Security and Trust in Virtualized Environments

Hagen Lauer

Supervisors: A/Prof. Dr. Carsten Rudolph,
Dr. Surya Nepal

Faculty of Information Technology
Monash University

This dissertation is submitted for the degree of
Doctor of Philosophy

January 2020

This research was supported by the Commonwealth Scientific and Industrial Research Organization (CSIRO) Data61 Postgraduate Research Scholarship as well as Monash University's Faculty of Information Technology (FIT) Tuition Fee Scholarship.

Abstract

Virtualization is a core concept in modern computing systems and clients place a vast amount of trust in the virtualization system to provide essential security guarantees such as data confidentiality and software integrity [47, 9, 116, 70]. A virtualization system's unlimited access to software and data in virtual environments presents a genuine scientific challenge. The Trusted Computing Module (TPM) as part of a trusted platform can be used to establish trust in a computer and this thesis discusses challenges and presents solutions related to establishing trust in a virtual environment.

The first finding is that currently available trust establishment strategies do not sufficiently support clients in establishing trust in their virtual environments. After reviewing relevant standards and related work, a *User-Centered Attestation* approach is defined as a set of requirements. The thesis reveals that trust establishment strategies must consider system layering and individual virtual environments. Recording integrity information relevant to a virtual environment in a trustworthy way is a key challenge.

The next part of this work is devoted to the design and verification of an *Enhanced Integrity Measurement Architecture* suitable for popular operating system virtualization technologies such as containers. To verify that the developed integrity measurement architecture is trustworthy and upholds necessary security guarantees, a formal system called LS^3 is presented and used in the design and verification of trusted virtualization systems.

Lastly, the virtual TPM as part of a virtual trusted platform is analyzed. Typically, the TPM is virtualized to provide an instance for each virtual environment. This thesis shows that any straightforward implementation of this may fall victim to a *Goldeneye* attack which uses a virtual TPM against a verifier. The attack is demonstrated using a formal model which captures relevant components of virtualization systems as well as trusted computing axioms. Potential solutions for commodity systems are discussed with the conclusion that further support is needed for virtual platforms that aspire to be trusted.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Hagen Lauer
January 2020

Publications during enrollment

- Lauer, H., Rudolph, C., and Nepal, S. (2018). User-centered attestation for layered and decentralized systems. In *Network and Distributed Systems Security (NDSS) Symposium 2018, Workshop on Decentralized IoT Security and Standards (DISS), 18-21 February 2018, San Diego, CA, USA*. ISOC [Published]

- Lauer, H., Sakzad, A., Rudolph, C., and Nepal, S. (2019b). A logic for secure stratified systems and its application to containerized systems. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications*, pages 1–8, Rotorua, New Zealand. IEEE [Published]

- Lauer, H., Rudolph, C., Nepal, S., and Sakzad, A. (2019a). Bootstrapping trust in a "trusted" virtualized platform. In *Workshop on Cyber-Security Arms Race (CSYARM) 2019, The 26th ACM Conference on Computer and Communications Security (CCS), 15 November 2019, London, UK*. ACM [Best Paper]

- Salehi, A., Lauer, H., Grobler, M., Rudolph, C., and Sakzad, A. (2020). Access control, key and database management, and trust for emerging wireless body area networks in healthcare application. *IEEE Journal of Biomedical and Health Informatics*, pages 1–15 [Submitted]

- Lauer, H., Rudolph, C., and Nepal, S. (2020). Design and analysis of a modern virtual trusted platform. *TBD*, pages 1–30 [Manuscript]

Impact during enrollment

At the time of the submission, this research has had considerable impact. So far our work has led to:

- Chair position of Virtual Platform Work Group [134] in the Trusted Computing Group [132] (Hagen Lauer, Monash and Rob Spiger, Microsoft).
- A new scope for a high-level Virtual Trusted Platform Architecture Specification based on our models and designs in Chapters 3 to 5.
- Several rounds of commentary to ISO 27070 [67, 66] in 2018, 2019, and 2020.

Some of the results of this thesis have also been presented and communicated to subject matter experts on occasions such as sponsored, invited, and R&D talks:

- "Research Proposal: Security and Trust in Virtual Environments" — SIGSOFT Scholarship, ACM 50 Turing Award Conference, 2017
- "Deciding Trust in Distributed Systems" — Invited Talk, Microsoft Trusted Computing Summit, 2017
- "ISO/IEC 27070 - Security requirements for virtualized roots of trust" — TCG commentary 2018 and 2019
- "Trustworthy Trusted Computing with LS^3" — Invited Talk, Trusted Computing R&D Session, 2019
- Virtual Trusted Platform Architecture Specification (updating [56]), 2019

Acknowledgements

I would like to thank my supervisor, Prof. Dr. Carsten Rudolph, for his unwavering trust in me and for his personal support throughout many years. Carsten's deep insights, professional fairness, and personal integrity are truly inspiring. I would also like to thank my co-supervisor, Dr. Surya Nepal, for his persistent optimism and enthusiasm with which he has helped me overcome many hurdles in a short time. As a team, my supervisors were superb and I would like to thank them for numerous discussions, careful advice, helpful feedback, and continuos encouragement.

My co-authors during the years deserve my deepest gratitude and I have benefited immensely from their competence and hard work. I also very much enjoyed the discussions and good times I had with my colleagues and friends Ahmad Salehi, Dr. Amin Sakzad, and Dr. Matthieu Herrmann as well as Prof. Dr. André Rein and Michael Eckel. Special thanks go to several members of the Trusted Computing Group and especially Rob Spiger, Dr. Eugene Myers, Dean Liberty, and Lee Wilson — their insights and tremendous experience have informed my work in many ways. I would also like to thank the reviewers of this thesis for their very helpful comments and suggestions.

I would especially like to thank Prof. Dr. Nicolai Kuntze for his patronage and support as an excellent past supervisor and present colleague and friend. During my undergraduate studies, I was fortunate to have been mentored by Prof. Dr. Michael Jäger who fostered my talents, taught me the meaning of *docendo discimus*, and introduced me to the world of research.

Most of who I am and what I do today is only possible because of my friends and family. I am incredibly lucky to have grown up around my family in a protected and loving environment and with support for my curiosity. I would also like to thank many close friends at home and overseas for the essential distraction and perspective which they have gladly provided. I am forever grateful for the environment which my family and friends have provided and I could not have completed my studies overseas without their close support from home.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Cloud computing realizes the idea that programs and data can be stored and executed on a number of remote computers which can be accessed at any time from anywhere using internet technologies [47, 9, 116]. Driven by increasing connectivity, cloud computing not only allows unprecedented access to programs and data but also augments computing power immensely and immediately on any device. Today, the integration of cloud computing is nearly ubiquitous and successive computing paradigms target remaining applications and architectures [47, 70, 49, 122, 123].

The key factors which drive the success of cloud computing are its flexibility, efficiency, and strategic values [62]. Clients can flexibly scale computing resources based on their application and needs (e.g., an environment for applications, larger systems, or an entire service infrastructure). Furthermore, the economies of scale apply and clients are able to deploy applications quickly and pay only for what they actually use (e.g., resources such as bandwidth, computing power, and storage). Lastly, the ability to focus on an application with ubiquitous access to up-to date and secure systems provides significant strategic values to clients. Undoubtedly, cloud computing has great potential to improve the security of applications in several ways [7, 62]. However, the same argument for possibly improved security can also be used to emphasize newly introduced and very limiting trust issues. Achieving both security and trust in cloud computing involves all aspects of making cloud computing *ver-*

ifiably secure. Many security and trust issues are not novel or specific to the cloud computing field: the cloud systems of today have evolved slowly over time and share their hardware architecture and code base with many other systems. Similarly, trusting a computer and its output has been an active area of research since we started to rely on outputs of stored programs. Consequently, securing cloud computing involves all current computer security disciplines, including secure hardware design, security oriented architectures, operating system and hardware level isolation, software engineering and verification, exploit detection and mitigation, access control in general as well as the design and verification of secure protocols and cloud operations [116]. Yet, with all security related concerns addressed it is still only one half of the solution which we would need to also trust a cloud service. After all, if we have no way to establish or verify that our security goals have been met, we can not integrate cloud services further without blindly trusting a massive remote system. Even worse, we might be convinced to trust and rely upon a service running on an insecure system. The real challenge is in fact to build (and run) secure systems and make others believe it.

Achieving security and trust in cloud security sounds like a monumental task which almost certainly can not be completed adequately within any reasonable period of time. Fortunately, many aspects of this task are shared with other well established domains and many problems are considered scientifically solved or reducible to engineering problems. Some aspects that seem to be characteristic for cloud computing security are [116, 108, 144, 6]:

- Cloud systems are typically shared resources and tenants mutually distrust another.
- Cloud services are accessed via public networks (i.e., the internet) and expose powerful APIs.
- Hosted data is subject to accidental deletion or modification.
- Clients have to trust the cloud provider for important security properties such as confidentiality in their application.

1.1 Scientific Challenge

Arguably, only the cloud providers near unlimited access to hosted services and data poses a practical *and* theoretical challenge today [116, 6]. Sharing a resource between mutually distrustful tenants has been an active area of research since the first time-sharing systems were created and used. Systems which provide strong isolation between different client or security domains have always been a key technology for secure and trusted systems. For instance, a major requirement for hypervisors or virtual machine monitors has always been the ability to isolate virtual machines from another [107, 12]. Furthermore, truly secure operating systems strive for the strongest possible isolation between processes which share the same hardware [74]. The usage of public networks and vulnerable protocols to access powerful APIs is a well known issue and solutions address it by improving on authentication, authorization, cryptography requirements, and protocol verification [116]. Recognizing and dealing with unwanted deletion and modification of cloud services and data has enjoyed a lot of attention in the past and the theoretical challenges are shared with many database and distributed systems. However, experience and the continuos discovery of new security flaws remind us that engineering, deploying, and operating cloud solutions still presents a significant and paramount challenge. We also observe that most of the current cloud security issues are related to implementing theoretical results and developing practical and scalable solutions. In contrast, reducing the amount of trust which is placed in the provider and other parties for important security guarantees such as confidentiality is still a complex issue in practice and theory. Achieving security and trust in this aspect has become even harder as "the cloud" is far more than a few virtual machines on a server. Today, infrastructure providers can be contracted by software providers which can be contracted by other service providers and so on. The cloud changes and seems to strive towards offering highest levels of abstraction for almost every aspect of computing: hardware and infrastructure, virtual servers and operating systems, application run-times and environments, domain specific functions and tasks are abstractions and may be provided as a service and good abstractions are convenient. Consequently, the trust problem

can not be deferred to cryptography alone as clients appreciate the cloud not only for secure storage but also for its high availability and scalable computing power. Clients need universal and complex computations to be performed on the data. In such a case, the cloud service provider will have to perform computations on behalf of the client which results in a hard trust-problem. Today, a variety of commercial services run partially or fully in the cloud and devices offload computations to remote systems while providing only the interface or some representation locally. A few examples are:

- Data and computation intensive tasks, common in today's artificial intelligence applications, can be performed on cloud infrastructure entirely and often alternatives are not feasible.
- Communication and collaboration services such as email, chat, or video conferencing solutions hosted in the cloud are commonplace today even in relatively large organizations.
- Office suites, engineering software, and creative tools are often hosted in the cloud with pay per usage or licensing models.
- Media and documents hosted in the cloud are often not only hosted but also created, accessed, managed, and distributed entirely in the cloud.

The cloud must be able to perform arbitrary computations on a client's data to be useful and verifiably uphold security guarantees to be trustable.

1.2 A Trusted Computing Solution

The contributions (Section 1.8) of this thesis aim to support a trusted computing solution to enforce security guarantees such as confidentiality. Trusted Computing is a security paradigm which relies heavily on the use of unconditionally trusted and hardware anchored security mechanisms with which a secure system is constructed. For the purpose of this introduction, we will consider confidentiality to be the major security property which we want as a security guarantee. The sketch of a possible solution can be very simple but

as ever: *the devil is in the detail.* The objective is for the cloud to compute on sensitive data but only in a secure environment which we can trust. We would like to reduce the amount of trust we place in the cloud provider, administrators, contractors, and other such parties and instead trust the cloud system to uphold necessary security guarantees under which sensitive data may be processed. A trusted computing approach allows a client to define and enforce security guarantees for computations and data processing on a provider's machine. A solution can be outlined as follows [116]:

- The cloud provider must use machines equipped with trusted computing hardware which protects keys and allows binding them to a cloud system's state.
- The client defines a (secure) cloud environment in which programs may run and modify data. Trusted computing techniques make it possible to correlate a cloud system's state s with the defined environment.
- The client generates a key k and binds it to a state s which represents a secure and trusted environment or simply a "good" system state. The bound key can be shared with the cloud system as it is protected by the trusted computing hardware.
- The client can now encrypt programs and data using k and upload them. The cloud system can access data or run programs if it is in the defined "good" state s. However, any divergence from s, including an unknown, unwanted, or a malicious one, revokes access to k and the ability to access data and programs.

As a result, trust in the cloud provider and other parties is reduced and the client relies on trusted hardware and a verifiable software system instead. The approach can be implemented by using trusted computing hardware such as the Trusted Platform Module (TPM) [64] which supports both system state tracking and key management on the provider's machine. The TPM is a widely adopted commodity chip and specifically designed to enable the required key management functionality. Accurately recording and reporting the state of a

complex system requires a co-design of the software system and the secure hardware and is a major topic later on.

1.3 Vision

In this section I will outline a longer term vision for (trusted) computing which I will gradually reduce to a near-term achievable goal: the Virtual Trusted Platform.

I envision a future in which distributed systems and applications can be run on a variety of computers and securely perform even sensitive tasks (e.g., analyzing and acting upon sensor data, controlling power networks, accessing emergency systems, managing traffic). Such systems are able to efficiently utilize the capabilities, flexibility, and performance afforded by future computers, a large amount of computing devices, and increased connectivity. Agents in such systems may be other autonomous systems, cyber-physical machines, or humans represented by their personal devices as interfaces. Agents of any kind should be able to access data and computing power anywhere and at any time with confidence and without reasonable doubt that their data or computations aren't stolen and all tasks are performed as specified.

The question of whether there will be an independent personal computer (i.e., a computer which provides all essential functionality for a single user without relying on others) in this future or which role it will assume is fascinating. It is likely that our personal computing devices will be designed as interfaces to applications with great potential to increase accessibility, making them last longer, and become more sustainable overall. The matching backend functionality will likely see more decentralization (in the sense of location and platform but not ownership) and can be brought closer to the relevant agents to reduce possible lag and overhead. In such systems, security, privacy, or trust are also significantly improved. Future computers support roots of trust which in turn facilitate the design of complex, secure, and trustworthy software systems. Such roots of trust will become the norm and a requirement for participating in trusted distributed computing systems. The security and reliability of the systems which we use will ultimately become the default and just like we are

able to prove functional properties of software, we should be able to specify, check, and guarantee security properties associated with certain tasks across different machines.

This is certainly a distant goal which requires many advances and perhaps failures in computer science and engineering. In theory, we must enable human and automatic agents to make informed and reasonable trust decisions. In practice, we can then build systems which *enforce* a set of default security trust policies *unconditionally* to ease adoption and integration of trusted systems.

Today, we get a glimpse of this vision in various cloud computing paradigms but with lacking security, privacy, and trust. The orchestration of virtualization in a cloud computing system has led to entirely new economic models and an efficient and more sustainable computing model for the future. Although virtual systems may benefit from managed security, better economy, and convenience, agents are required to surrender control and place trust in virtualization systems and their operators.

In contrast, trusted systems are systems running on trusted hardware which are sometimes only operated by a trusted party in a trusted location. We need to adapt and further develop trusted platforms and systems to at least modern standards without the loss of security qualities. I see the design and specification of a Virtual Trusted Platform as a natural next step for Trusted Computing and its proliferation. It may also prove to be the only way to provide trusted computing functionality to the scale of future systems. Today, a Virtual Trusted Platform is achievable and will continue to serve us well in the future as a flexible platform for integrating advances in systems, cryptography, and security use-cases.

1.4 Research Overview

Instead of starting over or using an approach which disregards an existing body of research, the thesis of this work is that we can construct trusted and secure virtual systems from existing trusted components. This thesis presents the design and analysis of systems which allow the establishment of trust and

enforcement of security policies in the components relevant to software and services running in a virtual environment.

Virtualization and the incorporation of a Virtual Trusted Platform poses an interesting issue for existing policies and compliance specifications as the trust placed originally only in hardware components (i.e., the trusted platform) needs to be extended to recording and reporting mechanisms and ultimately in the virtualization system and the Virtual Trusted Platform itself. While the hardware primitives and protocols for trust establishment are readily available, their semantics are ambiguous and a verifier has to decide whether a virtualization system or some of its components are trusted without much guidance, evidence, or support in reasoning for such a decision.

This leads to our first research item which presents an analysis of research, specifications, and recent proposals for a Virtual Trusted Platform architecture and trust establishment for software in virtual environments. We find that existing trust establishment approaches imply a particular topology, connectivity, and capability that does not reflect or accommodate a client verifying their software in their virtual environment. We outline requirements for a *User-centered Attestation System* in Chapter 3. Among the key requirements for a user-centered attestation system is the design of a *trustworthy* integrity measurement architecture which provides a chain of trust measurements for each virtual environment. The design has to ensure that we do not reveal information about neighbors in a multi-tenant system and it also needs to prevent potential attackers from hiding information from verifiers.

Our next research goal is the design and verification of an integrity measurement architecture which provides a chain of trust measurements between the (virtual) trusted platform and individual virtual environments. Proving that a trust establishment mechanism conforms to a security specification is difficult: commercially used virtualization systems are typically complex and not at all amenable to formal verification. Furthermore, current implementations offer hard- and software-based security mechanisms providing often very subtle but important security properties. For this reason, we introduce several formal abstractions summarized as a *Logic for Secure Stratified Systems* or LS^3 for trusted

computing systems in Chapter 4. Using our formal abstractions, we were able to design and verify the required enhanced integrity measurement architecture.

We included a virtual Trusted Platform Module (vTPM) as part of a Virtual Trusted Platform in the design of our integrity measurement architecture in Chapter 4. The choice to use a vTPM is uncontroversial and many cloud services include the option to pair a virtual machine with a vTPM to support some form of secure storage, trusted boot, disk encryption and so on for software in virtual environments. In our final research item (Chapter 5), we reveal that the association between Virtual Trusted Platform components (e.g., a virtual machine and a vTPM) is a critical concern and can not be solved by simply emulating the real machine—it is currently a matter of software and configurations. We are able show that by default a virtual machine paired with a vTPM may fall victim to an attack. A successful attack would trick a verifier into establishing trust in an untrusted virtualization system. We discuss several possible solutions for hardware and software systems as vTPMs gain traction in the industry.

This thesis has two main focus areas: (i) we aim to give formal abstractions for trusted systems and security properties to aid standardization, (ii) using our abstractions and formal analysis, we aim to design and verify novel trusted computing architectures towards a virtual trusted platform. The following research questions[1] detail our analysis further.

1. “How do we establish trust in a virtualization system and the services it hosts?"

   (a) How do we suitably abstract our target computer system and which basic properties do we expect from a trustworthy computer?

   (b) How should our virtualization system formally interact with trusted hardware?

[1]I would like to thank the Association for Computing Machinery’s Special Interest Group on Software Engineering (ACM SIGSOFT) for their stipend which allowed me to discuss and sharpen these research questions with selected laureates. I would also like to thank to Butler Lampson for discussing my questions on this occasion.

(c) How can we establish trust and security properties using both hardware and software abstractions?

2. "What measures can we take to increase our confidence that the virtualized environment is trustable and will uphold security guarantees?"

   (a) How do we record relevant information of a virtualization system and virtualized layers to establish desirable security properties?

   (b) How can we virtualize trusted hardware in a trustworthy way?

   (c) How do we practically report meaningful information of a virtualized system?

   (d) Which properties of a virtualization system can we remotely verify using trusted computing based recording and reporting architectures?

The remainder of this chapter briefly introduces our results and analyses in Sections 1.5 to 1.7 and summarizes the contributions of this research in Section 1.8.

1.5 Towards User-Centered Attestation

Cloud computing has become ubiquitous in a plethora of applications ranging from education, finance, and smart homes to healthcare, government, and military applications. Virtualization is omnipresent as the backbone of cloud computing offerings as well as *X-as-a-service* infrastructure and it continues to gain increased popularity even in end-user and embedded devices [9, 122, 16, 97].

We observe that the need for standards and specifications for secure and trusted collaboration becomes a pressing issue. Trusted Computing might be a practical solution which already enjoys widespread adoption and industry support. Furthermore, Trusted Computing is considered to be one of the pillars towards trusted and trustworthy systems both in terms of practical security mechanisms and supporting standards.

However, virtualization poses an interesting issue as the trust placed originally only in hardware components needs to be extended to reporting and

measurement mechanisms in upper layers. The extension has to happen in a trustworthy manner. Otherwise we would just declare arbitrary system layers and components as trusted. As we investigate current solutions and standards, we revisit the Trusted Computing tool-set and introduce its application to virtualization systems. We discuss challenges related to translating the predicate *trusted* between specifications for hardware modules such as the Trusted Platform Module (TPM) and specifications for operating systems, hypervisors, and virtual machines. We conclude that defining trust establishment strategies becomes crucial for specifications which extend trust beyond trusted hardware components. While approaches towards trust establishment exist, their semantics are ambiguous and an appraiser has to decide whether a virtualization platform or upper layers are trusted without much guidance or support in reasoning for such a decision. Furthermore, existing attestation approaches imply a particular topology, connectivity, and capability that does not reflect decentralized systems. When we discuss a virtual machine as a client-owned virtual environment, we need a trust establishment scheme which supports trust establishment in relevant components. Beyond privacy issues related to divulging information of other tenants, a client may simply not have the necessary knowledge base to establish trust in customized software running on a virtualization system and in virtual environments. Furthermore, a client may not have access to the virtualization system and instead only have access to a virtual machine or an application. A trust establishment mechanism must allow a bottom-up propagation of relevant information and the propagated information must be relevant to the virtual environment alone.

We conclude that we need a *user-centered* trust establishment method which aims at multi-tenant use-cases where tenants occupy virtual environments on a shared virtualization system (Figure 1.1). Such a novel recording and reporting (or attestation) system, is outlined to encompasses concerns we raise while evaluating current approaches. We propose a strategy for specifying and synthesizing suitable trust establishment mechanisms and hopefully inspire further research and contributions towards standards for open and collaborative trusted systems. We define User-centered attestation as a set of principles suitable for

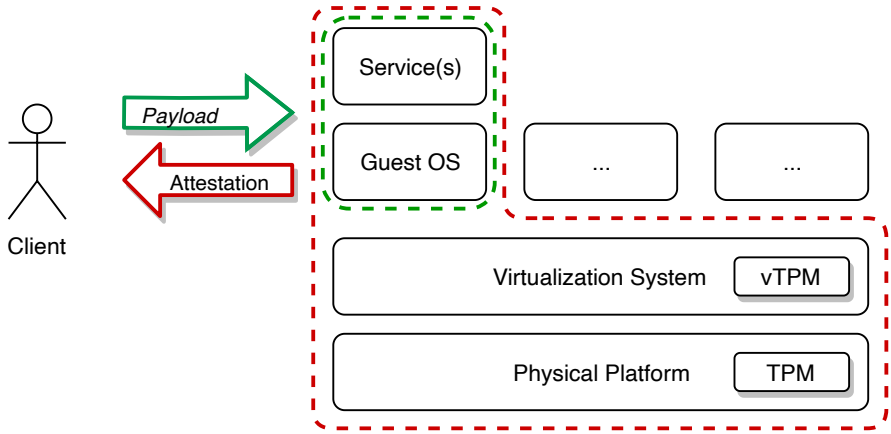

Fig. 1.1 A User-Centered Attestation mechanism must provide trust information about a client's software system in a virtual environment as well as the virtualization system.

layered, decentralized systems along with a methodology for specifying and synthesizing such a trust establishment strategy.

In Chapter 3, we introduce a set of principles for trust establishment using remote attestation and outline requirements for a truly user-centered attestation system. After reviewing existing approaches, we propose our own candidate for such an attestation system. Above all, we conclude that a user-centered attestation system must be able to provide a) user domain specific information and b) use a trustworthy mechanism to do so.

1.6 Domain Specific Integrity Measurements

Containerization is gaining increased popularity in a variety of settings and may be a key step towards future cloud computing models [70]. Containerization of applications removes the need to customize and configure an entire operating system in order to run a set of applications with their dependencies. Instead, an operating system is modified to include a support for containers, which supports and isolates containerized applications in their own virtual domain.

Since containerized applications share an operating system kernel, trust in a containerized application depends as much on the integrity of the host system as it depends on the hosted application itself. On this level of abstraction,

this relationship can be compared to current system virtualization software and guest operating systems running on virtual machines. We can infer the integrity of applications confined to their virtual user-spaces only by inferring the integrity of the operating system first,

For example, a virtualization service provider may host containerized applications and has Carol and Mallory as customers (or clients). Carol and Mallory both give a container with applications to the virtualization service provider, which promises to run them. As clients, Carol and Mallory *trust* the provider's setup to provide important security properties such as isolation both in the sense of performance and protection. Being able to establish trust in the operating system and provider infrastructure becomes crucial if the hosted service needs to be trusted. The provider needs a way of monitoring the virtualization system for compromises. Carol and Mallory want to monitor and establish trust in the virtualization system and the integrity of their containers. The previous Section 1.5 introduced this issue as the need for a *user-centered attestation*. User-centered attestation needs integrity measurements specific to the user environment. However, when reporting on the integrity of the system, the provider has to make sure that she provides *complete information* that is *constrained* to each container or domain. Otherwise, she might lose trust, or worse give Mallory an opportunity to launch targeted attacks against Carols applications. We conclude that there is a need to create a trustworthy recording mechanism which provides the necessary security properties.

Thus, in Chapter 4 we present the design and verification of a secure integrity measurement system for containerized systems (Figure 1.2). Containerized systems are complex and generally not directly amenable to formal verification. We provide formal abstractions for containerized systems by introducing LS^3, a formal model and logic with virtualization domain constructs to represent stratified systems and their interactions. Using our formal model, we show that the widely used Trusted Computing Group (TCG) based Integrity Measurement Architecture (IMA) [117] securely extends trust measurements from boot to applications. However, IMA is not designed to make domain specific trust measurements and is consequently incapable of creating domain specific integrity reports. In fact, the existing body of research on trust measurements

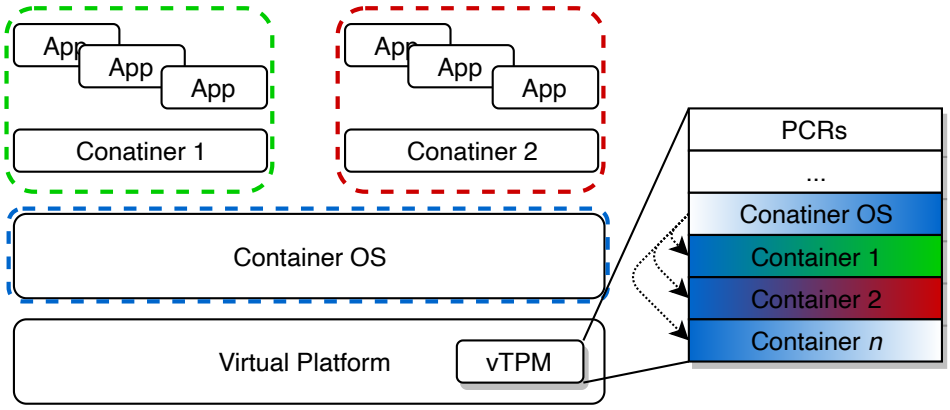

Fig. 1.2 An enhanced measurement architecture developed in Chapter 4 enables recording and reporting of integrity information for each virtual environment. The blue lines indicate the shared operating / virtualization system while red and green dashed lines indicate software in different virtual environments (or containers).

is focused on measurement comprehensiveness and performance. We propose a measurement architecture with constrained disclosure features built-in by making domain specific measurements to begin with. We argue that we preserve comprehensiveness and other desirable properties. To this end, we use the modular design of [36] and add to it support for virtual domains which yields our Logic for Secure Stratified Systems (LS^3). We conclude that providing domain specific integrity reports eases system and sub-system verification and yields desirable properties such as measurement log stability and constrained disclosure for multi-tenant systems. We verify and prove the correctness of our trust measurement architecture using our formal model.

1.7 Bootstrapping Trust in a "Trusted" Virtual Platform

In our last research item, we focus on establishing trust in a virtual environment (e.g., a virtual machine or container) using a virtual Trusted Platform Module (vTPM). In Chapters 3 and 4, we have already included a vTPM which together with a virtual machine serves as a virtual root of trust for various Trusted Computing protocols. The choice to use a vTPM is uncontroversial and has

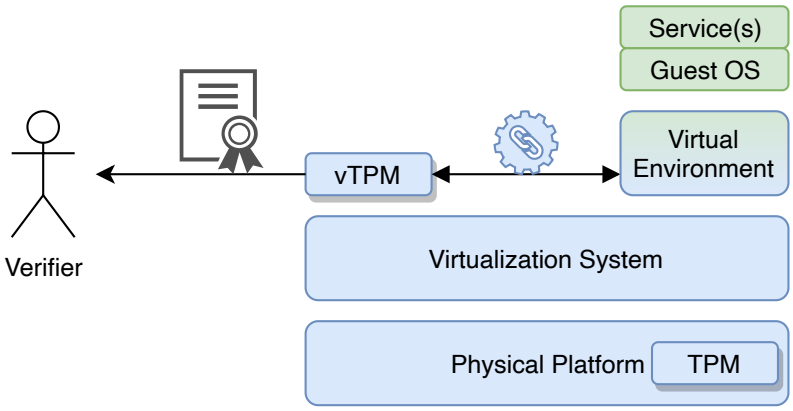

Fig. 1.3 Attestation of software in a virtual environment (e.g., guest operating systems or containerized applications) relies on a definitive association between the virtual TPM and the virtual environment. The association is an issue of the virtual trusted platform and can be exploited [82].

been proposed in [14] and adopted by cloud and virtualization system vendors like Microsoft, Google, and VMware. In fact, the TPM can not be shared across security domains [64] and approaches to properly virtualize the hardware module [40, 140] have not seen adoption [56, 66].

The virtual TPM [14, 56] may be software-defined and is used in a similar way to the physical TPM [64]: The TPM can be used to establish trust in the software running on a computer and the vTPM can be used to establish trust in the software running on a virtual machine. This is possible because the TPM is secure against software attackers on the computer and the vTPM is secure against an attacker in the virtual machine.

We were guided by the question: how can we establish trust in the virtual TPM? The question is legitimate and fundamental as the virtual TPM is supposed to function like a TPM but clearly does not have the same security properties. A good example is the association between a virtual machine and the virtual TPM. On a physical platform the association of the TPM with the rest of the platform is part of the hardware assumptions. Consequently, there is no analog which we can directly copy when we associate a vTPM with a virtual machine and the association is a pure software definition.

In [82], we investigate the vTPM and its association to a virtual machine (Figure 1.3). The specifications for “virtualized roots of trust” and “trusted

virtualized platforms" as well as the academic literature are not concerned with the issue of bootstrapping trust in a vTPM [66, 56, 14, 34, 120] and offer no help beyond requiring a strong association. However, the client interacting with a virtual machine typically has no direct access to the underlying hardware or the virtualization system—for good reasons. Realistically, a client will contact the virtual TPM first to learn about the virtual machine and use the vTPM as a proxy to the underlying TPM to learn about the software configuration of the virtualization system.

Based on this idea of bootstrapping trust in a virtual environment, we developed our *Goldeneye* attack and show that a virtual *trusted* platform may fall victim to it. We find that the association between virtual machines and vTPMs is not verifiable and without a way to assert that the virtual machine is in fact associated to a particular vTPM. Consequently, any verifier can be tricked into establishing trust in an untrusted system. In Chapter 5, we develop a formal model of a virtualization system and encode our trust establishment strategy. We demonstrate *Goldeneye* using our formal model and detail *why* it succeeds. Based on this, we evaluate existing work [14, 34, 120] and suggest improvements for virtualization systems which aspire to provide virtual trusted platforms for their clients.

1.8 Summary of Contributions

While we investigated trust establishment techniques for virtualized environments using the Trusted Platform Module, Virtual Trusted Platform, and associated protocols, this thesis makes several contributions:

1. An introduction and analysis of current research and standardization efforts towards trusted computing on virtual platforms and a proposal for a new user-centered trust establishment method (UCAS) in Chapter 3.

2. The design and verification of an enhanced integrity measurement architecture (EIMA) for user-centered attestation. To facilitate the design and verification of complex systems we introduce LS^3 as a formal system in Chapter 4.

3. A formal framework which captures functional components of a virtualization system in combination with a logical framework for the analysis of trust establishment in a virtual environments using vTPMs. We also present and demonstrate an attack on the trust establishment process called *Goldeneye* and discuss options for better trusted computing architectures in Chapter 5.

A more detailed list of contributions can be found in the introduction of each referenced chapter (Sections 3.1.1, 4.1.1 and 5.1.1).

The research questions (Section 1.4) which guided this thesis have been addressed in several ways: Our first contribution was to summarize, outline, and scope out major issues for trust establishment and attestation in virtual environments in Chapter 3. We reveal the need for user-centered and trustworthy integrity recordings and satisfy it with our system LS^3 in Chapter 4. As part of our second overall contribution, we fully address research questions 1a through 1c and research question 2a. Our third major contribution answers the remaining research items of question 2 by specifically investigating the (virtual) roots of trust of a virtualized system in Chapter 5 (research questions 2a to 2d).

