# A SECURE SYSTEM ARCHITECTURE FOR MEASURING INSTRUMENTS UNDER LEGAL CONTROL

vorgelegt von
Dipl.-Inform.
Daniel Peters
geb. in Temeschburg

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

| | |
|---|---|
| Vorsitzender: | Prof. Dr. Sebastian Möller, Technische Universität Berlin |
| Gutachter: | Prof. Dr. Jean-Pierre Seifert, Technische Universität Berlin |
| Gutachter: | Prof. Dr. Michael Waidner, Fraunhofer SIT und TU Darmstadt |
| Gutachter: | Prof. Dr. Volker Markl, Technische Universität Berlin |

Tag der wissenschaftlichen Aussprache: 14. Juni 2017

Berlin 2017

Ich versichere von Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

_____

Datum

## Abstract

In many fields, application software must be stable and withstand attacks. Due to the trend of the "Internet of Things" these attacks increase in all areas where devices are connected via the open network. Meanwhile, even everyday devices such as smartphones, tablets and measuring instruments, on which the concrete focus of this thesis lies, have evolved into powerful universal devices with an open system architecture. It should be noted that IT systems running conventional operating systems can be barely protected. New approaches for measuring instruments are needed to counter the growing number of threats. With the use of separation kernels, the components of a software system can be isolated in terms of space as well as time, which makes the whole system highly reliable. This ensures that the malfunction of a component may have no effect on the interoperability of other modules in the system. The risk of a malfunction can be lowered to the extent that critical applications are executable, which is also important for legally monitored measuring devices. In this thesis, a modular system architecture for measuring instruments under legal control, which is running on a micro-/separation kernel, is being constructed, fulfilling the European directives and guides.

A further requirement of regularized meters is the easy verification of system integrity, such as the file system. If the files and the file system structure have not changed on the instrument, one can assume that it is still running the same software, which has been approved. For this purpose, the measuring instrument shall output a so-called software identifier. Until now, no specific method has been described as to how this software identifier is to be formed. The method presented in this thesis is focused on an efficient and succinct verification of file system integrity. Its focus lies on the usability on embedded and low-resource devices. In doing so, the legal requirements of the measuring system are complied with and a comprehensible method is provided up to the hash calculation. It allows the files and the file system to be moved to other devices (for example, in the case of Cloud applications) and also remote hash verification over open networks, to replace the on-site examination. Hereby, an efficient calculation of the file system structure with low space consumption is made possible.

This thesis analyzes specifically the requirements for measuring instruments under legal control with the focus on software. A solution for a secure software system architecture, which meets all these requirements is given. Additionally, a new approach for file system integrity checking for market surveillance to verify the integrity of the software in commission is being described.

## Kurzzusammenfassung

In vielen Einsatzgebieten muss Software stabil laufen und Angriffen standhalten. Durch den Trend des "Internet der Dinge" häufen sich diese Angriffe in allen Bereichen, in denen Geräte über das offene Netzwerk verbunden sind. Mittlerweile betrifft das viele Alltagsgeräte, wie Smartphones, Tablets und Messgeräte, mit denen sich diese Arbeit im Detail beschäftigt, die sich alle zu leistungsfähigen Universalgeräten mit offener Systemarchitektur entwickelt haben. Es ist festzustellen, dass IT-Systeme mit konventionellen Betriebssystemen kaum noch abzusichern sind. Neue Ansätze für Messgeräte sind nötig, um der wachsenden Anzahl an Bedrohungen entgegenzuwirken. Mit dem Einsatz von Separationskernen wird das Ziel verfolgt, Teilkomponenten eines Softwaresystems nachweislich und mit hoher Verlässlichkeit sowohl räumlich wie auch zeitlich zu isolieren. Dadurch wird sichergestellt, dass ein Fehlverhalten einer Komponente keinen Einfluss auf die Lauffähigkeit anderer Module im System hat. Das Risiko einer Fehlfunktion kann soweit verringert werden, dass kritische Anwendungen lauffähig bleiben, was auch im Bereich der rechtlich überwachten Messgeräte wichtig ist. In dieser Arbeit wird nun eine komponentenbasierte Systemarchitektur für genau diese Messgeräte vorgestellt. Diese läuft auf einem Mikro-, bzw. Separationskern und hält alle europäischen Richtlinien und Leitfäden ein.

Eine weitere Anforderung an gesetzlich geregelte Messgeräte ist die einfache Überprüfung der Systemintegrität, wie beispielsweise des Dateisystems. Falls sich die Dateien und die Dateisystemstruktur auf dem Messgerät nicht verändert haben, kann man davon ausgehen, dass noch dieselbe Software auf dem Gerät läuft, die auch zugelassen wurde. Das Messgerät soll dafür einen sogenannten Software Identifikator ausgeben. Bis jetzt wird kein konkretes Verfahren beschrieben, wie dieser Software Identifikator gebildet werden soll. Das hier vorgestellte Verfahren konzentriert sich auf eine effiziente und platzsparende Prüfung der Dateisystemintegrität, um auch für eingebettete, ressourcenarme Geräte einsetzbar zu sein. Dabei werden die gesetzlichen Vorgaben des Messwesens eingehalten und ein nachvollziehbares Verfahren bis zur Hashberechnung geliefert. Es ermöglicht die Auslagerung der Dateien und des Dateisystems auf andere Geräte (z.B. für Cloud Anwendungen) und die Hash-Überprüfung über offene Netzwerke aus der Ferne, um die Vorort-Prüfung zu ersetzen. Dabei wird eine effiziente Berechnung der Dateisystemstruktur mit nur geringem Platzverbrauch ermöglicht.

Diese Arbeit analysiert konkret die Anforderungen an gesetzlich geregelte Messsoftware und gibt Lösungen wie eine Software-Systemarchitektur aussehen kann, die alle Anforderungen erfüllt und gleichzeitig der Marktüberwachung Mechanismen zur Verfügung stellt, um die Integrität von Messgerätesoftware im Umlauf zu überprüfen.

## Publication List

The primary results of this thesis have been presented in the following peer-reviewed publications:

### Journals

1. J. Fischer*, D. Peters*: GLOUDS: Representing tree-like graphs. Elsevier, Journal of Discrete Algorithms; 11/2015. https://doi.org/10.1016/j.jda.2015.10.004 (Section 2.4 and Chapter 4)

2. D. Peters, M. Peter, J.-P. Seifert, F. Thiel: A Secure System Architecture for Measuring Instruments in Legal Metrology. MDPI, Computers; 03/2015, 4(7):61-86. https://doi.org/10.3390/computers4020061 (Chapter 2 and Chapter 3)

### Conferences

3. D. Peters, F. Thiel: Software in Measuring Instruments: Ways of Constructing Secure Systems, SENSOR 2016, Nuremberg; 05/2016. https://doi.org/10.5162/sensoren2016/6.3.4 (Chapter 2 and Chapter 3)

4. D. Peters, F. Thiel, M. Peter, J.-P. Seifert: A Secure Software Framework for Measuring Instruments in Legal Metrology. Instrumentation and Measurement Technology Conference (I2MTC), 2015 IEEE International, Pisa; 05/2015. https://doi.org/10.1109/I2MTC.2015.7151517 (Chapter 2 and Chapter 3)†

5. J. Fischer*, D. Peters*: A Practical Succinct Data Structure for Tree-Like Graphs. Springer LNCS, WALCOM: Algorithms and Computation, Dhaka; 02/2015. (Section 2.4 and Chapter 4)‡

6. D. Peters, U. Grottker, F. Thiel, M. Peter, J.-P. Seifert: Achieving software security for measuring instruments under legal control. Federated Conference on Computer Science and Information Systems, ISBN 978-83-60810-57-6 (online), Warsaw ; ISSN 3300-5963; 09/2014 https://doi.org/10.15439/2014F460 (Chapter 2 and Chapter 3)

*The authors contributed equally to the work



‡The final publication is available at Springer via https://doi.org/10.1007/978-3-319-15612-5_7

# Acknowledgments

First of all, I would like to thank my parents, which have always supported me in all my decisions and have helped me in every way possible to achieve my goals. I especially thank my grandmother, Elisabeth Schreyer, who took care of me as long as she was able to and always proudly believed in me.

I would like to thank my sister and my cousin for being next to me all the way through this thesis, and my life. They always have been and will be an enrichment to my life. I would also like to thank my aunt and uncle that always listen to my problems, in science and life.

I would like to thank my PhD advisor Jean-Pierre Seifert, who pointed me into the right directions, believed in my research skills and made it possible for me to attend the Helmholtz Research School on Security Technologies, which gave me a broader insight into security related topics. He is a great expert in his field and an inspiration. For inspiring the joy about research, I would like to thank my former Diplom (Master) thesis advisor Johannes Fischer. With his expertise in succinct data structures he not only helped me then, but also in newer cooperations. It is always a pleasure working with him. A special thanks also goes to my supervisor at the Physikalisch-Technische Bundesanstalt (PTB), Florian Thiel, who not only believed in my competence from the first minute we met but the whole way through. Thanks to him, I always believed in my research, my thesis and our projects.

I would like to thank the team of the Chair for Security in Telecommunications in Berlin (SecT) for helping me out with problems concerning their microkernel, especially Michael Peter, with whom I had many interesting discussions in general. I would also like to thank all the colleagues in my Department (8.5) at the PTB for hearing me out and giving me helpful feedback.

At last, I would like to thank the anonymous reviewers of all the journals and conferences for several helpful suggestions that improved the quality of this thesis.

# "Computers Are Useless. They Can Only Give You Answers."

Pablo Picasso
The Paris Review 32: "Pablo Picasso: A Composite Interview"
1964

# Contents

# Listing of Figures

# Listing of Tables

# 1 Introduction

## 1.1 Problem Statement

In light of the fact that around 98% of the world's computer systems are embedded devices[29], security in embedded systems will inevitably play an important role in computer science. Embedded systems show the tendency of becoming more and more connected. This fact combined with the trend towards the Internet of Things, from which measuring instruments are not immune (e.g., smart meters), lets one assume that security in measuring instruments will inevitably play an important role soon. Additionally, measuring instruments have adopted general-purpose operating systems to offer the user a broader functionality that is not necessarily restricted towards measurement alone. Challenges in constructing a secure embedded system arise due to the increase in complexity, which is driven by the growing demand for improved capabilities, the digitization of manual and mechanical functions and interconnect-ability.

According to estimations, about four to six percent of the gross national income in industrial countries is accounted for by measuring instruments, which are subject to legal control[73], e.g., electricity meters, gas meters and their related measurements. In Germany alone, this corresponds to an amount of 104 to 157 billion Euros each year[73]. Hence, manipulations of measuring instruments' software could have far-reaching financial consequences. Clearly, special measures should be considered to secure such instruments.

Nowadays, most of the manufacturers of measuring instruments prefer

building their software stacks on general-purpose operating systems (GPOSs), like Linux and Windows, due to the wide availability of device drivers, software infrastructure and applications. These systems were not created with high security awareness in mind. Their total embedded software content often exceeds 10 million source lines of code (SLOC). The alarming danger becomes clear when knowing that tests place the number of severe bugs in well-written open source software code at the rate of about one per 2000 SLOC[21]. Such a high amount of erroneous code consequentially increases the vulnerability of these systems to attackers, because just one bug could be so severe that a sophisticated attacker is able to run arbitrary code on the measuring instrument. The National Vulnerability Database *, for example, reveals such bugs weekly.

First of all, one should have a look at the basic lawful requirements a measuring instrument in legal metrology has to meet with respect to security. Here, consumer protection and the certainty of a correct measurement are most important. A consumer must be sure that, for example, a fuel dispenser is not manipulated to charge more than what was fueled. Hence, in Europe, member states denominate institutions, called notified bodies, which are responsible for reviewing the measuring instruments before commissioning. Current scrutiny in the laboratory concentrates on the validation of correct measurements from the hardware parts, e.g., physical sensors. Software analysis is hampered by obstacles, like proprietary software, e.g., the operating system, where source code cannot be checked. The approval for commission is often given after a "black box" validation of the sensors, by application of some test vectors at the user interface and the sealing of as many interfaces as possible. The aforementioned fuel dispenser is stated to be not manipulated as long as no seal is broken. As mentioned before, this assumption is too optimistic, considering that just one open interface, e.g., an USB-port or WiFi, can allow a disposed attacker to run arbitrary code by exploiting a single vulnerability.

## 1.2 Research Question and Methodology

For measuring instruments **two challenges** arise, which are highlighted in this thesis:

1. Current systems were not constructed with security in mind, which leads to the point that a **new software system architecture** must be

*A catalog of software bugs published by the U.S. National Institute of Standards and Technology and the U.S. Department of Homeland Security's National Security Cyber Division

constructed, which is tailored to all the legal requirements, especially with security in mind.

2. Spotting malicious manipulation or changes in the software, and remote attestation and maintenance is very important. The directives and guides in legal metrology only speak about an **identifier** that shall be outputted to check **system integrity**, no concrete method is being described.

A major drawback in constructing a new software system is that drivers and software libraries available for known GPOSs, e.g., Linux, which uses a monolithic kernel design, need to be ported, or in the worst case, completely rewritten.

In a monolithic kernel system architecture, the entire operating system is working in privileged mode sharing a single memory space with the system software, such as file systems and complex device drivers with direct access to the hardware. The disadvantage of this approach is the resulting large trusted computing base (TCB). The TCB refers to those portions of a system (software and hardware) that are needed in a system to ensure that it works as expected. Therefore, it must be trustworthy.

In the microkernel design, the microkernel is the only software executed at the most privileged level. Hence, in contrast to a monolithic design, services are implemented in separate processes. The motivation to place as much functionality as possible in separate protection domains, not running in privileged mode, is to gain stability, because, for example, a crash in the network stack that would have been fatal for a monolithic system is now survivable. Consequently, the goal of this architecture is to keep the TCB small and under control.

Virtualization seems to be the right solution to incorporate the best implementations of both architectures in a single system. Through virtualization, device drivers and software libraries of general purpose operating systems (GPOS) can be reused, which makes the implementation of our system architecture for a manufacturer as easy as possible. In this thesis, we chose a microkernel design which encapsulates all security relevant modules in separate virtual machines. To show the feasibility of our approach, we started to build a system atop a L4-microkernel. In our opinion, the L4-microkernel family is a good starting point, because it is widely used and consists of third generation microkernels. One of these microkernels, called seL4, is even fully verified[66], inferring that classical security threats against operating systems, like buffer overflows, null pointer dereferencing, arithmetic overflows, arithmetic exceptions, pointer errors and memory leaks, are not

possible. Nevertheless, it should be mentioned, that the L4 microkernel we used for our tests, Fiasco.OC, is not formally verified. Another drawback is that Fiasco.OC does not support full-virtualization, and only a binary compatible para-virtualized Linux, called L4Linux, has been ported. This means, that often the newest Linux kernel version is not available, because every kernel version needs to be ported to Fiasco.OC manually. Concretely, we used L4Linux (kernel version 3.14) running on top of the open source Fiasco.OC L4-microkernel. Our test board was the PandaBoard Revision B3 equipped with the OMAP4460 SoC running a dual-core 1.2 GHz ARM Cortex-A9 MPCore with 1 GiB of DDR2 SDRAM. We think that by showing that our system framework is able to run in a para-virtualized environment, we do not restrict the manufacturers of measuring instruments to a kernel. Still it should be noted that a verified hypervisor with full virtualization support is the better choice.

The second point that is very important for devices, which are under legal control, is spotting malicious manipulation, and making remote attestation and maintenance possible. For measuring instruments, the manufacturer and the market surveillance want to check if system integrity is preserved. In Europe, legal requirements even state that a software identifier needs to be supplied/output by the device, which is often just a checksum over the files that are considered to be legally relevant for the measuring purpose. Data exchange between devices over the internet has become an important aspect, nowadays. In the era of the Internet of Things (IoT), the number of connected devices will, according to Gartner[111], exceed 25 billion in the year 2020. As storage units have become smaller and cheaper, these devices can already save millions of files. Considering that in the future the automatic data exchange between these devices will blossom, the creation of a small data structure listing all the files on a device is important. Despite being small, this data structure should also be quickly traversable. It should list as much information about a file as possible to check, for example, the name, the format, the size, and the checksum. As measuring instruments are often small embedded systems, the need for a fast algorithm arises that creates a small file system list containing as much information as possible. In this thesis, a new file system structure, we called FLOUDS, is explained that fulfills these requirements. Additionally, the FLOUDS may be used as a replacement for databases that are needed to efficiently find files, e.g. the database used by the *locate* command under Unix systems, or as the basis for small read-only file systems.

## 1.3 Impact of Thesis Contribution

The framework presented in this thesis addresses every powerful measuring instrument that is able to run general-purpose operating systems. Such measuring instruments are very common nowadays and are, for example, traffic enforcement meters, moisture meters, fuel dispensers, and many more. However, even small measuring instruments, like dosimeters, often present their measuring results on a separate display device, which can be a tablet running Android. For these powerful measuring instruments we present a new secure software framework, which uses known and sound techniques, like a microkernel/hypervisor and virtualization, and combines them with the requirements of directives and guides for measuring instruments in Europe, which are under legal control. Its modular design makes the framework very flexible and scalable, which leads to a design that is much better verifiable than existing architectures.

In many states, national laws oblige the manufacturer of these devices to output a software identifier for the user and the market surveillance. This identifier should be created in a way that data integrity can be inferred. Often, as mentioned before, this is done by just calculating a checksum over all the files that seem to be important for the measuring task and showing it on the screen. Considering that remote maintenance and automatic scanning of the devices will be the next step to achieve remote integrity checking, a file list containing as much information as possible is preferable. This thesis also describes such a data structure which makes use of succinct approaches to store trees. In this structure, a fast file search is made possible by using space-efficient algorithms to store the file names.

To sum up, the framework and the data structure described in this thesis, help manufacturer to build a secure software stack for every measuring instrument under legal control that fulfills every legal requirement, beginning from the fundamental design. By using this framework, the software testers of the notified bodies can easily check the design of the software and market surveillance has a tool to verify integrity quickly, which leads in total to a faster validation and therefore, to cost savings.

## 1.4 Structure of the Thesis

This thesis is organized as follows:

- Chapter 1 gave an introduction into the thesis, stating the problem (Section 1.1) and defining two main goals (Section 1.2), first, the construction of a new system architecture based on a micro/separation

kernel, which fulfills all legal requirements for measuring instruments under legal control, and second, the creation of a succinct data structure, which can be used for integrity checking and remote maintenance of the devices (Section 1.3).

- Chapter 2 firstly provides an introductory part about legal metrology (Section 2.1), especially legal requirements for measuring instruments in Europe, before looking at concrete software requirements (Section 2.2). In Section 2.3 the basics that help to construct a secure software system are described, like microkernels and virtualization. Afterwards, policies of secure systems and measures to achieve them are named. In Section 2.4, an introduction on succinct data structures and help functions is given, which we use to construct our succinct data structure (Chapters 4 and 5), like *rank*- and *select*, which are needed to traverse many succinct data structures, one such data structure is the "Level Order Unary Degree Sequence" (LOUDS). At the end, in Section 2.5, an overview about additional security aspects, like formal verification and hardware extensions is given, but with the question in mind, how much security can we ask for without restricting the manufacturer to special hardware, and without increasing the expenses in the software development process in an unreasonable way.

- Chapter 3 shows the framework of the system architecture (Section 3.1), describes its distributed system nature (Section 3.1.2) and the duties of the individual virtual machines (Section 3.2). Section 3.3 goes into more detail describing a special virtual machine, named the Inspector, which can be used as a monitoring virtual machine. It is responsible for system integrity checking and software-based attestation. In Section 3.4, we show the feasibility of our approach by an implementation on a demonstrator. We start all of the VMs on this demonstrator and measure the times that are necessary to transmit information from one VM to another through a virtual network. Section 3.5 analyzes the system by showing that the policies data isolation, information flow control, damage limitation and period processing are upheld. Section 3.6 sums up related work about microkernel and hypervisor architectures. In Section 3.7, the chapter is summarized.

- Chapter 4 explains the theoretical data structure we called "Graph Level Order Unary Degree Sequence" (GLOUDS), which is later on used in Chapter 5 to construct the file system datastructure. In Section 4.1, an introduction about our definition of tree-like graphs is given, outlining also related work (Section 4.1.1). In Section 4.2, the functions

*rank*-, *select* and the LOUDS are again explained shortly, because they build the basis for our new data structure. Afterwards, in Section 4.3 the GLOUDS is explained in detail and also the implementation possibilities we used (Section 4.4). In Section 4.5, the practicality of our approach on the example of phylogenetic networks (Section 4.5.1) is shown and afterwards on general graphs (Section 4.5.2) and web graphs (Section 4.5.3). In Section 4.6, the chapter is summarized.

- Chapter 5 explains the "File system Level Order Unary Degree Sequence" (FLOUDS), which is based on the GLOUDS, by starting with an introduction about the topic, outlining the importance and usability of a succinct file system structure (Section 5.1) before explaining it in detail (Section 5.2). Afterwards, practical tests in Section 5.3 show the efficiency of the FLOUDS by comparing it with the *locate* database of UNIX systems. At the end, before the summary of the chapter in Section 5.5, a discussion is given in Section 5.4 which describes where the FLOUDS can be used and how file modifications can be handled.

- Chapter 6 concludes the thesis and gives an outlook concerning other research opportunities, like tailoring the architecture for Cloud applications (Section 6.1).

# 2 Background

## 2.1 Legal Metrology

Legal metrology is comprised of measuring instruments that are employed for commercial or administrative purposes or for measurements that are of public interest. More than 100 million legally relevant meters are in use in Germany[73]. The majority of them are used for business purposes, in particular they are commodity meters for the supply of electricity, gas, water or heat. Other classical measuring instruments, with which the end user comes into contact, are, e.g., counters in petrol pumps or scales in the food sector. Measuring instruments are to a large extent also required in the public traffic system. Examples are speed or alcohol meters. The commonality of all of these applications is that the person executing or being affected by an official measurement cannot check the determined result; the parties concerned must rather rely on the accuracy of the measurement. Hence, the central concern of legal metrology is to protect and ensure that trust. In this context, legal metrology makes a lasting contribution to a functioning economic system by simultaneously protecting the consumers.

The International Organization of Legal Metrology (OIML) was set up to assist in harmonizing such regulations across national boundaries to ensure that legal requirements do not lead to barriers in trade. Software requirements for this purpose are formulated in the OIML D 31 document[90]. WELMEC is the European committee to promote cooperation in the field of legal metrology, for example by establishing guides to help notified bodies (responsible for checking the measuring instruments) and manufacturers implement the Measuring Instruments Directive described below. We will take a closer look at the European directives and guides formulated by WELMEC,

i.e. the WELMEC 7.2 Software Guide[114] described below.

### 2.1.1 Measuring Instruments Directive

Directive 2014/32/EU of the European Parliament and of the Council[88], which is based on Directive 2004/22/EC[87], known as the Measuring Instruments Directive (MID), are directives by the European Union to establish a harmonized European market for measuring instruments, which are used in different member states. The aim of the MID is to protect the consumer and to create a basis for fair trade and trust in the public interest. The directive is limited to ten types of measuring instruments that have a special economic importance because of their number or their cross-border use. These are:

1. water meters,
2. gas meters and volume conversion devices,
3. active electrical energy meters,
4. heat meters,
5. measuring systems for the continuous and dynamic measurement of quantities of liquids other than water,
6. automatic weighing instruments,
7. taximeters,
8. material measures,
9. dimensional measuring instruments,
10. exhaust gas analyzers.

