we need to introduce some concepts. Within each partition, we adopt the common approach of identifying blocks by a numbering, assigning zero to the first block of that partition. The size of the partitions are defined at the moment of their creation. Creation of partitions is handled by a specific software that can create only partitions with *empty state*, which are always recognised as genuine by USBCaptchaIn. This software just resizes existing partitions (like standard partitioning tools do) and create secure and ADS partitions with the empty state. This procedure can

be performed on regular machines without affecting security, since partition creation does not imply any data flow in the sense described in Sections 4.2 and 4.3.

Each USBCaptchaIn device keeps, in local storage, its own *private key* and a corresponding *certificate* signed by a unique *Certification Authortiy (CA)*, whose public key is also stored. We denote by U an instance of USBCaptchaIn and by CERT(U) its certificate. Given a certain state of the secure partition Z, the hash of its current content, i.e. the root-hash of the ADS, is denoted by $h(Z)$ and its signature is denoted by $\text{sign}_U(h(Z))$. The ADS partition contains special fields named *signature* and *last writer certificate*, that stores $\text{sign}_U(h(Z))$ and CERT(U), respectively. where U is the last USBCaptchaIn that wrote in that RSD.

During initialisation of the Integrity Subsystem, right after the authorisation of the RSD, the last writer certificate is read and verified against the certificate of CA. The root-hash and its signature are read and verified against the last writer certificate. If any of these steps fails, the RSD is blocked and no operations are allowed on it, otherwise the root-hash is considered trusted.

When the critical machine asks to read a block b from Z through U, U retrieves the proof of b from the ADS (see Section 4.1). If it is consistent with the current trusted root-hash, the content of b is deemed to be genuine and is passed to the critical machine. When the critical machine asks to update Z through U, the ADS (comprising root-hash) and the signature should also be updated. This fulfils Requirements Ds, T, R, and Dt as far as access to RSDs is concerned.

To fulfil Requirement E, USBCaptchaIn has a caching mechanism: it performs update of the ADS partition only when there are no write operations pending, with a timer triggering the actual write, similarly to what regular operating systems do. The last writer certificate is updated the first time it is changed.

This approach does not completely fulfil Requirement FI. In fact, it cryptographically detects all kinds of tampering except the full reversion of both secure and ADS partitions to an older genuine version. A version of our approach that completely fulfil Requirement FI is presented in Section 4.6. A detailed security analysis is provided in Section 4.5.

We now describe the representation of the ADS in the ADS partition and its relationship with the secure partition (see Figure 4.3).

For the sake of simplicity, we assume the secure partition contains n blocks, where n is a power of 2. In this case, our ADS has n leaves, in one-to-one correspondence with the blocks of the secure partition, and is a complete binary

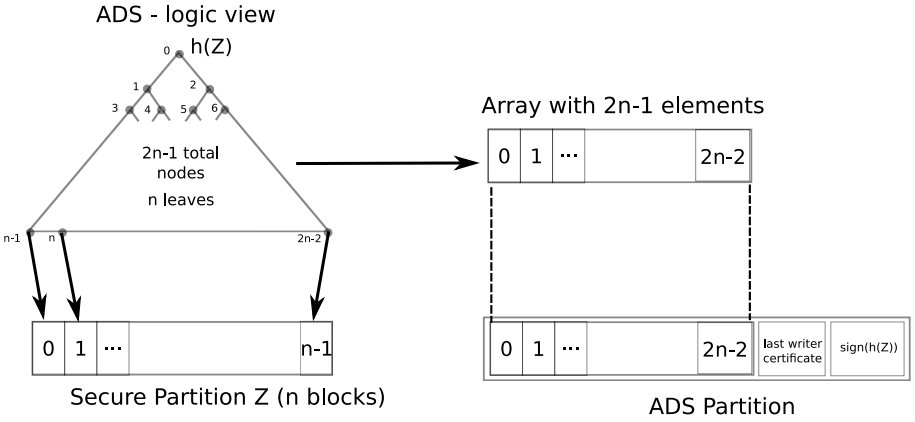

Figure 4.3: Relationship between secure partition, ADS, array, and ADS partition.

Markle Hash Tree (see Section 4.1). From the properties of binary trees, the total number of nodes (comprising internal ones) is $2n-1$. Each leaf of the ADS is the hash of the content of the corresponding block of the secure partition. We represent the ADS with an array of $2n-1$ elements (one for each node) following a sequential representation. We assume the nodes of the ADS to be numbered from 0 to $2n-2$, from the root to the leaves following a breath-first search order. According to this numbering scheme, each node v has $2v+1$ as left child and $2v+2$ as right child. We represent the ADS as an array whose elements correspond to the nodes of the ADS according to the above defined numbering. The array stores only the hash, since the relationship between nodes are implied by the above mentioned rules. The ADS partition additionally stores signature and last write certificate, whose size is fixed. Let m be the size of the cryptographic hash and B the size of the blocks in bytes. The size of the representation of the ADS is $(2n-1)m$ while the size of the secure partition is nB. It turns out that, for large n, the size of the ADS is $2m/B$ the size of the secure partition. For example, for $B = 4096$ (which is a typical size for disk I/O) and $m = 32$ (like for sha256), the ADS introduce a storage overhead of just about 1.6%.

HID Authorisation

In this section, we describe the HID authorisation process and interaction between USBCaptchaIn, the user, and the HID (i.e., a keyboard or a mouse), that occurs when the latter is plugged into the down port.

The authorisation process is based on a physical interaction between human beings and the HID just connected. After the enumeration phase, USBCaptchaIn starts the authorisation process asking to the user to input, by means of the HID itself, a randomly-generated challenge code.

The interaction for an HID that declares to be a keyboard is summarised in Chapter 3. Hereunder, we briefly reminder the procedure. The challenge code is a 5-characters alphanumeric string, which is communicated to the user by showing on the display a message. The user types the *elements* of the challenge code which for a keybord are characters. Each typed character is sent by the device to USBCaptchaIn according to the normal USB protocol for HID. The authorizator checks that each received element matches with the corresponding element of the challenge code. If it matches, USBCaptchaIn provides a visual feedback to the user. For a keyboard we colour green the corresponding correctly typed element on the display. Then, USBCaptchaIn loops expecting further elements and checking them until the challenge code is finished.

If the challenge inputs are correctly inserted, USBCaptchaIn notifies the user that the new device has been authorised and that it can be used.

At the same time, USBCaptchaIn impersonates the device toward the host by performing a "replay" of the enumeration that was previously recorded. Then USBCaptchaIn configures the screening router to short-circuit (logically) the up and down ports.

USBCaptchaIn keeps monitoring the exchanged packets between the host and the device in order to recognise hardware or logic *detach/re-attach* operations and then trigger a new authorisation procedure, when needed.

If any of the received elements does not match the corresponding element of the challenge code, the authorization is started over again showing a proper message

The authorisation process can be tried for a maximum of three times. After three wrong attempts, USBCaptchaIn definitely blocks the device, ignoring any more input from it.

To try again, the user has to detach and re-attach the device. This physical interaction makes a completely automatic brute-force attack impossible (see Section 4.5 for further details). Clearly, this is also the message displayed

when a device maliciously declares to be a keyboard and tries a repeated guess attack. This approach fulfil Requirements Ds and T.

If the device is a mouse, USBCaptchaIn asks the user to move a pointer on the display to draw a line between two randomly-placed targets, for 3 times. The procedure was selected to obtain a good compromise between security and usability.

In this case, each element of the challenge is a pair of points of the USB-CaptchaIn display.

Note that, instruction messages for the user are adaptively placed on the display on an unused zone, after the first point was chosen, to allow higher freedom in target positioning and make the required interaction harder to guess for a malicious RSD that pretends to be a mouse. See Section 4.5 for a security analysis.

Once the HID is authorised, USBCaptchaIn allows the HID to interact directly with the host, without adding any overhead to the USB packets exchanged by them, complying with Requirement E.

Gatekeeper

The above described architecture ensures high level of security and usability but does not support the relevant use case of bringing new legitimate data or software into the critical realm, like for example manuals or firmwares. They may be available in the regular realm and the first time they enter the critical realm we need to performs accurate audit. The role of the gatekeeper is to ensure that these checks are performed in accordance with the policy of the organisation.

The gatekeeper realises the "complete mediation" security principle [AJGS83]. It plays the same role of firewalls in networking and of the security reference monitor for operating systems. Practically, it is a dedicated machine that (i) has some USB ports non-protected by USBCaptchaIn from which any file can be read, (ii) runs a specific software that performs thorough audits on the read data, and (iii) writes the audited data on a secure partition on a RSD plugged into USBCaptchaIn, which in turn is connected to the gatekeeper on a different port. Clearly, the security of the software in the gatekeeper and of the whole gatekeeper machine is paramount.

The gatekeeper is the single point where data flow policies must be configured and are enforced. While the actual policies may depend on the specific industrial context, they should cover (a) authentication of the operator that requires the transfer, (b) authorization of the transfer according to the policy

of the organization, and (c) logging of the details of the operations performed to allow subsequent auditing. Our gatekeeping approach enables to scale-up the security level that an organization can achieve enabling the implementation of arbitrarily thorough analysis of the data to be transferred and arbitrarily complex workflow to obtain the authorisation, which may involve human decision.

4.5 Security Analysis

The intent of this section is to explicit a number of assumptions and to show that, under those assumptions, with the adoption of the proposed solution, the critical realm cannot be compromised. We recall that, protection from denial of service, i.e., making data not accessible, is not among the objectives of the proposed solution.

We consider the following assumptions:

(I) USBCaptchaIn does not have security bugs and it is not compromised,

(II) all HIDes and RSDs communicate with the host through USBCaptchaIn,

(III) the gatekeeper effectively checks (and possibly blocks) data flowing from the regular realm toward the critical realm,

(IV) critical machines cannot communicate with regular machines by means of other then RSDs and trusted console operators,

(V) attackers cannot obtain a private key (which we assume to be generated within USBCaptchaIn at production time and never exported) or force the CA to sign a new certificate and CA is not compromised,

(VI) the adopted cryptographic primitives have no security flaws, and

(VII) an attacker cannot unplug USBCaptchaIn from the host.

Our approach is not vulnerable to human mistakes or intentional misbehaviour. In fact, we do not ask to the user to take any decision. Integrity checks process is totally transparent to the user. During the authorisation process of HIDes, the user does not have to accept device features or choose in a list of allowed device classes. Actually, not even a malicious user can force USBCaptchaIn to authorise a device that claims to be an HID but offers no

physical means to the user to provide the requested challenge code. By the above considerations, we can argue that our approach fulfil Requirement R.

By Assumption (I), USBCaptchaIn cannot be compromised by inserting RSDs that contains malformed data (e.g., a malformed partition table or ADS partition).

By Assumption (IV), the only vector of attack are RSDs plugged into critical machines. By Assumptions (II) and (VII), all communications between USB devices and critical machines are mediated by USBCaptchaIn.

By Assumption (I), there is no way to bypass the authorisation process of HIDes than guessing and brute-force attack against the *challenge code*. While Assumption (I) is a demanding one, honoring it for a dedicated hardware is surely easier than honoring it for a software which runs on the host. In fact, the security of USBCaptchaIn is based on the security of a reasonably small and stable amount of software, while for a host-based solution all software running on the machine should be trusted unless proper mandatory access control configurations are in place, which may not be feasible in many environments. Further, certification procedures are much easier for an small embedded system whose elements do not change.

The authorisation process of HIDes is extremely well protected against guessing/brute-force attacks. For keyboard authorisation, the challenge code consists of 5 characters each ranging within 26 letters plus 10 digits. The probability for a malicious device to correctly guess a random challenge in three attempts is 3 *over* 36^5 (i.e., 1 over about 20 millions). For mouse authorisation, each challenge code consists of 3 pairs of points. The first point of each pair ranges within 24 possible positions. The second point ranges within the unused positions that remain after the text message is placed: 11 for the first elements b) and 17 for the second and third element. The probability of a successful attack in 3 attempts is 3 *over* $24^3 \cdot 11 \cdot 17^2$ (i.e., 1 over about 14 millions). The above considerations show that our approach meets Requirement Dt as far as HID authorisation is concerned.

It is worth noting that a malicious device has no clue about the success or failure of each attempt and after three failed attempts the human intervention is required to gain more attempts. Devices recognised as RSDs cannot maliciously mimic HID behaviour: once a device is authorised as non-HID, any attempt to mimic HID behaviour requires a re-enumeration that can be triggered by a logic detach signal, which in turn triggers a new authorisation process by USBCaptchaIn.

USBCaptchaIn cannot prevent a *malicious HID* (i.e., a keyboard or a mouse containing malicious code embedded in the firmware), to actually allow the

user to enter the challenge code. Other proposals, like for example [TBB15b], have similar limitations. However, solutions like [MW18] analyse the timing characteristics of the USB traffic which may detect a malicious HID. Nothing prevents to integrate into USBCaptchaIn a similar approach.

About the integrity checks process, critical machines can read only data stored in secure partitions. By Assumption (III), no malicious file from the regular realm is admitted in the critical realm. Hence, the only remaining possibility for an attack is trying to make critical machines to read tampered data from a secure partition.

Since RSDs are completely untrusted, the attacker (e.g., a compromised regular machine) can freely tamper with any data stored in the RSD (see Section 4.3). This includes data stored in the secure partition, and all data stored ADS partition, namely, ADS, signature, and last writer certificate. Let us consider a tampering of a secure partition. Since the result of any protected-read operation is checked against the signed hash, through the proof derived by the ADS in the ADS partition, the tampering of data stored in the secure partition is easily detected.

Now, let us consider an attacker who replaces the content of the secure and ADS partition with an old version of them. In this case, the check of the signed hash with the result of a protected-read matches. As mentioned in Section 4.4 we do not completely fulfil Requirement FI. See Section 4.6 for a description of the improvement that we propose to the approach described in Section 4.4 to fulfil Requirement FI completely.

The attacker can try to avoid detection by tampering also the ADS partition. An attack to the ADS, that does not change the root hash, requires to find a collision for the hash function on which the ADS is based, which is against Assumption (VI). On the other hand, to change the root hash the attacker should be able to violate the signature. However, this attack is ruled out by Assumption (VI) and, by Assumption (V). The attacker cannot get private key from USBCaptchaIn for Assumption (I).

Tampering only with the signature, the last writer certificate, or the ADS ends up in a false positive. In fact, in those cases, while data contained in the secure partition may be genuine, the integrity system has no way to prove it, hence, it behaves as if the secure partition was corrupted denying any access to it. We recall that ensuring data availability is not within the objective of our solution.

If a protected-write operation is partially executed, for example because the RSD is unplugged abruptly, the data in the secure partition, the ADS, the

signature and the last writer certificate may not be written or may be partially written. In this case, a protected-read, detects a tampering.

4.6 Ensuring Full Integrity and Providing Additional Functionalities

As noted in Sections 4.4 and 4.5, the solution presented till now, does not completely fulfil Requirement FI. In fact, reversion to a previous version of the secure parition, along with its consistent ADS parition, is not detected as tampering. This may be regarded as an acceptable behaviour or not depending on the context. In this section, we modify our solution so that Requirement FI is completely fulfilled. The changes that we propose are not free. They introduce additional complexity, and in a certain sense, additional costs. This is the reason why they are presented here as an improvement and not in the main design, leaving open the possibility to adopt the limited version in situations in which partial fulfilment of Requirement FI is acceptable. The changes described in this section also offer the opportunity to provide additional functionalities that may be desirable.

The new general architecture is shown in Figure 4.4. We introduce a *coordination service* (CS), whose functionality is to store key-value pairs in which each key is associated with one RSD (for example, the USB device identifier may be used for this purpose) and the value is the current root-hash of the ADS (and in turn of the current state of the secure partition). To connect with CS, a communication channel is needed. The bandwidth requirement for it is very small: only the RSD key and the cryptographic hash (the final root-hash) for each update of the secure partition should be sent. To support it, even a low bandwidth GPRS connection may be enough. We note that the availability of CS and of the connection might be an issue. In fact, if CS is not available or reachable, RSDs cannot be used. However, nothing prevents the adoption of standard high-availability approaches. Clearly, the communication between USBCaptchaIn and CS must be properly protected using standard technology (e.g., TLS [DR08]) and CS must be secured and authenticated.

If we accept the presence of a (low-bandwidth) communication channel, we can exploit it to provide an additional remote administration/monitoring functionality, which may be enabled on-demand under the control of CS. To realise this, we equip USBCaptchaIn with the capability to create a VPN toward CS, then a data connection between USBCaptchaIn and the critical machine can be created in two ways. If the critical machine supports fully fledged USB pro-

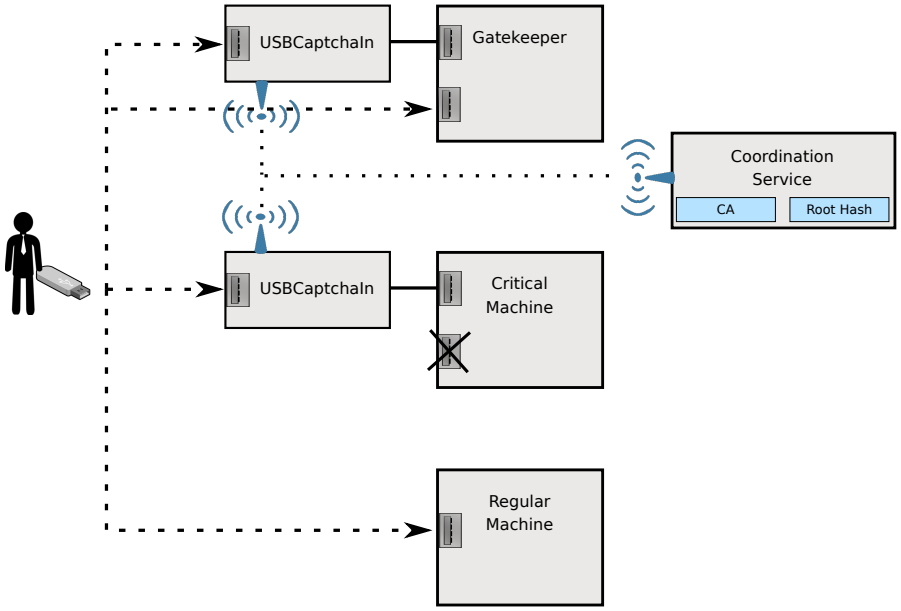

Figure 4.4: How the Coordination Service fits the architecture described in Section 4.4.

tocol, USBCaptchaIn may present itself during enumeration with an additional functionality of USB network card. Otherwise, a physical cable may connect USBCaptchaIn with a physical network interface of the critical machine.

This architecture may also support a certificate revocation procedure. In fact, CS may periodically provide to all USBCaptchaIn devices a certificate revocation list and possibly distribute new certificates.

Additional easy to implement functionalities are logging and monitoring of RSD activity and capability of administratively disable the use of RSDs on certain critical machines.

4.7 Applicability Considerations

In the following, we discuss the applicability of our approach in real ICS environments.

RSD Data Integrity Protection Contrary to current suggested best practices [SFS11], with our approach, users are allowed to use the same RSD with any machine without relaying on passwords and without the need to install specific software. It is enough to equip critical machines with USBCaptchaIn. Protections offered by the integrity subsystem are largely transparent to users, addressing Requirement U (see Section 4.2). When an RSD is plugged into a USBCaptchaIn, only the secure partition is accessible to the critical machine. When an RSD is plugged into a regular machine, secure and ADS partitions should not be touched. To avoid unintentional tampering of them, the secure partition can be logically write-protected using capabilities supported by certain filesystems (e.g., NTFS) or partition tables (e.g., GPT). In this cases, the secure partition can be mounted read-only on regular machines. Note that, if the above solutions cannot be adopted (for example for FAT-formatted drives with MBR partition tables), automatic read-write mounting of a secure partition may happen on a regular machine, which may end up in a detection of tampering due to automatic changes (e.g., update of mounted-bit flag, auto-defragmentation, etc.). To avoid this issue, we slightly change the behaviour described in Section 4.4: we store the content of the secure partition shifted of (at least) one block, so that, the first block(s) are unused and zeroed. USBCaptchaIn provides block numbers translations on-the-fly during protected-read/write operations. In this way, the filesystem cannot be recognised and mounting cannot occur on regular machines.

User Interaction with USBCaptchaIn We now discuss the interaction of the user with USBCaptchaIn with respect to usability (see Requirement U). For Concerning RSDs, no user interaction is required to authorise a RSD with well formatted secure and ADS partitions. HIDs authorisation requires the users to use the device in a regular way. If the device is a keyboard, they have to type. If it is a mouse, they have to click. A keyboard or a mouse attached to USBCaptchaIn can be used also to interact during the boot process and to execute recovery procedures.

Deployment Impact It is easy to deploy USBCaptchaIn in the critical realm since it does not need any driver on critical machines to work. USBCaptchaIn is totally independent from the operating system (OS) of the host it is connected to. It can be connect to any machine equipped with a USB port supporting standard USB protocol for keyboard, mouse and/or storage.

Our approach allows on organization to use any standard RSDs, promiscuously, complying with Requirement I.

The gatekeeper provides the organization with great flexibility about the security policies, but this means that they should be carefully designed and possibly integrated with business or decision processes. For strict security policies, traversing the gatekeeper may be costly, hence, it is advisable to deploy a critical machine realizing a repository of commonly used files ready to be used in the critical realm.

The coordination service may be realised in several ways. The simplest one, supporting only the full Requirement FI, is by ZooKeeper [HKJR10], which also supports high availability out-of-the-box.

The key management is quite simple. The integrity subsystem requires only the presence of an off-line certification authority which may already be present in the organization for other purposes. Aspects related with certificate revocations have been dealt with in Section 4.6.

We assumed RSDs and HIDes communicate with critical machines through USBCaptchaIn. This assumption can be easily guaranteed in several ways. For instance, a costly approach in term of money and changes required for the deployment, can be to integrate USBCaptchaIn directly into the hardware of critical machines. Instead, a cheaper approach, for example, can be to leave in critical machines only one USB port enabled and weld USBCaptchaIn to that port. Adopting one of these approaches, it is impossible to bypass the mediation of USBCaptchaIn without physical tampering.

4.8 Prototypical Realisation and Feedbacks from Experts

We realised two prototypes that implement the more important parts of the architecture described in Section 4.4. Our main objective is to evaluate the usability of USBCaptchaIn in practice. We showed the prototypes to ICS security experts. In this section, we also report their feedback.

We focused on assessing the fulfilment of the usability requirement for the authorisation process for HID devices and for accessing RSDs, that is for the use of the integrity system module.

Our first prototype realises USBCaptchaIn (see Section 4.4) on a Beaglebone Black board [bea16] on which a Linux kernel is running. In this prototype, we realised the elements that are needed to test the HID authorisation process, namely, screening router, controller, and HID authorizators. The software run-

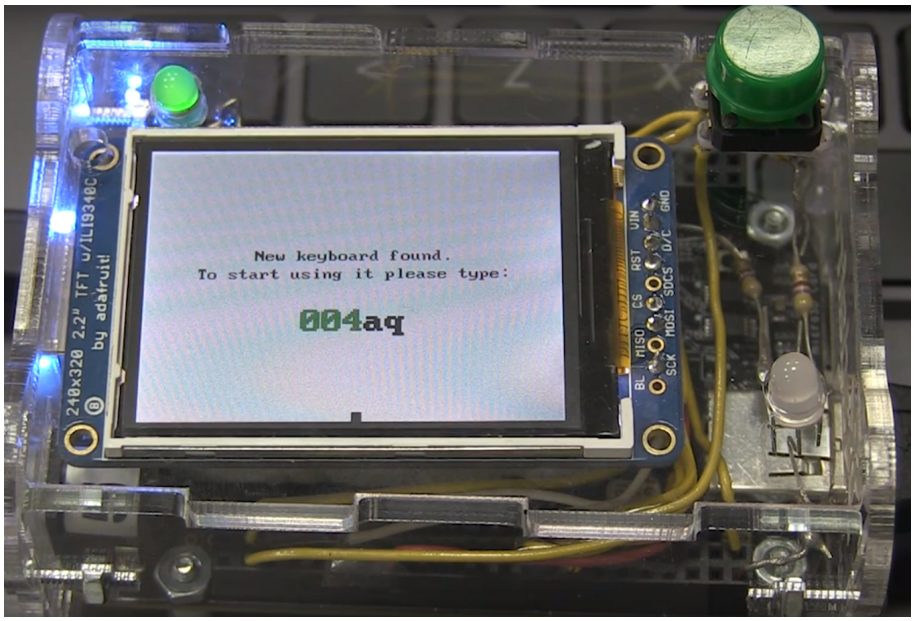

Figure 4.5: The hardware prototype for HID authorisation. The display shows a message during keyboard authorisation procedure.

ning on USBCaptchaIn is a customised version of USBProxy [Spi16]. The filter and the authorizator are USBProxy plugins. The communication between the device connected to a down port and the screening router is realised by means of libUSB [lib16], a library that supports the interaction with generic USB devices, while the communication between the host and the screening router is realised by means of gadgetFS, a linux kernel module that allows the Beaglebone to act as a client towards the host. We packaged our prototype so that it can be used on real systems. The final result is shown in Figure 4.5.

Our second prototype mimics the functionalities of the integrity system module (see Section 4.4). It consists of a software that, differently from our target design, is intended to be installed in critical machines. However, it provides the same level of security and a very similar user interaction. In our simplified realisation, we intercept file system actions at the system call level by using standard open source drivers [Fus, Dok]. User files and directories are stored in a regular directory of the RSD, while the corresponding ADS

is stored in an auxiliary directory according to a proprietary format. Files and directories can be accessed using the regular system calls of the operating system. Our software intercepts them and performs corresponding operations on plain data and on the ADS using regular read and write primitives of the kernel. More details about this prototype can be found in [GP16].

We showed our prototypes to security experts that we met within meetings of the Preemptive EU research project [MNG+17]. The feedback was very positive for both of them. We recorded no complaints about usability. One of the most appreciated aspects was the possibility to realise them in independent hardware. Some of the people we talked with, also suggested that USBCaptchaIn could be physically mounted within existing hardware whose USB ports should be protected, while keeping it isolated from the rest of the system for security. Some have pointed out that promiscuously using RSDs formatted for use with USBCaptchaIn into other systems may raise annoying false positives due to users accidentally manipulating integrity-protected data on a regular system. However, this can be easily mitigated by exploiting support for soft read-only protection of files (in the prototype) or partitions (in the target design), which should avoid most of the accidental modifications, as described in Section 4.7.

Concerning efficiency, on the HID authorisation side, once mice and keyboards are authorised, the mediation of USBCaptchaIn does not introduce any noticeable delay for a user with respect to the case where the HIDes are directly connected to the host. On the integrity protection side, our experiments performed with our prototype have shown that the additional overhead is negligible with respect to timings involved in normal user interactions, like working on documents and opening multimedia files. The overhead is mainly due to the fact that for each operation on data a corresponding operation on the ADS must also be performed. While for read operations this might be mitigated by proper caching (which is automatically performed by the operating system in our prototype), caching is not helpful for write operations. With the diffusion of USB 3.0 devices, this kind of overhead will progressively be less and less important. On the other hand, the limited computing power of a cheap board may severely limit the transfer bandwidth to and from the RSD, with respect to the bandwidth supported by USB 3.0. However, this aspect is more related with a market strategy trade-off and out of the scope of this chapter.

Chapter 5

Software Defined Networking applied in the Industrial Control System Environment

ICSs are the core of critical infrastructures. They are composed by many elements that interact by means of a communication network, which we call *ICS network*. Main elements of an ICS are embedded devices that control actuators or gather data from sensors. Special servers are in charge to collect data from these embedded devices, show them to the control room operators, record them in a database, change settings according to operators requests, etc. While the data that flow in an ICS network are very specific, standard networking technologies can be adopted for its implementation.

In the past decade, a growth of cyber-attacks directed toward ICSs has been observed [ICS11]. For the security of the ICS networks, best practices suggest to deploy network-based IDSes [SLP+15]. In regular networks it is acceptable to observe traffic in a small number of relevant points. However, for reliability reasons, in ICSs, Supervisory Control And Data Acquisition (SCADA) servers are close to sensors and actuators, hence, traffic is mostly local. Further, attacks to ICSs are potentially carried out by organizations (e.g., governments, intelligence agencies, terrorist groups) that can have insiders and that can carefully design attacks so that they pass unobserved by sparsely deployed IDSes. Tapping traffic close to all embedded devices and servers can easily lead to prohibitive costs. Certain solutions [nex11] make possible to route traffic replicas using the same ICS network towards one, or a few, IDSes, but

they are not able to guarantee the successful delivery of critical ICS traffic in all cases.

In this chapter, we present a methodological approach and an architecture to (i) allow an operator to choose which traffic has to be observed within an ICS network without installing new hardware, (ii) enable the use of the spare bandwidth in the network to forward the traffic to be observed toward an IDS, while avoiding packet loss for regular traffic, and (iii) guarantee that the IDS receives all the traffic that the operator configured to be observed in order not to introduce false negatives due to packet loss. Our solution takes advantage of the fact that topology and bandwidth usage are quite stable in ICS networks (see for example [TCB08]), allowing us to assume in advance knowledge of ICS network's traffic, since it derives from ICS design, and to perform a global off-line optimization of switching paths. Furthermore, we support the usage of the ICS network for additional and occasional traffic, which are always considered potentially dangerous. We assume that this traffic can be served with a best-effort approach while maximizing the endeavor in observing it. We propose an architecture that exploits the Software-Defined Network (SDN) approach as prescribed by the OpenFlow specifications [Spe13]. We evaluated our methodology against four network topologies, derived from real topologies and augmented with realistic networks in the domain of electrical distribution. Our experiments show that our optimization problem can be easily solved for those scenarios in reasonable time and our approach makes efficient use of the bandwidth when the topology allows it.

The rest of the chapter is organized as follows. In Section 5.1, we describe the state of the art. In Section 5.2, we describe the context of ICSs and introduce basic terminologies. In Section 5.3, we formally state the requirements that our solution should fulfill. In Section 5.4, we describe our methodology and our proposed architecture. Section 5.5 introduces the ILP formulation for our off-line optimization problem and in Section 5.6 we show the on-line algorithm for occasional traffic. In Section 5.7, we evaluate our approach against realistic scenarios. In Section 5.8, we extend our approach in order to relax some simplifying assumptions and handle special cases.

5.1 State of the Art and Background

ICS networks make use of proprietary protocols, as shown in [SLP+15]. Those protocols (e.g. ModBus [MOD96]) are tipically application-layer, and they allow the communication among ICS devices. In many cases, proprietary pro-

tocols are used also to compute routing [ODV], but this does not limit the adoption of different link-layer technologies [HCLP15] and new installations tend to be based on widely adopted standards, like Ethernet. Protocols adopted in ICS networks do not consider security aspects, hence, well known recommendations (e.g. [SLP+15]) suggest, among several other countermeasures, the adoption of IDSes. Forcing network traffic to cross the IDS is not so simple, especially if a network administrator needs to be flexible in the selection of traffic that has to be observed. Some flexibility can be gained by adopting proprietary protocols (like ERSPAN [nex11],[GM17]), which however offers an unhandy solution and does not guarantee that the rest of the traffic is not affected.

In the last years, a new centralized approach called Software-Defined Networking (SDN) is collecting the attention of the research community due to its promising benefits and, in particular, its flexibility in the selection of the paths to route packets [KF13]. There have been many attempts in exploiting SDN in security contexts. Some works [JW13, SBB13] propose to implement the IDS as an SDN controller module. We argue that such approach poses strong scalability issues and it is not advisable in the critical infrastructure context. A different approach consists in exploiting SDN to forward traffic towards one or more IDSes, as shown in [SSB+14, JHN+14] for the cloud computing applicative context. These solutions cannot be directly adopted in the ICS context since they do not provide any guarantee about the delivery of regular traffic.

A relevant aspect in our approach is traffic engineering. In [RTZ03], authors show that having a traffic-matrix allows traffic engineering problems to be easily solved. Usually, the traffic engineering problem is treated as a *multicommodity flow* problem whose solution is described in [AMO93]. Proposals that are specific to traffic engineering for SDN can be found in [ALW+14, BÍ15, AKL13]. At the best of our knowledge, our approach is the first attempt to apply traffic engineering to the specific context of traffic monitoring by IDS leveraging the coordinates of the topologies and traffic in ICS networks.

5.2 Application Context and Terminologies

For the sake of simplicity, we assume the ICS network to be isolated from the corporate network. While this is not completely true in general, still isolation (physical or by means of a firewall) is the best practice [SLP+15]. Hence, in the rest of the chapter, we only address traffic monitoring and management solely in the context of ICS networks. ICS networks connect several kinds of devices. For the purpose of our discussion we divide them in two categories.

We call the first category *essential*: devices in this category can have a very diverse nature, but they are essential for the correct operation of the ICS, are part of the ICS design, and are always connected to the ICS network. To let the reader better understand the applicative context, we provide a more concrete description. We distinguish them in *embedded devices* and *servers*. Embedded devices[1] control actuators gather data from sensors, and realize closed-loop control for restricted parts of the industrial system. They can send gathered data to servers and can be remotely controlled or configured, for example by asking to open/close a circuit switcher or by setting values, called *set-points*, that are objective of the closed-loop control, like, for instance, a target temperature of a heater. Typically, servers are (i) the *SCADA*, which gather data from embedded devices and process them, for example, to detect industrial process faults, (ii) the Human-Machine Interfaces (*HMI*) that show to *control room operators* the current status of the ICS and allow the operator to specify commands or new set-points for embedded devices, and (iii) the *historian* DB, which stores gathered data for future off-line analysis. We call the second category *non-essential*: occasionally, other devices can be attached to the ICS network, for example operators' notebooks to perform maintenance of ICS devices or to perform firmware updates.

We call *stream* a communication between two devices on the ICS network. We identify it by its source and its destination, specified by IP addresses. Even though communications are usually bidirectional, throughout this chapter we consider a stream to be unidirectional, which means that a full communication between two devices generally encompasses two streams. A stream can be critical or standard. In a *critical stream*, source and destination are essential devices and the properties about the stream are known in the ICS design phase. In particular their bandwidth demand, source, and destination are known. A reliable delivery of critical streams is considered fundamental for the proper working of the ICS and substantial resources are available to guarantee this, in term of design effort, equipment, etc. A *standard stream* is not essential for the current functioning of the ICS and it is not known in advance. It usually involves at least one non-essential device, but it can be involved in an occasional communication between two essential devices. Supporting standard streams is important to enable occasional use of the ICS network for maintenance or other non-critical activities, hence a best-effort delivery is enough for this kind of streams.

[1] For the reader that is acquainted with the ICS context, we are referring to Programmable Logic Controllers, Remote Terminal Units, Intelligent Electronic Devices, etc.

From the point of view of the security concerns, both kinds of streams are equally important, since attacks may involve any of the two with equal chance of disruptive effects. An *attack* to the ICS network consists in any action that introduces unexpected traffic or unexpected changes to standard traffic. To be more clear, it consists in a source of malicious traffic (e.g. a malware or a rogue device) or in the action of tampering with any critical or standard stream. We assume that switches cannot be tampered with. We point out that security of switching devices is out of the scope of this chapter. We suppose there exists a centralized Intrusion Detection System (*IDS*) in the ICS network, which is able to recognize malicious traffic and properly send alarms.

The goal of this chapter is to provide a flexible way to use a centralized IDS. To achieve this, we assume that a standard stream σ is duplicated, generating a *replica stream*; this action is performed at a network node that we call *observation point*. Each replica stream $\bar{\sigma}$, associated with σ, originates at the observation point and ends at the IDS. The extension to several IDSes requires minimal effort and it is discussed in Section 5.8.

5.3 Requirements

In this section, we list the requirements that our methodology should fulfill. We also point out the limitations of the current practice.

1. **Observation Points** – Our methodology should be able to support the observation of potentially any stream in the network, independently from topology and IDS placement. For security reasons, we prefer observation points close to the destination of streams.

   Concerning current practice, in certain switches, it is possible to remotely mirror a port and also tunneling the traffic of the replica (see for example the ERSPAN technology). However, this approach provides no control on the bandwidth occupation on each link and it is limited to specific vendors support.

2. **Reliable Replica Forwarding** – Our methodology should guarantee no packet loss for replica streams associated with critical or standard streams. This is important in order for the IDS to inspect all observed traffic and avoid false negatives due to packet loss.

   Concerning current practice, the adoption of remote mirroring technologies implies that the replica is delivered with a best-effort approach. To

overcome this, in principle, traffic engineering and QoS techniques might be applied. However, this considerably increases the architectural complexity. Further, a centralized management, like the one described in Section 5.4, is needed anyway.