The remainder of this thesis is structured as follows. Chapter 2 introduces and discusses alternative approaches towards secure and trusted cloud computing in as well as a summary of trusted computing concepts including the Virtual Trusted Platform in Section 2.3 and Section 2.4, respectively. The results and analyses of this thesis are presented in Chapters 3 to 5 which have published in [80, 83, 82], respectively. The thesis is discussed using the success criteria of a *modern* Virtualized Trusted Platform architecture (Section 2.4.2) and compared to related work in Chapter 6. The conclusions of this thesis are presented in Chapter 7.

Chapter 2

Background and Related Work

This chapter informally introduces the abstract concept of *trust* in computing and virtual environments in Section 2.1. Related work and approaches which have similar applications, goals, or security properties when compared to Trusted Computing are introduced and summarized in Section 2.2. Finally, Section 2.3 gives an introduction to relevant and auxiliary trusted computing concepts as well as Trusted Computing Group (TCG) components and protocols.

2.1 Why Trust Matters

Trust establishment has been the subject of a large body of work. Since humans began to involve computers in their calculations and decision processes, we asked ourselves whether or not we trust the results produced by any computer. Dijkstra went even further and demanded that ”even under the assumption of a flawlessly working machine, we should ask ourselves why we trust its results” [38]. Typically, the answer to such a questions has a number of dependencies and we struggle to enumerate them. There exits no “catch-all” solution which would allow us to answer ‘yes’ to Dijkstra’s question with great confidence. The research presented in this thesis is focused practical measures which can increase the confidence in the decision to trust a computer.

Increasing the confidence one can have in a result produced by a computer is fundamental to the field of trusted computing. Dijkstra introduces a mathematician and her reasonable distrust in a computer to highlight the issue:

Suppose a mathematician who is interested in number theory uses a computer with a program for factorizing numbers. The result of such a program is either the factorization of a given number or a statement that the number is a prime. For convenience, the mathematician uses the computer as much as possible but she also has to deal with problems which she could not solve without its aid.

The expected result of the factorization is clear: the given number is either a prime number or not. Assuming there are strong reasons to believe that the number is prime, the result of the program can either confirm this or give the factorization as proof that whatever intuition the mathematician had, has in fact fooled her.

The situation changes, however, if intuition suggests that a given number is not prime. The computer can still only produce two kinds of results. If the output is a factorization, the result itself serves as the proof that in this instance the computer's result can be trusted. However, if the computer's result is that the number is a prime, contrary to intuition or strong reasons to believe otherwise, why should the mathematician trust the result? In our modern world, we are constantly relying on computers which help with decisions or make them for us and we are increasingly forced to trust them or suffer consequences beyond just inconveniences. We should be able to trust computers with high confidence even though we have to doubt their results in lights of possible errors and malicious programs.

2.1.1 Trusting Computations

The example involving a mathematician and her computer is deliberately oversimplified but it introduces paradoxical aspects of outsourcing trusted computations. The kind of computations which are considered in this thesis are of course real applications and programs which form systems and do far more than producing relatively discrete results. Such applications and programs eventually

have to run on real computers using complex hardware machinery. Outsourced programs may be composed into application containers or shipped bundled with an operating system in virtual machines to provide a variety of services on a large number of systems and machines. When we run our programs, we expect not only functional correctness but also security properties. Such properties are often dependent on the computing environment which ensures that confidentiality, integrity, and availability expected by our application and its data is guaranteed.

The dependencies of trusted computing become quickly apparent when we change a hidden but important assumption in our example: the mathematician does not write all or any programs herself. This situation is already the 'status quo' of Thompson's "Reflections on Trusting Trust" [128] in the 1980s. Today, clients are busy with wrangling interfaces to common functionality and far removed or even deliberately prevented from programming or inspecting anything. Furthermore, we can not make a judgement based on a program alone and need to consider stacks of software right down to firmware and hardware components. The inherent complexity of computing systems today suggests that a binary trust decision, a definitive 'x is trusted', is inconceivably hard to justify. Lampson's "Note on Confinement" [77] elaborates on the idea of trusting a program in isolation and the practical implications and limitations. Trusted computing has an excess of dependencies both in production and operation of a computing system, which include hardware design and manufacturing, firmware code, operating system design, compilers, operating systems, program libraries and dependencies and sometimes other programs running on the same hardware. However, there are powerful tools and established concepts which help us to stay ahead of bugs and adversaries unconditionally [74, 143, 55]. On an architectural level, isolation between components and the ability to reset components into a trusted state help us to limit and reverse the impact which failures and adversaries can have on a system. Verifying that programs and systems implement certain functional and security related properties can be done offline with enough time and resources. In an operational scenario, computers only need to prove to clients that they run verified software which can be done using minuscule and verifiable roots of trust.

2.1.2 Trust in Virtual Environments

Developed in the 1960's with the humble idea to allow for "a computer to be used by two or more people, simultaneously", virtualization techniques have seen continuous development aiming to increase the efficiency of shared computer resources. This initial use-case lead to a major breakthrough in computing: the cost of providing computing capability dropped considerably and it became possible for organizations, and even individuals, to use a computer without actually owning one. Today, we enjoy much improved infrastructure surrounding cloud services, rapid development and deployment tools, and practical abstractions leading to a variety of new paradigms for distributed computing [9, 70]. We also gained the conclusion that cloud computing in particular is plagued by the fear of *dishonest* service providers and *untrusted* computer setups. In fact, after an initial spike, one of the major inhibitors for further adoption of cloud computing is a lack of security and trust.

This dateless conclusion coincides with the continuous incorporation of outsourced services in increasingly sensitive processes and decision making in security critical settings. In 2019, the United States department of defense (DoD) is expected to award a 10-year contract, known as Joint Enterprise Defense Infrastructure or *JEDI*, to an industry partner. The contract entails migrating a large amount of aging and expensive military infrastructure to cloud hosted services. If it was not obvious before, then at least today there is a quantifiable incentive for adversaries to withhold or misappropriate resources, modify tasks and data, or simply to spy on systems, programs, and data. Co-designing secure systems and trusted platforms makes it possible to detect, prevent, reverse, or downright preclude actions of an adversary but not unconditionally. For instance, it is notoriously hard to trust computers under the (physical) control of an adversary [105]. In general, adversaries in cloud systems are considered to be powerful. Related work (Section 2.2) often generalizes the adversary as the cloud service provider (CSP) which abstracts multiple parties and contractors [116]. This may be a good way to motivate and evaluate research and solutions which have fewer hidden assumptions or dependencies. For a majority of applications, regarding all parties including hardware owners as adversaries turns attempts

to secure cloud computing into an absurd exercise and forces solutions which negate any advantage a virtual environment may have. In reality, we should also consider that platform manufacturers and cloud providers have the greatest potential to run and maintain security on virtualization systems which provide secure and trustable virtual environments to a large number of clients.

2.1.3 Trust Model for Virtual Environments

The analyses and solutions in this thesis are contextualized in a *semi-trusted* model. The goal of using trusted computing in a *semi-trusted* model is to minimize the trust in all parties and systems involved, including the cloud provider and virtualization systems. The necessary assumptions, e.g. trusted parties and components, in such a model are accounted for and can be stated explicitly.

In the semi-trusted model, the cloud provider is trustworthy with regards to their motives and intentions such that one would want to be a customer or client. [120] summarize this neatly as "the cloud provider as an organization" is trusted. As outlined by [116], even if the provider's honest intentions are trusted, the intentions of insiders, contractors, administrators, or similarly privileged attackers have to be considered. The semi-trusted cloud provider is still susceptible to compromise by adversaries. Adversaries in a semi-trusted model can control a significant amount of the provider's infrastructure and there is no limit to the adversary's capability to compromise any specific part of the provider's infrastructure. This attacker assumption reflects an exploitable software component installed on a variety of severs or an administrator making changes with extensive privileges. To be precise, adversaries in this model can compromise software systems and obtain access to software and data running on any of the cloud provider's machines.

However, the adversary can not physically tamper with certain hardware components which include the CPU, bus systems, system memory, or the platform roots of trust (e.g., Trusted Platform Modules [64]). This has both a practical and a theoretical motivation: concerns of physical perimeter and

hardware security can be separated from the problem of designing secure systems supported by trusted computer platforms.

Aside from this threat model, several assumptions are necessary to support the work in this thesis. To facilitate solutions which involve recording and reporting software configurations, it must be assumed that there is a selection of trusted software. For instance, the security of software running in a virtual environment might rely on the functions and properties of a virtualization system. It is assumed that there is a selection of trusted virtualization software (e.g. operating system software or virtual machine managers). Such software might be trusted because of its origin or certification or because it is proven to be secure [74]. However, it is never assumed that only trusted software can be installed and executed even if the physical platform itself is trusted. Adversaries can still install arbitrary software configurations. Such software can be modified or entirely adversary supplied and range from high-level malicious applications to corrupted machine boot code.

Finally, Trusted Platform Modules have a special role as they implement a platform's roots of trust. It is assumed that a Trusted Platform Module (TPM) is genuine and comes from a trusted source. This implies that it has a trusted life cycle which starts from manufacturing and includes deployment and operation. Cryptographic functions and secure storage options implemented by a TPM are considered both trusted and secure against direct attack. The keys generated and used by the TPM exclusively cannot be extracted efficiently by the adversary and verifiers can establish a TPM's validity and a platform's identity using its unique and protected credentials [130].

2.1.4 Summary

The contributions of the remainder of this chapter are as follows: A brief overview of relevant background knowledge and related research is given along with the rationale behind choosing a Trusted Computing solution (Section 2.2). Subsequently, relevant concepts and primitives of Trusted Computing Systems are introduced and will serve as a reference throughout this thesis (Section 2.3).

2.2 Related Approaches

This section will review the related work of this thesis which is categorized into different approaches towards secure and trusted cloud computing. The first subsection reviews reputation-based approaches (Section 2.2.1) which is followed by a section on Trusted Execution Environments (TEEs) and Secure Enclaves in Section 2.2.2. Hardware anchored security and approaches using trusted execution features are summarized in Section 2.2.3 which is followed by formal methods based approaches to design and verify secure systems in Section 2.2.4. Lastly, approaches involving secure computations using homomorphic encryption are briefly discussed in Section 2.2.5. This section is concluded with a discussion of the individual approaches, their usefulness, applicability, and current limitations in Section 2.2.6. The conclusion of this analysis is that a trusted computing based solution must be combined with formal methods for the design of secure systems (Section 2.2.6).

2.2.1 Platform Identification and Reputation

A basic form of trust establishment depends on the ability to identify a subject and observe its behavior in certain interactions. Providing reliable temporary or permanent identification is crucial for all trust establishment strategies. Relying on some form of identification is the most basic way to establish trust and usually lacks direct evidence to justify it. In a perfect system, each entity has a cheap, anonymous, and yet unforgeable identity built in. In reality, solutions for secure identities vary between relatively unforgeable hardware based identities to highly privacy preserving identification mechanisms and tradeoffs have to be made. Whether or not an approach is feasible can depend on unit cost for hardware mechanisms and computational or infrastructure cost for more privacy preserving schemes [19]. Once an identity is established, services associated with that identity can be trusted if they have built up a good reputation [72]. Aside from such reputation based systems, the trust one has in a particular service can also be based on the service provider's identity with some offline information as context (i.e., contracts and agreements). Reputation can also be assigned to entire organizations or groups and device classes instead of

just individuals. Such approaches can help scaling up trust assessment when building up reputation between individuals is not possible. First time contact with subjects and opportunistic adversaries in particular are typical problems associated with purely identity and reputation based systems—especially when secret and confidential data has to be shared.

Kamvar et al. describe an algorithm named EigenTrust in [72]. The paper aims to reduce the amount of inauthentic downloads in the then popular *Gnutella* file sharing system by assigning trust values to each download peer and having each peer keep a record of its recorded trust values. The paper itself is one of the most cited papers concerning trust in peer-to-peer (P2P) systems as it presents an extensively evaluated algorithm for a variety of use-cases and attack scenarios. The algorithm itself is not concerned with a digital notion of trust and uses individual experience in a distributed system to establish peer reputations. The evaluation strategy and the adaption of a simple calculus have been influential in many extensions and further developments in the area. Liu et al. [85] outline the interdisciplinary nature of trust in peer-to-peer recommendation systems. One of the key contributions of this paper is a "non-binary" notion of trust. The non-binary notion of trust can be applied to the fact that trusting the entire computing system is often not realistic. The authors argue that trusting or not trusting heavily depends on the scope of such a decision and the parties as well as their competencies involved. The paper explains its semi-formal approach using examples from e-commerce and illustrates the application by applying it to the distribution and trustworthiness of identities in a PGP system. Richardson et al. [112] propose a trust management system for a *semantic web* based on trustworthy peers to improve the trust in content in such a network. While emphasizing the subjective nature of trust that is paired with a social notion of trust rather than a digital one, they propose storing local trust values of peers and composing them to global references for entities in their vision of the semantic web. They propose a model and calculus which enables creating and curating individual trust references as well as a strategy for merging trust values in case of cooperating peers. Ruan et al. [115] present a paper called "Achieving Fine-graded Cloud TCB Attestation with Reputation Systems" which leverages existing trusted computing strategies to establish

trust in cloud systems. The paper makes use of Trusted Computing Group technology, a defined trusted computing base, and trust establishment protocols. Their approach is based on mutual attestation using standard mechanisms while they suggest reputation records held by neighbors to overcome technical and theoretical limitations of their attestation model. This paper introduces reputation systems as a solution for hidden dependencies, architectural gaps, and lacking trusted computing support in typically complex cloud systems.

A promising approach which uses primarily identification is to create a composite of device identification and its software and security state. This concept is proposed and standardized as device identifier composition engine [131] and aims to provide a cryptographic identity which supports device attestation and data encryption. Such solutions are aimed at low-end devices with only a few hardware registers which can be used to store and update a device secret which is used to identify it. Based on this identity and cryptographic hashes created when bootstrapping the device, a cryptographic identity can be derived which allows tying the derived key to some software identity. However, important secrets may still leak, i.e. the device secret, and a corrupt bootstrapping process might also lead to incorrect key derivation. Such factors require solutions for emergency recovery and issuing new keys if old ones are considered compromised. Such functions might fall back on hardware-based secure environments to set up the device again [131].

2.2.2 Hardware-based Secure Environments

Hardware-based secure environments have traditionally existed only as standalone systems or as secure co-processors which could be used to store and execute a fixed set of programs which handled secrets and confidential data. Programs running in such environments are highly trusted if the program itself is trusted. Secure hardware execution environments are supposed to guarantee isolation from any untrusted application system. Some implementations of HSMs and co-processors even provide certified hardware tamper resistance. These add-on type security modules are typically fully independent computing systems with a high degree of isolation from the application system. Con-

sequently, all software attackers on the application system could not cross isolation boundaries and compromise data stored and processed in secure environments. Such solutions are typically expensive and suited only for special interest groups and organizations, i.e. in the financial sector, where they are used to securely generate, manage, and use cryptographic keys [14].

Another type of hardware-based secure environment uses application processor features to establish a Trusted Execution Environment (TEE) or a secure enclave [30] (Figure 2.1). Notable examples of TEEs and secure enclaves are

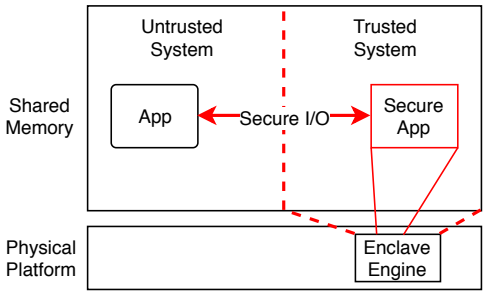

Fig. 2.1 Trusted Execution Environments and Secure Enclaves rely on a feature of the application processor (Enclave Engine) to establish, isolate, execute, and manage secure applications on shared hardware.

ARM TrustZone and Intel Software Guard Extensions. Compared to slower and expensive cards and co-processors, TEEs and secure enclaves especially incur some overhead when switching between untrusted and trusted executions but otherwise allow running trusted execution at application processor speeds.

TEEs use a number of isolation mechanisms to achieve their security goals. Common methods involve cheaper firmware functions for isolation, others provide dedicated hardware features for restricting access to trusted memory regions which may also be encrypted in main memory. Such hardware-based secure environments still share hardware with the application operating system as the isolation is a feature of the shared chipset. Sharing system resources between potential adversaries and the secure environment has already led to serious confidentiality breaches in unanticipated ways [84, 75] and the spillage of keys to certify the trusted execution of programs. On commodity platforms, the TEE and enclave capabilities are often very limited and allow only small

pieces of application logic and data to be fed into a trusted environment. However, some commercial solutions provide architectural support in hardware to run virtual machines or application containers in trusted execution environments [113] on mainframe systems. The approach of enshrouding applications and running them in (hardware) isolated environments is a promising commercial approach to secure cloud computing. Running trusted programs in secure environments on the application processor overcomes the typical high cost and bad performance of HSMs and slow co-processors. A co-design for isolation techniques which involves software, firmware, and hardware features provides an excellent platform for the contributions made in this thesis. Unfortunately, such co-designed solutions would require significant changes to commodity PC and mobile operating systems [100]. Truly secure enclaves and trusted execution environments could become an invaluable asset in an arms race against powerful attackers, if not all attackers, on the untrusted application system. However, TEE and secure enclaves are in many ways merely an engineering solution to often impractical and inefficient schemes for secure computing with some tradeoffs in security and hardware assumptions. Trusted execution environments and secure enclaves as a security feature do not immediately aid in the construction of secure systems. Instead, they provide an option for secure execution on an insecure or compromised operating system. The security and trustworthiness of programs running in TEEs and enclaves is still under investigation—adversaries can also run code their code in enclaves. If enclaves eventually run entire operating systems, we will likely see the return of exploits which hit commodity operating systems years ago.

Ironically, TEEs and secure enclaves today are already better known for their exploits than for the security features which they provide. Attacks which exploited micro-architectural and timing side-channels (or simply leaky hardware implementations) proved to be catastrophic and downright broke the security model. Attacks described in [84, 75, 33, 20, 51] allowed the extraction of enclave secrets and credentials from an untrusted part of the system. However, formal models for secure enclaves designs exist [30, 126, 106] and active research is underway to remedy current isolation flaws [59, 51, 17].

2.2.3 Hardware-anchored Security and Trust

Hardware-anchored security aids the construction of secure systems and supports trust establishment by using special hardware on the physical platform. Hardware security anchors first appeared as cryptographic co-processors which would provide support for cryptographic functions and local key management. Today, one of the most popular, standardized and commonly available hardware security anchors is the Trusted Platform Module (TPM) [64]. With the help of major hardware and software vendors, the TPM is well integrated into the majority of commodity platforms and operating systems. Similarly, there are practical, lightweight options for embedded and low-resource devices [131] as well as fully vendor specified systems on a chip (SoCs) such as secure enclave processors which support a variety of security related functionality. At a glance, the major difference in the design of hardware security anchors compared to hardware-based secure execution environments is that aside from a shared physical platform, e.g. a logic-board, the hardware security anchor is fully independent and isolated from the rest of the system. This includes main memory, caches, and CPUs. Secure and trusted execution environments aim to emulate or provide a virtually isolated environment while sharing hardware with potential adversaries. However, the degree of isolation depends on the implementation of the hardware security anchor (Figure 2.2). Unlike secure enclaves which

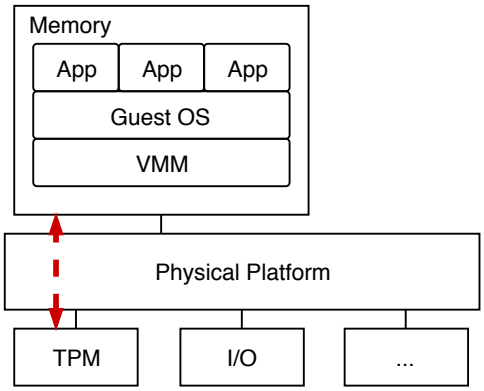

Fig. 2.2 Dedicated security chips such as the Trusted Platform Module (TPM) are available to the application system and aid in the construction of secure systems.

provide secure and attested execution, hardware anchors like the TPM do not offer similar features without software support. With such software support, the TPM can be used to establish trust in a virtualization system [105, 82]. The TPM provides a secure and trusted way of storing system states and provides cryptographic support for reporting states to a verifier.

Secure hardware anchors also support secure systems locally (i.e., without involving external verifiers). Today, secure boot technologies are commonplace and allow platform owners to specify which kind and version of an operating system should be running on a machine. If adversaries try to compromise the operating system or boot their own version, secrets such as disk encryption keys may not be released to them. Confidentiality provided by disk-encryption is supposed to prevent decryption of data by an insecure platform and operating system or an unauthorized client. Major research efforts have focused on another feature which is referred to as *late launch*, *trusted execution technologies (TXT)*, or *dynamic roots of trust for measurement (DRTM)* (more in Section 2.3.4). A CPU feature allows launching software in exclusivity while other processes are suspended. This allows potentially insecure and untrusted software to perform a system boot while the hardware and a small piece of trusted software launch a secure OS kernel. Solutions use this feature to provide protected execution of programs in an impoverished environment [89] or launch security monitors without having to place trust in an insecure operating system or hypervisor [88, 142]. Because of their isolation and ability to store system information in a secure way, hardware security anchors can be used to remotely verify a system. Using a procedure referred to as *remote attestation* allows a verifier to establish trust in a remote system [95, 87, 19, 121, 105, 28, 79]. Consequently, the verifier can then decide a security policy for the attested system, share data with it, or restrict the use of confidential data by crafting a policy which locks data to the security state of the system (see Section 2.3.3 for details).

2.2.4 Formal Verification for Trustworthy Systems

Trust in computing has been the subject of a large body of work. Practical approaches for systems verification are driven by proving functional correctness or compliance with a specification [74]. Eventually, functional correctness was extended to include more abstract security properties [99]. In parallel to that, formal models have been developed which formalized digital trust based on more social notions. Other approaches aim to formally prove a certain property (e.g., trustworthiness or cooperation) using game theory and formalizations of common dilemmas [76].

A study of formal semantics of trust and an argument for transitivity of trust is presented in [61]. The paper relies on a predominantly social notion of trust targeted at arbitrary cooperating systems. The paper attempts a formalization of trust based on business processes and builds a trust definition and simple trust inference rules for parties and actions. The vehicle for the argumentation is the *fluent calculus*, a form of a *situation calculus*, which models situations as states and actions which can be performed. The publication has gained some attention as an ontology and the formulation of theorems which can to some degree be translated to trust in a digital computer system.

Xiu et al. [139] propose a formal definition for trust in distributed systems and provide an extensive survey on trust in a social and computational sense. However, the model strives to define an *interpretation* of trust in a distributed system and is based on defined trust relations without regard to the actual functionality or computation in a distributed system. Furthermore, the notion of trust in this paper is binary, i.e. trust or no trust. A notion of this kind is prone to being overly crude or ill-informed.

Any notion of trust beyond an intuitive one depends heavily on the particular domain. Investigation of trust in a broader and generic sense may be of philosophical interest but it is debatable whether influential concepts in computer security have directly benefited from such research. Furthermore, developing generic concepts and formal models is a complex task. Without a target domain in the design of a model or method, such developments need

further time to model the target domain instead. Thus, more tailored models and methods are generally favored and produce useful results.

More domain specific models enable reasoning with an intuitive notion of trust in mind but with the target domain, i.e. a computing system, at the core of the formalism. The specific notion of what is and what isn't trusted or trustworthy is often encoded as a state of the model rather than being part of the model itself. As a result, the formalism can simulate real systems and one can formulate a number of trusted states and properties.

Orbaek et al.'s paper [104] introduces trust analysis in functional expressions by subjecting program data paths to checks so as to remedy problems related to trusted programs operating on untrusted input. For this purpose, the lambda calculus is extended with explicit trust operations and a trust-type system. The intention is that in order for results to be trustworthy not only the program must be trustworthy but also the input it receives. A programmer in this model will insert explicit checks whenever her program receives input so as to make sure that all data-paths have explicit trust states. The paper demonstrates of how a concept like trust can be explicitly incorporated at various levels in a language and how this can be used to insert a notion of trust directly into programs.

Rowe [114] defines a model and strategy for bundling evidence for layered attestation while introducing a use case for VM attestations using both bottom-up and program launch measurements [117] and dynamic attestation. The paper investigates bundling strategies and makes the case for attesting a system from the system root up (bottom-up). Until then, the notion that an attestation has to evaluate a system from the bottom up has been an intuitive one without much scientific backing. However, [114] deliberately conflates several related concepts in the paper and the theorems provided are based on a abstract model. The high level of abstraction and the deliberate confusion of capturing, reporting, and appraising evidence about a layered compute system make it a highly domain specific investigation with little possibilities to untangle conflated concepts or sharpen the model. Finally, [114] uses abstraction and mathematical reasoning to produce an informal argument which proved to be intuitive and not very useful for constructing new and checking old systems.

A more useful and likely better way to reason about and constructing systems is through the use of formal methods for designs. Formal methods for designs are used to understand a system before (or after) it is built.

Datta et al. [36] provide a system which allows encoding system states which are regarded as trustworthy using components which are trusted such as the trusted platform module (TPM). The paper extends a formal model designed for program verification with several trusted computing primitives, including a TPM, protected attestation keys, and special purpose secure registers. The inclusion of these elements in the formal system effectively allows the verification of trusted computing systems. Using only a few actions with side-effects such as read, write, jump and such, they were able to provide an abstract but credible view of programs and computer systems as they boot and run programs. [36] encodes a notion of trustworthiness by defining precisely in which state a machine is and which state is expected by a verifier. If the two states (i.e., actual and expected) match, a trust establishment strategy is considered trustworthy. This paper has been influential in the verification community even though the system itself is somewhat complex as it is overburdened with an obsolete formal system for protocol verification.

The authors of [36] later defined a logic for interface confined programs in [69]. The new logic is not concerned with networked computers and protocols anymore and instead focuses on reasoning about local safety properties only. The developed formal system allows modeling and proving safety properties of systems that execute untrusted code via interface confinement. Their system allows safety proofs for programs which execute untrusted code while the code itself does not have to be available for deep inspection. Our work in Chapter 4 builds upon [36] but removes the unnecessary and disputed network protocol reasoning capabilities [35, 31]. Instead, the focus is placed on an extended system interface which allows system layering with explicitly modeled actions. The work presented in [69] demonstrates that reasoning using interface confinement is a possible next step for the extension of [36] in Chapter 4.

A more focused vein of formal verification targets bug-free programming and programs. The effort of program verification is started by assigning a specific meaning to programs and introducing axioms and rules for reasoning

about individual program parts [60, 46]. Hoare logic, or Floyd-Hoare logic, is a formal system which allows us to reason about the correctness of computer programs. Hoare logic has been adopted and refined for various purposes. [60] introduced triples which had two assertions and a command—a pre and post condition as assertions and a command between them. If the precondition is met then reducing the command would establish the post-condition. Combined with logical rules for different cases allows the verification of currently dominant imperative programming languages using formulae of predicate logic. An alternative to writing programs and proving them correct is to write correct programs in the first place. Using Church's lambda calculus as the model for computation, simple type systems for programs were developed which proved to be equivalent to propositions of intuitionistic logic [53, 27, 137]. Today, we enjoy programming languages and type systems which take the correspondence between language and proof theory further to an isomorphism between programs and proofs.

A great example and respectable achievement for computer security research is the verification of the seL4 microkernel [74]. The verified seL4 project represents the successful application of program verification to an entire operating system kernel to be used as the foundation for trustworthy systems. The operating system kernel is not only proven to be stable and functionally safe but they are also able to prove strong isolation for the kernel itself as well as between applications. Similar properties are currently impossible to prove for commodity operating systems and remain an assumption which is based on best efforts rather than rigor. Klein et al. also report their approach and efforts: roughly 9000 lines of C code were verified to adhere to a high-level specification of their system. The overall size of the proofs about the system were 200.000 lines of code in a theorem prover. The time commitment to develop those proofs is estimated at twenty person years, depending on which items are factored in [74]. The authors note that in terms of practicality their effort was one of the more streamlined and less expensive ones. However, the details do not matter anymore when the verified lines of code are compared to commodity operating systems and virtualization systems which rely on millions of lines of privileged code.

In light of hardware based attacks launched from user-space, the verification of an operating system seems pointless when it is running on hardware which lets attackers bypass or corrupt a secure kernel. Efforts are made to produce verified hardware design for RISC processors which could add another piece to the foundation of verified, trustworthy systems. Despite those efforts, there is a strong case for formal modeling and formal methods for design: functional correctness alone does not imply that certain security goals are also met. Finally, a very practical question for verified systems arises from the work presented in this thesis: *how does a verifier learn that s/he is indeed working with a fully verified, trustworthy computer?* Trusted Computing and secure hardware anchors are perfectly suited as a system root of trust in this case. Secure hardware anchors such as the TPM have a symbiotic relationship with trustworthy operating systems and establishing trust in them is just one application [83].

2.2.5 Secure Computations

An unconditional solution to solve the problem of secure cloud computing would certainly involve having only encrypted data in the cloud. The data owner (e.g., a client) can upload only encrypted data and download it selectively for processing. Using a secure encryption scheme effectively prevents the cloud provider from accessing a client's data. For applications which involve database functionality, searchable encryption is being developed to allow the only somewhat trusted cloud service provider to perform operations on the client's database. The objective of searchable encryption is to prevent the cloud provider from learning about the data in the database, inputs, outputs, or operations [32]. Currently, the applicability of such schemes depends on a schemes scalability and the expected search complexity.

Another branch of cryptography research focuses on a more powerful approach: fully homomorphic encryption (FHE). Homomorphic encryption intends to overcome the limitation of having to operate on encrypted data locally (e.g., clients have to download and decrypt data) instead of directly in the cloud (e.g., the cloud computes using encrypted data and operations). Using homomorphic encryption, the cloud provider which holds only encrypted data

may perform specific operations on the encrypted data. However, homomorphic encryption is only possible with certain operations. In comparison, fully homomorphic encryption supports arbitrary computations on encrypted data [52]. In either case, it is important to note that both inputs and outputs of such schemes are encrypted. The promise of fully homomorphic encryption is immense and would partially solve the issue of processing encrypted data with weak trust models and powerful attackers (Section 2.1.3).

But even FHE schemes are an unlikely path towards secure cloud computing. The perhaps most obvious problem is that the output of a computation is encrypted. The operational consequence is that no actions can be performed on the result of a program in the cloud which makes such operations hard to integrate into systems. The result of a computation needs to be given back to the client for decryption before next steps can be executed. If the result of the computation was deterministic in some way, i.e., no longer seemingly random without the key and therefore usable in the cloud, the encryption would not provide the desired security property anymore and the cloud needs to be somewhat trusted.

A second major topic for FHE research is stopping or limiting its inefficiency in both time and space. A seizable amount of research has already produced speed-up by several orders of magnitude and much smaller key sizes [116, 25, 41]. Despite such improvements, the current state of fully homomorphic encryption ultimately limits the approach to certain key applications [116].

2.2.6 Discussion

The five approaches we have outlined differ significantly in the way they approach the problem of secure and trusted cloud computing, in the security properties which they provide, and most importantly in their range of applications.

Reputation based systems are clearly most applicable as they adopt an almost anthropological approach to security. It is easy to imagine that one might simply use a reputable cloud provider and trust that combined with legislation, preserving an excellent reputation by providing secure environments for clients

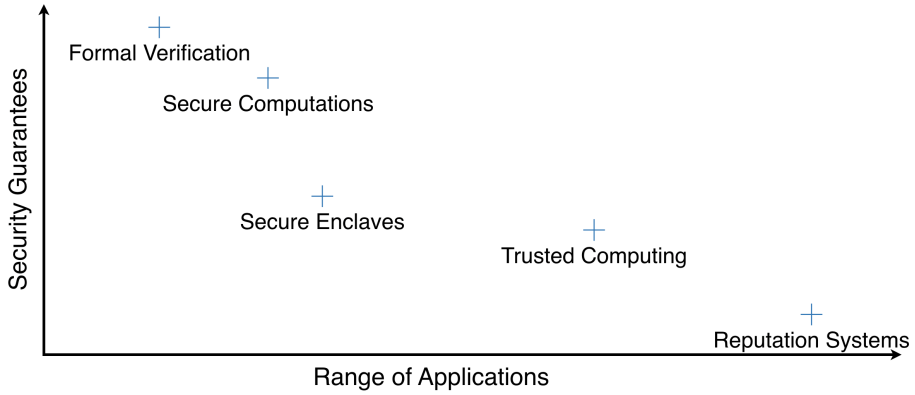

Fig. 2.3 A pictorial presentation of the security guarantees which each approach provides on the y-axis and the range of applications in our cloud computing setting on the x-axis.

is simply *good enough*. This is the status quo and demonstrably not enough to establish trust and protect data in a cloud service. Surprisingly, despite needing no technical adjustments, reputation systems do not scale even when they are supplemented by legislation. Cloud services often have more than one stakeholder, more than one administrator, and more than one software-provider. Failure in any one can lead to compromise. Without clear capabilities, responsibilities, and extensive logging it becomes hard to blame the responsible party. Besides making it easy to miss the security goal, the *low-tech* approach of reputation systems makes it difficult to determine who is to blame and lose reputation.

Cryptographically secure computations using fully homomorphic schemes are on the other end of the spectrum of approaches. Among the outlined solutions, they arguably provide the strongest security guarantees and have little to no dependencies on an actual cloud system (perhaps resources for availability). These security guarantees come at a significant cost both in terms of practicality and operation. Currently, all secure fully homomorphic schemes incur a comparatively large overhead in terms of computing power and storage. If such schemes were deployed widely, they would drive up cost and limit the usefulness of cloud services severely. Today, fully homomorphic encryption allows for confidential outsourced computing but offers no way to integrate

outsourced computations into classical and flexible service architectures. The result of fully homomorphic schemes is necessarily encrypted and any trade-offs one could make to allow further processing and integration in the cloud can easily lead to compromise which removes the strong but brittle security guarantees [116].