The MID defines basic requirements for these measuring instruments, e.g., the protection against tampering and the display of billing-related readings. Each measuring instrument manufacturer themselves decide which technical solutions they want to apply. Nevertheless, they must prove to a notified body that their instrument complies with the MID requirements. The notified bodies that must be embraced by the manufacturers are denominated by the member states. In Germany, for example, the Physikalisch-Technische Bundesanstalt (PTB) is such a notified body. The PTB is furthermore the German national metrology institute providing additional scientific and technical services, which is why it achieves the demanded technical expertise needed. In general, the combination of technical expertise related to

the measuring instruments, competence for the assessment, monitoring of product-related quality assurance systems and experience with European regulations are required. Additionally it is of particular importance that the notified body is independent and impartial.

### 2.1.2 WELMEC

WELMEC is the European cooperation responsible for legal metrology in the European Union and the European Free Trade Association (EFTA). Currently, representative national authorities from 37 countries are part of the WELMEC Committee.

WELMEC Working Groups (WG) are established by the WELMEC Committee for the detailed discussion of issues of interest and concern to WELMEC Members and Associate Members. Currently, there are eight active Working Groups, and one of them (WG7) is solely responsible for software questions and issues the WELMEC 7.2 Software Guide. As of this writing, its current version is WELMEC 7.2 Issue 5[114], with Issue 6 near its completion. The WELMEC 7.2 Software Guide provides guidance to manufacturers and to notified bodies, on how to construct or check secure software for measuring instruments. Although it is based on the MID and its addressed instruments, its solutions are of a general nature and may be applied beyond. The document states that by following this guide, a compliance with the software-related requirements contained in the MID can be assumed.

## 2.2 Software in Legal Metrology

### 2.2.1 MID Software Requirements

The WELMEC 7.2 Software Guide tries to break down the requirements for legal metrology software of the MID to specific technical examples and recommendations. Actually, the last chapter of the guide is solely devoted to document how the proposed guidelines are mapped to these requirements. The important MID software requirements are:

- Reproducibility of measurement results must be guaranteed, even if handled by different users.

Reproducibility implies that a measurement result should not depend on the user/consumer employing the instrument. From the software point of view, different processes with varying access rights performing the same measurement should yield the same result.

- Durability of the measuring instrument's software over a period of time must be guaranteed. A measuring instrument shall be designed to reduce as far as possible the effect of a defect (bug) that would lead to an inaccurate measurement result, unless the presence of such a defect is obvious.

- A measuring instrument shall have no feature to facilitate fraudulent use, and possibilities for unintentional misuse shall be minimal.

The latter points request that the impact of manipulations and bugs are reduced as far as possible.

- A measuring instrument shall be designed to allow the control of the measuring tasks after the instrument has been placed on the market and put into use. Software for this control must be available. Software identification shall be easily provided by the measuring instrument.

- Evidence of an intervention shall be available for a reasonable period of time.

The former points directly address software requirements for verifying measuring instruments in commission. Validating the software identification ensures that software was not switched or manipulated. Ancillary, an audit trail is needed to log interventions.

- If a measuring instrument has associated software which provides other functions besides the measuring function, the software that is critical for the metrological characteristics shall be identifiable and shall not be inadmissibly influenced by the associated software.

- The metrological characteristics of a measuring instrument shall not be influenced in any inadmissible way by the connection to it of another device, by any feature of the connected device itself or by any remote device that communicates with the measuring instrument.

- Software that is critical for metrological characteristics shall be identified as such and shall be secured.

- Measurement data, software that is critical for measurement characteristics and metrologically important parameters stored or transmitted shall be adequately protected against accidental or intentional corruption.

These points demand a strict separation of legally relevant parts and legally not relevant ones. Hereby, legally relevant parts are all functions that are important for the measurement purpose. Furthermore, legally relevant parts should be protected from any malicious intrusion.

- For utility measuring instruments the display of the total quantity supplied or the displays from which the total quantity supplied can be derived, whole or partial reference to which is the basis for payment, shall not be able to be reset during use.
- The indication of any result shall be clear and unambiguous and accompanied by such marks and inscriptions necessary to inform the user of the significance of the result.
- Easy reading of the presented result shall be permitted under normal conditions of use.
- Additional indications may be shown provided they cannot be confused with the metrologically controlled indications.
- A durable proof of the measurement result and the information to identify the transaction shall be available on request at the time the measurement is concluded.

Finally, there shall be no confusion between data generated from legally relevant modules and data from irrelevant ones, by marking them distinguishable on the screen and on prints. Additionally, relevant data, which is the basis for payment, shall not be deleted or resettable until the payment is conducted.

### 2.2.2 WELMEC

As mentioned in Section 2.1.2, WELMEC is the European cooperation responsible for legal metrology in the European Union and the European Free Trade Association (EFTA). Currently, representative national authorities from 37 countries are part of the WELMEC Committee. WELMEC Working Groups (WG) are established by the WELMEC Committee for the detailed discussion of issues. Currently, there are eight active Working Groups and one of them (WG7) is solely responsible for software questions and issues the WELMEC 7.2 Software Guide (W7.2).

Out of the MID and the assessment of hazards, the WELMEC 7.2 Software Guide defines six risk classes from A–F, evaluating the need for software protection, software examination and software conformity. The risk classes

are ascending in their demand for security, meaning that risk Class A instruments do not need any security-awareness mechanisms (no measuring instrument is classified that low) and risk Class F instruments need the highest (There is also no measuring instrument, included in the regulations of the MID, specified at risk Class F. Still, there are nationally variable ranked measuring instruments, e.g., in Germany, traffic enforcement cameras are viewed as F measuring instruments). Specific groups of measuring instruments are then, in all conscience, assigned to one risk class, e.g., taximeters are assigned to risk Class D. For our purposes, the best way to start is to construct a system architecture for risk Class F, because a system architecture for risk Class F conforms to all requirements of the other classes. For risk Class F measuring instruments, the following three main points must hold true:

1. Software protection against deliberate modifications with sophisticated software tools;

2. Ambitious software assessment with examination of source code has taken place;

3. Software conformity: the software implemented in the individual instruments is completely identical to the approved one.

Before constructing a secure measuring instrument software architecture, it is important to clarify legally relevant parts, because only these parts are critical, while of course ensuring that non-legally relevant parts do not effect legal ones. According to WELMEC 7.2, all modules are legally relevant that make a contribution to or influence measurement results. These modules facilitate auxiliary functions, like displaying data, protecting data, saving data, identifying the software, executing downloads, transferring data and checking received or stored data. The different points are listed in Section 3.1 and collated to their appropriate modules in the system architecture.

Additionally, the W7.2 differentiates between measuring instruments that are built solely for the measuring purpose and the ones that run universal software. The two classes are called P and U. Normally one can say, if a measuring instrument has an operating system installed, it is a U type, else it is situated in the P class. For both classes, four subclasses are defined which deal with following IT functions:

- L: long-term storage of measurement data,

- T: transmission of measurement data,

- D: software download,
- S: software separation.

### 2.2.3 Common Criteria and WELMEC

The ISO/IEC 15408, more commonly known as the Common Criteria for Information Technology Security Evaluation, is, with 26 participating countries, the international standard for IT security. The Common Criteria provide guidance for security evaluation by describing generic requirements for major security functionalities of IT products and assurance measures to be applied to these functionalities. Under Common Criteria, products are evaluated against Protection Profiles - PP (more specifically the Security Target - ST), which are the means to adapt the generic requirements to particular application areas. In this context, functional requirements are the security policies or protections a product claims to implement, and assurance requirements are the controls that the developer has to follow to ensure the realization of these functional requirements.

Basically, the development process of a Protection Profile starts with an analysis of the threats to which the target of evaluation (TOE) is exposed. Furthermore, assurance requirements are collectively tagged with an Evaluation Assurance Level (EAL) that represents the overall confidence stakeholders can have in the security functional requirements, *i.e.*, the confidence that these are actually met by the conforming product. Common Criteria assurance levels range from EAL1 to EAL7, and their meanings are shown in Table 2.1.

Table 2.1: Comparison between Common Criteria (CC) Evaluated Assurance Levels (EALs) and WELMEC risk classes (WRC).

| **CC** | **Meaning** | **WRC** |
|---|---|---|
| EAL1 | Functionally tested | B |
| EAL2 | Structurally tested | C |
| EAL3 | Methodically tested and checked | D |
| EAL4 | Methodically designed, tested and reviewed | E |
| EAL5 | Semiformally designed and tested | F |
| EAL6 | Semiformally verified design and tested | - |
| EAL7 | Formally verified design and tested | - |

Table 2.1 also tries to compare the requirements of the EALs with the risk classes of the WELMEC 7.2 Software Guide (W7.2). The juxtaposition in Table 2.1 is not a fixed mapping; it should be more seen as a guidance

for the future. That is to say, if a product meets EAL4 and was designed, tested and reviewed for measurement purposes, it satisfies the requirements of risk Class E. This comparison justifies the use of Windows or Linux operating systems for measuring instruments that need to be tested for risk Class E requirements, because the latter mentioned GPOSs are partially certified at EAL4. In our opinion, software for measuring instruments for risk Class F should be semiformally designed and tested. Still, it should be noted that measuring instruments classified as F instruments, e.g., traffic enforcement cameras running GPOSs, are frequently approved by notified bodies. Another example are smart meter gateways in Germany, for which the the German Federal Bureau (BSI) demands an EAL4+ certification, hence, in this case, a direct correlation between legal metrology directives and Common Criteria is apparent.

### 2.2.4 Types of Software Separation

As mentioned before the W7.2 formulates three requirements for software separation, which will be discussed in more detail here. These are:

- S1: Realisation of software separation: "There shall be a part of the software that contains all legally relevant software and parameters that is clearly separated from other parts of software."
- S2: Mixed indication: "Additional information generated by the software, which is not legally relevant, may only be shown on a display or printout, if it cannot be confused with the information that originates from the legally relevant part."
- S3: Protective software interface: "The data exchange between the legally relevant and legally non-relevant software must be performed via a protective software interface, which comprises the interactions and data flow."

Additionally the W7.2 also differentiates between low-level and high-level separation. These points are explained in the next sections and analyzed for their conformity to S1-S3.

Manufacturers of measuring instruments want to limit the scrutiny of their software to a minimum, to save time and money. A way of achieving this goal is by separating the software into legally relevant and non-legally relevant parts. If the manufacturer can believably show that some parts of the software have no influence on the measuring task, these parts can be modified and changed without further examination. Additionally, the

modularization of the source code facilitates software testing and minimizes software bugs in the development process.

What does "software separation" in legal metrology mean? Generally, one can say that software separation describes technical measures that prevent non-legally relevant software functions from influencing legally relevant ones. If software separation has been implemented fully and correctly, non-legally relevant software components can and may be exchanged or modified by the manufacturer and even user, after the measuring instrument is in commission. No conformity assessment, i.e. no reexamination of the software, is necessary.

To achieve software separation for measuring instruments under legal control, the requirements of the W7.2 should be followed; concretely, the sections S1-S3, which formulate the requirements for software separation. These requirements describe the application level and assume that the manufacturers control both pieces of software (legally relevant and non-legally relevant) and that they can ensure compliance with the requirements. But if the manufacturers have no full control of all the software parts, the legally relevant software needs to be protected against unknown influences by additional measures. These additional measurements depend on the respective hardware and software platform, e.g., support for the separation of program and data areas, management of the common resources, and access to the system by the user, etc.

For manufacturers and notified bodies (NBs), the software separation as described in the W7.2 has also a second benefit : It helps to decide which depth of testing for the various software components of a measuring instrument needs to be applied, and therefore, reduces the expenses for modifications in software throughout the life cycle. In general, one can say that the primary aim of the modularization according to W7.2 / S1-S3 is to facilitate conformity assessment, and not necessary to hinder unknown manipulations of software parts. Often the used platform of the measuring instrument does not have mechanisms to protect the legally relevant software part against interference from other software components, even if they are separated according to the rules of the W7.2 / S1-S3.

### 2.2.5 Low-Level Separation

According to W7.2, low-level separation means that software separation is realized independently from the operating system within an application domain, i.e., at the programming language level.

Figure 2.1 shows such a separation. Hereby, the source code is modularized into separate files. This is a first step to achieve software separation

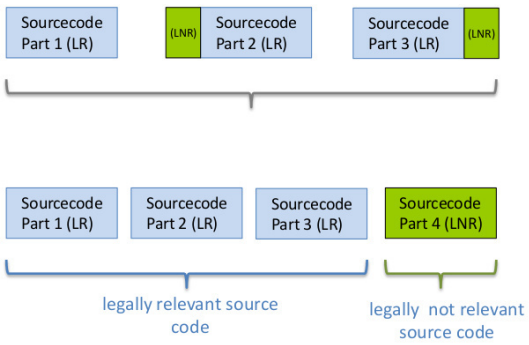



Figure 2.1: Low-level separation according to W7.2

and helps in the coding process because a clean modularized environment makes locating and emending bugs easier. Still, this is not enough if one executable is generated, as can be seen in Figure 2.2.

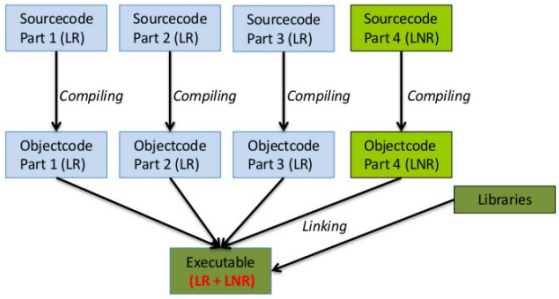



Figure 2.2: Software separation that still leads to one executable

At execution time a single binary is running on the device. In this binary legally relevant functions are again mixed together with non-legally ones, hence S1 and S3 is not satisfied. An example where two separate binaries are compiled can be seen in Figure 2.3.

Hereby, the separation of the two executables is achieved by copying the executables in separate memory banks. The data transfer between the executables can be managed through shared libraries. S1 is being satisfied, still S2 and S3 must be checked. Especially S3 states that the shared libraries must be protective software interfaces, i.e. legally-non relevant software is not allowed to effect legally relevant one in an unwanted way. Hence the libraries are legally relevant software.

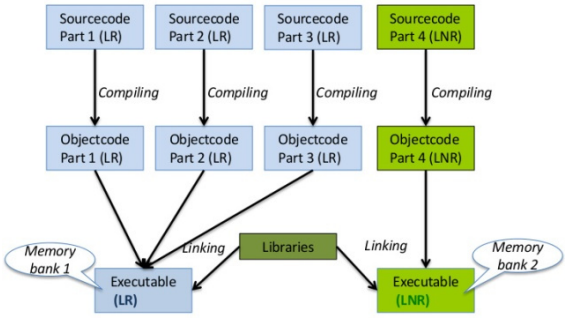



Figure 2.3: Software separation leading to two executables

### 2.2.6 High-Level Separation

High-level separation means that the software modules to be separated are realized as independent objects with the help of the operating system. An example which is similar to the last one is shown in Figure 2.4.

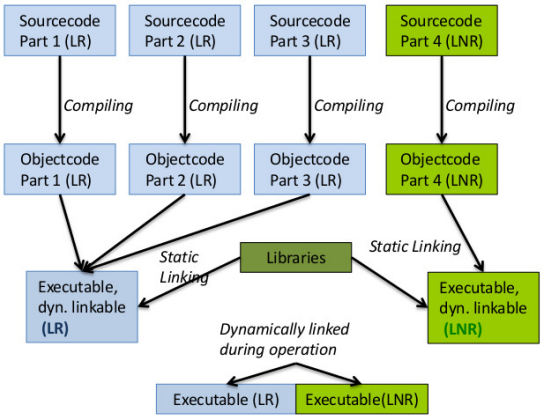



Figure 2.4: Difference between statically linked in libraries and dynamic ones

The different source code files generate different executables. Here again, libraries are the parts of code that are used by both executables. If the libraries are statically linked into the executables, the binaries are independent of each other, and the operating system makes sure that the applications are isolated. For this purpose, the operating system needs mechanisms to make isolation possible, like the use of separate address spaces for different processes. For general purpose operating system like Windows and Linux

this is the case. Still, many known bugs generate vulnerabilities in these operating systems that are used to subvert the isolation.

If the libraries are dynamically linked into the binaries, they represent shared code and need to be checked accordingly to S3 as mentioned before. These libraries can then be used for communication purposes between legally relevant and non-legally relevant software.

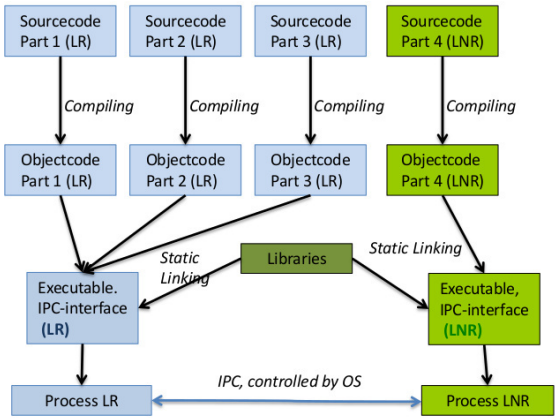



Figure 2.5: Inter Process Communication (IPC) controlled by the operating system

Lastly, Figure 2.5 shows a completely controlled communication by operating system mechanisms. Here, the used libraries are statically linked into the binaries, constructing separate isolated processes. These processes can than communicate through Inter-Process Communication (IPC) with each other, which is regulated by the operating system that needs to be trustworthy.

### 2.2.7 Hardware Separation

A secure way to separate software is by directly using separate hardware, e.g. two central processing units (CPUs). One is solely devoted to calculate legally relevant tasks and one is doing only non-relevant calculations, as can be seen in Figure 2.6.

The communication interface is legally relevant and needs to be checked to fulfill S3, if the two CPUs are communicating data to each other or use the display together. This method is the most expensive one, because it needs additional hardware, which the other methods do not need. A cheaper way is possible with virtualization technologies (see Section 2.3.4), hereby, standard

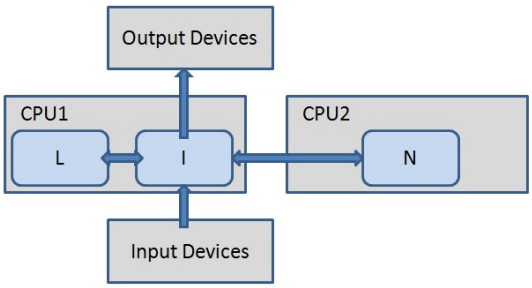



Figure 2.6: Software separation through hardware

operating systems that offer great functionality, a familiar user interface and many working drivers can be used securely only by software measures. This thesis will describe such a software architecture in Chapter 3.

### 2.2.8 Software Identification

Software validation is an important part of the type examination process for measuring instruments in legal metrology. After type examination engineers check the software, it must be made sure that it will not be changed in commission before reexamination. Therefore, in many states, national laws oblige the manufacturer of measuring instruments under legal control to output a software identifier for the user and the market surveillance. This identifier should be created in a way that data integrity can be inferred. Often, this is done by just calculating a checksum over all the files that seem to be important for the measuring task and showing it on the screen. Considering that remote maintenance and automatic scanning of the devices will be the next step to achieve remote integrity checking, a file list containing as much information as possible is preferable.

As already mentioned, estimations speak of more than 25 billion connected devices in the year 2020[111]. As storage units have become smaller and cheaper, these devices can already save millions of files. Considering that in the future the automatic data exchange between these devices will blossom, the creation of a small data structure listing all the files on a device is important. Despite being small, this data structure should also be quickly traversable. It should list as much information about a file as possible to check, for example, the file names, the file formats, the sizes, and the checksums.

Additionally to the description of the system architecture, this thesis describes such a data structure which makes use of succinct approaches to

store trees (Chapter 5). In this structure, a fast file search is made possible by using space-efficient algorithms to store the file names. Hence, the data structure is not only usable as a file list, but can easily be used as the fundamental structure for a read-only file system in which files can be located and listed efficiently.

## 2.3 Secure Software Systems

Before going into the technical details of our system architecture, it needs to be clarified what software security, especially confidentiality, integrity and availability, stand for in this document, because the literature does not always give a consistent definition.

Security is the ability of a component to protect resources for which it announces protection responsibility, and the component's security policies are its security-enforcing properties and requirements. Generally, security policies try to enforce confidentiality, integrity and/or availability.

- Confidentiality ensures no unauthorized disclosure of information;
- Integrity prevents modifications or corruptions of a resource without authorization;
- Availability ensures that authorized users have access to resources whenever needed.

For every one of these three points, authorization plays a key role and so does authentication. Authentication ensures that an entity is who it claims to be, and authenticity means that the data and communication partners are genuine. After authenticity is determined, authorization either grants the right to use a resource or denies that privilege.

### 2.3.1 Creating a Secure System

An operating system is the essential component for hardware abstraction in a software system. It acts as an intermediary between programs and the hardware and, from the security point of view, plays the most important part in constructing a secure system. Generally, operating system architectures are subdivided into two main designs, the monolithic kernel and the microkernel system architecture shown in Figure 2.7. It should be noted, that often, another architecture is mentioned, e.g., Windows is sold as a hybrid kernel architecture, combining aspects of a microkernel and a monolithic kernel architecture. We consider this kernel to be a "smaller" monolithic

kernel, because in contrast to a microkernel, many (nearly all) operating system services are in kernel space, like in a monolithic kernel.

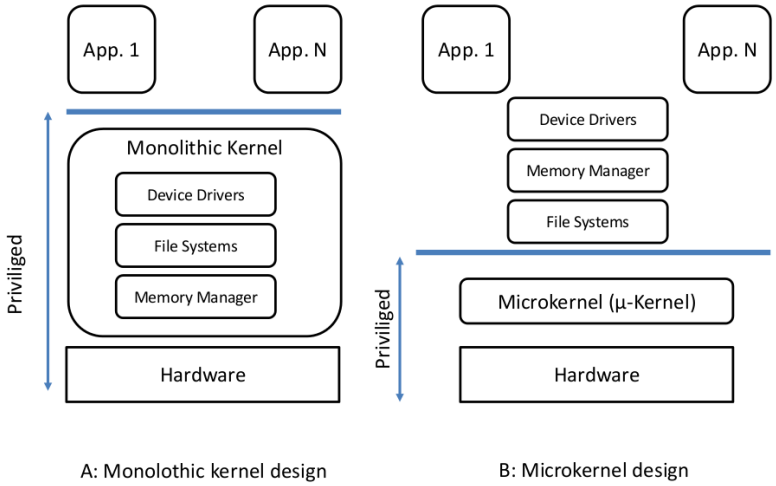



Figure 2.7: Comparison between a monolithic kernel design (**A**) and a microkernel (**B**).

In a monolithic kernel system architecture, the entire operating system is working in privileged mode sharing a single memory space with the system software, such as file systems and complex device drivers with direct access to the hardware; in Figure 2.7, the privileged mode modules have rounded corners and are depicted in red. The advantage of this architecture is performance, because user applications are able to access most services, e.g., I/O devices and TCP/IP networking, with a simple and efficient system call. The disadvantage of this approach is the resulting large trusted computing base (TCB). The TCB refers to those portions of a system (software and hardware) that are needed in a system to ensure that it works as expected. Therefore, it must be trustworthy.

In the microkernel design, the microkernel is the only software executed at the most privileged level. Hence, in contrast to a monolithic design, services are implemented in separate processes; in Figure 2.7, represented as blue components with sharp edges. The motivation to place as much functionality as possible in separate protection domains, not running in privileged mode, is to gain stability, because, for example, a crash in the network stack that would have been fatal for a monolithic system is now survivable. Consequently, the goal of this architecture is to keep the TCB small and under control, as even well-engineered code can have several defects per thousand SLOC[21]. Hence, a bigger system has inherently more bugs than a small sys-

tem and often a bigger attack surface. For comparison, modern microkernels have around 15K SLOC or less, and the monolithic kernel of Linux (version 3.6) at least 300K SLOC to a maximum of 16M SLOC, depending on the configuration.