3. **Reliable Critical Streams Forwarding** – Our methodology should be able to configure the ICS network so that, for the critical streams, no packets loss can occur due to congestion.

   This requirement is motivated by the fact that, due to Requirement 1, replica streams may easily overload some links and make the usual overprovisioning strategies ineffective. Actually, up to a certain extent, forwarding reliability can be realized by adopting reliable transport protocols like TCP. However, support of TCP is non-obvious for certain embedded devices. Further, retransmission could introduce a delay that is not acceptable in the ICS context and no bandwidth guarantee is provided. The adoption of QoS and traffic engineering exhibits the same drawbacks as discussed for Requirement 2.

4. **Standard Streams Usability** – Our methodology should allow operators to use the ICS network for occasional tasks, which results in injecting new standard streams. While the presence of these streams should not adversely impact the fulfillment of other requirements, we expect standard streams to be treated by the ICS network in fair way. Therefore, usage of the ICS network for occasional tasks produce the same outcome for all occasional users and applications.

We also consider the well-founded technology constraint that imposes not to split streams. In fact, if packets of the same stream take different paths, uncontrolled reordering can happen, which is detrimental for TCP performance at best and can change the semantics of datagram-based communications at worst.

5.4 A Methodology and an Architecture

In this section, we describe a methodology and architecture that solve the problem described in Section 5.2 with the aim of satisfying the requirements described in Section 5.3.

Our methodology assumes that the network is made of SDN switches that are compliant with the OpenFlow standard [Spe13]. We exploit the OpenFlow

features to: (i) configure network switches to forward critical streams on the basis of globally optimized paths, (ii) configure network switches to forward standard streams on the basis of paths chosen by an on-line greedy approach, (iii) instruct certain network switches (*observation points*) to duplicate traffic, for the streams that have to be observed (either critical or standard), and perform the first forwarding step of *replica streams* towards the IDS, (iv) configure network switches to forward replica for critical streams towards the IDS choosing paths that are globally optimized by our off-line approach, (v) configure network switches to forward replicas for standard streams along paths that are dynamically selected with our on-line greedy algorithm, and (vi) configure shaping of all streams at ingress network switches.

To meet Requirements 2 and 3, we configure the SDN network to shape each stream at its ingress node, so that packets enter the network at a specified constant rate and all packets exceeding the configured bandwidth are discarded. For critical streams, the configured maximum bandwidth is determined during the design as described below, so no packet drop should happen. For standard streams, this early limiting avoids congestion of internal nodes that could adversely impact critical streams. The shaping configuration exploits the *meter* feature of the OpenFlow specifications.

Our methodology encompasses a design phase and an operation phase (see Fig. 5.1). In the *design phase*, we require an *ICS designer* to determine the network topology and to list the critical streams along with their maximal required bandwidth. These data are provided as input to an *off-line routing solver*, which computes the configuration of the SDN switches for critical streams. More specifically, the input of the off-line routing solver encompasses (i) the network topology, (ii) the location of essential devices, (iii) the location of the IDS, and (iv) for each critical stream its source, its destination and its bandwidth requirement. The off-line solver produces, for each critical stream, (i) a forwarding path, (ii) an observation point, and (iii) a forwarding path for the corresponding replica stream starting at the observation point and ending at the IDS. The off-line solver is based on an ILP formulation, which is described in detail in Section 5.5.

In the *operation phase*, we mandate the adoption of a special architecture (shown in Fig. 5.1) in which an *SDN-controller* is in charge of configuring forwarding paths and meters to implement shaping. Its configuration is divided into two parts: one for critical streams and one for standard streams. The part related to critical streams is configured on the basis of the result of the off-line solver and does not change during operation. The part related to standard

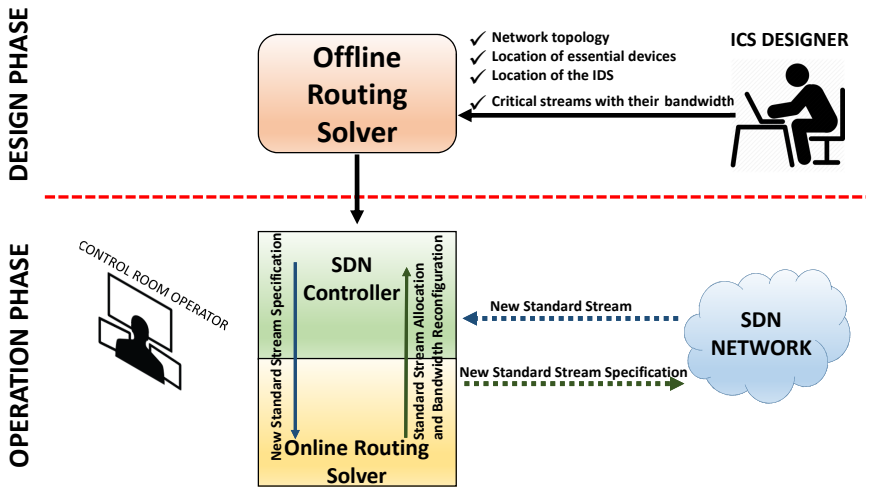

Figure 5.1: Architecture of our system, with both offline and online routing solvers.

streams dynamically changes during operation to adapt the configuration of the ICS network when the set of active standard streams changes. A *control room operator* can monitor the status of the ICS network during production time to have a clear picture of what streams are currently replicated and processed by the IDS. During operation, any new packet reaching a network switch that does not match any of the rules configured in the switch to forward critical streams is treated as the first packet of a standard stream σ. This packet is forwarded to the SDN-controller as in the classical SDN approach. To compute the forwarding path for σ, the SDN-controller takes advantage of an *on-line routing solver*. This solver shares with the controller the network topology, and the current available bandwidth on each link derived from currently allocated paths. It takes as input the source s and destination t of σ and computes (i) a forwarding path P for σ, (ii) an observation point $op \in P$ (preferably close to t according to Requirement 1), (iii) a forwarding path Q from op to the IDS, and (iv) a new assignment of bandwidth for all standard streams comprising

σ. The details of the on-line routing solver are described in Section 5.6. These information are used by the controller to re-configure the shaping for all standard streams but σ. The new standard stream σ is configured only after a small amount of time τ that is dimensioned so that packets related to previous standard streams that where admitted in the network with the old bandwidth allocation are guaranteed to reach destination.

Concerning the path selection, our algorithm has a greedy approach keeping unchanged all paths previously allocated for both kinds of streams. There are several reasons for this choice: (i) sophisticated optimization techniques, like those used in in Section 5.5, may take a considerable amount of time, which can easily be even larger than the lifespan of the new stream and impair the usability of the network for occasional activities, (ii) modifying the path of a current stream can introduce temporary inconsistencies in the routing that can lead to packet loss, which is against Requirements 3 and 2, (iii) since standard streams have usually a short lifespan, our main goal is to support them within the requirements listed in Section 5.3, keeping the optimization of their resource usage as a secondary goal.

5.5 Problem Formulation for the Off-Line Routing Solver

In this section, we present the ILP formulation that is at the basis of the off-line routing solver introduced in Section 5.4. For the sake of simplicity, we made a number of assumptions. Section 5.8 relaxes many of them and describes several extensions. Our formulation finds, for each critical stream σ, a forwarding path P_σ, an observation point op_σ, and the forwarding path of the replica stream $\bar{\sigma}$ from op_σ to the IDS d. Our formulation is a variation of the well-known multicommodity flow problem [AMO93]. In the following, the role of commodities are played by streams and we call *flow* the part of our solution that pertains to a certain critical stream. In this section, all the streams are critical unless different specification is provided. Our variation takes into account the following aspects: (i) streams are unsplittable, i.e., it is not allowed for a flow to bifurcate (see Section 5.3), (ii) flow demands (i.e., stream bandwidth) are fixed and all critical streams must be routed, (iii) each stream can generate a new replica stream originating at its observation point which must be the last traversed node before the destination, (iv) nodes of the network that represent embedded devices and servers do not have switching capabilities.

Since replica streams can take up a lot of bandwidth, we make the observation of a stream optional by introducing a *relevance* parameter ρ_σ for each stream σ, which indicates how important it is for σ to be the observed.

In our formulation, we use the following notation. The network is represented by a directed graph $G = (V, E)$, where V is a set of vertices and E is a set of directed edges. Each physical link corresponds to two oppositely directed edges (v, w). Each edge $e \in E$ has a capacity $C(e)$ that corresponds to the available bandwidth of the link in the corresponding direction. The set of vertices V is partitioned in two subsets: N, representing network switches, and M, representing devices with no switching capabilities (e.g., embedded devices and servers). We assume that there is no connection among vertices in M. The IDS is denoted by $d \in M$. For the sake of simplicity, we do not include the SDN-controller in this model, assuming that connectivity between SDN-controller and network switches is obtained either by a dedicated out-of-band network or by protecting part of the bandwidth of the SDN network using proper configurations. A stream is a quadruple $\sigma = (s_\sigma, t_\sigma, B_\sigma, \rho_\sigma)$ containing its source, its destination, its bandwidth demand, and its relevance, respectively. A corresponding replica stream is a triple $\bar{\sigma} = (op_\sigma, d, B_{\bar{\sigma}})$, where op_σ is its source (such that $(op_\sigma, t_\sigma) \in E$), d is its destination, and $B_{\bar{\sigma}} = B_\sigma$ is its bandwidth demand. The set of the critical streams is denoted *Crit*, the set of the corresponding replica streams is denoted *Rep*.

For each $e \in E$ we define $x_\sigma^e \in \{0, 1\}$ as a variable that has the following meaning

$$x_\sigma^e = \begin{cases} 1, & \text{if stream } \sigma \text{ is being routed through link } e \\ 0, & \text{otherwise} \end{cases}$$

Analogously, variables $x_{\bar{\sigma}}^e$ are defined for the corresponding replica stream $\bar{\sigma}$ associated with σ. If a stream σ is not observed, it will be $x_{\bar{\sigma}}^e = 0 \ \forall e \in E$.

We now define a few convenience functions. We provide definitions for a critical stream $\sigma \in \mathit{Crit}$ and a vertex $v \in V$, the corresponding definitions for replica streams $\bar{\sigma} \in \mathit{Rep}$ are analogous.

$$\textit{Outgoing flow} \qquad \mathrm{Out}_\sigma(v) = \sum_{(v,w)\in E} x_\sigma^{(v,w)} \tag{5.1}$$

$$\textit{Incoming flow} \qquad \mathrm{In}_\sigma(v) = \sum_{(u,v)\in E} x_\sigma^{(u,v)} \tag{5.2}$$

$$\textit{Vertex flow imbalance} \qquad F_\sigma(v) = \mathrm{Out}_\sigma(v) - \mathrm{In}_\sigma(v) \tag{5.3}$$

The bandwidth consumed by the critical and replica streams must comply with link capacities:

Capacity constraints.

$$\forall e \in E: \quad C(e) - \sum_{\sigma \in Crit} (x_\sigma^e + x_{\bar{\sigma}}^e) \cdot B_\sigma \geq 0 \tag{5.4}$$

For each critical or replica streams, we need to express flow conservation. Since flows are unsplittable, each stream generates (consumes) one unit of flow at its source (destination). Conservation is expressed separately for each stream:

Flow conservation and demand constraints for critical streams.

$$\begin{aligned} &\forall \sigma \in Crit \\ &\qquad \forall v \in V - \{s_\sigma, t_\sigma\}: \quad F_\sigma(v) = 0 \\ &\qquad \mathrm{Out}_\sigma(s_\sigma) = 1, \\ &\qquad \mathrm{In}_\sigma(t_\sigma) = 1 \end{aligned} \tag{5.5}$$

We now need to express similar constraints for replica streams. Let L_σ be the set of the possible observation points for σ, i.e., $L_\sigma = \{v \in N | (v, t_\sigma) \in E\}$. Flows should be balanced for all vertices in $N - L_\sigma$, and each vertex in L_σ can produce a unit of replica flow only if it is the last hop of the path assigned to σ (by unsplittable flow this is unique), and the IDS cannot be source of flow.

Flow conservation and demand constraints for replica streams.

$$\begin{aligned} &\forall \sigma \in Crit \\ &\qquad \forall v \in N - L_\sigma: \quad F_{\bar{\sigma}}(v) = 0 \\ &\qquad \forall v \in L_\sigma: \quad F_{\bar{\sigma}}(v) \leq x_\sigma^{(v,t)} \\ &\qquad \forall e \in E \text{ exiting } d: \quad x_{\bar{\sigma}}^e = 0 \end{aligned} \tag{5.6}$$

The above constraints also imply that $\mathrm{In}_{\bar{\sigma}}(d) \leq 1$, since for each σ only one variable $x_\sigma^{(v,t)}$ can be equal to one by the unsplittable flow property.

As stated above, only vertices in N have switching capabilities. Hence, all nodes in M should have, for their adjacent edges, flow equal to zero but for the streams for which they are source or destination:

$$\begin{aligned} &\forall \sigma \in Crit, \forall v \in M - \{s_\sigma, t_\sigma\}, e \text{ adjacent to } v \\ &\qquad x_\sigma^e = 0 \\ &\forall \bar{\sigma} \in Rep, \forall v \in M - \{d\}, e \text{ adjacent to } v \\ &\qquad x_{\bar{\sigma}}^e = 0 \end{aligned} \tag{5.7}$$

Our objective function consists of two parts: the first one expresses the residual capacity on all the links, while the second states the preference for observing the streams.

$$\max \sum_{\sigma \in Crit} \sum_{e \in E} \frac{C(e) - B_\sigma \cdot (x^e_\sigma + x^e_{\bar{\sigma}})}{C(e)} + \sum_{\sigma \in Crit} K \rho_\sigma \mathrm{In}_{\bar{\sigma}}(d) \tag{5.8}$$

Overall, we would like to maximize both parts. In the above formulation we give precedence to the second part. That is, we prefer to observe streams with respect to leaving more residual bandwidth. In order to enforce this, we multiply the second part by K, which we suppose to be big. We also state that ρ_σ must be integer and greater than or equal to one, and that K must be chosen to be larger than the range of values that the first part can take, namely $K > |E| \cdot |Crit|$.

5.6 Standard streams: methodology and algorithm

In this section, we describe our on-line algorithm for routing standard streams and their related replica streams. The algorithm takes as input a new standard stream $\sigma = (s, t)$, where s is its source and t is its destination, and, on the basis of the topology of the network, of the available bandwidth on the links, and of the previously allocated paths and bandwidth, it produces as result(i) a path P to be used to forward the packets belonging to σ, (ii) a switch $op \in P$ (observation point) where the traffic of σ is duplicated, (iii) a path Q to be used to forward the replica stream of the traffic of σ from op to the IDS, (iv) an assignment of bandwidth for all currently active standard streams, comprising σ, that should be configured in the ICS network as explained in Section 5.4, so that all streams are forwarded respecting Requirements 2 and 4.

Once the path for the new standard stream is computed, our algorithm re-assigns the bandwidth to all standard streams in order to fulfill Requirement 4. Bandwidth reduction entails a reconfiguration of limiting and shaping and we assume this operation can be safely performed without any packet loss. However, in order to avoid packet loss during the transition, we should ensure that no queue grows because of the simultaneous presence of packets bursts sent with previous configuration of bandwidth and packets of the new stream σ, which may account for an overall bandwidth greater than one of the links.

To address this issue, the new stream is admitted in the network only after a small amount of time τ that ensures that all packets injected with the previous bandwidth configuration are delivered. The parameter τ should be

On-Line Routing Algorithm for Standard Streams

Input:

- topology $G(V, E)$ where $V = N \cup M$ (see Section 5.5),
- va new standard stream $\sigma = (s, t)$ with $s, t \in M$,
- the IDS $d \in N$,
- sets S and C of standard and critical streams with **paths** and **bandwith assignment**.

Output:

- a path P from t to s,
- an observation point $op \in P$,
- a path Q from op to d,
- a new bandwidth assignment for streams in $S \cup \sigma$.

for all $e \in E$ **do** ▷ compute capacities for WidestPath()
  Let m be the number of standard streams that shares link e
  Let β be the capacity of e available for standard streams
  Assign to each edge $e \in E$ a capacity $C(e) = \beta/(m+1)$
end for
Let $L(i)$ be the list of vertices in N at distance i from t, with $i = 1 \dots k$, where $k = \text{dist}(s, t) - 2$.
$b_{\text{best}} \leftarrow 0$, $P_{\text{best}} \leftarrow$ none, $Q_{\text{best}} \leftarrow$ none
for i in $1 \dots k$ **do**
  for all v in $L(i)$ **do** ▷ v is a candidate observation point
    $SO \leftarrow$ WidestPath(G, s, v)
    $OD \leftarrow$ WidestPath(G, v, d)
    $OT \leftarrow$ WidestPath(G, v, t)
    $b \leftarrow \min(\text{bw}(OT), \text{bw}(SO), \text{bw}(OD))$
    if $b > b_{\text{best}}$ **then**
      $b_{\text{best}} \leftarrow b$
      $op \leftarrow v$
      $P_{\text{best}} \leftarrow SO|OT$
      $Q_{\text{best}} \leftarrow OD$,
    end if
  end for
  ▷ op is the best observation point at distance i from t
  if $b_{\text{best}} > 0$ **then**
    Recompute bandwidth assignment for streams $S \cup \sigma$ using Water Filling technique [RB07].
    return P_{best}, op, Q_{best}, new bandwidth assignment for $S \cup \sigma$
  end if
end for

Figure 5.2: Algorithm for handling a new standard stream.

greater than the maximum delivery latency of any packet, which, however, is a quite small number and is irrelevant for the vast majority of usage scenarios. The algorithm is formally described in Figure 5.2. As motivated in Section 5.3, the algorithm select observation points as close as possible to t and secondarily try to allocate the largest possible bandwidth. The latter choice takes advantage of the standard WidestPath() function [Pol60], which performs a depth first search with backtracking looking for the path with the widest bottleneck. Bandwidths to be used in WidestPath() are computed in the first step of the algorithm. To account for bandwidth reassignment for previously allocated standard streams, we estimated the bandwidth available for σ as the the total bandwidth available for standard streams divided by the number of streams after the allocation of σ.

Then, the algorithm starts enumerating the candidate observation points *op* ordered by increasing distance from t. Within the same value of distance, the *op* that allows the widest bandwidth b is chosen. Once b has been computed, it is compared with b_{best}, replacing it if and only if b is greater than b_{best} (lines 14 – 19). At this point, our algorithm recomputes all bandwidth assignment using the Water Filling (WF) technique [RB07] (lines 22 – 24), allowing us to find the maximum amount of bandwidth to assign to each stream. We realize WF in the following way. Suppose, the SDN-controller keeps a data structure that associates with each edge e the set of streams $S(e)$ passing through e. Let $c(e)$ be the available bandwidth for standard streams. WF looks for an edge $\bar{e}$ such that $\bar{e}$ has the minimum of $c(e)/|S(e)|$. WF consider $\bar{e}$ a bottleneck, hence, all streams in $S(\bar{e})$ are assigned bandwidth $c(\bar{e})/|S(\bar{e})|$ and discarded. Remaining bandwidth $c(e)$ are re-computed for all edges and the search is performed again until all streams are discarded and their bandwidth assigned. In this way, our algorithm successfully computes: i) P_{best}, namely the best available path; ii) *op*, namely the starting vertex for replica streams; iii) Q_{best}, namely the best path for replica stream; iv) new bandwidth assignment for S and σ.

The complexity of the WidestPath() functions is $O(|E|)$, as it is based on BFS algorithm, and it is run on each vertex a constant number of times. Hence, the observation point is found in $O(|V||E|)$ time. The WF takes $O(|E||S|)$. Therefore, the overall worst case time complexity of our on-line algorithm is $O(|E|(|V| + |S|))$. Actually, in the most common cases, we think the *op* is found in time much smaller than $O(|V|)$, so the time complexity can be often regarded to be $O(|E||S|)$.

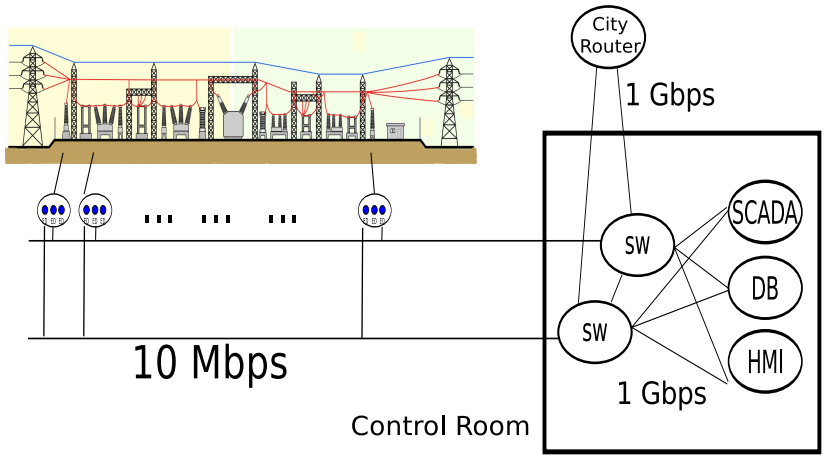

Figure 5.3: Details of the electricity distribution's substation.

5.7 Evaluation

We validated our approach from three points of view: (i) we assess the efficiency of our implementation with respect to computation time on realistic instances, inspired by the electricity distribution domain, for both on-line and off-line routing solvers, (ii) we show the efficiency of the bandwidth allocation of the on-line routing solver for standard streams, and (iii) we discuss the ability of our solution to meet requirements listed in Section 5.3.

We identified four different realistic topologies in the following way. We selected four large topologies form topology-zoo.org that are equipped with real link bandwidths or that are fairly mashed. When no links bandwidth are available 1Gbps links was assumed. We considered each node n to be a router associated with a *city*. We equipped each city with a number of electrical *substations* whose ICS network is connected to n. Let B_n be the sum of the bandwidth of all links incident to node n. The node with the largest value of B_n is also equipped with one IDS serving the whole network. The city associated with node n, is equipped with q_n identical substations. The total number of substations in the network is $q = \sum_n q_n$. The dimensioning of q_n is provided below. The network of a substation is designed on the basis of information that can be freely found in the Internet[2]. Figure 5.3 shows the

[2]Each of them modeled following the Wikipedia description `https://en.wikipedia.org/`

	Qty	From SCADA Bandwidth	To SCADA Bandwidth
Voltage Meter	2	10 Kbps	100 Kbps
Circuit Switches	2	1.5 Kbps	1.5 Kbps
Breakers	2	1.5 Kbps	1.5 Kbps
Current Meters	2	10 Kbps	100 Kbps
Power Transformer	1	50 Kbps	500 Kbps
HMI	1	30000 Kbps	3000 Kbps
Historian DB	1	30000 Kbps	3000 Kbps

Table 5.1: Elements of a substation with the bandwidth of the streams used for the evaluation.

topology of a single substation with its connection to the router and Table 5.1 shows the devices it contains. Industrial process data are communicated from embedded devices to the local scada system, and in turn to the HMI and to the DB. The amount of bandwidth required by these communications is shown in Table 5.1, which also show the quantity of each sensors/actuators. For the relevance, we chose always the value 1. We equip each city with a number q_n of substations according to a decreasing power law distribution. In practice, nodes n are sorted by their value of B_n. For n with the largest B_n, we state $q_n = 10$. For n in position i, $q_n = \lfloor 10/i^{\alpha} \rfloor$, where α is chosen between 0.7 and 1. When setting the capacities of the edges we reserved 5% of the bandwidth for standard streams. Data about used topologies are shown in Table 5.2.

To validate our off-line routing solver, we instantiated the ILP problem for our four topologies and solved them using Gurobi optimizer ver. 6.5. The formulation set up was performed by using the Python API. The corresponding code is available on the Internet [sdn]. The computation run on a workstation equipped with 8 processors Intel Xeon 2.8GHz. Results for the off-line solver are shown in Table 5.3. The evaluation shows that the formulation of Section 5.5 can be practically used. Considering that the foreseen usage of the formulation

wiki/Electrical_substation

	From Topology Zoo					Input for experiments			
	Name	$\|N\|$	$\|E\|$	min bw (bps)	max bw (bps)	q	$\|N\|+\|M\|$	$\|E\|$	num. strms
1	Cesnet	10	9	200M	600M	35	501	920	770
2	AttMpls	25	56	1G	1G	50	726	1357	1100
3	Agis	25	30	45M	155M	42	614	1123	924
4	Uninet	74	101	1G	1G	95	1405	2572	2090

Table 5.2: Data about original topologies, and topologies used in the experimentation.

	Results (off-line)		
	gurobi execution time	number of observed streams	max %bw on edge
1	12s	764	97.795%
2	30s	1100	62.060%
3	33s	869	98.058%
4	421s	2087	99.455%

Table 5.3: Results of the experimentation for the off-line routing solver.

is during design, running times are quite small. This makes us believing that our approach could be successfully used even in much larger scenarios. Even though, solving times are small, they are not suitable for an on-line use. This justify the introduction of the specific ad-hoc on-line solver, whose algorithm was presented in Section 5.6.

To validate the on-line routing solver, for each network, we randomly generated a sequence of events (available at [sdn]) as follows. We suppose that standard streams are initiated by (human) operators, whose number is proportional to the network size. We choose to have as many operators as substations (i.e., q). Each operator u is attached to a switch $s \in N$ chosen uniformly at random and generates a sequence containing two kinds of events: (i) begin(c, u, t) operator u starts a connection, identified by c, with machine $t \in M$, and (ii) end(c) connection c ends. Interarrival time between begin of connections is exponentially distributed with mean $1/\lambda$. Duration of each connection is exponentially distributed with mean $3/\lambda$ (i.e., each operator on average connects to 3 machines at the same time). We set $1/\lambda = 5$ minutes and the sequence spans about 10 minutes (from 176 to 576 streams).

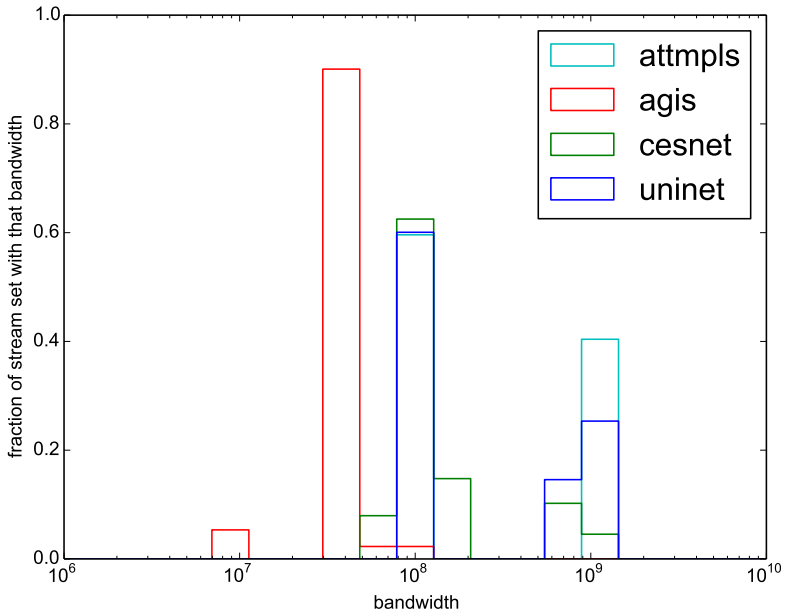

Figure 5.4: Density of bandwith assigned to streams for each topology (log scale on the x-axis).

We initialized the status of the solver with the output of the off-line solver for critical streams. Then, we run, for each network, the on-line solver on its sequence of events generated as described above. Figure 5.4 shows a density diagram, that has on the x-axis possible bandwidth values and on the y-axis the fraction of streams that had that bandwidth assigned in our experiments. In our experiment, assigned bandwidth is always very close to the maximum of the backbone bandwidth. Sometime, if source and destination of the stream are close each other, assigned bandwith can be larger (cf. Table 5.2).

The off-line optimization, together with the traffic shaping approach described in Section 5.4, ensures compliance to Requirements 1, 2, and 3. Further, the inclusion of standard streams is performed only by using the spare bandwidth of each link, thus protecting critical stream and replica streams from packet loss due to congestion (see Section 5.6). Requirement 4 encompasses two essential aspects: fairness of bandwidth allocation and response time. Our

approach handle all streams always assigning the same bandwidth to all of them and dynamically adapting it on the basis of the current needs. This ensures fairness at expense of some bandwidth waste, since certain streams may not use the whole bandwidth assigned to them. To improve this aspect, dynamic polling of bandwidth usage should be adopted [ALW+14], however, we believe that in the ICS context, this approach may not be worth the effort. Concerning response time, this mostly depends on the internal architecture of the SDN-controller. A further aspect is the time τ the controller have to wait to be sure no packet loss occurs when the bandwidth of certain streams have to be reduced (see Section 5.4). Since τ should be greater than the time a packet traverse the network, we expect it to be no more than a few milliseconds, which should be negligible for all applications that are reasonable to use in the ICS context.

5.8 Possible Variations and Improvements

In this section we discuss possible variations to the approach described in Sections 5.4, 5.5, and 5.6.

Bandwidth Reservation for Standard Streams. Our approach statically allocate bandwidth for critical streams and their replica streams, using the spare bandwidth for standard streams. However, it is easy to use our formulation to explicitly save some bandwidth for this purpose during design by artificially reducing the capacities $C(e)$ of Constraint 5.4.

Dynamicity. In the description of our approach, we suppose that the needs for monitoring the critical streams are known in advance and embodied in the relevance parameters ρ_σ. However, there are situations in which we may want to dynamically choose which stream IDS has to analyze. For example, when an anomaly is recognized, we may want the IDS analysis to focus on the devices close to it, possibly momentarily giving up the inspection of traffic of other devices to free up network and IDS resources. This can be supported by implementing in the controller with capability to switch off observation of critical streams upon request of the control room operator. Further, operator may explicitly ask for observation of a critical stream σ that was currently not observed. To implement this operation, a search for the widest path starting from the last hop before t_σ to the IDS have to be performed. If the resulting available bandwidth on the widest path is greater than B_σ, the SDN-controller set up the rules for duplication and forwarding toward the IDS, otherwise the search can be done backward along the path from the t_σ to s_σ. Alternatively,

since this somewhat relaxes the support for Requirement 1, the bottlenecks identified by the widest path algorithm can be used to suggest a set of streams whose observation can be switched off to free up enough network resources to satisfy the operator request.

Limited IDS Resources. In our description, we supposed that the IDS has unlimited computational power. While this might be reasonable if the IDS is based on cloud technologies, often the designer should deal with IDS limits. If we suppose that the IDS is known to scale up to a certain bandwidth B_d, the formulation of Section 5.5 can support it by simply introducing the following constraint.

$$\sum_{\bar{\sigma} \in Rep} \mathrm{In}_{\bar{\sigma}}(d) \leq B_d \tag{5.9}$$

However, special care should be taken in handling standard streams. In fact, during the off-line optimization, some IDS bandwidth should be saved for the analysis of standard streams replicas. Further, on-line routing solver must consider the IDS bandwidth when calculating the new bandwidth assignment for all the standard streams in the WF phase. Essentially, both on-line and off-line solver can address the problem as if the IDS were reachable only through a link of capacity B_d.

Support for Multiple IDSes. For the sake of simplicity, in our description, we assumed that only one IDS is present in the ICS network. However, there are situations in which it might be convenient to have more IDSes $d_1, \ldots, d_k \in D$ distributed across the ICS network. Hence, a stream can be observed by any of the IDSes. The formulation of Section 5.5 can be changed to support this in the following way. Variables $x^e_{\bar{\sigma}}$ are substituted with distinct variable sets $x^e_{\sigma,d}$ for each IDS $d \in D$. The functions $\mathrm{Out}_{\sigma,d}(v)$, $\mathrm{In}_{\sigma,d}(v)$, and $F_{\sigma,d}(v)$ are defined for each $d \in D$ as obvious variations of Equations 5.1, 5.2, and 5.3. In Constraint 5.4, $x^e_{\bar{\sigma}}$ should be substituted by $\sum_{d \in D} x^e_{\sigma,d}$. Constraints 5.6 should be substituted by

$$\begin{aligned} &\forall \sigma \in Crit \\ &\qquad \forall v \in N - L_\sigma : \quad F_{\sigma,d}(v) = 0 \\ &\qquad \forall v \in L_\sigma : \quad \textstyle\sum_{d \in D} F_{\sigma,d}(v) \leq x^{(v,t)}_\sigma \\ &\qquad \forall d \in D,\ \forall e \in E \text{ exiting } d : \quad x^e_{\sigma,d} = 0 \end{aligned} \tag{5.10}$$

Since only one variable among $x^{(v,t)}_\sigma$ can be greater than zero (by unsplittability of flows), the second inequality implies that only one IDS is involved in the

observation of σ. The second of Constrants 5.7 should be substituted by

$$\begin{gathered}\forall \sigma \in Crit, \forall d \in D, \forall v \in M - \{d\}, e \text{ adjacent to } v \\ x^e_{\sigma,d} = 0\end{gathered} \tag{5.11}$$

Finally, the objective function should be changed into

$$\max \sum_{\sigma \in Crit} \left(K \rho_\sigma \sum_{d \in D} \mathrm{In}_{\bar{\sigma}}(d) + \sum_{e \in E} \frac{C(e) - B_\sigma \cdot (x^e_\sigma + \sum_{d \in D} x^e_{\sigma,d})}{C(e)} \right), \tag{5.12}$$

With these changes, the formulation automatically perform IDS assignment to streams so that objective function is maximized.

Flow Table Size Control. In SDN networks, the number of rules configured in each network switch is a concern. In fact, rules occupy entries in limited size flow tables. Since, the SDN-controller configures a rule for each outgoing stream, limits to the flow table can be take into account by the following constraints, where $FT(v)$ is the maximum number of rules that can be configured in the switch v.

$$\forall v \in N \sum_{\sigma \in Crit} \left(\mathrm{Out}_\sigma(v) + \sum_{\forall d \in D} \mathrm{Out}_{\sigma,d}(v) \right) \leq FT(v) \tag{5.13}$$

Chapter 6

Integrated Solution for Industrial Control System Defence

Industrial Control Systems (ICSs) are involved in control processes of several kinds of critical infrastructures, like energy production and distribution, water distribution, and transportation. In the past decade, a growth of cyber-attacks against ICSs has been observed [ICS11]. Historically, SCADA (Supervisory, Control and Data Acquisition) systems, PLCs (Programmable Logic Controllers), RTUs (Remote Terminal Units) and other elements of ICSs are built to provide high levels of availability, safety and reliability, but are not prepared to contrast software attacks effectively. Specifically, crafted malware can be used by attackers to alter an industrial process, possibly endangering human lives, or to gather industrial secrets. In several cases reported, the target of the attack to a Critical Infrastructure was an organization aiming at gaining some market or political advantage. This kind of attackers may have available much larger resources than average hackers, being therefore able to perform quite advanced attacks. This kind of attacks are usually referred to as *Advanced Persistent Threats* (APTs). APTs may include exploits to several zero-day vulnerabilities, may perform special actions to evade antiviruses, and may carry out infiltration inside organisations that stay undetected for years (see for example [VGA13, FMC11b]).

A number of standardisation efforts aimed at providing guidelines for cyber-security within the ICS context (see for example [SFS13, Int15, Nor13]). Usually, they strive to fit standard IT (Information Technologies) approaches and tools to ICSs. Indeed, ICSs are very special systems, what makes the appli-

cation of usual approaches hard. A quite deep analysis of the shortcomings of current standards is provided in [KMP+15a].

In this chapter, we provide an overview of the architecture of a framework resulting from Preemptive project [Par] emphasising the role of the solutions described in Chapter 2. Furthermore, we suggest how the integration of the solutions presented in Chapters 3, 4, and 5 can enhance the effectiveness of the Preemptive framework.

The goal of Preemptive was to improve the cyber-security of ICSs by developing innovative methods and tools to detect and protect them from cyber-intrusions, aiming at being effective even for those involving APTs. Each Preemptive tool addresses a particular detection and/or prevention problem, focusing on one of several aspects, e.g., industrial process, communication network, embedded devices, SCADA servers, or the use of USB thumb devices. The Preemptive project has also devised a specific risk analysis methodology [KMP+15b].

Special care was put to consider the peculiarities of the ICSs, both for exploiting the advantages of the specific context and for taking into account its constraints. A remarkable feature of the Preemptive approach is the combination of data coming from all tools into a single stream for real-time analysis, and its storage into a single database for historical analysis.

Any detected event is reported to the user into customised Human-Machine Interfaces (HMIs), which show the cyber-security state of the ICS and its evolution over time, highlighting the occurrence of anomalies to the operators. In the described framework, correlation operations are carried out over the detected events, so that small, apparently irrelevant and independent, events coming from different detection tools can be aggregated into one single event with higher severity, allowing the operator to handle it properly. To conclude, a tool for asset assessment provides the baseline inventory for both event correlation and risk assessment methodology.