Hardware-based secure environment offer a practical solution for confidential and secure computations for cloud computing. Today, secure environments are available on commodity hardware and can provide a way to perform sensitive operations even when the operating system or the system owner is considered a threat. However, commodity CPU's do not offer enough secure enclaves and space to accommodate multiple clients on one machine and hardware enclaves are not yet supported in virtualized environments. Computing with enclaves in general needs more support both in terms of security and interoperability with commodity operating systems which will eventually arrive. Most programs will need to be adapted or rewritten to run certain parts of, e.g. processing confidential data, in isolation. The security guarantees heavily depend on the way enclaves are programmed and utilized [136]. Micro-architectural attacks [20, 51] demonstrate that current hardware-based secure environments are far less independent and isolated from the host operating system than promised. While we are yet to see public disclosure of successful exploits, the initially clear security benefit of hardware-based secure environments is now put in question. Without more verifiable designs secure environments can be seen as an exploitable black box on a remote system which can be strategically used against its clients.

Despite these current shortcomings, more secure enclave and system designs are bound to emerge, eventually. Support for enclaves in virtual environments would provide a way of excluding other tenants, virtual machine management software, and operators/administrators from a threat model. TEE's and secure enclaves have great potential and are already aiding the construction of secure systems by providing alternative and secure run-time environments for critical programs. Interestingly, secure enclaves and execution environments rely on a number of concepts adopted from hardware-anchored security. The results of

this thesis might in fact be applicable to the design of enclave features such as *remote attestation*.

Formal verification is currently the best known tool to construct a trustworthy system. Formal verification has evolved into a useful and in many cases mandatory approach [101]. The usual complaints about verification tasks is that the conclusions reached are not worth the time invested. However, computer security and especially the impact of security on the overall system safety might motivate extensive formal modeling and verification in some cases. Full formal verification produces proofs for software using a specification which is very close to the machine level but at great cost [74]. Full verification for functional correctness using a low-level specification is simply not an option for this thesis.

Summary

Producing a specification for a secure system is a mandatory task no matter how complex the system may appear. Without at least an informal method to capture and discuss a system and its properties, we can not determine or improve its security characteristics. The same rich techniques which are applied to a low-level specification can obviously be applied to a higher-level specification and with great success. Furthermore, the kinds of systems which we consider are generally large and typically receive gradual improvements. Producing formal proofs against a low-level specification might simply not yield great insights. However, significant effort must be directed towards providing proper abstractions. With good abstraction, even a high-level abstract state is effectively real (i.e., abstract states correspond to real states). Modeling an existing system and designing extensions based on a useful formal model is a proven way to approximate complex systems, separate concerns, and allow for incremental and compositional improvements of systems [101]. This also summarizes the approach to formal verification of this thesis.

As hinted previously, this thesis also focuses on establishing security and trust in virtualized environments using a Trusted Platform Module (TPM) as a secure hardware anchor for trustworthy systems. In short, the motivation for a trusted computing approach are: (i) native applications and infrastructure

runs without overhead on application processors, (ii) (remote) establishment of software system properties, and (iii) all trust is rooted in and extended from secure hardware. A trusted computing approach excels at leveraging existing infrastructure and benefits directly from improvements to hardware and software designs. As a system's root of trust, hardware security anchors are irreplaceable which can serve as an integrator component for the previously mentioned approaches as they become more applicable.

2.3 Trusted Computing

The concept of *Trusted Computing* is intended to protect systems and data from all software and some hardware attacks [95, 105, 87]. The trusted computing approach used in this thesis is based on the work of an industry consortium, i.e., the Trusted Computing Group (TCG), which promotes secure hardware, systems, and protocol designs for trusted systems [132]. TCG designs and concepts are based on a large body of research and development and their approach is highly applicable to today's hardware and software systems. Using TCG hard- and software is now a feasible approach to securing commodity systems and constructing trusted platforms for future systems. In this section, a brief introduction to Trusted Computing as defined by the Trusted Computing Group is given. The informal term 'trust' and its semantics are introduced in Section 2.3.1 and frequently used in the introduction to *Trusted Computing Platforms* in Section 2.3.2 and later sections. The most notable addition to computing platforms by TCG is the Trusted Platform Module (TPM). The TPM's properties, capabilities, and design limits are introduced in Section 2.3.3. Large parts of this thesis are concerned with integrity measurement and protection techniques. The TPM supports system integrity measurement and protection architectures by acting as a secure compartment for measurements and reference values. How exactly the TPM supports the work in this thesis and how integrity and other trust information can be used on a trusted platform is summarized in Sections 2.3.4 and 2.3.5, respectively. Finally, the general challenge of applying these concepts to virtual environments using a virtual trusted platform is dis-

cussed in Section 2.4. Virtual trusted platforms include a virtual TPM [14] to act as a TPM in virtual environments—the concept is reviewed in Section 2.4.1.

2.3.1 Trust in Trusted Computing

When we refer to something as trusted, we strictly mean that it can or must be trusted in order for some other, higher-level, and dependent concept to work. A good example for this is the definition of the Trusted Computing Base (TCB) in [135]. A common definition says that the trusted computing base is the part of the system which we essentially rely upon for security and a failure in the TCB may allow adversaries to attack the entire system [87, 95]. Similarly, today, trusted systems are the kind of systems which we rely upon in different ways and if they fail then it may negatively affect concepts built on top of that. Trust does not necessarily entail security. Instead, secure systems may be built from or on top of trusted components. Conversely, we would not attempt to build a secure system while relying on untrusted and untrustworthy components. We can trust a component if (1) we can identify it, if (2) it can operate freely, and if (3) it always operates as specified [87, 95, 48]. The first two points are intuitively important since without identification we would not be able to tell trusted components apart from untrusted or unknown ones. Additionally, trusted components must be able to operate as specified such that adversaries are not able to block important mechanisms. The last point can be achieved by formal modeling, specification, and verification which essentially helps to prove that the verification target (e.g., a piece of software or hardware) implements a specification. Once components are verified to conform to their specification, they can be referred to as trustworthy [87, 74]. For (software) systems without precise specifications, we often have to look for more practical methods such as fuzzing [127] or accept fewer guarantees by relying on testing instead. Trusted Computing requires a combination of trusted and trustworthy components to enable trustable computing, i.e. computing systems which we *can* rely on.

2.3.2 Trusted Computing Platforms

The *Trusted Computing Platform*, or *trusted platform*, has been a long standing and elusive target for several high-assurance projects [95, 48]. Approaches aiming to construct it often differ greatly based on the particular computing application and the available hardware. However, all trusted platforms must be identifiable and their current configuration must be known [87]. What the relevant configuration is varies again with the kind of system and the kind of application. Typically, the relevant configuration encompasses all components of a system which are not de facto "trusted" but need to be trusted. A simple dynamic emerges which suggests that if out of a total number of components only a few are trusted and the remaining components are part of the relevant configuration unless they can be excluded (e.g., because they are effectively isolated). Platform identity is commonly achieved by giving a machine its own digital fingerprint. Protecting the unique identification or restricting use of the identifier is a task often left to the platform manufacturer. We have to trust that the identity is both unique and *bound* to the platform. Other approaches might establish an identity for a platform during its operation based on some unique property, behavior, or structure of the system instead. If the identity is added to an otherwise anonymous platform, then we rely on a strong binding between the identity and the platform.

Securely conveying the relevant configuration typically leverages the platform identity for authentication and to constrain the disclosure to parties familiar with the platform. This requires that any dynamic configuration (i.e., one that can not reliably be predicted) is appropriately established and recorded locally on the trusted platform and reported if necessary. If the configuration of a system is static, pre-defined, or predictable it may also be implicitly established (i.e., without any further recording or reporting). On a PC system, relevant configurations may include hardware, firmware, and software as well as its current configuration. Configurations can be established via measurements of components by other *trusted* components. Measurements are typically produced using a cryptographic hash function with some object (e.g., a system component) as its input. The output of that function is then referred to as a local

measurement which can be compared against some reference value (sometimes referred to as 'golden value'). The system component which performs, stores, and reports measurements needs to be trusted if we want to establish trust in a platform.

2.3.3 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is a component designed by the Trusted Computing Group (TCG) in a set of specifications [79, 57, 64]. The specifications can be implemented as hardware, firmware, or software devices which support secure systems and security enforcement in several ways. A TPM can generate strong keys using true random numbers and use them in cryptographic operations, it can certify keys and other objects using its own key infrastructure, and it can make keys and certificates available to a system based on a set of policies. A TPM's root keys can never leave the TPM (or be used directly by an external party) and some of the keys it generates can be restricted to be used by their creator or by a set of specified TPMs only. Thus, TPM held keys are protected from a variety of attacks and should be protected from all software attackers ([96] reveals the implication of timing attacks on vulnerable implementations).

Beyond such cryptographic primitives for generating and using keys, the TPM also supports a variety of secure storage options by using internal keys to protect externally stored objects or storing objects internally in non-volatile memory regions and in Platform Configuration Registers (PCRs).

Conceptually, a trusted computer equipped with a TPM (and support) can be referred to as a *trusted platform* (or just *a platform* in TCG-speak) for building secure and trusted computing systems. To aid the construction of trusted computer systems on a platform, the TPM provides several so called *roots of trust*. Roots of trust are system components which need to be trusted, possibly without a way to directly verify them, but they are also meant to be trustable through assurances of several parties including the platform manufacturer. The TPM is intended to provide three essential roots of trust [87, 64]:

- **Root of Trust for Measurement (RTM)**: produces the initial integrity measurement and continue a chain of trust measurements on the platform.

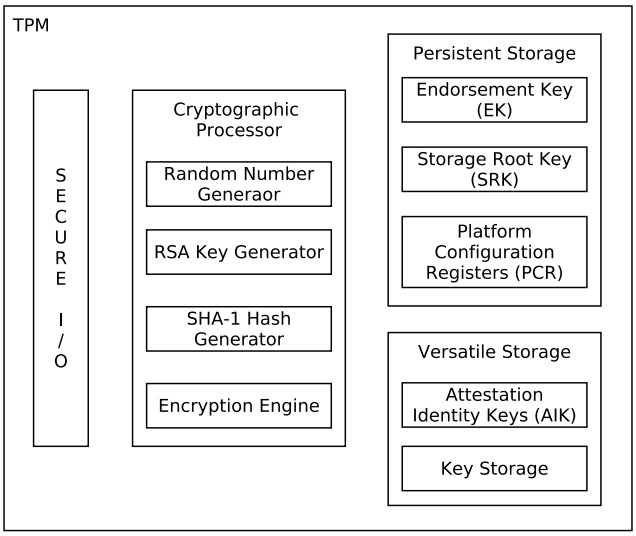

Fig. 2.4 A Trusted Platform Module (TPM) and its internals [79].

- **Root of Trust for Storage (RTS)**: protects secrets on the platform such as storage keys and other objects (e.g., integrity measurements).
- **Root of Trust for Reporting (RTR)**: provides a unique identity for the trusted platform which can be used to certify keys or report trust information (e.g., via attestation).

The interaction and need for these roots of trust is best explained though a challenge such as: "is the system on my platform in a good state?". To answer such a question, we need a TPM's roots of trust. The RTM creates cryptographic hashes of a loaded system object (e.g., an operating system kernel) and stores it in a Platform Configuration Register (PCR) (2.4). The PCR acts as a secure storage location for storing measurements. Now a trustable mechanism derived from the RTM is needed to take measurements and record them in a PCR (i.e., storing them in a secure location). After the measurement has been taken and stored in a PCR, the system state is now seen as *recorded* securely in the TPM. Assuming that a platform's identity is known in the form of a public key [105], the RTR can now certify the recorded measurement which can then

be conveyed to a verifier. This process is also known as *remote attestation* and adds a genuinely new security primitive to a trusted platform.

Another challenge would be to ask whether some secret can be kept by a particular platform. The RTR can certify that some storage key (secured by the RTS) is used to protect secrets on the platform. A verifier can now encrypt a secret using a key under the protection of the RTS to store confidential data securely on the platform. With support by system measurements (RTM) or verifier tokens, secrets can also be restricted to trusted system states. The roots of trust implemented by the TPM support enhanced and trusted features on the platform. These newly added features have both local and remote use-cases (more details in Section 2.3.5). However, the system state itself is always recorded via measurements and continuing a trustworthy chain of so called trust measurements is a major topic in Chapter 4 and the next section.

2.3.4 Integrity Measurements

A TPM can serve as a *root of trust for measurement* (RTM) and provides both a command to record a value using a cryptographic hash function as well as the hash function itself with its own cryptography suite. However, a TPM can not actively *measure* system components as a TPM has no visibility or capabilities outside of its boundaries. Consequently, so called measurement events have to be triggered outside TPM by the untrusted system while the TPM can *support* taking measurements. If one wants to record and report the system state, the TPM alone is not suitable.

Although the TPM provides some RTM functions (e.g., a hash generation and storage), in this thesis the RTM is treated as a part of the *trusted platform*. The trusted platform part of the RTM is referred to as the *Core Root of Trust for Measurement* (CRTM) which is intended as the *zeroth* component in a boot sequence before any actual boot code is executed. Although it is intended as a standalone, immutable, and trusted component, in practice the CRTM is often implemented by the few first boot instructions (e.g., ROM boot code) which perform a measurement of the current and succeeding component. The CRTM is a critical part of the platform's integrity measurement capability. The purpose

of the CRTM is to initiate a chain of trust measurements which effectively records the software system that executes on the trusted platform.

One way to record the systems is to use a *static root of trust for measurement* (SRTM) approach. Under the SRTM paradigm, a chain of trust measurements is created which starts with the CRTM and is extended all the way up to the operating system. The process of measuring and launching the next component in a boot sequence is depicted in Figure 2.5.

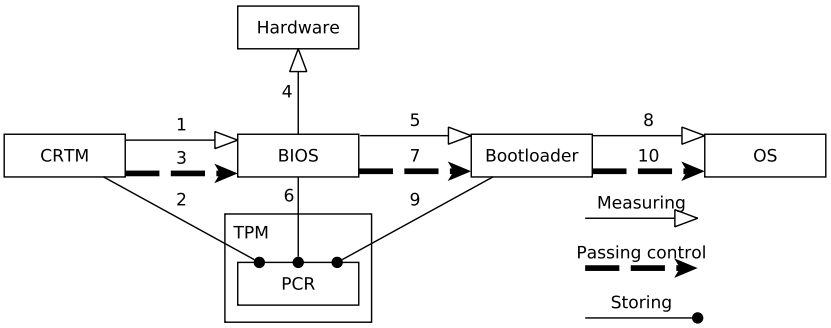

Fig. 2.5 A Static Root of Trust for Measurement (SRTM) boot sequence creates a chain of trust measurements from BIOS/CRTM to the OS [79].

The procedure in Figure 2.5 is intended to be straightforward and requires extra instructions in each component to create and record a hash of the successor component before jumping to it [36, 83]. In theory, this creates a simple sequence of measurements that is cryptographically chained and stored in the TPM for a verifier to review. Constructing and verifying a measured boot sequence is a major issue of [36, 83] and is discussed in greater detail in Chapter 4.

However, SRTM launch sequences are somewhat controversial and may lead to dangerous confusions. The CRTM in the SRTM launch sequence initiates a sequence of trust measurements. The continuation of this sequence and whether measurements are trustworthy depend on each successor of the CRTM. If adversarial code is loaded at any point in this sequence, the remainder of the trust measurements is, by definition, not trustworthy anymore. The scheme displayed in Figure 2.5 produces a chain of trust measurements in the TPM but at no point is trust ever placed in or delegated to the next component—a

verifier has to inspect each element and establish trust. Deciding whether or not there is a *chain of trust* from CRTM to the OS depends on the trustworthiness of each measurement and the trustworthiness of the individual components. Since the entire sequence of measurements has to be verified, ambiguity (i.e., a component is unknown or untrusted) or code that is executed but not measured for any element implies that successive measurements are not to be trusted. This leads to three important conclusions:

- A sequence of trust measurements does not necessarily imply a *chain of trust* between all measured components.
- Components in the sequence may not be able to statically determine or measure all components contributing to a system boot which gives adversaries the advantage to hide code from verifiers.
- Verification based on SRTM is vulnerable to time-of-check to time-of-use (ToCToU) attacks without extra assumptions or system guarantees since components need to be measured before they are executed [36, 83].

A *dynamic root of trust for measurement* (DRTM) presents an alternative to SRTM and remedies some of its shortcomings [89]. DRTM is supported by a special hardware/CPU instruction rather than "trusted" boot code. Examples of DRTM implementations are Intel's Trusted Execution Technologies (TXT) and AMD's Secure Virtual Machines (SVM). Once a DRTM system launch is triggered, the platform is put into a special, trusted mode. In this mode the component to be launched next (i.e., a piece of code) is loaded in a deprived but trusted platform mode (excluding adversaries) and recorded by the platform rather than some predecessor component. As a result, the chain of trust measurements in the TPM is much simpler and a piece of code is executed securely. DRTM was designed for securely launching a trustworthy operating system or hypervisor which then takes control of the platform. Consequently, trust is delegated from the platform directly to the trustworthy operating system or virtual machine manager.

Applications and virtual machines heavily depend on the operating system, hypervisor, and sometimes other applications or virtual machines. To continue

either measurement chains, operating systems have been extended with measurement architectures such as *Integrity Measurement Architecture* (IMA) [117]. Using the operating system to produce measurements instead of using DRTM for applications and virtual machines is considered more applicable during system operation [14, 79, 83]. A major concern when using DRTM is contention over the hardware/CPU and the overall inconvenience associated with using DRTM frequently: putting the platform into a trusted state results in a massive disruption of commodity systems and prevents all concurrent execution (even on other CPUs) until the launch sequence is completed. Because of this lack in support for DRTM-like approaches on commodity hardware and systems, this thesis focusses on integrity measurement architectures for commodity systems.

2.3.5 Using Trust Information

Trust information (e.g. integrity measurements) supplied by a trusted platform about the currently running system is used in a variety of applications. The two principal options can be described as *local* and *remote* uses of trust information. Combining them often yields considerable security benefits.

Device ID

A key usage of trusted platforms is the certification and provisioning of unique device or platform identification (device ID). As outlined in Section 2.2, a reliable identity is paramount in reputation-based systems and solutions which mix device identity with software configurations [131]. A reliable way to identify a device is the basis for trusting its configuration, actions or building up a knowledge-base of its behavior or compliance record. The device ID functionality is supported largely using the assumption that a TPM is affixed to a trusted platform and practically inseparable within any reasonable amount of time. The device identity can then be provisioned to and protected by the TPM. The TPM has built-in functionality which allow a device to derive privacy preserving identifiers for use in direct anonymous attestation [19, 24]. Devices with similar capabilities, such as HSMs, are more expensive by several orders of magnitude, removable, and completely unsuitable for certain device classes and sizes. The

hardware-based security, cryptography capability, and secure storage afforded by the TPM can reduce the risk of malicious device cloning, can significantly improve supply-chain security, and is the basis for device enrollment and failure recovery.

Using Trust Information Locally

A way of using trust information locally is to protect confidential data on the platform itself. Using the RTM for measured, trusted, or secure boot sequences, cryptographic keys and other secrets can be *sealed* or *bound* [64] to a platform security state. Sealing or binding of secrets usually results in a policy which is enforced by the TPM and defines under which condition(s) a secret may be released or used. A now common application for this technique which adds an immediate value to a system's security is implemented in storage encryption solutions. Although single files can be encrypted and effectively tied to a platform's security state, a more common application is to protect boot and operating system sectors and a user's file directories. For example, Microsoft's *Bitlocker* [90] aims to protect data in case of lost or stolen devices and corrupted operating systems [87]. This approach relies on a modified operating system which ensures that relevant data is encrypted while the device is shut down or the user is logged out. The encryption key is protected by the TPM and released only if the security policy (enforced by the TPM) allows it. Typically, the same can be achieved using a user token such as a password, USB-token, or a smart-card. However, with a TPM a selection of encryption keys can be securely generated, stored, and ultimately protected on the platform. Furthermore, TPM policies are typically enriched by additional trust information such as integrity measurements (e.g., of the OS, patches, or applications) which allows using both user tokens and a platform security state to protect secrets which prevents users from unintentionally booting an operating system that can not uphold necessary security guarantees. With techniques similar to [117], this can be extended to OS-extensions and applications as well.

Alternatively, entire run-times and system environments such as virtual machines and containers can be tied to a trusted platform and its security state.

By crafting security policies, users can define conditions under which their workload may be decrypted and run on a trusted platform [116]. This approach essentially enables the trusted computing solution outlined in the introduction and allows users to specify that certain operating system versions (e.g., outdated or vulnerable ones) are unsuitable or that unknown or blacklisted applications may not execute on the system at the same time.

Lastly, TCG also pursues approaches which protect the integrity of the trusted platform itself. More specifically, TCG's storage work group describes how entire storage devices can be encrypted at the device controller [87, 132]. In such a case, the trusted platform is treated as a composite of smaller distributed systems which must be bound together. The device runs a key exchange protocol with the TPM and its unique identity and derives a storage encryption key. If the device is detached from the trusted platform, an adversary can no longer obtain data without breaking the security of the trusted platform or the TPM. A similar approach is extended to other *removable* components such as chips containing important code (e.g., trusted boot code) and even CPU's which often hold secrets and security functionality that must not be replaced by an attacker.

In short, trust information can be used locally to enforce security and trust on the trusted platform.

Using Trust Information Remotely

A popular and well standardized application for trusted platforms is to connect them into networks of trusted platforms using Trusted Network Connect / Communications (TCG TNC) [87, 133]. With increasing connectivity and spacial distribution of computing devices, managing a device's access to certain networks becomes critical. This situation is compounded by the fact that access to a certain network is often used as a scalable access control mechanism. Good examples are computing and storage resources which are shared among a certain group of users and their machines. For example, access to the resource via public networks might be limited or blocked while access via a the correct network might give the machine read or write permissions [87].

Such approaches are deeply flawed, of course, if an adversary can access these networks by introducing a corrupted machine or by infecting one that is already connected. For this reason, network access and privilege in a trusted network are granted based on the security state (e.g., identifier, type, software configuration, or owner token) of the machine. Before the connection is allowed, the device has to provide trust information based on which its security state is assessed. The inclusion of 'endpoint security information' is a valuable addition to network security policies which still use easily cloned authentication tokens such as MAC addresses of removable network cards [87, 95, 48].

The TPM as a root of trust is perfectly suited for this task by supporting a procedure referred to as *remote attestation*. Remote attestation requires the target of the attestation to send trust information to a remote verifier. The conveyed information must attest to the security state of the device which the verifier may certify. Before a state is certified, a verifier may need to be convinced that the trusted platform's PCR values actually reflect and correspond to the system state — this is not a trivial task and not something that the TPM can do. The system's measurement architecture [117, 83] primarily determines how accurately trust measurements correspond to a system's state. Proving such a correspondence is a major topic of Chapter 4.

To prove the authenticity of trust information, the TPM implements a *quote* command. Quotes allow the TPM to certify relevant trust information in secure storage locations. The final building block of remote attestation assures freshness of the conveyed trust information. To assure freshness and prevent replay-attacks or simple re-use of quotes, a challenge and response protocol between verifier and the platform must include a verifier supplied one-time random nonce (or just *Nonce*) which a TPM includes in a quote. The key used for quoting a value is typically derived from the RTR and protected under the RTS. The key is assumed to be certified by a trusted party or simply known to the verifier. By using asymmetric cryptography initially, session keys and other short lived secrets with usage policies can then be given to the TPM for future interactions. Key usage policies are enforced by the TPM and allow simplified and more efficient challenge and response schemes. The use of crafted policies with secrets that are enforced and protected by the TPM demonstrates how

the remote use of trust information can lead to more efficient local usages of trust information. The interaction between local and remote uses of trust information enables the efficient and scalable management of trusted platforms and networks.

2.4 Virtual Trusted Platform

System virtualization is a popular way to manage both server and desktop environments. A virtual machine (VM) which can be migrated and copied enables efficient resource utilization, increased resilience, and security for software running in virtual environments. VMs can be configured by clients and uploaded to cloud providers which run client workloads in clusters of virtual machines and storage locations. The first wake of cloud computing's success was powered by system virtualization and VMs are still the principal way for many organizations to move expensive and locally managed data centers to scalable and outsourced virtual infrastructure.

While system virtualization might help scaling entire systems, operating system level virtualization technologies such as *containers* aim to provide virtual environments for highly portable applications. Operating system level virtualization using containers provides virtual environments, i.e. virtual userspaces, which can be isolated from another both in the sense of security and performance while still sharing the same operating system kernel and interfaces. In short, system virtualization virtualizes hardware interfaces and operating system virtualization virtualizes operating system interfaces or in other words: virtual machines emulate hardware and containers emulate operating systems (Figure 2.6) [45, 44].

Fundamentally, all virtual environments aim to provide some level of abstraction and isolation for software but operating system level virtualization is particularly appealing for applications. Many outsourced applications are compiled against portable system interfaces and can be packaged together with their dependencies such as libraries, configuration, and file systems. Furthermore, clients are not interested in the overhead of running, managing, and maintaining a secure operating system. This means that outsourcing a service as a VM

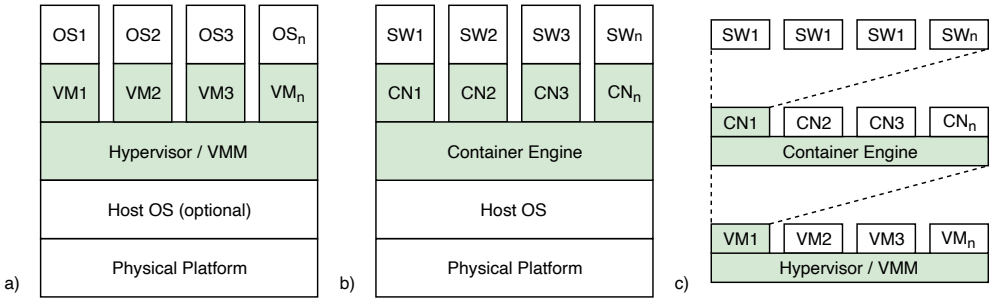

Fig. 2.6 Virtualization using a) virtual machines, b) containers, and c) a hybrid architecture. Relevant virtualization layer components are colored (green) with software / application layer components above and the infrastructure (e.g. Host OS) and physical hardware layers below. In case of a hybrid model c), VMs are used to run a guest OS and provide virtual infrastructure for containers [44].

can easily result in more operating system code than valuable application payload being shipped. Consequently, the virtual machine model for outsourcing applications often means that virtually the same operating system instructions are duplicated and executed across many virtual environments which produces obvious overhead. Container technologies are a way to provide a portable and isolated virtual environment which can share an operating system.

Irrespective of the virtualization level, isolation between virtual environments is a valuable security property. Systems which execute mixed criticality tasks, tasks with different degrees trustworthiness, or tasks of mutually distrustful tenants are often implemented around security guarantees such as strong isolation capabilities. Based on isolation guarantees, software in a virtual environment can enjoy increased security compared to systems and applications running on individually managed platforms. The increase in security is possible due to some level of homogeneity with regards to hard- and software type, quality, management, and maintenance[1]. Virtualization systems which can be monitored and managed by more capable agents such as today's cloud providers tend to provide a much more secure platform when compared to our personal devices.

[1]Not unlike Tolstoy's Anna Karenina principle which (roughly) states that *all happy families are alike* while *each unhappy family is unhappy in its own way.*

On the other hand, the trust that is placed into software running in a virtual environment is also heavily *dependent* on the virtualization system. A Virtual Trusted Platform, or simply *virtual platform*, primarily enables the use of TCG technologies in virtual environments (Figure 2.7). TCG-enabled virtual environments need to have access to security features similar to those accessed by systems running on (physical) trusted platforms. Virtual Trusted Platforms directly support security in virtual environments by establishing trust and enforcing trusted configurations (Section 1.2).

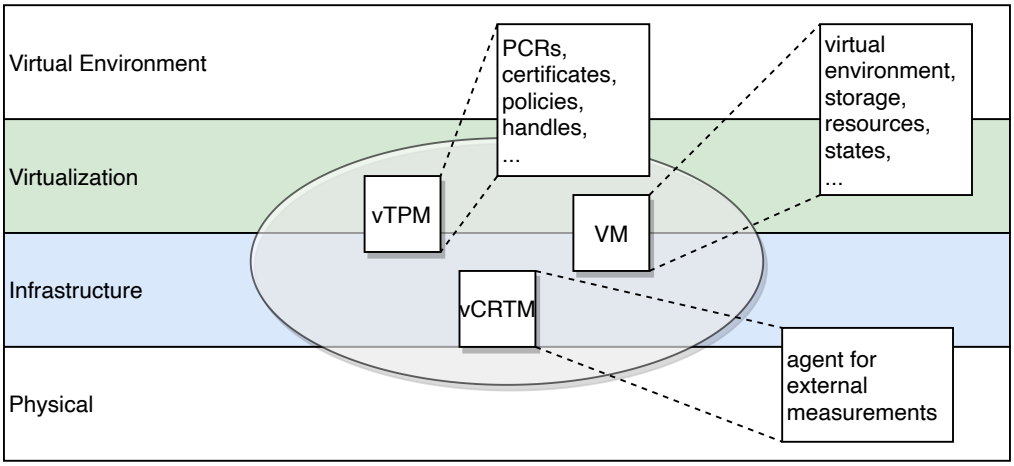

Fig. 2.7 The Virtual Trusted Platform is a concept which currently allows many options for implementations which might change the security and trust properties.

For example, a trusted virtualization system can be launched by using techniques described in Sections 2.3.4 and 2.3.5. A DRTM approach also helps to exclude a number of boot components in the measurement chain which one has to trust. From a trusted launch onwards, a virtual machine manager (VMM) has the capability to measure and record virtual machine or container images which might be especially useful when specific images have to be used every time. Recording the virtualization system and continuing measurements for virtual environments creates a chain of measurements between the trusted platform and the virtual environment and its software. Moving security critical functionality and dependencies away from the virtual machine manager to the trusted platform also limits the impact malicious administrators can have on a system. Hardware virtualization features are available on commodity platforms

and with support across many operating and virtualization systems. In theory, this allows the virtual machine manager or operating system to delegate trust back to the trusted platform. Consequently, the virtualization software system is treated like a manager under constant performance review and is trusted only to compile lists of recourses, start, and stop virtual machines [87]. Meanwhile, all actions are essentially supervised and executed by the trusted platform. If virtual environments are set up and isolated in such a system, an administrator with some control over the software system is no longer able to modify or introspect VMs without being detected. Aside from measuring and recording setups, clients do not have to trust the virtualization software system for correct execution of their VM. Such setups make processing sensitive data or sharing confidential material with software in virtual environments a viable option as both software and data can be sealed to a trusted platform, virtualization system, virtual machine, and other software (Section 1.2).

However, a shared hardware root of trust with software in virtual environments will in many cases lead to contention over the hardware resource [109] and might even violate carefully established security boundaries [79].

2.4.1 Virtual TPM (vTPM)

To solve issues related to contention, scalability, and isolation, a virtual TPM (vTPM) is presented to software in a virtual environment by some part of the virtualization system.

A vTPM is a virtualized TPM intended for use by software in a virtual environment such as a guest operating system or containerized applications. The kind of software in a virtual environment must be irrelevant to the vTPM's security and trustworthiness as it can range from trusted operating systems to adversary supplied code. A vTPM must be implemented in a way that confines a potential adversary to the usual TPM interface which is not trivial as the virtualization system may have access to a vTPM process. On the other hand, virtualizing the TPM by using the physical TPM as much as possible may improve security in some ways, e.g. key generation and protection capabilities, but will not result in better (or suitable) vTPM performance.

Despite possible differences in vTPM implementations, a vTPM appears as a standard TPM device to software in a virtual environment which fully complies with the specification and implementations must draw cryptographic primitives from certified sources. Trusted Computing Group documents do not limit or instruct how vTPMs must be implemented—current efforts seem to converge on a software / simulator based TPM instance as a vTPM [14, 34, 23, 109, 120]. However, the security features which a vTPM has to offer depend almost entirely on its integration with the virtualization system and the trusted platform [82].

For instance, if a vTPM is implemented as a process on an operating system, any attacker with the capability to introspect or modify running processes can do the same to a vTPM. Unlike a physical TPM, a vTPM does not have discrete storage locations where its state or secrets are kept and protected when the virtual platform is shut down. Continuing the vTPM as a process example, the vTPM state would be a file with options for protection offered by the file system. Again, an adversary with certain access rights can potentially steal, modify, or plant secrets. We can encrypt vTPM states and protect them from modification but this still does not prevent adversaries such as an administrator to spy on vTPM files and processes [13]. Several improvements which include utilizing secure enclaves and trusted execution environments as vTPM run-times are discussed in Chapter 5.

In summary, software in a virtual environment can use a vTPM as a root of trust for measurement, storage, and reporting but never unconditionally. A vTPM can currently not be treated as necessarily trusted, equally secure, or as well protected as hardware roots of trust. Instead, the trust placed in or delegated to intermediary, virtual roots of trust must eventually be justifiable using an actual root of trust. A positive outcome of this conclusion is that unlike hardware roots of trust, a vTPM is verifiable to some degree and verifiers may decide to use a vTPM only if trust in it can be established.

From a security perspective, a simulator or process based vTPM implementation can be dramatically hardened and improved in almost every way. From an operational perspective, a reasonably secure vTPM implementation offers many possibilities to improve on performance, scalability, maintenance, availability, and ubiquitous access to Trusted Computing technologies.