### 2.3.2 Security Kernels

A security kernel in a system ensures that subjects have access only to objects that are given to them by a security policy. A common way of expressing these requirements is given by the acronym NEAT, which defines the following criteria for security kernels:

- Nonbypassable: The safety concept of the system cannot be bypassed. Components cannot create communication paths, different from the determined ones to bypass the safety concept.
- Evaluatable: The security architecture is small and has a low level of complexity, in order to make a formal verification possible. The components must be small and modular, to facilitate verification.
- Always-invoked: The safety concept is always active. Every access and communication must be checked and accepted by the security architecture (the security architecture normally verifies only the first access to an object, all other requests are forwarded without a recheck to speed up the process).
- Tamper-proof: The system has a strict access control management, specifically handling the modification of data or code. The security architecture strictly controls which components can modify the system to prevent unauthorized changes.

A security kernel is not necessary an operating system kernel. Rather, it refers to those components that perform the functions of a reference monitor within an operating system kernel. If access to a sensitive object is requested, the system first asks this reference monitor for permission. The reference monitor itself then checks the access rights by some kind of policy table. Hereby, the objects can be hardware, e.g. CPUs, memory segments, harddisk blocks; or software, e.g. processes or files.

Usually general purpose operating systems use a Discretionary Access Control (DAC) model, in which the individual users are able to decide who can have what access to their documents. In systems were stronger security is needed, the systems themselves should have additional rules that decide what objects can be accessed by whom, e.g. in a hospital, the doctor should not

be able to give the janitor reading or writing rights to his patience database. These systems should also have so called Mandatory Access Control (MAC) models in place, for example like the Bell-LaPadula model or the Biba model.

A well-known example of a security kernel is SELinux[105], an implementation of the Flux Advanced Security Kernel (FLASK)[7] for Linux. SELinux replaces the normally used Discretionary Access Control (DAC) by the more restrictive one, the Mandatory Access Control (MAC). In general, FLASK utilizes a security server to make access decisions, and object managers that enforce those decisions. This separation of access control decision from enforcement, allows the support of flexible mandatory access control.

### 2.3.3 Separation Kernels

A separation kernel[116] is a small software component, which divides the system into partitions - alternatively also called domains - (Figure 2.8).

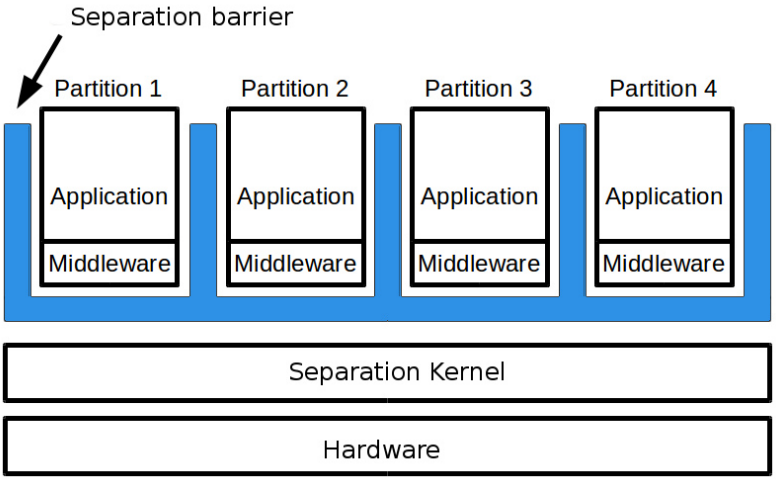



Figure 2.8: A separation kernel dividing the system into partitions.

A complete separation of the partitions from each other, both in time and space, is ensured. Partitions can only communicate through strictly controlled channels. The term separation kernel arises from the field of embedded systems, where isolation of individual components often plays an important role. Accordingly, the requirements for separation kernels are very high. The ARINC653[104] specification defines requirements that need to be met by operating systems, if they are used in applications where functional safety must be guaranteed. Hereby, four requirements must be met by a

separation kernel:

1. Temporal separation
2. Spatial separation
3. Information flow control
4. Fault isolation

The term separation kernel is often used in conjunction with Multiple Independent Levels of Security / Safety (MILS)[2,8] (see also Section 2.3.5). Here, the separation kernel represents the lowest layer of this architecture. In the partitions, a middleware layer is running as a connecting layer to the applications. This is needed because the provided separation kernel interfaces are often very rudimentary and provide only a minimum of functionality, to keep the complexity in kernel low. Therefore, the middleware implements missing functionality, often in the form of libraries to offer applications a standardized interface (e.g. POSIX). These libraries contain a variety of functions such as memory management, threading or mathematical functions. However, the middleware can also offer a virtualization layer that offers the possibility to run operating systems in a partition with a wider range of functions, for example, Linux or Windows. We use this approach for our architecture, which is explained in Chapter 3.

### 2.3.4 Virtualization

A major drawback in constructing a new software system on a microkernel is that drivers and software libraries available for known GPOSs, e.g., Linux, which uses a monolithic kernel design, need to be ported, or in the worst case, completely rewritten. Virtualization seems to be the right solution to incorporate the best implementations of both architectures in a single system. Through virtualization, device drivers and software libraries can be reused, which makes the implementation of our system architecture for a manufacturer as easy as possible.

Virtualization can be divided into two main approaches[69]. Pure virtualization, sometimes also referred to as faithful or full virtualization, supports unmodified guest operating systems, running atop another kernel, safely encapsulated. The advantage of this approach is that closed-source operating systems are theoretically directly executable. Commodity processors often do not have adequate support for pure virtualization, requiring complex technologies, such as binary translation, to be used[1]. For the second approach, called para-virtualization, the guest operating system is presented with an

interface that is similar, but not identical to the underlying hardware to make virtualization possible. To improve performance and allow virtualization at all, the guest operating system is often modified. The approach used depends on the guest operating system that should be employed and the hardware features available.

In the past, embedded systems used to be relatively simple devices, and their software was dominated by hardware constraints, which made virtualization unattractive. Nowadays, most embedded systems have the characteristics of general purpose systems and increasingly have the power to actually run applications built for PCs. This power together with the low development costs for a board combined with virtualization technologies like ARM TrustZone[39] makes virtualization in the embedded world, and therefore, in measuring instruments, attractive.

A strong motivation for virtualization is security. By running an operating system in its own environment safely encapsulated—a so-called virtual machine (VM)—the damage of an attack is restricted to this virtual machine, because access to the rest of the system can be prohibited, as shown in Figure 2.9.

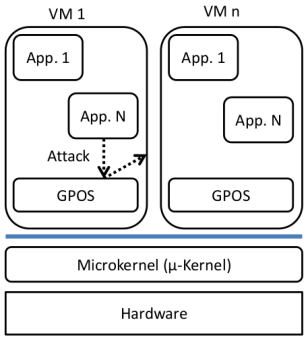



Figure 2.9: The general-purpose operating system (GPOS) of the VM in the middle is compromised by a pernicious application. Due to isolation, the other VMs are not vulnerable.

The access is not just restricted to the VM; the whole hardware access can be redirected through the underlying kernel through virtualized device drivers, preventing direct communication with the hardware or any hardware access at all. This isolation can only be assumed if the privileged software managing the virtual machines, called the virtual machine monitor (VMM) or hypervisor * is correctly implemented. For example, in Xen—a popular

*In this thesis, we do not differentiate between a VMM and a hypervisor, as is sometimes done. Both refer to the underlying microkernel

VMM—all VMs depend not only on the hypervisor, but on a full operating system, because the first VM—called dom0—is launched by default with direct access to the hardware. From the dom0, the hypervisor can be managed, and unprivileged VMs can be launched. This makes the TCB very large and the construct vulnerable; actually, fully exploitable vulnerabilities to subvert the hypervisor and take over the whole system have been found in the past[115]. In our opinion a microkernel, because of its minimality principle, seems to be a good choice for implementing a hypervisor[56,75,70,57,54,94].

### 2.3.5 Policies of a Secure System

Multiple Independent Levels of Security/Safety (MILS) is a high-assurance security architecture based on the concepts of separation and controlled information flow. The foundation of the system is a small kernel, as used in our design, implementing a limited set of critical functional security policies. This special kernel, often called the separation kernel or partition kernel, implements the policies for information flow control, data isolation, damage limitation and period processing[8].

- Information flow control ensures that information cannot flow between partitions unless explicitly permitted by the system security policy.
- Data isolation ensures that a partition is provided with mechanisms, whereby isolation within it can be enforced.
- Damage limitation ensures that a bug or attack damaging a partitioned application cannot spread to other applications.
- Period processing ensures that information from one component is not leaked into another one through resources, which may be reused across execution periods.

To guarantee these policies, the separation kernel must ensure the following properties called NEAT (see also Section 2.3.2):

- Non-by-passable: a component cannot use any communication path, including lower level mechanisms, to bypass the security functions.
- Evaluable: security assurance must be given by evaluable (even mathematically verifiable) security claims.
- Always-invoked: the security functions are invoked each and every time.

- Tamper-proof: applications must not be able to tamper with the security policy or its enforcement mechanisms (e.g., by exhausting resources or overrunning buffers).

As stated in Section 2.3.1, modern monolithic kernels and, of course, whole GPOSs consist of tens (sometimes hundreds) of millions of SLOC. Hence, they are too difficult and expensive to evaluate. The separation kernel, which, with sometimes no more than 10K SLOC, is small enough to be thoroughly evaluated and mathematically verified for the highest assurance level. The applications managing sensitive data are then built on top of the secure separation kernel. An advantage of this approach is its modularity, allowing software of varying security demands to run on the same microprocessor by means of software partitioning through the separation kernel. This way, the MILS security policies are also stacked, meaning that a module layered atop the separation kernel cannot circumvent the enforced restrictions, which the separation kernel defines for it.

### 2.3.6 Principles to Satisfy Security Properties

In our opinion the MILS system architecture, especially the separation kernel, together with VMs as its components, furnishes a great base for high-assurance software combined with the ability to run GPOS applications. An example of a well-designed separation kernel is Integrity. The Integrity operating system from Green Hills is the world's first software to achieve an EAL 6+/high robustness certification. Integrity was constructed with the PHASE (Principles of High Assurance Software Engineering) methodology in mind[86]. PHASE is a methodology that is not restricted to separation kernels; it can be used to create reliable software in general. In our case, reliability means to achieve the assurance level required for lawfully-used measuring instruments, which is indisputably high. As an aid to fulfill this goal, PHASE names five principles to be upheld:

1. Minimal implementation;
2. Component architecture;
3. Least privilege;
4. Secure development process;
5. Independent expert validation;

As previously highlighted, the security of a system highly depends on the amount of code used to solve a problem. Therefore, finding a minimal implementation is one of the most important tasks in secure software development. In our proposed architecture, this also implies using the minimal configuration for the GPOS running in the virtual machine, e.g., while compiling Linux, only needed modules should be appended to the kernel.

A component architecture follows the principle of damage limitation: critical operations should be protected from non-critical ones by placing them in different components. Damage limitation is not the only benefit of a component-based architecture; other points to be named are the facility to swap modules for costumer-specific changes and the overall improved maintainability. A downside is the need for well-defined interfaces for communication between the components, in our case VMs. The principle of the MILS system architecture goes hand in hand with the component architecture principle.

Every component in the system should be designed with the least privilege concept in mind. The VMs should be given access only to those resources, e.g., I/O devices, that they absolutely require. In our proposed system, direct access to I/O devices is omitted as much as possible.

A secure development process is the basis for high-assurance software. It covers, for example, coding, testing, formal design and formal proof of a system and paves the way for an EAL6/7 certification. Our proposed framework should use all tools available to safeguard the development of the system from the beginning, like model-driven design, coding standards and static and dynamic code analysis.

In legal metrology, independent expert validation is stipulated and conducted by notified bodies. The manufacturer must provide its design documentation together with user manuals for testing purposes, e.g., to check if all input possibilities are documented and permitted. Additionally, complete access to the source code must be granted for risk Classes E and F measuring instruments.

## 2.4 Succinct Data Structures

As mentioned before, the second point this thesis describes, is a new approach to check file-system integrity. Hereby, a new succinct data structure will be created, hence, this section gives an introduction into succinct data structures. For clarification, it should be noted that all the results are in the word-RAM model of computation, where it is assumed that the machine consists of words of width $w$ bits that can be manipulated in $O(1)$ time by

a standard set of arithmetic and logical operations, and further that the problem size $n$ is not larger than $O(2^w)$.

Succinct data structures have been one of the key contributions to the algorithms community in the past two decades. Their goal is to represent objects from a universe of size $u$ in information-theoretical optimal space $\lg u$ bits of space.[†] Apart from the bare representation of the object, fast operations should also be supported, ideally in time no worse than with a "conventional" data structure for the object. For this, one usually allows extra space $o(\lg u)$ bits[60].

A prime example of succinct data structures are ordered rooted trees, where with $n$ nodes we have $\lg u \approx 2n$. In 1989, Jacobson made a first step towards achieving this goal, by giving a data structure using $10n + o(n)$ bits, while supporting the most common navigational operations in $O(\lg n)$ time[60]. This was further improved to the asymptotically optimal $2n + o(n)$ bits and optimal $O(1)$ navigation time by Munro and Raman[82]. Note that a conventional, pointer-based data structure for trees requires $\Theta(n \lg n)$ bits, which is off by a factor of $\lg n$ from the information-theoretical minimum.

Since the work of Munro and Raman, the research on succinct data structures has blossomed. We now have succinct data structures for bit-vectors[93], permutations[80], binary relations[6], dictionaries[92], suffix trees[102], to name just a few.

The practical value of those data structures has sometimes been disputed. However, as far as we know, in all cases where genuine attempts were made at practical implementations, the results have mostly been successful[44,62,48] etc., to cite some recent papers presented in the algorithm engineering community. Further examples of well-performing practical succinct tree implementations will be mentioned throughout this thesis.

### 2.4.1 Rank and Select

Many succinct data structures make use of two fundamental operations, called *rank-* and *select*. In this thesis, these operations on $S$, with $S[1,n]$ being a *bit-string* of length $n$, are defined as follows:

- $\mathsf{rank}_1(S,i)$ gives the number of 1's in the prefix $S[1,i]$
- $\mathsf{select}_1(S,i)$ gives the position of the $i$'th 1 in $S$, reading $S$ from left to right $(1 \leq i \leq n)$

Operations $\mathsf{rank}_0(S,i)$ and $\mathsf{select}_0(S,i)$ are defined similarly for 0-bits. $S$ can be represented in $n + o(n)$ bits such that rank- and select-operations are supported in $O(1)$ time[82].

[†]Function lg denotes the binary logarithm throughout this thesis.

The approach used to achieve this, is to divide the bit-string $S$ into blocks of fixed size, e.g. 64 bit, and store the number of ones before the blocks. These blocks can be combined into bigger blocks, e.g. size of 4096 bits, often called super-blocks, which again store the number of ones at the position in front of each super-block. After each super-block the number of ones for the upcoming smaller block is reset to 0. Every super-block needs $\lg n$ bits and with the example of 4096 bit size super-blocks, each small block needs only $\lg(4096) = 12$ bits to store the number of ones till this position. $\mathsf{rank}_1(S,i)$ can then be performed by accessing the blocks and adding them together. Getting the number of ones inside a block is done by bit-operations and table-lookups to speed up the process. The approach for $\mathsf{select}_1(S,i)$ is similar but a little bit trickier, a nice description can be found in[23] where a three level directory structure is used.

### 2.4.2 Wavelet Trees

The operations *rank-* and *select* have been extended to sequences over larger alphabets, at the cost of slight slowdowns in the running times[45,9]. In this section a practical approach is discussed, called *wavelet tree*[47]. A wavelet tree is constructed as follows: First each character $c$ in a text $S$ is assigned to exactly one bit (a 0 or a 1). The root node $v_1$ is situated on the first level and contains the bit-vector $B_1$ and the actual text $S_1 = S$. Now the tree is built recursively: If a node $v$ contains a text $S_v$ that has at least two different characters, then two child nodes $v_l$ and $v_r$ are created. All characters which are marked with a 0 go to the left node and all other characters go to the right node. Note that at the end the $S_v$'s of every node are not saved, only their bit vectors $B_v$ with the *rank-* and *select* data structures, and the mappings "$c$ to leaf" and "leaf to $c$". If a balanced wavelet tree is constructed, in which the first half of a node's alphabet is written into the left child and the other half into the right one, the mappings from "$c$ to leaf" and from "leaf to $c$" do not need to be stored, and the tree can still be easily traversed. Figure 2.10 shows such a balanced wavelet tree for $S = 303302013032012010010$.

The advantage of these wavelet trees is that $\mathsf{select}_c(S,i)$, $\mathsf{rank}_c(S,j)$ and access($j$) queries for an alphabet of size $\sigma$ can be answered in $O(\lg \sigma)$ time for every character $c$ in $S$ and position $j$ in $S$, while using only $n \lg \sigma + o(n \lg \sigma)$ space.

To give an example, we look at Fig. 2.10. Here, $\mathsf{rank}_3(S_1,6)$ can be calculated as follows. We know that 3 is in the second half of our alphabet ($0, 1$ are represented as a 0 in $B_1$ and $2, 3$ as a 1). So first $\mathsf{rank}_1(B_1,6) = 4$ and then $\mathsf{rank}_1(B_3,4) = 3$ (here again 3 is represented as a 1 in $B_2$ because it is in the second half of the alphabet, 2 in the first), meaning in total

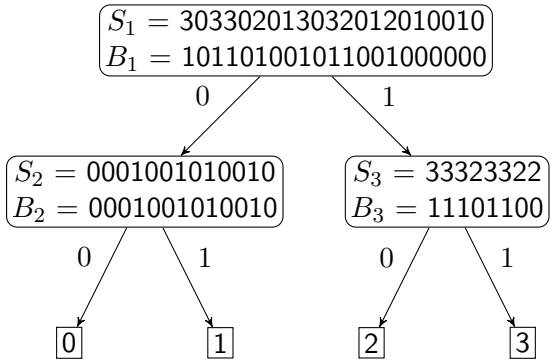



Figure 2.10: Illustration of a balanced wavelet tree.

$\mathsf{rank}_3(S_1,6) = 3$. For select a similar procedure is being used only that now we start from the leaves. To calculate $\mathsf{select}_1(S_1,2)$, we know that 1 represents a 0 in $B_1$ and a 1 in $B_2$. Hence, we get $\mathsf{select}_1(B_2,2) = 7$ and afterwards $\mathsf{select}_0(B_1,7) = 14$, so the result is $\mathsf{select}_1(S_1,2) = 14$.

### 2.4.3 Storing Trees Succinctly

As mentioned, there are several ways to represent an ordered tree with $n$ nodes using $2n$ bits. The best known are the "level order unary degree sequence" (LOUDS), the "balanced parantheses" (BP) and the "depth first unary degree sequence" (DFUDS)[81,10,60]. A comparison of these structures is depicted in Figure 2.11.

The LOUDS is formed by performing a breadth-first traversal (BFT) on the tree, starting with an artificial super-root which is added in front of the real root node and connected to it. At every step of the BFT $1^d0$ is added to the LOUDS for each node, with $d$ being the number of children of the respective node.

The BP and DFUDS use the depth-first traversal (DFT) for their construction. The BP is constructed as follows, when the DFT descends a level an opening parenthesis is written out, and when it ascends, a closing one is written out.

The DFUDS combines the BP and the LOUDS. Hereby, when the DFT descends to a node, $d$ open parentheses are written out, with $d$ being the number of the children of that node. After the node traversal a closing parenthesis is written out. Like in the LOUDS, an opening parentheses is added at the beginning, similar to the artificial super-root.

For the LOUDS we just need *rank*- and *select* to enhance the data struc-

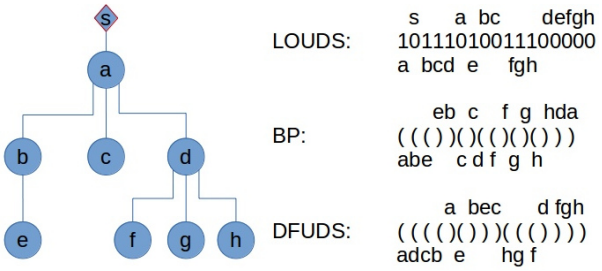



Figure 2.11: Comparison between LOUDS, BP und DFUDS ($s$ is the artificial super-root)

ture with navigational operations. The BP and the DFUDS need some more structures:

- enclose($S$,$p$): Finds the pair of parentheses, which encloses the open parentheses at position $p$ most tightly and returns the position of the open parenthesis of the pair of parentheses.
- findclose($S$,$p$): Returns the position of the closed parentheses which belongs to the open parentheses at position $p$.
- findopen($S$,$p$): Returns the position of the open parentheses which belongs to the closed parentheses at position $p$.

All these functions can be implemented in $O(1)$ time with $o(n)$ space. As can be noticed, all succinct data structures for trees[60,81,10,36,25] must have the freedom to fix a particular naming for the nodes; natural such namings are post- or pre-order[60,81,10], in-order[25], and level-order[60].[‡]

As the LOUDS is easier to implement and needs only *rank*- and *select*, it practically also uses less space. Therefore, we think the best choice is to use it as the fundamental structure of the file system structure explained in Chapter 5.

We identify the nodes with their level-order number, since both the 1- and the 0-bits appear in this order in $B$. Augmenting rank and select to the LOUDS, results in the total space of $2n + o(n)$ bits, where the basic navigational operations on trees are simulated in $O(1)$ time: Getting the parent of node $i$ ($1 \leq i \leq n$) is done by jumping to the position $y$ of the $i$'th 1-bit in $S$ by $y = \mathsf{select}_1(S,i)$, and then by counting the number $j$ of 0's

[‡]If the naming is arbitrary (e.g., chosen by the user), then $n \lg n$ bits are inevitable, since *any* memory layout of the nodes has $n!$ possible namings.

that are present before $y$, with $j = \mathsf{rank}_0(S,y)$. The resulting $j$ represents the level-order number of the parent of $i$. Listing the children of $i$ is done by going to the position $x$ of the $i$'th 0-bit in $S$ by $x = \mathsf{select}_0(S,i)$, and then iterating over the positions $x+1, x+2, \ldots$, as long as the corresponding bit is '1'. For each such position $x+k$ with $S[x+k] = 1$, the level-order numbers of $i$'s children are $\mathsf{rank}_1(S,x) + k$, which can be simplified to $x - i + k + 1$.

## 2.5 How much Security do we need?

A point which is very interesting is formal verification of the microkernel that is used in the system architecture we construct in Section 3. Code verification faces great challenges with many compromises that arise[28]:

- the logical foundations are quite problematic
- the code size needs to be fairly small
- the underlying execution model of C and assembly often makes severe simplifications
- the abstract model must be confirmed by the underlying models and not based on the informal understanding of the lower layers

Despite the challenges, the first formal verifications of operating system kernels already took place in the 1970s. The Provably Secure Operating System (Psos) comprised hardware and software and aimed at a useful general purpose operating system with demonstrable security properties, as described in a report of 2003[84]. Still, no code proofs were undertaken. In the UCLA Secure Unix project[113], verification of a kernel supporting threads, a capability-based access control, virtual memory, and device accesses, was done. It relied on the assumptions that the compiler and the hardware work correct. The Kit kernel[11] that relied on a LISP execution model, is the first kernel that can be considered as formally verified.

While these projects pioneered the field, none of them come up with an operating system kernel written in an imperative language with full implementation proofs. The main reason for this were the missing or to a large extend underdeveloped tool environments, which has changed over the years. One example is given in[19], where memory isolation properties of a hypervisor developed at the TU Berlin are shown using a language and tools from a company called Prove & Run. Other examples from the L4 microkernel family[§] are:

[§]We are using an L4 microkernel called Fiasco.OC in our tests in Section 3.4

- The verification of the Fiasco microkernel is the aim of the VFiasco project[58]. Besides model checking of basic safety properties for a simplified version of the Fiasco Inter-Process Communication (IPC), also properties concerning ready and waiting lists have been verified.
- The L4.verified project focuses on the seL4 kernel[55]. It is based on the ARM11 platform and comprises 8,700 lines of C and 600 lines of assembly code. A machine-checked proof of the functional correctness of the seL4 microkernel is provided with respect to a formal and high level description.