This chapter is organised as follows. In Section 6.1, we review the state of the art. Section 6.2 provides an overview of the project and mentions its comprehensive architecture. Section 6.3 shows the results of a realistic testbed. Section 6.4 discusses limitations and hypothesis. Section 6.5 we analyses how integrating solution presented in Chapter 2, 3, 4, and 5 can improve the capa-

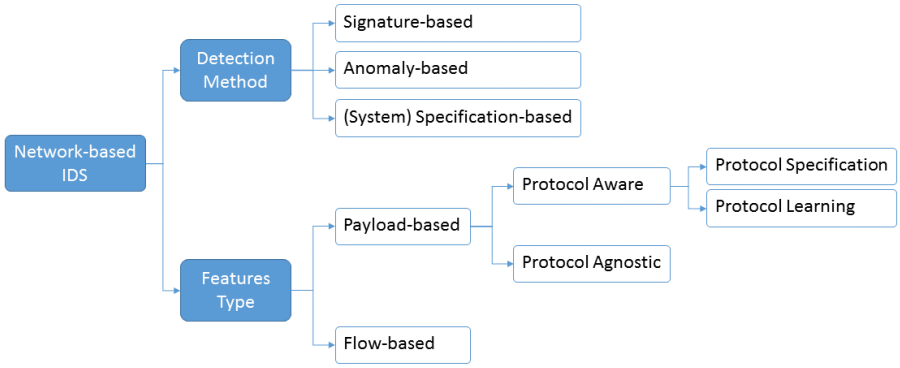

Figure 6.1: A Taxonomy for IDS.

bilities of Preemptive framework.

6.1 State of the Art

Preemptive is a quite large project. In this section, we provide only an overview of the most relevant references. Examples of surveys about Network-based Intrusion Detection Systems (IDSs) are [DC11, SM08, SSS+10]. Specific results for SCADA systems are listed in [BS]. A possible taxonomy for IDSs derived from literature is provided in Figure 6.1, while a taxonomy of Host-based Intrusion Detection techniques is provided in [JP11].

Examples of different approaches that can be found in literature are [HYQC09, WFP99, ADCE10, CH14, WD01, WZY06]. Yang et al. [YUH06] proposed an approach tailored for SCADA systems.

General techniques for anomaly detection in discrete sequences are described in [CBK12].

Part of the Preemptive research work performs anomaly detection on data representing the evolution of the industrial process. The used techniques belong to the class of Artificial Immune Systems [AIS] and in particular to the class of Negative Selection Algorithms (NSA) [FPAC94].

Concerning intrusion detection in embedded systems, Reeves et al. [RRL+12] propose a rootkit detector. Cui et al. [CS11] propose a "symbiote" mechanism, specifically designed to inject intrusion detection functionality into the firmware

of the device. Hardware based solutions were also proposed in [AvdWL+13, FPC09, DKS14, AGH14].

6.2 The Preemptive Project

The Preemptive (PREventivE Methodology and Tools to protect utilitIEs) FP7 European Project aims at providing innovative solutions to enhance cyber-security of ICSs. It proposes tools and methodologies to detect and prevent cyber-attacks targeting ICSs of utility companies [VS16]. Preemptive combines typical low-level detection tools (e.g., dealing with network traffic and system calls), with process-level misbehaviour detection tools, which are able to detect anomalies by analysing the physical quantities measured on the system. Thereby, the main contribution of Preemptive consists in the combination of process, network and host IDSs able to monitor the whole critical infrastructure, by means of a correlation engine that collects all the events, warnings and alarms generated by the different tools, elaborating runtime and historical data to identify APTs, zero-days attacks and other possible complex attacks.

All these components have been designed taking into account common ICSs and SCADA vulnerabilities that may be exploited by resourceful and motivated attackers, among which we can distinguish:

- poor networking stack implementations, that make components vulnerable to Denial of Service (DoS) and buffer overflow attacks;
- components exposing interfaces, that allow the configuration or control of process automation functionalities;
- protocols not defining user authentication or data integrity features, that allow attackers with network access to manipulate process control information.

For what concerns the architecture, the tools and modules that constitute the Preemptive platform and monitor the ICS components, devices and networks are eight, namely:

Host level IDSs:

- IT-HIDS (Host IDS for IT components): monitors and checks anomalies in standard IT devices (e.g., SCADA servers, historian server, engineering workstations).

- ED-HIDS (Host IDS for Embedded Devices): monitors and checks anomalies in embedded devices (e.g., PLCs, RTUs).
- HIS (Host-based Integrity System): checks the integrity of storage devices (e.g., USB thumb drives).

Network level IDSs:

- P-NIDS (Payload-based Network IDS): analyses the packets content to check anomalies.
- F-NIDS (Flow-based Network IDS): monitors the network looking for anomalous traffic behaviours.

Process level IDSs:

- PR-IDS (Process Related IDS): detects any abnormal behaviour in the normal operation of the industrial process.

Discovery Tools:

- ASAS (ASset ASsessment): vulnerability assessment tool that scans the network to discover the existing devices and provides hosts and network information (e.g., IP address, Operating System (OS), software version, open ports, and known vulnerabilities).

Correlation tool:

- CAEA (Context Aware Event Analysis): constitutes the core of the correlation engine, aiming at the integration and correlation of all the events/alarms raised by and collected from the tools.

The Preemptive platform's architecture is depicted in Figure 6.2.

6.3 Tools Integration and Evaluation

Several tests have been carried out in the testbed environment Hybrid Environment for Development and Validation (HEDVa), located at the IEC laboratory. It is composed by a combination of virtual machines (VMs), and real field devices. As basis for the validation of the Preemptive platform, we exploited a SCADA/HMI Server, a database (Historian), an Engineering workstation, an attacker machine, a virtual switch, and a physical switch, a firewall and 11 RUTs/PLCs (see Figure 6.3), all connected by a number of Virtual Local Area Networks (VLANs).

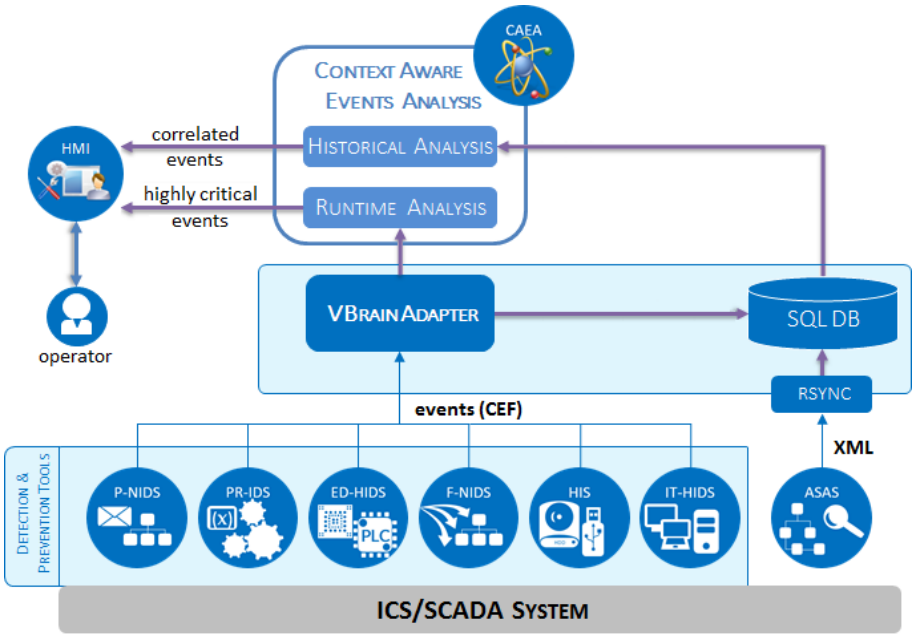

Figure 6.2: General architecture of the Preemptive platform.

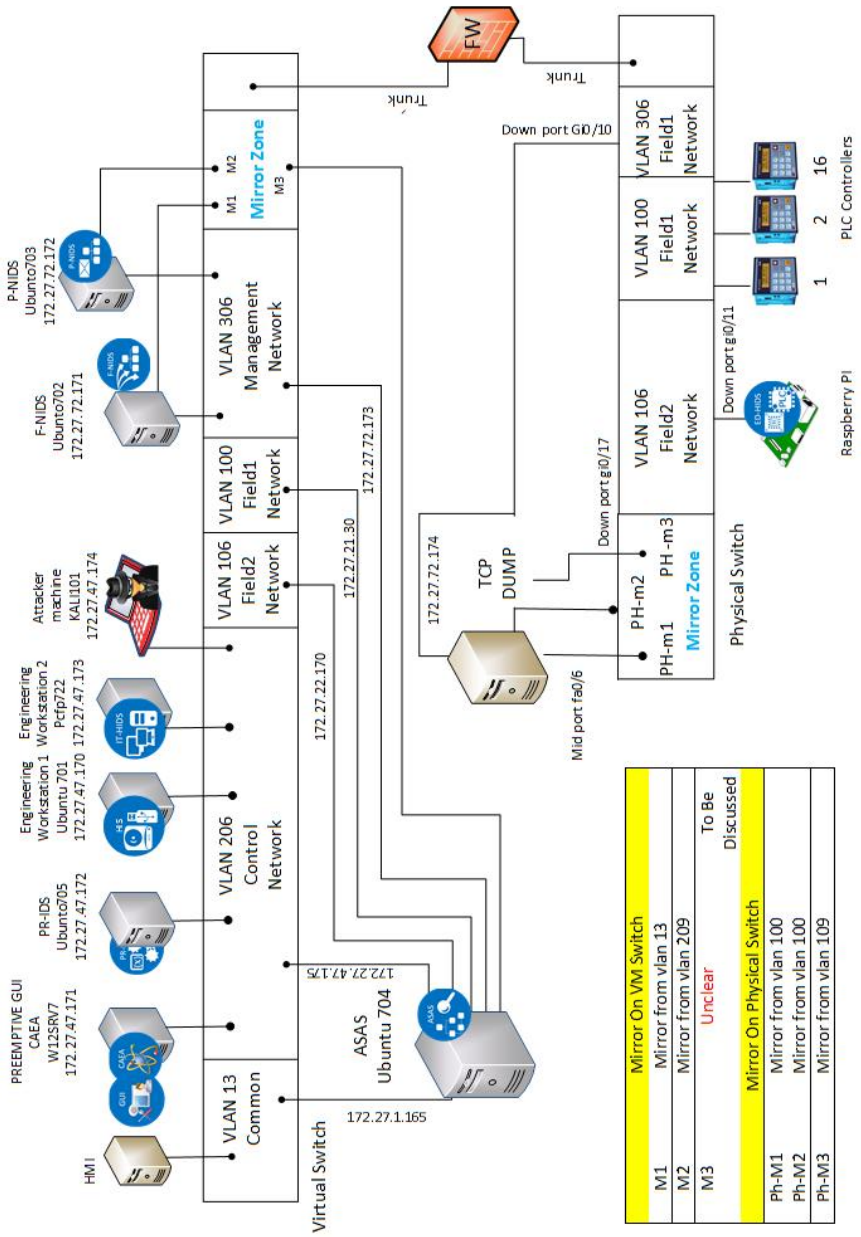

Figure 6.3: The HEDVa testbed and its use for the Preemptive project.

The HEDVa allows the emulation of high, medium and low voltage distribution and transmission grids, implemented on the RTUs using real historical data. For the Preemptive framework validation we chose the medium voltage distribution grid as testbed environment, where the voltage and current values are not actually obtained from real power sources, but are provided by a master hidden PLC that sends precomputed values, related to real consumption data captured in a 24h scenario.

As depicted in Figure 6.3, the Preemptive tools have been implemented in HEDVa as follows:

- IT-HIDS is installed in the SCADA Server as an activity monitoring agent;
- PR-IDS is deployed as a VM and analyses both process data flowing on one of the VLANs and data stored in the Historian by the SCADA Server;
- P-NIDS is on a dedicated VM, analysing traffic to and from the SCADA/HMI Server;
- F-NIDS is on a dedicated VM and acquires data flowing among the control network and the field devices;
- HIS is installed on Servers and the Engineering workstation;
- ED-HIDS is installed on a field device, emulated by a Raspberry-PI;
- ASAS is a VM gathering data from all the VLANs;
- CAEA and the Preemptive HMI are installed on a VM, collecting data from the VLAN and displaying the alarms triggered by the various tools to the operator;
- the attacker machine is connected to different VLANs, so as to be able to launch attacks to the various devices of the testbed.

A wide number of cyber-attacks has been performed, among which MitM for data manipulation, buffer overflow, malicious request to field devices, malware on USB drives and on host devices, and fuzzing attacks. Table 6.1 shows the detection results for each type of attack, demonstrating that these are all detected by at least one Preemptive tool. More specifically, the HIS, IT-HIDS and F-NIDS tools are able to alert when an attack is entering a system before

perpetrating damages to the physical system. The P-NIDS, PR-IDS and ED-HIDS can alert once the attacker has already managed to enter the system and it is aiming at compromising the physical process. Finally, the CAEA enriches the alerts with semantics information and provides to the operator additional details, useful to arrest or mitigate the attack. The result is visualised by the Preemptive HMI, as in the example shown in Figure 6.4.

Attack	P-NIDS	PR-IDS	F-NIDS	IT-HIDS	ED-HIDS	HIS	CAEA
MitM & data manipulation	✓*	✓	✓*				✓*
Buffer overflow	✓	✓			✓		✓
Malware on USB						✓	
Fuzzing			✓				
Malicious request	✓						
Malware on Host				✓			

Table 6.1: Summary of the attacks and detection results. (* The effectiveness depends on the kind of data manipulation.)

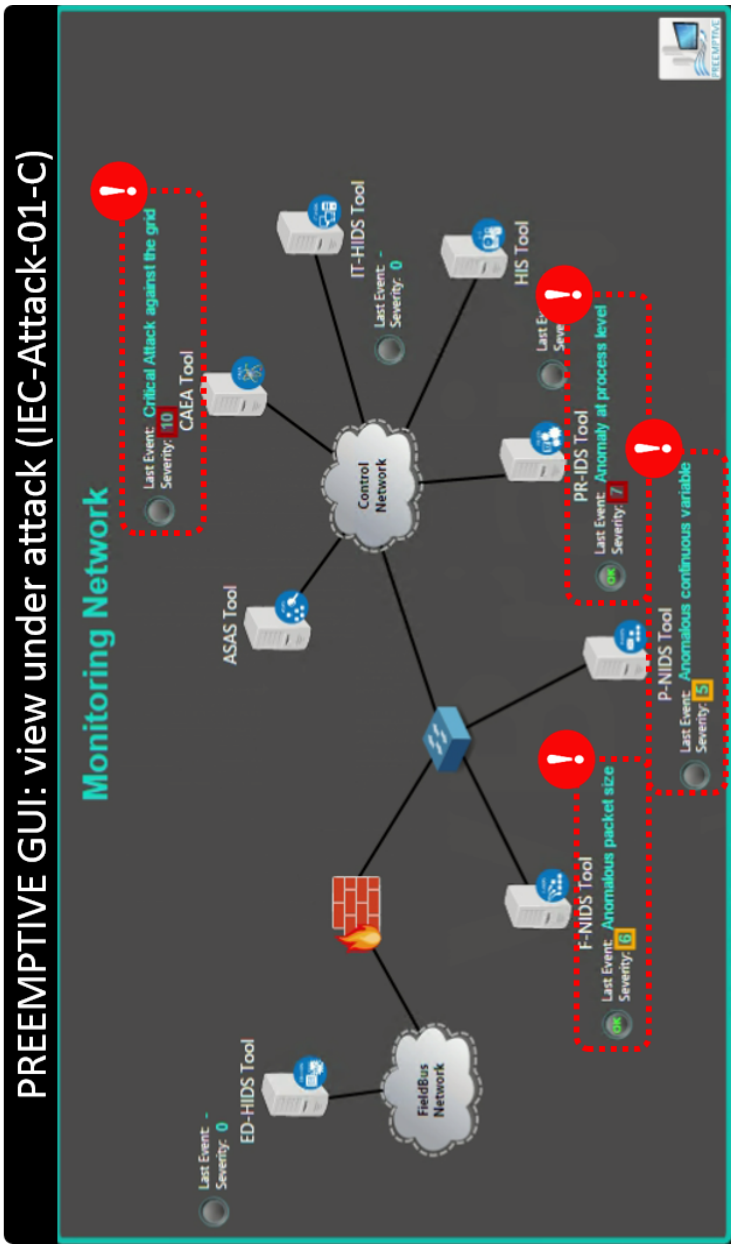

Figure 6.4: Example of detected attacks as shown in the Preemptive HMI.

6.4 Discussion

As the vast majority of security countermeasures, the tools presented in this chapter shows strengths as well as limitations. This section provides some discussions about them.

The Preemptive platform mostly targets detection of attacks, albeit prevention is addressed for USB thumb drives. Most of the effort was put into detecting unknown (zero-day) attacks which might be part of an APT. Hence, the platform is supposed to be used in a context where baseline risks mitigation is performed by adopting other conventional countermeasures.

Each tool provides a unique trade-off between effectiveness in pursuing its specific objectives and certain assumptions on the applicative context but all tools strive to be compatible with the peculiar needs of an industrial context. Several tools (IT-HIDS, P-NIDS, F-NIDS, and PR-IDS) embrace the anomaly detection approach in which deviation from a baseline behaviour is recognised as malicious. Regardless from the specific technique adopted, this approach has the following weak points.

1. The assessment of the baseline (*training* or *learning*) must be performed on a data set which undoubtedly represents correct behaviour.

2. When the system legitimately evolves, changing its behaviour, false positives are detected.

3. A human intervention for re-training is needed when system evolves.

4. The set of behaviours detected as malicious is mostly implicitly described, essentially impossible to manually tune, and the generalisations with respect to the provided baseline depends on the chosen underlying technique.

Considering the extreme stability over time of ICSs, Items 2 and 3 have a small impact. Item 1 requires an appropriate procedure to collect data sets for future use. Regarding Item 4, a possible approach for mitigating it is to associate a rule-based engine to allow for manually-configured exceptions.

The tools that are supposed to detect or prevent intrusions on hosts (IT-HIDS and HIS) have the problem to require the installation of new software, which might me problematic in practice. For ED-HIDS it even requires to change PLC kernel, which make that contribution essentially a proof-of-concept that can inspire some vendor to include that feature in its products. The

ASAS assessment tool, has the obvious limitation that cannot be employed to detect vulnerabilities which are not known to the community of the libraries on which it is based. Concerning the correlation engine (CAEA), the current implementation of the runtime data analysis is rule-based, which implies that configuration should be carried out by an expert. A small help is provided by the historic data analysis, which suggests rules to be adopted in the runtime part. However, at the moment, its accuracy is bound to proper parameter tuning of the underlying Apriori algorithm.

Table 6.2 summarises the tools with their timescale of action. The tools whose action is immediate have no memory and perform their analyse, and possibly detect anomalies, for each sample of data that is provided to them. In those cases, since the time spent in performing the analysis is negligible, timescale is essentially fixed by the technology context in which they act (e.g., network packet transmissions and system calls). Others need to know some history before the analysis and introduces a delay (like F-NIDS and IT-HIDS).

Tool	Level	Action	Approach	Timescale of action
IT-HIDS	host	detection	anomalay detection	In the order of one millisecond.
ED-HIDS	embedded devices	detection	heuristic	Immediate, at the timescale of interrupt handling
HIS	host	prevention	cryptographic	Immediate, at the timescale of system calls.
P-NIDS	network	detection	anomalay detection	Immediate, at the timescale of network protocol communication.
F-NIDS	network	detection	anomalay detection	In the order of a few minutes.
PR-IDS	process	detection	anomalay detection	Immediate, at the timescale of process measures.
ASAS	network and host	vulnerability assessment	passive and active	n/a
CAEA	correlation	detection	rule-based	Depends on rules

Table 6.2: Summary of the tools with some of their features.

6.5 Improvement of Preemptive Framework

Integrating data generated by Preemptive framework tools aim at reducing the false positives and increasing the detection capability. The use of heterogeneous data, i.e. data at industrial level and ICS network level as input provides different points of view of the same issue. Comparing such data potentially enables a detection of effects due an attack which would not detected being focus on only a specific point of the system. In this way it is also possible to compensate the limits due to the detection approach used by single tool. This approach used by Preemptive project is driven by the fact that APT-like attacks target a system masking malicious activities with legitimate traffic and it is needed to use innovative detection and prevention approach to face such malware.

As mentioned in Section 6.2, the initial design of Preemptive framework encompasses the integrity system presented in the Chapter 2. We call regular machines the machines representing the potential source of malwares. We call critical machines the machines to be protected from malwares generated in the regular machines. The integrity system allows a promiscuous use of a USB thumb drive both in the regular machines and critical machines while preserving the security in term of integrity. The integrity system provides an high level of usability. It requires only the installation of a software in the critical machines. The actions performed to guarantee that critical machines do not read tampered data from USB thumb drives are totally transparent to the users. Indeed, such actions do not affect the way the users perform read and write operations from/to the USB data storage. Integrating in Preemptive platform the solution presented in the Chapter 3 it is guaranteed a protection also against malwares BadUSB-like. It does not require any change on the machine to protect. It is an hardware-based solution that can be used with any host equipped with a USB port, hence, comprising embedded systems. It provides protection also during boot, namely, before any operating system is running. The idea behind USBCheckIn is to force users to interact physically with the USB device to use in order to ensure that a real human-interface device is attached.

Using USBCaptchaIn 4 as a part of the Preemptive framework provides a total protection against infections spread by means of USB thumb drives, i.e., both against “traditional” and BadUSB-like malwares. It is an independent hardware and, hence, it can easily deployed in industrial control systems. Indeed, USBCaptchaIn requires only a USB port and does not require any change to the host it is plugged into.

The idea is to integrate the Preemptive framework not only in the new system but also in systems that have been working for a while. Before starting the integration it is needed to design carefully the deployment. It is important to decide in advance the traffic to be captured and the specific tapping points, i.e., the specific points where the traffic is collected.

We argue this is a common approach used by network-based intrusion detection systems both in the IT and OT domain. In this way, it is hard to model the detection process to the specific conditions of the system. The choice of the specific traffic that is monitored strongly impacts the capability to detect advanced attacks. Also if the it is legit to assume the traffic of patter of ICS is mostly identifiable a priori, a static approach could represent a limit against attacks that are tailored to the architecture of the target-system. It is worthy to consider that attacks against ICSs are potentially launched by wealthy organizations, like governments and terrorist group, that can have insiders and, certainly have enough resources to design the attack so that it can bypass unobserved and circumvent common defences.

In Chapter 5 (in the rest call for simplicity also *SDN-monitoring*) we present an architecture and a methodological approach that allow an operator to choose dynamically the traffic to be observed. This solution leverages on software defined network technology that allows to have a centralised managed network configuration. It enables an flexible adoption of one (or more than one) IDS, while having the chance to monitor any traffic forwarding it to the IDS independently from its location.

Such solution enables the adoption of innovative detection techniques. Preemptive is not an exception. Indeed, assuming a ICS equipped with SDN-switches, the deployment of the Preemptive framework, along the SDN-monitoring, is immediate and does not require any specific effort in term of design. The Preemptive tools can be placed everywhere since their location become irrelevant in term of detection capability. Furthermore, adopting Preemptive platform along the SDN-monitoring enables additional benefit in term of monitoring capability. The possibility to change dynamically the monitored traffic could help in reducing the false positives.

If the events sent to the VBrain Adapter suggest the eventuality of a malicious activities ongoing in the network, it is possible, in a reasonable time, to change the traffic observation points and/or add new ones. Changing the tapping points provides a new point of view on the system. Adding new tapping points provides to the Preemptive Context Aware Events Analysis to process more data. It means the integration process and, hence, the whole detection process can be more accurate. At the end, if the Preemptive framework still

detects malicious activities, it is likely the system is real under attack.

Chapter 7

A Scalable Way to Use Authenticated Data Structures in the Cloud for Industrial Control Systems

Public cloud infrastructures are popular since they enable virtually unlimited scaling paying only the amount of needed resources, on-demand. However, the public cloud model, inherently implies that the user delegates to the cloud provider the management of the infrastructure, and, with it, many security guarantees that were clearly under her/his control with in-house solutions. Most users are concerned with data confidentiality, for which there are a large number of effective cryptography-based tools that can be used. These tools allow the user to keep control over data confidentiality while keeping the advantages of the public cloud. The same objective is still difficult to meet for "fully fledged" data integrity. For *data integrity* we intend the capability to detect (malicious or accidental) changes of user data that do not conform to the will of the user. Usually this capability is provided by (a trusted part of) the same system that manages user data. If there is only one user, just a sequence of updates is enough to specify the intention of the user. However, when many users concurrently update the data, additionally *consistency rules* should be specified (see Sections 7.3 and 7.3).

Integrity is usually taken for granted by users, but a cloud provider might change user's data either by mistake or maliciously. For example, files or

records might be changed, got lost, reverted to a previous version, or deleted files or records might be restored. Depending on the kind of application, injecting an outdated version of the data into a business process might lead to huge loss or damage. Typical sectors in which integrity is paramount are banking, financial, health, legal, defence, industrial control systems and critical infrastructures. For this reason, practical means to check the integrity of data returned by a public cloud before use (i.e., *on-line*) is highly desirable in these contexts. Further, both the user and the cloud provider might want to have a cryptographic proof of the genuineness or of the inaccuracy of the data to use for dispute resolution. Note that, for deletion, restoration, or reversion to previous version, signing each file or record individually does not help in detecting the tampering, while signing the whole dataset with conventional techniques is effective but highly impractical for most applications. To solve this, *authenticated data structures* (*ADS*) [Tam03] can be adopted.

Using an ADS, an untrusted entity can store a dataset and can answer queries providing a proof of the validity of the answer to the client. Essentially, an ADS is a mean to keep a cryptographic hash of the dataset that enable efficient update of the hash upon small changes and efficient integrity checks of small parts of the dataset against a trusted version of that hash. Traditionally, ADSs are adopted in a model where a single *source* asks the untrusted entity (*responder*) to update the dataset and a plurality of *users* can perform authenticated queries to the responder. Users can validate the queries results against a cryptographic hash obtained from the source by a trusted channel. In another traditional model, a single client performs updates and queries to the untrusted storage.

A large body of research work deals with integrity of outsourced database with many different approaches that may favour security, efficiency, flexibility of the queries, etc. However, an on-line integrity verification system for a public cloud service need to fulfil very strict requirements to avoid impairing the advantages of the cloud adoption. In particular, the solution should scale with respect to the amount of data, updates, and clients, with the same approach the typical cloud storage solutions do. For this reason, the above mentioned idea of keeping up-to-date a cryptographic hash for a large dataset has been regarded by many authors as impractical (for example, see [AEK+17, PZM09]).

In this chapter, we address the problem of adopting ADSs while maintaining the possibility to achieve high throughput keeping limited latency. We define *throughput* as the maximum rate of updates per second the system can process. We define *latency* (or *response time*) as the time elapsed form the update reception to the instant when the server is able to serve a read that includes that

update. We use ADSs in a model that is different than the traditional ones. In our setting, a possibly large number of clients performs both authenticated updates and queries to a single untrusted storage. This model is quite challenging. Any change in the dataset is reflected in a change of the single hash of the whole dataset. This turns out to be a bottleneck, since for each change, the server should contact a trusted party for a signature. In existing proposals for the same setting, updates are usually applied in batches. However, processing of each batch starts after the completion of the previous one, since the signature of the previous hash is needed (see, for example, [BCK17]). This is critical if the network latency, between the server and the trusted party, is non-negligible. Additionally, clients may expect that deviations of the behaviour of the server from certain consistency rules to be detected. For example, the system should detect answers from the server that are not consistent with its past answers.

Our main contribution is a protocol, called *pipeline-integrity protocol*, that allows a server to ask (possibly far) trusted parties to authenticate the hash of a new version of a continuously updated dataset without hindering scalability of the whole system. We address the scalability problem by allowing the server to start independent authentication processes that can proceed pipelined. In this way, both network and trusted resources are shared among several concurrent authentications, achieving much higher resource usage. We also introduce a new concept of consistency in a security setting called *quasi-fork-linearisability*, which is compatible with our pipelining approach and is only slightly weaker than *fork-linearisability* [CSS07]. Fork-linearisability is a form of strong consistency in which the server is allowed to partition clients so that a strongly consistent history (linearisable) is shown to each partition. This form of consistency is proven to be the best possible in a setting where clients cannot directly communicate.

We theoretically prove that the pipeline-integrity protocol allows us to achieve high throughput with practically bounded latencies, provides quasi-fork-linearisability, and enables clients to detect Byzantine servers that deviates from the required behaviour. We analytically and experimentally compare latency and throughput of the pipeline-integrity protocol against the traditional approach, which performs only one authentication at a time.

The rest of the chapter is structured as follows. In Section 7.1, we review the state of the art. Section 7.2 is dedicated to background on authenticated data structures and their use. Section 7.3 introduces models and definitions that will be used in the rest of the chapter. In Section 7.4, we analyse the performances of the typical interaction of ADS-based client-server protocols. In Section 7.5,

we provide a brief and intuitive overview of the protocols described in the next three sections.

In Section 7.6, we describe a simplified version of the pipelined-integrity protocol, which has weak consistency properties but clearly show the main ideas we propose to achieve scalability. In Section 7.7, we show a protocol, and a data structure, to achieve quasi-fork-linearisability that has poor scalability properties but turns out to be compatible with our pipeline-integrity technique. Section 7.8 describes the complete version of our pipelined-integrity protocol, which unifies the results of the two previous sections and provide both scalability and quasi-fork-linearisability. In Section 7.9, we discuss how to cope with real (non ideal) communication and computation resources. Section 7.10 provides experimental evidence of the feasibility and scalability of our approach.

7.1 State of the Art

A large wealth of research works has dealt with the problem to verify the correct behaviour of an untrusted storage and many of them explicitly refer to a cloud computing setting. A good survey of the research in this area can be found in [ZKM+17].

The effectiveness of each approach can be evaluated with respect to several aspects. Some of the coordinates that are relevant for this chapter are: (1) the presence of a trusted entity in the cloud or if only clients are trusted, (2) the number of clients supported, (3) the load of each client in terms of data stored and computation performed, (4) the efficiency of the client-server protocol, (5) the probability with which an anomalous behaviour of the server is detected, (6) the ability to deal with an unbounded number of queries, and (7) the support for efficient updates and the consistency model supported.

A *proof of retrievability* [JK07, BJO09, SW13] is a compact proof by a filesystem (prover) to a client (verifier) that a target file is actually stored. The *proof of data possession* [ABC+07] adds the possibility of data recovery. A typical limitation of these schemes is that they can only be applied to a limited number of requests, decided upfront. Also, they usually do not support efficient update. Some works, such as [FLY+17, LTC+15, ADPMT08], describe protocols that, up to a certain extent, admit the dynamic update of stored data. In [WWL+09], the proof of retrievability approach is enhanced so that updates are efficiently supported and a third party auditor can perform the verification.

Many of the works in this area adopt *Authenticated Data Structures (ADS)*

[Tam03], especially when dynamic data operations are required. ADSs have many advantages: they provide deterministic verification, support dynamic operations and require the clients to keep only a constant amount of data: a digest of the whole dataset. This approach has also the advantage to detect attacks like deletion or reversion to a previous (authentic) version of part of the data, which require to consider the dataset as a whole. ADSs were successfully adopted in many works concerning integrity of outsourced Databases (see, for example, [LHKR06, ZKP15, SP08, YPPK09, PPP10b]). A typical problem tackled in these works is to support a broad class of queries, efficiently. In research about verifiable databases a randomized periodic verification process was proposed (see, for example, [EBM+17, ZAH+13]).

When using ADSs, the single digest must be updated and propagated to all clients at each update in a secure way. In fact, the approaches based on ADSs treat a dataset as a single object: if even only one bit is updated the whole dataset is considered updated. Some works explicitly rule out ADSs on the basis of their inefficiency in the client-server setting when high concurrency is needed [AEK+17, PZM09]. Proposing an efficient way of using ADSs in this setting is exactly the problem addressed by this chapter. The work [AEK+17] proposes a system based on deferred verification which requires a trusted entity in the cloud (e.g., a special processors like Intel SGX Enclave [MAB+13]) which, however, works only in-memory. The work [PZM09] proposes a dynamic solution based on signature aggregation [LHKR08], which are much more expensive than ADSs, in terms of cryptographic computation.

Whenever more clients can concurrently perform updates, a consistency problem arises. Consistency has a long-standing research history, which was developed mainly in the areas of databases and multiprocessors architectures with shared memory (see, for example, [WV01, B+14, VV16, HW90]). Many papers address the problem of verifying the correct behaviour of an untrusted storage service in the context of concurrent accesses, with the focus of providing provable guarantees about consistency. A strong notion of consistency is embodied in the definition of *linearisability* [HW90], which essentially states two things.

1. The outcome of the operations on a shared object have to be consistent with a sequence of operations H (a *history*) that conforms with the sequence of operations as invoked by each client. It should be noted that each client can only perform operations sequentially so, from the point of view of the client, they are totally ordered.

2. If two operations are not concurrently invoked (by distinct clients), they must appear in that order in H.

The history H is "chosen" by the server or, more often, is the outcome of unpredictable network latencies. The server might maliciously show different values for H to different clients. This violates the obvious notion of integrity in a shared environment. Ideally, we would like to impose that all clients see the same H. In [MS02], it is shown that this is impossible to achieve in a setting in which clients cannot directly communicate or are not synchronised. Hence, the authors introduce a weaker form of consistency called *fork-consistency*. This was lately renamed as *fork-linearisability* in [CSS07]. Fork-linearisability admits that the clients observe an execution that may split into multiple linearisable "forks", which must never join. In other words, the union of the H shown to the clients must be a tree where, at a fork, the set of operations of the branches are pairwise disjoint. The security aspect in this definition is bound to the capability of the client to detect the fork as soon as the server tries to merge two forks, i.e., to propose to a client updates that were kept hidden to that client till that time. A system realising fork-linearisability is shown in [LKMS04] proposing a quite inefficient protocol. In [WSS09, FZFF10] ways to enforce fork-linearisability are proposed in a setting where the whole storage is replicate on each client. The research described in [LKMS04, CSS07, CKS11] allows to store the data on an (untrusted) server and use vector clocks to give to the clients a partial view of all operations executed on the data. It can be proven [CSS07] that to ensure fork-linearisability a blocking condition is unavoidable. Namely, it is impossible to avoid situations in which a client must wait another client to perform some actions. The results in [WSS09, CO14, BCK17] show protocols that allow certain classes of operations to proceed without waiting. VICOS [BCK17] is probably the work that is more akin to this chapter, regarding targeted problems. It shows a protocol that allows several clients to share an ADS preserving fork-linearisability. This work does not address the problem of the throughput and put considerable burden on the clients, since each client have to process all updates on the data, even if they are performed by other clients.

The Depot storage system [MSL+11] provides *fork-causal* consistency, which is weaker than fork-linearisability but enables to join forked histories and to cope with eventual consistency. The Depot approach is compatible with typical availability and scalability requirements of the cloud but its form of consistency is harder to handle for applications.

In [CO14], a survey of works providing integrity in the dynamic client-server setting with different consistency guarantees is provided.

7.2 Background

In this section, we recall basic concepts, terminology and properties about *authenticated data structures* (*ADS*), limiting the matter to what is strictly needed to understand the rest of this chapter. Further details can be found in [Tam03, MND+04].

For this chapter, an ADS is a container of implicitly ordered key-value pairs, denoted $\langle k, v \rangle$, which are also called *elements*. The content of the ADS at a given instant of time is its state. The ADS deterministically provides a constant-size digest of the set of the key-value pairs of its content with the same properties of a cryptographic hash of that set. We call it *root-hash*, denoted by r. If any element of the set changes, r changes. It is hard to find a set of elements whose root-hash is a value given in advance. An ADS provides two operations, the authenticated query of a key k and the authenticated update of a key k with a new value v'. A query returns the value v and a proof of the result with respect to the current value of r. If a trusted entity safely stores the current r, it can query the ADS and execute a check of the proof against its trusted version of r to verify that the query result matches what expected. The update operation on k changes v associated with k into a provided v' and changes r in r', as well. The interesting aspect is that a trusted entity that intends to update k can autonomously compute r' starting from the proof of $\langle k, v \rangle$ that can be obtained by a query.