From a trust perspective, the trustworthiness of a vTPM largely depends on how it is *bound* to a platform's root of trust, e.g. a physical TPM, and how strongly and under which conditions it is associated to a particular virtual environment. In fact, the trusted computing capabilities of a vTPM may be harmful if an adversary is able to a) run it on an untrustworthy platform, b) access its secrets through a different virtual environment (Section 5.3).

2.4.2 Goals and Challenges for a Virtualized Platform

The adoption and support of Trusted Computing technologies in virtual environments is a necessary step for the trusted computing community and adopters of TCG technologies. Any successful approach has great potential to further increase the widespread use of safe and trusted computing technologies and will close an important gap in the federation of trust [79]. Using the physical platform's trusted computing capabilities to establish and enforce trust and security properties in the virtualization system can significantly limit the impact which adversaries can have on hosted systems and data.

Using a vTPM for virtual machines and containers does not immediately improve security for software and data in a virtual environment. In fact, when a virtual machine is attested using only a vTPM, there is great potential that the vTPM can be used against a verifier [82]. The use and adoption of trusted computing technologies requires a specification for virtual platform implementers and verifiers to establish necessary security and trust guarantees. In the past, TCG's Virtual Trusted Platform Architecture has collected and solved several functional and security goals and challenges for such a specification [56].

However, this document is now severely outdated and we summarize additional goals and challenges for a *modern* Virtual Trusted Platform (items 1,2,3,4,5):

1. Document a high-level architecture and system model.
2. Include operating system level virtualization in the architecture.
3. Address integrity of the virtual platform and its components (VM, vTPM, v(C)RTM).

4. Detail how a virtual platform can be rooted in a physical platform and its roots of trust.

5. Provide interfaces for existing specifications and protection profiles.

Goal 1 is a result of lessons learned since the last publication of [56] in 2011 as well as the research conducted in Chapters 3 to 5. It became increasingly clear that specifications driven by a software implementation become outdated quickly and are hard to adopt across different software and hardware systems. It also proved challenging to write "sub-references" for other specific implementations which resulted in a significant duplication of efforts to address a limited number of virtual platform implementations and use-cases. Using models instead of implementations as references could help to write more precise and adoptable specifications. However, there currently is a lack of such models which can express security and trust properties in virtualization systems.

Goal 2 is a direct response to the increasing popularity of containers as an alternative to traditional virtual machine solutions. The integration of containers into a virtual platform architecture which includes a suitable root of trust for measurement and chain of trust measurement design is a pressing task. The virtualization of operating systems is not currently supported in current trusted computing work and neither are hybrid architectures, i.e. VM's for infrastructure and containers for applications and services (Figure 2.6).

Goal 3 refers to the concept of a virtual platform which includes a vTPM and an analog to a CRTM/RTM as depicted in Figure 2.7. The association between components of the Trusted Platform, on a physical machine is typically a manufacturing problem. On a virtual platform, depending on the virtualization mechanism the association between the components might be entirely emulated or software defined. This represents a significant challenge for the use of trust information on a virtual platform as the *association* between components impacts the trustworthiness of many higher level trusted computing protocols.

Rooting a virtual platform in a physical platform (**goal 4**) has many advantages for the assurances which a virtual platform can provide. It allows verifiers to establish external facts (i.e., facts external to the virtual environment) which would be unknown if the physical platform type or identity was unknown.

Binding a virtual platform to a physical platform also returns control to clients over where and under which conditions their systems are running. Finally, virtualizing the roots of trust will give software in virtual environments access to many desirable features such as hardware-based key generation, protection, and policy management.

The last goal **5** for a *modern* Virtual Trusted Platform is a result of a number of developments in security research and standardization. Virtual platforms naturally encompass a large portion of today's technology and the concepts involved in virtual platforms are often at the cutting cutting edge of systems and security research. This makes research and development in the area rewarding but it also emphasizes that developments must provide interfaces to enjoy the benefits of other ongoing and future efforts. A number of security standardization and research groups are currently working on different aspects of trusted computing in virtual environments and expect TCG specifications to handle and provide interfaces to TPM-based solutions[2].

The following goals and challenges are shared with existing virtual platform specifications and still relevant:

6. Discuss responsibilities and potential issues of components of a virtualization system.

7. Minimize change to systems.

8. Provide support for the migration of virtual platforms.

Goal number 6 is similar to creating a high-level architecture and model for a Virtual Trusted Platform (item 1). Documenting individual components and putting them into the context of a virtual platform helps identifying essential functionality and relevant components depending on the virtualization model. [56] concluded that isolation and the components enabling it are most critical when trust in a virtual environment is established. Isolation can also help reduce the number of components a verifier has to trust if they are not dependencies and properly isolated.

[2]ISO, TCG, NIAP, ETSI, and GlobalPlatform as of 2020.

TCG technologies are mature and the TPM is a part of commodity platforms. Significant effort has led to many protocols and applications which must also be supported in virtual environments which is expressed in goal 7. The principle of minimizing change applies to the system which accommodates the virtual platform as well as the software system in a virtual environment. If necessary, changes of system or protocol configurations are preferred over changes in operations and implementations [78].

Goal 8 refers to a key feature of a virtual system, which is the ability to run it from backups, clone it if needed, and perform emergency recovery procedures which results in a very high availability. Virtual platforms must support at least some kind of migration between different physical platforms. A promising avenue towards migrating a Virtual Trusted Platform is to require homogenous physical platforms and virtualization systems which would provide essentially the same kind of environment and security guarantees. The issue of migration becomes significantly more complex if platforms with different security properties and trust models are considered.

The following goals can be seen as desirable features which a virtual platform implementation should possess:

9. Allow software in virtual environments to run unmodified.

10. Allow binding a virtual platform to a physical platform.

11. Allow verifiers to determine security guarantees, assumptions, and trust model.

12. Provide a way for software in virtual environments to establish trust in the virtual environment, the virtual platform, and the virtualization system.

Goal or feature 9 targets a trend towards moving dedicated infrastructure, service architectures, and applications to virtual systems and servers. A Virtual Trusted Platform should not require significant changes to TCG applications when the are copied from physical machines. Minimizing changes in this aspect

improves the portability of new developments and it opens a virtual platform up to a large amount of commodity systems with TCG support.

Goal 10 refers to binding a virtual platform to a physical one. Such a feature is important for a number of use-cases ranging from Network Functions Virtualization (NVF) to Internet-of-Things (IoT) applications. One immediate benefit of an explicit binding is the additional control over the execution environment, platform type, and even physical location.

Running TCG-enabled applications in a virtual environment should improve security and trust properties of the application. However, it is clear that some reduction in assurance or expansion of the trust model can be the consequence of using a vTPM implementation [56]. The protection afforded by the vTPM implementation and the virtual platform as a whole must be clearly communicated to software in the virtual environment, relying parties, and verifiers (goal 11).

Lastly, feature 12 refers to the use of trust information of the entire virtualization stack locally, e.g. for data protection, or remotely during an attestation (Section 2.3.5). The security of software in a virtual environment may depend heavily on the underlying system and the virtual platform should support the attestation of both application and system software.

2.4.3 Summary

The Virtual Trusted Platform is a concept with many options for virtualizing the trusted platform. However, the security and trust properties of the virtual platform heavily depend on its implementation. The definition of a *modern* Virtual Trusted Platform needs to include containerization, remove implementation specific details, and clarify essentials such as the integrity of a virtual platform and how it relates to physical platforms. The following chapters elaborate on attesting a virtual environment and the virtualization system, the inclusion of containers in integrity measurement architectures, and the integrity of the resulting virtual platform in Chapters 3 to 5. The goals and challenges presented in the previous section are picked up again later and serve to discuss practical

aspects of this thesis in Section 6.1.1. Ultimately, the practical success of the Virtual Trusted Platform will be determined by its adoption in future systems.

Chapter 3

User-Centered Attestation

This chapter continues Section 2.3 by briefly revisiting the Trusted Computing tool-set and its current application in virtualized systems. Virtualization is omnipresent as the backbone of cloud, edge, and fog computing as well as *X-as-a-service* infrastructure. It continues to gain increased popularity even in edge or end-user and embedded devices. The need for standards and specifications for secure and trustworthy collaboration with virtualized systems becomes a pressing issue. We discuss challenges related to translating the term *trust* between specifications for hardware modules such as the Trusted Platform Module (TPM) and applied specifications for operating systems, virtualization systems, and virtual machines—defining trust establishment becomes crucial for specifications which aim to extend *trust* beyond the TPM. We define User-centered attestation as a set of principles suitable for layered, decentralized systems along with a methodology for specifying and synthesizing such a trust establishment strategy. The contributions of this chapter are listed in Section 3.1.1. The results presented in this chapter have been published in [80].

3.1 Introduction

The cloud-centered paradigm faces a major shift: the success of the Internet-of-Things causes the generation of the majority of data at the outer edges of a network [122, 123, 49]. *Edge computing* refers to the set of technologies

allowing computations to be performed along the edges of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services [122, 123]. *Fog computing* is closely related to the general concept of *edge computing*[16, 26, 141] with a strong focus on performing task in nearby, decentralized systems. For some tasks, this yields considerable feats such as lower latency and improved user-experience as well as resilience through redundancy for services. Virtualization, essential for concepts ranging from *Infrastructure-as-a-Service* to *Functions-as-a-Service* implemented by vendors like Microsoft, Google, and Amazon has fundamentally changed the way software and data is being handled and is the backbone of modern computing infrastructure, especially with an increase in service decentralization and dedicated, collaborating nodes[97, 122, 16].

The importance of trust and trust establishment strategies becomes apparent in decentralized systems with no immediately recognizable authorities. The rather ambiguous issue of trust and collaboration can be demonstrated using a very small, discrete example:

Suppose a mathematician who is interested in number theory uses a computer with a program for factorizing numbers. The output that will be produced by that program is either the factorization of a given number or a statement that the given number is a prime. Now suppose that the same mathematician wishes to inspect a large number, too large to verify without the aid of the computer. The mathematician can have two possible expectations at this point: the given number is a prime number or not. Assuming there are strong reasons to believe that the number is a prime, the result of the program can either confirm this by telling that the number is a prime or give the factorization as evidence that intuition has in fact fooled the mathematician. The situation changes, however, if the mathematician has strong reasons to believe that the given number is not prime. Again, the computer can produce two possible outputs: the number is a prime or a factorization. If the output is a factorization, the mathematician can confirm the belief by recalculating the given number. However, if the computer comes back with the result that the number is a prime, contrary to strong reasons leading one to believe otherwise, why should the mathematician trust this result? [38] This example illustrates that even in completely discrete

problems, the computation may not be worthwhile if it lacks convincing power w.r.t. the *quality* of the result. As possibly dated and oversimplified as it may seem, the issue raised here, instead of being remedied, is being amplified by modern efforts using more flexible, decentralized computing systems. As a practical set of standards-based technologies, Trusted Computing [64] can serve to supply evidence about a computing platform. The process of collecting, supplying, and appraising evidence, and ultimately a result, is referred to as Remote Attestation [28, 19, 79].

3.1.1 Contribution

This chapter introduces current technological and standardization efforts towards trustworthy cloud computations. Implementing trustworthy virtualized systems currently requires the adoption of at least two standards for hardware and application level trust. We outline challenges and potential conflicts related to translating *trust* across such standards. We then review and evaluate current trust establishment methods and put them into perspective of decentralized systems. Finally, we propose user-centered attestation as a candidate for layered, decentralized systems along with a methodology and strategy for specifying and synthesizing such an attestation system.

3.2 Trusted Virtualization Platform

The notion of a trusted virtualized platform is coined by Trusted Computing Group's (TCG) companion architecture specification "Virtualized Trusted Platform Architecture" [56]. The TCG coined term and the architecture itself is rather vague but decisively extends the adjective *trusted*, which should cause curiosity since intuitively only the TPM [64] *should* be trusted but not the entire platform. Recently, Akram et al. [1] have adopted and developed the term in a position paper on digital trust in their vision for trusted cloud computing using the architecture in Figure 3.1.

While the concept of processing and storage hardware and a respective management (OS) is uncontroversial and not of general concern, the notion

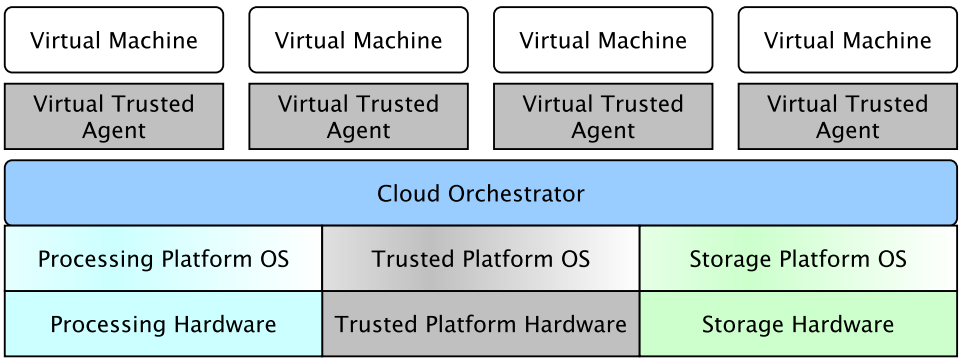

Fig. 3.1 Proposed architecture for Trusted Computing for Cloud Computing. [1]

of Trusted Platform Hardware and a Trusted Platform OS raises questions, especially, in combination with Virtual Trusted Agents (represented by vTPMs). Returning to the mathematicians' problem, it seems acceptable, even reasonable, that the Trusted Platform Hardware is in fact trustworthy and can be trusted. After all, specifications like [64] describe which functions and properties are required to synthesize a TPM and manufacturers can follow them in their implementations. By inspecting a key credential, unique to each TPM, called Endorsement Key (EK) and a manufacture certificate, one can infer the authenticity and trustworthiness of a TPM—assuming the manufacturer is *competent*.

3.2.1 Levels of Trust and Specification

The supposedly trusted platform OS seems to be a *déjà vu* for our mathematician [38]: there are no indications about the trustworthiness other than that it carries the adjective *trusted* from an external perspective. Described as quality of results, the mathematician upon interacting with the OS will need supporting evidence, or metadata other than the result itself to be convinced in every case that the interaction is correct or at least as desired or expected. This problem, although derived from an abstract case is quite intuitive: Unlike with the TPM, assuming trustworthiness of the OS is hardly justifiable. The reasons as to why that assumption is hard to substantiate are complex, especially since the specification is concerned with security. TCG's specification for Virtualized

Trusted Platforms is considered a specification for *TCG Applications*. It describes general requirements such as minimum required key sizes and lengths, it has a glossary of terms, and most importantly it describes interactions with virtualization layers, or hypervisors, such as the *self-defending* Trusted Computing Base along with processes for instantiating, storing, or migrating virtual machines along with their *Virtual Trust Agents*. The idea is that a *vendor* can implement these requirements and refer to the conceived OS or platform as *Trusted* Virtualization Platform. The problem here becomes apparent however, when the predicate *trusted* derived from this specification is compared with the TPM itself: The Trusted OS along with Virtual Trusted Agents must be *fully* specified and (remotely) verifiable[74] to account for issues related to trusting software[77, 128].

3.2.2 Remote Attestation and Virtual Machines

From an external perspective it seems unreasonable for a specification to suggest to a suspicious party that it should simply trust whoever claims to implement the specification. While this might seem reasonable from a perspective of contracts in business to business scenarios, the nature of distributed IoT systems and the idea of plug-and-play configurability of systems invalidates such *out-of-band* assurances. In order to be meaningful to a consumer or any interacting party, such a specification must provide a method to collect metadata about the potential trustee as to why it can be trusted within the scope of a specification. Defining a remote attestation process seems like a high priority task for any specification that builds on top of a TPM while requiring compliance of the implementer. Especially, since this compliance is easy to break in any software system if the potential trustee has malicious intent but also when the trustee is unaware of the fact that part of its software configuration are not suitable to an external party. The external party on the other hand would certainly like to see proof that its potential trustee implements the specification and has no additional or undesirable configuration on top of it. A focus on making a specification which systems can be checked or attested against yields mutual benefits both for trustor and trustee.

3.3 Towards a User-Centered Attestation

Instead of defining yet another Trusted Virtualized Platform, the following subsections will explore what sort of properties and operations can lead to the conclusion that a particular platform can be *trusted*.

3.3.1 Principles of Remote Attestation

The need for remote attestation and its process have been defined by the TCG [56, 64]. However, besides the basic interaction with the TPM, which can be summarized as a trustworthy mechanism, an attestation should follow general principles outlined by [28]:

Principle 1. *Fresh information. Information about the target of an attestation should reflect the running system, e.g. programs it is currently running and not just disk images. The reasoning is that showing only disk images tells very little about which part of this configuration is actually running. Inspired by measurement tools that provide merely measured boot, i.e. a boot sequence that reports the disk contents in the form of a hash will not supply information as to whether or not any (protection) mechanisms available on disk are actually being executed. Other measurement tools provide* load-time *information by reporting hashes of executed binaries, while in more recent years approaches for measuring active memory have emerged. An appraiser may, however, have its own expectations as to how* fresh *the evidence is which leads to the next principle.*

Principle 2. *Comprehensive information. An attestation mechanism should be capable of delivering comprehensive information. This implies that such an attestation mechanism must have access to all internal states and must be able to report these using local measurement tools. While this inevitably leads to concerns about* dangerous *disclosure, i.e. reporting vulnerable states, and ultimately raises privacy questions, it also* should *consider a problem related to the amount of reference measurements in an appraisers' database — especially if there aren't any specialized appraisers.*

Principle 3. *Constrained disclosure. An attestation target should be able to decide which information is sent to a particular appraiser. The appraiser should be*

identifiable and the target platform must be able to enforce policies defining what evidence it supplies to an appraiser. Rather than suggesting the appraiser to be identifiable, it should be suggested that the appraiser reveals the type or amount of evidence it requires *in order to be convinced. This implies that there is a semantic for attestations which leads to a conclusion as to whether or not a target can be trusted.*

Principle 4. *Semantic explicitness. The semantics of an appraisers' trust decision should be explicitly defined in logic. As an example on a local scale, it should define that a trust decision is made about a particular service, or the entire target platform and how trust in a particular service is established by supplying evidence of supporting or coexisting services on the target. Furthermore, it must be defined how subsequent attestations affect the initial decision and how they can be correlated for logical inferences.*

Principle 5. *Trustworthy mechanism. Appraisers need to infer the trustworthiness of the mechanism that is used to deliver evidence to them. This principle, although introduced last, is intuitively critical as not fulfilling it invalidates all principles so far since the evidence might simply not be convincing as it can not be trusted.*

These principles were defined as general requirements for attestation architectures utilizing a trustworthy mechanism to supply convincing evidence about a particular platform. However, w.r.t. virtualization and in an Network Functions Virtualization (NFV) scenario Lauer et al. [79] have extended these principles to address issues related to the multi-user and multi-layer architecture of virtualized environments:

Principle 6. *Layer linking. When a virtual environment is attested, its underlying components or layers must also be attested. Attesting a virtual platform without inspecting its underlying layers and ultimately the physical platform supplies only very limited evidence to an appraiser making a trust decision. A quote generated by a virtual trust agent such as the vTPM must be substantiated by a TPM quote so as to indicate the trustworthiness of the virtual trust agent's quote.*

Principle 7. *Scalability. Since attestation, i.e. (v)TPM quotes, can occur for any virtual machine triggered by any user or appraiser, substantiating each vTPM quote with a TPM quote will inevitably lead to the TPM becoming a bottleneck in a virtualization scenario. An attestation mechanism must treat the TPM as a limited and shared resource and offer a scalable protocol between vTPM and TPM quotes.*

Principles 6, 7 can be seen as a virtualization specific addition to principle 5. Following these principles also reveals and important proposition: Using a trustworthy mechanism implies that all other principles apply to the trustworthy mechanism itself.

3.3.2 Attestation in Virtualized Environments

Following these principles, two distinct approaches (or variants) have emerged in research and current standardization work. The first approach being a direct translation of remote attestation protocols using the vTPM associated with a VM as the key component of the trustworthy mechanism in combination with an attestation of lower layers using the same protocol but this time with the TPM. The later approach [79] was introduced subsequently as a solution for NFV and *X-as-a-Service* infrastructure. It assumes multiple VMs and *few* or no VM users and a single hypervisor operator. The trustworthy mechanism relies solely on the TPM while vTPMs are utilized as potentially untrusted sinks for upper layer measurements. The following paragraphs will detail these two approaches based on the principles 1-7.

Attestation Approach I

Attestation approach I attests virtual and physical environments using the *same* attestation mechanism (Fig. 3.2). An approach treating VM and lower layers equally has intuitive benefits: it is easy to integrate in an existing protocol landscape [19, 64], architectures implement vTPMs as virtual devices[14, 79], appraisers can reach a trust decision using a *standard* evaluation of the evidence or include the properties of layered systems dynamically.

Table 3.1 Fulfillment of attestation principles in a system using separate VM - hypervisor attestations.

Principle	Fulfillment
1	Requires measured or trusted boot, can accommodate Integrity Measurement Architecture (IMA) [117] for load time measurements of binaries.
2	During boot each component measures its successor into a PCR. As soon as the kernel is measured and loaded, the kernel load mechanism is responsible for measuring newly loaded binaries.
3	Appraisers do not have to be known to the target, mutual attestation is not part of any proposal.
4	The trust decision is based on whether or not supplied evidence as hash values of binaries can be found in a reference database. The appraiser then correlates attestations of any layer n with layer $n-1$. An appraisal of a VM depends on the appraisal of its hypervisor.
5	The trustworthy mechanism includes a TPM quote over a PCR (along with IMA log for hash verification) for the hypervisor. The attestation process is completed through a vTPM quote over vTPM PCRs and respective logs explaining the platform configuration hash. Credentials in a vTPM must be *trustworthy* and protected from leakage.
6	The appraiser must have *a priori* information about the locality of a VM or a list of VMs hosted by an hypervisor, a VM - hypervisor mapping. Assuming integrity and availability for such a mapping, the appraiser can correlate TPM and vTPM quotes when evaluating evidence in a decision process.
7	1:1 relation between TPM and vTPM quotes assuming each vTPM quote must be preceded or succeeded by a TPM quote (depending on principle 4,5).

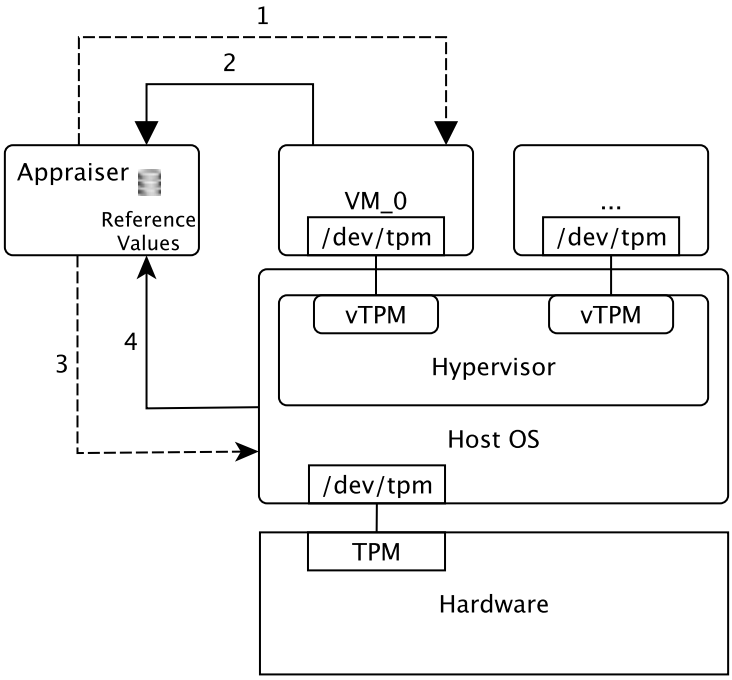

Fig. 3.2 Attestation Approach I illustrated. The appraiser, holding *suitable* reference values, requests evidence from different layers sequentially. Boxes denote hardware modules, rounded containers denote attestable software components. Dashed lines indicate requests for meta data and bold lines indicate evidence responses.

Discussion. Separate attestation approaches are favorable for an implementer as they require little modification to existing trust establishment processes. However, this also implies specific assumptions towards the entire attestation system. Principles 1, 2 are fulfilled using what is the *de-facto* standard in Trusted Computing approaches for accessing and measuring components for later evaluation by an appraiser. In a virtualized scenario, it should be noted that it is the hypervisors responsibility to maintain necessary components such as the vTPM as a root of trust for storage while the VM configuration must supply boot code that acts as the root of trust for measurement. Principle 4 and 5 are closely related and demand a rigorous definition of trust, especially, concerning the correlation of the two kinds of measurements. Explicit semantics of a trust decision are critical for a protocol design and evaluation and the traditional approach of matching well-known hashed to measured hashes is not suitable for

comprehensive decisions. As far as the trustworthy mechanism is concerned, the presented approach relies on a critical assumption: VMs and associated vTPMs are not only strongly coupled but also isolated. Since the vTPM is required to perform *quotes* over its Platform Configuration Registers (PCRs), credentials must only be accessible to authorized parties such as the VM itself. However, even under the assumption of *strong* isolation, the underlying hypervisor still has responsibilities which, in an effort to provide comprehensive information, must also be reviewed. Consequently, a VM attestation must be followed by an attestation of the hypervisor — or vice versa? Intuitively, causality dictates that the hypervisor provides the run-time for a VM and therefore any property of the hypervisor affects the trustworthiness of the VM: attesting the hypervisor and subsequently attesting the VM seems effective. However, another interpretation would be that the appraisal of a VM becomes *effective* only after the hypervisor has been attested. Both interpretations seem to fulfill the attestation requirement in prose which again emphasizes the need for clear semantics.

Attestation Approach II

The second approach towards attestation of virtual machines is distinctly different from separate, legacy attestations insofar as it is intended for virtualized infrastructure such as *X-as-a-service* and NFV. Entitled "Hypervisor-based Attestation", the approach relies solely on the TPM and mechanisms recorded into PCRs to collect and supply evidence to an appraiser — this implies that even the *trustworthy mechanism* can be verified. The vTPM itself does not need to perform a quote over internal data structures and therefore, with attestation in mind, does not need to be raised to a *somewhat* trusted level. The attestation flow is outlined in Fig. 3.3.

Discussion. Hypervisor-based attestation approaches (Fig. 3.3) have three common benefits: Bottom-up attestation, coupled hypervisor-VM attestations, and inherently *en bloc* evaluation. Regarding the trustworthy mechanism (Principle 5), a hypervisor-based, bottom-up attestation is, at a glance, uncontroversial and in-line with conventional attestation strategies. However, common approaches use IMA [117], a kernel-based loader extension, exclusively once

Table 3.2 Fulfillment of attestation principles in a system using hypervisor-based attestations.

Principle	Fulfillment
1	Same as Table 3.1.
2	Same as Table 3.1.
3	Same as Table 3.1.
4	Supplied hash-logs are compared against *golden values*, unknown values indicate untrustworthy states.
5	The trustworthy mechanism is comprised of a two stage measurement process. The first stage measures hypervisor components, including the attestation manager and vTPMs. The second stage measures VM components into vTPMs. For reporting, only the TPM is consulted and vTPM PCRs are collected and attached to a hypervisor attestation.
6	Once the appraiser has verified the attestation manager and vTPMs, the inspection of VM measurements reveals the hypervisor - VM mapping. Prior knowledge of VMs and hypervisor mappings is not required.
7	1:n relation between TPM and VM attestations.

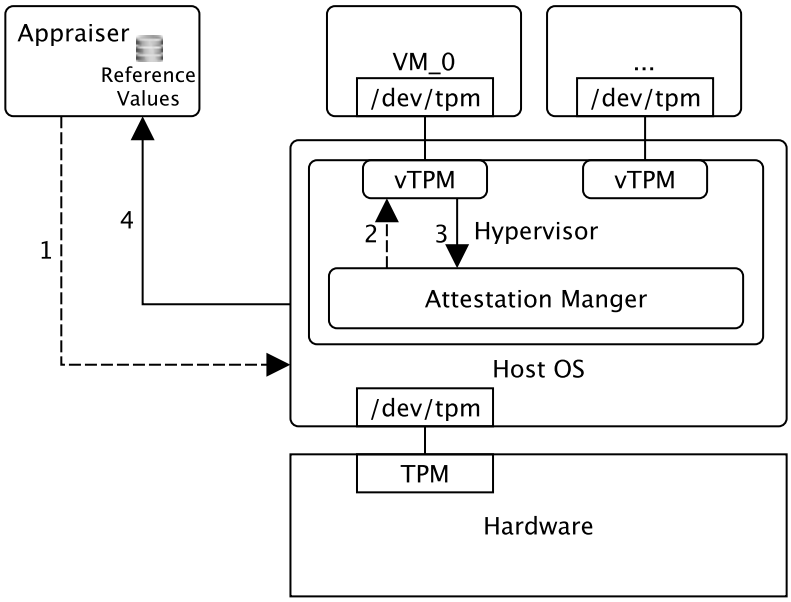

Fig. 3.3 Hypervisor-based Attestation. The Appraiser attests the Target including *n*-VMs. Attestation Manager, a process in the Host OS or hypervisor, collects individual evidence produced by VMs from a vTPM interface. Subsequently, the attestation target sends a response containing its own and each VM's evidence.

the module is loaded to measure and propagate loaded components to the TPM. An attestation manager or vTPM will, instead of reporting to the TPM, keep *own* records - the integrity of this functionality is then appraised using the load time hash of these components. As a result, the trust that can be placed in the completeness and correctness of such records is transitive and relies on trusting supporting mechanisms such as the attestation manager and vTPM instance. In fulfillment of Principle 4, such considerations should be addressed by a defined trust decision process that includes and respects the concept of *transitive* trust relationships. Furthermore, [79] does not specify how individual VM appraisals affect trust decisions for other VMs running on the same platform — a lack carried over from utilized measurement architectures.

3.3.3 Observations

Having revisited both attestation approaches, the following observations can be made: (i) Attestation of a layer inadvertently requires attestation of the layers below. (ii) Even if the Virtual Trusted Agent, or vTPM, is used for a signature, the trustworthiness of that signature relies on the appraisal of the hosting layer. (iii) Neither approach is semantically explicit with respect to their treatment of evidence in layered systems beyond suggesting exhaustive platform hash comparisons. (iv) Designated appraisers are present and capable of correlating and evaluating supplied evidence. (v) Most importantly, knowledge and connectivity of lower layers is assumed.

3.4 User-Centered Attestation

The observations of Section 3.3.3 will serve as defining considerations of the attestation strategy entitled *User-Centered Attestation* (UCAS) (Tab. 3.3).

Discussion. Unlike separate and hypervisor-based attestation, UCAS assumes connectivity only to a VM. Appraisers, which could be any agent or device, must therefore receive evidence of the VM running a service in question along with information of its lower layers such that their appraisal is based on comprehensive information. Such evidence must be gathered bottom-up [114, 79] so as to reflect causalities like boot order and relations between components loading, instantiating, and guarding other components.

Additionally, changes to localities of VMs or changes in a lower layer must be measured, recorded, and reported accordingly. In order to enable attestations initiated by VMs and with VMs as the only point of contact for appraisers, lower layer information must be propagated such that the evidence a VM supplies automatically reflects a current state.

As far as the semantics are concerned, trusting a VM must be treated as a decision an appraiser has to make. Components involved in measurement and reporting processes in layers above trusted hardware may not carry the predicate *trusted* by default. Based on this notion, it becomes clear that recording trust information needs special attention. While the appraiser can rely on the

Table 3.3 User-centered attestation approach in fulfillment with attestation principles.

Principle	Requirements
1	Measured or Trusted Boot, IMA [117] providing at least load time integrity which can be extended to run-time integrity [111].
2	Causally ordered, bottom-up event log.
3	Peer-to-peer trust propagation, only information relevant to the virtual environment is revealed.
4	Quotes must convey security guarantees, transitive relations, support for partial attestations, and appropriate use of safe-guards.
5	VMs should be capable of gathering and sharing relevant information of lower layers and with appraisers using RTR, RTM, and a trustworthy measurement architecture.
6	VM evidence must contain proof that the included lower layer evidence is relevant, i.e. the lower layer has created or is currently hosting the VM.
7	1:n relation between TPM quotes and upper layer attestations, TPM quotes can be used implicitly with safe-guards such that vTPM quotes of upper layers can be related to them according to the semantics in principle 4.

TPM or vTPM to act as a root of trust for reporting (RTR), the root of trust for measurement (RTM) design is critical (Section 2.3.3). Using a nuanced and layer specific notion of trust enables reasoning about the trustworthiness of supplied evidence. With proper isolation guarantees in the virtualization system, we should be able to exclude unrelated virtual environments from an attestation. This idea can be continued to exclude other unprivileged or isolated software and improves the explicitness of the supplied evidence significantly. However, balancing explicitness with completeness (i.e., no relevant information

is omitted) is a key quality of a trusted measurement recording and reporting mechanism. This concern is discussed further in Chapter 4.

The supplied evidence must also carry information as to why and to which degree the evidence itself can be trusted. A big step towards this goal is to use a trustworthy integrity measurement architecture as a root of trust for measurement (RTM) in combination with the TPM as a root of trust for recording (RTR). The chain of integrity measurements can then be evaluated from system boot (i.e., bottom-up) to its current state. Appraisers must be able to check each record and its impact on the trustworthiness of the remaining measurements.