For measuring instruments, an important aspect is the correct execution of programs. In Section 3.3 we describe mechanisms to detect rootkits (programs that allow an attacker to gain root access) by checking the snapshots of the virtual machines. A newer project describing such an approach can be found in[112], where a rootkit detector for Android devices is running in a separate virtual machine called detector VM. The detector VM can initiate a state snapshot of the supervised Android VM, which is stored in a dedicated memory region. The snapshot buffer appears as special (guest physical) memory region in the detector VM, from where a kernel driver maps it into the rootkit detectors, which run as user-space processes. There are also other opportunities that enable software to run securely in an unsecure environment. In[22] the authors describe a software solution by changing the FreeBSD 9.0 kernel to divide the address space of the processes in three partitions, the kernel space, the user space, and a new one, which they call the ghost memory partition. With the help of the compiler and an extra allocation function (to get ghost memory), the ghost memory partition can be used, and guarantees confidentiality and integrity of the application even if the operating system is not trustworthy. One hardware solution that can also be named here, are the Intel Software Guard Extensions SGX which enable similar execution environments with a set of new CPU instructions that can be used by applications to set aside private regions of code and data.

So, in total the question arises how much security do we need, or better said, can we demand for measuring instruments in legal metrology? As mentioned in Section 2.2 the directives and guides demand for mechanisms that spot changes in the software. Hereby, the documents only talk about an adequate security and that a software identifier shall be printed on the screen for the user and the market surveillance. This identifier should change if software parts have changed. Still, there is no concrete example how this identifier shall be created, and manufacturer shall not be restricted, especially no hardware shall be stipulated. Our approach in this thesis, explains

a solution, which is new and flexible and fulfills these requirements without dictating the manufacturers which hardware to use. A hypervisor architecture based on a microkernel seems to be the most flexible approach. First of all, there are microkernels that run on nearly every hardware architecture like the Fiaso.OC kernel which runs on ARM, PowerPC, Sparc, x86, x86_64, and more systems. Secondly, to the Fiasco.OC kernel for example, Linux has already been ported and with the next step to full virtualization technologies in embedded devices (e.g. ARM TrustZone), operating systems are directly runnable on these hypervisors. As mentioned, there is an ongoing process to verify these kernels so in future the manufacturer themselves can switch to a verified kernel without changing their software. Nevertheless, a formally verified microkernel, or better said, separation kernel would just make sure that the separation between the VMs holds, the VMs themselves could still be malicious, because GPOS are running inside of them, making the effort and demand for a formally verified kernel not justifiable.

Supervising the whole system through a detector VM seems very cost intensive in time and effort (for example creating and analyzing the snapshots, switching on every critical system call e.g. forks), we think, that scanning the file-system and constructing the data-structure we called FLOUDS (see Chapter 5) in a system architecture that runs on minimal configured VMs seems more than enough security. Still we have such a detector VM in our architecture called Inspector VM (see Section 3.2.3) that can be used for this purpose. Mechanisms that enforce mandatory access control (MAC) like SELinux (see Section 2.3.2) can still be running in the VMs, so we build a flexible secure system architecture without discarding other security mechanisms.

# 3
# System Architecture

## 3.1 The Framework

Our proposed framework, shown in Figure 3.1, consists of three approaches.

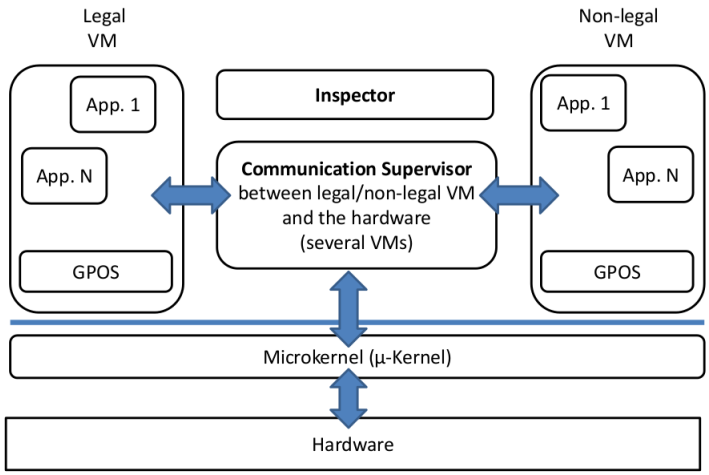



Figure 3.1: The framework.

Firstly, we separate the legally relevant parts from the irrelevant ones by putting them in different virtual machines. Secondly, we make sure that their virtual machines have no direct access to I/O devices and that they are scanned and integrity checked by a dedicated VM (the Inspector). Lastly, we construct a secure framework (the Communication Supervisor) which provides services to these VMs. This framework, also consisting of separated

VMs, monitors the information flow and correctly delegates requests from and to I/O devices. It is shown in more detail in Figure 3.2.

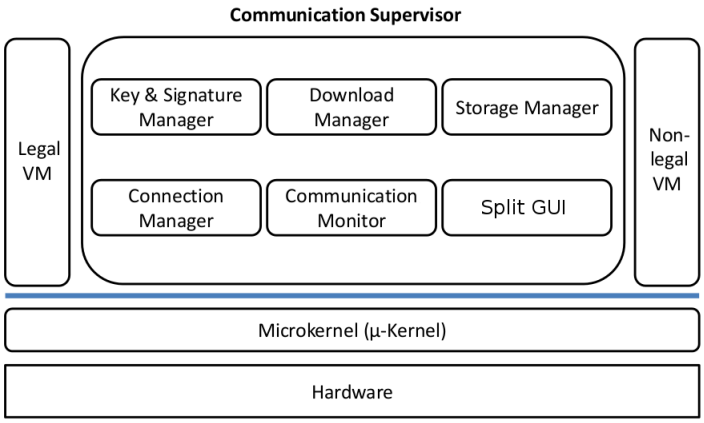



Figure 3.2: Communication Supervisor.

In the legal VM (L VM), all of the computations that are needed for the measurement procedure are executed, e.g., image processing in a traffic enforcement camera. The non-legal VM (N VM) is only allowed to run software that has no measurement purpose, e.g., like showing the manual or starting a calculator. This strict separation ensures that non-legally relevant software has no effect on the legally relevant one, as postulated in the MID. The other blocks form our framework. Their general purpose is to supervise the communication between the L/N VM and the hardware. These modules can be native microkernel processes or VMs themselves. In our opinion, the VM approach seems to be the better one, because communication between the individual VMs can take place through network protocols already implemented in the GPOS, and the GPOS device drivers can be used[53,57]. Through the implemented network stacks, virtual private networks (VPNs) can be created to encrypt communication. The VM approach even allows modules to be transferred out to different computers, creating a distributed system over a network. A disadvantage of this concept is the invalidation of the minimal implementation principle, which should be counteracted by using minimal configurations for the GPOS.

Our framework consists of inclusively legally relevant modules, fulfilling legally relevant functions, as demanded in the WELMEC 7.2 Guide and mentioned in Section 2.2.2. The mapping of the functions to the modules is as follows:

- Split GUI: displaying data;

- Key & Signature Manager: protecting data;
- Storage Manager: saving data, recording modifications;
- Inspector: identifying the software, software integrity checking;
- Download Manager: executing downloads;
- Connection Manager: transferring data over the network;
- Communication Monitor: redirecting queries from and to the I/O devices, e.g., sensors and keyboard.

### 3.1.1 System Requirements

The confidence of a user in a modular system is based on their confidence in the individual system components. Thus, an important aspect of the system is the necessity of a secure boot mechanism, where all modules are checked for authenticity and integrity. Only starting from an untampered kernel on an untampered central unit can the kernel check the other modules for authenticity. In the case of measuring instruments, where seals are used to detect hardware manipulation, the boot-loader should lie on a separate tamper-proof sealed storage unit that is not readable/writable for the other system components. The first check then starts at the boot process, where, for example, the hash value, e.g., the secure hash algorithm *SHA-2*, of the kernel binary is calculated and checked with the pre-calculated value stored in the storage unit of the boot-loader. If the two values are identical, the kernel can be loaded and starts with similar tests on the individual modules.

After a successful boot process, confidential conversation channels between the VMs must be established. The most important system requirement is a correct microkernel/VMM, which enforces isolation of the VMs and has no covert communication channels. It must be impossible to subvert the VMM or to attack other VMs through a compromised VM. The microkernel needs to assign unique unchangeable identifiers to the virtual network interface controllers of the VMs and must create buffers for every virtual connection the system needs. The buffers are not directly accessible by the VMs; they only simulate a network transmission.

There must be a mechanism to switch from legally relevant to non-legally relevant mode, if input devices are shared, e.g. a mouse. A hardware switch accessible by the VMs would be an example. If the switch is set to legal mode, every input from devices that can be used for non-legally relevant tasks, e.g., a keyboard, would be redirected to the non-legally relevant VM and to the legally relevant VM, the other way around. Another method

would be to use a touch screen that is divided into legally relevant parts and non-legally relevant ones.

The scheduler implemented must ensure fixed runtimes for every VM to ensure worst-case response times and to minimize the potential for denial-of-service attacks. A measuring instrument is a real-time system, meaning that, if within a maximum time frame, measuring tasks are not completed, failure has occurred. Therefore, absolute worst-case execution times (WCET) for the VMs must be guaranteed. An advantage of our system and of embedded systems in general is their static behavior. The amount of needed VMs remains constant over the whole execution time; therefore, a static schedule can be declared from the beginning. A temporal partition schedule should be applied[64], assuring that VMs do not starve, *i.e.*, do not get execution time. Each VM is provided a window of execution within the repeating timeline.

In our framework, some VMs, e.g., the Connection Manager that is responsible for redirecting interrupts, can be split into more than one window of execution. In this way, the system can react faster to input devices. Other VMs that need to execute their calculation as fast as possible, e.g., the L VM, could get a bigger window of execution than the rest of the VMs or could run with fewer VMs on a separate core in a multi-core system, as can be seen in Section 3.4.

### 3.1.2 Distributed System View

As already mentioned, we have built a virtual distributed system in which the VMs communicate through a virtual network. The microkernel/hypervisor ensures that the network is reliable; hence, for the transport layer protocol, we can use the User Datagram Protocol *UDP*. The virtual network is shown in Figure 3.3.

For security reasons, the network layer protocol used when a VM communicates with the Connection Manager should be encrypted, e.g., Internet Protocol Securiy *IPsec* in Transport Mode. To encrypt the packets, IPsec uses the mechanism Encapsulating Security Payload (ESP), encrypting the payload by the symmetric encryption algorithm like Advanced Encryption Standard in Cipher Block Chaining mode *AES-CBC* with the keys, that are being managed by the Key and Signature Manager. By encrypting the transmission with a key unknown to the Connection Manager, we make sure that if it is compromised, e.g., because it is accessible by the real network, it is not able to modify messages that are unnoticed. Communication between other VMs is not that critical and, therefore, does not need encryption.

For the application layer, a protocol must be defined that encapsulates the commands and data. Thereby, every VM could use its own protocol or

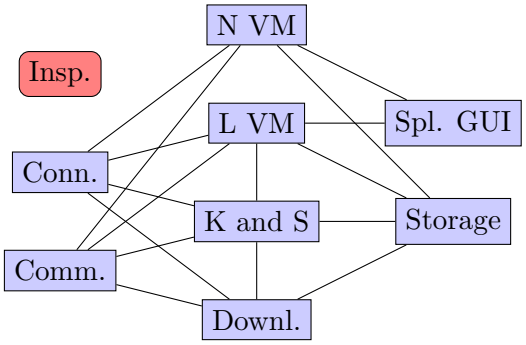



Figure 3.3: Connected VMs: Inspector is monitoring the VMs. In our demonstrator from Section 3.4 it is a normal VM that has a connection to the VMs it supervises (not shown in this figure).

interpretation of payload, e.g., the Storage Manager accepts requests, like storing data to disk or getting data from disk, which other VMs do not need. Hence, every VM needs to know the structure of the protocols, which other VMs use, if they want to communicate with each other.

Each VM has a server application that listens to a predefined port to receive and afterwards respond to the queries. In our architecture, the VMs fulfill client and server duties, because they redirect tasks to and process tasks for other VMs, which makes them so-called servants. The VMs can be divided into three core layers: the user interface layer, the processing layer and the data layer. Tasks for the user interface are executed by the Communication Manager that redirects I/O from and to devices, the Split GUI, which displays the graphical user interface (GUI), and the Connection Manager communicating to the outer world. The processing is done by the L/N VMs, the Inspector and the Download Manager. Finally the data are managed by the Key and Signature Manager and the Storage Manager. If the scope is to construct a real distributed system, the only modules needed on the measuring instrument are the user interface layer VMs and the Inspector, which does integrity checking on the other VMs (described in detail in Section 3.3); all of the other modules can be outsourced to other machines. When using an open network, reliability is not ensured. Therefore the cryptographic protocol Transport Layer Security *TLS* should be used as the transport layer protocol, which the Connection Manager should enforce for network communication.)

## 3.2 Description of the Modules

By dividing the framework into modules, it can be tailored individually for every measuring instrument. For example, if a measuring instrument does not need a download mechanism, the Download Manager should be removed, and if it does not need network access, the Connection Manager is unnecessary. Besides the L VM, the modules every measuring instrument needs are the Communication Monitor and the Inspector. A closer look at the individual modules is given below.

### 3.2.1 Key and Signature Manager

The Key and Signature Manager is responsible for assuring confidentiality and integrity by managing the public keys of the VMs, which want to communicate to the outside world over the Connection Manager. The Key and Signature Manager serves as a certification authority (CA), assigning and dispensing public keys to the corresponding VMs, which are, in turn, used to negotiate a symmetric key. The manager should have exclusive rights to a portion of the storage device to hold sensitive information. Every VM has its own key database holding the symmetric keys. If a public key gets changed, the manager informs the other VMs to renegotiate a symmetric key. If a key is compromised, the certificate can be invalidated, and the respective VM can be prompted to regenerate a key pair and to transmit the new public key to the manager. Only an authorized entity should be able to operate the Key and Signature Manager in such a way. Even a sealed hardware switch that needs to be broken to allow the reassigning of keys could be a possible solution.

Furthermore, the manager incurs the protection of legally relevant data by holding the keys for file system data encryption and the hash values of the Software IDs for integrity checks at boot and runtime.

### 3.2.2 Connection Manager

The Connection Manager is the only VM with physical network access. All data transmitted from and into the network goes through this module. Hence, the Connection Manager is critical from a security point of view. The manager is a firewall for the system, analyzing received data and, according to its rules, redirecting the packets to the appropriate VM. For this purpose, a well-defined protocol must be established. If a packet does not conform to the protocol or the firewall rules, it is discarded.

Another one of its duties is the encryption of legally relevant data in transit, by building up a VPN to only trusted end-points. Data coming

from other VMs is itself already encrypted by the symmetric keys that the individual VMs have pre-negotiated with the end-points and cannot be read out by the Connection Manager, it can only be redirected. In this way, a compromised Connection Manager is not able to modify data undetected. Non-legally relevant data can be sent unencrypted, but firewall rules for outgoing packets should be obeyed.

### 3.2.3 Inspector

A measuring instrument shall be designed to allow surveillance control by software after the instrument has been placed on the market and put into use. Hereby, software identification shall be easily provided by the measuring instrument. To achieve these requirements, the Inspector is the monitoring VM that can be advised by authorized entities to show the unique ID of the device. The Connection Manager (or Communication Monitor) redirects the requests after checking if an authorized person is connected to the device.

The Inspector itself automatically checks the system after a specified time interval and after failure has occurred. The Inspector module can advise the Storage Manager to check the file-system and the individual measurements for integrity, to transmit and check the identifications of all modules, to advise the Storage Manager to print out the logging, to check for enough storage capacity and, if needed, to check the other VMs for malware. The Inspector receives a snapshot of every VM from the hypervisor to do this; a detailed description is given in Section 3.3. For our purposes, we think that a detailed file-system check with our new data structure FLOUDS (described in Chapter 5) is sufficient, because creating snapshots and intercepting system calls as described in Section 3.3, can lead to the problem that time constraints cannot always be met. Nevertheless, we leave this open for the manufacturer to decide.

The Inspector can also be an autonomous watchdog timer that ensures correctness in the event of a delay that could harm the measurement. In extreme cases, it even deletes or marks measurements as invalid or shuts the system down. If a VM is not responding, the Inspector can restart it. For restarting VMs, shutting down the system and getting snapshots from the hypervisor, the Inspector needs special access rights that no other VM has. In our demonstrator (see Section 3.4) the Inspector has no extra rights and is connected to the other VMs by virtual TCP/IP connections.

### 3.2.4 Download Manager

In legal metrology, legally relevant software can only be updated if the software update was checked prior to download on the device and afterwards by breaking a seal. Downloads for non-legally relevant parts are allowed without new checking. In our system architecture, this is no problem, due to the strict isolation. Before legally relevant software is updated, the Download Manager checks if the sealed hardware switch for downloading legally relevant software is set. If this hardware switch is set, measuring must be disabled. For non-legally relevant updates, it only must be ensured that the computational time that the download mechanism needs does not disturb correct measuring. Therefore, the Download Manager should get a minimum running time, if a non-legally relevant software download is performed.

An upload can take place through different interfaces. If the download is started through the Ethernet interface, the Connection Manager receives the request and redirects the download to the partition of the Download Manager through the Storage Manager. If another interface is used, e.g., USB, the data goes through the Communication Monitor. After the download is finished, the respective module informs the Download Manager that checks the update on its partition for authenticity and integrity through hash values. If everything is correct, the Download Manager advises the Storage Manager to copy the update to the legally relevant or non-legally relevant partition, respectively. Afterwards, the Download Manager reports the download to the Storage Manager for the audit trail and advises the Inspector to restart the legal and/or non-legal VM, respectively.

### 3.2.5 Communication Monitor

The Communication Monitor supervises the queries from and to the I/O devices. This module ensures that user input and software cannot influence measurement data in an unwanted way, that peripheral devices can only be accessed by legally relevant software and that the transmission of data from the legally relevant VM to the non-legally relevant one is licit. If an input device is allowed to send data to the non-legally relevant VM, a switch must be set as described in Section 3.1.1.

This monitor serves as a kind of firewall for internal communication of the legally relevant VM, blocking packets that do not conform to well-defined rules and checking that confidential data is not transmitted to the non-legally relevant VM. The communication monitor is the only VM that has direct access to the peripheral devices besides the physical network card (accessible by the Connection Manager), the display (accessible by the Split GUI) and the storage device (accessible by the Storage Manager). Other

modules can just communicate with each other through their virtual network cards. An interrupt from an input device is first directed to the driver in the Communication Monitor, which, in turn, translates it to a network package and sends it to the legally relevant VM.

### 3.2.6 Split GUI

Non-legally relevant output must be unambiguously marked to distinguish it from the legal one. The Spit GUI supervises this output to the screen and is the only VM that can write directly to the screen. One possible way is to define selected parts of the screen to the legally relevant software that cannot be changed by the non-legally relevant VM. Another solution is to change the running mode via a hardware switch. Hence, when the switch is set, non-legally relevant software cannot write to the screen. In either case, the Split GUI module conducts the buffering for both VMs. For that purpose, the legally relevant and the non-legally relevant VMs could each have their own virtual video card driver that redirects requests to the Split GUI, which, in turn, visibly separates the output for the user. Another solution is to communicate the screen output through the virtual network cards, by using a well-defined protocol to winnow the screen output data from other message data. By using this solution, the Split GUI communicates the same way with the VMs as every other module, and no extra driver must be written, respecting the minimality principle. The drawback is of course that no additional measures to prevent DMA attacks, for example, are being taken. The security of the framework relies on the correct implementation of the virtual network driver which prevents the manipulation of the reserved communication buffers to other VMs. Still, we used the second approach for our demonstrator (see Section 3.4), as it does not restrict the manufacturer. The manufacturers themselves can still implement an additional GPU driver, which can be also used to allow hardware acceleration.

### 3.2.7 Storage Manager

The Storage Manager is the only VM with access to the storage device. Every other module that wants access to its storage partition must send the read and write commands through network packets. The Storage Manager assures strict isolation of the individual module's stored data in use. It checks the module's file permissions, making sure that no illicit manipulation of data takes place. It can also be responsible for the encryption of data-at-rest, making the file-system data unreadable if the key is unknown. In the case of errors, the Storage Manager informs the Logger and, in extreme cases, e.g.,

no storage capacity and nothing can be deleted, the Inspector to shut down the system.

Additionally, The Storage Manager is responsible for tracking interventions. The other modules inform the Storage Manager about interventions and errors by sending a network packet. This message is then concentrated to an aligned format and saved in a log-file for the audit trail. On demand, the Storage Manager sends its log-file to the L VM, which, in turn, can send it to the Split GUI, the Connection Manager or the Communication Manager to transmit it to other devices.

## 3.3 System Integrity Checking

In legal metrology, measurement devices need to be able to prove the integrity to a remote party, e.g., the control agent or the consumer, which is called remote attestation. The problem that arises with general purpose operating systems is that they are complex programs with millions of SLOC. Therefore, building high-assurance applications on these systems seems impossible, because they depend on the OS as part of their trusted computing base (TCB). In our system, we try to keep the TCB as small as possible by using a small hypervisor as the only underlying program that runs in privileged mode. This hypervisor enforces isolation: every VM needs to be securely isolated to provide confidentiality and integrity. Because the reliability of the applications still depends on the guest operating system, the VMs as a whole need to be integrity checked to make remote attestation possible. Furthermore, we need a trusted path, which ensures that the remote party knows that it communicates with an untampered VM and also makes sure that the VM notices, if it communicates with malicious software. Hence, the trusted path ensures the integrity and privacy of the communication.

Attestation authenticates what software was started at each layer of the software stack, from the firmware up to the VM. For this purpose, a certificate chain is needed. It should be built from tamper-resistant hardware, which stores the embedded cryptographic key signed by the vendor to the application. The tamper-resistant chip certifies the firmware, and the firmware certifies the boot loader. Afterwards, the boot loader certifies the VMM, which, in turn, certifies the VMs. The applications get their certificates by fist generating a private/public key pair and then by calling the VMM with a special call consigning the public key. The VMM generates and signs the certificate containing a hash of the attest-able component and its public key with application data. This certificate binds the public key to a component whose hash is given in the certificate. In this attestation scenario, integrity checking is only done at boot time, and no runtime protection is given.

Runtime integrity measurement requires more attention. First of all, the continuity of the integrity beyond the measurement time should be guaranteed. An attacker could, for example, change a program after it is measured and before it is executed. This problem is known as time of check to time of use (TOCTTOU) consistency. To overcome this problem, the applications need to be actively monitored by intercepting critical system calls and by guest memory protection. To intercept system calls, interrupts and exceptions, the VMM needs to be modified and enhanced by integrity measurement functions that calculate the hash of the effected memory area. In our architecture, this hash combined with its ID, e.g., file name and VM, can be transmitted to the Inspector VM. The Inspector keeps track of the memory layout of all of the processes running in the other VMs, and it stores the hash image of each of the pages that form a measured memory area. With this information, the Inspector can compare the correct hash values in the database with the ones that will be loaded into memory. If they do not match, one can assume that the process is not trustworthy. Until the Inspector has not finished its comparisons, the inspected VM is not allowed to execute or modify the memory area to guarantee TOCTTOU. This can be achieved by hardware support. For example, in HIMA (Hypervisor-based Integrity Measurement Agent)[5], this is done by the No-eXecute (NX) bit page protection flag, which is available on most hardware platforms. Executing an instruction from a page that is protected by the NX flag will cause an exception that is trapped in the hypervisor. After a page was measured and verified by the Inspector, the hypervisor marks this page as executable, but not writable.