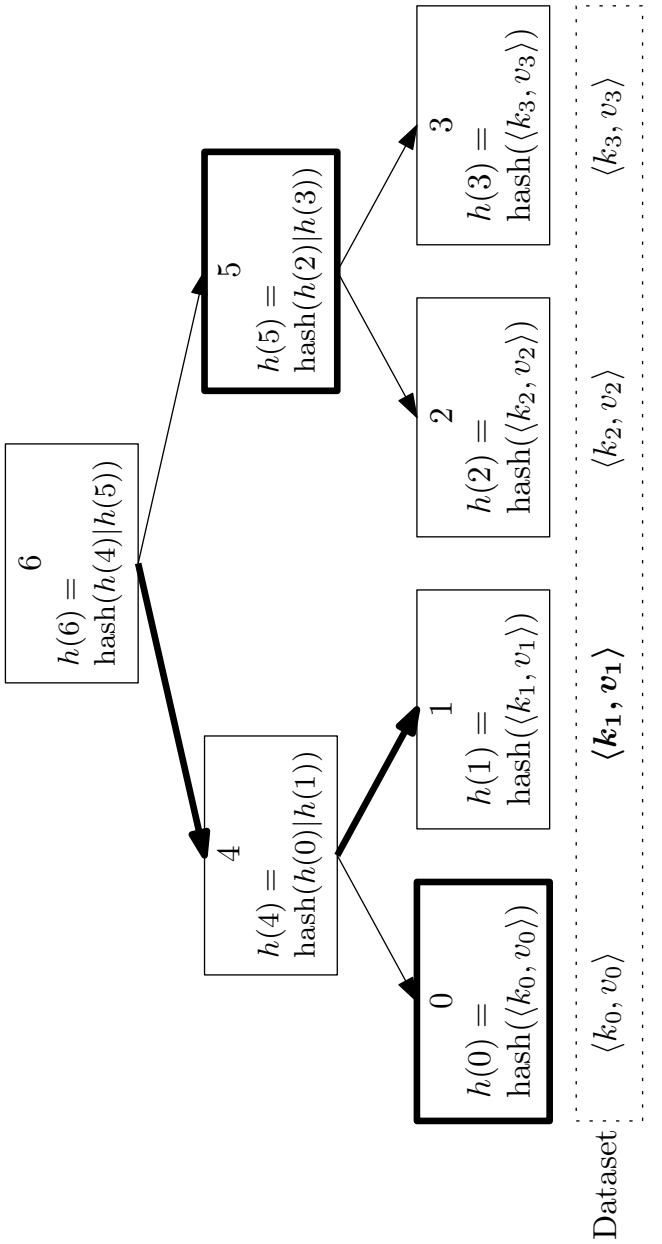

Figure 7.1: An example of Merkle Hash Tree with four leaves and a binary structure. We evidenced the elements regarding the proof of $\langle k_1, v_1 \rangle$.

As an example, we briefly introduce a specific ADS, the *Merkle Hash Tree* (MHT), however, the same properties hold for others ADSs, like, for example, the *authenticated skip list* [GTS01]. A MHT is a binary tree T that is composed of internal nodes and leaves, see Figure 7.1. Every leaf is associated with a key-value pair $\langle k, v\rangle$. The tree is managed as a binary search tree. Let hash$(\cdot)$ be a cryptographic hash function. Every node n is labelled by a cryptographic hash $h(n)$. If n is a leaf, we define $h(n) = \text{hash}(\langle k, v\rangle)$. If n is an internal node, with n' and n'' its children, $h(n) = \text{hash}(h(n')|h(n''))$. If n is the root of T, $r = h(n)$ is the root-hash of T. Let k be a key in T, and let l be its associated leaf. Consider the path $(n_1, n_2, \ldots, n_m)$, from $l = n_1$ to n_m, where n_m is the root of T. For each n_i $(i = 1, \ldots, m-1)$, let $\bar{n}_i$ be the sibling of n_i. The *proof* for $\langle k, v\rangle$ according to T, denoted proof$(T, \langle k, v\rangle)$ possibly omitting T and/or v for short, is the sequence $(h(\bar{n}_1), d_1, h(\bar{n}_2), d_2, \ldots, h(\bar{n}_{m-1}), d_{m-1})$, where $d_i \in \{L, R\}$ indicates if n_i is the left or the right child of n_{i+1}. For example, according to Figure 7.1, proof(T, k_1) is the sequence $(h(0), R, h(5), L)$.

It is easy to see that, given proof(T, k) and $\langle k, v\rangle$, it is possible to compute r and that creating a different proof that gives the same r implies braking hash$(\cdot)$. Also, considering the update of $\langle k, v\rangle$ into $\langle k, v'\rangle$, the new root-hash r' can be easily computed from proof(T, k) just pretending that $\langle k, v'\rangle$ is the value of l. The proof that a key k is not present in T can be given by providing the proof(T, k_1) and proof(T, k_2), where k_1 and k_2 are two consecutive keys such that $k_1 < k < k_2$. After the authenticity of proof(T, k_1) and proof(T, k_2) is verified, the proof that k_1 and k_2 are consecutive can be obtained by checking that the sequences of d_i matches regular expressions R^*Lz for k_1 and L^*Rz for k_2, where z is a possibly empty common suffix. We do not go into the details of the addition and deletion of a key. We just note that incomplete binary trees can be allowed by minimal changes in the above definitions. Changing the structure of the tree, even without changing the dataset, changes the root-hash, so the tree structure is part of the state of the MHT, as well. This contradicts the hypothesis of the deterministic link between the root-hash and set of elements contained in the ADS. This is essentially a technical problem that can be solved, for example, by defining a canonical structure of the tree that deterministically depends on the contained elements and caring that every operation leaves the tree structure in the canonical state. Clearly, the trusted entity that is going to compute r' should get all needed information to re-create locally the correct path(s) from the leaves involved in the update to the root, exactly as the ADS is supposed to do. Another approach is to resort to associative cryptographic hash functions [TZ94] so that root-hash is independent from the way leaves are grouped.

When we have a large set of elements stored in an ADS, but we only need authentication for a small number of them, known in advance, we can resort to the *pruning technique*. Pruning reduces the storage size of the tree, without changing the root-hash, by removing sections of the tree that are no longer needed for the expected queries. The basic idea is very simple. Whenever a subtree have only unneeded leaves, we can remove all the subtree maintaining only its root. Pruning an ADS reduces the required space, preserves the root-hash, preserves the capability of producing proofs for the needed keys, and keeps security intact. Pruning is obvious for a MHT but also other ADSs may support it.

A typical example of use of an ADS is for outsourcing a key-value store in a single client setting, keeping in the client only the root-hash r while keeping the ADS in an untrusted server. In this setting the query and update operations are as follows.

Query(k). Server returns v and $p = \text{proof}(\langle k, v\rangle)$. The client verifies the consistency of v with p and the local copy of r.

Update(k, v'). The client preventively performs Query(k) getting v and $p = \text{proof}(\langle k, v\rangle)$, and checks p against the local copy of r. Then, the client pretends the stored value to be $\langle k, v'\rangle$ and compute all values along the path of p accordingly. It comes up with a new value r' for the root-hash, which is considered the current root-hash for the next operation. Then, the client send the operation Update(k, v') to the server and forget anything else but r'. When the server receives Update(k, v'), it update the ADS accordingly recomputing all the hashes all the way up to the root. Its current root-hash should turn out to be exactly the r' computed by the client.

We say that a root-hash r *contains* an update u if u is part of the sequence of updates that was applied to the dataset before reaching the state corresponding to r.

7.3 Models and Terminology

In this section, we provide basic definitions, assumptions and models we use throughout this chapter. First, we introduce general assumptions and our definition of scalability. Then, we formally introduce a model of the service we intend to support assuming correct behaviour of all actors. Finally, we

formally define the consistency model that will be supported by our approach in the case of a Byzantine server.

General Setting and Assumptions

The results of this chapter are stated in the setting in which there are a (possibly large) number of mutually trusted clients, with limited storage that need to store and share an arbitrarily large amount of data. They do that by relying on an untrusted *server*. Certain special clients are in charge of authenticating operations invoked by regular clients. They do not invoke operations themselves. They are called *authenticators* and we reserve the word *clients* for regular clients. We collectively refer to clients and authenticators as *trusted entities*. In practice, if deemed convenient, one machine can play both roles. However, in this chapter, we always deal with them as if they were separate entities.

Trusted entities can only communicate with the server and are not synchronised. For simplicity, we assume all network communications are reliable and timings predictable. In other words, we assume that no message is lost, no network congestion occurs, and the network behaves deterministically and consistently over time. Since for real systems this assumption does not hold in general, we discuss the issues arising when network and clients are not reliable and timings not predictable in Section 7.9. In that section, we also show how to deal with those issues.

Each trusted entity e can sign data d, by appending it with an asymmetric encryption of the hash. The signature is denoted by $[d]_e$. We also write $[d]$ when e is not relevant. We assume that each trusted entity has certificates of all other trusted entities and hence can securely verify all signatures.

Scalability

For the purpose of this work, when we say that a service *scales*, we intend that it is possible to increase volume of operations, data size, and number of clients (by increasing hardware resources dedicated to the server or network bandwidth) while keeping the response time bounded. As we will see in the following, when ADSs are adopted, the client-server protocol plays a fundamental role in the scalability of the whole system. In particular, for all protocols described in this chapter, part of the processing must be performed client-side or generally by a trusted entity. The usual approach blocks the server while waiting a reply from the trusted entity and shows very bad resource usage. In Section 7.4, we

formally analyse a client-server protocol adopting this approach and we show that its response time very badly depends on the throughput. In Section 7.8, we propose a protocol that does not have this problem while keeping a strong notion of security.

Key-Value Stores and Consistency

We focus on key-value stores, where we assume keys and values have limited size. This assumption simplifies the description and the analysis of protocols and algorithms shown in this chapter. In fact, under this assumption, the time taken to process or transmit each operation is bounded by a constant. This help us in focusing the chapter on the interesting aspects of our solution. In the rest of the chapter, for simplicity, we assume the store offers only two kinds of operations: (1) *read*, which gets the value v currently associated with a key k and (2) *update* of a key k with a value v, which creates k if it does not exist or delete it if $v = \perp$. These basic functionalities are the core of the features provided by several commercial services and open-source projects of the NoSQL landscape, see for example [DHJ+07, Apa18, SN16].

Each operation begins with its *invocation* at the client and terminates when its *response* reaches the client. Invocation and response occurs at certain instant of time and are called *events*. A sequence of events is *complete* if each invoke event is matched by one, and only one, following response event in the sequence, for the same operation, and viceversa. In the following, we mostly deal with complete sequences and omit to state it explicitly. Each operation spans an *interval* of time between the sending of its invocation and the receiving of its response. Two operations are *concurrent* if their intervals overlap otherwise one of the two *precedes* the other and they are *sequential*. A complete sequence of events is *sequential* if all operations in it are pairwise sequential. In other words, in a sequential sequence of events, for each operation, its invocation event is followed by its response event with no other event in between. A sequential sequence also implies a total order on the operations of that sequence.

Often, we consider a sequential permutation π of a complete sequence of events σ. This is essentially a way to represent a choice of an order of the operations cited in σ. For example, let $\sigma = i_1 r_1 i_4 i_2 i_3 r_2 r_3 r_4$. Where i_x and r_x denote invocation and response events of operation x. A possible sequential permutation of σ is $\pi_1 = i_1 r_1 i_4 r_4 i_2 r_2 i_3 r_3$, expressing the order of operations 1 4 2 3. Another possible sequential permutation of σ is $\pi_2 = i_3 r_3 i_1 r_1 i_2 r_2 i_4 r_4$, expressing the order of operations 3 1 2 4. We note that π_1 respects the

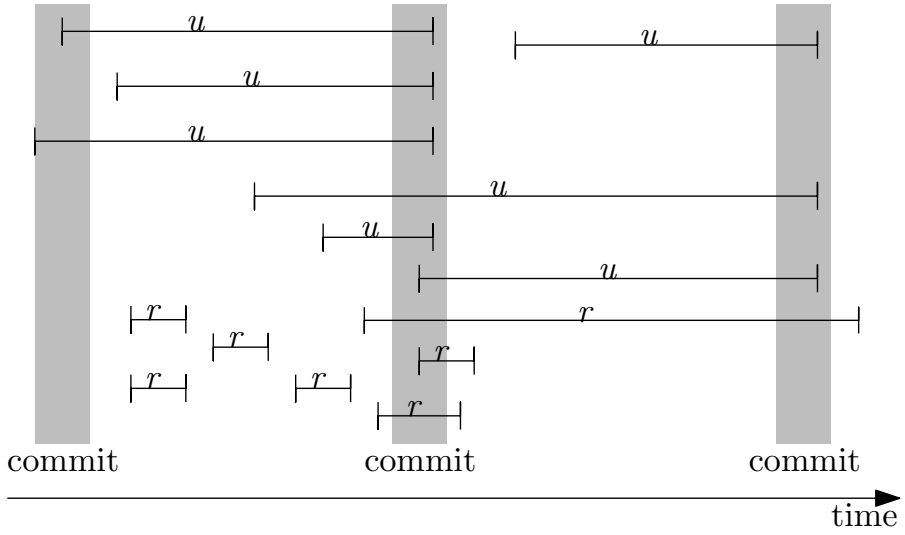

Figure 7.2: Relationships between operations and commit phases. Operations are represented by horizontal bars, where invocation is received by the server at the left extreme of the bar while the operation is considered concluded by the server at the right extreme of the bar. Each update is labelled u and always ends during its associated commit. Each read is labelled with r, cannot end during a commit, and it is associated with its preceding commit.

precedence of operations implied by σ while π_2 does not (operations 1 and 3 are reversed).

Consistency is the property of a distributed system to behave according to the expected semantics of the operations as in a sequential setting, at least up to a certain extent. Typically, consistency guarantees are formalized in a setting in which operations are partitioned in *sessions*, where the operations of each session are sequential and hence fully ordered in time. Sessions are supposed to be associated with a client, which expects to see the sequential behaviour of the operations if no other client interferes. In our model, the interactions between clients and the server, deviates a bit from this approach. We now formally describe this interaction. Since, in our setting, consistency is tightly linked with security, the formal definition of our consistency model is provided in Section 7.3.

We allow each client to invoke operations concurrently. We force update

operations to be executed only during *commit* phases, which are periodically triggered. Updates are applied respecting the invocation order of each client, but updates invoked by distinct clients can be arbitrarily interlaced by the server. Reads can be executed at any time but not during a commit. They return values according to the state of the key-value store as updated by the preceding commit. Figure 7.2 pictorially shows an example of how read and update operations can evolve over time. At the end of a commit, for each executed update, a corresponding response is sent to the invoking client. While for a practical implementation this response is optional, in our model we always consider it.

More formally, we denote by σ a real-time ordered sequence of events. The invocation event of operation o is denoted by $\text{inv}(o)$, its response event by $\text{res}(o)$. Operations in σ are possibly concurrent. We associate with events a *server-time* that, for invocation, is the arrival time at the server and, for responses, is the sending time from the server. For an operation o, its server-time interval is denoted $I_o = (t_{\text{inv}(o)}, t_{\text{res}(o)})$.

A commit is an atomic procedure executed on the server in a time interval $(t_{\text{begin}}, t_{\text{end}})$ during which no other operation can change the state of the key-value store. For brevity, we may treat commits as intervals of time to simplify notation. Commits do not overlap. An update u is *associated with* commit χ, if $t_{\text{res}(u)}$ is in χ and u is executed in the context of χ. Each update is associated with one and only one commit. The only way to change the content of the key-value store is to commit updates. A read operation r *is associated* with a commit χ if χ is the last commit before $t_{\text{res}(r)}$. If r is executed before all commits, it is associated with no commit, and it is called *initial*. Each non initial read is associated with one and only one commit. A read operation returns a result on the basis of the state of the key-value store after the associated commit or on the basis of the initial state of the store for initial read operations.

Consider a sequence σ of events. The server executes read and update operations according to a certain sequential permutation π and is supposed to apply updates only during commits. One may ask if π is consistent with the commits. The following definition formally describes this.

Definition 1 (Commit-correctness)**.** *A sequential permutation π of a complete sequence of events σ is* commit-correct *with respect to the sequence of commits $\chi_1, \ldots, \chi_n$ if*

$$\pi = \rho_0\omega_1\rho_1 \ldots \omega_i\rho_i\omega_{i+1}\rho_{i+1} \ldots \omega_n\rho_n$$

where

1. *ρ_0 is a sequential permutation of all and only the events of the initial read operations in σ,*

2. *ρ_j, with $j = 1, \ldots, n$, is an arbitrary sequential permutation (of events) of read operations associated with χ_j, and*

3. *ω_j, with $j = 1, \ldots, n$, is an arbitrary sequential permutation (of events) of update operations associated with χ_j that conforms to the invocation order of each client.*

Two operations *commute* if they provide the same results and state changes independently on the order they are executed. In our case, any two read operations associated with the same commit always commute. In all other cases, this property depends on the keys involved and in general may not commute. This definition can be naturally extended to a set of operations. Reordering read operations associated with different commits is forbidden in our setting, so it does not make sense to ask if they commute.

In the following, we introduce consistency (see Definitions 2 and 5), where a role is played by preservation of real-time order of events when permuting them. The following lemma states the relation between commit-correctness and preservation of the real-time order.

Lemma 1. *Given a complete sequence of events σ and a sequence of commits $\chi_1, \ldots, \chi_n$ such that all and only update operations end during a commit, let π be one sequential permutations of σ. If π is* commit-correct *with respect to $\chi_1, \ldots, \chi_n$, it preserves the real-time order of all non commuting operations of σ.*

Proof. We prove the statement by induction on the number n of commits. In the base case, $n = 0$ and $\pi = \rho_0$ which only contains read operations. Since all operations in π commute the statement is trivially true. Now, we prove the inductive case. Suppose the statement is true for $\pi' = \rho_0\omega_1\rho_1\omega_2\rho_2 \ldots \omega_{n-1}\rho_{n-1}$,we prove the statement is true for $\pi = \pi'\omega_n\rho_n$.

Consider a non commuting pair of operations. If they are both in π', they are in real-time order by the inductive hypothesis. Now, we prove that any $o \in \omega_n\rho_n$ is in real-time order with any distinct o', if they do not commute (i.e., if they are not both read operations).

If o is a read operation then $o \in \rho_n$. Operation o can not occur in σ before an operation $o' \in \pi'\omega_n$ because o is associated with commit χ_n and must end after χ_n. Hence, if o' is an update u associated with χ_j, $t_{\mathrm{res}(u)} \in \chi_j \leq \chi_n < t_{\mathrm{res}(o)}$

with $j \leq n$ and, if o' is a read r associated with χ_j, $\chi_j < t_{\text{res}(r)} < \chi_{j+1} \leq \chi_n < t_{\text{res}(o)}$ with $j < n$. This means that o and o' are either correctly ordered or concurrent. Clearly, if o' is in ρ_n, too, they commute and the statement does not apply.

If o is an update operation then $o \in \omega_n$. Following the same reasoning as above, if o' is an update u associated with χ_j, $t_{\text{res}(u)} \in \chi_j \leq \chi_n \ni t_{\text{res}(o)}$ with $j < n$. If $j = n$, u and o are concurrent, hence, they are not real-time ordered. If o' is a read r associated with χ_j, $\chi_j < t_{\text{res}(r)} < \chi_{j+1} \leq \chi_n \ni t_{\text{res}(o)}$ with $j < n$. Again, this means that o and o' are either correctly ordered or concurrent. □

Threat Model and Consistency

Clients rely on an untrusted service to store their data. We suppose the server is operated or hosted by a cloud provider, which may change the data stored in it, deliberately or by mistake. In this chapter, we assume that all trusted entities (hence all clients) trust each others and the only possibly malicious actor is the server. A fundamental requirement of our approach is that it should allow the clients to recognize any data tampering, right after the reception of the data. We also mandate that this should be done with high probability, so that it can be considered deterministic for any practical purpose (like many cryptographic hash properties are). The attacker can either be the cloud operator itself or be a third party that compromises the server forcing it to behave maliciously. From our point of view, both situations are attacks that we aim to detect and we do not distinguish them in the rest of the chapter.

To define clearly our threat model, i.e., to distinguish between honest and malicious behaviour, we formally define our consistency model. We first introduce some basic definitions. We consider a set of clients, denoted by C, that ask the server to perform possibly concurrent operations (read or update). Consider the invoke and response events corresponding to these operations. Events occurring in the system are totally ordered in a sequence σ, according to their (invocation sending or response reception) real-time instant at the client. A sub-sequence of σ is an ordered subset of σ whose order conform to that of σ.

We can consider a subsequence σ_i of σ, for each client $c_i \in C$, so that (at least) all completed operations occurring at c_i are in σ_i ([1]). It is also useful to

[1] Actually, here and in the following definitions of fork-linearisability and quasi-fork-linearisability we might restrict σ_i to contain *only* completed operations occurring at c_i. This change would not affect the following theory. However, we decided to avoid unneeded changes, with respect to definitions that can be found in literature, in order to ease the

consider a sequential permutation π_i of σ_i, which is essentially a sequence of operations, expressing the order in which the effect of those operations should be considered when executed according to their sequential semantics specification. A specific kind of consistency is defined in terms of the existence of σ_i and π_i satisfying certain conditions. The following is the traditional definition of the fork-linearisability consistency adapted from [CSS07], for our definition of key-value store.

Definition 2 (Fork-Linearisability)**.** *A sequence of events σ is* fork-linearisable *with respect to the semantics of a key-value store, if and only if, for each client c_i, there exists a complete subsequence σ_i of σ and a sequential permutation π_i of σ_i such that*

1. *all completed operations of σ occurring at client c_i are in σ_i,*
2. *π_i preserves the real-time order of σ,*
3. *the operations of π_i satisfy the semantics of their sequential specification, and*
4. *for every $o \in \pi_i \cap \pi_j$, the sequence of the events that precede o in π_i is the same as the sequence of the events that precede o in π_j .*

Definition 2 should be intended in *monotonic sense*. That is, consider the instants in which clients receive operation responses, denoted by $t_1, t_2, \ldots$. Any consistency definition should hold for the sequences σ_i^j and π_i^j seen by each client c_i at each instant t_j. Clearly, we expect from a system a consistent *monotonic* behaviour in the sense that, σ_i^j and π_i^j should be prefix of σ_i^{j+1} and π_i^{j+1} respectively. This aspect is largely left implicit in previous literature, however, in the following we provide definitions that explicitly take it into account.

Condition 2 of Definition 2 makes sense for the general case tackled by [CSS07] (a generic functionality) but is unnecessarily restricting in our case (a key-value store). Consider two read operations r_1, r_2 appearing in this real-time order in σ. Suppose no update operation is between r_1 and r_2 or is concurrent to them in σ. Clearly, preserving their real-time order is irrelevant. In general, it is important to preserve the real-time order only for operations that do not *commute*. For this reason, our consistency definition, provided in the following, relaxes that condition.

---

comparison of results.

Note that, Definition 2 does not refer to the fact that the operations invoked by each client should be sequential, hence, it applies also to our setting that do not force each client to invoke operations sequentially (see Section 7.3).

Condition 4 of Definition 2 embodies the possibility that, at a certain instant, the server can partition clients showing to two distinct partitions different histories (π_i) that "fork" starting from a certain common event. Fork-linearisability is a fork-allowing variant of the definition of *linearisability* [HW90], which is considered a strong form of consistency in the literature that assume the server is not Byzantine. However, authors of [Mer87] prove that certain kind of Byzantine behaviour (the forks) cannot be detected. On the contrary, a suitable protocol can detect if the server deviates from fork-linearisability behaviour, where forks are accepted if they do not join again. In the following, we further slightly relax fork-linearisability to make it compatible with our pipelining approach.

Definition 3. *Two sequences π_1 and π_2 are* disjoint-forking *iff $\pi_1 = \alpha\beta_1$, $\pi_2 = \alpha\beta_2$, with α maximal and non-empty, and either $\beta_1 = \beta_2 = \emptyset$ or $\beta_1 \cap \beta_2 = \emptyset$.*

A set of n pairwise disjoint-forking sequences constitutes a tree (a path with no fork is a special case) with at most n leaves, and after each fork the two branches have to be set-disjoint.

The following property links Definition 3 with Condition 4 of Definition 2.

Property 1. *Two sequences π_1 and π_2 are disjoint-forking if and only if for every $o \in \pi_1 \cap \pi_2$, the sequence of the events that precede o in π_1 is the same as the sequence of the events that precede o in π_2 .*

Proof. First, we prove the necessary condition. Let $\pi_1 = \alpha\beta_1$ and $\pi_2 = \alpha\beta_2$. In the case $\beta_1 = \beta_2 = \emptyset$ the proof is trivial since the thesis holds for all $o \in \pi_1 = \pi_2$. In the case $\beta_1 \cap \beta_2 = \emptyset$, we have $\pi_1 \cap \pi_2 = \alpha$, hence for every $o \in \pi_1 \cap \pi_2 = \alpha$, the preceding elements in π_1 and π_2 are the same, by construction of α.

Now, we prove the sufficient condition. Consider the latest (i.e., rightmost) o for which it holds $o \in \pi_1 \cap \pi_2$. The preceding events are the same in π_1 and π_2 by hypothesis. We denote α this prefix, which contains o, and the remaining parts β_1 and β_2, so that $\pi_1 = \alpha\beta_1$ and $\pi_2 = \alpha\beta_2$. Sequence α is maximal by construction and non empty since contains o, at least. If $\pi_1 = \pi_2$ then $\beta_1 = \beta_2 = \emptyset$. If $\pi_1 \neq \pi_2$, β_1 and β_2 are not empty, but $\beta_1 \cap \beta_2 = \emptyset$, otherwise o would not be the latest event satisfying $o \in \pi_1 \cap \pi_2$. □

Property 1 justifies the introduction of a weaker form of disjoint-forking and the corresponding slightly weaker form of fork-linearisability, which are defined in the following and will be used in Section 7.7.

Definition 4 (Quasi-Disjoint-Forking)**.** *Two sequences π_1 and π_2 are* quasi-disjoint-forking *iff $\pi_1 = \alpha\beta_1$, $\pi_2 = \alpha\beta_2$, with α maximal and non-empty, and either $\beta_1 = \beta_2 = \emptyset$ or the following holds. Let O^c be the operation invoked by client c in $\beta_1 \cap \beta_2$. For each client c, all o in O^c are invoked before (in the real-time order) the first response to an invocation in $\beta_1 \cup \beta_2$.*

The above definition is clearly weaker than Definition 3, allowing partial overlap of branches, however, it states that those overlaps are limited. The extent of this limit depends on when responses are received by c. For example, c may will to wait a response for an operation o in order to be sure that the following updates are in the same branch of o, in case of malicious server. Definition 4 motivates the introduction of the following.

Definition 5 (Quasi-Fork-Linearisability)**.** *A sequence of events σ is* quasi-fork-linearisable *with respect to the semantics of a key-value store, if and only if for each client c_i, there exists a complete subsequence σ_i of σ and a sequential permutation π_i of σ_i such that*

1. *all completed operations of σ occurring at client c_i are in σ_i,*
2. *π_i preserves the real-time order of σ of all non commuting operations,*
3. *the operations of π_i satisfy the semantics of their sequential specification, and*
4. *each pair π_i, π_j is quasi-disjoint-forking.*

We note that, Conditions 2 and 4 of Definition 5 are slightly weaker forms of the ones that are present in Definition 2, while the other conditions are the same.

The capability of a protocol to detect deviation from quasi-fork-linearisability is formalised by the following definition adapted from [CSS07].

Definition 6 (Byzantine Emulation)**.** *A protocol P* emulates *a key-value store on a Byzantine server with quasi-fork-linearisability, if in every admissible execution of P the sequence of events observed by the clients is quasi-fork-linearisable in monotonic sense. Moreover, if the server is correct, then every admissible execution is complete and has a linearisable history.*

It is worth to further elaborate our comments following Definition 4. Suppose that all clients always wait the response to the previous operation before invoking the new one. In this case, Definition 3 holds, hence, fork-linearisability is guaranteed. In this sense, quasi-fork-linearisability can be regarded as a generalisation of fork-linearisability, which allows one to trade consistency for efficiency (see Section 7.8).

7.4 The Blocking Approach

The aim of this section is to show formally how adopting ADSs in a client-server setting with a naive protocol falls short of scalability. Our analysis shows that the throughput of the system (i.e., the maximum update invocation rate the system can sustain) can be approximated only at the cost of a very high latency (i.e., the time an update takes to be included in a read).

In our setting, we have many trusted entities that share a single root-hash. Since they cannot communicate directly, the common way to share the root-hash is to sign it and store it in the server. Clearly, only a trusted entity can legitimately update it. When the dataset have to be updated, the server must ask a trusted entity, an authenticator, to perform due checks and sign the new root-hash. The authenticator performs the checks on the basis of proofs derived from the current instance of the ADS and possibly other information. The kind of checks the trusted entity performs before signing the root-hash are responsible of the level of consistency guarantees provided by the whole system.

We introduce a very simple protocol, which can be regarded as an abstraction of the authentication part of other protocols described in literature (for example, see [BCK17, EKPT15, EKPT15, WWL$^+$09, PT07]). For the sake of simplicity, in this section, we focus on the interaction scheme among the actors, disregarding all security and consistency aspects that are not strictly needed. We call it *blocking protocol*, since its main characteristic is that while the server is waiting a signed root-hash from a trusted entity c_1, it cannot ask another trusted entity c_2 to sign another root-hash. In fact, the checks that c_2 should perform are usually based on data that is part of the reply from c_1, for example the signature of the root-hash provided by c_1. We analyse the performance of this protocol in term of the relation between throughput and response time.

The server keeps a *dataset* D equipped with an ADS. A group of update operations are applied to D during a *commit* as described in Section 7.3. After each commit D changes version. We denote the versions of D by D_i, where i

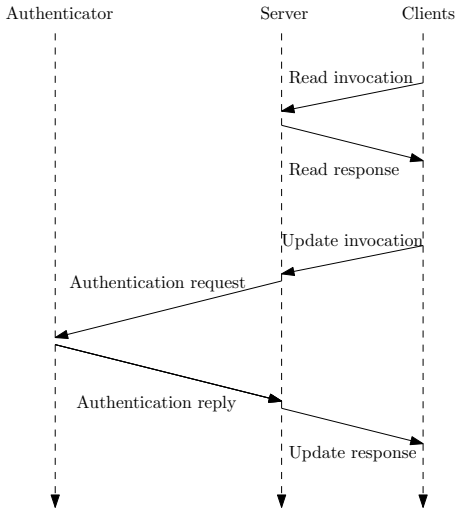

Figure 7.3: Interaction according to the blocking protocol.

is the index of the version. Version D_i has root-hash r_i. The *authentication* of D_i is $[r_i]_a$, which means that trusted entity a has checked that D_i derives from D_{i-1} by the application of a certain set of updates that conforms to certain consistency rules.

We consider three different roles.

Client. It is a trusted entity in charge of invoking operations.

Server. It is in charge of executing operations and sending the response to the client along with an authentication that the server should obtain from an authenticator.

Authenticator. It is a trusted entity in charge of providing the authentication for the next version of the dataset upon server request.

Figure 7.3 depicts an example of interaction according to the blocking protocol. A client starts an operation sending an *update invocation* or a *read invocation* to the server.

The *read invocation* specifies the key k to read. The server gets the value v associated with k and generates the corresponding proof(k) against the current root-hash authentication $[r_i]$. The *read response*, sent from the server to the

client, contains k, v, proof(k), and $[r_i]$. The client, at the receiving of the response, verifies the consistency of k, v and proof(k) against $[r_i]$.

The *update invocation* specifies the key k to update and the new value v'. At its reception, the server can perform the update procedure autonomously but cannot produce the authentication for the new version of dataset, since it cannot sign the new root-hash. It sends an authentication request to an authenticator containing k, its current value v, proof(k), v', $[r_{i-1}]$, where $i-1$ is the current version index.

Upon reception of an authentication request, the authenticator performs the following actions.

1. It checks proof(k) of k, v, against $[r_{i-1}]$.
2. It computes r_i from k, v', and proof(k).
3. It sends the authentication reply to the server containing $[r_i]$.

To increase the throughput, we allow queueing several update invocations and let the server asks an authenticator to authenticate all of them, cumulatively.

When the server receives an authentication reply, it updates the value of D from D_{i-1} into D_i, by exploiting the same information that were present in the request, and consider $[r_i]$ as the current authentication. It also sends to all clients, whose updates were executed, an update response.

Now, we analyse the scalability of the blocking approach. We call *authentication round* (or simply *round*) the process that start when the server sends an authentication request and ends when it receives the authentication reply. In the blocking protocol, there is only one round ongoing at a time. We denote by T the duration of a round. We denote by λ the frequency according to which update requests are received by the server, expressed in update requests per unit of time. We assume λ, as well as other parameters, to be constant in time. Let $m = \lambda T$ be the number of update requests received by the server during a round. If λ is big enough $m > 1$, hence, when a round terminates, there are already further update requests queued. We assume the server immediately starts a new authentication round when the previous one ends. This setting is depicted in Figure 7.4. Let t_S to be the time needed by the server to prepare one update to be sent to the authenticator. For simplicity, we assume that all update requests take the same time t_S, m updates take time mt_S. Let t_N be the time needed to put an update request into the network for transmission.

For simplicity, we assume that all update requests take the same time t_N and, if m update requests are cumulated into one authentication request, they take time mt_N to be transmitted. We denote by d the transmission (one-way) delay of the network. We assume this delay to be symmetric. We assume no network errors. Let t_A be the time taken by the authenticator to process one update request. For simplicity, we suppose that the processing time is the same for all updates and if m update requests are cumulated into one authentication request, the time taken by the authenticator to process all of them is mt_A. We assume all other overheads to be negligible, as well as the transmission time of the authentication reply. It holds that

$$T = 2d + m(t_S + t_N + t_A). \tag{7.1}$$

If we suppose the system to work at steady pace, we can substitute $m = \lambda T$, getting

$$T = \frac{2d}{1 - \lambda(t_S + t_N + t_A)}. \tag{7.2}$$

Figure 7.5 shows how T changes with λ according to Equation 7.2. The maximum throughput is $\tau = \frac{1}{t_S+t_N+t_A}$. Since a client can see its updates requests accepted only after that the authenticator replies, T is a lower bound of the response time and goes hyperbolically with λ.

We observe that resources tend to be mostly idle. For simplicity, we suppose $t = t_S = t_N = t_A$. The fraction of the round for which each resource is busy is $mt/T = \lambda Tt/T = \lambda t$, hence, the idle time ratio for each resource is $1 - \lambda t$. Note that, decreasing t (i.e., increasing the speed of the resources) so that T approaches $2d$, makes the idle time ratio to approach 1.

Clearly, increasing the throughput $1/t$ of the resources increases the cost of the system. We express the cost of the system vs. the required throughput of the system λ, for constant T, in the following way. We substitute $t_S + t_N + t_A = 3t$ and $m = T\lambda$ into Equation 7.1 and solve by t. We obtain $\frac{1}{t} = \frac{3\lambda T}{T-2d}$ as the cost of each resource.

These results strongly motivate the introduction of a pipelining approach, which is described in Section 7.6.

7.5 Overview of Intermediate and Main Results

The blocking protocol is not scalable and provides very weak security guarantees (for example, the server can easily reorder updates and reply on the basis of old versions). In this chapter, we provide a scalable and secure protocol

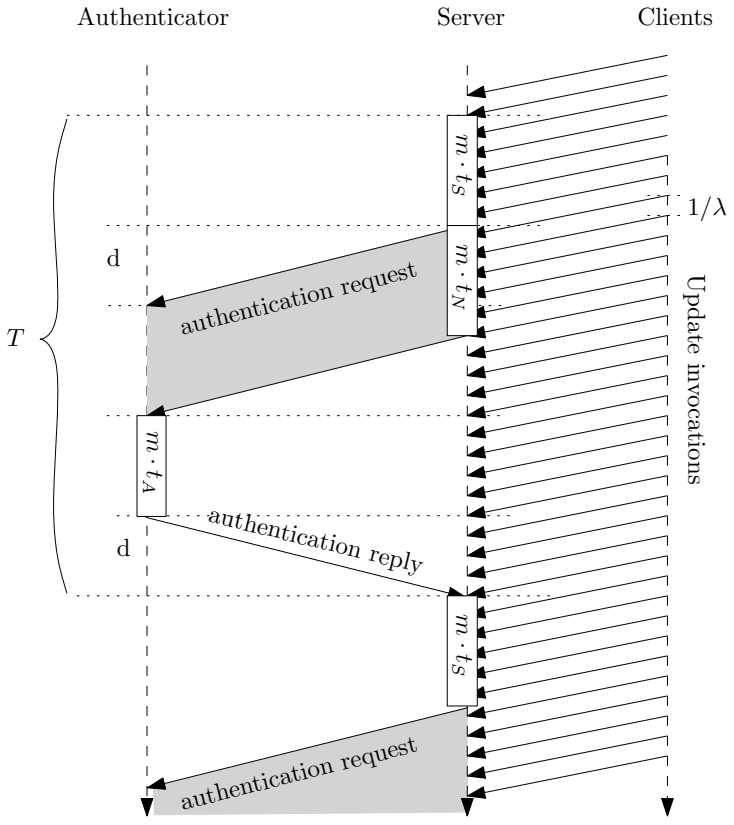

Figure 7.4: The model of server-authenticator interaction, according to the blocking protocol.