As a first step towards semantic explicitness, vTPM quotes will need to convey trust information based on the dependencies of the vTPM component itself and the associated virtual environment (e.g., a VM). Appraisers can then decide whether the attestation is sufficient under a certain trust model or if further evidence is required to resolve issues. Common issues are ambiguity in the virtualization system state (e.g., unknown software or configurations) or lacking security enforcement on the system in which case more components of the system need to be attested.

Achieving explicitness can aid scalability issues for the recording, reporting, and evaluation of evidence. For instance, recording stable parts of a system separately (e.g., boot and hypervisor components) from dynamic layers (e.g., software in virtual environments) results in more stable PCR values in the TPM an vTPM. Consequently, remote attestation can be done by using certificates or secrets bound to specific PCR values. This reduces the amount of TPM and vTPM quotes which are needed but it also allows verifiers to predefine and reuse appraisals.

Lastly, explicit semantics (i.e., what information is needed in an attestation) in combination with trustworthy mechanisms will serve to derive suitable trust propagation mechanisms. Using certificates as a replacement for slow TPM quotes together with safe-guards to ensure that certificates are kept current can provide a simple and scalable way to propagate trust information securely into upper layers and to other parties. Trust propagation itself is essential to decentralized and distributed systems. Individual nodes and parts of the system might not be directly accessible to a curious party. Being able to trust appraisals

and reusing them significantly reduces the amount of individual attestation requests to nodes and remedies scalability issues related to TPM and vTPM quotes.

3.5 Related Work

Further concerns related to managing trust in Trusted Computing specifications are discussed in [129]. The semantics of trust, trustworthiness, and trust establishment in peer-to-peer systems are discussed further in [15, 112, 114]. Alternative approaches towards trust and trust management based on social notions and reputation systems are presented in [72, 86, 115]. The related work of this chapter is discussed in greater detail in Chapter 6.

3.6 Summary

Virtualization poses an interesting issue for specifications towards trustworthy systems as the *trust* placed originally only in hardware components needs to be extended to reporting and measurement mechanisms in upper layers. While approaches towards trust establishment exist, their semantics are ambiguous and an appraiser has to decide whether a virtualization platform or upper layers are trusted without much guidance or support in reasoning for such a decision. Furthermore, existing attestation approaches imply a particular topology, connectivity, and capability that does not reflect decentralized systems. A *User-Centered Attestation*, as a novel attestation system, encompasses these concerns and proposes a strategy for specifying and synthesizing suitable trust establishment mechanisms and hopefully inspires further research and contributions towards standards for open and collaborative trustworthy systems.

Chapter 4

Domain Specific Measurements

This chapter presents the design and verification of a secure integrity measurement system for containerized systems. Containerization of applications allows fine-graded deployment and management of services and dependencies but also *needs* fine-graded security mechanisms.

We provide formal abstractions for containerized systems by introducing LS^3, a formal model and logic with sub-domain constructs to represent stratified systems and their interactions. Using our formal model, we prove that the widely used Trusted Computing Group (TCG) based Integrity Measurement Architecture (IMA) securely extends trust measurements from boot to applications.

However, IMA is not designed to make domain specific trust measurements and is consequently incapable of creating domain specific integrity reports. Current research aims to improve either trust measurement performance or comprehensiveness but does not improve the measurement function and its semantics to allow remote verification of measurements per domain. We present an enhanced trust measurement architecture design, which produces domain specific integrity measurements suitable for fine-graded remote attestation.

Providing domain specific integrity reports eases system and sub-system verification and yields desirable properties such as measurement log stability and constrained disclosure for multi-domain systems. We verify and prove the correctness of our trust measurement architecture using our formal model. The

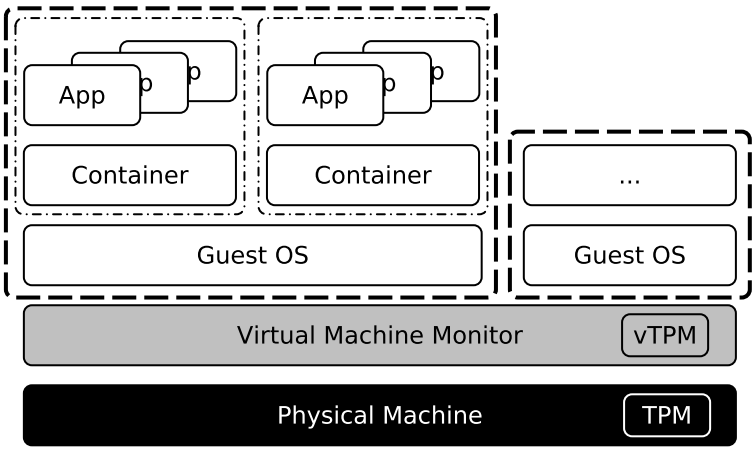

Fig. 4.1 A container system running on top of virtualized infrastructure. The dashed line indicates the system components we are interested in: the operating system *domain* and contained applications in *sub-domains*. The operating system can run either bare-metal on the physical platform and Trusted Platform Module (TPM) or as a *guest* on a virtual machine monitor using a virtual TPM.

contributions of this chapter are listed in Section 4.1.1. The results which we present are published in [83].

4.1 Introduction

Containerization is gaining increased popularity in a variety of settings and may be a key step towards future cloud computing models [70]. Containerization of applications removes the need to customize and configure an entire operating system in order to run a set of applications with their dependencies. Instead, an operating system is modified to include a *container-engine*, which runs and isolates containerized applications in their *virtual user-space* or virtual domain (Fig. 4.1). This concept is also referred to as *light-weight* operating system level virtualization because containers still share an operating system. Removing the overhead of running an entire operating system lets a customer handle her applications while a provider can handle everything else: instance selection, scaling, deployment, fault tolerance, monitoring, logging, security patches, and so on [70].

Since containerized applications share an operating system kernel, *trust* in a containerized application depends as much on the *integrity* of the host system as it depends on the hosted application itself. Only by inferring the integrity of the operating system *first*, we can infer the integrity of applications confined to their virtual user-spaces. For example, Alice may offer to host containerized applications and has set up infrastructure similar to Figure 4.1. Carol and Mallory both give a container with applications to Alice, which promises to run. As customers, Carol and Mallory *trust* Alice's setup to provide security properties: most importantly, isolation both in the sense of performance and protection. Being able to establish trust in the operating system and infrastructure becomes crucial if the provided service itself needs to be trusted. Alice needs a way of monitoring her host system for compromises. In addition, Carol and Mallory want to monitor and establish trust in the host system and the integrity of their container environments themselves. However, when reporting on the integrity of the system, Alice has to make sure that she provides *complete information* that is *constrained* to each container or otherwise she might lose trust or worse, give Mallory an incentive to launch targeted attacks against Carols applications.

Container systems are specifically designed to not only give the illusion of a isolated user-space per container but to enforce it. Establishing its integrity is an important step towards establishing trust in applications running in their virtual environments. One way to establish trust in an operating system is to use the Trusted Platform Module (TPM) to record its software state. Using a process called remote attestation this information can be securely conveyed to a remote party. Given the software state, an agent may then infer the operating systems integrity and further properties to decide whether it can be trusted or not. While the TPM is capable of securely storing and reporting software recordings, a *measurement function* within the target system is required to take and extend measurements to the TPM in a trustworthy manner. Trusted and secure boot [50] create a chain of trust measurements from system boot all the way to when control is transferred to an operating system kernel. The Integrity Measurement Architecture (IMA) [117] is a kernel function that immediately continues chain of trust measurements by recording code and data before it is mapped to the system memory.

The existing body of research on trust measurements is focused on *comprehensiveness* and *performance*. Examples include moving from measuring code at load-time to measuring memory segments during run-time [111] on one side and policy reduced measurement targets [125, 119], e.g. kernel modules only, or batch-extend techniques on the other side so as to increase the overall performance of the measurement function. None of these target measurement semantics, remote verifiability, and constraining disclosure. Formal investigations targeting secure execution environments [126] are complementary and can be combined with our work.

We propose a measurement architecture with constrained disclosure features built in by making domain specific measurements to begin with. We argue that we preserve comprehensiveness and other desirable properties. To this end, we use the modular design of LS^2[36] and add to it domains. Domains are logical *strata*, i.e. VMM, OS, container, with a logical relation defining control and interface confinement. In short, the operating system and its threads are the primary domain. We then create *lightweight* domains under the control of but with interfaces to a primary domain. This corresponds to an operating system controlling and managing all memory while providing each container with a partition of *virtual* addresses. Using the idea that resources are allocated to domains, we simplify a proof made with LS^2 in our extension titled LS^3. We extend prior work to prove the trustworthiness of measurements taken by IMA. We show that through our domain-subdomain construct measurements can be recorded into dedicated TPM locations and we link them with a technique called *forward linking*. Consequently, when we wish to report on the software state of a container and its host operating system, we can report on a domain and a particular sub-domain in a fine-graded manner instead of divulging all measurements. We prove the properties with the same rigor as LS^2 has proven a sequential boot sequence and show that while we still measure *everything*, through sorting and linking we achieve a much more verifiable and constrained system state per container.

4.1.1 Contribution

In this chapter, we make the following contributions:

- LS^3 extensions for containerized systems in Section 4.2,
- the design and verification of our Enhanced Integrity Measurement Architecture (EIMA) in Sections 4.3 and 4.3.3,
- an analysis of the trustworthiness of integrity measurements for IMA in addition to constrained disclosure in EIMA (Section 4.3.3).

4.2 A Logic for Secure Stratified Systems

Our work is based on the prior work in [36] which itself represents an improved version of several logic systems [35]. The work we present can be seen as a major update and represents a new strand of the logic: we externalize the creation and confinement of new executions in a virtual system which adds consequential modeling abilities.

4.2.1 A Logic of Secure Systems

Our work draws from the successful analysis performed in [36] which presented a logic for secure systems or *LS²*. We build upon *LS²* and use what [36] refers to as constrained by system interface (CSI) adversaries, which are simply modeled as extra threads existing on the target system. Our extension exploits LS^2's modularity and allows us to *explicitly* create new threads executing untrusted code. We found that since LS^2 was intended for analyzing only a small part of trusted computing based systems relying on *loading* and *jumping* to new code, reasoning about *launching* a complex system at kernel and user level threads are impossible. We extend the capabilities of LS^2 well into application space and potentially virtual machines to fulfill some of the promises made in [36].

LS^2 is a metalanguage which has two main design goals: (i) allow for the abstraction of thread based sequential execution of programs, and (ii) provide a proof system and logic for defining and verifying desirable *safety* and *security*

properties. We revisit LS^2 [36] as a three part formal system consisting of an impure functional programming language, reduction-based operational semantics, and a first-order logic system with hybrid extensions over security properties. The programming environment has a syntax close to that of functional programming languages like ML [94]. Most notably, programs are a sequence of *actions*, represented as a stack in [69] and need only the sequential composition operator ";" to indicate that one action is to be executed after the other. The set of *actions* conveniently abstract manipulation of memory locations, network communication, anonymous function definition and evaluation, as well as simple program flow controls based on comparing values instead of *if-else* constructs. Most notably, LS^2 has explicit actions for obtaining and releasing write locks on (memory) locations. Such programs end with $\cdot$ or "no-op" or with a *jump* to another program:

$$
\begin{array}{llll}
\textit{Location} & l & & \\
\textit{Expression} & e & ::= & n \mid x \mid (e, e') \mid \ldots \\
\textit{Actions} & a & ::= & \texttt{read}\ l \mid \texttt{write}\ l, e \mid \texttt{hash}\ e \mid \\
 & & & \texttt{lock}\ l \mid \texttt{eval}\ f, e \mid \texttt{match}\ e, e' \mid \\
 & & & \ldots \\
\textit{Program} & P & ::= & \cdot \mid \texttt{jump}\ P \mid x := a; P
\end{array}
$$

The accompanying parallel composition operator "|" is used to execute entire programs in parallel using *threads* which supports concurrent execution by *interleaving*. The companion machine model of LS^2 can be summarized as follows:

$$
\begin{array}{llll}
\textit{Thread ID} & I & & \\
\textit{Thread} & T & ::= & [P]_I \\
\textit{Store} & \sigma & : & l \mapsto e \\
\textit{Lock Map} & \iota & : & l \mapsto I \cup \{\_\} \\
\textit{Configuration} & C & ::= & \iota, \sigma, T_1 | \ldots | T_n
\end{array}
$$

Threads are identified using I, which may be a triple identifying *owner, machine,* and a *name* that we have not included above. All executing threads I sequentially reduce actions of a program $[P]_I$. The systems memory is modeled

using σ which maps locations l of machines to expressions e, read as "l holds e". Most notably, LS^2 uses a global lock map which allows to specify whether a particular memory location is *unlocked* or *locked* by a thread I, i.e. modifiable only through a program or action $[P]$ reduced by I. Finally, configurations C include mappings of locations, locks, as well as running threads.

LS^2 uses explicit *operational semantics* to explain *how* one configuration transitions to another. Each transition has a *rule* in the operational semantics detailing its effect as the following example shows:

$$
\begin{aligned}
&[\texttt{jump}\ P]_I \longrightarrow [P]_I \\
&[x\ \texttt{:=}\ \texttt{hash}\ e; P]_I \longrightarrow [P(Hash(e)/x)]_I \\
&\iota[l \mapsto \_], [x\ \texttt{:=}\ \texttt{lock}\ l; P]_I \longrightarrow \iota[l \mapsto I], [P(0/x)]_[I] \\
&\iota, \sigma[l \mapsto e'], [x\ \texttt{:=}\ \texttt{write}\ l, e; P]_I \longrightarrow \iota, \sigma[l \mapsto e], [P(l/x)]_I
\end{aligned}
$$

Transitions from $C \longrightarrow C'$ are written with the *redex*, i.e. the state to be changed, on the left hand side of the arrow pointing to the *reactum*, i.e. the changed state, on the right hand side. For example, when `jump` P, x `:=` `hash` $e; P$, x `:=` `lock` $l; P$, and `x :=` `write` $l, e; P$ are reduced by thread I. As expected, a `jump` P has the effect that I executes actions of $[P]$ next. Reducing x `:=` `hash` $e; P$ has the effect that I continues with $[P]$ while x is now bound to $Hash(e)$. Similarly, `lock` $l; P$ returns 0 and requires that l is not locked when it is reduced. The effect states consequence that l then maps to I and I continues with $[P]$. Each transition is then labeled with a covariant index t such that transitions can be labeled $C \xrightarrow{t_0} C' \xrightarrow{t_1} C'' \xrightarrow{\cdots} C \ldots$. The labeled transitions in combination with the new state are referred to as the *trace* of the system and t may be interpreted as *time*. We can then reason about configurations which must have happened *before* another. Lastly, Datta et al. construct a predicate logic with temporal extensions. Formulas of the logic support the usual connectives and between *general* and *action* predicates.

Action	R	::=	$\mathsf{Read(I, l, e)} \mid \mathsf{Write(I, l, exp)} \mid$
			$\mathsf{Hash(I, exp)} \mid \ldots$
General	M	::=	$\mathsf{Jump(I, P)} \mid \mathsf{Mem(l, e)} \mid \mathsf{Locked(l, I)} \mid$
			$e = e' \mid t \geq t' \mid \ldots$
Formulas	A, B	::=	$t \mid \iota \mid \top \mid \bot \mid R \mid M \mid A \wedge B \mid$
			$A \vee B \mid A \supset B \mid \neg A \mid$
			$\forall x.A \mid \exists x.A \mid A \ @ \ t$
Modal Formulas	J	::=	$[P]_I^{t_b,t_e} A \mid [a]_{I,x}^{t_b,t_e} A$

Actions reduced by threads I are presented as action predicates and are kept separate from general facts and machine specific predicates, i.e. such as equality of expressions, memory states or locks of a location. The formula A @ t is used to capture "A is true at t" [18], e.g. $\mathsf{Jump(I, P)}$ @ t is true if I reduces [`jump` P] at t. LS^2 often uses intervals over traces in the usual form using (t_1, t_2),$[t_1, t_2]$,$(t_1, t_2]$, and $[t_1, t_2)$. For intervals i, A on i is defined as A holds on all points t in i. Security and safety properties are expressed as one of two modal formulas. Formula $[P]_I^{t_b,t_e} A$ means that formula A is holds or is true whenever thread I executes P in the right-open interval $(t_b, t_a]$. A can be used to express security properties of P and may contain variables unbound in P.

4.2.2 Trusted Computing

The Trusted Platform Module (TPM) is a component of the physical machine in Figure 4.1 and implements the specification defined by the Trusted Computing Group [65]. From an abstract perspective, a TPM can be described as an extra hardware chip equipped with a public and private key pair $\{K_{TPM}, K_{TPM}^{-1}\}$ and a set of secure memory regions known as Platform Configuration Registers (PCRs) (details in subsection 4.2.3). The TPM specification requires that the private key K_{TPM}^{-1} to be kept secret and the PCRs be resistant at least against software attacks. The private key may be used to sign contents of PCRs, and if an external verifier knows K_{TPM} she can verify that the signed value is indeed a PCR value. The process of signing PCRs on a target platform and evaluation of PCRs is part of the process called remote attestation. In fact, a verifier assuming malicious software on the machine can only trust the signed PCR values, i.e.

$K^{-1}_{TPM}(PCRs)$. Hence, we need to convey facts about the target system using PCRs only and use them to record *configurations* of a platform in a trustworthy manner.

4.2.3 Platform Configuration Registers (PCR)

PCRs are treated as a special form of memory with only one dedicated command to *modify* them: `extend` l, e. Our `extend` corresponds to the command *TPM_extend* of [65] and requires that l be a PCR index or handle, and e be a hash of e. Each PCR represents a sequence $\langle sinit, v_1, \ldots, v_v \rangle$ with $sinit$ being its initial value to which values $v_1, \ldots, v_v$ must have been added *sequentially* using `extend` for each value v. The only way to reset a PCR is to reboot the physical machine for which LS^2 has a dedicated rule and general predicate. A TPM singed sequence $\langle sinit, v_1, \ldots, v_v \rangle$ lets a verifier gather that (a) the machine hasn't been restarted and (b) that `extend` l, v_1 must have happened *before* `extend` l, v_2 and so forth. PCRs and the key pair $\{K_{TPM}, K^{-1}_{TPM}\}$ form the root of trust for storage (RTS) and reporting (RTR) of a physical machine [65].

4.2.4 Root of Trust for Measurement

The RTS and RTR are completed by a root of trust for measurement (RTM) [65]. Informally, the RTM is a thread I on the physical machine which reduces or executes `extend` / *TPM_extend*. The root of trust for measurement is responsible for initiating a chain of trust measurements. When a machine is powered on, the first program executed is the *core root of trust for measurement* (CRTM). The CRTM is responsible for measuring, i.e. hash and extend, the first piece of code in a boot sequence, e.g. the BIOS. The CRTM program itself can not be measured and a verifier has to trust that a machine executes exactly the sequence CRTM whenever the physical machine is booted.

4.2.5 Measured Boot

The process of creating a chain of measurements from CRTM to the operating system is referred to as Measured Boot (Fig. 4.2). The chain of trust measure-

ments initiated by the CRTM is the continued using the following program pattern:

$$[p := \texttt{read}\ P; h := \texttt{hash}\ p; \texttt{extend}\ h; \texttt{jump}\ p] \tag{4.1}$$

We have omitted known locations l. This sequence is started by the CRTM using the BIOS code as P, and continued by the BIOS. The BIOS code reads boot loader (BL(m)) code and jumps to it, and finally, the BL(m) code reads operating system (OS(m)) code and jumps to it. In the PCR used by CRTM(m), BIOS(m), and BL(m), this produces the following sequence:

$$\langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle \tag{4.2}$$

This patten in Equation 4.1 can have an arbitrary number of programs in between as long as the pattern of reading some P, extending the hash of P, and jumping only to P is maintained.

4.2.6 Integrity Measurement Architecture (IMA)

The TCG-based Integrity Measurement Architecture (IMA) was originally proposed in 2004 [117] and has since been adopted in the security sub-system of the Linux kernel (version ≥ 2.6). Today, IMA forms the basis of many integrity verification frameworks which aim for run-time integrity verification [111], policy-based attestation [119], and is actively being used, studied, and adapted to increase overall measurement performance [79, 80, 125]. The sole purpose of IMA is to extend the chain of trust measurements from the OS well into *user-space* by measuring potentially *everything* that is mapped to system memory beyond the OS, including loadable kernel modules, applications, and configuration files. For practical reasons, we will focus on OS components such as kernel modules and *user-space* applications. IMA is part of the operating system and once the OS is running, the sequence in equation 4.2 in the PCR used to boot the OS is hashed and extended into a fresh PCR for IMA. This creates the sequence $\langle sinit, \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle\rangle$ in the PCR used by IMA. By *hooking* onto loader functionality, i.e. the part of the OS responsible for memory mapping, IMA is able to continue the pattern of reading, hashing, and

extending values into a PCR. For example, launching an application creates the sequence $\langle sinit, \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m)\rangle, \text{App}(m)\rangle$ in the PCR used by IMA. Effectively, IMA continues a Measured Boot by performing a measured launch of *user* applications.

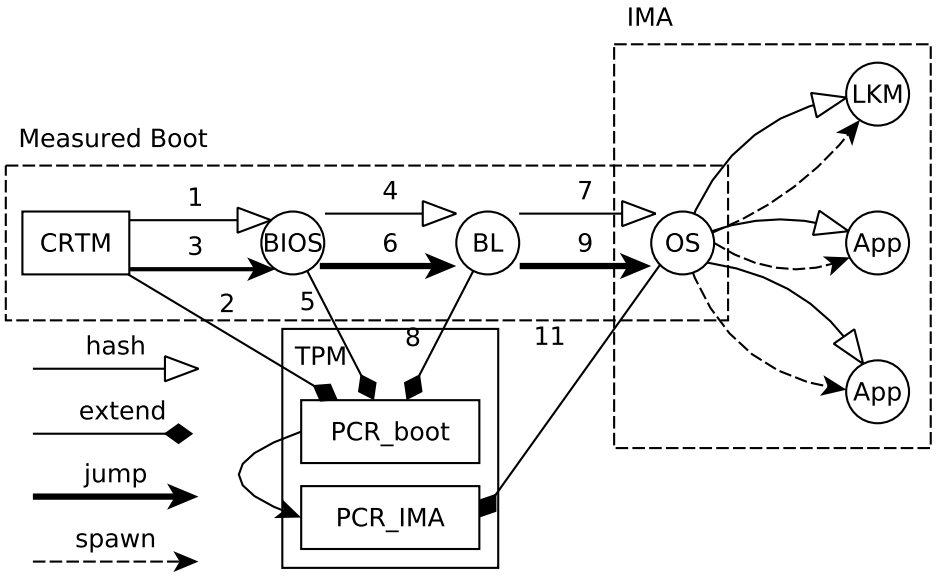

Fig. 4.2 The `read`, `hash`, `extend`, then `jump` sequence of Measured Boot ends at but includes the operating system. Instead of switching the program of $[\text{OS}(m)]_I$, I spawns new threads running loadable kernel modules [LKM(m)] and *user-space* applications [App(m)].

4.2.7 Security Properties

The following security properties have guided our formal analysis:

1. The *Integrity of a System* (such as the operating system running on a machine) is not compromised if no unauthorized modification to the system configuration can be made without the modification being recorded. We limit the scope of modifications to loading kernel modules and launch applications, i.e. events we can measure and record.

2. *Trustworthiness of Measurement* is maintained if the measurements taken reflect the system configuration with regards to its *integrity*. A version of this property forms the basis of the formal analysis in [36] but is not extended beyond loading operating system code.

3. *Forward Integrity* of the measurement log is achieved if during a *run*, i.e. between reboots, recorded events can not be deleted, removed, hidden, or overwritten. *Forward Integrity* is the main property of IMA and the reason why PCRs are used to record events.

4. *Adversary Confinement* is achieved when code supplied by an adversary is recorded before its actions can take effect. Although not explicitly mentioned, both IMA and Measured Boot inherently provide this property by recording read code into a PCR before further actions can be reduced.

5. Lastly, *Constrained Disclosure* is a desirable property of any recoding and reporting procedure. While recording any modifications of a system, we report only the modifications which can affect a systems *integrity*. For instance, when reporting on the operating systems integrity, we don't need to include reports of *user-space* applications. Similarly, when reporting the integrity of a sub-domain, we need a report of the host domain but not of other sub-domains of the system.

In this work, we focus on security property (2) and (5). Security properties (1), (3), and (4) can be achieved using properties of the TPM and a measured boot sequence [124].

4.3 Domain Specific Measurements

Separation between components and the *Principle of Least Authority* (POLA) are key ingredients of secure systems [37] and compartmentalization is generally regarded as a good engineering practice [71]. Typically, modern computer systems have different abstraction and security layers with different privileges such that components in the same layer may not interfere with each other without the supervision of a lower layer and higher privileged component.

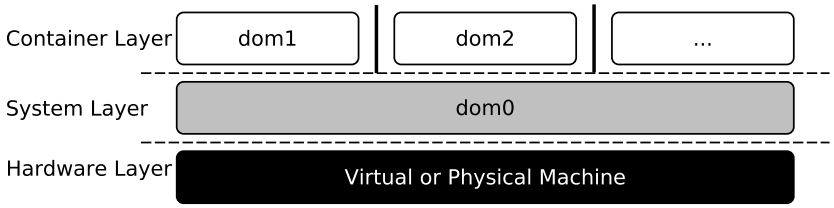

Fig. 4.3 *LS*[3] *domains* and *sub-domains* in relation to the machine which may be physical or virtual. Dashed lines indicate stratification using different *layers* and solid lines between *sub-domains* indicate compartmentalization using containers.

We refer to this more generally as system-*stratification* and component-*compartmentalization* (Fig. 4.3). Containers as a summarizing concept for operating system level virtualization techniques allow us to create *virtual user-spaces* to run applications directly on the host operating system. As it is with containers, our virtual user-spaces are endowed with restricted and possibly minimal access to shared resources. Containers are treated as virtual environments in user-space. Consequently, the host operating system controlling *all* system resources will be represented as a *domain*. Containers, dependent on a host operating system, are referred to as a *sub-domain*.

In our review of *LS*[2], we have described its system model using *Machines*, *Threads*, *Storage*, and *Locks*. Threads in *LS*[2] are associated to machines using an identifier structure I which contains a reference to a machine identifier m. In order to reason over which thread must have caused an effect on a storage location such as PCRs, *LS*[2] uses locks as a reasoning tool. Locks are necessary in *LS*[2] as there is no stratification and no privilege separation, i.e. all threads, including those executing adversarial code can access any data structure unless it is explicitly locked.

LS[3] adds *domains* and *sub-domains*. First, we associate threads with a domain and for now, there is only $dom0$ and dom_n ($n > 0$). Dom_0 is a domain model borrowed from Xen [12] which includes the operating system kernel, kernel level threads, and possibly privileged user-space with management interfaces for a service provider. Containers, or *virtual* user-spaces, are sub-domains dom_n with $n > 0$. To reflect a threads association with a particular domain we extend the thread identifier I to include a domain identifier d. With n as its

given name, m to indicate the machine, and d its domain or privilege level on m. Threads I in LS^3 run programs LS^2 programs P including all actions described in [36]. However, LS^2 only has locks, and locking a location l for each thread I is not practical. Instead, we represent container semantics by partitioning our machine's storage according to the domains. We say that a machine m has storage of type $disk$, ram, or pcr. Each type of storage can be partitioned by adding a domain name or storage ID after it.

Thread ID	I	::=	(n, m, d)
Thread	T	::=	$[P]_I$
Machine	m		
Storage Type	sty	::=	disk | ram | pcr
Location	l	::=	$m\ .\ sty\ .\ ref$
Store	σ	:	$l \mapsto e$
Allocation	α	:	$l \mapsto d$
Configuration	C	::=	$\alpha, \sigma, T_1 \mid \ldots \mid T_n$

For example, a *container-image* which holds a programs P to be executed in the corresponding container domain d_n ($n > 0$) is stored at location $l.disk.dom\_n.P$. In this case ref would be $dom\_n.P$ where dom_n is the partition in which we would look for P. Similarly, storage types ram and pcr can be partitioned. The partitioning is enforced in the operational semantics for `read` and *write*:

$$\alpha[l \mapsto I_d], \sigma[l \mapsto e'], [\texttt{write}\ l, e; P]_I \longrightarrow \alpha, \sigma[l \mapsto e],\ \ldots$$
$$\alpha[l \mapsto I_d], \sigma[l \mapsto e], [x\ :=\ \texttt{read}\ l; P]_I \longrightarrow \alpha, \sigma[l \mapsto e],\ \ldots$$

The system property which states that domain dom_0 controls all resources can be expressed by adding the default mapping to our allocator:

$$\alpha\ :*\mapsto\ dom_0$$

Allowing multiple entries in α for a location l allow us to model logical sub-partitions for dom_n under the control of dom_0. This also gives us the basis for reasoning about *effects* in the system. For example, an expression written in a location allocated to some domain dom_n could only have been written by some

thread in dom_0, i.e. the host system, or by a thread of dom_n but not by a thread of another untrusted domain dom_m, $m \neq 0, n$.

4.3.1 Measured Boot and IMA

The first application of LS^3 is the chain of trust measurements from system boot to application launch which we do in two steps: in step one, we discuss measured boot using our machine model and in step two, we extend the measurements by explicitly adding threads to our system.

To express measured boot, we need a predicate for a machine start, reboot, or reset. Following LS^2 we introduce the *general predicate* $\mathsf{Reset}(m, I)$ which states that machine m is restarted producing the initial thread I. The corresponding rule in the operational semantics states that all volatile memory, i.e. RAM and PCRs, is reset and allocations on m are removed:

$$C \longrightarrow \alpha \setminus (m.l.*), \sigma[m.pcr.* \mapsto sinit] \setminus (m.l.ram.*), \\ (T_1|...|T_n) - T_m \mid [\mathrm{CRTM}(m)]_I)$$

Upon a reset, all threads on m are removed and replaced by *just* one thread $I_{(n,m,dom_0)}$ executing exactly the program CRTM(m). The program CRTM(m) corresponds to the only piece of trusted code that is executed right at the beginning. When a machine in LS^2 is reset, the same thread I is created but it is assumed that there may be more than this thread after $\mathsf{Reset}(m, I)@t$. Hence, in order to reason over which thread extended the first piece of code into a PCR, LS^2 resorts to global locks. Threads in LS^3 are created explicitly which brings us the following lemma:

Lemma 1 (Replacing locks by launching only I)**.** *When a machine is reset in* LS^3 *at time t_R, only a thread I is created in the default domain dom_0. For any location l allocated to dom_0 in α at time t_R, we have*

$$\forall t', t''.(t_R < t' \leq t''), \nexists e, \mathsf{Mem}(\mathsf{l}, \mathsf{e})@t', \neg\mathsf{Write}(\mathsf{I}, \mathsf{e})@t''.$$

The following program sequence (first shown in Equation 4.1) is used to bootstrap the operating system in Figure 4.2:

$$[p := \texttt{read}\ P; h := \texttt{hash}\ p; \texttt{extend}\ h; \texttt{jump}\ p] \tag{4.3}$$

By replacing the generic sequence above with P as BIOS(m), BL(m), and OS(m) we can construct a chain of measurements from CRTM(m) to OS(m). That is, if all programs in the sequence are *exactly* those programs and if they are executed fully. Programs are typically loaded from the $disk$ of m which is generally regarded as untrusted, meaning that anything we read from the disk is potentially adversarial code. In order to establish that the sequence in the PCR used by P, i.e. PCR_boot in Figure 4.2, implies that the correct boot sequence has been executed, we need to reason step by step starting at CRTM(m). We get proof of the execution of CRTM(m) by finding the BIOS(m) logged in the PCR.

However, by finding BIOS(m) in the PCR a verifier can only infer that action `extend` has been reduced of CRTM(m). In oder to establish that CRTM(m) has been *fully* and *exclusively* executed, i.e. no further code, we need to use the predicate Jump(I, P). Using Jump(I, P) as a tool for reasoning [36] lets us connect proof of the execution of BIOS(m) with a property we wish to prove about I executing CRTM(m). By finding BL(m) we get prove that CRTM(m) has not only measured and recorded BIOS(m) but must have also *jumped* to it. This lets us prove an important security property: *Trustworthiness of Measurement*.
This style of reasoning using the predicate Jump(I, P) can be continued infinitely but will always *exclude* the last element because we are not able to prove *full* or *exclusive* execution [36]. Hence, the sequence in Equation 4.1, without assumptions about the execution environment, lets us prove to a verifier that at least some portion of the OS(m) actions has been reduced.