## 3.4 Experimental Evaluation

To show the feasibility of our approach, we have started to build a system atop an L4-microkernel. In our opinion, the L4-microkernel family is a good starting, because it is widely used and consists of third generation microkernels. One of these microkernels, called seL4, is even fully verified[66], inferring that classical security threats against operating systems, like buffer overflows, null pointer de-referencing, arithmetic overflows, arithmetic exceptions, pointer errors and memory leaks, are not possible. A drawback is that only para-virtualization is supported like L4Linux for Fiasco.OC.

For our demonstrator, we used L4Linux running on top of the open source Fiasco.OC L4-microkernel, which yields good results even for real-time applications[68]. Our test board was the PandaBoard Revision B3 equipped with the OMAP4460 SoC running a dual-core 1.2 GHz ARM Cortex-A9 MPCore with 1 GiB of DDR2 SDRAM.

We compiled Fiasco.OC with multicore support and used the open source tool Buildroot to generate our eleven RAM disks for the VMs. First, we compiled the L4Linux with a virtual network driver and simulated cross-over cable connections between the VMs that need to communicate with each other, adding a virtual network interface controller *NIC* for every point-to-point connection to the VM. Hereby, 15 connections were established, which are shown in Figure 3.3 (Section 3.1.2). Afterwards, we included the Dropbear SSH server in the RAM disks, to remotely connect from one VM to another for test purposes (Allowing to remotely connect to one VM through another is a security risk and should not be permitted in a real system).

We start our VMs with a predefined memory and RAM disk size and assign the nine VMs to one of the two cores, with five VMs running on the first core and four on the second one, as also shown in Figure 3.3. The VMs are all scheduled with the round robin algorithm, where each VM has the same priority and a 21 microsecond time slice. The configurations are shown in Table 3.1.

Table 3.1: Configuration of the individual VMs.

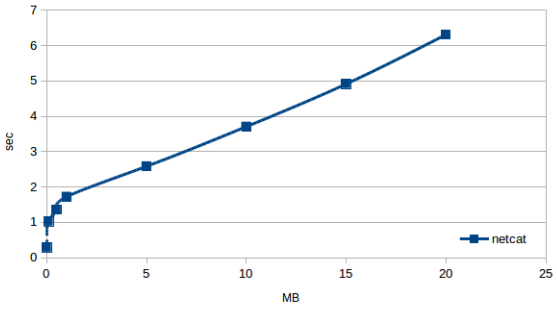
| **VM** | **Core** | **Memory** | **RAM Disk Size** |
|---|---|---|---|
| Key & Signature | 1 | 30 MB | 5 MB |
| Inspector | 1 | 30 MB | 5 MB |
| Storage. | 1 | 110 MB | 40 MB |
| Download. | 1 | 30 MB | 5 MB |
| Non-legal VM | 1 | 30 MB | 5 MB |
| Communication | 2 | 100 MB | 40 MB |
| Legal VM | 2 | 300 MB | 65 MB |
| Connection | 2 | 30 MB | 5 MB |
| Split GUI | 2 | 30 MB | 5 MB |
| $\Sigma$ | 2 | 690 MB | 175 MB |

We focused our test on the transmission of data between three VMs, the Communication Monitor, the legally relevant VM (L VM) and the Storage Manager. For the remaining VMs, we simulated a constant workload by an auto-start shell script, which traversed the file-system in an infinite while loop, adding files and afterwards deleting them. In this way, no VM became idle. In total, we tried to emulate a speed camera where the Communication Manager receives the pictures from the camera. In our demonstrator, we put seven different gray-scale pictures in PNG format on the RAM disk of the Communication Manager. These were transmitted to the L VM with the Netcat program using a TCP connection. As the sizes of the pictures were scattered and too small for meaningful tests (from 100 kB to 4.6 MB),

we additionally created dump files between 100 kB and 20 MB to get wider test results. In Figure 3.4, Table 3.4a lists their sizes and the measured transmission times of the file transfers between two VMs (every test was repeated 10 times; the times shown in the tables in this section are the averages of these 10 repetitions).

| File Size | Time |
|---|---|
| 10 kB | 0.289 s |
| 100 kB | 1.026 s |
| 500 kB | 1.358 s |
| 1 MB | 1.724 s |
| 5 MB | 2.585 s |
| 10 MB | 3.707 s |
| 15 MB | 4.913 s |
| 20 MB | 6.307 s |

(a) Table




(b) Graph

Figure 3.4: Measured times for TCP connections using Netcat.

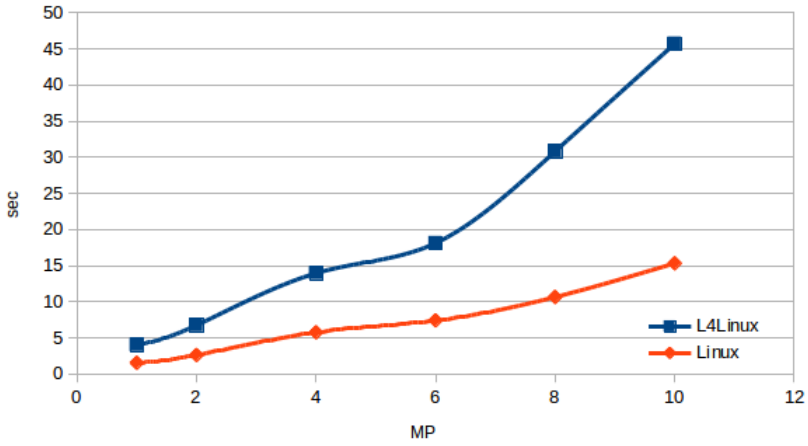
After the L VM has received the pictures, it converts the images with the program ImageMagick (convert command) by adding a label with a black background of 60 pixels in height on top of the pictures, which in national German law is postulated to show the velocity and other information corresponding to the picture. In another test scenario, the pictures are scaled to a quarter of their original resolution with auxiliary programs from the JPEG library (djpeg and cjpeg). The measured times and their graphs for these operations are shown in Figure 3.5. Additionally, we compiled a normal Linux kernel (Version 3.14, which corresponds to the version of L4Linux) for the PandaBoard, which loads the L VM RAM disk. Afterwards, the same convert commands were executed on the images to compare the times, also shown in Table 3.5a.

Lastly, the converted image is transmitted to the Storage Manager, where the data are encrypted by the OpenSSL program with AES 256-bit CBC. The encryption times and their graph are shown in Figure 3.6 with the corresponding times needed on the normal Linux kernel.

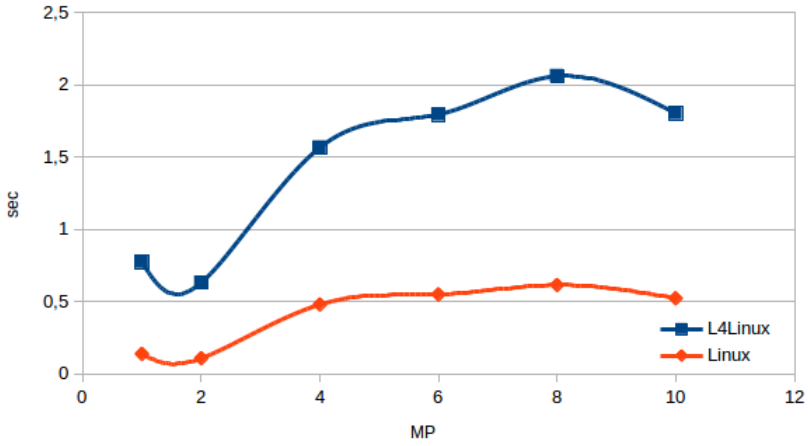
To combine all of the steps into one, we measured the time of the transmission and converting of a 10-MP image, which first was sent from the Communication Monitor to the L VM. Here, it was scaled down by a factor of sixteen and afterwards edited with ImageMagick and sent to the Storage Manager. Finally, the Storage Manager used OpenSSL to decode the pic-

| | L4Linux | | Linux | |
|---|---|---|---|---|
| **Resolution** | **Convert** | **djpeg** | **Convert** | **djpeg** |
| 1216 × 817 (1 MP) | 4.034 s | 0.774 s | 1.562 s | 0.138 s |
| 1920 × 1080 (2 MP) | 6.761 s | 0.631 s | 2.638 s | 0.108 s |
| 2580 × 1600 (4 MP) | 13.955 s | 1.567 s | 5.776 s | 0.480 s |
| 2832 × 2236 (6 MP) | 18.115 s | 1.795 s | 7.422 s | 0.550 s |
| 3264 × 2448 (8 MP) | 30.814 s | 2.061 s | 10.656 s | 0.615 s |
| 3835 × 2855 (10 MP) | 45.696 s | 1.804 s | 15.324 s | 0.523 s |

(a) Table




(b) Graph showing times using the convert functions

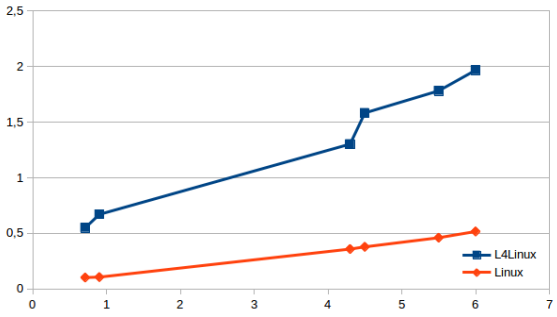



(c) Graph showing times using djpeg

Figure 3.5: Comparison of image processing times on normal Linux and L4Linux in our architecture.

| Size | L4Linux | Linux |
|---|---|---|
| 712 kB | 0.550 s | 0.103 s |
| 904 kB | 0.671 s | 0.107 s |
| 4.3 MB | 1.303 s | 0.359 s |
| 4.5 MB | 1.583 s | 0.379 s |
| 5.5 MB | 1.782 s | 0.461 s |
| 6 MB | 1.967 s | 0.517 s |

(a) Table




(b) Graph

Figure 3.6: Comparison of OpenSSL execution times.

ture. All of these steps were finished in around 12 s. Normal Linux achieved the same result in around 2 s, omitting the Netcat transfers, which were not necessary.

In our opinion, the slower processing time is justifiable considering the security gain. Furthermore, the test results seem to be promising, because it must be considered that eleven VMs run in parallel with a constant workload, whereas in the normal Linux architecture, just one operating system runs directly on the hardware, which leads to a relatively small speed gain of about 600%. It should be also noted that all of the tests were achieved by using Linux shell scripts (bash), which invoked standard Linux processes.

## 3.5 Analyzing the System

We analyze the system by showing how PHASE is followed and that this way, the policies that should be implemented according to MILS, to be specific, data isolation, information flow control, damage limitation and periods processing, are upheld. Afterwards, the basic framework is evaluated by time measurements.

### 3.5.1 Data Isolation

Data isolation is generally enforced by the component architecture principle and by using a small microkernel (even better would be a theoretically verified separation kernel) as our VMM. Hence, data isolation between the VMs can be presumed. As we also enforce the least privileged principle, our untrusted VMs running the variable software, *i.e.*, the L and N VMs

(see Section 3.1), have no direct access to the outside world. Their software must be checked in an independent expert validation, as done by the notified bodies, to ensure data isolation inside the VMs. Data-at-rest security is managed by the Storage Manager, which is responsible for the isolation and encryption of data on the storage device.

### 3.5.2 Information Flow Control

Every VM is provided with a virtual network interface for communication. The separation kernel ensures that no other communication is possible. Information from the outside world, *i.e.*, from peripheral devices and the network, go through the Communication Monitor and the Connection Manager, respectively. These VMs, in turn, communicate with the other VMs after carefully checking conformity to a well-defined protocol, as mentioned in Section 3.2.2, and checking access permissions enforcing the least privileged principle. Through unique network card numbers (MAC address) and predefined static connections, the communication partners are known. Additionally, sensitive data can be encrypted to protect the confidentiality and integrity of the communication.

### 3.5.3 Damage Limitation

The directly exposed VMs are the Communication Monitor and the Connection Manager, because they are the only VMs accessible through peripherals from which attacks could be mounted (NIC, USB, SD, *etc.*). These, in turn, have no direct access to the storage device and no access to measurement data, as the least privileged principle is applied. To prevent damage, the minimal implementation principle must be followed; hence, the modules, which are not just processes, but VMs with GPOSs, must have minimal configurations. A verified network stack and network interface card driver would drastically reduce the attack vectors.

As described in Section 2.3.4, virtualization adheres to the component architecture principle, limiting the damage to the respective VM in which it occurs. Additionally, independent expert validation by the notified bodies ensures damage limitation in the legally relevant VM.

Another measure taken is monitoring performed by the Inspector. If a VM does not conform to its predefined rules, the Inspector notices the misbehavior and takes action, e.g., reloading and restarting the module. In the worst case, the Inspector can shut down the system.

### 3.5.4 Periods Processing

The micro-/separation kernel must ensure that no hidden channels, through which information could leak out, are present. These could arise due to scheduling, because the VMs run consecutively, often using the same resources, e.g., shared caches.

Another way of getting secret information is to exploit variations for covert channels or side channel attacks. In general, side channel attacks benefit from variations, e.g., in timing, power consumption, electromagnetic emanation and temperature, to gain information of a cryptosystem. A high-awareness security system should be tested for side channel attacks and should take sophisticated attacks into account, e.g., cache-based side channel analysis[85]. A covert channel exploits the same variations as a side channel attack, but is used by malicious processes to exchange information. For example, a VM could try to reduce or extend its execution time, which then can be noticed by the ensuing VM and analyzed. Hereby, a reduced execution time could stand for a zero and an extended one for a one.

Most of the counter-measures are taken by the micro-/separation kernel or can be taken care of by special hardware. Covert channels that arise due to the scheduling policy must be considered separately. A partition schedule should be employed to eradicate timing variations in the scheduling process, because every VM has its fixed running window, which cannot be released. To ensure that a VM cannot delay the beginning of the execution time of the ensuing VM, e.g., by a system call before the end of its execution window, a time buffer is needed between the VMs.

## 3.6 Related Work

For measuring instruments under legal control, there are papers that point out the risk general purpose operating systems have on these devices, e.g.,[109]. Other documents talk more generally about Linux system configurations for general-purpose computers, e.g., the Guide to the Secure Configuration of Red Hat Enterprise Linux 6[49]. These documents try to secure the system within the system by restricting the user, e.g., restricted access rights. Vulnerabilities that lie in the operating system or in user programs can be used to subvert these settings, making these protections too weak.

There are some papers that stress that the microkernel approach for virtualization is recommended to construct secure systems[56,75,70,57,54,94]. Their approaches are similar; they try to create a secure system by modularizing the system architecture, but of course, lack the concrete analysis of legal metrology aspects, because they talk about general purpose systems. NOVA

(short for NOVA OS Virtualization Architecture)[106], for example, describes a microhypervisor architecture, which is written with only around 40K source lines of code, which is a magnitude smaller than popular virtualization architectures.

A modular system architectures for general purpose systems that is similar to ours is Terra[40]. It is based on virtualization by dividing critical parts of software from non-critical ones through virtual machines. Here, software-based attestation is realized by building a certification chain that begins at the hardware with its private key embedded in a tamper-resistant chip. Afterwards, the hardware certifies the firmware, which, in turn, certifies the system boot loader, which certifies the virtual machine monitor (VMM). The VMM at last certifies the virtual machines. The individual application can then be certified by constructing a private/public key pair and advising the VMM by a special call to sign their public key. Hence, Terra provides integrity checking only at boot time and no runtime protection. A better approach is given by[5], called HIMA (Hypervisor-based Integrity Measurement Agent), which can be also combined with Terra; a similar approach is described in Section 3.3. In HIMA, "out-of-the-box" measurement is done by active monitoring of critical guest events (e.g., creation of a process) and guest memory protection (by using hardware support, e.g., NX-bit).

In the measuring instruments domain, there are some papers that talk more specifically about smart meter devices and point out that software-based attestation is critical[74,27]. To our knowledge, most of the research done in software security for legal metrology concerns smart grids (e.g., also[15] and[26]). Here, threats, like network attacks, that exist for these devices are analyzed to point out challenges and solutions.

With our approach, a general and detailed solution for a secure system architecture is given. A bridge between academia and industry is created that makes sure that all of the requirements from European directives for measuring instruments are upheld by using secure and sound techniques. We tailor our architecture to fulfill legal requirements for measuring instruments in Europe and keep system integrity checking in mind.

## 3.7 Chapter Summary

In this chapter, we presented a framework for a new secure system architecture for measuring instruments in legal metrology. We constructed our architecture by analyzing the requirements for measuring instruments demanded in the MID and the WELMEC 7.2 Software Guide, combined with methodologies and concepts from high-assurance software systems, *i.e.*, MILS and

PHASE. To harness device drivers and network stacks of general purpose operating systems, we came to the conclusion that virtualization through a small microkernel is the right solution to combine security with usability.

We took a three-pronged approach. Firstly, we separated the legally relevant parts from the irrelevant ones by putting them in different virtual machines. Secondly, we made sure that their virtual machines have no direct access to I/O devices. Lastly, we constructed a secure framework, which provides services to these VMs. This framework, also consisting of separated VMs, monitors the information flow, correctly delegates requests from and to I/O devices and helps control agencies to verify system integrity.

Furthermore, the framework was evaluated by building a system atop an L4-microkernel. For our demonstrator (PandaBoard Revision B3 equipped with the OMAP4460 SoC running a dual-core 1.2 GHz ARM Cortex-A9 MPCore with 1 GiB of DDR2 SDRAM), we used L4Linux running atop the open source Fiasco.OC L4-microkernel, which yields good results, even for real-time applications. These tests show that it is practically feasible to construct a configurable framework using our architecture, which is applicable for powerful measuring instruments under legal control.

# 4

# Succinctly Storing Tree-Like Graphs

## 4.1 Tree-Like Graphs

We focus on the succinct representation of a very practical class of directed graphs: graphs that are "tree-like" in the sense that the number of edges, which can potentially be $\Theta(n^2)$ for an $n$-node graph, is much lower. We measure this tree-likeness by introducing two additional parameters:

1. $k$, the number of "additional" edges that have to be added to a spanning tree of the graph (note that $k = m - n + 1$ if $m$ denotes the total number of edges), and

2. $h \leq k$, the number of nodes having more than one incoming edge (also called non-tree nodes in the following).

Our focus are graphs where $k = O(n)$, with a small constant in the big-O. This definition of tree-likeness is similar in flavor to the $k$-almost trees by Gurevich et al.[50], but in the latter the number of additional edges is counted separately for each biconnected component, with $k$ being the maximum of these.

We think that our definition of tree-likeness encompasses a large range of instances arising in practice. One important example comes from computational biology, where one models the ancestral relationships between species by phylogenetic trees. However, sometimes there are also non-bifurcating specification events[59]. One approach to handle those events are phylogenetic networks, which have an underlying tree as a basis, but with added cross-edges to model the passing of genetic material that does not follow the tree. Other example of tree-like graphs are *compact directed acyclic word*

*graphs* (CDAWGS), a well-known text indexing data structure[14], and, of course, file-systems with links, which we will explain in detail in Chapter 5.

Our first contribution (Section 4.3) is a theoretical formulation of the GLOUDS, a succinct data structure for graphs with the above mentioned parameters $n$, $m$, $k$, and $h$. It uses space at most $(2n + m)\lg 3 + h \lg n + k \lg h + o(m + k \lg h) + O(\lg \lg n)$ bits, which is close to the $2n + o(n)$ bits for succinct trees if $k$ (and hence also $m$ and $h$) is close to $n$. This should be compared to the $O((n + m) \lg n)$ bits that were needed if the graph was represented using a pointer-based data structure.

Our second contribution is that we show that the data structure is amenable to a practical implementation (Section 4.4–4.5). We show that we can reduce the space from a conventional pointer-based representation by a factor of about 20, while the times for navigational operations (moving in either direction of the edges) increase by roughly the same factor; such a space-time trade-off is typical for succinct data structures.

### 4.1.1 Further Theoretical Work on Succinct Graphs

Farzan and Munro[35] showed how to represent a general graph succinctly in $\lg \binom{n^2}{m}(1 + o(1))$ bits of space, while supporting the operations supported both by adjacency lists and by adjacency matrices in optimal time. Other results exist for special types of graphs: separable graphs[12], planar graphs[82], pagenumber-$k$ graphs[41], graphs of limited arboricity[63], and DAGs[34]. However, to the best of our knowledge, only the approach on separable graphs has been implemented so far[13]. Also, none of the approaches can navigate efficiently to the sources of the *incoming* edges (without doubling the space), as we do. We also presented a similar structure to the one here in[95], which has its focus on phylogenetic networks (see Section 4.5.1).

For a good overview of the theoretical work on succinct graph representation, see the recent survey by Munro and Nicholson[79].

## 4.2 Preliminaries

In this section we introduce existing data structures that form the basis of our new succinct graph representation, which can also be found in Section 2.4.

As already stated in Section 2.4, all these results (hence also our new one) are in the word-RAM model of computation, where it is assumed that the machine consists of words of width $w$ bits that can be manipulated in $O(1)$ time by a standard set of arithmetic and logical operations, and further that the problem size $n$ is not larger than $O(2^w)$.

### 4.2.1 Rank and Select

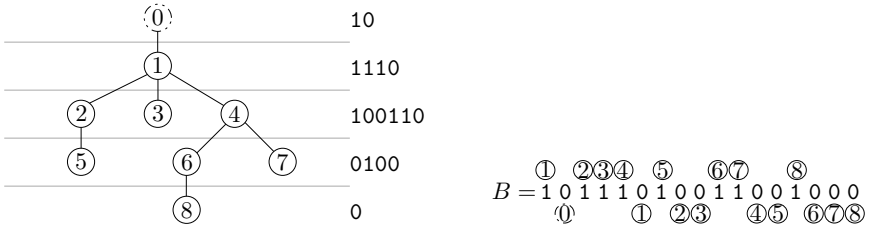
Let $S[0,n)$ be a *bit-string* of length $n$. We define the fundamental *rank*- and *select*-operations on $S$ as follows: $\mathsf{rank}_1(S,i)$ gives the number of 1's in the prefix $S[0,i]$ $(0 \leq i < n)$, and $\mathsf{select}_1(S,i)$ gives the position of the $i$'th 1 in $S$, reading $S$ from left to right $(1 \leq i \leq n)$. Operations $\mathsf{rank}_0(S,i)$ and $\mathsf{select}_0(S,i)$ are defined similarly for 0-bits. $S$ can be represented in $n + o(n)$ bits such that rank- and select-operations are supported in $O(1)$ time[60,78]. See also Section 2.4.1.

These operations have been extended to sequences over larger alphabets, at the cost of slight slowdowns in the running times[45]: let $S[0,n)$ be a *string* over an alphabet $\Sigma$ of size $\sigma$. Then $S$ can be represented in $n \lg \sigma(1 + o(1))$ bits of space such that the operations $\mathsf{rank}_a(S,i)$ and $S[i]$ (accessing the $i$'th element) take $O(\lg \lg \sigma)$ time, and $\mathsf{select}_a(S,i)$ takes $O(1)$ time (all for arbitrary $a \in \Sigma$ and arbitrary $0 \leq i < n$). Note that by additionally storing $S$ in plain form, the access-operation also takes $O(1)$ time, at the cost of doubling the space. In some special cases the running times for the three operations is faster. For example, when the alphabet size is small enough such that $\sigma = w^{O(1)}$ for word size $w$, then Belazzougui and Navarro[9] proved that $O(1)$ time for all three operations is possible within $O(n \lg \sigma)$ bits of space. For another practical example, see also the wavelet tree explained in Section 2.4.2 or Section 4.4.3

### 4.2.2 The Level Order Unary Degree Sequence (LOUDS)

There are several ways to represent an ordered tree on $n$ nodes using $2n$ bits[81,10]; in this thesis, we focus on one of the oldest approaches, the *level order unary degree sequence*[60], which is obtained as follows (the reasons for preferring LOUDS over BPS[81] or DFUDS[10] will become evident when introducing the new data structure in Section 4.3). For convenience, we first augment the tree with an artificial *super-root* that is connected with the original root of the tree. Now initialize $B$ as an empty bit-vector and traverse the nodes of the tree level by level (aka breadth-first). Whenever we see a node with $k$ children during this level-order traversal, we append the bits $1^k0$ to $B$, where $1^k$ denotes the juxtaposition of $k$ 1-bits. See Fig. 4.1 for an example. In the resulting LOUDS, each node is represented twice: once by a '1,' written when the node was seen as a *child* during the level-order traversal, and once by a '0,' written when it was seen as a *parent*. The number of bits in $B$ is $2n + 1$.