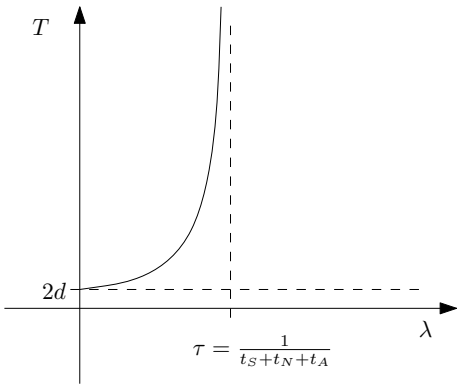

Figure 7.5: In the blocking protocol, the duration of an authentication round (T) hyperbolicly goes to infinite when the arrival frequency of update requests (λ) approaches the maximum throughput (τ).

that solves the same problem. We incrementally describe the solution in the next three sections. First, we show how it is possible to pipeline requests to the authenticator without waiting for its responses. At this stage, no particular consistency and security are provided. Then, we show a distinct result in which we do not care about efficiency, but we deal with strong consistency and security. Lastly, we show how to combine this two results.

In this section, we informally describe the ideas underlining these results, while complete details and formal proofs are provided in Sections 7.6, 7.7 and 7.8.

The Simplified Pipeline-Integrity Protocol

Our first objective is to devise a protocol that allows the server to send an authentication request without waiting for the result of the previous one. This is an essential aspect of our pipelining approach. We observe that the only information the authenticator sends back to the server is the signature of the new root-hash. This means that the server may decide to send an authentication request at any time while being able to build it with all the information, as in the blocking approach, except for the signature of the previous root-hash. Hence, our goal is to allow the authenticator to provide a proof that all the checks it performed were successful, without relying on the signature of the

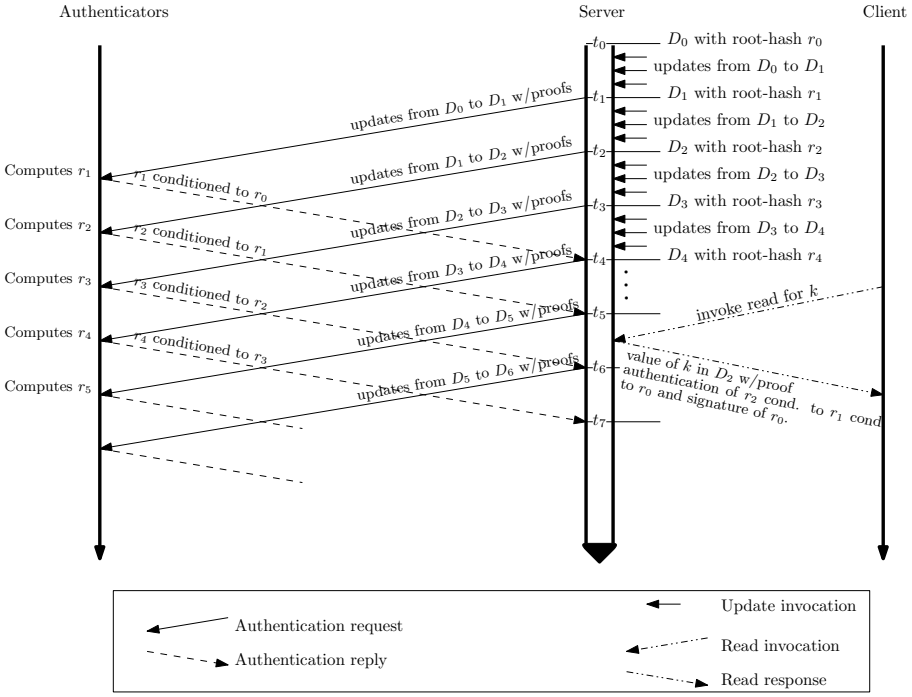

Figure 7.6: An example of use of conditional authentications to enable pipelining of authentication requests.

previous root-hash. The resulting proof should be usable by the server to build complete authentications to be used, for example, in read replies. In our approach, the authenticator can do that for any kind of checks, no matter how complex they are.

We introduce the concept of conditional authentication, which is formally defined in Section 7.6. It expresses the fact that a certain root-hash r_i is correct, if the previous one (r_{i-1}) was. Root-hash r_i results from r_{i-1} by the application of a sequence of update invocations that passes certain (consistency) checks. A *conditional authentication* is a signature of the ordered pair of root-hashes r_{i-1} and r_i. Conditional authentications can be *chained* with other (conditional or regular) authentications, if certain conditions are met (see Section 7.6).

In Figure 7.6, we provide an example of how we use conditional authenti-

cations. A regular stream of updates arrives to the server. The server sends authentication requests to the authenticator at regular intervals of time. In the example, authentication requests are pipelined since, between an authentication request and its reply, the server sends other authentication requests. No root-hash signature is sent in these authentication requests, hence, the authentications contained in the replies are conditioned. In the figure, authentication requests are sent at instants $t_1, t_2, \ldots$. An authentication request sent at time t_{i+1} includes the updates arrived since the sending of the previous authentication request at time t_i. The state of the dataset right after t_i is denoted D_i and its root-hash is denoted r_i. The proofs in the authentication request sent at time t_{i+1} are based on D_i. Suppose that between t_5 and t_6 a read invocation is received. To authenticate the root-hash of the proof contained in the response, the server can include the conditioned authentications it received (r_2 with respect to r_1 and r_1 with respect to r_0). If the server knows a signature that authenticate r_0, it can be included with those conditional authentications to provide a *chain* that has the same semantics of a regular authentication. In Section 7.6, we formally prove this, we show how to keep the chain bounded, we provide a formal description of our protocol, and we analyse its scalability.

An ADS-Based Quasi-Fork-Linearisabile Protocol

After having provided a scalable protocol, we focus on consistency and security. We introduce a protocol, called *history-integrity protocol*, that securely ensures quasi-fork-linearisability (see Definition 5) in the sense that any deviation of the server from that behaviour is detected. We recall that quasi-fork-linearisability is a consistency model in which the server can fork the history of the updates showing distinct branches to distinct clients and where intersection among branches is forbidden, except right after the fork. Essentially, our objective is to fulfil the following *security requirements*, which are tightly linked with some of the consistency constraints introduced in Section 7.3.

R1 Each update should appear exactly once in the sequence of updates to be applied to the dataset. The order chosen by the server should conform to the order each client issued its updates. A violation of this rule by the server must be detected. This requirement is linked with Items 1 and 2 of Definition 5.

R2 Clients should be able to detect if the server is trying to propose outdated versions of the dataset. That is, each client c should check that each

dataset version proposed by the server follows the last one that c has knowledge of. This requirement is linked with the monotonicity definition introduced in Section 7.3.

R3 Trusted entities should be able to detect the joining of forks according to the definition of quasi-fork-linearisability. That is, overlapping between distinct forks is allowed only for the updates invoked before the client receives any response from the server after the fork. This requirement is linked with Item 4 of Definition 5.

In Section 7.7, we describe a number of techniques that address the above requirements in a blocking setting. These techniques turn out to be compatible with the pipelining approach described above. Now, we briefly describe the intuition underlying those techniques.

Requirement R1 is addressed by hash-chaining the update invocations of each client and checking the consistency of the chain for each client on the authenticator. To keep track of the hash of the last update invocation across consecutive authentications, we authenticate this information in the very same ADS used to authenticate the dataset, under special *client-keys*.

Requirement R2 is addressed hash-chaining the root-hashes of consecutive versions of the dataset. The server responses to read invocations are always based on a certain version identified by a root-hash. Consider two consecutive read responses, ρ_1 and then ρ_2, sent to a client c based on versions identified by r_1 and r_2, respectively. In each response, the server provides a proof of monotonicity. In our example, this is the proof that r_1, that c saw in ρ_1, precedes r_2 in the hash-chain of the root-hashes. To obtain a short proof, we adopt an additional *history ADS* on this hash-chain whose root-hash is itself authenticated by the authenticator.

To address Requirement R3, each client sends, along with each invocation, the indication of the last dataset version it knows. The server must include this information, equipped with a proof obtained from the history ADS, in any authentication request. This is enough to enable authenticators to detect violations of the quasi-disjoint-forking rule.

In Section 7.7, we formally describe the above mentioned techniques and provide proofs of their security and correctness.

The Pipeline-Integrity Protocol

In Section 7.8, we show that it is possible to combine the above results. Even if this is the main result of the chapter, the resulting protocol and algorithms

inherit the technicalities of the previous intermediate results without adding any new fundamental concept. The messaging scheme is the same that we show for the simplified pipeline-integrity protocol, hence, the scalability properties, shown in Section 7.6, are preserved. Security and correctness of the combined solution, derive from the corresponding security and correctness properties of the history-integrity protocol, proven in Section 7.7. This extension is possible because of the chaining properties of the conditional authentications, introduced in Section 7.6.

7.6 The Simplified Pipeline-Integrity Protocol

From the analysis provided in Section 7.4, it is evident that the blocking approach obtains very poor results, in terms of throughput or latency of the whole system, compared with the theoretical capability of the distinct elements of the system. We recall that, according to the blocking approach, the rounds of authentication of the root-hashes are executed sequentially and the server blocks until the authentication reply is received (see Section 7.4).

In this section, we show how it is possible to create a protocol, which we call *simplified pipeline-integrity protocol*, that achieves much better results by pipelining authentication rounds. For the sake of simplicity, in this section, we focus only on the interaction scheme among the actors, disregarding all security and consistency aspects that are not strictly needed to explain it. Since the guarantees of the simplified pipeline-integrity protocol are quite modest, we do not provide any formal proof about them. A consistent and secure (but inefficient) protocol is shown in Section 7.7. In Section 7.8, that protocol is enriched with the interaction scheme shown in this section obtaining consistency, security and efficiency.

In the simplified pipeline-integrity protocol, invocations and responses for read and update operations have format and semantics very similar to those of the blocking approach. The only difference is related to the authentications of root-hashes, which are substituted by *chained authentications*, introduced in the following section.

Conditional and Chained Authentications

Consider an authenticator that is performing consistency checks and is computing and signing the new root-hash. At the same time, the server can apply updates it is receiving, creating a new status of the dataset and ADS. We recall

that using ADSs, we can efficiently link a cryptographic hash, called root-hash, with a large dataset of key-value pairs and that root-hash is supposed to be authenticated (usually signed) by a trusted entity. A fundamental idea of our contribution is that an additional authenticator can *conditionally authenticate* a root-hash r_i even if the signature of the previous root-hash r_{i-1} is not known yet. This is not a real authentication of r_i but it is still something that can be used together with an authentication of the of r_{i-1}, when it will be available. The simplified pipeline-integrity protocol allows the server to start a new authentication round when the previous one is not finished yet. Actually, the server may create a pipeline of authentication rounds which can be arbitrarily deep. By pipelining authentication rounds, we get three important advantages.

- We make better use of resources, since server, network and authenticators all work in parallel.
- We can achieve a much better trade-off between throughput and latency, since the authentication of a sequence of updates is split into several short rounds that are processed concurrently.
- We can have several authenticators working in parallel, each addressing a different set of updates.

As we will see, the cost to pay for this approach is that additional root-hash signatures have to be enclosed in the messages sent form server to trusted entities. However, this cost turns out to be quite small compared with the large advantages obtained (see Sections 7.6 and 7.10).

We denote by q the number of ongoing authentication rounds at steady operational pace. For the sake of simplicity, we assume the q ongoing authentications are performed with q different authenticators denoted by $a_1, \ldots, a_q$, where each authenticator can be in charge of only one authentication request at a time.

In the following, we deal with root-hash authentications in three forms: plain, conditional and chained. The plain authentication was introduced in Section 7.4 and it consists of just a signature of a root-hash. We call *conditional authentication* a signed pair $[r_i, r_{i+j}]$ ($j \geq 1$), whose semantics is the following: r_{i+j} is authenticated on the basis of data that are supposed to be genuine against r_i, hence, if an authentication for r_i is provided, also r_{i+j} can be considered authentic. The first root-hash of the pair is said to be *conditioning* while the second is said to be *conditioned*.

Two conditional authentications form a *chain* if the conditioning root-hash of the second is equal to the conditioned root-hash of the first. The chain of two conditional authentication is written $[r_i, r_j]\ [r_j, r_l]$ with $i < j < l$.

Property 2. *The chain of two conditional authentications* $[r_i, r_j]\ [r_j, r_l]$ *is semantically equivalent to the conditional authentication* $[r_i, r_l]$.

Proof. Consider an authenticator a generating $[r_i, r_j]_a$. When a assumes that r_i is a valid root-hash, it is equivalent to assume that its associated dataset D_i complies to a number of consistency rules. We can summarise this saying that a certain logic predicate p_i about D_i is true. Equivalently, stating that r_j is a valid, is equivalent to stating that predicate p_j is true. Hence, when a signs the pair (r_i, r_j), it states that the logic formula $p_i \rightarrow p_j$ holds. Analogously, $[r_j, r_l]$ and $[r_i, r_l]$ are equivalent to stating that $p_j \rightarrow p_l$ and $p_i \rightarrow p_l$ holds, respectively. By the rules of predicate logic, $(p_i \rightarrow p_j) \wedge (p_j \rightarrow p_l)$ entails $p_i \rightarrow p_l$. □

Property 3. *A plain authentication* $[r_i]$ *with the conditional authentication* $[r_i, r_j]$ *is semantically equivalent to the plain authentication* $[r_j]$.

Proof. Consider an authenticator a generating $[r_j]$ ($[r_i]$). Before signing it, a checks that $[r_j]$ ($[r_i]$) complies to a number of consistency rules, which can by summarised by logic predicate p_i (p_j) about dataset D_i (D_j). Also, signing $[r_i, r_j]$, is equivalent to state $p_i \rightarrow p_j$ (see proof of Property 2). By the rules of predicate logic, $p_i \wedge (p_i \rightarrow p_j)$ entails p_j. □

Properties 2 and 3 justify the extension of the definition of chained authentication to a sequence starting with one plain authentication followed by several chained conditional authentications. For example, from the above definitions, the sequence $[r_0]\ [r_0, r_5]\ [r_5, r_9]$ is a chained authentication, which, by Properties 2 and 3, is semantically equivalent to $[r_9]$.

Authentication chains can be arbitrarily long. Before trusting a chained authentication, we should check its coherency according to the above properties and verify its signatures. This procedure is formalised by Algorithm 1. We call *compaction* the process of reducing an authentication chain into a plain authentication, like $[r_9]$. This process is formalised by Algorithm 2.

Compaction should be performed by a trusted entity, since the final result requires a signature. In the simplified pipeline-integrity protocol, compaction is performed by authenticators.

Algorithm 1 Verification of a chained authentication.

Require: a chained authentication $A = [r_0][r_0, r_1] \dots [r_{n-1}, r_n]$ and the sequence of root-hashes $\bar{A} = r_0, r_1, \dots, r_n$. In the algorithm, we refer to elements of A as the 0-th, 1-st,..., n-th.

```
Check the signature [r_0] of r_0
for i in 1...n do
    Check the signature [r_{i-1}, r_i] of the pair (r_{i-1}, r_i)
    if i = 1 then
        Let the initial elements of A be [y][x, r_1]
        Check that x = y
    else
        Let the (i-1)-th and the i-th elements of A be [r_{i-2}, y][x, r_i]
        Check that x = y
    end if
end for
A and Ā are verified if all the above checks are successful.
```

Algorithm 2 Compaction of a chained authentication.

Require: a chained authentication $A = [r_0][r_0, r_1] \dots [r_{n-1}, r_n]$ and the sequence of root-hashes $\bar{A} = r_0, r_1, \dots, r_n$.

```
Perform Algorithm 1 on A and Ā.
if A and Ā are successfully verified then
    return [r_n]
else
    fail
end if
```

Pipelined Execution of Authentication Requests

The adoption of chained authentication allows us to pipeline authentication requests. In Section 7.4, a commit starts when an authentication request is sent and ends at the reception of the corresponding reply. While waiting for the reply, the server cannot do anything. According to definitions in Section 7.3, the server cannot even reply to read operations. However, nothing prevents to think about commits as if they were limited to the processing of the authentication reply only. In this way, the server can, at least, reply to read requests on the basis of the previous state of the dataset, while waiting for the

authentication reply. We now go further showing how it is possible to start an authentication request before receiving the response to the previous one.

To simplify the explanation, in the following, we assume all processing time on server and authenticators to be negligible as well as transmission time, but we assume the one-way transmission delay to be non negligible and denoted by d. These hypotheses are relaxed at the end of this section when we evaluate the scalability of the protocol. A commit χ_i encompasses all the operations needed (on server or authenticator) to authenticate version i of the dataset, which we denote D_i. We denote D_0 the initial state of the dataset. An ADS on D_i is denoted by Δ_i and its root-hash r_i. The reader should consider all these symbols as abstract mathematical values. The state of the server will be introduced later. We call t_i the instant when the commit request ρ_i related to χ_i is sent, which contains all information needed by the authenticator to compute authentication for D_i. We associate with χ_i all updates whose requests are received in the interval (t_{i-1}, t_i).

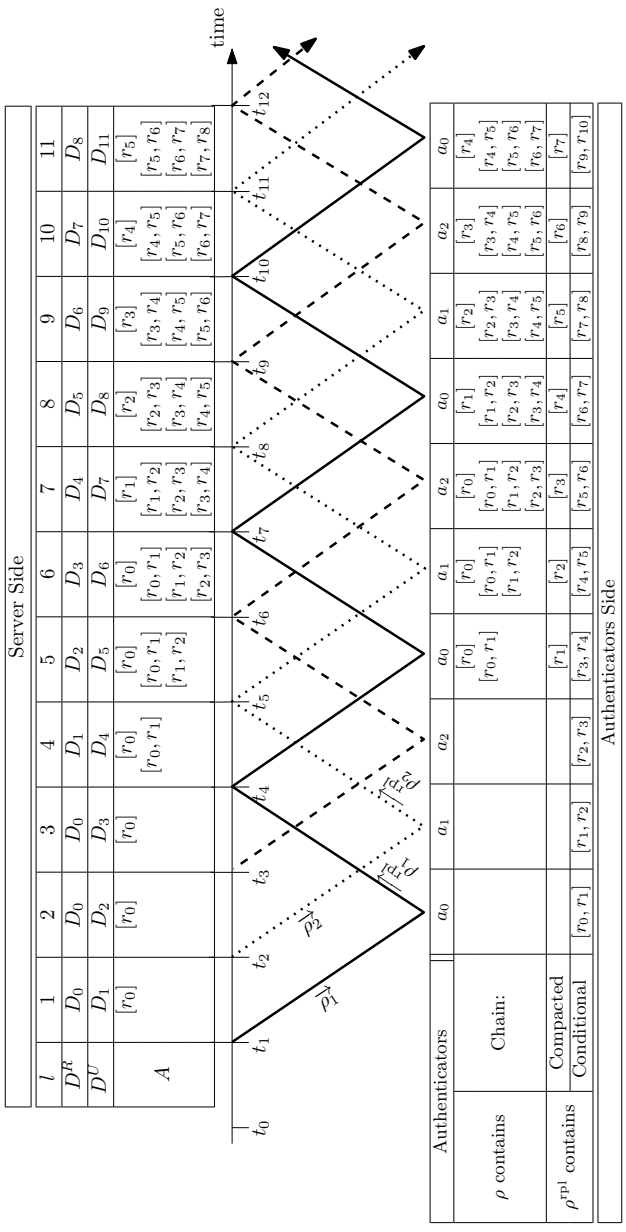

Figure 7.7: An example of execution of the simplified pipeline-integrity protocol with three authenticators.

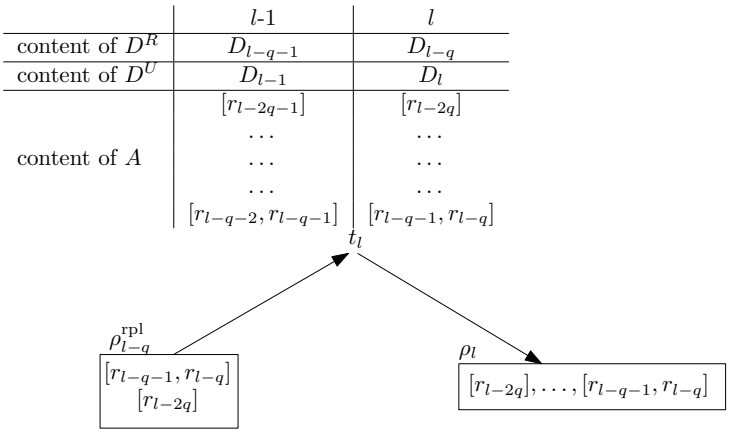

Figure 7.8: The general scheme that links authentication requests/replies to the changes of the state of the server in the simplified pipeline-integrity protocol.

We assume synchronous and reliable operation (these hypotheses are relaxed in Section 7.9). Figure 7.7 depicts the communication between the server and q authenticators (where $q = 3$ in the figure) for the simplified pipeline-integrity protocol. To have a pipeline with q stages (one for each authenticator), the server must send an authentication request every $\Delta t = t_i - t_{i-1} = 2d/q$ for all $i > 2$. Let ρ_i^{rpl} be the reply to authentication request ρ_i. From the above assumptions, starting from t_{q+1}, ρ_i^{rpl} is received at t_{i+q}. As detailed below, ρ_i^{rpl} contains conditional authentication $[r_{i-1}, r_i]$ to be used to conditionally authenticate D_i. This is computed on the basis of values taken from D_{i-1}, of proofs derived from Δ_{i-1}, and of all updates arrived at the server between (t_{i-1}, t_i). We assume that D_0 is empty and the corresponding r_0 is authenticated by $A = [r_0]$ know by the server. When ρ_i^{rpl} is received, the conditional authentication $[r_{i-1}, r_i]$ is appended by the server to A to obtain the chained authentication of D_i. To avoid that A grows indefinitely, the server includes into ρ_i the current content of A. The authenticator performs its compaction, obtaining $[r_{i-q}]$, and includes it into ρ_i^{rpl}. The server uses $[r_{i-q}]$ to shorten A. This is done starting from t_{2q+1}. In this way, A turns out to be bounded in length.

An authentication request is *outstanding* if no corresponding reply was received for them yet. In our setting, at most q authentication requests can be

outstanding.

Server Data Structures

To realize the simplified pipeline-integrity protocol, the server keeps two notable categories of data structures. They are *R-data-structures* and *U-data-structures*. They are distinguished by superscript ^{R} and ^{U} respectively. The first category is dedicated to serving read invocations, the second is dedicated to the processing of update invocations. For the simple pipeline-integrity protocol, the following are parts of the status of the server: dataset D^R with its ADS Δ^R and dataset D^U with its ADS Δ^U, which are stored by the server.

These data structures change value only at instants t_i. The values assumed by each data structure between instants t_i and t_{i+1} is denoted by the same symbol, with subscript i, like D_i^R and Δ_i^R. We denote by l the index of the last time instant t_l in which an update request was sent, which is also the last instant in which the status was updated. In the absence of subscript, current value is assumed, for example, $D^R = D_l^R$ and $D^U = D_l^U$.

For the hypothesis of synchronous operation with $\Delta t = 2d/q$, ρ_i^{rpl} is always received at t_{i+q}. At t_l (see Figure 7.8), an authentication reply ρ_{l-q}^{rpl} is received and R-data-structures are updated to version D_l^R and Δ_l^R on the basis of the updates contained in ρ_{l-q}, i.e., contained in the corresponding request. Further authentication request ρ_l is sent containing proofs based on D_{l-1}^U and Δ_{l-1}^U. Each R-data-structure tracks the corresponding U-data-structure with a delay of $q\Delta t$ and the following hold: $D^U = D_l$, the root-hash of Δ_i^U is r_i, $D^R = D_{l-q}^U$, $\Delta^R = \Delta_{l-q}^U$, the root-hash of Δ_i^R is r_{i-q}.

Even if theoretically we say that the server keeps D^R (Δ^R) and D^U (Δ^U), since the first is a delayed version of the second, they only differ for the updates arrived after t_{l-q}. Efficient storage solutions can be devised to do that without doubling space occupation.

Additionally, the server keeps

- a chained authentication A for D_l^R with the following structure $[r_{l-2q}]$ $[r_{l-2q}, r_{l-2q+1}]$ $[r_{l-q-2}, r_{l-q-1}] \dots$ $[r_{l-q-1}, r_{l-q}]$ and the corresponding sequence of root-hashes $\bar{A} = r_{l-2q}, r_{l-2q+1}, \dots, r_{l-q}$,
- a queue Ω of all outstanding authentication requests, which, after t_{q+1}, is $\rho_{l-q}, \dots, \rho_l$, and
- for each client c, a queue Q^c containing all the update operations invoked by client c and received by the server (in the invocation order)

Variable	Description
D^R	Dataset used to serve read invocations.
Δ^R	ADS related to D^R.
D^U	Dataset used to record updates and send authentication request.
Δ^U	ADS related to D^U.
l	Index of the dataset version contained in $D^U = D_l$.
A	Authentication of D^R and Δ^R in the form $[r_{l-2q}]$ $[r_{l-2q}, r_{l-2q+1}]$ $[r_{l-q-2}, r_{l-q-1}] \dots$ $[r_{l-q-1}, r_{l-q}]$.
$\overline{A}$	Sequence of the root-hashes r_{l-2q}, $r_{l-2q+1}, \dots,$ r_{l-q} on which A is based.
Ω	Queue of outstanding authentication requests $\rho_{l-q}, \dots, \rho_l$
Q^c	A queue for each client c containing update invocations of c for which no authentication request was sent yet.

Table 7.1: State of the server for the simplified pipeline-integrity protocol.

that are not associated with a commit, i.e. that have not been sent in an authentication request, yet.

Table 7.1 summarises the content of the state of the server for the simplified pipeline-integrity protocol.

Authentication: Messages and Processing

In the simplified pipeline-integrity protocol, authenticators do not keep any state. The server sends an authentication request ρ_l at time t_l with the purpose of

1. getting what is missed in A to get a chained authentication of D_l (containing updates invocation arrived up to t_l), which will be the content of D^R after the reception of ρ_l^{rpl}, and
2. getting a compacted version of the current A, which will be equal to $[r_{l-2q}]$ after the reception of ρ_l^{rpl}.

Authentication request ρ_l sent at time t_l contains

- a sequence of all update operations received by the server between t_{l-1} and t_l (currently stored in the queues Q^c), preserving the order that they

have in Q_c, where the interlacing of the sequences of updates of distinct clients is arbitrarily chosen by the server,

- the proofs for all $\langle k, v\rangle$ involved in the above updates, computed according to Δ_{l-1}, where v is the value as in D_{l-1},
- the current authentication chain A, with the corresponding root-hashes $\overline{A}$, to be compacted.

Algorithm 3 Simplified pipeline-integrity protocol – Authenticator. Operations performed by an authenticator a upon reception of an authentication request ρ.

Require: An authentication request ρ that was sent by the server at time t_l, containing:

- a sequence B of updates in the form $u^c = \langle k, v'\rangle$, where c is the client that invoked the update and v' is the new value of k,
- for all keys k involved in B, proof$(\langle k, v\rangle)$ where v is the previous value of k,
- chained authentication $A = [r_{i-q}]\ [r_{i-q}, r_{i-q+1}] \dots\ [r_{i-1}, r_i]$ and corresponding sequences of root-hashes $\overline{A}$ (see the status of the server in Table 7.1).

1: Arbitrarily select one of the proofs and compute the root-hash $\bar{r}$.
▷ $\bar{r}$ is supposed to match r_{l-1} on the server when ρ is sent.
2: Check all other proofs against $\bar{r}$ to verify that they all comes from the same dataset version.
3: Computes from the proofs and from new values, the new root-hash $\widetilde{r}$.
▷ $\widetilde{r}$ is supposed to match r_{l-q} on the server when ρ^{rpl} is received.
4: Sign the conditional authentication $[\bar{r}, \widetilde{r}]_a$.
5: Based on A and $\overline{A}$, compute a compact version $[r_i]_a$ of A.
▷ i turns out to be $l - 2q$ when ρ^{rpl} is received.
6: Sends the authentication reply ρ^{rpl} containing $[\bar{r}, \widetilde{r}]_a$ and $[r_i]_a$.

Upon reception of an authentication request, the authenticator a performs the actions described in Algorithm 3. Since the server does not provide authentication for $\bar{r}$, the authenticator only provides a conditional authentication of the subsequent root-hash $\widetilde{r}$ on the basis of the assumption of authenticity of $\bar{r}$.

Proving the authenticity of $\bar{r}$ is up to the trusted entity that will use that conditional authentication. In our approach, this is essentially done considering the conditional authentication within the context of an authentication chain.

Supposing synchronous operation, ρ^{rpl} reaches the server after $q\Delta t$ with respect to the instant ρ was sent. This means that at each t_l the server gets $[r_{l-2q}]$ and $[r_{l-q-1}, r_{l-q}]$. These values are used by the server to update A so that current value of D^R can be authenticated with a chain of q conditional authentications plus one plain authentication.

When the server receives ρ^{rpl}, it executes Algorithm 4 to update its state and to send the new authentication request. The algorithm starts its execution at t_l by incrementing the variable l. Lines 2-4 are related to the processing of ρ^{rpl}_{l-q}. After them, the following read invocations are served on the basis of $D^R = D_{l-q}$. Lines 5-11 are related to the creation of ρ_l on the basis of $D^U = D_{l-1}$ and to the update of D^U to be ready for t_{l+1}.

Scalability

With the intent to evaluate the scalability of the pipeline-integrity protocol, we relax the hypothesis of negligible computation and transmission time, but we keep operations synchronous. Essentially, we put ourselves in a setting comparable with the setting shown in Section 7.4. As stated in Section 7.3, ideally we would like to achieve high throughput while keeping response time bounded.

More formally, let λ be the arrival rate of the updates and d be the one-way network delay between server and authenticators. Our ideal scalability objective is to have response time $O(d)$, that is, independent from how λ is large.

We now show that with the above described protocol we can get very close to the ideal goal.

Let t_S, t_N, and t_A be the time taken to process or transmit one update operation during one authentication round by the server, the network and the authenticator, respectively. Let $\bar{t}_S$, $\bar{t}_N$, and $\bar{t}_A$ the time taken for processing or transmitting one conditional authentication of the authentication chain by the server, the network and the authenticator, respectively. Let $\tilde{t}_S$, $\tilde{t}_N$, and $\tilde{t}_A$ a constant amount of time spent in a round by the server, the network and the authenticator, respectively.

Algorithm 4 Simplified pipeline-integrity protocol – Server. Operations performed by the server upon reception of an authentication reply.

Require: An authentication reply ρ^{rpl} from authenticator a containing $[r_{l-q-1}, r_{l-q}]$ and $[r_{l-2q}]$.

1: $l \leftarrow l + 1$
2: Pull from Ω the authentication request ρ corresponding to ρ^{rpl}.
▷ ρ should be the first in Ω, due the timing hypothesis.
3: Update D^R and Δ^R according to the update operations specified in ρ.
4: Update A using authentications of ρ^{rpl}, namely, $[r_{l-q-1}, r_{l-q}]$ is added to the right of A and $[r_{l-2q-1}][r_{l-2q-1}, r_{l-2q}]$ is substituted by $[r_{l-2q}]$.
▷ A should turns out to be the authentication of current D^R.
5: Let Y be an empty sequence of updates.
6: **for each** client c **do**
7: Pull from Q^c all updates and append them to Y in the same order.
▷ The interlacing of updates of different clients may be arbitrarily chosen
8: **end for**
9: Prepare a new authentication request ρ' for a, containing all updates in Y with their signatures, proofs computed according to the current value of D^U and Δ^U, new values, and the current value of A to be compacted.
10: Push ρ' as last element of Ω.
11: Send ρ' to a.
12: Update D^U and Δ^U according to the updates of Y.

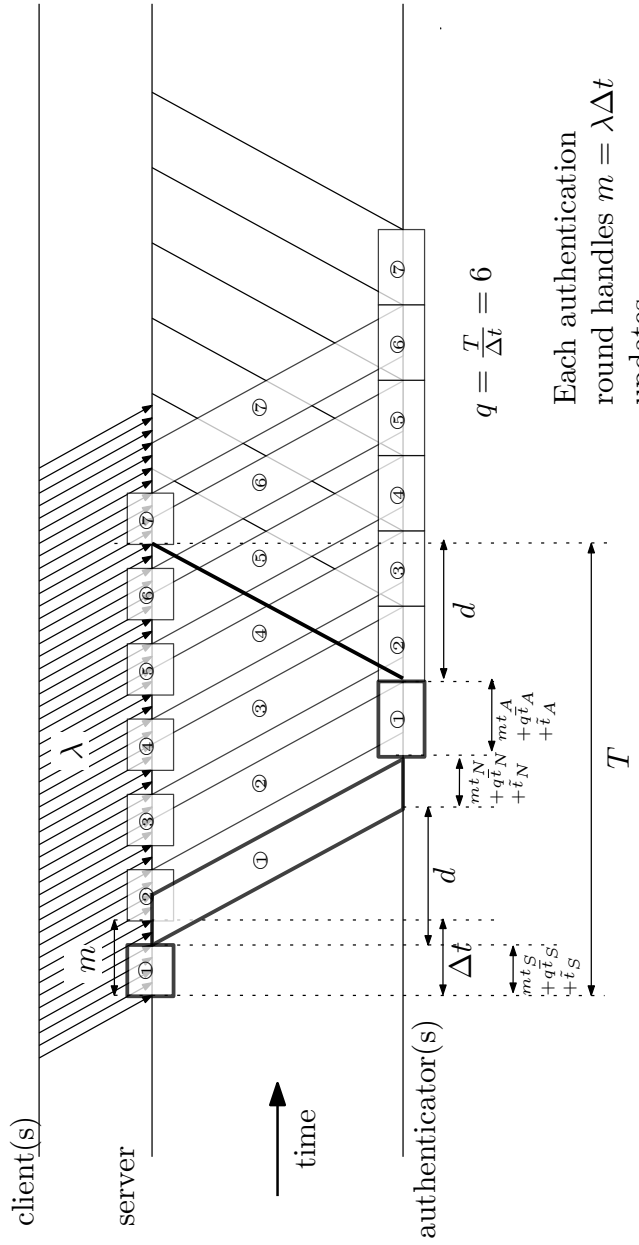

Figure 7.9: Interaction and timings between client(s), server, and authenticator(s) for the simplified pipeline-integrity protocol, when non-negligible computation and transmission time are considered.

Figure 7.9 depicts this new setting, where T is the time taken by each authentication round, λ is the arrival frequency of the updates, d is the one-way network delay between server and authenticators, q is the number of authenticators (which equals the length of the authentication chain), $\Delta t = T/q$ is the interval of time between the start of two consecutive authentication rounds, and $m = \lambda \Delta t$ is the number of updates to be processed by one round.

We denote by $\alpha = t_S + t_N + t_A$ the total processing/transmission time for one update, by $\beta = \bar{t}_S + \bar{t}_N + \bar{t}_A$ the total processing/transmission time for one conditional authentication, and by $\gamma = \tilde{t}_S + \tilde{t}_N + \tilde{t}_A + 2d$ the constant terms.

Now, suppose to keep α, β, γ and λ constant and to increase q, while correspondingly decreasing Δt and m, and to observe how the duration of a round varies. The duration of a round is

$$T = m\alpha + q\beta + \gamma. \tag{7.3}$$

Substituting $m = \lambda T/q$ and solving by T, we have

$$T = \frac{q\beta + \gamma}{1 - \frac{\alpha\lambda}{q}}. \tag{7.4}$$

We consider T as a function of q, defined for $q \in (\alpha\lambda, +\infty)$, and find the value of q for which $T(q)$ is minimum. By regular calculus, the minimum is reached at

$$q_{\min} = \alpha\lambda + \sqrt{(\alpha\lambda)^2 + \frac{\gamma\alpha\lambda}{\beta}}. \tag{7.5}$$

For $q \in (\alpha\lambda, q_{\min})$, $T(q)$ is decreasing. For $q > q_{\min}$, $T(q)$ is above the asymptote $T = \beta q + \gamma$. By simple substitution, it is easy to see that $T(q_{\min})$ is not bounded by a constant when λ increases. However, by monotonicity of square root, $2\alpha\lambda < q_{\min}$ and $T(q_{\min}) < T(2\alpha\lambda) = 4\beta\alpha\lambda + 2\gamma$, which is a line with a very small slope, since α and β are usually quite small compared to γ (see below). One may object that $q_{\min}$ is fractional, in general, and we are forced to choose either $\lfloor q_{\min} \rfloor$ or $\lceil q_{\min} \rceil$. It is easy to show that $q_{\min} - 2\alpha\lambda > 1$ when $\lambda > \frac{1}{\alpha(\gamma/\beta - 2)}$. Hence, for λ large enough, we have $2\alpha\lambda \leq \lfloor q_{\min} \rfloor \leq q_{\min}$. In this case, since $T(q)$ is decreasing up to $q_{\min}$, we obtain $T(\lfloor q_{\min} \rfloor) \leq T(2\alpha\lambda) = 4\beta\alpha\lambda + 2\gamma$.