$[\text{OS}(m)]_I$ represents known minimal operating system kernel running in dom_0 on m in control of all resources of that machine. It becomes critically important that any *runtime* additions to this kernel are recorded in order to provide or verify the systems integrity [117] (Section 4.2.7). IMA is designed to fill this gap and extend the chain of trust measurements further into a running system. As mentioned earlier, to model this properly LS^3 lets us add threads *explicitly*

using the action `spawn` P along with the according operational semantics rule and general predicate `Spawn(I, P)`

$$C, [\texttt{spawn}\ P;\ Q]_I \longrightarrow \{T\} \cup T_J, [Q(\ldots)]_I, [P]_J, \\ (dom(J) = dom(I))$$

Where we specify that the expression of action `spawn` is a program (e:P), the newly created thread J executes P, and that the caller I continues with the remainder of her program Q. As such, `spawn` corresponds to allocating executable memory for a thread in dom_0 containing P, mapping P to some $m.ram.J$, assigning J to the domain of the caller, and scheduling J for execution.
This construct allows us to continue reasoning over the actions of program OS(m) and how IMA continues the chain of trust measurements in a trustworthy manner. For convenience, we make the IMA kernel code itself measurable by defining the following programs for OS(m) and LM(m), i.e. the OS and the loader module (LM):

$$[pcr\ :=\ \texttt{read}\ PCR\_boot; h\ :=\ \texttt{hash}\ pcr; \\ \texttt{extend}\ h, PCR\_IMA;\ \texttt{spawn}\ l.ram.OS.LM]_{\mathrm{OS}(m)}$$

and finally LM(m):

$$[p\ :=\ \texttt{read}\ P; h\ :=\ \texttt{hash}\ p; \\ \texttt{extend}\ h, PCR\_IMA;\ \texttt{spawn}\ P]_{\mathrm{LM}(m);\ldots}$$

The OS(m) is responsible for *linking* the PCR values used to boot the machine to the PCR which will be used to record loadable kernel modules (LKMs) and applications by the program LM(m). For simplicity, we say that LM(m) is a subprogram of OS(m) and that by measuring and recording the operating system, we also get a hash of LM(m). The PCR configuration of m *before* the LM(m) visible to a verifier is then:

$$\langle sinit, \mathrm{BIOS(m)}, \mathrm{BL(m)}, \mathrm{OS(m)} \rangle_{PCR_{boot}}$$

$$\langle sinit, \langle sinit, \text{BIOS(m)}, \text{BL(m)}, \text{OS(m)} \rangle \rangle_{PCR_{ima}}$$

Based on this the verifier can only infer that OS(m) must have reduced the actions before [spawn], but not that the loader module is in fact running now. Once the first LKM or application has been loaded, the PCR used by IMA will change to:

$$\langle sinit, \langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m) \rangle, \text{P(m)} \rangle_{PCR_{ima}}$$

As before, P(m) could be any *untrusted* program on the disk, and may or may not allow us to reason about whether or not it is being executed. However, using the predicate and the rule Spawn(I, P) similar to a *jump* to link knowledge of the OS(m) and the fact that only some thread running LM(m) could have extended PCR_{ima} allows us to prove to a verifier that IMA is running and performing measurements.

4.3.2 Containers and Domain Specific Measurements

Physical co-residency is the center of hardware-level side-channel attacks in the cloud [73, 70]. As a first step in these types of attacks, the adversarial tenant needs to confirm the cohabitation with the victim on the same physical host, instead of randomly attacking strangers [70]. For this reason, we argue that it is not acceptable to use the IMA code in the operating system loader component shared among domains.

In LS^3, containers are represented by *container-images* on a machines disk, and by a container-engine sub-system part of the operating system. The container engine is *trusted* to maintain isolation in the sense of performance and protection. In short, we assume that a container-engine component of the operating system securely manages containers. Before a container is run, we assume that the store σ and allocation α have been prepared: both $m.disk$ and $m.ram$ receive a partition dom_n, ($n > 0$). The disk region $m.disk.dom_n$ corresponds to the container image and the memory is used to execute programs shipped with the container. The allocator is prepared to enforce this accordingly by adding $m.ram.dom_n \longrightarrow dom_n$ and $m.disk.dom_n \longrightarrow dom_n$.

To allow for *spawning* threads running programs of the container we add an *overloaded* command `spawn` P, d to our system which adds a parameter for the target domain of the newly scheduled thread.

$$C, [\texttt{spawn}\ P, d;\ Q]_{I_{dom_0}} \longrightarrow \{T\} \cup T_J, [\ldots]_I, [P]_{J_d}$$

Most importantly, the command for spawning new processes in domains can only be reduced by a thread of dom_0, i.e. the host operating system. Modifying the loader code to use `spawn` P, d is intuitive but violates the *constrained disclosure* property: Instantiating LM(m) with `spawn` P, d would necessarily result in PCR_{ima} holding hash values of *all* programs ever spawned or loaded on m between a reboot. A remote verifier wishing to attest the system would learn not only about programs of her container but also about other programs which may currently be running on the system. Measuring *everything* into *one* record invalidates any efforts to randomize the distribution of services and acts like a system layout reverser for an attacker.

We propose an Enhanced Integrity Measurement Architecture (EIMA) which extends IMA measuring newly launched programs in a way which respects system isolation and partitioning. We claim that EIMA produces domain specific measurements. When a new container is set up on our system we also partition PCRs. Before adding the allocation entry we *link* the PCR for a new domain (dom_n) with the PCR of the host system (dom_0):

$$\langle sinit, \langle \ldots \rangle_{PCR_{dom_0}} \rangle_{PCR_{dom_n}} \tag{4.4}$$

After that, we add the entry $m.pcr.dom_n \longrightarrow dom_n$ to our allocator α. Bootstrapping the container environment after that can be done by spawning an *init*-like process from the container image.

$$\begin{aligned}[p\ :=\ &\texttt{read}\ m.disk.dom_n.Init; h\ :=\ \texttt{hash}\ p;\\ &\texttt{extend}\ h, pcr_{dom_n};\ \texttt{spawn}\ P, dom_n]_{\mathrm{ELM}(m);\ldots}\end{aligned}$$

The modifications to the *enhanced loader module* (ELM(m)) are minimal: we read a program Init(…) from the container image located at $m.disk.dom_n$, hash

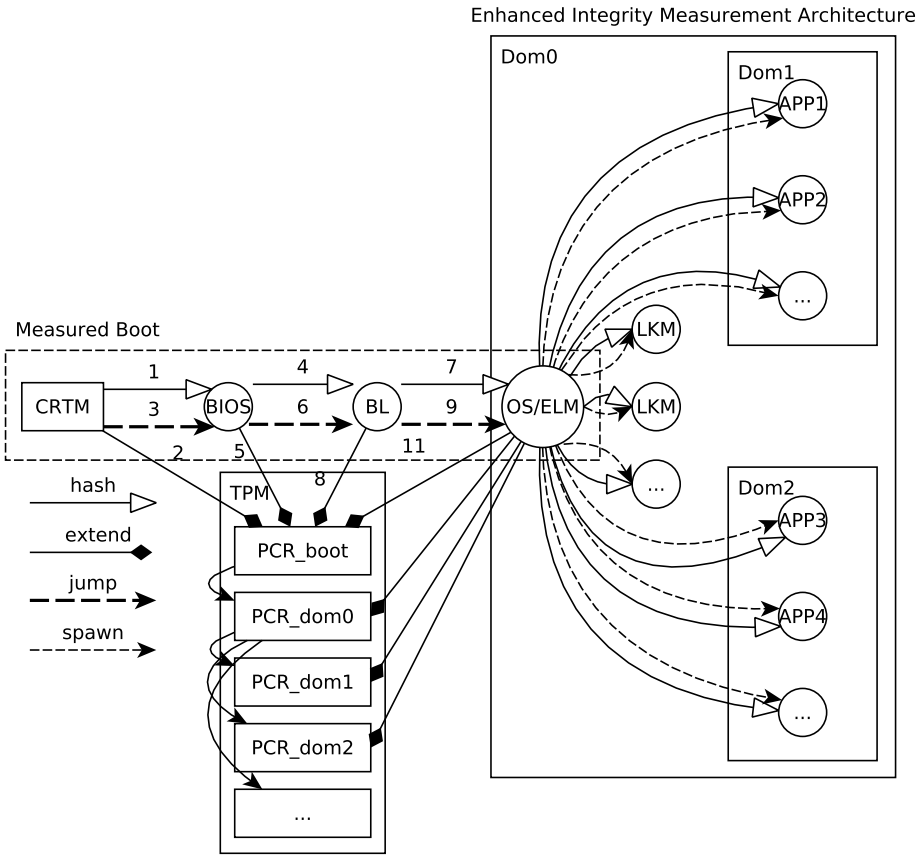

Fig. 4.4 LS^3 Domain Specific Measurements using our Enhanced Integrity Measurement Architecture (EIMA) loader module (ELM) (ELM(m)). The shared system loader program ELM(m) is responsible for spawning new threads. ELM(m) is extended to record programs according to the domain that will be assigned to the thread executing them.

it and extend it to pcr_{dom_n} before spawning a new process running the program in dom_n.

Figure 4.4 shows three domains: dom_0 as the domain created after a machine reset and dom_1, dom_2 as sub-domains or isolated *user-spaces* hosting different applications as per our introduction. Having a fully booted host system running containers as in Figure 4.4 produces the following PCR configurations:

$$\langle sinit, \text{BIOS}(m), \text{BL}(m), \text{OS}(m) \rangle_{PCR_{boot}} \tag{4.5}$$

$$\langle sinit, \langle ... \rangle_{PCR_{boot}}, \text{LKM}(m), \ ... \rangle_{PCR_{dom_0}} \tag{4.6}$$

$$\langle sinit,\ \langle ... \rangle_{PCR_{dom_0}},\ \text{App1}(m),\ \text{App2}(m), ... \rangle_{PCR_{dom_1}} \tag{4.7}$$

$$\langle sinit,\ \langle ... \rangle_{PCR_{dom_0}},\ \text{App3}(m),\ \text{App4}(m), ... \rangle_{PCR_{dom_2}} \tag{4.8}$$

Hence, each container can be remotely verified using only *relevant* information, i.e. system boot, host operating system, and sub-domain measurements. Consequently, a service provider hosting containers can offer *trustworthy measurements* for containerized environments without having to reveal information about other, isolated work-loads running in other containers using different partitions.

4.3.3 EIMA Achieving Security Properties (2) and (5)

In this subsection, we first prove the trustworthiness of the measured boot sequence. We then show how EIMA achieves constrained disclosure which also implies that IMA's integrity measurements are trustworthy.

Definition 1 (Measured Boot Sequence)**.**

$$\begin{aligned}\mathsf{SRTM(m,t)} = \exists t_S, t_B, t_{BL}, t_O, I\ .\ (&t_S < t_B < t_{BL} < t_O < t)\\ &\wedge \mathsf{Restart(m,I)}@t_S \wedge \mathsf{Jump(I}, \mathit{BIOS}\mathsf{(m))}@t_B\\ &\wedge \mathsf{Jump(I}, \mathit{BL}\mathsf{(m))}@t_{BL} \wedge \mathsf{Jump(I}, \mathit{OS}\mathsf{(m))}@t_O\\ &\wedge \neg\mathsf{(Restart(m))\ on}\ (t_S, t] \wedge \neg\mathsf{(Spawn(I))\ on}\ (t_S, t]\\ &\wedge \neg\mathsf{(Jump(I))\ on}\ (t_S, t_B) \wedge \neg\mathsf{(Jump(I)\ on}\ (t_B, t_O))\end{aligned}$$

Theorem 1 (Trustworthiness of SRTM Integrity Measurement)**.** *Let* $seq = \langle sinit, \mathit{BIOS}(m), \mathit{BL}(m), \mathit{OS}(m), \mathit{APP}(m)\rangle$, $\forall t.\ \mathit{Restart}(m, I)@t, \alpha : m.pcr_{boot} \longrightarrow dom_0$, *and* $\mathsf{Mem(m.pcr.boot, seq)}@t$ *then* $\mathsf{SRTM(m,t)}@t$.

Proof. The proofs and programs in for CRTM(m), BIOS(m), BL(m), and OS(m) in LS^3 follow Datta et al., except that we use Lemma 1 instead of protected PCRs (see discussion before Lemma 1).

■

So far we have shown that based on the sequence in PCR_{boot} the system must have performed a measured boot sequence to bring up the OS. We now show the trustworthiness of (E)IMA integrity measurements.

Definition 2 (Domain Launch using ELM(m,t))**.**

$$
\begin{aligned}
&\mathsf{DomainLaunch(m,t)} = \exists t_E, t_{LKM_1}, t_{LKM_2}, t_{App1}, t_{App2}, I, J\ .\\
&\quad (t_E < ((t_{LKM_1} < t_{LKM_2}) \| (t_{App1} < t_{App2})) < t)\\
&\quad \mathsf{SRTM(m,t)}@t \wedge \mathsf{Spawn(I, J}, \mathit{ELM}\mathsf{(m))}@t_E\\
&\quad \wedge \mathsf{Spawn(J, LKM_1, dom_0)}@t_{LKM1}\\
&\quad \wedge \mathsf{Spawn(J, M_2, dom_0)}@t_{LKM2}\\
&\quad \wedge \mathsf{Spawn(J, App1, dom_1)}@t_{App1}\\
&\quad \wedge \mathsf{Spawn(J, App2, dom_1)}@t_{App2}\\
&\quad \wedge \neg\mathsf{Spawn(dom_0)\ on}\ (t_E, t_{LKM_1})\\
&\quad \wedge \neg\mathsf{Spawn(dom_0)\ on}\ (t_{LKM_1}, t_{LKM_2})\\
&\quad \wedge \neg\mathsf{Spawn(dom_0)\ on}\ (t_{LKM_2}, t] \wedge \neg\mathsf{Spawn(dom_1)\ on}\ (t_E, t_{App1})\\
&\quad ...
\end{aligned}
$$

Theorem 2 (Trustworthiness of EIMA Integrity Measurement)**.** *Let* $srtm = \langle sinit, BL(m), OS/ELM(m)\rangle$, $dom_0 = \langle sinit, srtm, LKM_1(m), LKM_2(m), ...\rangle$, $dom_1 = \langle sinit, dom_0, App1(m), App2(m), ...\rangle$, $\alpha : m.pcr_{dom_0} \longrightarrow dom_0, m.pcr_{dom_1} \longrightarrow dom_1$, *and at time* t *we have* $\mathsf{Mem(m.pcr.srtm,\ srtm)}$, $\mathsf{Mem(m.pcr.dom_0,\ dom_0)}$, *and* $\mathsf{Mem(m.pcr.dom_1,\ dom_1)}$, *then* $\mathsf{DomainLaunch(m,t)}$.

Proof. The proof follows the basic proof of Measured Boot with the same assumptions. Using the sequence in $m.pcr.srtm$ combined with the fact that only the initial thread I executing $\mathrm{CRTM}(m), \mathrm{BIOS}(m), \mathrm{BL}(m), Q \in \mathrm{OS}(m)$ recorded and then spawned $\mathrm{ELM}(m)$ at some time t. The extend operation in $\mathrm{OS/ELM}(m)$ gives us the necessary proof that $\mathrm{OS}(m)$ has in fact reduced `spawn` $\mathrm{ELM}(m)$ at t_E. $\mathrm{ELM}(m)$ continues the measured launch sequence. ∎

4.4 Summary

In this chapter, the design and verification of an enhanced integrity measurement architecture (EIMA) was presented. EIMA addresses the trustworthiness and constrained disclosure of integrity measurements for containerized systems. Our development and design was guided by a precise formal model of strati-

fied systems. The formal model was conceived by adding required constructs for domains and domain specific measurements to an existing and established formal model. The additions allowed a proof of a trustworthy measurement architecture from system boot all the way up to containerized applications. The proof of EIMA has also shown the security properties of the commonly used IMA. However, EIMA does not reveal information about a target container to other untrusted tenants on the same system. Our future work will have to address more detailed interactions between otherwise isolated domains and sub-domains via system calls and privileged administrator commands. The verification of properties which were deferred to the TPM is a pressing task and a suitable remote attestation protocol as outlined in Chapter 3 is still needed.

Chapter 5

Bootstrapping Trust in a Virtual Platform

The Trusted Platform Module (TPM) can be used to establish trust in the software configuration of a computer. Virtualizing the TPM is a logical next step towards building trusted cloud environments and providing a virtual TPM to a virtual machine promises a continuation of trusted computing concepts. The association between a virtual TPM and a virtual machine is a critical concern. In this chapter, we show that a "trusted" virtualized platform may fall victim to a *Goldeneye* attack. We put forward a formal model for virtualization systems and virtual trusted platforms in Section 5.3.2. We pair this with a model for establishing trust in a virtualized platform following conventional reasoning over trusted computing systems in Section 5.3.3. We show that if a *Goldeneye* attack is successful, it would allow a verifier to establish trust in an untrustworthy platform. We discuss attack vectors in related work Section 5.4.1 and possible solutions which would mitigate *Goldeneye* in Section 5.4.2. The contributions of this chapter are detailed in Section 5.1.1 and the results are published in [82].

5.1 Introduction

Before entrusting a computer with a secret, a client needs some assurance that the computer can be trusted [105]. Without such trust, performing tasks

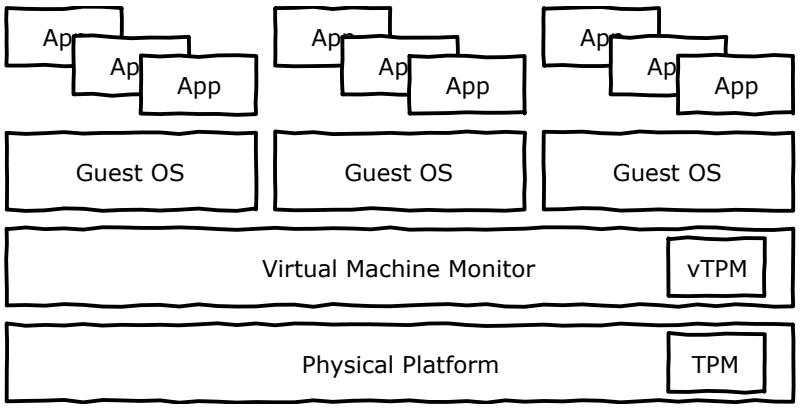

Fig. 5.1 A sketch of a virtualization system equipped with a TPM which hosts virtualized systems with virtual TPMs. A guest OS running in a virtual machine and composed with a virtual TPM is referred to as a trusted virtualized platform.

involving secrets or sensitive data securely is currently not possible. Practical trust establishment technologies are crucial for the secure operation, proliferation and adoption of current and future services [70]. In fact, the need to bootstrap trust is most evident when using hosted and remote computing systems [48, 95, 87, 116].

For example, a cloud provider P offers to host an instance of a customer's OS of choice in a virtual machine which the customer may freely use. This allows customer C to run certain applications forming a service. Conveniently, C does not have to maintain any hardware. Customers need to trust P's virtualization system (Fig. 5.1) to provide important properties: isolation between their system and other customers as well as access to interfaces, keys, storage objects, and resources. If a customer wants to establish trust in her services, i.e. the guest OS and applications in Fig. 5.1, then an associated vTPM should contain information about the software state of that part of the system. Unless C trusts P implicitly, C will also have to establish trust in the virtualization system which hosts her workload, i.e. the machine and virtual machine monitor in Fig. 5.1. Lastly, cloud customers C hosting their service on P's virtualization system may need to prove compliance to other parties as well. Being able to establish trust in hosted services and the underlying virtualization system becomes crucial if the hosted service needs to be trusted. A provider should be able to demonstrate

continuous security compliance and prove the integrity of her provided service to customers. Alternatively, customers may want an option to monitor the system hosting their services.

One way to establish trust in a computer is to use the secure hardware such as the Trusted Platform Module (TPM) (Section 2.3.3) to record and report its software state. Using a process called remote attestation (Section 2.3.5) this information can be securely conveyed to a remote party. Given the software state, the client (or any other agent) can decide whether the platform should be trusted—assuming that a trustworthy software state and configuration is known to the client. Similar forms of this approach are the basis of many practical security features of modern operating systems. Microsoft operating systems use the TPM for BitLocker and protecting and limiting the use of cryptographic keys [90]. Similarly, Linux systems offer secure boot features and integrity subsystems which also use the TPM as hardware anchor.

In short, with appropriate software support, the TPM can be used to measure and record each piece of software loaded [117] and securely convey this information to a remote party [87, 19]. Consequently, the TPM can be used to effectively establish trust in the software configuration of a machine. This can in principle be applied to a virtual machine monitor using a physical TPM. It can also be applied to its guests with the support of a virtual TPM [34, 120] as shown in Fig. 4.1. The virtual machine associated with a virtual TPM form what is called the virtualized trusted platform [56].

The question remains: How do we bootstrap trust in the association between a virtual machine and a vTPM? Currently, the specifications for "virtualized roots of trust" and "trusted virtualized platforms" as well as the academic literature offer no answer [66, 56, 14, 34, 120]. Instead, it is assumed that a client has prior knowledge of the relevant virtualization system and its TPM. Supposedly, a client can then "verify" her way up and determine, through a few recursive schemes that a particular vTPM is associated to her VM. While this assumption conveniently suspends the issue and allows higher level discussions, it never actually solves it. Practically a client knows nothing about the (1) virtualization system, (2) the physical TPM, or (3) the vTPM associated to her virtual machine. Unfortunately, without a way to assert that the VM is in fact associated to a

particular vTPM and vice versa, any verifier can be tricked into establishing trust in an untrusted system.

We exploit the lack of a verifiable association in our *Goldeneye* attack. The attack leads a verifier to establish trust in an untrustworthy VM and establish trust in untrustworthy virtualization system. We use our *Goldeneye* attack as a vehicle to emphasize the need to bootstrap trust in a trusted virtualized platform by asserting and allowing the remote verification of the association between VMs and their vTPMs.

5.1.1 Contribution

In this chapter, we make the following contributions:

- We highlight the importance of the VM-vTPM association by introducing and demonstrating *Goldeneye* which uses vTPMs *against* verifiers.
- We formally define a virtualization system with vTPMs (using bigraphs) and provide a trust model which captures our attack on the VM-vTPM association (using predicate logic).
- We show how related work [14, 34, 120] falls victim to *Goldeneye* and we discuss solutions to address the issue of associating VMs and vTPMs.

No straightforward solution appears to preclude *Goldeneye* entirely and we suggest improvements for virtualization systems which aspire to provide virtualized trusted platforms for their clients.

5.2 Background and Prior Work

This section will briefly summarize relevant background terminology including virtualization systems and the Trusted Platform Module (TPM), both physical and virtual.

We make our definitions clear and easy to translate to systems by adopting relevant terminology and descriptions from protection profiles [102] and Trusted Computing Group (TCG) specifications [132]. Our contribution is not the

description itself (similarity to other definitions is intentional and desirable) which is repeated for the reader's convenience. We annotate definitions to lay out how elements are composed to form systems as a primer for our formal concept notation later on.

5.2.1 Virtualization System

The virtualization system is a software system which enables multiple independent computing systems to execute on the same physical machine [102, 107]. The virtualization system in Fig. 5.1 consists of the physical platform (hardware), the virtual machine monitor / hypervisor, a management system and other helper components. The definition of a virtualization system is *relative* to the virtualization target or virtual environment. In this chapter, the virtualization system is the software system which supports virtual machines running guest operating systems. However, if we would like to describe OS-level virtualization, the virtualization system would include all components required to reach the desired level of virtualization. The virtualization system is treated as the all encompassing abstraction enabling the execution of multiple independent computing systems on the same physical machine. A virtualization system also supports the virtual platform. The virtual platform, in case of Fig. 5.1 is the abstraction around the guest OS. Similar to a trusted platform, i.e. hardware + TPM, we define the virtual trusted platform as virtual machine + virtual TPM [56]. We give an inductive definition of a virtualization system and highlight the composition of the elements in the process.

Process (App)

In our model, processes and apps are instances of programs. Each process is said to have a set of instructions (code), a finite address space (memory), a state structure, and resource descriptors. We assume that process are isolated from other process in the sense that one process can not directly modify resources owned by another. The process structure in general is defined by an operating system. In our model, there is a one-to-one correspondence between processes and apps and we use them interchangeably. Running apps as processes are

supervised by an operating system. The security properties of processes and apps are relative to and best explained through an operating system.

Operating System

An operating system provides the *process* abstraction and runs programs. An operating system itself is a program with the sole purpose to control, abstract, and multiplex hardware for the purpose of running multiple programs as processes on the same machine. The minimal operating system is referred to as the *kernel* which provides the process abstraction, virtual memory, scheduling, and inter-process communication. The operating system may also provide a number of other services which are not necessarily part of the kernel. We make no assumptions about the kind of operating system. Instead, we allow modelling an operating system by either defining additional functionality as part of the kernel or we model it using additional processes which implement functionality on top of a kernel. As an element of our model, the OS abstracts a hardware interface and provides a process abstraction.

Virtual Machine

A virtual machine is an environment with hardware interfaces in which a program such as an operating system may execute [102]. Similar to a process, each virtual machine is said to own some memory partition, structures for the VM state as well as other resources. Operating systems running on a virtual machine are referred to as guest operating systems (Fig. 5.2.1). The concept of a virtual machine is relative to and best explained through the idea of a *virtual machine manager*.

Virtual Machine Manager (VMM)

The virtual machine manager provides a virtual machine for other programs, e.g. guest operating systems, which is essentially identical to a physical machine [107]. Despite allowing for a guest OS to run on the same machine, the virtual machine manager is in complete control of the machine and its resources. The minimal set of VMM functionality facilitating virtual machines is referred

to as the *hypervisor*. The hypervisor provides the virtual machine abstraction and acts as a dispatcher, resource manager, and if necessary interpreter for certain instructions. VMMs may also provide additional components such as drivers, emulators, virtual devices and management systems. Further functionality can be modelled by including it as part of the hypervisor or using virtual machines or processes—depending on the *type* of hypervisor we wish to model. We consider two *types* of hypervisors in our model. Type-I or *bare-metal* hypervisors expect a hardware interface and provide virtual machines. Type-II or *hosted* hypervisors are part of an operating system which abstracts the hardware while hypervisors provide the virtual machine abstraction. However, both types provide a (virtual) machine (i.e., hardware) interface in our model.

5.2.2 Trusted Platform Module

A Trusted Platform Module (TPM) [64] (Section 2.3.3) is a hardware, firmware, or virtual device which acts as a secure co-processor and supports securing machines in a number of ways: it can securely generate, store, and apply symmetric and asymmetric keys internally. A TPM can certify internal keys based on its root Endorsement Key (EK) [130] which is typically signed by the device or platform manufacturer. For simplicity, we say that TPMs are uniquely identified by a key pair $\{K_{TPM}, K_{TPM}^{-1}\}$ [105]. Such root keys never leave the TPM and are therefore considered *secure* against all software attackers including compromised operating systems and highly privileged admins.

A particularly useful TPM feature is a special type of register called Platform Configuration Register (PCR). TPMs have a number of them and they essentially provide an append-only update operation which allows us to record transitions in the systems state in a concise way. The process of recording, or appending, system states, if continued properly, creates a "chain of trust measurements" which allows PCRs to reflect the boot process, launched applications [117], and internal states of a system [83]. Signing the recorded system state using an internal key produces a *quote*. The PCR values along with the quote can be securely conveyed to a (remote) verifier. Based on the PCR values, a verifier may check the system state and make judgments (e.g., are whether a machine is

trusted or not). This process is referred to as remote attestation [64, 57, 19, 87]. Overall, we say that if we trust the TPM, we are able to establish trust in the state of a (software) system.

virtual TPM

In our model, all machines have TPM devices as separate chip or part of their firmware. TPMs, unfortunately, have no support for virtualization and are generally not shared among operating systems (hosts and guests). Instead, virtual TPMs are employed to serve as roots of trust for virtual machine. All virtual TPMs are essentially programs which need some execution environment. Their security directly depends on the environment and must be architecturally isolated from a guest operating system so as to support guest facing TPM use-cases (e.g., key storage outside guest OS and append only PCR updates).

Beyond architectural soundness[1], the vTPM must be implemented in a way that associates it to a VM the same way a real TPM is associated to a machine. On a real machine, this association can be safely assumed as the TPM is typically part of the chip-set. However, a vTPM is an element of the system and not implicitly associated to a VM. This weakness is exploited in our *Goldeneye* attack.

5.3 Goldeneye Attack

The goal of a verifier is to establish trust in a VM and whatever runs in it using trusted computing technology. To this end, we need some fundamental trust assumptions outside of the typical trusted computing framework. The first assumption is that the verifier, or agent acting as a verifier, has a trusted device with which she can connect to a VM. Examples for this would be a local, trusted device which is used to connect to a service interface or create and manage VMs. An alternative to a trusted verifier setup would be using a trusted third party which can help a curious client with a trust decision. Further, we need agents

[1] We have an example of an *unsound* construction later on but we omit the formal definition here.

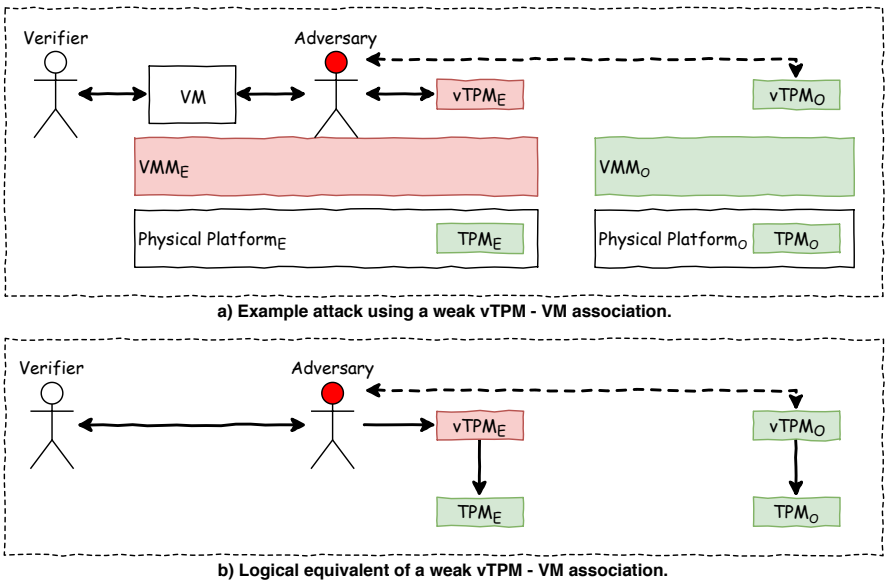

Fig. 5.2 Sketch of our *Goldeneye* attack variant 2. The adversary has control over the vTPM-VM association. An adversary uses the vTPM (vTPM$_O$) to trick a verifier into establishing trust in another virtualization system (VMM$_O$).

to be able to trust someone (at least themselves) to vouch for the integrity of some system element. Without such assumptions, we can show that it may be hard to satisfy a general, secure cloud computing use-case. In summary, we assume that

1. verifiers and their setups are trusted and
2. trusted agents can vouch for other agents and system elements.

Practically, these two assumptions allow us to describe the cloud provider P as *semi-trusted*, i.e. its machines are trusted but the software may get compromised or misconfigured. Using (1) and then (2) we can state that verifiers trust cloud providers and that cloud providers vouch for the security of their machines, respectively. Consequently, we may at any point trust a physical platform and it's TPM. This and the overall complexity of our system and the fidelity of our model is what distinguishes *Goldeneye* from similar attacks like cuckoo [105]. Our attack succeeds on a physically secure machine.

5.3.1 Informal Description

In a *Goldeneye* [2] attack, an adversary effectively controls the vTPM associated with a virtual machine. In this attack, an adversary with some control over the virtualization infrastructure may swap out the vTPMs used to establish trust in a VM.

Setup

Figure 5.2 shows this attack in frame a) with a VMM_E which is the VMM expected by the client, the red coloring indicates that this VMM may not be acceptable because it does not provide necessary security properties or may be used to violate them. The vTPM is the VM's interface to the underlying trusted computing infrastructure. A verifier with access only to the VM relies on a vTPM to indicate the virtualization system correctly. To the right of VMM_E is another virtualization system which is part of the cloud infrastructure. The system to the right is virtually identical except that it runs a secure virtualization system. We simply name it VMM_O to which the vTPM_O points (O for *other*). An adversary can now, through a variety of channels (Section 5.4), funnel VM-vTPM communication to vTPM_O. The possibility of a *Goldeneye* attack has an effect on a number of security mechanisms. We focus on establishing trust in the software state of the virtualization system and the VM in this work. We identify two main variants of *Goldeneye* and discuss them below.

Variant 1 - changing virtual Root of Trust

Exploiting the VM-vTPM association on the same virtualization system enables the first variant of *Goldeneye*. The goal of the adversary in this variant is to allow trust establishment in a potentially untrustworthy VM. The variant is executed by changing the (virtual) root of trust associated with a virtual machine. An adversary with the ability to clone or fabricate vTPMs, can freely drop or inject certain TPM commands such as PCR updates. Furthermore, it would allow adversaries to install malware on in the VM and hide it by allowing only

[2] Common goldeneyes are known to lay her eggs in the nest of other goldeneyes. By doing so they escape their parental investment which allows them to do other things.

known-good PCR values in the vTPM. Secrets bound to a known-good state of the vTPM become accessible and remote attestation would allow a verifier to establish trust and release secrets to a VM which is not trustworthy.

Variant 2 - changing physical Root of Trust

The variant shown in Fig. 5.2 likens *Goldeneye* to a cuckoo attack [105]. The verifier in our setup has no prior knowledge about a vTPM, a virtualization system, and a TPM but trusts the physical components. An adversary with access to the association between VMs and vTPMs can exploit the fact that commonly the virtual TPM is used as a pointer to the physical TPM. By associating a VM on some virtualization system with a vTPM on another virtualization system, an adversary can convince the verifier to establish that her VM is running on another virtualization system which may be trustworthy. This would allow an adversary to run virtual machines on bad virtualization systems while using the virtual TPM to hide the fact.

Executing the Attack

The attack exploits an architectural gap in the design of a virtual trusted platform. A TPM is used to establish the software state of a virtualization system which itself hosts VMs and vTPMs. The vTPM presents TPM PCRs or a reference to a TPM encoded in a vTPM Endorsement Key Certificate [14]. From a verifiers perspective, the vTPM tells the software state of the VM as well as the underlying virtualization system.

Although a "strong" VM-vTPM association is a topic and requirement in prior work [56, 14, 34, 120], the software defined association is not verifiable (nor enforceable by trusted elements) — a concerned client can not establish trust in it. This in turn allows an adversary to carry out a *Goldeneye* attack without detection or spoiling the known-good state of any virtualization system. We found that *Goldeneye* can be executed by using standard tools to manage VMs and vTPMs. For example, Xen allows attaching and detaching vTPMs to VMs (Xen-domains) as part of the vTPM management interface [138]. Usage of such commands does not spoil the known-good state of the virtualization

system in the TPM. Given the complexity of a virtualization system and its interfaces, we expect more opportunities for an adversary.