We identify the nodes with their level-order number, since both the 1- and the 0-bits appear in this order in $B$. It should be noted that all succinct data structures for trees[60,81,10,36,25] must have the freedom to fix a particular




(a) tree (b) LOUDS

Figure 4.1: An ordered tree (a) and its level order unary degree sequence (b).

naming for the nodes; natural such namings are post- or pre-order[60,81,10], in-order[25], and level-order[60], as here.

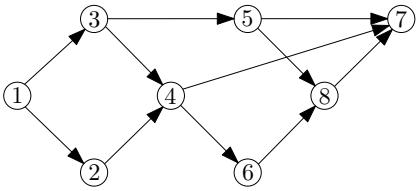
If we now augment $B$ with data structures for rank and select (see Section 4.2.1 or Section 2.4.1), then the resulting space is $2n + o(n)$ bits, but basic navigational operations on the tree can be *simulated* in $O(1)$ time: for moving to the parent node of $i$ ($1 \leq i \leq n$), we jump to the position $y$ of the $i$'th 1-bit in $B$ by $y = \mathsf{select}_1(B,i)$, and then count the number $j$ of 0's that appear before $y$ in $B$ by $j = \mathsf{rank}_0(B,y)$; $j$ is then the level-order number of the parent of $i$. Conversely, listing the children of $i$ works by jumping to the position $x$ of the $i$'th 0-bit in $B$ by $x = \mathsf{select}_0(B,i)$, and then iterating over the positions $x+1, x+2, \ldots$, as long as the corresponding bit is '1.' For each such position $x+k$ with $B[x+k] = 1$, the level-order numbers of $i$'s children are $\mathsf{rank}_1(B,x+k)$, which can be simplified to $x - i + k + 1$.

## 4.3 Graph Level Order Unary Degree Sequence

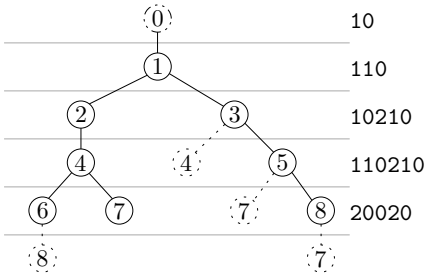
We now propose our new succinct data structure for tree-like graphs, which we called *graph level order unary degree sequence* (GLOUDS).

Let $G$ denote a directed graph. We use the following characteristics of $G$:

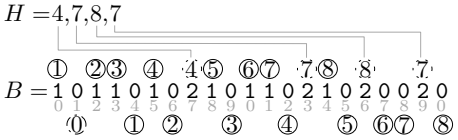
- $n$, the number of nodes in $G$,
- $m$, the number of edges in $G$,
- $c \leq n$, the number of *roots* in $G$, i.e., the size of a minimal set of nodes from which directed paths to all other nodes exist[67] p. 372,
- $k = m - n + 1$, the number of non-tree edges in $G$ (the number of edges to be added to a spanning tree of $G$ to obtain $G$), and



(a) A graph $G$.




(b) The resulting tree $T_G$. Shadow nodes are depicted as dashed lines.




(c) The resulting GLOUDS $B$ and the array $H$ of non-tree edges.

Figure 4.2: Illustration of our new data structure. The nodes are numbered such that they correspond to the level-order numbers in the chosen BFT-tree.

- $h \leq k$, the number of non-tree nodes in $G$ (nodes with more than 1 incoming edge).

For simplicity, assume for now that there exists a node $r$ in $G$ from which a path to every other node exists (i.e., $c = 1$). From $r$, perform a breadth-first traversal (BFT) of $G$. Let $T_G^{\text{BFT}}$ denote the resulting BFT-tree. We augment $T_G^{\text{BFT}}$ as follows: for each node $w$ that is *inspected but not visited* during the BFT at node $v$ (meaning that it has already been visited at an earlier point), we make a *copy* of $w$ and append it as a child of $v$ in the BFT-tree $T_G^{\text{BFT}}$. We call those nodes *shadow nodes*. Finally, we add a super-root to $r$, and call the resulting tree $T_G$, which has exactly $m + 2$ nodes. See Fig. 4.2a and 4.2b for an example of $G$ and $T_G$.

If no such node $r$ exists, we perform the BFT from $c$ roots $r_1, \ldots, r_c$, and obtain a BFT-*forest*. All roots of this forest will be made children of the super-root. This adds at most $c$ additional edges to $T_G$.

We now aim at representing the tree $T_G$ space efficiently, similar to the LOUDS of Section 4.2.2. Since we need to distinguish between real nodes and shadow nodes, we cannot construct a *bit*-vector anymore. Instead, we construct $B$ as a sequence of *trits*, namely values from $\{0,1,2\}$, as follows: again, $B$ is initially empty, and we visit the nodes of $T_G$ in level-order.

**Function** children($i$): find the nodes directly reachable from $i$.

```
x ← select_0(B, i) + 1;        // start of the list of i's children
while B[x] ≠ 0 do
    if B[x] = 1 then output rank_1(B, x);          // actual node
    else output H[rank_2(B, x) − 1];               // shadow node
    x ← x + 1;
end while
```

For each visited node, the sequence appended to $B$ is constructed as in the original LOUDS, but now using a '2' instead of a '1' for shadow nodes. The shadow nodes are *not* visited again during the level-order traversal and hence *not* represented by 0's.* We call the resulting trit-vector $B$ the *GLOUDS*. It consists of $n+m+c+1$ trits. See Fig. 4.2c for an example.

We also need an additional array $H[0,k)$ that lists the non-tree nodes in the order in which they appear in $B$. This array will be used for the navigational operations, as shown in Section 4.3.1. For the operations, besides accessing $H$, we will also need select-support on $H$. For this, we use the data structures mentioned in Sect. 4.2.1 [9,45].

### 4.3.1 Algorithms

The algorithms for listing the children and parents of a node are shown in Functions children($i$) and parents($i$). These functions follow the original LOUDS-functions as closely as possible. Listing the children just needs to make the distinction if there is a '1' or a '2' in the GLOUDS $B$; in the latter case, array $H$ storing the shadow nodes needs to be accessed.

Listing the parents is only slightly more involved. First, the (only) tree parent can be obtained as in the original LOUDS. Then we iterate through the occurrences of $i$ in $H$ in a while-loop, using select-queries. For each occurrence found, we go to the corresponding '2' in $B$ and count the number of '0's before that '2' as usual.

As in the original LOUDS, counting the number of children is faster than traversing them: simply calculate $\mathsf{select}_0(B,i+1) - \mathsf{select}_0(B,i) - 1$; this computes the desired result in $O(1)$ time.†

*Listing the shadow nodes by 0's would not harm, but does not yield any extra information; hence we can omit them.

†For calculating the number of parents in $O(1)$ time, we would need to store those numbers explicitly for hybrid nodes; for all other nodes it is 1.

**Function** parents($i$): find the nodes from which $i$ is directly reachable.

output $\mathsf{rank}_0(B, \mathsf{select}_1(B, i))$; `// tree parent`
$j \leftarrow 1$;
$x \leftarrow \mathsf{select}_i(H, j)$;
**while** $x < k$ **do**
  output $\mathsf{rank}_0(B, \mathsf{select}_2(B, x+1))$; `// non-tree parent`
  $j \leftarrow j + 1$;
  $x \leftarrow \mathsf{select}_i(H, j)$;
**end while**

### 4.3.2 Space Analysis

The trit-vector $B$ can be stored in $(n + m + c)(\lg 3 + o(1))$ bits[93], while supporting $O(1)$ access on its elements. Support for rank and select-queries needs additional $o(n + m)$ bits[60,78].

There are several ways to store $H$. Storing it in plain form uses $k \lg n$ bits. Using another $k \lg n(1 + o(1))$ bits, we can also support $\mathsf{select}_a(H,i)$-queries on $H$ in constant time[45]. This sums up to $2k \lg n + o(k \lg n)$ bits.

On the other hand, since the number $h$ of non-tree *nodes* can be much smaller than $k$ (the number of non-tree *edges*), this can be improved with a little bit of more work: we store a translation table $T[0,h)$ such that $T[i]$ is the level order number of the $i$'th non-tree node. Then $H[0,k)$ can be implemented by a table $H'[0,k)$ that only stores values from $[0,h)$, such that $H[i] = T[H'[i]]$. The combined space for $T$ and $H'$ is $k \lg h + h \lg n$ bits. To also support select-queries on $H$ within less than $k \lg n$ bits of space, we can use the *indexable dictionaries* of Raman et al.[101]: store a bit vector $C[0,n)$ such that $C[i] = \mathtt{1}$ iff the $i$'th node in level order is a non-tree node. $C$ can be stored in $h \lg n + o(h) + O(\lg \lg n)$ bits[101] Thm. 3.1, while supporting select- and partial rank-queries (only $\mathsf{rank}_1(C,i)$ with $C[i] = \mathtt{1}$, which is what we need here) in constant time. Now we only need to prepare $H'$ for select-queries, this time using $k \lg h + o(k \lg h)$ bits. Queries $\mathsf{select}_a(H,i)$ can be answered by $\mathsf{select}_{\mathsf{rank}_1(C,a)}(H', i)$, so $H$ can be discarded. Since the data structure of Raman et al.[101] automatically supports select-queries, we also do not need to store $T$ in plain form anymore, since $T[i] = \mathsf{select}_1(C, i)$. Thus, the total space for $H$ using this second approach is $h \lg n + k \lg h + o(h + k \lg h) + O(\lg \lg n)$ bits.

Summing up and simplifying ($c \leq n$), the main theoretical result of this chapter can be formulated as follows:

**Theorem 1.** *A directed graph $G$ with $n$ nodes, $m$ edges, and $h$ non-tree*

*nodes ($k = m - n + 1$ is the number of non-tree edges) can be represented in*

$$(2n + m) \lg 3 + h \lg n + k \lg h + o(m + k \lg h) + O(\lg \lg n)$$

*bits such that listing the $x$ incoming or $y$ outgoing edges of any node can be done in $O(x)$ or $O(y)$ time, respectively. Counting the number of outgoing edges can be done in $O(1)$ time.*

## 4.4 Implementation Details

We now give some details of our implementation of the data structure from Section 4.3, sometimes sacrificing theoretical worst-case guarantees for better results in practice.

### 4.4.1 Representing Trit-Vectors

We first explain how we store the trit sequence $B$ such that constant time access, rank and select are supported. We group 5 trits together into one tryte, and store this tryte in a single byte. This results in space $\lceil (n + m + c)/5 \rceil \cdot 8 = \lceil 1.6\,(n + m + c) \rceil$ bits for $B$, which is only $\approx 1\%$ more than the optimal $\lceil (n + m + c) \lg 3 \rceil \approx \lceil 1.585\,(n + m + c) \rceil$ bits. The individual trits are reconstructed using Horner's method, in just one calculation.[‡]

For rank and select on $B$, we use an approach similar to the *bit*-vectors of González et al.[46], but with a three-level scheme (instead of only 2), thus favoring space over time. This scheme basically stores rank-samples at increasing sample rates, and the fact that the bits are now intermingled with 2's does not cause any troubles. We used sample rates 25, 275, and 65 725 trits, respectively, which enable a fast byte-aligned layout in memory. On the smallest level we divided a 25-trit block into five trytes. Using the table lookup technique[78] on the trytes the calculation for rank on a 25-trit block is done in at most five steps with an overhead of $3^5 = 243$ bytes of space.

As in the original publication[46], select queries are solved by binary searches on rank-samples, again favoring space over time.

### 4.4.2 Representing $H$ as an Array

Instead of the complex representation of $H$ as described in Section 4.3.2, needed for an efficient support of the parent-operation, we used a simpler

[‡]We did not theoretically investigate codes that exploit the fact that the distribution of the 0's, 1's, and 2's in $B$ is not necessarily uniform. However, in Section 4.5.2 we present a practical way achieving exactly this (basically a two-level wavelet tree[47] consisting of two 0-order compressed bit-vectors[101]).

array-based approach: we mark the non-tree nodes in a bit-vector $P[0,n)$. (In the example of Fig. 4.2, we have $P = \texttt{00010011}$ for the non-tree nodes 4,7, and 8.) A second array $Q[0,k)$ lists the positions of the other occurrences of the non-tree nodes, in level order (in the example, $Q = [7; 13,19; 16]$). A final third array $N[0,h)$ stores the starting positions of the non-tree nodes in $Q$ (in the example, $N = [0,1,3]$). Then with $P$ we can find out if a node $i$ has further shadow copies, and if so, list them using $Q$ and $N$. Note that with these arrays, we can also efficiently list (in $O(1)$ time) the number of parents of non-tree nodes.

### 4.4.3 Using a Wavelet Tree for $H$

A different practical approach for representing $H[0,k)$ is the *wavelet tree*[47]. A wavelet tree on $H$ is a binary tree with $h$ leaves and is recursively constructed as follows: if the sequence consists of at least 2 different characters ($h \geq 2$), we split the alphabet into two halves: the first half consists of all characters $\leq h/2$, and the second of those characters $> h/2$. Then we construct a bit-vector $V[0,k)$ such that $V[i] = 0$ iff $H[i]$ belongs to the first half of the alphabet; this bit-vector $V$ is stored at the root and partitions $H$ into two subsequences, $H_\ell$ and $H_r$. The left and right children of the root are then the (recursively constructed) wavelet trees for $H_\ell$ and $H_r$. By adding rank- and select-support on all bit-vectors of the tree, the wavelet tree supports $\mathsf{select}_c(H,i)$ and $\mathsf{rank}_c(H,i)$ queries in $O(\lg h)$ time for every character $c$ in $S$, while using $(k \lg h)(1 + o(1))$ bits. [§] See also Section 2.4.2 for a more detailed description.

## 4.5 Practical Results

We conducted three tests. A first test (Section 4.5.1) compares a basic implementation of our GLOUDS to a conventional adjacency-list based graph representation, using the example of phylogenetic networks. The second test (Section 4.5.2) on general tree-like graphs uses an even more space-conscious implementation of the GLOUDS, and (for fairness of comparison) also makes some space-optimization on the pointer-based representation. A final test

[§]Note that having rank-support on $H$ is useful for an additional query on directed graphs, namely that of checking the presence of a (directed) edge: The edge $(i,j)$ is present in $G$ if and only if $j$ "appears" between positions $a := \mathsf{rank}_0(B,i)+1$ and $b := \mathsf{rank}_0(B,i+1)$ in $B$. This appearance can be either as a tree edge (which is easily checked with $\mathsf{rank}_1$-queries on $B$), or as a non-tree edge. The latter can be resolved by checking if or not $\mathsf{rank}_j(H, \mathsf{rank}_2(B,a)) \stackrel{?}{=} \mathsf{rank}_j(H, \mathsf{rank}_2(B,b))$. The total time for this operation is dominated by the time for rank-queries on $H$, which is $O(\log h)$ when using a wavelet tree.

compares our GLOUDS with a data structure for web graphs by performing breadth-first-traversals on real-world graphs.

Our machine was equipped with an Intel Core i7@2.2GHz and 8GB of RAM, running under Windows 7. We compiled the program of Section 4.5.1 for 32 bits, in order not to make the pointer-based representation unnecessarily large. All programs used only a single core of the CPU.

### 4.5.1 Graphs Representing Phylogenetic Networks

The aim of this section is to show the practicality of our approach on the example of phylogenetic networks. Such networks arise in computational biology. They are a generalization of the better known phylogenetic trees, which model the (hypothetic) ancestral relationships between species. In particular for fast reproducing organisms like bacteria, networks can better explain the observed data than trees. Quoting Huson and Scornavacca[59], phylogenetic networks "may be more suitable for data sets where evolution involves significant amounts of reticulate events, such as hybridization, horizontal gene transfer, or recombination."

Since large real-life networks are not (yet) available, we chose to create them artificially for our tests. We did so by creating random tree-like graphs with 10% non-tree edges ($k = n/10$), by *directly* creating random trit-vectors of a given length, and randomly introducing $k$ 2's to create non-tree edges. We further ensured that shadow nodes have different parents, and that all non-tree edges point only to nodes at the same height (in the BFS-tree), mirroring the structure of phylogentic networks (no interchange of genetic material with extinct species).

For our first test, we constructed the GLOUDS as described in Section 4.4, and compared it to a conventional pointer-based data structure for graphs (where each node stores a list of its descendants, a pointer to an arbitrary father, and the number of its descendants). We also added a bit-vector $D = [0,n)$ with $D[i] = 1$ iff node $i$ is a leaf node. This way, the question if a node has children can be quickly answered by just one look-up to $D$, omitting rank and select queries.

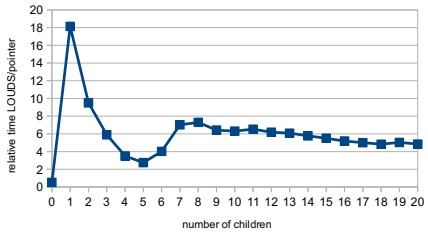
While there exist many implementations of succinct data structures for trees[¶], we are not aware of any implementations for graphs, hence we did not compare our data structure to others.

Table 4.1 shows the sizes of the data structures and the average running

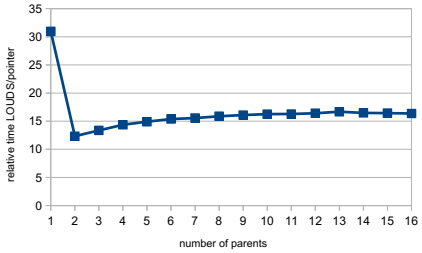
[¶]For example, the well-known libraries for succinct data structures https://github.com/fclaude/libcds and https://github.com/simongog/sdsl both have well-tuned succinct tree implementations. Other sources are[3,42].

Table 4.1: Comparison between a pointer based graph and our succinct GLOUDS representation with 10% non-tree edges.

| | space [MByte] | | time for children [$\mu$sec] | | time for parents [$\mu$sec] | |
|---|---|---|---|---|---|---|
| $n$ | LOUDS | pointer | LOUDS | pointer | LOUDS | pointer |
| 10 000 | 0.0159 | 0.3654 | 0.3203 | 0.0295 | 0.3315 | 0.0129 |
| 100 000 | 0.1682 | 3.6533 | 0.3458 | 0.0311 | 0.3472 | 0.0130 |
| 1 000 000 | 1.6818 | 36.5433 | 0.3884 | 0.0332 | 0.3614 | 0.0136 |
| 10 000 000 | 18.8141 | 365.4453 | 0.3889 | 0.0337 | 0.3812 | 0.0138 |
| 100 000 000 | 188.1542 | 3 654.4394 | 0.4095 | — | 0.4198 | — |




(a) Listing the children of a node. The graph shows the relative slow-down of our LOUDS over a pointer-based representation for nodes with varying number of children.




(b) The same as in (a), but now for listing the parents of a node.

Figure 4.3: Detailed evaluation of running times.

times for the children- and parents-operations with either representation.[‖] We averaged the running times over 1 000 tests for $n = 10\,000$, over 100 tests for $100\,000 \leq n \leq 1\,000\,000$, over 15 tests for $n = 10\,000\,000$, and over 5 tests for $n = 100\,000\,000$. It can be seen that our data structure is consistently about 20–25 times smaller than the pointer-based structure, while the time for the operations increases by a factor of about 12 in case of the children-operation, and by a factor of about 25 in case of the parents-operation. Such trade-offs are typical in the world of succinct data structures.

To further evaluate our data structure, we more closely surveyed the children- and parents-operations in a graph with 1 000 000 nodes and 10% of non-tree edges, in which a node has no more than 16 incoming edges. We

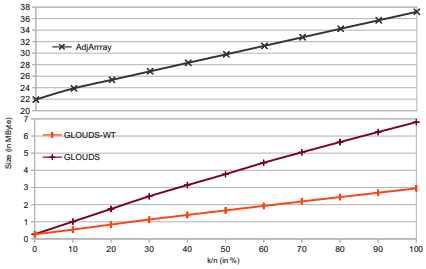
[‖]For memory reasons, the running times of the pointer-based representation could not be measured for the last 3 instances.

executed both operations on every node in the graph and grouped the running times by the number of children and parents, respectively. The results are shown in Fig. 4.3. In (a), showing the results for the children-operations, several interesting points can be observed. First, for nodes with 0 children (a.k.a. leaves), our data structure is actually *faster* than the pointer-based representation (about twice as fast), because this operation can be answered by simply checking one bit in the bit-vector $D$. Second, for nodes with 5 children the slowdown is only about 3, then rises to a slowdown of about 7 for nodes with 8 children, and finally gradually levels off and seems to convert to a slowdown of about 5. We think that this can be explained by the different distributions of the *types* of the nodes listed in the children operation: while for tree-nodes the node numbers can be simply calculated from the LOUDS, for non-tree nodes this process involves further look-ups, e.g. to the $H$-array. Since we tested graphs with 10% non-tree edges, we think that at about 7–8 children/nodes this effect is most expressed. In (b) the parents operation on our LOUDS for nodes with one parent is around 30 times slower than the pointer representation. For a greater number of parents it is about 16 times slower. Our explanation is that at first a rank and select query is necessary to retrieve the first parent node, afterwards if the node has more than one parent the $H$-array is scanned. With our practical implementation of the $H$-array from Section 4.4.2 the select results are directly saved in the $Q$-array, hence there is no need for select queries anymore and a rank query seems to be around 16 times slower than a look-up.

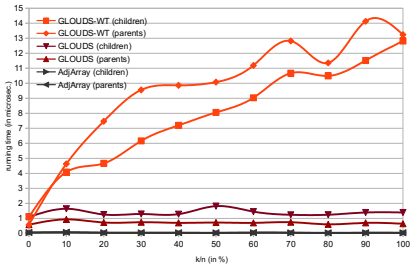
### 4.5.2 General Tree-Like Graphs

For our second test we used the succinct data structure library *sdsl* (`https://github.com/simongog/sdsl`) to represent all data-structures, and created general graphs with no restrictions on the non-tree nodes (also allowing loops). We implemented and compared the performance of three data structures:

**GLOUDS** A slightly improved version of the implementation presented in Section 4.5.1. The trit-vector (as described in Section 4.4.1 is replaced by two bit-vectors: one of length $2n + k - 1$, where the `1`'s represent the trits `1` and `2`, and a second bit-vector of size $n + k$ to distinguish the `1`'s from the `2`'s (this is basically a two-level wavelet tree on the trit vector). Both bit-vectors are compressed with the technique by Raman et al. (RRR method)[101]. This representation turned out to be smaller than the one from Section 4.5.1, at no observable costs in running times for the operations.




(a) Sizes of the three representations.




(b) Running times to retrieve one child/parent.

Figure 4.4: Detailed evaluation of the three representations.

**GLOUDS-WT** The same as GLOUDS, but now $H$ is stored as a wavelet tree (as described in Section 4.4.3 and Section 2.4.2) and the bit-strings in the WT are also compressed using the RRR method.

**AdjArray** An adjacency-array based representation for static graphs[77], tuned for space efficiency by using small pointers of size $\lceil \lg m \rceil$ bits using sdsl.