The above arguments support the following theorem.

Theorem 1 (Scalability)**.** *In the simplified pipeline-integrity protocol, there exists a value of the number of authenticators q for which the response time for each authentication request is bounded by $4\beta\alpha\lambda + 2\gamma$ for λ large enough, where α is the total processing/transmission time for one update, β is the total processing/transmission time for one conditional authentication, and γ is the remaining processing/transmission time and network delay in a round that does not depend on the number of updates in the request or q.*

We point out that the results stated by Theorem 1 is very close to our ideal objective. In fact, from measurements performed contextually to the experiments of Section 7.10, we observed that the product $\alpha\beta$ is in the order of 10^{-6} and γ is in the order of 10^{-1} seconds. For the two terms to be comparable λ should be in the order of 10^5 updates per seconds. If we consider negligible the first term, T is $O(\gamma)$, where γ is largely dominated by d.

The scalability of the simple-pipeline integrity protocol is further supported by the following analyses.

The presence of the authentication chain introduce some overhead. Every authentication request carries an authentication chain of length q. We have q rounds every T, hence, the overhead introduced on the work of authenticator, network, and server is $q^2\bar{t}_A/T$, $q^2\bar{t}_N/T$, and $q^2\bar{t}_S/T$, respectively. In practice, we expect q to be small (it is less than 8 in the experiments of Section 7.10). Further, $\bar{t}_A$, $\bar{t}_N$, and $\bar{t}_S$, depend on the choice of cryptographic primitives, but we do not expect them to be much larger than t_A, t_N, and t_S, respectively. Hence, we expect the overhead to be quite small in practice.

The bottleneck of the system is either the authenticator or the network or the server. Supposing βq, $\tilde{t}_S$,$\tilde{t}_N$, and $\tilde{t}_A$ to be small, the throughput of the system is approximatively given by $\min\left(\frac{1}{t_S}, \frac{1}{t_A}, \frac{1}{t_N}\right)$. Supposing the resources to be perfectly balanced, we can state $t = t_A = t_N = t_S$. Under this assumption, when the system is computing at its maximum speed, all resources are fully busy. This is much better than what we noted for the blocking approach (Section 7.4), which heavily underutilises resources.

Now, we aim to understand how much this solution costs in terms of additional throughput to be provisioned to resources in order to increase the maximum throughput of the system while keeping T constant. We additionally assume the time spent to perform distinct activities on the same resource to be proportional with the same factor. This allows us to state $\alpha = 3t$, $\beta = 3at$, and $\gamma = 3bt$, where a and b are constants. In this way, $1/t$ is proportional to both the throughput of each resource and to the cost of the whole system.

	Blocking	**Pipelining**
Maximum throughput	$\frac{1}{t_S+t_N+t_A}$ theoretical since $T \to \infty$	$\min\left(\frac{1}{t_S}, \frac{1}{t_N}, \frac{1}{t_A}\right)$ approximated
Round duration (proportional to response time) $\alpha = t_S + t_N + t_A$	$\frac{2d}{1-\alpha\lambda}$ $\alpha = t_S + t_N + t_A$	$\frac{q\beta+\gamma}{1-\frac{\alpha\lambda}{q}}$ $\alpha = t_S + t_N + t_A$ see text for definitions of β, γ, and q
Unused fraction of each resource	$1 - \lambda t$ where $t = t_S = t_N = t_A$ for $t \to 0$, $1 - \lambda t \to 1$ and $T \to 2d$	0 for $t_S = t_N = t_A$ at maximum throughput
Overhead	0	$\frac{q^2\bar{t}_S}{T}, \frac{q^2\bar{t}_N}{T}, \frac{q^2\bar{t}_A}{T}$
Cost vs. throughput i.e., the throughput required for each resource ($1/t$) to make the system have a given throughput (λ).	$\frac{1}{t} = \frac{3\lambda T}{T-2d}$ for $t = t_S = t_N = t_A$	$\frac{1}{t} = \frac{3\lambda T/q+aq+b}{T-2d}$ for $t = t_S = t_N = t_A$, see text for the definition of a and b

Table 7.2: Summary of scalability analysis results for the pipelining approach compared against the same results for the blocking approach.

From Equation 7.3 we obtain

$$\frac{1}{t} = \frac{3\lambda T/q + aq + b}{T - 2d}.$$

Essentially, supposing $aq + b$ to be small, the simplified pipeline-integrity protocol cuts by q the cost to increase the throughput of the system by a given amount, with respect to the same cost for the blocking approach.

Table 7.2 summarises the above results and compares them with those obtained in Section 7.4 for the blocking approach.

7.7 An ADS-Based Quasi-Fork-Linearisabile Protocol

In this section, we show a protocol named *history-integrity protocol* that provides quasi-fork-linearisability and allows clients to detect deviation from it. In more formal terms, it emulates a key-value store with quasi-fork-linearisability on a Byzantine server. We recall that quasi-fork-linearisability is a consistency model in which the server can fork the history of the updates showing distinct branches to distinct clients. Intersection among branches is ruled out, except for a limited number of updates right after the fork (see Section 7.3).

This protocol does not have the scalability of the simplified pipeline-integrity protocol shown in Section 7.6, but its construction turns out to be compatible with that approach and it is a fundamental part of the main result of this chapter shown in Section 7.8.

In the rest of this section, we refer to the requrements introduced in Section 7.5. We recall that Requirement R1 is about ensuring that the server does not reorder the updates of each client, Requirement R2 is about ensuring that the server cannot go back in time with the version of the dataset, and R3 is about ensuring quasi-fork-linearisability.

Server Status

The status of the server for the history-integrity protocol is fully summarised in Table 7.3. It contains a key-value store D and an ADS Δ over D. Values of D and Δ change at each commit. We denote by D_j and Δ_j their value after the j-th commit, where j is the *version index* of the dataset. The root-hash of Δ_j is denoted r_j. The index l of the version that was produced in the last commit is also part of the state of the server. In principle one may expect the authentication $[r_l]$ of the current version of D (and Δ) to be also stored by the

Variable	Description
D	The dataset
Δ	ADS related to D.
l	Index of the version stored in D.
Q^c	A queue for each client c containing update invocations of c still not associated with a commit.
Π	History ADS, for the authentication of the sequence of history-pairs $\langle j, H_j \rangle$ (see text).
P	A mapping from each client c to a queue of history-pairs, ordered from head to tail by increasing version index.
A	Authentication of D, Δ and Π in the form $[R_l]$ where R_l is the root-hash of the current value of Π.

Table 7.3: State of the server for the history-integrity protocol.

server. However, as will be clear in the following, this is not necessary. The other elements of the server state are introduced in the rest of the description.

Following the client-server interaction described in Section 7.3, the server replies to read requests immediately, if no commit is ongoing. As in the blocking approach (see Section 7.4), it accumulates the update requests received by client c in a queue Q^c. The authentication request for the j-th commit contains a sequence of updates in the order they are supposed to be applied to D_{j-1} to obtain D_j. The next root-hash r_j is computed by an authenticator on the basis of this sequence and of the proofs of the modified keys, which are preventively checked against $[r_{j-1}]$.

Consistency Enforcement

To fulfil Requirement R1, we introduce the following construction. Each update operation invoked by client c is represented as a tuple $u(i) = \langle k_i, v_i, \text{hash}(u(i-1)), i \rangle$

(this is enriched below to satisfy further requirements). When useful, we specify also the client as superscript writing $u^c(i)$. We assume each client specifies a sequence number i for each update, independently from other clients, starting from $i = 0$. In the update invocation, the client sends $u(i)$ along with its signature $[u(i)]_c$. Each client keeps $\text{hash}(u(i))$ to be used in the construction of $u(i+1)$. The only exception is the first update invocation which is $u(0) = \langle k_0, v_0, \eta_0^c, 0\rangle$, where η_0^c is a constant that is different for each client (e.g., a random number locally generated by c) and play the role of $\text{hash}(u(-1))$. Clearly, it is possible to check the integrity of a sub-sequence of update invocations $u(i), u(i+1), \ldots$ provided that $\text{hash}(u(i-1))$ is known. Suppose that an authentication round commits, for a certain client c, updates up to $u^c(i)$. At the next authentication round, we call *past-hash* for c the value $\eta^c = \text{hash}(u^c(i))$, that is the hash of the last update that was committed. To check the correctness of the sequence of the updates specified in an authentication request for each client c, an authenticator needs η^c and a way to authenticate it. We introduce special *client-keys*, one for each client, denoted κ^c. We store the pairs $\langle \kappa^c, \eta^c\rangle$, for each c, in D so that they can be authenticated, as if they were regular data. The initial state of the dataset $D = D_0$ stored by the server does not contain any regular key but contains all $\langle \kappa^c, \eta_0^c\rangle$ for each client c. Pairs $\langle \kappa^c, \eta^c\rangle$ are sent with $\text{proof}(\Delta, \langle \kappa^c, \eta^c\rangle)$ and are used by authenticators to verify the sequence of $u^c(i)$ specified in the authentication request. During each commit, authenticators also consider the update of η^c when computing r_j. The effective update of $\langle \kappa^c, \eta^c\rangle$ in D_j (and in Δ_j) is performed by the server, as for regular keys.

To fulfil Requirement R2, we introduce the concept of *history-hash*. After the j-th commit, the history-hash H_j is defined as $H_j = \text{hash}(H_{j-1}|r_j)$ (we assume H_{-1} to be an arbitrary constant value to initialise the chain). Clearly, H_j uniquely identifies a sequence of root-hashes and hence a sequence of datasets, up to D_j. We also consider pairs $\langle j, H_j\rangle$, that we call *history-pairs*. These are stored in an ADS on the server that we call Π, ordered according to increasing j. See Figure 7.10 for a picture representing this construction. The state of this ADS also changes at each commit, hence the state of Π after the j-th commit is denoted Π_j. Its root-hash is called history root-hash and denoted R_j. The current history root-hash, between two commits, is R_l. Its authentication $[R_l]$ is stored by the server in A. Note that, to authenticate a key-value pair in the current $D = D_l$ by a proof obtained from $\Delta = \Delta_l$, we do not need to store $[r_l]$. In fact, $[R_l]$ is enough: to authenticate a key-value pair p in D_l, we need p, l, $\text{proof}(\Delta_l, p)$, H_{l-1}, $\text{proof}(\Pi_l, \langle l, H_l\rangle)$, and $A = [R_l]$. Each time the server sends

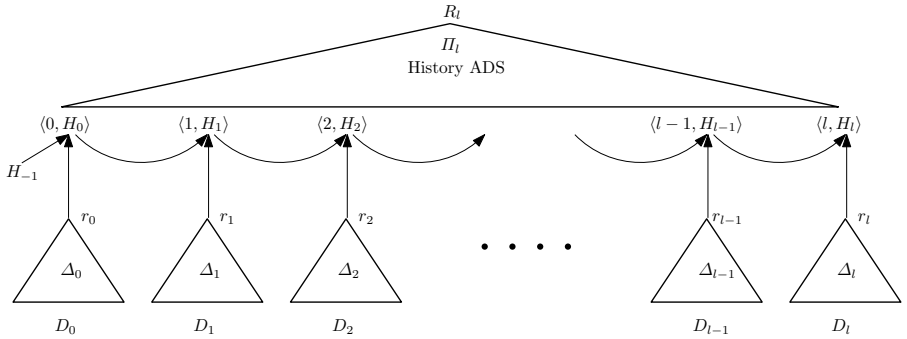

Figure 7.10: Conceptual construction to fulfil Requirement R2.

a response to a client, it includes this information. The verification procedure is a naive variation of the procedure described in Section 7.2.

Each client c stores a queue Γ of history-pairs, ordered from head to tail in increasing value of version index. We call them *local history-pairs for* c and are history-pairs that c received from the server. In other words, if c receives a response based on $p = \langle l, H_l\rangle$, where l is the last committed version, c should push p into Γ, at some point. The server always equips each response with an additional history-pair $\langle V^c, H^c\rangle$ which should be a local history-pair of c. The server also includes proof$(\Pi_l, \langle V^c, H^c\rangle)$ in the response messages. When receiving a message from the server, c always checks that $\langle V^c, H^c\rangle \in \Gamma$, both p and $\langle V^c, H^c\rangle$ are authenticated by $[R_l]$, and $V^c \leq l$. Finally, c pushes $\langle l, H_l\rangle$ into Γ. Further details are given in Section 7.7.

The server sends to authenticator a an authentication request containing $l-1$, H_{l-2}, proof$(\Pi_{l-1}, \langle l-1, H_{l-1}\rangle)$, and $A = [R_{l-1}]$. Authenticator a computes R_l and sends $[R_l]_a$ to the server. In computing R_l, besides regular key-value updates and updates of past-hashes for client-keys, a consider also Π_l deriving from Π_{l-1} by adding $\langle l, H_l\rangle$, where $H_l = \text{hash}(H_{l-1}|r_l)$, as by Figure 7.10. To enable monotonicity checks by authenticators and to fulfil Requirement R3, we slightly modify the format of updates as follows: $u^c(i) = \langle k_i, v_i, \text{hash}(u(i-1)), i, \langle V^c, H^c\rangle\rangle$. With respect to the definition of $u^c(i)$ given above, we add the tuple $\langle V^c, H^c\rangle$, that is, the latest history-pair pushed into Γ by c. Each $u^c(i)$ is put into an authentication request by the server along with the corresponding proof$(\Pi, \langle V^c, H^c\rangle)$. With this information, a can perform an additional check to verify that $\langle V^c, H^c\rangle$ is authentic with respect to the history root-hash that A is authenticating and $V^c \leq l-1$.

This is enough to detect violations of the quasi-disjoint-forking rule of Definition 5 (see the proof of Theorem 2) and of monotonicity. In Section 7.7, we provide further details about management of Γ and Π so that storage is kept bounded and both server and clients always store the needed information to perform the above operations.

We now formally prove some fundamental properties of the history-integrity protocol.

Property 4 (Monotonicity)**.** *In an execution of the history-integrity protocol in which no trusted entity detects any tampering, the sequence of update operations seen by each client c monotonically grows.*

Proof. Note that, by construction and by security of the cryptographic hash, R_l is uniquely associated with a sequence of history-pairs and, in turn, to a sequence of updates. When a client c receives, from the server, a response based on the history-pair $\langle l, H_l \rangle$, it checks its authenticity against authentication $A = [R_l]$. Let $\langle V^c, H^c \rangle$ be the local history-pair of c that the server associated with the above response. The client checks that $\langle V^c, H^c \rangle$ is authentic with respect to A and hence is on the same history of $\langle l, H_l \rangle$. If $V^c \leq l$, than the server declared a version of D which is equal or after that identified by V^c, respecting monotonicity. Analogous reasoning can be done for authenticators. They additionally authenticate a new version of Π with the new history-pair, but only if the monotonicity checks were successful. □

Lemma 2 (Commit-Correctness)**.** *In an execution of the history-integrity protocol in which no trusted entity detects any tampering, the sequence of events seen by each client c is commit-correct.*

Proof. Let σ be the sequence of the events of an execution of the history integrity protocol in which no trusted entity detects any tampering. Let π_c be the complete sequential permutation of the subsequence of σ seen by c. In the history-integrity protocol the evolution of data occurs at each commit. Each commit monotonically grows the history of root-hashes (see Property 4) currently seen by authenticators. We denote by χ_i the commits seen by c.

We note that the alternating structure of π_c mandated by Definition 1 is implied by the fact that commits deal only with updates and are atomic. Then, to prove that π_c is commit-correct, we have to prove that the three conditions of Definition 1 holds for the sequence χ_i.

Conditions 1 and 2 are verified since, each read has in its response the indication of the associated version of the history-hash. Further, c performs

verification, by checking proofs, that the returned value of the read is indeed associated with the current version declared by the server in the response or it is initial.

Concerning Condition 3, at each commit, the server proposes to the authenticator a sequence of updates. The authenticator checks that their order conforms to that specified by each client (by hash chaining) and that the server does not propose already processed updates (by checking past-hashes). Since in each authentication round the authenticator deals only with updates that have to be associated with the current commit, Condition 3 is verified.

□

Theorem 2 (Quasi-fork-linearisability for the history-integrity protocol)**.** *The history-integrity protocol emulates a key-value store on a Byzantine server with quasi-fork-linearisability in monotonic sense.*

Proof. We consider a generic execution of the history-integrity protocol, in which no trusted entity detects any tampering. The execution is represented by a real-time ordered sequence of events σ.

Recalling Definition 5, we should proof that, for all client c_i, the corresponding σ_i and π_i, chosen by the server, satisfy the four conditions of quasi-fork-linearisability and that each π_i grows monotonically over time. Monotonicity is stated by Property 4.

About Condition 1, client c_i receives $[R_l]$, signed by an authenticator, which also authenticates H_l. Hash H_l is uniquely associated with the sequence of updates seen by c_i which is π_i. The protocol mandates that the authenticator, which receives from the server proofs based on $[R_j]$, creates $[R_{j+1}]$ by adding, among all the others, all updates of c_i communicated by the server. The server can not skip or reorder any of them since they are hash-chained, and can not go back in time since past-hashes are checked. Hence, the authenticator provide a proof that they are contained in π_i. This is true for all authentication rounds. Since no trusted entity detects any tampering, all completed updates of c_i are contained in the last version of the dataset D_l seen by c_i. This proves Condition 1.

About Condition 2, by Lemma 2, all π_i are commit-correct. By Lemma 1, they preserve the real-time order of all non commuting operations in σ.

About Condition 3, for the updates this condition is enforced by an authenticator when it checks proofs and computes the new $[R_l]$ for the new D_l. For the read operations, this condition is enforced by the checks performed by

each client c_i. Clients check that the read result comes from D_l and that D_l is an updated version of the last version seen by c_i.

About Condition 4, if the server does not introduce any fork, this condition is trivially verified by $\sigma = \sigma_1 = \sigma_2$, $\pi_1 = \pi_2$. Let assume that the server does fork, and $\pi_1 = \alpha\beta_1$ and $\pi_2 = \alpha\beta_2$ be the sequential sequences seen by clients c_1 and c_2 as in Definition 4.

Without loss of generality, we consider the point of view of c_1. We now prove that all update invocations of c_1 in $\beta_1 \cap \beta_2$ are before the first response to c_1 in $\beta_1 \cup \beta_2$, in σ. By contradiction, we suppose that this is not true and prove that a tampering must be detected by a trusted entity. Let u be an update whose invocation is in $\beta_1 \cap \beta_2$ and, against Condition 4, let o be the last operation whose response is in $\beta_1 \cup \beta_2$ and $\text{res}(o) < \text{inv}(u)$. When c_1 receives $\text{res}(o)$, the history-pair associated with the version of the dataset on which $\text{res}(o)$ is based is pushed into local history-pairs Γ of c_1. By the way history-pairs are built, the last inserted element of Γ of c_1 uniquely identify one of the two branches, since $\beta_1 \neq \beta_2$ by definition. The server is showing to c_1 operations according to π_1, hence, at this point, the tail p of Γ is associated with $\alpha\beta_1$. Client c_1 prepares $\text{inv}(u)$ specifying p as history-pair and signing it with the whole u. When the server prepares the authentication request containing u, it cannot change p embedded in u. Since u is in both branches of the fork, two authentication requests containing u, one for each branch, are sent by the server to (possibly distinct) authenticators. The processing of the authentication request associated with $\alpha\beta_1$ completes successfully since p is proven to be authentic with respect to the history root-hash provided by the server. The processing of the authentication request associated with $\alpha\beta_2$ fails when trying to authenticate p in u against the history root-hash associated with $\alpha\beta_2$ which is not the one seen by c_1. Hence, the tampering is detected. □

Note that in the above description authenticators are not linked to a branch. In fact, since they are stateless, they have no mean to detect they are used by the server to authenticate commits for distinct branches. This does not have any impact on quasi-fork-linearisability from the point of view of the clients. However, nothing prevents to equip authenticator with a state similar to that of the clients to allow them to perform similar checks. In this way, the number of branches that the server could possibly create would be bounded by the number of available authenticators.

Corollary 1 (No False Negatives for the history-integrity protocol)**.** *In the history-integrity protocol, whenever a trusted entity detect a tampering, the*

server deviated from the quasi-fork-linearisability behaviour.

Corollary 1 directly derives from Theorem 2.

Theorem 3 (No false positives for the history-integrity protocol)**.** *In the history-integrity protocol, whenever the server behaves according to quasi-fork-linearisability, trusted entities do not detect any tampering.*

Proof. We assume correct implementation of ADSs and cryptographic primitives. Tampering is detected by trusted entities when one of the checks they perform fails. All trusted entities checks that the version declared in each message form the server is in the same history of their locally stored history-pair (monotonicity). Clients check correctness of replies against last root-hash authentication (correct operation execution). Authenticators check that for each client c each $u^c(i)$ is correctly hash-chained to the one before or with current past-hash η^c and check the authenticity of η^c for all c (server does not reorder updates). Authenticators check the authenticity of previous values v of k (correct operation execution). Authenticators check the history-pair p^c contained in the last $u^c(i)$ specified in the sequence for c is in the history of the current history root-hash (Condition 4 of Definition 5). All these checks are successful if server and all authenticators behaved correctly till that moment, which is true by hypotheses. □

Limiting the storage needed by server and clients

According to the above description, the server should store Π, each client should store its Γ and these data structures grow over time. For datasets that last long and change frequently, this is an overwhelming burden. We now show how to bound the storage taken by Π and Γ.

The server keeps a mapping P from each client c to a queue of history-pairs. We denote $P(c)$ the queue associated with c. An history-pair $p = \langle l, H_l \rangle$ is pushed into $P(c)$ when a response to c is sent with the version l, unless $P(c)$.tail is already equal to p. Hence, in queues $P(c)$, history-pairs are stored, from head to tail, in ascending order of version. A client c that receives a response containing p pushes it into Γ, unless Γ.tail is already equal to p. Hence, also Γ stores history-pairs in ascending order of version. When c sends an update invocation, it uses $p = \Gamma$.tail as history-pair in the construction of the update. When c sends a read invocation, it additionally specifies p in the message. When the server receives a read invocation with p, or puts an update containing p into an authentication request, it pulls from $P(c)$ all history-pairs

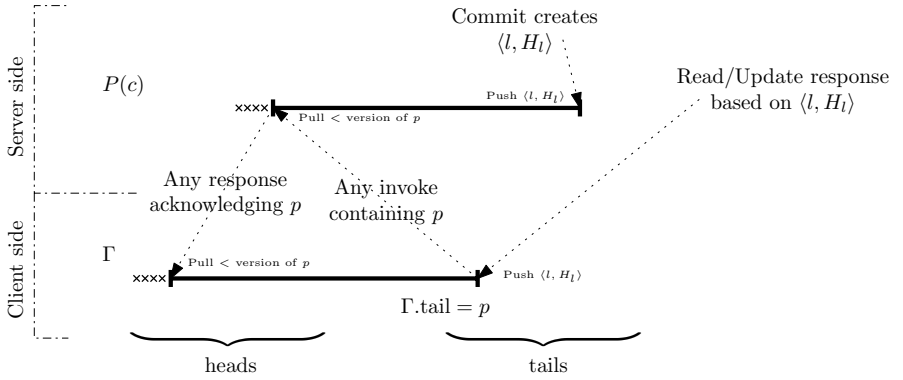

Figure 7.11: Relationship between messages and changes applied to $P(c)$ and Γ.

with a version less then that of p, so that $P(c)$.head $= p$. When the server sends a response to c, it adds $P(c)$.head as acknowledgement of its reception, piggybacked. When c receives a response that acknowledges p, it pulls from Γ all history-pairs with a version less then that of p, so that Γ.head $= p$. The way messages affect $P(c)$ and Γ is summarised in Figure 7.11.

The server exploits the pruning feature of ADSs (see Section 7.2), keeping in Π only history-pairs that are mentioned at least once in any queue $P(c)$ for any c (see Figure 7.12). Note that, pruning does not change the current history root-hash, it just reduces the memory occupation.

This scheme ensures that (i) each time a client c uses p as history-pair in an update, the server can provide proof(Π, p) since p is not pruned, (ii) each time c receives a response with acknowledge p, c has p in Γ (if the server has not forked) hence checks for monotonicity and quasi-fork-linearisability work, (iii) the size of each queue $P(c)$ is bounded by the number of commits involving updates from c that are sent in the time of serving one update, and (iv) the same bound holds for Γ.

Pruning does not make proofs shorter, this means that their length is $O(\log l)$, for a typical ADS, which might be not acceptable for datasets that are updated regularly and must last long. For simplicity of explanation, we suppose Π is realised with a binary MHT. If a client c regularly performs queries, the version V^c of $P(c)$.head is close to l. If this is true for all clients, the left subtree of the root in Π is completely pruned and we can substitute its current

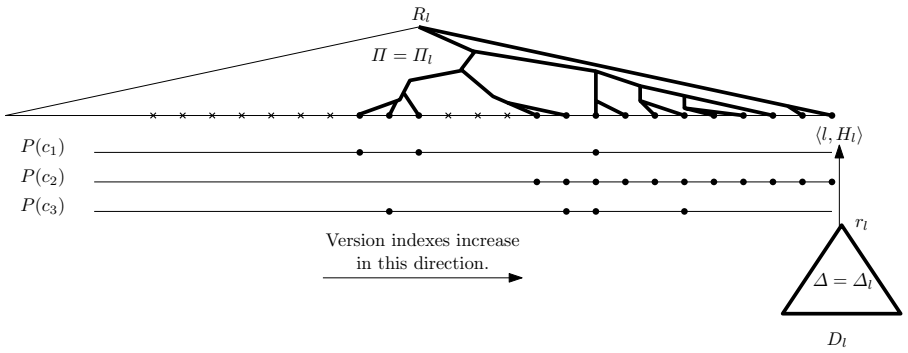

Figure 7.12: An example of pruning. Dots represent unpruned versions, while crosses represents pruned ones. Pruning of a history-pair from Π occurs when no $P(c)$ contains it.

root with its right child shortening the length of the proofs. While this makes the implementation of server and authenticators a bit more complicated, it allows us to have the proofs derived from Π of size $O(\log(l - \min_c V^c))$. Size of the proofs can be kept bounded if we "detach" clients that are stale for a number of commits greater than a fixed threshold.

7.8 The Pipeline-Integrity Protocol

This section presents a protocol, which we call *pipeline-integrity* protocol, that is scalable and achieves quasi-fork-linearisability. Essentially, we prove that results shown in Section 7.6 and those shown in Section 7.7 can be combined. We just describe the specificities of the use of the two approaches together. The complete pseudocode for the resulting pipeline-integrity protocol is provided in the Appendix.

In the following description of the protocol, we reuse many concepts and assumptions introduced in the simplified version (Section 7.6). Namely, we assume

- to have q authenticators,
- to have negligible execution time on server and authenticators,
- to have reliable and synchronous communications,

- to send an authentication request to authenticator a when an authentication reply is received from a, and by synchronous operation this occur every $\Delta t = 2d/q$,
- to have a pipeline-like interaction scheme between server and authenticators, and
- to have a server with U/R-data-structures, where an R-data-structure tracks the corresponding U-data-structure with a delay of $q\Delta t$.

We also reuse the same notation introduced in Section 7.6 to distinguish U/R-data-structures, with superscripts ^{R} and ^{U}, and for denoting instances of server variables between instants t_j and t_{j+1}, like D_j^U.

From the history-integrity protocol (Section 7.7), we reuse

- the concepts of history-hash $H_j = \text{hash}(H_{j-1}|r_j)$ and history-pairs $\langle j, H_j \rangle$,
- the fact that each client c keeps a queue Γ of local history-pairs that it is aware of.
- the content of the update operations and their notation: $u^c(i) = \langle k_i, v_i, \text{hash}(u(i-1)), i, \langle V^c, H^c \rangle \rangle$
- the use of the mapping $P(c)$ to track, on the server, the last history-pairs sent by the server to each client c,
- the use of pruned authenticated data structures on the sequence of history-pairs that, for the pipeline-integrity protocol, are two: Π^R and Π^U, and
- the notation for history root-hash R_j, that we use to denote the root-hash of Π^U between instants t_j and t_{j+1}.

While we refer the reader to the proper section for an explanation of the above concepts, we now explicitly describe the parts that need specific explanation.

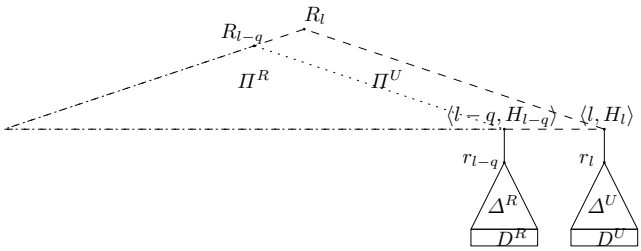

Figure 7.13: A scheme showing the data structures involved in the (complete) pipeline-integrity protocol.

Variable	Description
$D^R, D^U, \Delta^R, \Delta^U, l, Q^c, \Omega$	See Table 7.1.
Π^U	ADS on the sequence of history-pairs up to $\langle l, H_l \rangle$, used in conjunction with D^U and Δ^U. The root-hash of Π^U_j is denoted R_j.
Π^R	ADS on the sequence of history-pairs up to $\langle l-q, H_{l-q} \rangle$ used in conjunction with D^R and Δ^R. Its root-hash is R_{l-q}.
P	A mapping from each client c to a queue of history-pairs. The queue associated to c is denoted $P(c)$.
A	Authentication for D^R, Δ^R and Π^R in the form $[R_{l-2q}]\,[R_{l-2q}, R_{l-2q+1}]$ $[R_{l-q-2}, R_{l-q-1}]\ldots\,[R_{l-q-1}, R_{l-q}]$.
$\overline{A}$	Sequence of the root-hashes R_{l-2q}, $R_{l-2q+1}, \ldots,$ R_{l-q} that are the basis of A.

Table 7.4: Status of the server for the (complete) pipeline-integrity protocol.

Status Table 7.4 summarises the variables that form the status of the server. Many variables are the same as in the simplified version, in particular l is the index of the last instant in which the server received an authentication reply (and sent an authentication request). Consider the sequence of history pairs $\langle 0, H_0\rangle, \ldots, \langle l, H_l\rangle$. The server keeps two history ADS, denoted Π^R and Π^U, on this sequence. Structure Π^U is over the whole sequence and its history root-hash is denoted by R_l. Structure Π^U is used with D^U and Δ^U to compute proofs for a new authentication request ρ_l to be sent at instant t_l. Structure Π^R is limited up to index $l-q$ and its history root-hash is R_{l-q}, since $\Pi^R_l = \Pi^U_{l-q}$. Structure Π^R is used with D^R and Δ^R to reply to read requests. Figure 7.13, pictorially shows the relationships among all the data structures. Structures Π^R and Π^U are kept pruned by using the mapping P as explained in Section 7.7.

The server keeps a chained authentication A for the history root-hash R_{l-q} in this form: $\underline{[}R_{l-2q}]\ [R_{l-2q}, R_{l-2q+1}]\ [R_{l-2q+1}, R_{l-2q+2}]\ldots\ [R_{l-q-1}, R_{l-q}]$. The sequence $\overline{A}$ of the history root-hash involved in A is kept as well.

Messages The messages follow the scheme of the simple pipeline-integrity protocol with the following changes.

- Whenever a (signature of a) root-hash is specified for the simplified version, a corresponding (signature of a) history root-hash is specified for the complete version.
- In responses from server to clients, proofs of the kind $\text{proof}(\Delta^R, \cdot)$ in a message for the simplified version are substituted with quadruple containing $\text{proof}(\Delta^R_l, \cdot)$, H_{l-q-1}, $l-q$, $\text{proof}(\Pi^R_l, \langle l-q, H_{l-q}\rangle)$. Additionally, each message sent to client c contains the head of $P(c)$, stored by the server, denoted by $\langle V^c, H^c\rangle$, and its $\text{proof}(\Pi^R_l, \langle V^c, H^c\rangle)$.
- For authentication requests, updates are represented as $u^c(i) = \langle k_i, v_i, \text{hash}(u(i-1)), i, \langle V^c, H^c\rangle\rangle$ (see Section 7.7). In each request to authenticator a, H_{l-2}, l and $\text{proof}(\Pi^U_{l-1}, \langle l-1, H_{l-1}\rangle)$ are additionally sent. For each client c involved in the updates in the request, the following are included: $\langle \kappa^c, \eta^c\rangle$, $\text{proof}(\Delta^U_{l-1}, \langle \kappa^c, \eta^c\rangle)$, p^c, $\text{proof}(\Pi^U_{l-1}, p^c)$ where p^c is the history-pair $\langle V^c, H^c\rangle$ in the last update of c included in the request.

Behaviour Clients behave the same as in the history-integrity protocol.

The server executes both the behaviour specified for the history-integrity protocol and for the simplified pipeline-integrity protocol as follows. When

handling the authentication reply ρ^{rpl}, to a request ρ, from a, $P(a)$ is updated, D^R, Δ^R and Π^R are updated according to the updates specified in ρ, as in the history-integrity protocol, and A and $\overline{A}$ are updated, as in the simplified pipeline-integrity protocol. When creating the new authentication request ρ' to be sent to a, the sequence of updates is created as in the history-integrity protocol, and proofs needed to create ρ' are collected. Then, D^U, Δ^U and Π^U are updated to be ready for the next authentication request. Finally, pruning is done on both Π^R and Π^U based on the content of P.

Concerning authenticators, when authentication request ρ is received, compaction of chained authentication A contained in ρ is executed as in Algorithm 2, but substituting regular root-hashes with history root-hashes. About the authentication of update operations, the performed checks are the same as in the history-integrity protocol, but executed in a conditional manner. That is, no verification is performed against the authentication A communicated by the server, since this is late of q instants with respect to proofs. Instead, first the conditioning history root-hash $\overline{R}$ is computed from an arbitrarily chosen proof in ρ. Then, all checks are performed against $\overline{R}$ to verify coherency of ρ. If all checks are successful, the new conditioned history root-hash $\widetilde{R}$ is computed on the basis of update operations in ρ and new past-hashes for each client. Then, $[\overline{R}, \widetilde{R}]$ is put into ρ^{rpl} along with the compacted version of A.

Theorem 4 (Quasi-fork-linearisability for the pipeline-integrity protocol)**.** *The pipeline-integrity protocol emulates a key-value store on a Byzantine server with quasi-fork-linearisability in monotonic sense.*

Proof. The proof of this theorem is a consequence of Theorem 2, of Properties 2 and 3, and of the following considerations.

Consider a generic execution of the pipeline-integrity protocol, in which no trusted entity detects any tampering. We call it $\mathcal{P}$. The execution is represented by a real-time ordered sequence of events σ. The server may decide to arbitrarily fork, hence, each client c_i sees a subsequence of events σ_i, executed according to a sequential permutation π_i of σ_i and a sequence χ_j^i of commits seen by c_i. Consider σ as an execution of the history-integrity protocol (see Section 7.7), that we call $\mathcal{H}$ and use Theorem 2 with the same choices of σ_i, and π_i to prove quasi-fork-linearisability. In $\mathcal{H}$, commits χ_j^i are instantaneous and are in one-to-one correspondence with the commits of $\mathcal{P}$. Association between operations and commits in $\mathcal{H}$ also mimic what happen in $\mathcal{P}$. In $\mathcal{H}$, all update operations are executed not before q commits. This might sound weird for a real non-pipelined system, but it conforms to the interaction scheme introduced

in Section 7.3 and it is compatible with the history-integrity protocol. Hence, Theorem 2 applies. Note that, the checks performed by trusted entities in $\mathcal{H}$ and $\mathcal{P}$ are the same, with the only difference that in $\mathcal{P}$ proofs are verified against chained authentications. The second notable difference is that authenticators in $\mathcal{P}$ provide a conditional authentication not a plain one. However, in our synchronous and reliable execution, after q instants, the chain is complete and, by Properties 2 and 3, what was conditionally verified q instants before is verified unconditionally at current instant. □

Corollary 2 directly derives from Theorem 4.