Without improvements to the VM-vTPM association, the possibility of a *Goldeneye* attack contradicts the declared objective of [34, 120] to reduce the amount of trust needed in a *semi-trusted*[3] provider.

5.3.2 Graphical Model

A key feature of the model visualized in our introductory figure (Fig. 5.1) is its inherent layering. The entire system is visually and functionally separated into different layers such as hardware, VMM, virtual machines / guest OSes and applications. The form of its structure and transactions can be captured quite naturally using layered graphs. A bigraph [93] is a form of layered graph that superimposes a spatial *place graph* of physical or virtual locations and a *link graph* which denotes interaction on a single set of nodes.

The system in our introductory example (Fig. 5.1) can be translated using virtualization system terms to form the graph of Fig. 5.3. We formalize it in a bigraphical fashion by defining *linking* and *placing* independently. Thereupon, we describe *interfaces* for *composition* (i.e., how to create larger systems out of elements) which we constrain with *controls* for different kinds of nodes. Finally, we offer a natural translation of our graphical structure and compositions to formulae of first-order logic with which we further formalize *Goldeneye*.

Interaction

To express interactions between nodes (elements) in our model we introduce a *link graph*. In a graph with nodes V and edges E, edges join pairs of nodes. A *hypergraph* is a generalization which allows edges to join any number of nodes. Hyperedges can be seen in a straightforward manner using only edges (u, v), $u, v \in V$ with node u being part of the original graph and placing v as a inserted "hyperedge node" for each hyperedge $e \in E$ to join any number of nodes at v.

[3] Semi-trusted agents and elements are trusted but they may become compromised in an attack.

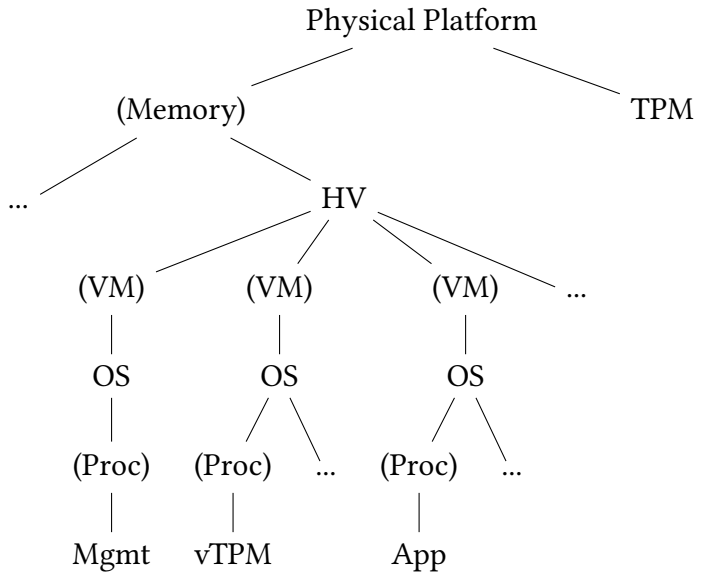

Fig. 5.3 Graphical model of a virtualization system showing the *placing* of its elements. VMs and Processes are included as abstractions for guest OS and apps, respectively.

Definition 3 (Hyperedge). *A Hyperedge is an edge $e \in E$ between any number of nodes V. E is a non-empty set of non-empty subsets of V ($E \subset \mathcal{P}(V) \setminus \{\emptyset\}$).*

Each node $v \in V$ has an *arity* and has *ports*. The arity, e.g. a natural number, both names and limits the number of ports for each node available in the hypergraph ($P_v := \{(v, i) \mid i \in ar(v)\}$). Thus the ports available in the hypergraph is the union of disjoint sets P_v, $v \in V$ ($P_V := \uplus_{v \in V} P_v$). The hypergraph is then defined by the quadruple

$$(V, E, ar, link)$$

in which the ar is a mapping of nodes to ordinal numbers ($ar : V \longrightarrow ordinal$), and $link$ assigns each port to an edge in E ($link : P_V \longrightarrow E$) (Fig. 5.4) .

We describe VM-vTPM association using a link graph later on. Another typical interaction in a virtualization system is *controlling* or *configuring* in the sense that one element configures another one, e.g. launching or deleting virtual machines.

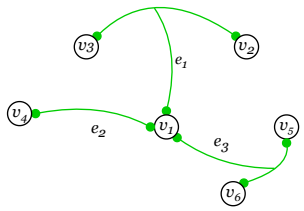

Fig. 5.4 A hypergraph with nodes $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ and edges $E = \{e_1, e_2, e_3\}$. Nodes are circles, ports are blobs, and an edge links ports.

Placing

We use a place graph to express Fig. 5.3 more formally and using the same nodes as before. This gives us a bigraph with nodes V and edges E has a hypergraph (V, E) and a superimposed forest with nodes V [93]. Our representation of a virtualized system naturally resembles a parent-child relationship where some elements supervise others. Including a mapping over nodes $v \in V$ in our hypergraph which allow us to express parent-child relationships.

Definition 4 (Place Graph)**.** *The graph of Fig. 5.3 is a hierarchical tree of vertices and a parent mapping. V is the set of nodes and $prnt : V \longrightarrow V$ is the parent map and defines the* nested *place structure. The parent map is acyclic (i.e., $\forall k > 0, v \in V, prnt^k(v) \neq v$).*

With the superimposed forest and parent mapping from $prnt : V \longrightarrow V$ we make our hypergraph into a bigraph. Consequently, the 5-tuple

$$(V, E, ar, link, prnt)$$

describes our bigraphs alone and without *interfaces*.

Continuing the example in Fig. 5.4, we superimpose a place graph in Fig 5.5. The resulting bigraph allows nodes to be placed and linked independently and freely. However, we need a way to make more precise definitions for the kinds of system we wish to model. Indeed, components of a virtualization system can not be nested arbitrarily and not all nodes may be linked or be part of interactions.

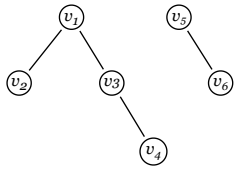

Fig. 5.5 A forest over nodes $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ defined using a parent mapping $prnt_V(v \in V)$.

Controls

The nodes of a bigraph are of different kinds and they contribute differently to its dynamics. To each node in a bigraph is assigned a *control* to express properties of its kind. Each model may define different controls, specified—together with arities—in a *signature* such as

$$\mathcal{K} = \{\mathrm{X} : n\}$$

which states that nodes of a kind X have n links and so on. Controls serve a purpose similar to a type system by defining the arity for kinds of nodes. The set of controls is referred to as a bigraphs *signature*. Consequently, in any bigraph with a control $\mathcal{K}$, the arity of a node is the arity of its control. Bigraphs with controls are written as

$$(V, E, ctrl, link, prnt)$$

where ar has been replaced by the now appropriate $ctrl$ for each node. To model our virtualization system, we might simply say that nodes of the vTPM kind have exactly one link (to a VM). We might say the same about a VM which also has only one link (to a vTPM). Stating only those controls ($\mathcal{K}_{\mathrm{VS}} = \{\mathrm{vTPM} : 1, \mathrm{VM} : 1\}$) lets us constrain our model to only consider interactions like VM-vTPM association.

Interfaces

A bigraph has interfaces which define its use as a building block to make larger bigraphs from smaller ones. Interfaces are defined independently for the *link*- and *place graph*. Interfaces are separated into *inner* and *outer* faces allowing

composition of bigraphs in an intuitive way: if the outer faces of a bigraph match the inner names of another, the graphs may be connected at these vertices.
All interfaces take the form $I = \langle n, X \rangle$. If $X = \emptyset$ we say $I = n$, if $n = 0$ we say $I = X$, or $I = x$ if $X = \{x\}$. If $n = 1$ the interface is said to be *prime*. The empty interface $\epsilon = \langle 0, \emptyset \rangle$ is called the *origin* of a bigraph [93].
Bigraphs can be joined either by nesting or by linking. Linking simply means to connect the inner names of some host or contextual bigraph with the outer names of the bigraph which we would like to compose. E.g., if a host graph H has inner names $\{x, y\}$ and graph F has $\{x, y\}$ as its inner names, then we may join the link graphs of H and F at $\{x, y\}$.

The interfaces of a place graph are referred to as *sites* (or holes) and *regions* (or roots) for it's *inner* and *outer face*, respectively. Sites and roots are devices to define composition by *nesting* in a place graph. The abstraction is quite intuitive in that sites act as *holes* for nodes. Given that each node without another node as parent must be a *root*, we still need a way to express *where* this root node can be placed. To do so, we explicitly give each such node a virtual *root* node $m,\ (m, v)$. The set of roots and the set of nodes are disjoint. The counterpart to the virtual root is the *site*. Nodes may or may not have a site, i.e. a point at which we can add another graph. Simply put, graphs may be joined at the roots and sites by *plugging* the roots of one graph into the sites of another. The sites are the inner face of a place graph. As an example, we could describe the forest in Fig. 5.5 as a place graph which has exactly two roots, the root of v_1 and the root of v_2, and offers no sites at which we could add more nodes (or a graph). Furthermore, the links according to Fig. 5.4 are all closed, so there are no inner names at which we could join other nodes (or a graph). We write this as $\epsilon \longrightarrow \langle 2, \emptyset \rangle$.

Composition

When treated categorically, bigraphs are simply *arrows* and interfaces are corresponding *objects* in a symmetric (partial) monoidal category (spm-category) [93]. As a result, the composition of bigraphs is definable in terms of the composition

of arrows in the category. Elements of the virtualization system are the arrows and their interfaces are objects.

We distinguish between composition by nesting and composition by juxtaposition. If we want to express nesting, we use a . (dot). E.g. to express Fig 5.5 we can write $v_5.v_6$ to express that v_6 is nested in v_5.

If we want to express that two nodes are juxtaposed we write | between them. We express the first tree in the same figure as $v_1.(v_2 \mid v_3.v_4)$. By using || to indicate different roots, we can express the entire forest as $v_1.(...) \parallel v_5.v_6$.

In short, the composition of two bigraphs is defined by matching the inner interface of the first graph with the outer interface of the second. For nesting, this means (a) filling the sites of the first graph (holes) with regions of the second graph and (b) merging the inner names of the first graph with the outer names of the second graph.

The virtualization system can be described in terms of elements. We can describe their interfaces independently for each element and then composing them as individual bigraphs with one node each. We will simply treat the virtualization system as a contextual graph since we are not concerned with different kinds of virtualization systems. For simplicity, we say that each virtualization system has exactly one root and offers *sites* for a number of vTPMs and VMs. Virtualization systems can be juxtaposed. VMs and vTPMs have exactly one link x which allows them to be joined. VMs offer sites for an OS whereas vTPMs offer no sites for nesting.

Semantics

Since we aim to develop a model with which we can reason about simple properties of a system, we need to attach some meaning to the graphical model we have introduced. Most importantly, we will be likening the place graph and nesting with an implicit dependence, i.e. a child node depends on parent, and the link graph with an optional dependence, e.g. one element might control another. The integration of bigraphs with logical frameworks is studied in depth and in relation to spatial and separation logic in [29]. Our reasoning follows the naive notion that a child depends on a parent—any assumptions we make about

a node directly depend on the assumptions we make about the parent of that node. This dynamic can be given as a set of inference rules for introduction

$$\frac{P(y) \qquad P(x) \qquad (x.y)}{P(x.y)}\ \text{-Dot} - \text{I}$$

and elimination, respectively:

$$\frac{P(x.y)}{P(x)}\ \text{-Dot} - \text{E}_1 \qquad \frac{P(x.y)}{P(y)}\ \text{-Dot} - \text{E}_2 \qquad \frac{P(x.y)}{x.y}\ \text{-Dot} - \text{E}_3$$

The transitivity of (.) *dot* composition is studied and detailed in [93]. Our reasoning over properties follows this exactly and we can intuitively explain the transitivity of our Dot rules above.

In addition to nesting (Dot-Rule), we have a *controls/configures* relation between two nodes which is derived from the link graph and therefore orthogonal to nesting. In short, the *controls/configures* relationship extends the nodes which we have to consider in our reasoning. Whenever we naturally have to consider parent nodes of a node of interest, we can expand this set of nodes by declaring nodes which *controls* the node of interest. The reasoning is no different in the sense that any property we want to show about a node, we also have to show in the node that configures it. The *configures* relationship is an edge in the hypergraph and requires that if x configures y ($x, y \in V$) then, in order to prove a property of y, we need to also prove the property in x. In other words, if x configures y, we can conclude $P(y)$ only if we can prove P for x *and* y. The rules for this kind of composition allow similar inferences to the Dot – E/I rules above. However, x and y are not necessarily in a *parent-child* relationship. In short, if we wish to prove a property about $P(y)$ with the knowledge that y is configured by x, $P(x)$ and $P(y)$ need to agree as the proof of $P(y)$ depends on the result on a proof of $P(x)$.

5.3.3 Trust Model

In this section, we construct a trust model which will bring our formal discussion as close as possible to actual reasoning about trusted systems. To analyze *Goldeneye*, we model different situations using predicate logic with our graphical model as context. The graphical model describes the system of interest and

Table 5.1 Trusted Computing Predicates.

Predicate	Meaning
$\text{Trusted}_{\text{A}}(a)$	Agent a is trusted
$\text{SaysSecure}(a, e)$	a says element e is secure
$\text{Trusted}_{\text{VS}}(vs)$	vs is trusted
$\text{On}(tpm, m)$	TPM tpm is on m
$\text{Trusted}_{\text{TPM}}(t)$	TPM t is trusted
$\text{Trusted}_{\text{VTPM}}(\text{vtpm})$	vTPM $vTPM$ is trusted
$\text{Bound}(vtpm, vm)$	$vTPM$ is bound to vm
$\text{SaysBound}(vm, vtpm)$	vm says $vtpm$ is bound
$\text{Trusted}_{\text{VM}}(\text{vm})$	vm is trusted

Table 5.2 Trusted System Axioms.

$\forall a, e\ \text{Trusted}_{\text{A}}(a) \wedge \text{SaysSecure}(a, e) \supset \text{Secure}(e)$
$\forall t, m\ \text{On}(t, m) \wedge \text{Secure}(m) \supset \text{Trusted}_{\text{TPM}}(m.t)$
$\forall vm, vtpm\ \text{Secure}(vm) \wedge \text{SaysBound}(vm, vtpm) \supset \text{Bound}(vm, vtpm)$
$\forall vtpm, vm\ \text{Bound}(vtpm, vm) \wedge \text{Trusted}_{\text{VTPM}}(m.vs.vtpm)$
$\quad \supset \text{Trusted}_{\text{VM}}(m.vs.vm)$
$\forall t, vs\ \text{Trusted}_{\text{TPM}}(m.t) \supset \text{Trusted}_{\text{VS}}(m.vs)$
$\forall t, vs, vtpm\ \text{Trusted}_{\text{TPM}}(m.t) \wedge (m.vs.vtpm) \supset Trusted_{VTPM}(m.vs.vtpm)$

predicates and axioms describe conventional reasoning strategies for trusted computing systems. Table 5.1 details the trust model with predicates and general deductions are given in this Table. Table 5.2 shows axioms of our system. The axioms allow inferences which follow a natural and mostly conventional reasoning over trusted computing systems.

We begin our formal modelling process by defining a vulnerable system as a bigraph. Then we state our assumptions and exercise a reasoning process about the system by applying accepted axioms to assumptions. This allows us to conclude that a VM is trusted when the VM should not be trusted—we use the contradiction to demonstrate the possibility of a *Goldeneye* attack.

Variant 1: changeing virtual Root of Trust

In the first variant of *Goldeneye* the adversary changes the virtual root of trust of a VM by switching between multiple instances of a vTPM. The following

bigraph describes the system setup:

$$/x\ (vm\ :\ VM)_x \| (vtpm1\ :\ VTPM)_x \| (vtpm2\ :\ VTPM)_x$$

We have a closed link x denoted by $/x$ among VM and vTPMs after composing them. We can then compose this graph with a virtualization system as context:

$$m.tpm | vmm.(\square\ :\ VM | \square\ :\ VTPM)$$

The squares indicate sites and $:\ VM$ to indicate which kind of element it is for. This effectively encodes the setup of the first variant of a *Goldeneye* attack. Next, we encode assumptions about our semi-trusted provider P:

1. $\text{Trusted}_{\text{A}}(P)$—P is a trusted agent,
2. $\forall m, \text{SaysSecure}(P, m)$—P asserts that machines are secure,
3. $(1), (2) \supset \forall m.\text{Secure}(m)$,
4. $\forall m, t.\text{On}(m, t) \supset \text{Trusted}_{\text{TPM}}(m.t)$

The semi-trusted provider allows us to regard all TPMs involved as trusted. Then we encode the assumptions about our client as follows:

1. $\text{Trusted}_{\text{A}}(C)$—C is a trusted agent,
2. $\text{SaysSecure}(C, vm)$—C assumes that vm is secure,
3. $\text{SaysBound}(vm, vtpm_1)$—Client's VM says that $vtpm_1$ is bound
4. $(m.vmm.vtpm1_x)$—vTPM 1 is on $m.vmm$
5. $(m.vmm.vtpm2_x)$—vTPM 2 is on $m.vmm$
6. $(vtpm1_x), (vtpm2_x) \supset \neg\text{Bound}(vm, vtpm1)$ — VM could be associated to either $vTPM1$ or $vTPM2$.

In summary, we assume that the client trusts herself. The client provided what she believes to be a secure VM. The VM says its bound to a vTPM

$m.vmm.vtpm_1$. By applying our axioms, we can establish trust in the virtualization system ($m.vmm$) and $m.vmm.vtpm_1$. From our trusted vTPM, we now conclude that our bound virtual machine is trusted as well. The conclusion is $\text{Trusted}_{\text{VM}}(m.vmm.vm)$. At the same time, we also know that $vTPM2$ exists with association x. This implies that neither $vtpm1$ nor $vtpm2$ are bound to vm. Consequently, $m.vmm.vm$ is not trusted.

Effectively, we would have to prevent the existence or prove the absence of more than one vTPM with port x, where x is the VM's port to which vTPMs can be attached. We will discuss possible mitigation below.

Scenario 2: changeing physical Root of Trust

In the second variant, the adversary does not use the possibility of more than one vTPM per VM. Instead, she exploits that we assume the root of trust of the vTPM to be the root of trust of the VM (Fig. 5.2). We conclude that a particular VM is trusted based on a chain of trust measurements of a bound virtual TPM on a different machine. The following bigraph represents the target of the attack:

$$/x\ vm_x \,||\, vTPM_x$$

With two different virtualization systems as a context graph:

$$m1.vmm1.(\square : VM|\square : VTPM)||m2.vmm2.(\square : VM|\square : VTPM)$$

We adopt the assumptions of our semi-trusted provider P and our client C as the verifier but explicitly state that the machines of vm_x and $vtpm_x$ are different. When we apply axioms, again, we conclude that the vTPM $m2.vmm2.vtpm_x$ is bound to the vm. TPM $m2.tpm$ lets us establish trust in the virtualization system and consequently, we trust the vTPM. A trusted and bound vTPM lets us establish trust in the VM.
However, vm is not on $m2$ and consequently, we can not say anything about it. In fact, we have to assume that $m1.vmm1$ is not trusted and that consequently the VM is not trusted.

5.4 Evaluation and Discussion

In this section we will review existing work, show how and why a *Goldeneye* attack could succeed, and discuss possible improvements towards the end. To our knowledge, Berger et al. presented and evaluated the first vTPM implementation [14]. A secure vTPM migration process and security improvements are introduced in [34]. Finally, Schear et al. introduce *keylime* as a cloud key-management system which involves integrity reporting using virtual TPMs and so called *deep quotes* of the infrastructure. We demonstrate our attack for each proposal by giving with an informal overview and an abstract architecture and applying our reasoning process.

5.4.1 Goldeneye in Related Work

Virtualizing the TPM [14]

The implementation of the TPM in software presented by Berger et al. presented in 2006 has evolved and is now used across architectures and platforms. The vTPM implementation is composed of a vTPM manager and a number of vTPM instances. Intuitively, each VM is assigned a virtual TPM instance. The vTPM manager performs functions such as creating vTPM instances and "multiplexing" requests from VMs to their associated vTPM instances. The architecture of the solution follows the Xen model by running vTPM extensions in a VM (Fig. 5.6).

Association between vTPM instances and VMs is one of the concerns raised in their paper. To address this concern, instantiating, migrating, destroying etc. have all been implemented to require some form of authorization. In short, unless a command to create or migrate a new vTPM instance is authorized, it would be blocked. Whoever administrates the system is also allowed to handle vTPM management. We model the system using three bigraphs: the virtualization system as the contextual graph for the vTPMs and VMs (Fig. 5.7).

The intention of [14] is that once a VM with vTPM support is launched, a vTPM instance is created. The VM has x as its inner-face and can host arbitrary apps / processes. The vTPM on the other hand requires a vTPM manager in its root (we do not explicitly encode this) and has x as its outer face. After

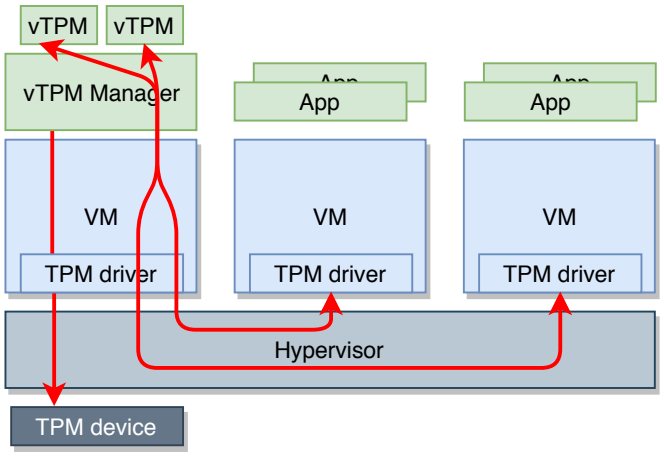

Fig. 5.6 Architecture of the Xen-based vTPM extension.

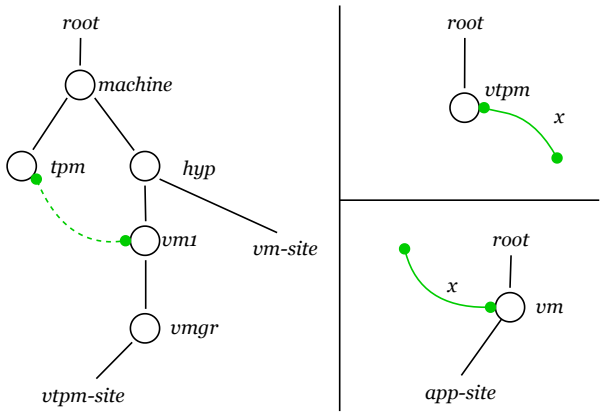

Fig. 5.7 Bigraphical abstraction of the Xen-based vTPM extension. We model three separate graphs: the virtualization system and the vTPM extension (left), virtual TPM instances (top right), and VMs with vTPM support (bottom right).

composing the graphs and joining VM and $vTPM$ at x the link is closed and a vTPM associated. This process is the same for every other vTPM and VM. Without further assumptions, we can reduce this scenario to the attack scenario one. Following the same reasoning, we arrive at the conclusion that we can establish trust in our VM. However we can not exclude that there can be no other vTPM instance associated to our VM ($x = \{vm_x, vtpm1_x, vtpm2_x\}$).

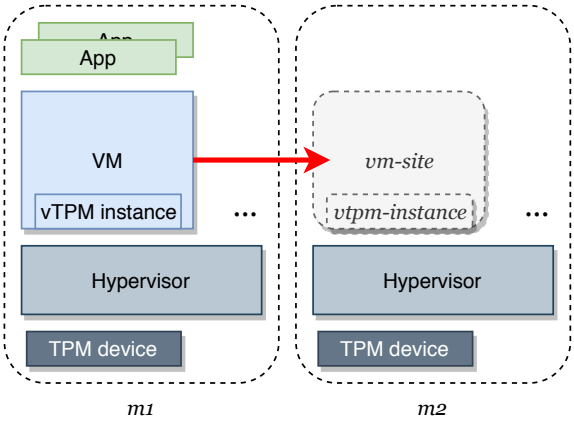

Fig. 5.8 Secure vTPM-VM migration setup. The vTPM instance, according to [34] is running in a VM itself. When migrated, the VM is shutdown and transferred under encryption from $m1$ to $m2$, deleted on $m1$ and resumed on $m2$. We denote this without discussing the protocol by placing a bigraphical $vm - site$ on $m2$ which may become inhabited by the target VM.

Secure vTPM-VM Migration [34]

Migrating VMs and the ability to do so relatively freely and at any time is one of the most distinguishing features of virtual machines compared to operating systems running on hardware. Performing migration securely is a challenge and performing migration of the vTPM in a secure and trustworthy manner a priority. Typically, the cloud provider can be modelled as an agent having more than two machines at its disposal [34], all equipped with physical TPMs and virtual TPMs for VMs. Each virtual machine interfaces with the physical TPM through a software vTPM. [34] assumes that vTPMs do not contain hardware and hypervisor configuration information—unlike [14]. Instead, this information is held by the physical TPM and obtained by querying it directly. Vice versa, the hardware TPM does not include any VM specific information. A perfect setup for *Goldeneye*. The setup evaluated by Danev et al. is shown in Fig. 5.8.

The migration protocol dictates that only upon successful attestation, implying that both machines are *honest* [34], the VM and vTPM are migrated—the memory image of the selected VM which includes the vTPM is migrated. Ignoring the dynamics of this scenario, we express the system proposed as the following bigraph (Fig. 5.9):

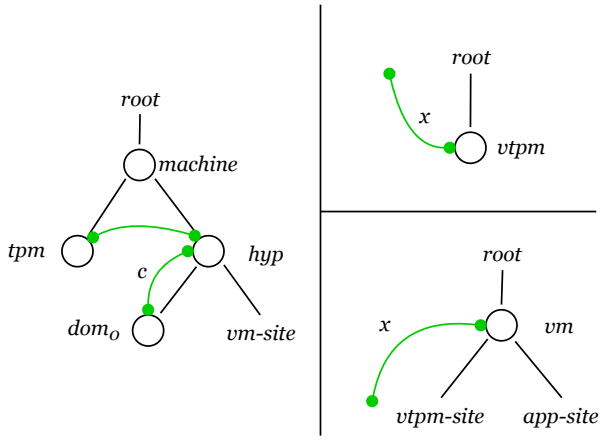

Fig. 5.9 The proposed system model of Danev et al. which uses Xen with virtual machines hosting their own vTPMs. The solution makes no claims about the Xen specific privileged domain dom_0 managing the virtualization system (left). We denote the privilege using the edge $c = \{dom_0, hyp\}$ to state that dom_0 configures hyp. To make the vTPM definition compatible with the [34], we denote x as being an inner face of $vTPM$ and an outer face of a VM. The composition $vm.vtpm$ with the association $\{vm, vtpm\}$ is discussed later on.

The construction in 5.8 and 5.9 allows a straightforward migration process [34].

Intuitively, running the vTPM $vtpm$ in the VM vm while serving as the root of trust for said VM does not fit into the way we naturally reason about trusted computing systems. Figure 5.10 allows us to conclude that $vtpm$ can only be isolated from applications but not the VM.

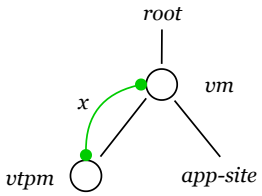

Fig. 5.10 Assembled vTPM-VM in the system model of [34]. The composition of $vm.vtpm$ with the association $x = \{vm, vtpm\}$ produces a circular argument when establishing trust.

Using our axioms, we would like to establish trust in a VM, in this case vm. As per usual, we trust it when the VM tells us that it is bound to some vTPM.

We also need to find and establish trust $vtpm$ which takes us back to proving Trusted($m.hyp.vm$). Danev et al. explicitly do not include VM information in a TPM and this prevents us from moving beyond our initial assumption Secure(VM).

We wanted to show the vulnerability to *Goldeneye* by proving the usual contradiction Trusted(vm) $\wedge$ $\neg$Trusted(vm) but we simply failed to establish trust in the vTPM. In our model, the presented construction is simply not sound.

Bootstrapping and Maintaining Trust in the Cloud [120]

Schear et al. describe a system which aims to provide a *hardware rooted* root of trust for virtualized environments in an infrastructure as a service (IaaS) environment. In the IaaS environment, a cloud provider offers to run a client's workloads, as virtual machines for example, on a provided system. Since the TPM is proven to be useful in physical infrastructure, the authors aim to port the functionality to virtualized systems using vTPMs. To this end, *keylime* is introduced as an end-to-end solution for both bootstrapping hardware rooted cryptographic identities for IaaS nodes and for system integrity monitoring of those nodes via remote attestation. This is supported both on bare metal and in virtualized environments using virtual TPMs. Such an architecture can in theory simply provide vTPMs in support of higher level security mechanisms such as disk encryption and integrity monitoring.

Like previous papers, *keylime* also assumes a semi-trusted cloud provider and considers rogue administrators as part of their threat model. Furthermore, it is assumed that no tampering may occur on physical components, including the TPM, and that the hypervisor is not compromised. The goal is to detect or defend against an attacker trying to gain persistent access to a resource. It is assumed that in order to do so, the attacker would have to perform the kind of modification which is detected by integrity measurements.

A node in *keylime* is implemented based on Xen and with extensions of [14]. Consequently, vTPMs are isolated from other guest VMs. The vTPM interface is TPM compatible, meaning that no modifications in trusted computing enabled guest operating systems is required. A special command called *DeepQuote* is

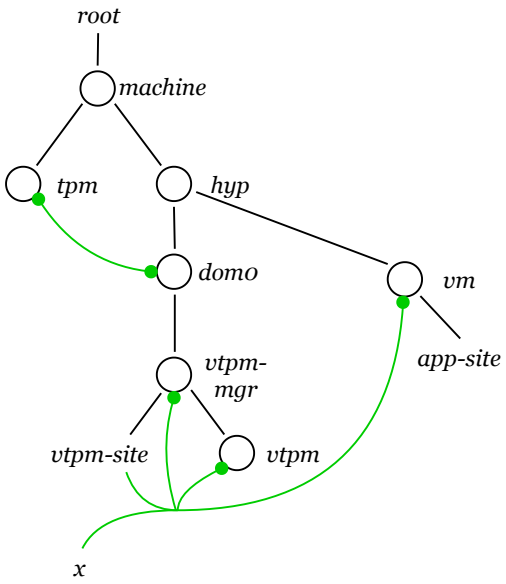

Fig. 5.11 IaaS node model of *keylime* using vTPMs associated to VMs. All vTPMs are isolated from guest VMs vm and hosted on a privileged domain with access to the real TPM. The hyper-edge x denotes the vTPM-VM association and that the vTPM-manager is responsible for the association $\{vtpm, vm\}$.

added in *keylime* which will get a quote form the TPM device. A quote of the TPM which can be *linked* to the vTPM quote is considered to be a *deep* quote. The process of deep quoting in [120] is defined as performing a vTPM quote first (*shallow* quote). The hash of a nonce with the output of a shallow quote is then provided as the nonce for the TPM quote. Discussing the soundness of this approach for certain use-cases is a different concern which will have to be addressed separately. However, we use this deliberate two step protocol with which the vTPM quote seems to be authenticated in our trust model. We express the node model of *keylime* as the following bigraph (Fig. 5.11):

One of *keylime*'s contributions is to form a pool of trusted IaaS nodes via TPM endorsement key enrollment and continuous attestation. The VM can be attested using the vTPM and if necessary, the infrastructure below the VM and vTPM can be attested using a *deep*-quote. Following our process we would conclude that a vTPM is bound to the VM. The vTPM is also bound to the *other* hypervisor—in fact, the binding of the vTPM to a machine and its TPM is not useful in this case. By way of a deep quote—as defined in [120]—we learn

from ($m.tpm$), the trusted TPM device of a platform m that the VS and vTPM ($m.hyp.dom0.vtpm - mgr.vtpm$) can be trusted. Subsequently, we are able to conclude that then also ($m.hyp.vm$) is trusted which was what we set out to prove. However, there is no guarantee that $m.hyp$ is the same for the VM and the associated vTPM. Without extra assumptions, *keylime* is vulnerable to a *Goldeneye* attack.

5.4.2 Solutions

The *Goldeneye* attack is possible whenever an attacker can control the association between VM and a virtual TPM. Indeed, our review shows that this association is at least mentioned and sometimes addressed in solutions involving a vTPM. However, we only see assertions about a *strong* association. No mechanism is proposed which would with a degree of certainty assure a strong association (i) or, perhaps more importantly, enable a verifier to detect potentially weak or broken associations (ii).

A similar problem partially exists in a *bare-metal* scenario. It is elaborated in [105] that without *knowing* the TPM on a particular machine, an adversary on that machine can convince a verifier to establish trust in another machine equipped with another TPM. The association between a TPM and an execution environment is generally not a problem on a real machine and needs special consideration on a virtual platform. Consequently, there is no analog concept which we can simply adopt.

We discuss state-of-the-art trusted computing techniques and how they can be applied to remedy the problem of a remotely verifiable association.