In the initial test phase we also evaluated a data-structure consisting of two RRR-compressed adjacency matrices[101], one for the children and one for the parents. This data structure was extremely big for tree-like graphs, even compared to the pointer-based representation (at least 70 times bigger) and also very slow (around 200 times slower than the GLOUDS representation), so it was not considered in further tests.

Figure 4.4 shows the sizes and running times of all three implementations, with $n = 1{,}000{,}000$ nodes and values for $k$ varying between 0 and $n$. It can be observed (a) that GLOUDS-WT is even smaller than GLOUDS, in particular for larger values of $k$. However, this comes at another increase in running time for the operations (b), by roughly one order of magnitude when compared with GLOUDS. (The AdjArray was again 10–20 times faster than GLOUDS.) Interestingly, the running time for GLOUDS-WT rises with increasing value of $k$, while those for GLOUDS and AdjArray stay more or less constant. This can naturally be explained by the increasing height of the wavelet tree for larger values of $k$ (and hence also larger $h$).

### 4.5.3 Breadth First Traversals on Real-World Graphs

The aim of this section is to show the practicality of our data structure by using it on real-world graphs, and applying a natural procedure on the resulting representation, namely a breadth-first-traversal (BFT). We compared the GLOUDS (The GLOUDS-WT from Section 4.5.2) with a framework called WebGraph, which was especially designed to compress web graphs[24,18,17,16]. In these graphs the nodes represent web-pages, and the directed edges are the hyper-links. Several properties of web graphs have been identified and exploited to achieve compression:

- **Locality of reference**: Most of the links from a site point within the site. By lexicographical URL ordering, the outgoing links point to nodes whose position is close to that of the current node. Gap encoding techniques can then be used to encode the differences.
- **Similarity of adjacency lists**: Many outgoing links are shared from nodes close in URL lexicographical order. Hence, compression can be achieved by using references to the similar list, plus a list of edits. Thereby, long intervals of consecutive numbers are formed, which again can be easily compressed.
- **Skewed distribution**: The distribution of the in-degrees and out-degrees of a node is bound to a power law.

In[24], RePair[71] was used to get further compression. RePair is a phrase-based compressor that permits fast and local decompression. It consists of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol, until no more replacements are possible. Our GLOUDS representation was not changed at all to take advantage of these facts.

We used the following real world graphs from `http://law.di.unimi.it` for our comparison (see also Table 4.2 for some characteristics of these graphs):

**amazon-2008** A symmetric graph describing similarity among books as reported by the Amazon store.

**cnr-2000** A small crawl of the Italian CNR domain.

**in-2004** A small crawl of the .in domain performed for the Nagaoka University of Technology.

**uk-2002** This graph has been obtained from a 2002 crawl of the .uk domain performed by UbiCrawler.

**enwiki-2013** This graph represents a snapshot of the English part of Wikipedia as of late February 2013. The pages are the nodes, and links between pages are the edges of the graph.

We also included a couple of *undirected graphs* to our test:

**dblp-2010** DBLP is a bibliography service from which an undirected scientific collaboration network can be extracted: each vertex represents a scientist and two vertices are connected if they have worked together on an article. The DBLP database is from the year 2010.

**dblp-2011** The DBLP database from the year 2011.

**hollywood-2011** One of the most popular undirected social graphs: the graph of movie actors. Vertices are actors, and two actors are joined by an edge whenever they appeared in a movie together.

In cases where the graphs consist of more than one root, we used Tarjan's algorithm[107] to identify the strongly connected components, from which the roots can be easily inferred.

The advantage of the GLOUDS representation is that one can also efficiently navigate to the incoming edges, which the WebGraph framework cannot. Here, also the transposed graph needs to be saved. Additionally, our representation allows random access to every node, whereas the WebGraph framework in its most compressed form has only sequential access. Hence, we compared our structure to the bigger web-graphs with random access, adding the offsets and the size of the transposed graph.

Table 4.3 compares the GLOUDS with the WebGraph representation. As mentioned before, we added the sizes of the transposed graphs to the whole size in the WebGraph representation. This was not done for the graphs *dblp-2010*, *dblp-2011* and *hollywood-2011*, as these are undirected graphs. Still, the GLOUDS representation only needs to store half of the edges, because it can also traverse the incoming edges.

As the GLOUDS is already in level order, we can efficiently do a BFT by just traversing the tree edges starting from the super-root of our GLOUDS, because every non-tree edge points to a node that was already found. Still, if the traversal needs to be a BFT starting from an arbitrary node, the non-tree edges must be checked directly, by looking them up in the wavelet-tree which takes additional time. Depending on the number of non-tree edges

Table 4.2: Number of nodes, edges, components and non-tree nodes in the graphs.

| | Graph | nodes | edges | components | non-tree nodes |
|---|---|---|---|---|---|
| directed | amazon-2008 | 735 323 | 5 158 388 | 1 | 4 423 066 |
| | cnr-2000 | 325 557 | 3 216 152 | 1 | 2 890 596 |
| | in-2004 | 1 382 908 | 16 917 053 | 211 | 14 420 349 |
| | uk-2002 | 18 520 486 | 298 113 762 | 138 916 | 279 606 387 |
| | enwiki-2013 | 4 206 785 | 101 355 853 | 367 984 | 97 089 663 |
| undir. | dblp-2010 | 326 186 | 1 615 400 | 74 724 | 515 966 |
| | dblp-2011 | 986 324 | 6 707 236 | 174 519 | 2 463 041 |
| | hollywood-2011 | 2 180 759 | 228 985 632 | 267 394 | 111 787 890 |

Table 4.3: Comparison between the WebGraph representation and the GLOUDS.

| | | space [MByte] | | time for BFT [sec] | | BFT per node [$\mu$sec] | |
|---|---|---|---|---|---|---|---|
| | Graph | GLOUDS | WebGraph | GLOUDS | WebGraph | GLOUDS | WebGraph |
| directed | amazon-2008 | 8.551 | 13.654 | 0.95 | 0.86 | 1.29 | 1.17 |
| | cnr-2000 | 3.069 | 2.765 | 0.36 | 0.63 | 1.12 | 1.95 |
| | in-2004 | 15.813 | 11.231 | 1.60 | 1.59 | 1.16 | 1.15 |
| | uk-2002 | 324.108 | 187.494 | 32.04 | 22.96 | 1.73 | 1.24 |
| | enwiki-2013 | 208.896 | 322.257 | 5.58 | 12.57 | 1.32 | 2.99 |
| undir. | dblp-2010 | 0.992 | 1.752 | 0.36 | 0.54 | 1.11 | 1.68 |
| | dblp-2011 | 5.348 | 8.501 | 1.18 | 1.26 | 1.20 | 1.28 |
| | hollywood-2011 | 202.991 | 18.532 | 2.59 | 14.37 | 1.20 | 6.59 |

$h$, we observed that the BFT can be slowed down by a factor of 15. For our tests we were interested in the minimum time a complete traversal took. Therefore, we started the BFT at the root node of the GLOUDS.

As the WebGraph framework was initially developed for Java, we conducted our BFT tests with the original Java implementation. Afterwards, we exported the graphs into a readable binary format for our C++ implementation, which then constructed the GLOUDS and started the BFT. We conducted the test, with the Java Framework and our C++ implementation, on the machine described at the beginning of this section.

It can be observed that graphs with few edges yield good results with the GLOUDS representation, in some cases they are even smaller: e.g., *dblp-2010* compresses to nearly half of the WebGraph representation. Nevertheless, graphs with many edges like *hollywood-2011* are more than 10 times bigger. Table 4.4 shows the sizes in proportion to each other, also looking

Table 4.4: Size comparison of WebGraph representation and GLOUDS.

| | Graph | $\frac{m}{n}$ | $\frac{\text{sizeGLOUDS}}{\text{sizeWebGraph}}$ | sizeTrit | sizeWT |
|---|---|---|---|---|---|
| directed | amazon-2008 | 7.015 | 0.626 | 0.802 | 7.749 |
| | cnr-2000 | 9.879 | 1.109 | 0.287 | 2.782 |
| | in-2004 | 12.232 | 1.408 | 1.302 | 14.511 |
| | uk-2002 | 16.096 | 1.728 | 23.022 | 301.086 |
| | enwiki-2013 | 24.093 | 0.648 | 7.006 | 201.890 |
| umdir. | dblp-2010 | 4.952 | 0.566 | 0.171 | 0.821 |
| | dblp-2011 | 6.800 | 0.628 | 0.639 | 4.699 |
| | hollywood-2011 | 105.003 | 10.95 | 5.057 | 197.934 |

at the quotient $\frac{m}{n}$. Additionally, it shows the sizes of the parts of which the GLOUDS consists of: the wavelet tree (of size sizeWT) for the non-tree edges, and the trit-vector (of size sizeTrit), which is actually made out of a second wavelet-tree compressing two bit-vectors, as mentioned at the beginning of Section 4.5.2.

To conclude this section, we think that our GLOUDS also compresses web-graphs really well, if $\frac{m}{n} < 10$, where $m$ is the number of directed edges. For undirected graphs $\frac{m}{n} < 30$ seems to be a good estimation, where $m$ still represents the number of directed edges, which are twice as many as the original undirected edges, one for every direction. For the graphs that are called "social graphs" at `http://law.di.unimi.it`, like *enwiki-2013*, the compression rate of the WebGraph framework is not that high, which makes the GLOUDS compression better even for $\frac{m}{n} \approx 24$. But as mentioned before, the BFT times rise with the number of non-tree edges, if the BFT is started from an arbitrary node. Hence, we still think that $\frac{m}{n} < 10$ is a good estimation for using the GLOUDS over the WebGraph framework. If in-degree node access is not needed $\frac{m}{n} < 5$ should be the limit.

Still, it should be noted that our GLOUDS could be further compressed for web-graphs, namely by reducing the wavelet tree size, which makes up for most of the space. This could be achieved by finding similarities, e.g., by using the RePair method on $H$. Hereby, out-neighbors could be listed by "unravelling" the blocks, and in-neighbors could be found by first looking in which blocks the respective node lies in, and then by selects of the block-numbers on the initial vector, which would not be a trit-vector any more. This should lead to higher access times and higher creation time of the GLOUDS, in favour of reducing space for special graphs. As we focused our GLOUDS representation on arbitrary tree-like graphs, we leave this open to

further research.

## 4.6 Chapter Summary

In this chapter we presented a framework and implementation for a new succinct data structure for "tree-like" graphs based on the LOUDS representation for trees, which we called GLOUDS. We took a three-pronged approach to evaluate our representation. Firstly, we created random data structures inspired by phylogenetic networks with 10% of non-tree edges and compared the runtime of the operations children and parents on every node matched to a pointer based representation. The evaluation confirmed that our succinct data structure is practically feasible with a space reduction of around 95%. Secondly, a test on general graphs was performed by the use of a succinct library, also compressing the pointer-based data-structure. The practical evaluations again confirmed that the GLOUDS achieves a significant space reduction. Lastly, we showed how well our data-structure performs against a framework for web-graphs. The evaluation on "real-world" graphs shows that the GLOUDS performs well for graphs where $m < 10n$. In total, a trade-off between space and time can be observed, which is common in the world of succinct data structures.

# 5 Filesystem Level Order Unary Degree Sequence

## 5.1 Introduction

Data exchange between devices over the Internet has become an important aspect, nowadays. In the era of the Internet of Things (IoT), the number of these devices will, according to Gartner[111], exceed 25 billion in the year 2020. As storage units have become smaller and cheaper, these devices can already save millions of files. Considering that in the future the automatic data exchange between these devices will blossom, the creation of a small data structure listing all the files on a device is important. Despite being small, this data structure should also be quickly traversable. It should list as much information about a file as possible to check, for example, the name, the format, the size, and the checksum.

Listing these properties is also interesting for remote maintenance and attestation. One example are measuring instruments under legal control, e.g., commodity meters for the supply of gas, water and electricity, traffic enforcement cameras, etc. In many states, national laws even oblige the manufacturer of these devices to output a software identifier for the user and the market surveillance. This identifier should be created in a way that data integrity can be inferred. Often, this is done by just calculating a checksum over all the files that seem to be important for the measuring task and showing it on the screen. Considering that remote maintenance and automatic scanning of the devices will be the next step to achieve remote integrity checking, a file list containing as much information as possible is preferable.

This chapter enhances the data structure from Chapter 4 which makes use

of succinct approaches to store trees, with file properties. In this structure, a fast file search is made possible by using space-efficient algorithms to store the file names. Hence, the data structure is not only usable as a file list, but can easily be used as the fundamental structure for a read-only file system in which files can be located and listed efficiently.

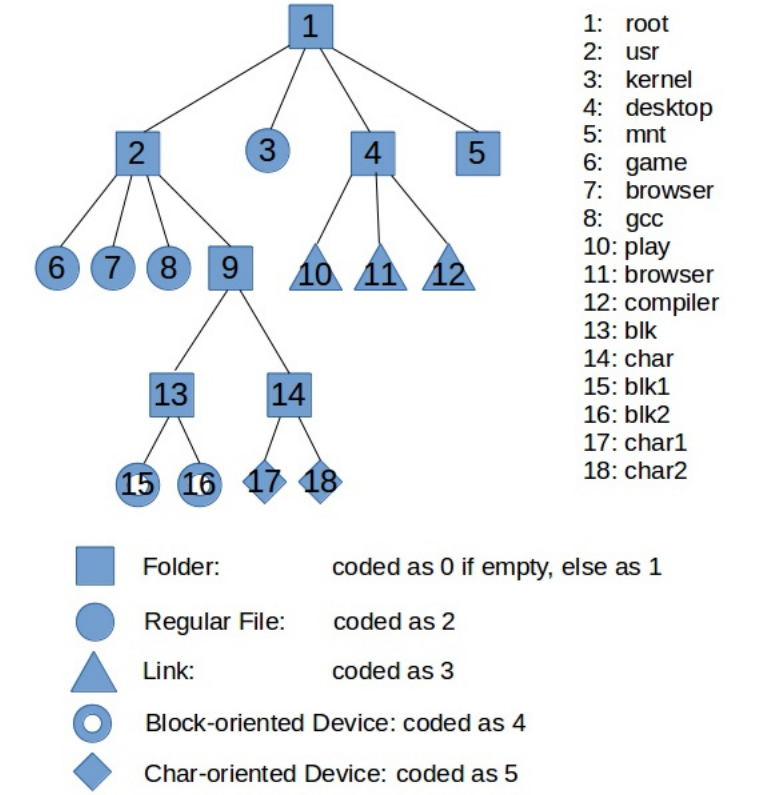
## 5.2 The FLOUDS

An approach that is also based on the LOUDS to store tree-like graphs, which file systems with links can be regarded as, was explained in Chapter 4. We first described this method in[95,37] primarily for storing phylogenetic networks (phylogenetic networks are used in biology to express relationships between species). In its original design, the structure consisted of a trit (ternary digit) variant to store the enhanced LOUDS, which is rather not suitable for file systems, because one can only differentiate between two types of files, e.g., links and regular files (the 0s in the trit-variant are used to end the children listing of a node as in the LOUDS). In this section, a new version called FLOUDS ("File-system Level Order Unary Degree Sequence") is described, which makes use of the wavelet tree presentation described in Section 2.4.2 (see also Section 4.4.3). Therefore, files can be divided into more than two types, e.g. the types listed in Figure 5.1:

- regular files
- directories
- links (hard-, soft)
- block-oriented device
- char-oriented device

This list can be expanded, e.g. to sockets, pipes, MIME types etc., or shortened as needed.

The FLOUDS is created as follows: First a BFT at the root of the file system tree is started. For every node (file/folder) a predefined number representing the file type is appended to $S$. In the example of Figure 5.1, we defined following numbers: 0 for an empty folder, 1 for a folder which contains something, 2 for a regular file, 3 for links, 4 for block-oriented devices and 5 stands for char-oriented devices. If the node is the first child of its father, a 1 is written to $B$, otherwise a 0, which is necessary as kind of a start bit to know where the elements in each folder begin and end. The array $S$ can be created directly as a wavelet tree, when the number of file




FLOUDS:

S = 1 1 2 1 0 2 2 2 1 3 3 3 1 1 4 4 5 5

B = 0 1 0 0 0 1 0 0 0 1 0 0 1 0 1 0 1 0

H = 6 7 8

N = r o o t u s r k e r n e l d e s k t o p ... c h a r 2

$B_n$ = 1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 ... 1 0 0 0 0

Figure 5.1: File system structure FLOUDS, consisting of $S$, $B$, $H$, $N$ and $B_n$.

types $t$ is known from the beginning, which is $t = 5$ in the example; if not, it can be created later, after the complete BFT run.

Additionally, the file/folder names are successively written to $N$, whereas $B_n$ marks the beginning of a new file name by a 1. After a complete BFT $S$, $B$, $N$ and $B_n$ are created and the number of nodes $n$ and the number of links $l$ are known. This information can then be used to create the wavelet tree $H$, which is needed to retrieve the link address to the file. In total $H$ requires $l \lg n + o(l \lg n)$ space. In $H$ the FLOUDS numbers of the real files of the links are stored, in the order in which they were traversed by the BFT. Now, $H$ can be used to check if a file is a link, and afterwards get the linked file, as described in the next section.

### 5.2.1 Navigating through the FLOUDS

With the help of $S$ and $B$, a file system tree traversal can be done much like it in the LOUDS. Functions *parent* and *child* are as follows:

- $\text{child}(x, i) = \mathsf{select}_1(B, \mathsf{rank}_1(S, x)) + i - 1$, if $S[x] = 1$ and $i <$ number of children
- $\text{parent}(x) = \mathsf{select}_1(S, \mathsf{rank}_1(B, x))$

For example, in Figure 5.1, the folder-entries of node 9 can be listed by first checking if node 9 is a folder $S[9] = 1$, and then getting its folder-number $f_n = \mathsf{rank}_1(S, 9) = 4$. Afterwards, a jump to its first child is done, $\text{child}(9, 1) = \mathsf{select}_1(B, 4) = 13$. All the children of node 9 are between the first and the last child $l$: $\text{child}(9, l) = \mathsf{select}_1(B, 5) - 1 = 14$ (the position before encountering the next '1': $4 + 1 = 5$). So the folder-entries of node 9 are 13 and 14 (and $l = 2$).

The array $H$ can be used to check if a node is a link (in the example from Figure 5.1: $S[n] = 3$) or if a file is referenced by links, and where these links are situated in the file system tree, by using the wavelet tree of $H$:

- $\text{getLink}(n, i) = \mathsf{select}_3(S; \mathsf{select}_n(H; i))$
- $\text{getOrig}(n) = H[\mathsf{rank}_3(S; n)]$, if $S[n] = 3$

Looking again at the example of Figure 5.1: the first link that points to node 7 is $\text{getLink}(7, 1) = \mathsf{select}_3(S; \mathsf{select}_7(H; 1)) = 11$; and the file-number (node) 12 points to is $\text{getOrig}(12) = H[\mathsf{rank}_3(S; 12)] = 8$.

Additionally, with the help of *rank*- and *select*, the following functions are directly executable:

- Getting the number of files in a folder and listing them.
- Getting the number of files of a specific file-type in a folder and listing them.

For example, if every file is assigned a MIME-type number, all pictures in the file system can be listed efficiently.

### 5.2.2 Efficiently Storing File Names

In our description of the FLOUDS in Section 5.2, we construct a simple representation for file names by writing them successively into $N$. A simple method to find files faster can be achieved by sorting the folder entries in alphabetical order. Hereby, the prefix of a file name in a folder can be found by a binary search.

A more efficient search that also finds substrings of file names can be achieved by applying 2-Way dictionaries. An example would be using the Burrows-Wheeler Transform (BWT)[20] with Run-Length Encoding. Such a method is described in[33], for example. Hereby, substrings can be found efficiently, and afterwards be mapped to their FLOUDS numbers, or the file names can be read out via the FLOUDS numbers, respectively.

Another method that aims at compression, is described in[4]. There, the Lempel-Ziv algorithm 78 (LZ78[117]) is altered in a way that makes locating prefixes possible. The LZ78 compression algorithm for a string $S[1, n]$ proceeds by parsing $S$ from left to right. Hereby, $S$ is divided into blocks that are one-letter extensions of previously parsed substrings. The set of current blocks is called the phrase dictionary $D$. The dictionary $D$ is prefix-closed and represented with a trie (the LZ-trie), which is stored in[4] with some enhancements to achieve look-up and access support. The method is good for checking the integrity of file system structures, because finding substrings of file names is not really needed (the exact file name is known a priori).

### 5.2.3 Integrity checking

There are several variations to check file system integrity on request:

1. The FLOUDS is signed and transferred with the file names.
2. The FLOUDS is newly created and while traversing the file system tree, a hash value of each file is calculated. These hash values are written into a hash array of size $n$ and transmitted together with the FLOUDS and the file names.

3. To save space, the file name list can be omitted. The file names can just be included in the computation of the hash values, mentioned in point 2.

4. Only a predefined number of files are hashed to save even more space.

5. Only one hash value over the entire structure is calculated and transmitted.

The fifth method requires the least amount of space, because only a hash value needs to be transmitted. This value can, for example, be displayed on the device's display. If the hash has an unexpected value, one of the other four methods can be executed to check, how the file system structure has changed and which files have been altered.

The used hash algorithm should be as collision-free as possible. Still, it can be freely chosen, for example from a simple checksum like CRC16, to secure hashing algorithms such as SHA-2, depending on performance, space or safety demands.

## 5.3 Practical Results

The aim of this section is to show the practicality of our approach by comparing it with a well known database for Unix systems, which the *locate* command uses to efficiently find files.

For our test we used the succinct libraries `https://github.com/ot/succinct` and `https://github.com/simongog/sdsl`, which have well-tuned succinct data structure implementations (other sources are[3,42]). Our machine was equipped with an Intel Core i7@2.2GHz and 8GB of RAM, running under Ubuntu 14.04.

Table 4.1 shows the sizes of the *mlocate* database, which in Unix systems is normally situated at "/var/lib/mlocate/mlocate.db", the size of this database compressed by 7zip, the size of the FLOUDS with the file names just stored in plain text, and the lzFLOUDS with the file names stored by the LZ78 method[4] described in Section 5.2.2.

We used 4 different file lists for comparison:

1. buildroot: buildroot (`http://buildroot.uclibc.org/`) is a tool to generate embedded Linux systems, the file system we generated contained 435 nodes (files/folders/link etc.).

2. linux_src: The Linux 4.2 Kernel source tree containing 54 171 nodes.

3. comp1: A small file system of a desktop computer containing 333 854 nodes.

4. comp2: A bigger file system of another desktop computer with 1 853 354 nodes.

Table 5.1: Comparison of sizes in MB: 1. buildroot, 2. linux_src, 3. comp1, 4. comp2.

| F | mlocate | mlocate.7z | FLOUDS | lzFLOUDS |
|---|---|---|---|---|
| 1. | 0.004 | 0.002 | 0.003 | 0.002 |
| 2. | 0.957 | 0.193 | 0.625 | 0.218 |
| 3. | 7.722 | 1.093 | 4.964 | 1.932 |
| 4. | 42.876 | 7.494 | 22.928 | 9.141 |

The second table, Table 5.2, compares the running times to find a file name. Hereby, the LZ78 variant of the FLOUDS searched for exact matches, whereas the other two searched for substrings. We averaged the running times over 100 tests, searching for random strings.

Table 5.2: Comparison of running times in sec: Legend as in Table 4.1.

| F | mlocate | FLOUDS | lzFLOUDS |
|---|---|---|---|
| 1. | 0.004 | 0.027 | $8 * 10^{-6}$ |
| 2. | 0.035 | 0.105 | $10 * 10^{-6}$ |
| 3. | 0.381 | 0.565 | $9 * 10^{-6}$ |
| 4. | 0.823 | 9.854 | $12 * 10^{-6}$ |

It can be observed that the naive FLOUDS representation is already around 40% smaller than the *mlocate* database. Still the running times for file-systems with many files are up to 12 times slower. On the other hand, the lzFLOUDS is some magnitudes faster than all the other structures and can compete with the 7zip compressed *mlocate* database in size, which cannot be traversed. The drawback of the lzFLOUDS is that it can only be used to find prefixes, and in our tests it just output a result, if the exact match was found. We think that this is enough, if the FLOUDS is used for integrity checking, because the exact file names should be known.