Corollary 2 (No false negatives for the pipeline-integrity protocol)**.** *In the pipeline-integrity protocol, whenever a trusted entity detect a tampering, the server deviated from the quasi-fork-linearisability behaviour.*

Theorem 5 (No false positives for the pipeline-integrity protocol)**.** *In the pipeline-integrity protocol, whenever the server behaves according to quasi-fork-linearisability, trusted entities do not detect any tampering.*

Proof. The proof of this theorem is a consequence of Theorem 3, of Properties 2 and 3, and of the following considerations.

We assume correct implementation of ADSs and cryptographic primitives. Tampering is detected by trusted entities when one of their checks fails. Each check that is performed by trusted entities in the pipeline-integrity protocol matches one in the history integrity protocol, with the exception of the checks introduced by the verification of the coherency of the chained authentication. The following differences should be considered. In the pipeline-integrity protocol, clients perform their checks against a chained authentication of the history root-hash instead of against a plain authentication. The semantics equivalence of the two approaches is stated by Properties 2 and 3 and, for clients, the statement holds by Theorem 3. Authenticators do not perform real checks, but assume that a certain history root-hash $\bar{R}$ is authenticated and perform their checks against it. Then, they state the authenticity of the new history root-hash $\widetilde{R}$, after updates application, conditioned to the authenticity of $\bar{R}$. These conditioned authentication will form a chained authentication whose coherency is checked by trusted entities (see Algorithm 1) when they are received from the server. All these checks are successful if server and all authenticators behaved correctly till that moment, which is true by hypotheses. □

Concerning scalability, we can follow the same reasoning of the scalability analysis shown in Section 7.6, considering non-negligible computation and transmission time. The following holds.

Theorem 6 (Scalability)**.** *In the pipeline-integrity protocol, there exists a value of the number of authenticators q for which the response time for each authentication request is bounded by $4\beta\alpha\lambda + 2\gamma$ for λ large enough, where α is the total processing/transmission time for one update, β is the total processing/transmission time for one conditional authentication, and γ is the remaining processing/transmission time and network delay in a round that does not depend on the number of updates in the request or q.*

Proof. The statement is a direct consequence of Theorem 1 and of the fact that the pipeline-integrity protocol follows the same interaction scheme of the simplified pipeline-integrity protocol. □

7.9 Dealing with Non-Ideal Resources

In practice, assuming that the pipeline-integrity protocol can run synchronously is unrealistic for most applications, because real networks may suffer packet losses and congestion. Further, performances of machines may vary depending on other processes. First, we observe that we can adopt a reliable transport protocol, like TCP, that implements acknowledgements and retransmissions. It allows the two parties to send an amount of non-acknowledged data that is enough to saturate the (estimated) bandwidth of the channel. This permits us to deal only with uncertainty about network delays or with the (un)availability of an authenticator. For a practical realization of the pipeline-integrity protocol, we suggest to decouple sending of authentication requests from the reception of authentication replies. The easiest approach is to send authentication requests at scheduled instants or when the number of accumulated updates reaches a certain threshold, but other policies might be adopted to reduce response time. At the receiving of an authentication reply, if this was the first outstanding request in the Ω queue, the regular processing is performed. If this is not the case, an intermediate authentication response was missed. This might be late or the corresponding authenticator might be unavailable or unreachable. We can wait a timeout and possibly re-send the same request to another authenticator. During the handling of a missing request, the read operations are served against the last D^R and processing of updates goes ahead, but authentication requests and replies should be buffered till the missing piece will

arrive. Application specific considerations should lead the decision on which architecture elements should play the role of authenticators. This choice should depend not only on their computational power but also on the probability to become unreachable and on the fact that this is not a problem if at the same time also clients get disconnected. To reduce the probability of missing an authentication reply, more than one authenticator might be involved for each authentication round. A flow control should also be realised so that the server could communicate to clients that it cannot go ahead with updates (or can only at a slow pace) due to non responding authenticators. In this way, the clients can limit or stop updates and possibly informing the user of the problem.

In a real environment, clients may be abruptly disconnected or switched off. In this case, some response messages may be lost, even when a reliable transport protocol like TCP is used. In general, when the client reconnects, the server does not know which is the last version it is aware of since it does not know which is the last response messages the client processed. However, the protocol described in Section 7.7 is tolerant with respect to this problem, provided that client can keep content of Γ across reconnection, for example, by persisting it.

7.10 Experimental Study

We developed a prototype with the intent to provide experimental evidence of the feasibility of our approach in a realistic setting. Since our contribution provides means to scale the trusted side, we designed our experiments so that the bottleneck of the whole system is always the computing power dedicated to authenticators. We note that the average latency (i.e., the time taken for an update request to be authenticated) is proportional to the duration of one authentication round. We measured the duration of one authentication round and other parameters at increasing update invocation rate. We measured the maximum update invocation rate of that the system can sustain (i.e., its *throughput*) with different numbers of CPUs dedicated to run authenticators.

We note that authenticators may be implemented, as processes, threads or *actors* (see for example the *Akka actors* [Gup12]) which essentially allow us to increase their number without substantial additional cost. Clearly, their overall speed is bounded by the number of available physical processors. We denote by $\overline{q}$ the number of physical processors dedicated to execute authenticators, by t_A the time each of them takes to process one update (ignoring all other contributions for simplicity), and by λ the update arrival frequency. For a well

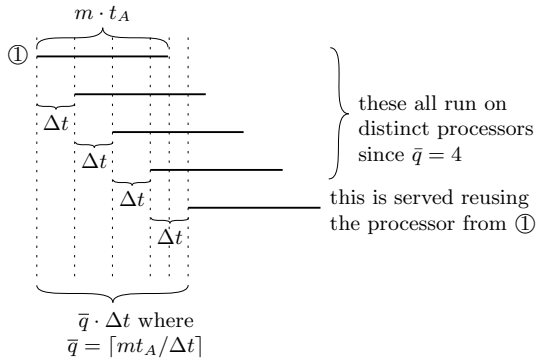

Figure 7.14: Construction to prove that $\bar{q} \geq \lceil m t_A / \Delta t \rceil = \lceil \lambda t_A \rceil$ is enough to ensure that each request has a free processor for it. In the example $\bar{q} = 4$.

dimensioned system it should be $\bar{q} \geq \lceil m t_A / \Delta t \rceil = \lceil \lambda t_A \rceil$ (see Figure 7.14). In this way, when a new authentication request arrives, there is always a free processor for it. Hence we expect a linear relationship between throughput and number of CPUs. The objectives of our experiments are the following.

O1. We intend to show that our approach supports update frequencies, up to the saturation of the CPUs of the authenticators, keeping the response time practically constant.

O2. We intend to show that it is possible to increase the throughput of the system by increasing the number of processors dedicated to execute authenticators.

O3. We intend to show that it is possible to obtain a minimum response time, as described in Section 7.6, by tuning the number of updates contained in each authentication request.

Our experiments focus on update operations, since read operations are only marginally affected by the introduction of the pipeline-integrity protocol. Our prototype is explicitly targeted to this experimentation. Providing a fully fledged implementation was not among our objectives. We implemented the simplified pipeline-integrity protocol described in Section 7.6, supporting read operations and update operations only for keys already existing in the dataset. We note that the scheme of client-server interaction and server-authenticator

interaction in the simplified and in the complete versions of the protocol is the same. The most notable difference between the two versions of the protocol is that in the simplified version no history ADS is kept, so proofs do not encompass the integrity of the history. This means that the number of cryptographic hashes that all machines have to compute is less than in the case of the complete version (see Section 7.7) and the length of messages is correspondingly shorter. However, regarding the management, transmission and check of the chained authentication, which is the core of our approach, there is no relevant difference.

We performed our tests with three machines: S running our server, C running a number of clients that perform update requests to S and A running authenticators to serve authentication requests sent by S. All machines are Linux-based. Machines A and C were located at the Computer Network Research Laboratory at Roma Tre University. Machine A was an old machine with 4 CPUs at 2.4GHz. Machine S was a large server (64 CPUs) located in a cloud that is operated by Consortium GARR, which is also the connectivity provider of the Roma Tre University. Round-trip delay between S and A was about 20ms. The bandwidth between the laboratory and S is large enough to rule out any congestion, however, due to an old 100Mbit ethernet switch near to A, the bandwidth between S and A turned out to be 10.5Mbyte/sec[2]. Since this bandwidth is quite small, to avoid the risk for the network to be the bottleneck of our experiments, we **artificially increased the CPU consumption** on A of each authentication round by performing additional dummy cryptographic operations. In all tests, we also checked that the server was not a bottleneck[3]. Section 7.9 deals with a number of non ideal aspects of real systems. Most of them turned out to be irrelevant in the controlled environment of our experiments. Our software is based on the Akka library realizing the actor model [Gup12]. The communication among machines takes advantage of the Akka proprietary protocol (TCP-based). Since this protocol is designed to work in LAN, we set up a tunnel between our laboratory and S[4]. Our software runs on the Java Virtual Machine. Since the JVM incrementally compiles the code according to the "HotSpot" approach [PVC01], for each run we took care to let the system run long enough before performing the measurements, to be sure that compilation activity was over. Concerning the ADS, we realized

[2]The measurement was performed by the standard iperf tool (`https://iperf.fr/`).

[3]The measurement of the consumption of system resources was performed by the standard dstat tool (`http://dag.wiee.rs/home-made/dstat/`).

[4]The tunnel was realized by the socat tool (`http://www.dest-unreach.org/socat/`).

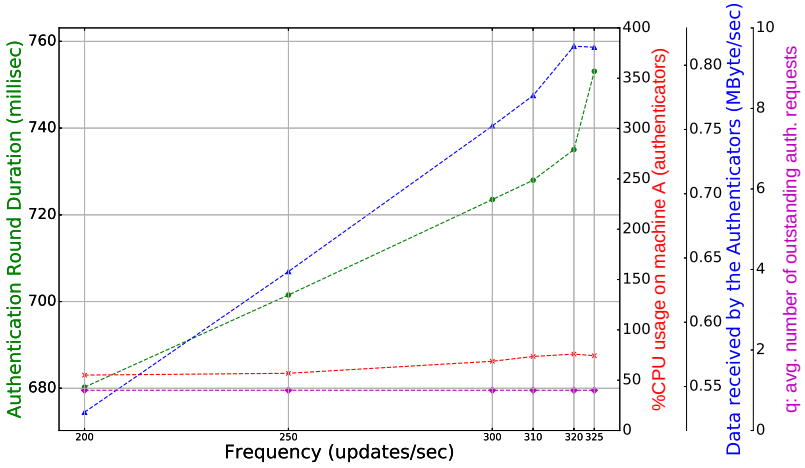

Figure 7.15: Performance of the blocking approach described in Section 7.4. Four CPUs are dedicated to authentication on machine A, but at most one authentication request can be outstanding.

an Authenticated Skip list containing 100000 keys [GTS01]. This structure is created and randomly initialised at the start up of the system.

We first measured the performances of the blocking approach (see Section 7.4),

which we realised using the same software imposing $q = 1$, where q is the number of outstanding authentication requests. Due to this constraint, in this case, we always wait for the arrival of the authentication reply and send, in the next request, all updates arrived in the meantime, which is larger than 200, much like in the description given in Section 7.4. Figure 7.15 shows how several parameters changes (y-axes) when the frequency of updates arrivals (x-axis) increases. We observe that near 325 upd/sec the round duration increases steeply, and above that frequency, the messages grow so big that they can hardly be handled without errors. Hence, the throughput of this setting is about 325 upd/sec.

To meet Objectives O1 and O2, we measured the performance of the simplified pipeline-integrity protocol described in Section 7.6 with an increasing

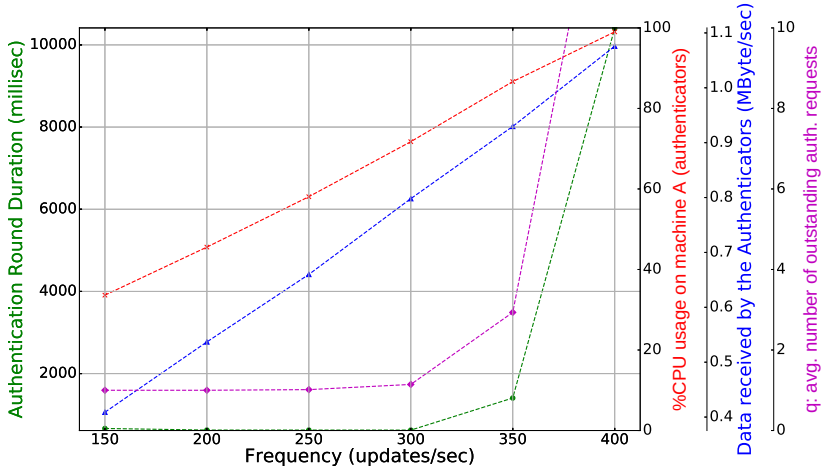

Figure 7.16: Performance of the simplified pipeline-integrity protocol when authenticators are limited to use **1 CPU**. Throughput is between 350 and 400 upd/sec. Latency is 622–662msec.

number of CPUs on the authenticator side[5]. We let q to increase freely when some authentication requests were still outstanding. We increase the frequency of the updates until the allocated CPUs of A reach saturation. We decided to send authentication requests containing $m = 200$ updates. That is, we send an authentication request at each 200-th update invocation received. The results are shown in Figures 7.16, 7.17, 7.18 and 7.19. The x-axis shows the update invocation rate. The y-axes show the authentication round duration, the percentage of the CPU used by Machine A (400% means that all 4 CPUs are fully used), the data received by Machine A, and the average value for the outstanding authentication requests q. The tests show that the round duration is largely constant until the allocated CPUs are close to saturation (see Objective O1) and when q increases, as in Figure 7.19, even slightly decreases. We note that, **adding CPUs increases the throughput but has no detrimental effect on latency**, which was the scalability objective that was stated

[5]The limit on the number of CPUs used is imposed by using the CPU affinity setting provided by Linux.

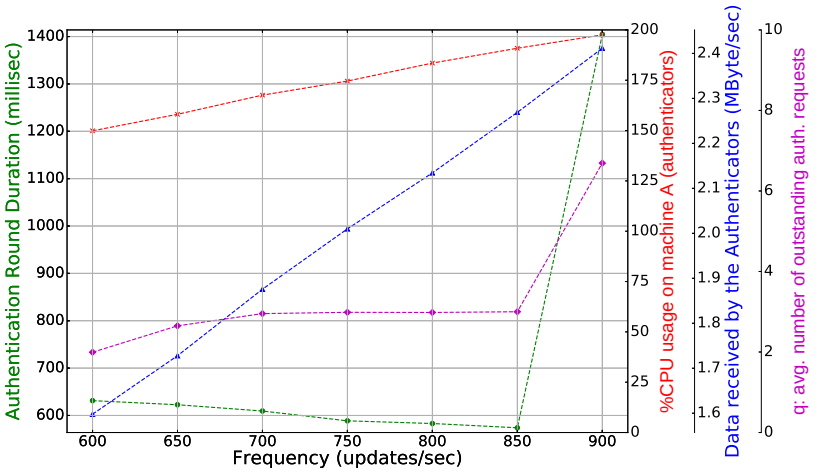

Figure 7.17: Performance of the simplified pipeline-integrity protocol when authenticators are limited to use **2 CPUs**. Throughput is between 850 and 900 upd/sec. Latency is 580–631msec.

in Section 7.3. We also note that, for one CPU, the throughput is similar to the one for the blocking approach. In addition, we obtain the advantage that, for update rate close to the throughput, adopting the pipelining approach allows the (average) value of q to increase making the system much more stable than for the blocking approach. For each experiment, we visually identified the consecutive frequencies close to the frequency where CPUs of machine A saturate. We considered this pair of frequencies an approximation of the throughput of the system. Figure 7.20 summarises the measurements of the throughput of all four experiments. The x-axis shows the number of CPUs of Machine A that authenticators are allowed to use. The y-axis shows the throughput of the system deduced from the inspection of Figures 7.16, 7.17, 7.18 and 7.19. The chart shows that the throughput linearly increases with the number of CPUs used by the authenticator, as expected (see Objective O2).

To meet Objective O3, we performed an experiment to show the relationship between m, q and the round duration T. Since in our software q is automatically adjusted, to increase q we decreased m and computed an average value of q as $(\lambda T)/m$. In this experiment, we removed all additional dummy crypto-

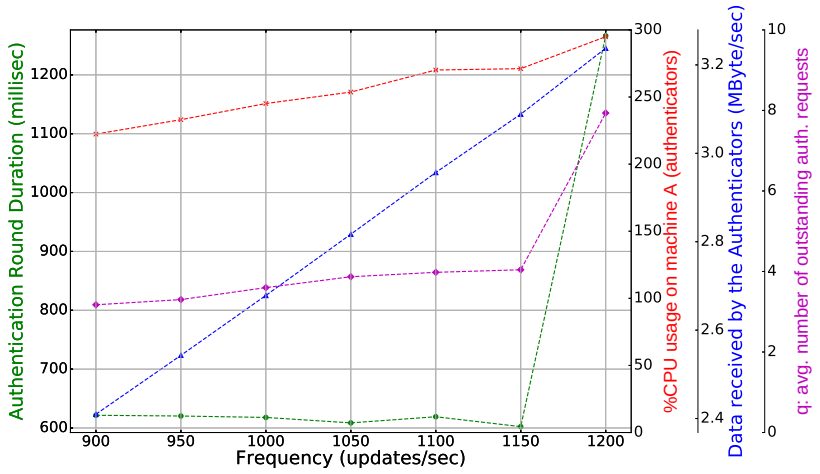

Figure 7.18: Performance of the simplified pipeline-integrity protocol when authenticators are limited to use **3 CPUs**. Throughput is between 1150 and 1200 upd/sec. Latency is 602–621msec.

graphic operations introduced for the previous experiments and fixed $\lambda = 500$. This value ensured that we never reached any bottleneck of the system during the test. In this experiment, we allowed all four CPUs to work. The result is depicted in Figure 7.21. The x-axis shows the average number of outstanding requests q. The y-axis shows the duration of the authentication round. Each point is labelled with the average value of updates in one authentication request. The chart shows that the round duration decrease until $m = 35$, where $q_{\min} \cong 2.21$ and $T_{\min} \cong 154.97$. After $q_{\min}$, T rises with small slope as predicted by the theory (see Section 7.6).

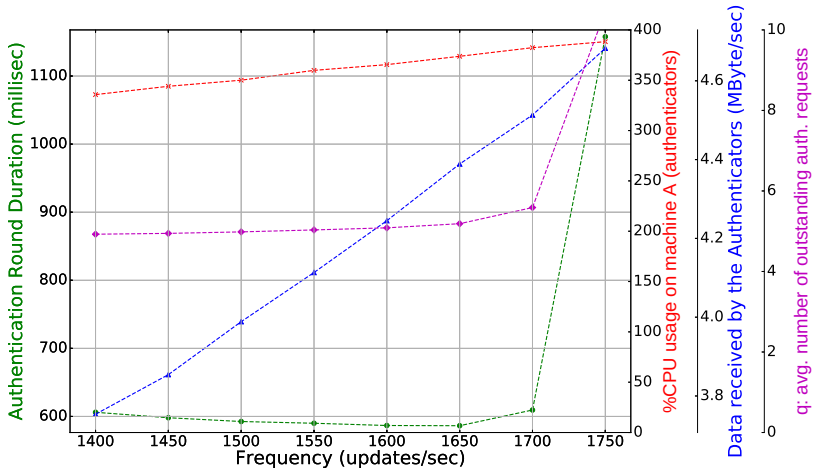

Figure 7.19: Performance of the simplified pipeline-integrity protocol when authenticators are limited to use **4 CPUs**. Throughput is between 1700 and 1750 upd/sec. Latency is 588–616msec.

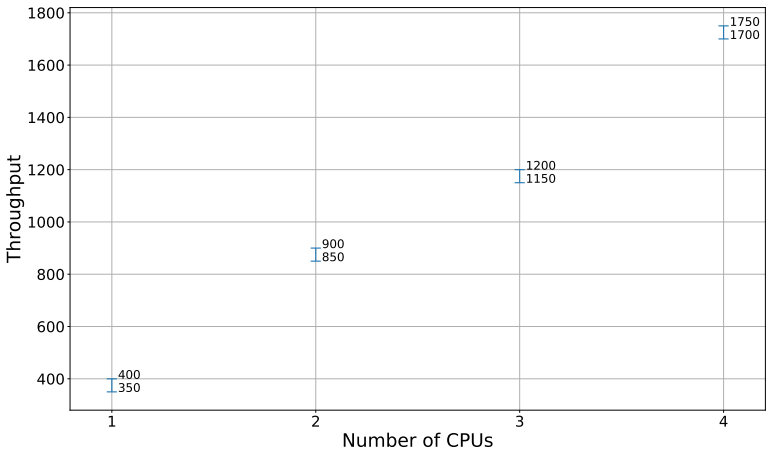

Figure 7.20: Relationship between the number of CPUs available for the authenticators and throughput of the system.

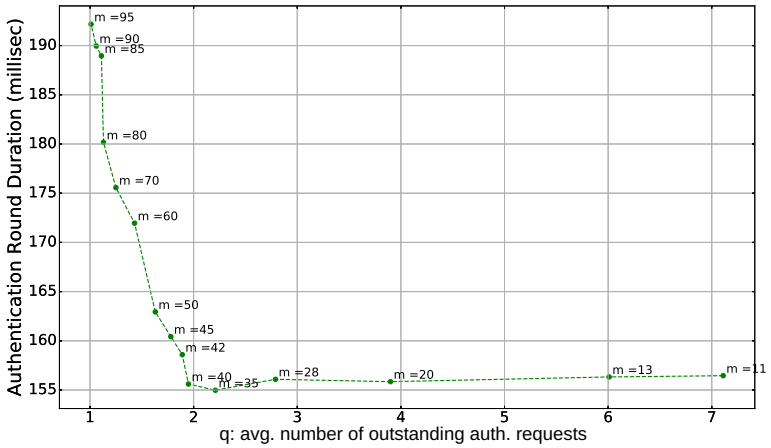

Figure 7.21: Relationship between authentication round duration, number of update in an authentication request (m), and outstanding authentication requests (q).

Chapter 8

Conclusions

Industrial control systems integrate hardware, software, and connectivities to support critical infrastructure like energy production and distribution, water distribution, and transportation. The architecture of these systems has been changing in the last years to increase flexibility and efficiency. A growth of the interconnectivity and integration of IT-typical components within the ICSs has been observed. This introduces new risks that can be challenging to mitigate in ICSs which are built to provide high levels of availability, safety, and reliability but without cybersecurity in mind.

Industrial Control Systems are sensible targets for high profile attacks, which are able to circumvent traditional protection methods like antiviruses. Nowadays, the main attackers are criminal organizations and countries, like foreign intelligence, with a lot of capital to invest in this activities. Cyber-attacks against ICSs represent a serious risk for the society and can be considered a new weapon for opposite countries and terrorists.

In this research work, we addressed the issue of security of industrial control systems proposing techniques that provide an effective way to deal with sophisticated attacks, which would mostly go unnoticed by conventional security countermeasures.

In the Chapter 2, we have introduced a model for protecting critical machines in an ICS context from attacks that use removable storage devices like USB thumb drives. Our architecture realises the proposed security model and fulfils a certain number of requirements that arise in the ICS context. The pro-

posed approach makes very limited assumptions on the nature of the attack, it proactively blocks attacks before they reach the target, and does not need any special support by already deployed products.

In the Chapter 3, we presented USBCheckIn, a hardware that realises a new approach to protect hosts from BadUSB attacks in which generic USB devices maliciously mimic the behaviour of HID devices. Our proposal does not relies on decisions of the users and it is compatible with any host or device. Our first experiments with our prototypical implementation show that the presence of USBCheckIn has no impact on HID devices responsiveness (e.g., mouse polling frequency is the same) and CPU consumption is unnoticeable. Future plans encompass performing deep performance tests and a formal user study for assessing the usability in real environments.

In the Chapter 4, we presented an architecture based on a dedicated hardware for protecting critical machines in a ICS against malware spread by means of USB devices. USBCaptchaIn realises a new approach to protect hosts, both against BadUSB attacks, in which generic USB devices maliciously mimic the behaviour of HID devices, and against malware embedded in data stored in RSDs. The proposed approach proactively blocks attacks before they reach the target and does not rely on decision of the users. It is highly compatible with already deployed products, comprising embedded devices, like Programmable Logic Controllers, Remote Terminal Units, etc., on which is often not feasible to install new software. The user experience is only slightly altered by the protection offered by our solution and the presence of USBCaptchaIn does not impact USB devices responsiveness. Our informal contacts with experts, to whom we have shown our prototypes, have reported that a solution like the one we described might be accepted in a real ICS environment. Further research work may encompass the study of supporting a Biba security model with more than two security levels. This may be useful, for example, to support a split of the critical realm in a *production* realm (highly critical) and a *testing* realm (less critical). Further classes of HIDs may also be supported (e.g., touch tablet, track-ball, joystick, etc). The integration in USBCaptchaIn of a solution like the one described in [MW18] may add a form of protection against real keyboards with malicious firmware without affecting security and usability.

In the Chapter 5, we proposed a methodology and an architecture that enable flexible adoption of one IDS (or a few of them), while keeping the possibility to mirror any stream in the network and forward it toward the IDS

independently from its deployment location. While we think that our approach can be useful in many contexts, we tailored it for the usage within ICS networks, where most of the traffic flows are critical and known in advance, and occasional usage can be handled with a best effort approach. We base our work on SDN technology, which allowed us to keep a simple centrally managed network configuration. We presented several small-effort extensions to the basic description in Section 5.8. However, the integration of a distributed approach for the SDN-controller, like the one presented in [TG10], in our architecture, may be the subject of additional research. Further, in our solution, we statically assigned bandwidth to all critical streams, disregarding cases in which traffic is not stable over time. Better usage of the bandwidth could be achieved by taking this into account.

In the Chapter 6, we presented an overview of the results attained in the framework of the Preemptive European project, which provides innovative methods and tools to detect and protect ICSs from cyber-intrusion. We illustrated the Preemptive architecture and the tools functionalities, which perform detection at host, network, and process levels. The Context Aware Event Analysis tool correlates events raised by the different tools and sends them to the Humane-Machine Interface that make operators aware about the status of the system in terms of security. Thereby, the peculiarity of Preemptive is the capability of the proposed solution to correlate several events, which individually would appear to be irrelevant, allowing the detection of skilled attacks that otherwise may be hidden. The tools have been tested using real data provided by the End Users Advisory Board that supported the project over time. Our experimental results have shown the feasibility of an integrated approach to cyber-attacks detection and prevention that can greatly improve cyber-security awareness in the current ICS context. Concerning future research directions, we think that the needs in terms of cyber-security awareness will be shaped by the adoption of ICS models tailored to support Industry 4.0. In particular, technologies like cloud computing, Internet of Things, and artificial intelligence, along with the trend to distribute decisions and control, will provide formidable opportunities for malicious agents that are likely to require specific detection and prevention strategies.

In the Chapter 7, we presented the pipeline-integrity protocol, which enables the use of authenticated data structures to check integrity of outsourced key-value stores at untrusted servers in a setting in which many clients concurrently perform updates. We think that the pipeline-integrity protocol may

help the adoption of authenticated data structures in industrial systems where a large number of small devices may need to access and update integrity-critical data. The main features of our methodology are its scalability and its ability to detect server misbehaviour in the quasi-fork-linearisability consistency model. This consistency model is almost as strong as the one which was theoretically proven to be the strongest possible in that setting (fork-linearisability). Our approach is scalable in the sense that, it is possible to saturate the system bottleneck without substantial increase of the response time. We also support concurrent authentications, which makes much easier to solve bottlenecks on the trusted side. One notable aspect of our result is that scalability holds even when the round-trip delay between server and authenticators is large and update rates are high, which were commonly regarded as bad situations for adopting authenticated data structures. We proved practical performance with an experimental evaluation. There are several future research directions suggested by our work. On the theoretical side, one can ask if the quasi-fork-linearisability is the strongest feasible consistency model that scales, in the sense adopted in this chapter. Regarding the adoption in cloud facilities, we note that many of the technologies that underlie the cloud favour availability vs. consistency, according to the CAP/PACELC theorem [Aba12]. Since our approach detects consistency violations, it does not fit well with these technologies. This essentially limits the use of the pipelining-integrity protocol to situations where partitioning is ruled out by proper network configurations. On this side, one may ask if the detection of a fork may be recovered gracefully by authenticators. They may recognise the join as not harmful in many non malicious cases and resort to a human-based fix when there is no safe automatic merge approach. Further, one can ask if the techniques described in this chapter can be adopted in conjunction with a decentralised P2P network (like, for example, that described in [HNKÖ18]), where the P2P network is considered an untrusted storage. In this context, all nodes have to agree on the same sequence of updates in order to consistently reply to read invocations. Finally, it should be nice to have a publicly available library implementing the pipelining-integrity protocol to smooth its adoption into real applications.

Chapter 9

Acknowledgments

Let me write the acknowledgements in my mother tongue.

Scrivere i ringraziamenti è sempre una cosa difficile. L'ho lasciati volutamente alla fine di questo lavoro di tesi perchè, a mio avviso, dopo la testa è giusto lasciare lo spazio al cuore.

Prima di tutto vorrei ringraziare il mio advisor Maurizio Pizzonia per la fiducia e il sostegno che ha sempre dimostrato nei miei confronti ma soprattutto per la sua onestà intellettuale. Un grazie speciale a Diego per la preziosa collaborazione. Un grazie di cuore a tutto il gruppo di ricerca, per avermi dimostrato la loro vicinanza durante il mio percorso di dottorato.

A Federico, amico lontano ma sempre vicino. Le sua parole sono state immensamente preziose, mi hanno aiutato a stringere i denti ed andare avanti, sempre. Mi hanno aiutato a trovare il lato leggero delle cose.

A zio Tano e zia Emma, per avermi fatto sentire sempre a casa. Le chiacchierate estive con zio Tano sono state calde compagne negli inverni freddi.

A Kostis, Raffaele e Mimmo per le risate, le giornate spensierate e i discorsi profondi. Anche se sparsi nel mondo saranno sempre cari amici da portare nel cuore.

Grazie a tutte le persone che la vita mi ha fatto incontrare, buoni e cattivi perchè grazie a tutti loro ho avuto la possibilità di mettermi in gioco e crescere.

Provo una profonda gratitudine per i miei genitori che mi hanno donato la vita e a tutta la mia famiglia d'origine. In particolare per mio padre che con l'esempio e la sua umiltà mi ha insegnato il vero significato della dedizione, della passione e dell'onestà. Il suo insegnamento è sempre stato che in ogni

lavoro l'essere umano è la vera ricchezza.

Ho sempre creduto che senza radici forti non si può volare. Grazie papà per avermi permesso di essere figlio e, come tale, padre di mio figlio. Grazie per permettermi di volare.

Un grazie particolare ad Aniceto, che mi ha presentato me stesso, insegnato ad accogliere le mie ferite ed averne cura a tal punto che, quelle ferite, sono oggi la mia più grande forza. Grazie per avermi insegnato la legge del paradosso e a guardare la vita da una prospettiva tutta nuova.

Un caldo ringraziamento a mio figlio Samuele, alla sua vivacità e ai suoi occhi lucenti. È bello tornare bambini accanto a lui e lasciarsi contaggiare dal suo amore per la vita. Spero che tu possa trovare tutto quello di cui hai realmente bisogno. Ti auguro di non perderti mai.

Vorrei in realtà dedicare questo lavoro di tesi ad Ilaria, mia moglie, che mi ha permesso di specchiarmi nei suoi occhi profondi, porte di un mondo incredibile. Grazie per avermi compreso oltre le parole e avermi fatto sentire accolto nelle mie imperfezioni e stranezze. Dovunque ci porterà la vita, quello che provo con te e per te sarà sempre uno dei miei più grandi tesori. Grazie Ilaria per esserti fatta trovare da me.

Chapter 10

Appendix

Formal Description of the Pipeline-Integrity Protocol

Notation

For a detailed description, please refer to Sections 7.7 and 7.8.

Symbol	Description
k	A key in a dataset.
v	Value associated with a certain key in a dataset.
$\langle k, v \rangle$	Key-value pair in a dataset.
κ^c	A client-key for client c.
η^c	The past-hash associated with κ^c.
q	Number of authenticators and depth of the pipeline.
t_i	Instants at which the server receive authentication reply and sends authentication requests.
ρ_i, ρ_i^{rpl}	Authentication request (sent at instant t_i) and the corresponding reply (recevied at instant t_{i+q}).
l	Index of the last instant of time just executed by the server, which is denoted t_l.
D^R, D^U, Δ^R, Δ^U, Π^R, Π^U	Variables of the server containing readable and updated datasets, their ADS, and their history ADS. Their value changes at instants t_i. A subscritpt, like in D_i^R, denotes the value of that variable between t_i and t_{i+1}.
l, Q^c, Ω, P, A, $\overline{A}$	Other variables of the server.
r_i	Root-hash of Δ_i^U. The root-hash of Δ_i^R is r_{i-q}.
R_i	History root-hash at instant i, that is, root-hash of Π_i^U. The history root-hash for Π_i^R is R_{i-q}
$[x]_e$, $[x]$	Signature of data x performed by trusted entity e. If x is a root-hash, it is a plain authentication of x. The indication of e can be omittied if irrelevant.
l, Q^c, Ω, P, A, $\overline{A}$	Variables on the server.
H_i	History hash associated with D_i^U and Δ_i^U.
$\langle i, H_i \rangle$	History-pair related to version i.
$p^c = \langle V^c, H^c \rangle$	A history-pair known by client c
$\text{proof}(y, x)$	Proof of x against an instance of an authenticated data structure y (see Section 7.2).
$u^c(i)$	$i-th$ update invoked by client c, it is a shorthand for $\langle i^c, k_i^c, v_i^c, \text{hash}\left(u^c(i-1)\right), p^c \rangle$
Operations on queues	Standard semantic of queues is adopted with the following terminology. After *pushing* x into a queue Γ, $\Gamma.\text{tail} = x$. If $\Gamma.\text{head} = x$, *pulling* from Γ returns x.

State and Behaviour of the Server

Algorithm 5 Variables kept by the server to store its state.

l:	Index of the last instant in which the server performed a processing of authentication reply and sent an authentication request. That instant is denoted t_l.
D^R:	The current readable dataset. This dataset is authenticated by A. It holds that $D^R = D^R_l = D^U_{l-q}$.
D^U:	The dataset that is up-to-date with respect to update invocations for which an updated request was already sent. This dataset is not authenticated. It holds that $D^U = D^U_l = D^R_{l+q}$.
Q^c:	For each client c, a queue of updates with their signatures, that is, of pairs $\langle u^c(i), [u^c(i)]_c \rangle$ where $u^c(i) = \langle i, k^c_i, v^c_i, \mathrm{hash}\,(u^c(i-1)), p^c \rangle$. It contains updates that are waiting to be put into an authentication request.
Ω:	A queue, of length at most q, of outstanding authentication requests: $\rho_{l-1}, \ldots, \rho_{l-q}$ (from last to first).
P:	A mapping from each client c to a queue of history-pairs. The queue associated to c is denoted $P(c)$.
Δ^R:	The ADS related to D^R. It holds that $\Delta^R = \Delta^R_l = \Delta^U_{l-q}$.
Δ^U:	The ADS related to D^U. It hods that $\Delta^U = \Delta^U_l = \Delta^R_{l+q}$.
Π^U:	History ADS that contains history-pairs $\langle i, H_i \rangle$, with $i \leq l$, ordered by i. It holds that $\Pi^U = \Pi^U_l = \Pi^R_{l+q}$. Its root-hash is denoted R_l. It is stored pruned as described in Section 7.8.
Π^R:	History ADS that contains history-pairs $\langle i, H_i \rangle$, where $i \leq l-q$, ordered by i. It holds that $\Pi^R = \Pi^R_l = \Pi^U_{l-q}$. Its root-hash is denoted R_{l-q}. It is stored pruned as described in Section 7.8.
A:	The chained authentication for the current version of Π^R in the form $[R_{l-2q}]\ [R_{l-2q}, R_{l-2q+1}] \ldots [R_{l-q-1}, R_{l-q}]$.
$\overline{A}$:	The sequence of history root-hashes on which A is based: $R_{l-2q}, R_{l-2q+1}, R_{l-2q+2}, \ldots, R_{l-q-1}, R_{l-q}$.

Algorithm 6 Server behaviour. Pruning.

Require: A version index V and the identifier c of a client.
1: **while** $p = P(c)$.head has a version less than V **do**
2: Pull p from $P(c)$.
3: **if** version of p is not referred in P (for any client) **then**
4: Prune from $\Pi^R$and Π^U the entry for p and all unneeded parts.
5: **end if**
6: **end while**

Algorithm 7 Server behaviour. Processing of read and update invocations.

1: **upon receiving** an update invocation from client c **containing**
2: $u^c(i)$ and its signature $[u^c(i)]_c$
3:
4: **do**
5: Push $\langle u^c(i), [u^c(i)]_c \rangle$ into Q^c
6:
7: **upon receiving** a read invocation from client c **containing**
8: the key k to be read and $\langle V^c, H^c \rangle$.
9:
10: **do**
11: Execute Algorithm 6 on V^c and c to update $P(c)$, Π^R and Π^U.
12: **send** read response to c **containing**
13: the value v for k according to D^R
$l - q$ as the version index for this response
proof $\left(\Delta^R, \langle k, v \rangle\right)$, H_{l-q-1}, proof $\left(\Pi^R, \langle l - q, H_{l-q} \rangle\right)$
proof $\left(\Pi^R, \langle V^c, H^c \rangle\right)$
$\langle V^c, H^c \rangle$
A and $\overline{A}$, which authenticate all the above data
14: **if** $P(c)$.tail $\neq \langle l - q, H_{l-q} \rangle$ **then**
15: Push $\langle l - q, H_{l-q} \rangle$ into $P(c)$
16: **end if**
17:

Algorithm 8 Server behaviour. Reception of an authentication reply and update of R-data-structures.

1: ▷ The variable l, part of the state of the server, is incremented in the algorithm below in the first step. Below, the use of the symbol l always conforms to the use of l throughout the whole algorithm (i.e. after the increment).
2: **upon receiving** authentication reply $\rho^{\text{rpl}} = \rho^{\text{rpl}}_{l-q}$ from a **containing**
3: the identifier of the associated authentication request $\rho = \rho_{l-q}$
4: $[R_{l-2q}]_a$
5: $[R_{l-q-1}, R_{l-q}]_a$
6:
7: **do**
8: $l \leftarrow l + 1$
9: Get $\rho = \rho_{l-q}$ from Ω deleting it from the queue.
10: Let $\rho.B$ the sequence of updates contained in ρ.
11: For each client c involved in $\rho.B$, update value for $\langle \kappa^c, \eta^c \rangle$ in D^R and Δ^R, so that $\eta^c = \text{hash}\,(u^c(i_{\max}))$ where $i_{\max}$ is the index of the last update of c in $\rho.B$.
12: Apply all updates of $\rho.B$, in the specified order, to D^R and Δ^R.
13: ▷ Now, it holds $D^R = D^R_l$, $\Delta^R = \Delta^R_l$ and its root-hash is r_{l-q}.
14: Let $H_{l-q} = \text{hash}\,(H_{l-q-1}|r_{l-q})$
15: Add $\langle l-q, H_{l-q} \rangle$ to Π^R.
16: ▷ Now, it holds $\Pi^R = \Pi^R_l$ and its root-hash is R_{l-q}.
17: Update A by substituting the two leftmost elements with $[R_{l-2q}]_a$ and appending $[R_{l-q-1}, R_{l-q}]_a$ to its right. Update $\bar{A}$ accordingly.
18: For each c involved in $\rho.B$ **send** an update response to c **containing** A, $\overline{A}$, proof$(\Pi^R, P(c).\text{head})$, $P(c).\text{head}$, $l-q$, proof$(\Pi^R, \langle l-q, H_{l-q} \rangle)$.
19: For each c involved in $\rho.B$, if $P(c).\text{tail} \neq \langle l-q, H_{l-q} \rangle$ push $\langle l-q, H_{l-q} \rangle$ into $P(c)$.
20: Execute Algorithm 9 to send the authentication request ρ_l.
21:

Algorithm 9 Server behaviour. Preparation and sending of an authentication request ρ_l to authenticator a, and update of U-data-structures.

1: $C \leftarrow$ all clients c for which Q^c is not empty.
2: $\rho \leftarrow$ an empty authentication request
3: ▷ It holds that $\rho = \rho_l$. See Algorithm 10 for meaning of the fields of ρ.
4: $\rho.l \leftarrow l$
5: $\rho.A \leftarrow A$
6: $\rho.\overline{A} \leftarrow \overline{A}$
7: $\rho.B \leftarrow$ an empty sequence
8: **for all** c in C **do**
9: For all tuples in Q^c, add them to $\rho.B$ keeping their order.
10: make Q^c empty
11: **end for**
12: $\rho.\overline{\overline{H}} \leftarrow H_{l-2}$
13: ρ.mainproof $\leftarrow$ proof $\left(\Pi^U, \langle l-1, H_{l-1}\rangle\right)$ ▷ Note that Π^U has not been updated yet, while l was, hence it holds $\Pi^U = \Pi^U_{l-1}$. The same is true for Δ^U and D^U.
14: **for all** c involved in $\rho.B$ **do**
15: $\rho.p^c \leftarrow p^c$ as specified in the last update of c in $\rho.B$.
16: ρ.histproof$(c) \leftarrow$ proof $\left(\Pi^U, p^c\right)$, where p^c is as above.
17: ρ.pasthash$(c) \leftarrow \langle \kappa^c, \eta^c\rangle$ obtained from D^U
18: ρ.pasthashproof$(c) \leftarrow$ proof $\left(\Delta^U, \langle \kappa^c, \eta^c\rangle\right)$
19: Execute Algorithm 6 on the version of p^c and c to update $P(c)$, Π^R and Π^U.
20: **end for**
21: **for all** k involved in $\rho.B$ **do**
22: Let v be the value of k in D^U
23: ρ.oldval$(k) \leftarrow v$
24: ρ.proof$(k) \leftarrow$ proof $\left(\Delta^U_{l-1}, \langle k, v\rangle\right)$.
25: **end for**
26: **send** authentication request ρ to a.
27: Push ρ as last element of Ω.
28: Apply all updates of $\rho.B$ to D^U and Δ^U respecting their order in $\rho.B$.
29: For each c with an update in $\rho.B$, update $\langle \kappa^c, \eta^c\rangle$ in D^U and Δ^U, where $\eta^c = \text{hash}\left(u^c(i_{\max})\right)$ and $i_{\max}$ the index of the last update of c in $\rho.B$.
30: Let r_l be the root-hash of Δ^U and $H_l = \text{hash}\left(H_{l-1}|r_l\right)$.
31: Add $\langle l, H_l\rangle$ to Π^U.

Algorithm 10 Authenticator. Processing of an authentication request.

Let a be this authenticator, and $\rho = \rho_l$ an authentication request sent by the server at instant t_l.
state
    Stateless

upon receiving authentication request $\rho = \rho_l$ **containing**
    identifier of ρ
    $\rho.l = l$
    $\rho.A$ = chained authentication $[R_{l-2q}][R_{l-2q}, R_{l-2q+1}] \dots [R_{l-q-1}, R_{l-q}]$
    $\rho.\overline{A}$ = a sequence $R_{l-2q}, R_{l-2q+1}, \dots, R_{l-q-1}, R_{l-q}$
    $\rho.B$ = a sequence $b_1, b_2, \dots, b_z$ of signed updates $b_j = \left\langle u_j^{c(j)}, [u_j]_{c(j)} \right\rangle$, where $c(j)$ is the client that invoked u_j.
    $\rho.\bar{\bar{H}} = H_{l-2}$
    ρ.mainproof= proof $\left(\Pi_{l-1}^U, \langle l-1, H_{l-1}\rangle\right)$
    for each client c involved in $\rho.B$
        $\rho.p^c$ is the history-pair specified in the last update of c in $\rho.B$
        ρ.histproof(c) = proof $\left(\Pi_{l-1}^U, p^c\right)$
        ρ.pasthash$(c) = \langle \kappa^c, \eta^c \rangle$
        ρ.pasthashproof(c) = proof $\left(\Delta_{l-1}^U, \langle \kappa^c, \eta^c \rangle\right)$.

    for each key k involved in $\rho.B$
        Let v be the value of k in D_{l-1}^U
        ρ.oldval$(k) = v$
        ρ.proof(k) = proof $\left(\Delta_{l-1}^U, \langle k, v\rangle\right)$.

do
    Compact $\rho.A$ and $\rho.\overline{A}$ into $[R_{l-q}]_a$ by executing Algorithm 2.
    Compute $[R_{l-1}, R_l]_a$ by executing Algorithm 11.
    send authentication reply ρ_l^{rpl} **containing**
        identifier of ρ
        $[R_{l-q}]_a$
        $[R_{l-1}, R_l]_a$

Algorithm 11 Authenticator. Compute conditional authentication performing all related checks. In the comments, we write "should" when a certain condition is supposed to hold for executions with correctly behaving servers.

1: For each $b_j = \left\langle u_j^{c(j)}, [u_j]_{c(j)} \right\rangle$ in $B = \rho.B$, verify the signature of u_j in b_j.
2: $\overline{R} \leftarrow$ the root-hash computed using ρ.histproof(c) and $\rho.p^c$, where c is an arbitrarily chosen client among those involved in B.
▷ It should hold that $\overline{R} = R_{l-1}$
3: $\overline{r} \leftarrow$ the root-hash computed using ρ.pasthashproof(c) and ρ.pasthash(c), where c is an arbitrarily chosen client. ▷ It should hold that $\overline{r} = r_{l-1}$
4: For each key k involved in B, compute the root-hash using ρ.proof(k) and ρ.oldval(k) and verify it is equal to $\bar{r}$.
5: **for all** c involved in B **do**
6: Verify that the updates of c in B are correctly hash-chained.
7: Verify that the hash in the first update of client c in B is equal to η^c.
8: Verify that the root-hash computed using ρ.pasthashproof(c) and ρ.pasthash(c) is equal to $\overline{r}$.
9: Verify that the history root-hash computed using ρ.histproof(c) and $\rho.p^c$ is equal to $\overline{R}$.
10: Verify that version of $\rho.p^c$ is less than or equal to $\rho.l - 1$.
11: **end for**
12: $\bar{H} \leftarrow \text{hash}(\rho.\bar{\bar{H}}|\overline{r})$. ▷ It should holds that $\bar{H} = H_{l-1}$.
13: Verify that the history root-hash computed using ρ.mainproof and $\left\langle \rho.l - 1, \bar{H} \right\rangle$ is equal to $\overline{R}$.
14: $\widetilde{r} \leftarrow$the root-hash computed by applying all the updates $u^c = \langle i, k, v, \text{hash}(u^c(i-1)), p^c \rangle$ in B respecting their sequence in B:
15: for each key k involved in B, consider ρ.proof(k) and the new value v for k in the last update involving k
16: for each client-key κ^c for client c involved in B, consider ρ.pasthashproof(c) and the new value η^c for κ^c with $\eta^c = \text{hash}\left(u^c(i_{\max})\right)$ and $i_{\max}$ as in Algorithm 9.
▷ It should hold that $\widetilde{r} = r_l$.
17: $\widetilde{H} \leftarrow \text{hash}(\bar{H}|\widetilde{r})$ ▷ It should hold that $\widetilde{H} = H_l$.
18: $\widetilde{R} \leftarrow$ the history root-hash computed using ρ.mainproof and $\left\langle \rho.l, \widetilde{H} \right\rangle$.
▷ It should hold that $\widetilde{R} = R_l$.
19: **return** $\left[\overline{R}, \widetilde{R}\right]_a$ ▷ This should turn out to be $[R_{l-1}, R_l]_a$

State and Behaviour of a Client

Algorithm 12 Client. Reception of a read response.

Let l, Δ^R, Π^R, A, and $\overline{A}$ denote the values of the corresponding variables of the server when the response is sent.
Let c be this client.
state
    Γ: queue of history-pairs.

upon receiving read response for key k **containing**
    v: the value read.
    $l-q$: the version index of the dataset this response is based on.
    H_{l-q-1}: the history-hash of the previous version index.
    A and $\bar{A}$: authentication of R_{l-q}
    proof$(\Delta^R, \langle k, v\rangle)$
    proof$(\Pi^R, \langle l-q, H_{l-q}\rangle)$
    proof$(\Pi^R, \langle V^c, H^c\rangle)$
    $\langle V^c, H^c\rangle$

do
    Verify A and $\bar{A}$ by Algorithm 1.
    Compute r_{l-q} from proof$(\Delta^R, \langle k, v\rangle)$ and $\langle k, v\rangle$.
    $H_{l-q} \leftarrow \text{hash}(H_{l-q-1}|r_{l-q})$.
    Compute R_{l-q} from proof$(\Pi^R, \langle l-q, H_{l-q}\rangle)$ and $\langle l-q, H_{l-q}\rangle$.
    Verify that R_{l-q} is equal to the last hash in $\bar{A}$.
    Verify that $\langle V^c, H^c\rangle$ is in Γ.
    Pull from Γ all $\langle V, H\rangle \in \Gamma$ with $V < V^c$.
    Compute $\bar{R}$ as root-hash from proof$(\Pi^R, \langle V^c, H^c\rangle)$ and $\langle V^c, H^c\rangle$.
    Verify that $\bar{R} = R_{l-q}$.
    Verify that $V^c \leq l-q$.
    Push $\langle l-q, H_{l-q}\rangle$ into Γ.

Algorithm 13 Client. Reception of an update response.

1: ▷ The behaviour of the client when the response to an update $u(i) = \langle i, k_i, v_i, \text{hash}(u(i-1)), p\rangle$ is received is the same as in Algorithm 12. Note that, the value returned by the server can differ from v_i. This occurs when, in the same commit, a distinct update $\bar{u}$ for the same key k_i and value different from v_i is after $u(i)$ in B and overwrite k_i with a different value.

Nomenclature

ICS	Industrial Control System
IDS	Intrusion Detection System
RSD	Removable Storage Device
PLC	Programmable Logic Device
APT	Advanced Persistent Threat
DoS	Denial of Service Attack
HMI	Human-Machine Interface
ADS	Authenticated Data Structure
MHT	Merkle Hash Tree
HID	Human Interface Device
SDN	Software Defined Network

List of Publications

- **Federico Griscioli**, Maurizio Pizzonia, and Marco Sacchetti.
"USBCheckIn: Preventing BadUSB attacks by forcing human-device interaction."
2016 14th Annual Conference on Privacy, Security and Trust (PST). IEEE, 2016.
- **Federico Griscioli** and Maurizio Pizzonia.
"Securing promiscuous use of untrusted usb thumb drives in industrial control systems."
2016 14th Annual Conference on Privacy, Security and Trust (PST). IEEE, 2016.
- Miciolino EE, Di Noto D, **Griscioli F**, Pizzonia M, Kippe J, Pfrang S, Clotet X, León G, Kassim FB, Lund D, Costante E.
"Preemptive: an integrated approach to intrusion detection and prevention in industrial control systems."
IJCIS 13.2/3 (2017): 206-237.
- di Lallo, R., **Griscioli, F.**, Lospoto, G., Mostafaei, H., Pizzonia, M. and Rimondini, M.
"Leveraging sdn to monitor critical infrastructure networks in a smarter way."
2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). IEEE, 2017.
- **Federico Griscioli**, and Maurizio Pizzonia. *"USBCaptchaIn: Preventing (Un) Conventional Attacks from Promiscuously Used USB Devices in Industrial Control Systems."*// arXiv preprint arXiv:1810.05005 (2018).

Submitted

- **Federico Griscioli**, Diego Pennino, and Maurizio Pizzonia
"Toward a Cloud-Compatible Use of Authenticated Data Structures for Scalable On-the-fly Integrity Checks of Outsourced Data Storag"
Future Generation Computer Systems Journal, Elsevier
- **Federico Griscioli** and Maurizio Pizzonia
"USBCaptchaIn: Preventing (Un)Conventional Attacks from Promiscuously Used USB Devices in Industrial Control Systems"
Computer and Security Journal, Elsevier

Bibliography

[Aba12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[ABC+07] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. Acm, 2007.

[ABGS18] Khandakar Ahmed, Jan Blech, Mark Gregory, and Heinz Schmidt. Software defined networks in industrial automation. *Journal of Sensor and Actuator Networks*, 7(3):33, 2018.

[ADCE10] Lanzi Andrea, Balzarotti Davide, Kruegel Christopher, and Christodorescu Kidra Engin. AccessMiner: Using System-Centric Models for Malware Protection. In *Proceedings of the 17th Annual Computer Security Applications Conference*, Chicago, Illinois, USA, 2010. ACM.

[ADPMT08] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 9. ACM, 2008.

[AEK+17] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 251–266. ACM, 2017.

[AGH14] Carlos Aguayo Gonzalez and Alan Hinton. Detecting malicious software execution in programmable logic controllers using power fingerprinting. In Jonathan Butts and Sujeet Shenoi, editors, *Critical Infrastructure Protection VIII*, volume 441 of *IFIP Advances in Information and Communication Technology*, pages 15–27. Springer Berlin Heidelberg, 2014.

[AIS] AISWeb: The Online Home of Artificial Immune Systems. http://www.artificial-immune-systems.org/.

[AJGS83] Stanley R Ames Jr, Morrie Gasser, and Roger R Schell. Security kernel design and implementation: An introduction. *IEEE computer*, 16(7):14–22, 1983.

[AKL13] Sankalp Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219. IEEE, 2013.

[ALW+14] Ian F. Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1 – 30, 2014.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Apa18] Apache Cassandra. Apache cassandra documentation v4.0, 2018.

[AvdWL+13] F.A.T. Abad, J. van der Woude, Yi Lu, S. Bak, M. Caccamo, Lui Sha, R. Mancuso, and S. Mohan. On-chip control flow integrity check for real time embedded systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on*, pages 26–31, Aug 2013.

[B+14] Sebastian Burckhardt et al. Principles of eventual consistency. *Foundations and Trends® in Programming Languages*, 1(1-2):1–150, 2014.

[BÍ5] Genge Béla. Networked Critical Infrastructures: Secure and resilient by design. In *The Proceedings of the EUROPEAN INTEGRATION BETWEEN TRADITION AND MODERNITY Congress*, volume 6, pages 753–760, 2015.

[BCK17] Marcus Brandenburger, Christian Cachin, and Nikola Knežević. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):8, 2017.

[bea16] beagleboard.org. BeagleBone Black. `https://beagleboard.org/black`, [Online; accessed 27-July-2016].

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[bit] Bitlocker drive encryption overview. On-line `http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview`.

[BJO09] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[BPBF11] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. Duqu: A stuxnet-like malware found in the wild. *CrySyS Lab Technical Report*, 14:1–60, 2011.

[BS] Zhu B. and Sastry S. Scada-specific intrusion/prevention systems: A survey and taxonomy. *Department of Electrical Engineering and Computer Science*.

[CBK12] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 24(5):823–839, May 2012.

[CH14] Gideon Creech and Jiankun Hu. A Semantic Approach to Host-Based Intrusion Detection Systems Using Contiguousand Discontiguous System Call Patterns. *IEEE Transactions on Computers*, 63(4):807–819, 2014.

[CKS11] Christian Cachin, Idit Keidar, and Alexander Shraer. Fail-aware untrusted storage. *SIAM J. Comput.*, 40(2):493–533, April 2011.

[CO14] Christian Cachin and Olga Ohrimenko. Verifying the consistency of remote untrusted services with commutative operations. In *International Conference on Principles of Distributed Systems*, pages 1–16. Springer, 2014.

[Col13] Eric Cole. *Advanced Persistent Threats*. Elsevier, 2013.

[CS11] Ang Cui and Salvatore J Stolfo. Defending embedded systems with software symbiotes. In *Recent Advances in Intrusion Detection*, pages 358–377. Springer, 2011.

[CSS07] Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 129–138. ACM, 2007.

[CW09] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

[DBP07] G. Di Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. In *Data and Applications Security XXI*, pages 31–46. Springer, 2007.

[DC11] Jonathan J. Davis and Andrew J. Clark. Data preprocessing for anomaly based network intrusion detection: A review. *Computers & Security*, 30(6–7):353 – 375, 2011.

[DGMS03] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[DKS14] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 133:1–133:6, New York, NY, USA, 2014. ACM.

[Dok] Dokany. Dokan - user mode filesystem for windows os. On-line `http://fuse.sourceforge.net/` [Accessed 24-November-2015].

[DR08] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. Technical report, 2008.

[EBM+17] Faryed Eltayesh, Jamal Bentahar, Rabeb Mizouni, Hadi Otrok, and Elhadi Shakshuki. Refined game-theoretic approach to improve authenticity of outsourced databases. *Journal of Ambient Intelligence and Humanized Computing*, 8(3):329–344, 2017.

[EKPT15] C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):15, 2015.

[FLY+17] Anmin Fu, Yuhan Li, Shui Yu, Yan Yu, and Gongxuan Zhang. Dipor: An ida-based dynamic proof of retrievability scheme for cloud storage systems. *Journal of Network and Computer Applications*, 2017.

[FMC11a] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[FMC11b] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5, 2011.

[FPAC94] Stephanie Forrest, A.S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, 1994.

[FPC09] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, SecuCode '09, pages 19–26, New York, NY, USA, 2009. ACM.

[Fun12] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2:2–6, 2012.

[Fus] Fuse. Fuse - filesystem in userspace. On-line `http://fuse.sourceforge.net/`.

[FZFF10] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, volume 10, pages 337–350, 2010.

[GFG+17] E Goldin, D Feldman, Georgios Georgoulas, Miguel Castaño Arranz, and George Nikolakopoulos. Cloud computing for big data analytics in the process control industry. In *25th Mediterranean Conference on Control and Automation, MED 2017, University of Malta, Valletta, Malta, 3-6 July 2017*, pages 1373–1378. Institute of Electrical and Electronics Engineers (IEEE), 2017.

[GM17] Foschiano Ghosh and Mehta. Cisco systems' encapsulated remote switch port analyzer (erspan) draft-foschiano-erspan-03.txt. On-line `https://tools.ietf.org/html/draft-foschiano-erspan-03` [Accessed March-2015], 2017.

[GP16] Federico Griscioli and Maurizio Pizzonia. Securing promiscuous use of untrusted usb thumb drives in industrial control systems. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, pages 477–484. IEEE, 2016.

[GT00] Michael T Goodrich and Roberto Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.

[GTS01] Michael T Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 2, pages 68–82. IEEE, 2001.

[Gup12] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[HCLP15] Miguel Herrero Collantes and Antonio López Padilla. Protocols and network security in ics infrustructures. Technical report, INCIBE, 2015.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[HNKÖ18] Yahya Hassanzadeh-Nazarabadi, Alptekin Küpçü, and Öznur Özkasap. Decentralized and locality aware replication method for dht-based p2p storage systems. *Future Generation Computer Systems*, 84:32–46, 2018.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[HYQC09] Jiankun Hu, Xinghuo Yu, Dong Qiu, and Hsiao-Hwa Chen. A simple and efficient hidden Markov model scheme for host-based anomaly intrusion detection. *Network, IEEE*, 23(1):42–47, 2009.

[ICS11] ICS-CERT. Incident response summary report 2009-2011. Technical report, ICS-CERT, 2011.

[Int15] International Electrotechnical Commission (IEC). IEC 62443 Industrial communication networks - Network and system security, 2015.

[IRO16] IRONKEY$^{\text{TM}}$. Secure USB Devices: Protect Against BadUSB Malware. `http://www.ironkey.com/en-US/solutions/protect-against-badusb.html`, [Online; accessed 27-July-2016].

[JHN+14] Chiwook Jeong, Taejin Ha, Jargalsaikhan Narantuya, Hyuk Lim, and Jong-Won Kim. Scalable network intrusion detection on virtual sdn environment. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 264–265. IEEE, 2014.

[JK07] Ari Juels and Burton S. Kaliski. Pors: proofs of retrievability for large files. In *In CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. ACM, 2007.

[JP11] V. Jyothsna and V.V. Rama Prasad. A Review of Anomaly based Intrusion Detection Systems. *International Journal of Computer Applications*, 28(7):26–35, August 2011.

[JW13] Ruofan Jin and Bing Wang. Malware detection for mobile devices using software-defined networking. In *Proceedings of the 2013 Second GENI Research and Educational Experiment Workshop*, GREE '13, pages 81–88, Washington, DC, USA, 2013. IEEE Computer Society.

[Kál16] György Kálmán. Prospects of software-defined networking in industrial operations. *International Journal on Advances in Security Volume 9, Number 3 & 4, 2016*, 2016.

[KF13] H. Kim and N. Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, February 2013.

[KMP+15a] J. Kippe, D. Meier, S. Pfrang, R. Barbosa, A. Skene, T. Kassim, M. Pizzonia, F. Griscioli, E. Zambon, and A. Ursini. Security frameworks: State of the art evaluation. Technical report, The Preemptive project, 2015. `http://preemptive.eu/wp-content/uploads/2016/05/preemptive_d4.1.pdf`.

[KMP+15b] J. Kippe, D. Meier, S. Pfrang, X. Clotet Fons, G. Eliana Leon, M. Wrightson, , A. Skene, T. Kassim, M. Pizzonia, F. Griscioli, E. Zambon, E. Etchevés Miciolino, and A. Ursini. Preemptive methodology reference. Technical report, The Preemptive project, 2015. `http://preemptive.eu/wp-content/uploads/2016/05/preemptive_d4.2.pdf`.

[KS17] Myung Kang and Hossein Saiedian. Usbwall: A novel security mechanism to protect against maliciously reprogrammed usb devices. *Information Security Journal: A Global Perspective*, 26(4):166–185, 2017.

[Lew14] Ted G Lewis. *Critical infrastructure protection in homeland security: defending a networked nation*. John Wiley & Sons, 2014.

[LHK+16] Edwin Lupito Loe, Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Shao-Chuan Lee, and Shin-Ming Cheng. Sandusb: An installation-free sandbox for usb peripherals. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*, pages 621–626. IEEE, 2016.

[LHKR06] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132. ACM, 2006.

[LHKR08] Feifei Li, Marios Hadjileftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for outsourced databases. In *Handbook of Database Security*, pages 115–136. Springer, 2008.

[lib16] libusb: A cross-platform user library to access USB devices. `http://libusb.info/`, [Online; accessed 28-July-2016].

[LKMS04] Jinyuan Li, Maxwell N Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *OSDI*, volume 4, pages 9–9, 2004.

[LTC+15] Jin Li, Xiao Tan, Xiaofeng Chen, Duncan S Wong, and Fatos Xhafa. Opor: enabling proof of retrievability in cloud computing with resource-constrained devices. *IEEE Transactions on cloud computing*, 3(2):195–205, 2015.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP@ ISCA*, 10, 2013.

[Mer87] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer, 1987.

[Mer88] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO'87*, pages 369–378. Springer, 1988.

[MG+11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[MND+04] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[MNG+17] Estefanía Etchevés Miciolino, Dario Di Noto, Federico Griscioli, Maurizio Pizzonia, Jörg Kippe, Steffen Pfrang, Xavier Clotet, Gladys León, Fatai Babatunde Kassim, David Lund, et al. Preemptive: an integrated approach to intrusion detection and prevention in industrial control systems. *International Journal of Critical Infrastructures*, 13(2-3):206–237, 2017.

[MOD96] Industrial Automation Systems MODICON, Inc. Modbus protocol – reference guide. Technical report, 1996.

[MS02] David Mazieres and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117. ACM, 2002.

[MS05] Gerome Miklau and Dan Suciu. Implementing a tamper-evident database system. In *Advances in Computer Science–ASIAN 2005. Data Management on the Web*, pages 28–48. Springer, 2005.

[MSL+11] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[MW18] Collin Mulliner and Edgar R Weippl. Usblock: Blocking usb-based keypress injection attacks. In *Data and Applications Security and Privacy XXXII: 32nd Annual IFIP WG 11.3 Conference, DBSec 2018, Bergamo, Italy, July 16–18, 2018, Proceedings*, volume 10980, page 278. Springer, 2018.

[nex11] Cisco nexus 7000 series nx-os system management configuration guide. Technical report, Cisco Systems Inc., 2011.

[NK16] Lell J Nohl K. BadUSB - On Accessories that Turn Evil. `https://www.blackhat.com/us-14/briefings.html#Nohl`, [Online; accessed 27-July-2016].

[NL14] Karsten Nohl and Jakob Lehl. Badusb–on accessories that turn evil. *Black Hat USA*, 2014.

[Nor13] North American Electric Reliability Corporation (NERC). Critical Infrastructure Protection (NERC CIP). `http://www.nerc.com/pa/Stand/Pages/CIPStandards.aspx`, 2013.

[ODV] Inc. ODVA. The common industrial protocol (cip). https://www.odva.org/Technology-Standards/Common-Industrial-Protocol-CIP/.

[Par] Preemptive Project Partners. Preemptive: Preventive methodology and tools to protect utilities. March 2014 – February 2017. Funded by the European Commission under FP7, G.A. 607093. On-line. `http://preemptive.eu`.

[Per] Etienne Perot. Fuse-jna - no-nonsense, actually-working java bindings to fuse using jna. On-line `https://github.com/EtiennePerot/fuse-jna` [Accessed 26-July-2016].

[PGG+18] Andrés F Murillo Piedrahita, Vikram Gaur, Jairo Giraldo, Alvaro A Cardenas, and Sandra Julieta Rueda. Leveraging software-defined networking for incident response in industrial control systems. *IEEE Software*, 35(1):44–50, 2018.

[Pol60] Maurice Pollack. Letter to the editor—the maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.

[PPP10a] B. Palazzi, M. Pizzonia, and S. Pucacco. Query racing: Fast completeness certification of query results. In *Proc. Working Conference on Data and Applications Security and Privacy (DBSEC'10)*, volume 6166 of *Lecture Notes in Computer Science*, pages 177–192, 2010.

[PPP10b] Bernardo Palazzi, Maurizio Pizzonia, and Stefano Pucacco. Query racing: fast completeness certification of query results. In *Data and Applications Security and Privacy XXIV*, pages 177–192. Springer, 2010.

[PT07] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *International conference on information and communications security*, pages 1–15. Springer, 2007.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, volume 1, 2001.

[PZM09] HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidis. Scalable verification for outsourced dynamic databases. *Proceedings of the VLDB Endowment*, 2(1):802–813, 2009.

[Rau11] Suhas Rautmare. Scada system security: Challenges and recommendations. In *India Conference (INDICON), 2011 Annual IEEE*, pages 1–4. IEEE, 2011.

[RB07] Bozidar Radunović and Jean-Yves Le Boudec. A unified framework for max-min and min-max fairness with applications. *IEEE/ACM Transactions on Networking (TON)*, 15(5):1073–1083, 2007.

[RRL+12] Jason Reeves, Ashwin Ramaswamy, Michael Locasto, Sergey Bratus, and Sean Smith. Intrusion detection for resource-constrained embedded control systems in the power grid. *International Journal of Critical Infrastructure Protection*, 5(2):74–83, 2012.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption*, pages 371–388. Springer, 2004.

[RS09] M. Russinovich and D. A Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista*. Microsoft press, 2009.

[RTZ03] Matthew Roughan, Mikkel Thorup, and Yin Zhang. Traffic engineering with estimated traffic matrices. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 248–258. ACM, 2003.

[SBB13] Richard Skowyra, Sanaz Bahargam, and Azer Bestavros. Software-defined ids for securing embedded mobile devices. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–7. IEEE, 2013.

[sdn] **Companion Website with code**. `https://bitbucket.org/sdnci/sdn-ci/`.

[SFS11] Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to industrial control systems (ics) security. *NIST special publication*, pages 800–82, 2011.

[SFS13] Keith Stouffer, Joe Falco, and Karen Scarfone. Nist special publication 800-82 - guide to industrial control systems (ics) security, may 2013.

[SLP+15] Keith Stouffer, Suzanne Lightman, Victoria Pillitteri, Marshall Abrams, and Adam Hahn. Guide to industrial control systems (ics) security – nist special publication (sp) 800-82 revision 2. Technical report, NIST, 2015.

[SM08] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *Systems and Networks Communications, 2008. ICSNC '08. 3rd International Conference on*, pages 23–26, Oct 2008.

[SN16] SALVATORE Sanfilippo and P Noordhuis. The redis documentation, 2016.

[SP08] Sarvjeet Singh and Sunil Prabhakar. Ensuring correctness over untrusted private database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 476–486. ACM, 2008.

[Spe13] OpenFlow Switch Specification. Version 1.3.3 (wire protocol 0x04), Sept 2013.

[Spi16] Dominic Spill. USBProxy. `https://github.com/dominicgs/USBProxy`, [Online; accessed 27-July-2016].

[SPL+11] Keith Stouffer, Victoria Pillitter, Suzanne Lightman, Marshall Abrams, and Adam Hahn. Sp 800-82r2. guide to industrial control systems (ics) security: Supervisory control and data acquisition (scada) systems, distributed control systems (dcs), and other control system configurations such as programmable logic controllers (plc). 2011.

[SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSB+14] Praveen Kumar Shanmugam, Naveen Dasa Subramanyam, Joe Breen, Corey Roach, and Jacobus Van der Merwe. Deidtect: towards distributed elastic intrusion detection. In *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing*, pages 17–24. ACM, 2014.

[SSS+10] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller. An overview of ip flow-based intrusion detection. *Communications Surveys Tutorials, IEEE*, 12(3):343–356, 2010.

[SvDJO12] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 229–238. ACM, 2012.

[SW13] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of cryptology*, 26(3):442–483, 2013.

[Tam03] Roberto Tamassia. Authenticated data structures. In *ESA*, volume 2832, pages 2–5. Springer, 2003.

[TBB15a] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending against malicious usb firmware with goodusb. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270. ACM, 2015.

[TBB15b] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending against malicious usb firmware with goodusb. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270. ACM, 2015.

[TCB08] Steven Tom, Dale Christiansen, and Dan Berrett. Recommended practice for patch management of control systems. *DHS control system security program (CSSP) Recommended Practice*, 2008.

[TDF+16] Matthew Tischer, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein, and Michael Bailey. Users really do plug in usb drives they find. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 306–319. IEEE, 2016.

[TG10] Amin Tootoonchian and Yashar Ganjali. Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[top] Secure usb drive review. On-line `http://secure-usb-drive-review.toptenreviews.com/`.

[Typ] Typesafe. Akka. On-line `https://www.typesafe.com/community/core-projects/akka`.

[TZ94] Jean-Pierre Tillich and Gilles Zémor. Hashing with sl 2. In *Annual International Cryptology Conference*, pages 40–49. Springer, 1994.

[VGA13] Nikos Virvilis, Dimitris Gritzalis, and Theodoros Apostolopoulos. Trusted computing vs. advanced persistent threats: Can a defender win this game? In *Ubiquitous Intelligence and Computing, 2013 IEEE UIC/ATC*, pages 396–403. IEEE, 2013.

[vid] **USBCheckIn companion video**.
`http://www.dia.uniroma3.it/~pizzonia/pst/`
username: "pst2016" password: "pst2016".

[VS16] A. Valentini and G. Sinibaldi. PREEMPTIVE - PREventivE Methodology and Tools to protect utilitIEs. In *Fast abstracts at International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Trondheim, Norway, September 2016.

[VV16] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):19, 2016.

[WD01] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. Technical report, U.C. Berkeley, 2001.

[WFP99] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proc. of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145. IEEE, 1999.

[Wil98] Timothy Williams. The purdue enterprise reference architecture and methodology (pera). *Handbook of life cycle engineering: concepts, models, and technologies*, 289, 1998.

[WSJ17] Martin Wollschlaeger, Thilo Sauter, and Juergen Jasperneite. The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1):17–27, 2017.

[WSS09] Peter Williams, Radu Sion, and Dennis E Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *NDSS*, 2009.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[WWL+09] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *European symposium on research in computer security*, pages 355–370. Springer, 2009.

[WZY06] Miao Wang, Cheng Zhang, and Jingjing Yu. Native API based windows anomaly intrusion detection method using SVM. In *Proc. of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, volume 11. IEEE, 2006.

[YPPK09] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 5–18. ACM, 2009.

[YUH06] Dayu Yang, Alexander Usynin, and J Wesley Hines. Anomaly-based intrusion detection for SCADA systems. In *Proc. of the 5th international topical meeting on nuclear plant instrumentation, control and human machine interface technologies*, pages 12–16, 2006.

[ZAH+13] Yan Zhu, Gail-Joon Ahn, Hongxin Hu, Stephen S Yau, Ho G An, and Chang-Jun Hu. Dynamic audit services for outsourced storages in clouds. *IEEE Transactions on Services Computing*, 6(2):227–238, 2013.

[ZKM+17] Faheem Zafar, Abid Khan, Saif Ur Rehman Malik, Mansoor Ahmed, Adeel Anjum, Majid Iqbal Khan, Nadeem Javed, Masoom Alam, and Fuzel Jamil. A survey of cloud computing data integrity schemes: Design challenges, taxonomy and future trends. *Computers & Security*, 65:29–49, 2017.

[ZKP15] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491. ACM, 2015.