## 5.4 Discussion

Succinct data structures perform very well on static objects and not that well on dynamic ones. The same applies to the FLOUDS, hence it is better suited for a read-only file system. For many embedded devices this poses no restriction, because mostly they are put in commission to fulfill only a certain scope; changes to the file system structure are often not allowed and would be a sign of malicious manipulation. However, if changes are to be made, the FLOUDS needs to be rebuilt. A fast rebuild can be easily achieved if the names are not compressed and saved in plain form. For the other representations, the file name string needs to be completely rebuilt, which can be slow. Still, dynamic succinct data structures exist that can be used to this end, e.g.,[72].

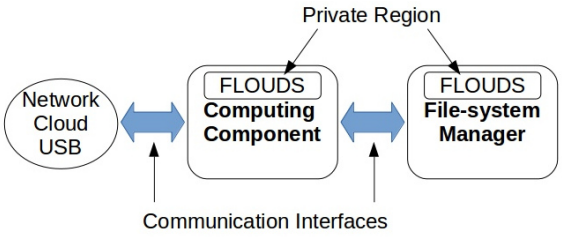



Figure 5.2: Separating the file system Manager

Figure 5.2 shows how the file system manager should be separated from the computing component to hinder malicious applications from tampering with files if file integrity is an issue. These components can be separate devices, where the file system component is at a secure location. Hereby, the computing component could have a copy of the FLOUDS in its internal storage, to speed up locating files. If a file is needed, the FLOUDS-number and the number of requested bytes can be sent to the file system manager, which then answers the request. This communication can be encrypted if an open network is used for communication. In the same manner, an entitled entity can check the integrity of the file system by just sending a request to the computing component, which in turn gets the FLOUDS encrypted with the private key of the file system manager. Afterwards, it sends this encrypted FLOUDS back to the entitled authority. After decrypting it with the public key of the file system component, the entitled authority can make sure that the computing component did not manipulate the FLOUDS-file (assuming of course that the private key of the file system manager is not known by the computing component).

Another approach is, of course, software-separation through a micro-/separation kernel and virtualization as described in this thesis. But as mentioned, there are many papers that come to the conclusion that the microkernel approach for virtualization is recommended to construct secure systems[56,75,70,57,54,94]. In our system architecture, the calculation of the FLOUDS could be handled by the Inspector.

Another area of application for the FLOUDS is to use it as the fundamental structure of a whole file system. A possible approach would be to combine it with a read-only, compressed file system like squashFS (`http://squashfs.sourceforge.net/`). The idea is to use the FLOUDS numbers of the nodes to represent the block numbers, which in turn point to compressed blocks. Nevertheless, if compression is not that important, the FLOUDS needs to be evaluated against a dynamic file system, e.g., based on the $B^\epsilon$-tree, because these show good results for locating files and have fast write operations[32,61].

## 5.5 Chapter Summary

In this chapter, a flexible file list structure called FLOUDS was presented. This structure was explained in detail to show that it can be used in many ways. Firstly, it is well suited for embedded devices that need to be validated for integrity in commission, or which want to exchange file lists. Secondly, it can be used as the groundwork for a read-only file system, which uses as little space as possible. Lastly, it can be used to efficiently find and list files by using succinct data structures for trees and 2-way dictionaries. For the latter, the FLOUDS was evaluated by comparing it to another file name database, which is used in Unix systems by the *locate* command. These tests show that the FLOUDS stores more information, is smaller, and for integrity checking also faster.

# 6 Conclusion

This thesis analyzed ways of constructing secure software systems for measuring instruments under legal control. Hereby two main points emerged. First, current systems were not constructed with security in mind, which leads to the point to construct a new software system architecture which is tailored to all the legal requirements, especially with security in mind. Second, spotting malicious manipulation, and making remote attestation and maintenance possible is very important. Hereby, a watchdog, we called Inspector, was inserted into the system architecture from ground up to scan the virtual machines if needed. Additionally, a data structure was created (FLOUDS) which helps in checking file system integrity.

The presented system architecture was constructed by the requirements for measuring instruments under legal control in Europe, e.g. the WELMEC 7.2 Software Guide and the Measuring Instruments Directive (MID). We combined these requirements with methodologies and concepts from high-assurance software systems, *i.e.*, MILS and PHASE. The creation of the architecture took a three-pronged approach:

1. The legally relevant parts were separated from the irrelevant ones by putting them in different virtual machines
2. Their virtual machines have no direct access to I/O devices
3. A secure framework was constructed, which provides services to these VMs. This framework, also consisting of separated VMs, monitors the information flow, correctly delegates requests from and to I/O devices and helps control agencies to verify system integrity.

To harness device drivers and network stacks of general purpose operating systems, we came to the conclusion that virtualization through a small

microkernel is the right solution to combine security with usability. Software security is enhanced through the microkernel because nowadays, most measuring instruments are connected over open networks just running general purpose operating systems. Furthermore, the framework was evaluated by building a system atop an L4-microkernel. For our demonstrator (PandaBoard Revision B3 equipped with the OMAP4460 SoC running a dual-core 1.2 GHz ARM Cortex-A9 MPCore with 1 GiB of DDR2 SDRAM), we used L4Linux running atop the open source Fiasco.OC L4-microkernel, which yields good results, even for real-time applications. These tests show that it is practically feasible to construct a configurable framework using our architecture, which is applicable for powerful measuring instruments under legal control.

In the second part of this thesis, a new succinct data structure for "tree-like" graphs based on the LOUDS representation for trees, which we called GLOUDS, was also evaluated by a three-pronged approach:

1. We created random data structures inspired by phylogenetic networks with 10% of non-tree edges and compared the runtime of the operations children and parents on every node matched to a pointer based representation. The evaluation confirmed that our succinct data structure is practically feasible with a space reduction of around 95%.

2. A test on general graphs was performed by the use of a succinct library, also compressing the pointer-based data-structure. The practical evaluations again confirmed that the GLOUDS achieves a significant space reduction.

3. We showed how well our data-structure performs against a framework for web-graphs. The evaluation on "real-world" graphs shows that the GLOUDS performs well for graphs where $m < 10n$. In total, a trade-off between space and time can be observed, which is common in the world of succinct data structures.

Afterwards out of the GLOUDS we created a flexible file list structure called FLOUDS. The validation of software in commission is very important and required by law (see MID[88] Annex I 7.6, Annex I 8.2, Annex I 8.3). With the FLOUDS, it is easier for the market surveillance to check software integrity. This structure was explained in detail to show that it can be used in many ways:

1. It is well suited for embedded devices which need to be validated for integrity in commission, or which want to just exchange file lists.

2. It can be used as the fundamental file system structure of a read-only file system which uses as little space as possible.

3. It can be used to efficiently find and list files by using succinct data structures for trees and 2-way dictionaries.

By combining the FLOUDS with our system architecture a complete framework can be created that fulfills all legal requirements in Europe and makes verification and validation before and in commission easily possible.

## 6.1 Future Research

It is expected that in the next years the industry will develop and implement realistic strategies for the use of Cloud Computing in legal metrology. The architecture described in this thesis, which was developed for embedded devices can easily be adopted for Cloud applications. Hereby, the suggested architecture can be deployed by the manufacturer in cooperation with the cloud provider. It should scale well with the legal requirements, and enforces security by virtualization. In Fig. 6.1, an approach, which we first described in[108], can be seen that tailors the system architecture for cloud systems (a recent research project took up this architecture in[91]).

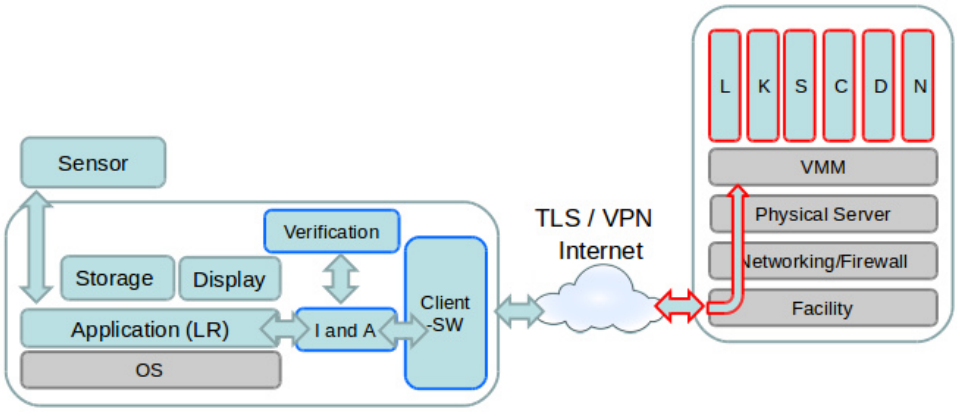



Figure 6.1: Cloud Framework: Securing Software through virtualization. (I: Integrity, A: Authenticity, LR: legally relevant, VMM: Virtual Machine Monitor, L: legally relevant VM, K: Key & Signature Manager, S: Storage Manager, C: Connection Manager, D: Dowload Manager, N: non-legally relevant VM) taken from[108]

In this framework legally relevant tasks (tasks that are needed for the measurement purpose) are separated from non-legally relevant ones by putting

them in different VMs, the N VM (non-legally relevant VM) and the L VM (legally relevant VM).

The rest of the cloud-framework consists of four basic virtual machines that help to fulfill the legally relevant functions. These VMs can be present in a plurality of units depending on the workload.

1. The Key & Signature Manager VMs save the keys and signatures which are communicated only to the L VM to check, encrypt and decrypt data.

2. The Connection Manager VMs receive and send the data from and to the internet and to the other VMs. They are the only VMs which are directly accessible from the internet. Because they do not have access to the keys, they cannot be used to modify data unnoticed. The L VMs encrypt or sign the data before it is send to the Connection Manager.

3. The Storage Manager VMs handle the data storage and have exclusive access rights to the storage infrastructure of the cloud. If data should be retrieved or saved the other VMs talk to the Storage Manager.

4. The Download Manager VMs are responsible for software updates and patches. Before the software of the other VMs are updated, the update is first downloaded into a Download Manager, where it is checked and afterwards distributed.

This modular system architecture scales well, because the virtual machines can just be copied, multiplied and dynamically swapped from server to server, whenever more or less computation time is needed. It is even possible to swap out non-legally relevant task (N VMs) to a public cloud which is not subject to legal control.

# References


[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006.

[2] J. Alves-foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS architecture for high-assurance embedded systems. Journal of Embedded Systems, 2:239-247, 2006.

[3] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX*, pages 84–97. SIAM, 2010.

[4] J. Arz and J. Fischer. LZ-Compressed String Dictionaries. In *Data Compression Conference (DCC)*, pages 322 – 331, 2014.

[5] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In Proceedings of the 2009 Annual Computer Security Applications Conference, Honolulu, HI, USA, pp. 461–470. 7–11 December 2009.

[6] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. LATIN*, volume 6034 of *LNCS*, pages 170–183. Springer, 2010.

[7] J. Barr. The Flask Security Architecture. In Computer Science 574. 2002.

[8] R. W. Beckwith, W. M. Vanfleet, and L. MacLaren. High Assurance Security/Safety for Deeply Embedded, Real-time Systems. *Embedded Systems Conference*, 2004.

[9] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. ESA*, volume 7501 of *LNCS*, pages 181–192. Springer, 2012.

[10] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[11] W. R. Bevier. Kit and the short stack. In J. Autom. Reasoning 5(4), 519–530, 1989.

[12] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proc. SODA*, pages 679–688. ACM/SIAM, 2003.

[13] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *ALENEX/ANALC*, pages 49–61. SIAM, 2004.

[14] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

[15] D. R. Boccardo, L. F. R. da Costa Carmo, M. H. Dezan, R. C. S. Machado, and S. de Aguiar Portugal. Software evaluation of smart meters within a Legal Metrology perspective: A Brazilian case. In Proceedings of the Innovative Smart Grid Technologies Conference Europe (ISGT Europe), Gothenburg, Sweden, 11–13 October 2010; pp. 1–7.

[16] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[17] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[18] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[19] P. Bolignano , T. Jensen, and V. Siles. Modeling and Abstraction of Memory Management in a Hypervisor. In Fundamental Approaches to Software Engineering, Springer LNCS, pp. 214–230, 2016.

[20] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[21] B. Chelf. Measuring Software Quality - A Study of Open Source Software. Coverity, 2011.

[22] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In ASPLOS14, ACM, 2014.

[23] D. Clark. Compact Pat Trees. Phd Thesis presented to the University of Waterloo, Canada. 1996.

[24] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Trans. Web, 4:(4), 2010.*

[25] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *Proc. COCOON*, LNCS, pages 396–407. Springer, 2012.

[26] C. B. Do Prado, D. R. Boccardo, R. C. S. Machado, L. F. R. da Costa Carmo, T. M. do Nascimento, L. M. S. Bento, R. O. Costa, C. G. de Castro, S. M. Camara, L. Pirmez, *et al.* Software Analysis and Protection for Smart Metering. *NCSLI Meas.* 2014, *9*, 22–29.

[27] K. Doeornemann, and A. von Gernler. Cybergateways for Securing Critical Infrastructures. In Proceedings of International ETG-Congress 2013, Symposium 1: Security in Critical Infrastructures Today, Berlin, Germany, 5–6 November 2013.

[28] J. Dörrenbächer. Formal Specification and Verification of a Microkernel. PhD thesis at "Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes", Saarbrücken, November 2010.

[29] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4), April 2009.

[30] K. Elphinstone and S. Götz. Initial Evaluation of a User-Level Device Driver Framework. In *9th Asia-Pacific computer systems architecture conference*, 2004.

[31] K. Elphinstone and G. Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, 2013. ACM.

[32] J. Esmet, M. A. Bender , M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*, 2012.

[33] P. Ferragina and R. Venturini. The compressed permuterm index. In *ACM Trans. Algorithms 7, 1*, Article 10, 21 pages, 2010.

[34] A. Farzan and J. Fischer. Compact representation of posets. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 302–311. Springer, 2011.

[35] A. Farzan and J. I. Munro. Succinct representation of arbitrary graphs. In *Proc. ESA*, volume 5193 of *LNCS*, pages 393–404. Springer, 2008.

[36] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. SWAT*, volume 5124 of *LNCS*, pages 173–184. Springer, 2008.

[37] J. Fischer and D. Peters. A Practical Succinct Data Structure for Tree-Like Graphs. In *WALCOM: Algorithms and Computation*, pages 65–76, LNCS Springer, 2015.

[38] J. Fischer and D. Peters GLOUDS: Representing tree-like graphs. Elsevier, Journal of Discrete Algorithms; 11/2015.

[39] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. ARM TrustZone as a Virtualization Technique in Embedded Systems. *Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi*, 2010.

[40] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 19–22 October 2003; pp. 193–206.

[41] C. Gavoille and N. Hanusse. On compact encoding of pagenumber $k$ graphs. *Discrete Mathematics & Theoretical Computer Science*, 10(3):23–34, 2008.

[42] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[43] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *9th SIGOPS European Workshop*, 2000.

[44] S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proc. ALENEX*, pages 25–34. SIAM, 2011.

[45] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. SODA*, pages 368–373. ACM/SIAM, 2006.

[46] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[47] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.

[48] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, volume 7933 of *LNCS*, pages 5–17. Springer, 2013.

[49] Guide to the Secure Configuration of Red Hat Enterprise Linux 6. Available online: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/index.html (Accessed: 2016-07-31).

[50] Y. Gurevich, L. Stockmeyer, and U. Vishkin. Solving NP-hard problems on graphs that are almost trees and an application to facility location problems. *J. ACM*, 31(3):459–473, 1984.

[51] S. Hand, A. Warfield, K. Fraser, E. Kottsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? *Proceedings of the 10th Workshop on Hot Topics in Operating Systems, Santa Fe*, 2005.

[52] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[53] H. Härtig, J. Loeser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O Architecture for Mikrokernel-Based Operating Systems. *TU Dresden technical report TUD-FI03-08, Dresden*, July 2003.

[54] G. Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, New York, NY, USA, 2008. ACM.

[55] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM Operating Systems Review*, July 2007.

[56] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.*, 40(1), January 2006.

[57] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-machine Monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM.

[58] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel: the VFiasco project. In ACM SIGOPS European Workshop, pp. 165–169, 2002.

[59] D. H. Huson and C. Scornavacca. A survey of combinatorial methods for phylogenetic networks. *Genome Biology and Evolution*, 3:23, 2011.

[60] G. J. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

[61] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, 2015.

[62] S. Joannou and R. Raman. Dynamizing succinct tree representations. In *Proc. SEA*, volume 7276 of *LNCS*, pages 224–235. Springer, 2012.

[63] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. *SIAM J. Discrete Math.*, 5(4):596–603, 1992.

[64] T. Kerstan, D. Baldin, and S. Groesbrink. Full virtualization of real-time systems by temporal partitioning. *Proceedings of the Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels*, 2010.

[65] D. Kleidermacher and M. Kleidermacher. *Embedded Systems Security.* Newnes, 2012.

[66] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*. ACM, 2009.

[67] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, 3rd edition, 1997.

[68] A. Lackorzynski, J. Danisevskis, J. Nordholz, and M. Peter. Real-Time Performance of L4Linux. *Proceedings of the Thirteenth Real-Time Linux Workshop, Prague*, 2011.

[69] A. Lackorzynski, A. Warg, and M. Peter. Virtual Processors as Kernel Interface. *Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi*, 2010.

[70] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, New York, NY, USA, 2011. ACM.

[71] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. DCC*, pages 296–305. IEEE Press, 1999.

[72] S. Lee, and K. Park. Dynamic Rank-Select Structures with Applications to Run-Length Encoded Texts. In *Lecture Notes in Computer Science 4580*, Springer, pages 95–106, 2007.

[73] N. Leffler and F. Thiel. Im Geschäftsverkehr das richtige Maß. In *Schlaglichter der Wirtschaftspolitik*, Monatsbericht November, 2013.

[74] M. LeMay, and C. A. Gunter. Cumulative Attestation Kernels for Embedded Systems. *IEEE Trans. Smart Grid* 2012, *3*, 744–760.

[75] A. S. Liebergeld, M. Peter, and A. Lackorzynski. Towards Modular Security-Conscious Virtual Machines. *Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi*, 2010.

[76] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, New York, NY, USA, 1995. ACM.

[77] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[78] J. I. Munro. Tables. In *Proc. FSTTCS*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.

[79] J. I. Munro and P. K. Nicholson. Compressed representations of graphs. In M. Kao, editor, *Encyclopedia of Algorithms*. Springer, 2015.

[80] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 345–356. Springer, 2003.

[81] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. FOCS*, pages 118–126. IEEE Computer Society, 1997.

[82] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[83] NIST Recommended Key length. http://www.keylength.com/en/4/. Accessed: 2016-07-31.

[84] P.G. Neumann and R.J. Feiertag. PSOS revisited. In ACSAC, pp. 208–216. IEEE Computer Society 2003.

[85] M. Neve and J.-P. Seifert. Advances on Access-driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, SAC'06, Berlin, Heidelberg, 2007. Springer-Verlag.

[86] D. O'Dowd. RTC Magazine: Green Hills Software. http://www.rtcmagazine.com/articles/view/101494. Accessed: 2016-07-31.

[87] Official Journal of the European Union. *DIRECTIVE 2004/22/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL*, March 2004.

[88] Official Journal of the European Union. *DIRECTIVE 2014/32/EU OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL*, February 2014.

[89] OIML. *International Vocabulary of Terms in Legal Metrology*, July 2013.

[90] OIML D 31. *General requirements for software controlled measuring instruments*, 2008.

[91] A. Oppermann, J.-P. Seifert, and F. Thiel Secure cloud reference architectures for measuring instruments under legal control. Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016): 1 (2016), pp. 289 - 294.

[92] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[93] M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313. IEEE Computer Society, 2008.

[94] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, VDTS '09, New York, NY, USA, 2009. ACM.

[95] D. Peters. Platzsparende Datenstrukturen für explizite phylogenetische Netzwerke. Diplomarbeit am Karlsruher Institut für Technologie, 2013.

[96] D. Peters, J. Fischer, J.-P. Seifert, and F. Thiel FLOUDS: Eine platzsparende Dateisystemstruktur. DPMA - Deutsches Patent- und Markenamt / German Patent and Trademark Office, DE 10 2016 110 479.5; (07.06.2016).

[97] D. Peters, U. Grottker, F. Thiel, M. Peter, and J.-P. Seifert Achieving software security for measuring instruments under legal control. Federated Conference on Computer Science and Information Systems, 09/2014.

[98] D. Peters, M. Peter, J.-P. Seifert, and F. Thiel A Secure System Architecture for Measuring Instruments in Legal Metrology. MDPI, Computers; 03/2015, 4(7):61-86.

[99] D. Peters and F. Thiel Software in Measuring Instruments: Ways of Constructing Secure Systems. SENSOR 2016, Nuremberg; 05/2016.

[100] D. Peters, F. Thiel, M. Peter, J.-P and . Seifert A Secure Software Framework for Measuring Instruments in Legal Metrology. Instrumentation and Measurement Technology Conference (I2MTC), 2015 IEEE International, Pisa; 05/2015.

[101] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. *ACM Trans. Algorithms*, 3(4):Article No. 43, 2007.

[102] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst*, 41(4):589–607, 2007.

[103] D. Sangorrin, S. Honda, and H. Takada. Dual Operating System Architecture for Real-Time Embedded Systems. *Proceedings of the Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels*, 2010.

[104] S. Samolej. Arinc specification 653 based real-time software engineering. E-Informatica, 5(1):39-49, 2011.

[105] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. NAI Labs Report 2001.

[106] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In Proceedings of the 5th European Conference on Computer Systems (EuroSys '10), Paris, France, 13–16 April 2010; pp. 209–222.

[107] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[108] F. Thiel, M. Esche, D. Peters, and U. Grottker Cloud computing in legal metrology. 17th International Congress of Metrology: (2015).

[109] F. Thiel, U. Grottker, and D. Richter. The challenge for legal metrology of operating systems embedded in measuring instruments. *OIML Bull.* 2011, *52*, pp 5–14.

[110] F. Thiel, U. Grottker, and D. Richter. The challenge for legal metrology of operating systems embedded in measuring instruments. In *OIML Bulletin*, 52 (LII). OIML Bulletin, 2011.

[111] A. Velosa, J. F. Hines, H. LeHong, E. Perkins, R. M Satish. Predicts 2015: The Internet of Things. In https://www.gartner.com/doc/2952822/predicts-internet-things, 2014.

[112] J. Vetter, M. Junker-Petschick, J. Nordholz, M. Peter, and J. Danisevskis. Uncloaking Rootkits on Mobile Devices with a Hypervisor-based Detector. In Information Security and Cryptology (ICISC 2015), Springer LNCS, 2015.

[113] B. J. Walker, R. A. Kemmerer, and G.J. Popek. Specification and verification of the ucla unix security kernel. Commun. ACM 23(2), 118–131, DOI http://doi.acm.org/10.1145/358818.358825, 1980.

[114] WELMEC European cooperation in legal metrology. *WELMEC 7.2 Issue 5 Software Guide*, March 2012.

[115] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat USA*, Invisible Things. Las Vegas, NV, USA, 2008. Lab.

[116] Y. Zhao, M. Dianfu, and Y. Zhibin. A survey on formal specification and verification of separation kernels. Technical report, Tech. rep., National Key Laboratory of Software Development Environment (NLSDE). Beihang Univerisity, 2014.

[117] J. Ziv and A. Lempel Compression of individual sequences via variable-rate coding. In *Information Theory, IEEE Transactions*, 24(5), pages 530–536, 1978.