Trust by default

The most obvious solution to the association problem is to make more assumptions or trust the provider's assertion that VMs and vTPMs are bound. The problem and possibility of an exploit is known. A standard way to address it on the provider's behalf is to acknowledge the problem and make assertions through some level of validation: it is impossible to change initial associations and a vTPM would never indicate a root of trust other than the one underlying

the VM as well.
Pros: Requires no changes to the sate of the art and requires only awareness of the problem on client and providers' side. Clients can simply trust that *Goldeneye* configurations are not present.
Cons: Virtualization systems in their entirety quickly grow complex and expose attack surfaces and enable bad configurations. Xen-based solutions today offer specific tools for *listing*, *attaching*, and *detaching* vTPM instances from Xen domains. For a verifier, this means that large parts of the infrastructure simply have to be trusted from the start in order to establish trust in a virtual machine and virtualization system.

Removing malware

An alternative approach to thwart *Goldeneye*'s attacks is to remove malware on a cloud system. With a sure way to run only malware-free cloud nodes, it wouldn't matter which virtualization system in particular a VM is running on—from a narrow security perspective. In our formal model, this would result in adding the assumption that all components of the virtualization system, including the vTPM, are already trusted.
Pros: All issues related to security and trust are now deferred to a client's VM. As long as this VM is trusted, naturally, the composition of the VM with a virtualization system is also trusted.
Cons: This approach tends to be circular—the whole point of getting quotes of a VM or deep quotes of the virtualization system is to check its make, compliance, and ultimately absence of any malware and to make a trust decision of the running system.

Removing interfaces

We showed that *Goldeneye* attacks are thought to be possible since there is no way for a verifier to check the setup of her virtual platform as soon as the provider runs it. The easiest way to produce a *Goldeneye* attack is to change vTPM instances *locally*. As outlined earlier, those actions are considered to be *malicious configurations* and do not require malware or any other detectable

software additions.
Pros: Prevents the reconfiguration of VM-vTPM associations while the VM or the vTPM is running by malicious admins.
Cons: Actions such as attaching and detaching vTPM instances are still needed. If implemented elsewhere—potentially as part of the hypervisor—will increase the trusted computing base and negate disaggregation efforts.

Integrity measurements

Various integrity measurement architectures have been proposed to extend a chain of trust (measurements) from the TPM and a trusted BIOS and boot loader all the way into operating systems and application space [117, 111, 125, 83]. Similar concepts have been proposed for hypervisors and VMMs [10, 79]. Using such measurement architectures, we could record running VMs and their images as well as vTPM software in the physical TPM.
Pros: Integrity measurements of vTPM and VMs in the TPM could help linking a VM to a hypervisor or VMM. Combined with integrity measurements of the vTPM software, this would allow us to conclude that an instance of a VM is executed on a virtualization system running trusted vTPMs.
Cons: Asserting (using measurements) that a virtualization system runs a certain *kind* of vTPM, shown via hash of its binary, is helpful but provides little information as to whether a particular *instance* is associated to a VM on the platform. Furthermore, measuring vTPM instances as well as VMs will cause frequent updates to the TPM's PCRs. Aside from performance issues, frequent changes to PCRs make it hard to support TPM-based encryption and binding of secrets to a platform state.

Recording vTPM-VM Association

A promising solution which would confine adversaries, is to record interactions with vTPM and vTPM management interfaces outside of the normal operation. An example would be to allow changes to the vTPM association but to record them before applying those changes. This would prevent an adversary from switching between local instances, or routing vTPM commands to another vTPM

instance on another machine using known channels. Interface access and issued commands can be extended into TPM and vTPM PCRs to record and signal the interference. **Pros:** Implementing this can be straightforward by extending the vTPM implementation and manager at the interfaces. Supporting protocols and TPM policies could react to such changes by making keys for storage and attestation keys unavailable. **Cons:** This approach needs assumptions about the vTPM implementation on a virtualization system. Asserting that an implementation will record adversary interference and concluding that—unless reported—there can be no interference requires that we fully trust the implementation in the first place (comparable to "trust by default" in 5.4.2).

So far, we have viewed vTPMs as processes running alongside other vTPMs in an environment controlled by the vTPM manager (Fig. 5.7). We will now discuss architectural solutions which change how vTPMs are implemented on a virtualization system.

Unikernel vTPMs

Disaggregation is a core concept in the development of Xen[12] and comparable in philosophy to *microkernelification* of systems [74]. The same principles can be applied to implementing vTPMs. Currently, vTPMs are implemented as applications / processes which run on top of an OS. Unikernels [110] provide the necessary interfaces between applications and real hardware without the need of a complete OS. Instead, unikernels allow us to run applications with minimal overhead directly on a physical or virtual machine. Each vTPM in Xen is running on a small guest operating system as a virtual machine or domain. All vTPM domains are controlled by a vTPM manager in a privileged domain which supervises VM-vTPM communication. By excluding the vTPM management, its code base, and interfaces, we could achieve a design which practically involves *only* vTPMs and associated VMs.

Pros: Using unikernels would allow the implementation of vTPMs as standalone components. With the association deferred to the hypervisor, this would significantly reduce the interfaces and amount of code that is part of the trusted computing base in our vTPM design.

Cons: We would have to implement the *association* as part of the hypervisor or involve a privileged domain to handle association for us. Further, migration in such a design is not as straightforward as in [34] and would require migrating vTPM and VM separately.

vTPM as part of VM abstraction

When we discussed possible vTPM architectures, we have so far only considered having vTPMs as *hosted* processes. Alternatively, we could also include vTPM functionality in a hypervisor or at least provide a way to save and restore vTPM states per VM as part of the virtual machine control structure.
Pros: We effectively reduce the trusted computing base of the vTPM and place the responsibility of associating and maintaining the association between VM's and vTPMs in the hypervisor. This limits the interfaces to the vTPM and the possibilities for attackers to make changes.
Cons: Putting vTPM code and functionality in the hypervisor would obviously contradict the design philosophy of disaggregation. Furthermore, vTPMs are essentially VM controlled, we would provide clients with a *window* to the hypervisor through vTPM commands. Tools limiting resources of VMs would not be able to also protect hypervisors if they executed vTPM commands on a client's behalf.

Secure execution environment for vTPMs

The default solution today is to run vTPMs as processes. For KVM/Qemu based solutions, those would be processes implementing the back-end of a Linux vTPM device and on Xen it would be vTPM processes as in [14, 120]. Running a vTPM in a secure execution environment is the next step to reducing the vTPM's trusted computing base (TCB). Separating the vTPM's TCB from the *untrusted* virtualization system would allow clients to use vTPMs even if the virtualization system is not trusted. Examples of *secure* execution environments, i.e., isolated from the VS, include Intel's Software Guard Extensions and MIT's Sanctum [30] as well as trusted execution environments (TEEs). Environments which are isolated from the VS also include code which runs in system management mode

(SMM), or generally, closer to what would be considered firmware of a machine.
Pros: The trusted computing base of the vTPM would be independent from the VS and comparatively minimal. Furthermore, the attestation and root of trust of such a solution could be independent from a TPM and would therefore scale better than deep quotes involving TPMs.
Cons: While vTPMs are architecturally isolated from the virtualization system, we still rely on the VS to make the connections between VMs and the vTPMs. Furthermore, since vTPMs and VMs no longer have a shared trusted computing base, vTPMs would have to explicitly include the VMs TCB as well. Generally, we found that changing the trusted computing base of vTPM and VM alone does not sufficiently address VM-vTPM association as well.

Hardware assisted vTPMs

A native solution would to integrate vTPM functionality as part of the machine architecture and machine virtualization support. Such support would effectively extend a physical TPM, often implemented as part of firmware, to have native support for virtual machines. Such a solution would effectively provide a TPM with a shared context for the entire machine and an individual context for each VM to capture state information. In such a case, the virtualization system would only be entrusted with the setup of virtual machines and would delegate vTPM allocation and association to the hardware.
Pros: This would allow us to set up VMs in the hypervisor and with added hardware support we could use a vTPM with a trust model similar to the real TPM. The physical platform alone would be responsible for pointing vTPMs to the relevant TPM PCRs when establishing trust in a virtualization system.
Cons: Requires by far the most significant changes in hardware and implementation. The TPM itself is not designed to context switch. With TPM 2.0 some of this could be emulated but the context switching alone would cause severe performance issues with currently available TPM devices.

5.5 Summary

Creating a "trusted" virtualized platform is necessary for a variety of secure cloud computing scenarios. Ideally, we should be able to use secure hardware such as the TPM to bootstrap trust in a virtualization system and utilize a virtual TPM as a root of trust for a virtual machine. Our formal model reveals that the process of extending trust from the TPM all the way up into a VM is vulnerable to a *Goldeneye* attack. The *Goldeneye* attack emphasizes the importance of a trusted association between VM and vTPM. The association between a system and its root of trust is typically not considered on a real machine but we emphasize that it needs special considerations in case of a VM and a vTPM. We demonstrated attack vectors in recent papers and proposed solutions which could be implemented today. Currently, we rely on assuming an invariant about the association between VMs and vTPMs. We suggest and discuss several solutions including constraining interfaces, changes to measurement architectures, and vTPM architectures. A clean solution might be to provide architectural support for roots of trust for virtual machines. In future work, we will discuss cryptographic approaches and explore some these options in depth. The combination of bigraphs with a trust model was useful in modelling trusted virtual platforms structurally. We expect interesting results when we add dynamics and transitions to our trust model in the future.

Chapter 6

Discussion & Future Work

The challenge of secure cloud computing has helped motivate this thesis and many other works. What sets this thesis clearly apart from other efforts is its focus on TCG-based trusted computing technologies which include effective tools for trust establishment, software integrity, and data confidentiality. The close relation with practical trusted computing technologies is in fact a key feature of this thesis. Many of its contributions align well with Trusted Computing Group (TCG) efforts and have already informed them in various ways. The synergy between model-driven, theoretical contributions and TCG specifications promises widespread adoption and ultimately improvements to real-world systems.

The following sections present a discussion of contributions towards a Virtual Trusted Platform in Section 6.1 based on the goals outlined in Section 2.4.2. This chapter concludes with directions for further research and development in Section 6.2.

The discussion in Section 6.1.1 as well as directions for future work related to the virtual platform in particular are part of a prepared article on *A Modern Virtual Trusted Platform* [81].

6.1 Discussion

A number of the designs, solutions, and proposals of this work are based on domain specific analysis, abstraction, and models which ensured a degree of generality and allowed for big leaps towards a modern virtual platform. The relation between this thesis and real-world TCG-technologies also comes with responsibilities.

On one hand, practical aspects of proposed solutions and the progress made in this work must be put into perspective. The question "Can we *specify* and *build* a virtual trusted platform based on the results of this thesis?" guides the discussion on achieved goals and features of a potential implementation in Section 6.1.1.

On the other hand, the models and abstractions in this thesis are up for debate and such theoretical aspects are compared to other work with a similar focus in Section 6.1.2.

The following section presents a detailed discussion of expectations, contribution, and a judgement for each of the virtual platform goals in Section 2.4.2.

6.1.1 Contribution to Virtual Trusted Platform

In this section, the thesis will be discussed based on the goals and challenges introduced in Section 2.4.2. The discussion is started with the challenges for an urgently needed *modern* virtual platform specification. The remainder of this section discusses legacy goals [56] which are still relevant today.

The goals below concern a *modern* Virtual Trusted Platform specification and are discussed in the following section:

1. Document a high-level architecture and system model.
2. Include operating system level virtualization in the architecture.
3. Address integrity of the virtual platform and its components.
4. Detail how a virtual platform can be rooted in a physical platform and its roots of trust.

5. Provide interfaces for existing specifications and protection profiles.

The fulfillment of these goals is systematically evaluated below, where for each goal the expectations, contributions, and judgments are presented.

In summary, this thesis has contributed significantly to four out of five major goals towards defining and specifying a modern Virtual Trusted Platform. A key contribution is the inclusion of containers in both *standard*- and *hybrid*-modes without nesting virtual platforms or vTPMs. Furthermore, the model driven approach of this thesis has led to a functional high-level architecture definition which provides interfaces to other standards, specification, and research. A key goal which has not sufficiently been addressed are a virtual platform's roots of trust and how virtual platforms can be rooted in or bound to physical ones.

1. High-level Architectural Model

The first goal is to produce virtualization system model which can be used to express different Virtual Trusted Platform architectures. Sufficiently high-level models should include both virtual machines and containers and should avoid vendor and implementation specific details.

Chapter 3 makes only minor contributions towards this goal and presents only a very informal model. Historically, these models have not led to much success in [56] and often relied on hidden details and assumptions which made specifications hard to comprehend and adopt. The first contribution towards comprehensively modeling a Virtual Trusted Platform is presented in chapter 4. The chapter focus on a formalization of certain system components as well as virtual environments. A byproduct of this is a model which assumed that the components had an associated virtual TPM available, i.e. a virtual platform, and that during the operation, software in the virtual environment could only access it using its special interface. This approach is a likely candidate for describing a root of trust for measurements (RTM) for containerized systems. Documenting and modeling a high-level architectural view is also a major topic in Chapter 5. The model aims to explain the overall architecture in terms of its components and their interfaces. This allows users of the model to compose arbitrarily

complex virtual systems which is a unique feature. The modeling is based on Milner's idea of Bigraphs [93] and the components are drawn from standards and protection profiles for virtualization systems [56, 102]. This leads to an expandable and functional model with concrete foundations. This approach is also a likely candidate for an update to the virtual platform specification.

The goal is achieved. The thesis has provided an informal overview of the Virtual Trusted Platform as defined in [56, 79]. A formal model is given which supports integrity measurement architecture and RTM design. Lastly, the bigraphical model is the first attempt at formalizing a functional model which can be used to abstract usually complex virtualization systems.

2. Including OS-level Virtualization

Modern virtualization architectures include OS-level virtualization. The integration of containers into a virtual platform architecture including a root of trust for measurement and chain of trust measurement design which supports containers is a pressing task. Containers often run on virtual infrastructure and hybrid architectures, e.g. containers on top of virtual machines, have several benefits which include better isolation capabilities in the sense of security, performance, and fault tolerance. Not considering hybrid virtualization constructions would severely limit the usefulness of any work in the future.

This thesis makes a major contribution which directly aim at supporting hybrid architectures. The first and perhaps most significant contribution is the design of an integrity measurement architecture which naturally supports containers. The work presented in Chapter 4 suggests using a single vTPM in the virtual platform on which the container OS and containers are running. This effectively turns the design of EIMA into the root of trust for measurements (RTM) for containers. Using a single virtual platform for multiple containers is a deliberate design choice which improves a vTPM's isolation from containerized apps and avoids nesting virtual platforms entirely. By using one vTPM per container OS, supporting containers is a matter of suitable integrity measurement architectures while the general virtual trusted platform model for virtual machines remains unaffected. Thus, the design of *EIMA* as a RTM for containers

represents a solution which inherently supports hybrid virtualization and avoids complex matters of nested virtual platforms. **The goal is achieved**. Standard and hybrid virtualization architectures are addressed with a major contribution. However, vendors are also proposing an implementation which assigns a virtual platform to each container. Such solutions reduce the isolation between vTPMs and virtual environments and introduce the issue of nesting virtual platforms which this thesis does not address.

3. Integrity of a Virtual Platform

The binding between components of the trusted platform on a physical machine is typically a manufacturing problem. Not all platforms are manufactured to the same standard. A number of attacks target the integrity of the physical platform in order to break security tools like Microsoft's *BitLocker* which relies on the assumption that the TPM is fixed to a platform. Furthermore, if an attacker is able to replace or interdict the CRTM, which links the bootstrapped system with the platform, arbitrary faux system measurements can be given to the TPM. On a hardware level, solutions range from physical solutions (e.g., glue on the low end) to cryptographic solutions which authenticate or encrypt TPM commands on a platform level. However, today it is unclear how to solve such a problem with generality and in a way that allows integrity enforcement and verification for a Virtual Trusted Platform.

The thesis makes major contributions towards this goal. In fact, the problem is first discovered and formalized within this thesis. The problem is brought to attention with the *Goldeneye* attack which highlights the importance of integrity between virtual platform components. Following the precise problem definition, potential solutions for a better virtual platform are presented in Section 5.4.2. The solutions include a wide variety of options which should address the attack on various platforms and virtualization systems.

The goal is achieved. Without the contributions of this thesis, this problem would have likely been sidelined as previously in [56, 120]. The attack is demonstrated and solutions are proposed. The solutions are not yet formalized

and the model needs to be extended to include some of the solutions in order to be standardized further.

4. Virtual Platform Roots of Trust

Requirements for binding a Virtual Trusted Platform to physical roots of trust is a crucial step when a trusted platform is to be virtualized. A virtual platform can be constructed on any platform and without directly virtualizing trusted hardware. Such constructions are common in cloud use-cases where *virtualizing* secure hardware is not the goal but providing a *virtual* alternative is. TCG applications however, are centered around the notion of a physical platform and in those cases a Virtual Trusted Platform which is detached from a physical one is not acceptable.

Achieving this goal not been an issue of this thesis. The ability to *bind* a virtual platform to a physical can offer security benefits and would prevent attackers in Chapter 5 from running virtual machines or containers on corrupted hardware. Vendors and virtual platform operators also have interest as they would be able to sell virtual instances to clients and at the same time prevent cloning and running instances on excluded hardware. The model in Chapter 5 allows expressing this problem with some some refinement. The refinement would have to address how virtual platform credentials can be bound or restricted to a physical platform in particular.

Not achieved: Achieving this goal is deeply related to research into virtualizing the TPM [14] (e.g., as a means for device/platform identification, as a root of trust for reporting (RTR), and as a root of trust for storage (RTS)). Solving this goal is important and may lead to improved solutions which allows the protection of virtual platform credentials by using the trusted platform. Directions towards achieving this goal are given in Section 6.2 as future work.

5. Interface With Related Specifications

As mentioned in the introduction, constructing a virtual trusted platform is unlikely going to be completed in a single work. Instead, utilizing a variety of existing and ongoing research as well as commodity systems seem to be

preferable for both industry and academia. For this reason, contributions with a virtual trusted platform as their goal must provide interfaces to existing systems, standards, and research. This is an important requirement for this thesis which aims to abstract commodity systems in order to analyze and design solutions towards a virtual platform.

The work presented in this thesis is largely based on standardization efforts and commodity systems supported by both industry and academia. Specifically, Chapter 3 analyzes a high level architecture for trusted cloud computing [1] which uses TCG primitives. The discussed remote attestation protocols which are meant to provide trust information about a virtualization system and virtual environments are taken from earlier standardization proposals and prior work [79]. The primitives (i.e., TPM and vTPM) are based on relevant TCG specifications [64, 57]. Furthermore, the principles for remote attestation in virtual environments are refinements of [28] and [79] which also resulted from discussions with standardization bodies and industry partners. The enhanced integrity measurement architecture (EIMA) provides both formal and practical interfaces. From a practical perspective, the model clearly represents a variety of popular container technologies and the TPM model is essentially a simplified TPM with only platform configuration registers [64, 57]. The system provides insights for OS-level isolation techniques in general and relies only on a subset of TPM functionality. Consequently, the contributions can be applied to a variety of systems and extended in many ways. The formal aspects allow the adoption of a large body of existing and ongoing research by such as [36, 69]. Finally, the model used in Section 5.2 is derived directly from prior work [56] and current standards and protection profiles [102]. Chapter 5 provides interfaces between components and allows users of the model to attach further security requirements for trusted systems to each component.

The goal is achieved. The models of Chapters 4 and 5 include standardized components which are compatible with definitions and security requirements of protection profiles such as those of the National Information Assurance Partnership (NIAP) Common Criteria Evaluation and Validation Scheme (CCEVS) [102]. Despite sharing some of the same functional components, mapping the contributions of this thesis to systems defined by other standards is still not straight-

forward. Related documents which are either published or under review often define artificial system boundaries, parties, and use-cases which may add significant complexity. Addressing certain domain specific models is a task for future research.

The following goals and challenges are shared with existing specifications and still very relevant:

6 Discuss responsibilities and potential issues of components of a virtualization system.

7 Minimize change to systems.

8 Provide support for the migration of virtual platforms.

In summary, the two remaining goals for a virtual platform specification are achieved but migration is still an open issue. Goal 6 is achieved in part by our inclusion of containers with *EIMA* (Section 4.3) and the system model in Section 5.2. Changes to systems are also minimal as *EIMA* and *UCAS* (Section 3.4) are software solutions designed to work with commodity hardware. The integrity of a virtual platform can be improved and enforced in various ways (Section 5.4). Migration is still an open issue and the process very nuanced and not standardized itself. Addressing migration of a virtual platform is part of the future work.

6. Discuss Issues and Responsibilities of Virtualization System Components

The components of the virtualization system are the essential functional components which provide virtual environments as either virtual machines or containers. The fulfillment of this goal is related to documenting a high-level architecture in goal 1. One of the key features which the virtualization system must provide is isolation. Traditionally, isolation or managing access to resources is the responsibility of hypervisors and virtual machine management. However, in recent years hypervisors have benefited from added hardware support which shifted functionality and responsibilities related to isolation to

the hardware. A clear description of each component and its support can be helpful to indicate where implementations need to pay attention with regards to security. Clearly, moving functionality away from a mutable software component to firmware or even hardware can have an effect on the trust model and the amount of trust which must be placed into certain components.

Chapter 5 makes the biggest contributions towards the goal in this thesis. The model of a virtualization system presented is meant to provide an interface to the contributions of other works. The interfaces can help to integrate the focus on trust and trust establishment with efforts of systems and security communities. Implementing a general purpose, secure, and trusted system will require the adoption of many approaches and associated standards. The research presented in this thesis is meant to be translatable and applicable to a variety of systems by being largely driven by standard models.

This goal is seen as achieved. The virtualization system components are at some point discussed in nearly every part of this thesis. With its strong focus on trust and trusted computing, the thesis lacks some of the details of works on systems engineering. However, the contributions remain highly compatible. The role of virtualization system components in this thesis is often purposefully reduced to providers of isolation as a security primitive. A more complete model and formalization, i.e. a formalization of both functional and security properties in one model, might be an interesting academic exercise but is at risk of offering only limited practical insights beyond confirming common isolation assumptions [100].

7. Minimize Changes to Components

Trusted Computing systems as of today are largely built around standardized protocols and interfaces. The implementation of protocols and interfaces cannot be changed if any proposed solution intends to remain compatible with a large amount of commodity systems. In recent years, TCG has focused on publishing *core* specifications while extensions are published separately. Compatibility with core specifications is essential but extensions are so well established that they are closer to *mandatory* than optional additions. In this thesis only two

things are essential: core TPM specifications and extensions aimed at both PC client hosts and guest systems.

The work presented in this thesis requires several changes to Trusted Computing technologies. However, none of these changes affect established interfaces or protocols. For example, the vTPM proposed in Chapter 4 might need some additional registers which can be added on a physical TPM 2.0 device by using parts of the available storage [57]. The remote attestation protocol supported by the enhanced integrity measurement architecture remains unaffected and compatible with current standardization efforts. The changes suggested in Chapter 5 are related to system or integrity measurement architectures which are not standardized and new security features added regularly. The standardized components are modeled as primitives or fixed components. Consequently, the trusted platform, the TPM specification, and currently used attestation protocols remain unaffected.

This goal is seen as achieved. The thesis certainly requires change and sometimes even mandates changes for the benefit of increased security and trust. A good example for this is the inclusion of EIMA in container operating systems (Section 4.3) or additional measures to improve on a virtual platform's integrity (Section 5.4.2). Otherwise, only changes to *configurations* (e.g., adding registers and including them in an attestation protocol) are required while *operation* remains unaffected [78].

8. Migrating a Trusted Virtual Platform

Migration is a key feature of virtualized systems. The ability to run them from backups, clone them if needed, and perform emergency recovery procedures which results in high portability, resilience, and availability. Virtual platforms must support at least some kind of migration between different physical platforms. Unfortunately, migration will have to be solved at a very high-level as implementations of the procedure vary between virtualization systems, platforms, and operators. A simple way to address migration is to create solutions which require homogenous platforms and virtualization systems.

Not achieved: Migration is not a feature of the virtual platform in this thesis. In fact, this thesis does not make contributions towards migrating virtual platforms. Related work which is discussed in Section 5.4 targets migration specifically but also fails to take concerns such as *hot-standbys*, virtual platform sleep states, or live migration into account. Migration remains a complex issue and will have to be addressed in future work.

Lastly, the practical aspects of this thesis are discussed from a functional perspective. The remaining goals 9-12 (Section 2.4.2) guide the following discussion and are treated as a set of *desirable features* which a virtual platform should possess in order to be useful.

9 Allow software in virtual environments to run unmodified.

10 Allow binding a virtual platform to a physical platform.

11 Allow verifiers to determine security guarantees, assumptions, and trust model.

12 Provide a way for software in virtual environments to establish trust in the virtual environment, the virtual platform, and the virtualization system.

In summary, the goals or *desirable features* are partially achieved and this thesis ensures that applications are unaffected and it supports establishing trust in relevant parts of the virtualization system as well as virtual environments. Goal 9 is easily achieved as generally no assumptions about applications in a virtual environment are made. Goal 10 is related to virtualizing the TPM in a way which would allow restricting virtual platform secrets to certain physical platforms which this thesis hasn't addressed. Determining security guarantees and trust models is complex since inferring or communicating them itself is not well defined. While this thesis makes contributions, such a feature wouldn't be completed based on the work in this thesis alone. However, goal 12 is easily achieved as trust establishment is a major topic in this thesis.

9. Allow software to run unmodified in a virtual Environment

The expectation for this feature is that software written to interact with other trusted systems or trusted computing hardware can also run in a virtual environment using a Virtual Trusted Platform. Examples for such software include system boot code for guest operating systems, integrity measurement architectures, and traditional TPM applications which perform some form of remote attestation. Trusted computing extensions of commodity software needs to run unmodified in a virtual environment under an adjusted trust model [56].

The goal is achieved. Software systems running on a virtual platform do not require any modification in terms of their operation. In fact, the software systems running on a virtual platform is seen as untrusted and no expectations are made with regards to the software. However, should the software system be part of the virtualization system, then the changes might be significant. This thesis makes no remarks about recurring or nested virtualization where the virtual environment presented to a client or verifier might be established using a second virtualization solution. Such nested systems have not been considered, since in this case a virtual environment would be part of the virtualization system and the requirements for remote attestation, enhanced modified measurement architecture, and virtual platform association in Chapters 3 to 5 would apply.

10. Allow Binding Between Virtual Platform and Physical Platform

A number of use-cases for a virtual trusted platform demand that the virtual platform can be *bound* to a particular physical one. The binding in this case is likely targeting a vTPM and a physical TPM which may contain secrets and credentials associated with their virtual or physical platform. Arguably, allowing a virtual platform to be instantiated on *any* physical platform might also leave the vTPM and its secrets vulnerable [82].

Unfortunately, the meaning of a *binding* in this context is not well established. In a weak sense, binding a vTPM to a physical platform can simply mean that the vTPM becomes usable on that physical platform. In a much stronger interpretation, binding a vTPM to a physical platform can mean that the vTPM

is accessible if and only if it is running on the physical platform it is bound to. Binding may be taken literally and a vTPM may simply be a frontend for a physical TPM: The physical TPM holds and protects all secrets and credentials including those of virtual platforms. The vTPM functionality is then augmented by the TPM. Credentials and key hierarchies of virtual platforms may be based on the physical. In such a case, a vTPM's key hierarchy can be implemented using a TPM's key management capabilities [57]. Solutions based on such virtualized TPM features align with a strong *binding* interpretation.

A weaker binding interpretation can implemented by encrypting a virtual platform using a key held by the target particular physical platform. An efficient solution would involve encrypting root keys of a virtual TPM for a trusted platform such that secrets can not be decrypted and become available on unauthorized platforms. Consequently, an unspecified or untrusted physical platform may be able to run a VM image but it may not be able to decrypt and use the vTPM as well. This would prevent a number of simple attacks where corrupt platforms are used to spy on client's confidential data. However, once decrypted and in operation, the secrets and security of a virtual platform depend entirely on the runtime which may be a commodity OS running on a virtualization system [14].

Either of these approaches would provide some form of *binding* between a virtual and a physical platform but with different levels of assurance and dependencies. The former solution would ensure that virtual platform keys can never leave the TPM. However, restricting access to virtual platforms based on the physical platform alone is likely insufficient for weak or strong solutions. Clearly, secrets and credentials of a virtual platform must also be protected from untrustworthy run-times (i.e., virtualization and operating systems). Such a concern makes the contributions of this thesis towards trust establishment and enforcement very relevant.

The goal is not achieved: Explicitly providing a *binding* between a virtual platform and a physical one is not a concern of this thesis. To enable such a feature, a clearer definition of binding is needed. However, it is apparent that binding a virtual platform to a physical one will be useful only if the layers in between, i.e. the virtualization and operating system, are involved. Consequently,

the contributions of this thesis will play a role for some implementations of this feature.

11. Verifiers Assess Assumptions, Trust Model, and Security Guarantees

Running applications in virtual environment should among many things also improve their security. However, it is clear that such benefits may need an adjusted trust model and additional assumptions. Allowing verifiers to determine and assess security guarantees, assumptions, and the trust model of the virtual trusted platform is important, especially in a distributed setting and during trust establishment.

This feature can be implemented in two *extreme* ways. On one hand, static certificates can be used to establish a chain of trust for each aspect which can be traced back to hard- and software manufacturers and operators. Such certificates are used to assert certain facts and properties. On the other hand, trust establishment strategies such as integrity reporting in combination with comprehensive inspection of reported aspect can be used to *infer* relevant properties. The use of certificates is of course inherently more scalable in many ways but also increases the impact which corrupted links in a chain of trust can have. Inferring all properties is at least equally problematic as inferring all assumptions, the trust model, and provided security guarantees from the ground up is not feasible. A potential solution will likely involve both approaches and find a good middle ground between certificates for static and trust establishment strategies for dynamic aspects. This thesis contributes to the latter approach with user-centered attestation and the enhanced integrity measurement architecture which allow verifiers to establish properties of the software system. However, the thesis relies on an existing certificate infrastructure which is necessary to identify legitimate, trusted platforms [105].

Consequently, **the goal is only partially achieved.** Recording and reporting system integrity measurements is certainly a part this feature. But beyond the software system, defining a suitable certificate infrastructure which extends existing protocols is an easily overlooked priority.

12. Establishing Trust in Virtual Environment, Virtual Platform, and Virtualization System

Providing a way for software in a virtual environment to collect, report, and make use of relevant trust information of the virtual environment, virtual platform, and the virtualization system is perhaps the key feature of a virtual platform. A trusted computing solution to data confidentiality in the cloud will involve decrypting and working with sensitive data only if the computing environment is trusted. Sealing data to a trusted software system operating in a virtual environment adds a clear advantage over potential adversaries in the same virtual environment. The further application of trusted computing concepts, e.g., sealing data to a trusted guest OS, still leaves the virtualization system and its interfaces practically unwitnessed and dangerous. Being able to record the virtual platform as part of the virtualization system is a natural next step when further assurances are needed. This feature is extremely relevant if a solution aims to reduce an adversaries attack surface and limit the impact which infected systems and malicious administrators can have.

The contributions of Chapter 4 are most significant here. Practically, the virtual environment (i.e., the container) is linked with the virtualization system and clients can assess container trust information together with the operating system and the container engine. Beyond this immediate and practical contribution, the work presented in Chapter 4 also outlines a number of important security requirements. Once the goal of *linking* trust measurement chains was achieved, an important step was to make sure that trust information is also constrained and revealed only carefully. A real challenge is to provide a way to record *complete* but *constrained* trust information which is relevant to a particular virtual environment. It was revealed that finding such an approach reduces the typical brittleness of PCRs and improves on privacy which often isn't a primary concern in traditional trusted computing applications. The mentioned contributions have already improved ongoing standardization work towards a Virtual Trusted Platform.

This goal is achieved with *UCAS* and *EIMA* and both virtual machines and containers are addressed. The work presented in this thesis even extends

the feature in order to include cloud setups with multiple mutually distrusting tenants on a system. Previous works are concerned with (functionally) emulating a trusted platform instead of providing such important security properties [14, 56, 120] on a virtual platform.

6.1.2 Discussion of Models and Analysis

This thesis develops several models for the design, analysis, and verification of certain aspects of virtual trusted platforms. In this section, the proposed models and conducted analyses will be discussed from a theoretical perspective and compared to related work.

Model for User-Centered Attestation

The analysis which led to the definition of User-Centered Attestation in Chapter 3 is based on our prior work [79] as well as the write-up of Coker et al. [28]. The analysis followed a set of principles of remote attestation which were extended to include scalability and layer linking. The model which was used in [79] and Chapter 3 is informal and relies on a conceptual understanding of virtualization systems and accepted trusted computing mechanisms. The presented analysis is also informal and discusses abstracted properties of implementations which use common trusted computing and virtualization technologies. When compared to later work in Chapters 4 and 5, the contributions of Chapter 3 are supported by intuitive reasoning which internalizes a lot of complications. In fact, the analysis does not allow a deeper reflection on trusted computing principles which must simply be accepted. Despite this, the informal analysis has significantly impacted further research undertaken in this thesis. Chapter 3 gives directions on the security properties and design aspects which seem most important for effective trust establishment on a virtual platform.

LS^3: Logic for Secure Stratified Systems

The work presented in Chapter 4 is directly informed and impacted by the conclusions in the previous section and the definition of a user-centered attestation for layered systems. One of the important conclusions of Chapter 3 is that the

integrity recording mechanism (RTM) itself must be trustworthy in order to support the attestation of virtual environments and underlying virtualization layers.

The work in Chapter 4 draws from the successful analysis performed in [36] which presented a logic for secure systems and its application to trusted computing. Programs in our system LS^3 externalize a lot of important features such as explicitly creating new threads to execute potentially untrusted code. This is supported with appropriate operational semantics which express system internals. The added functionality proved to be useful for modeling existing container systems and designing new integrity measurement architectures. A more general formalization of containers and container operating systems was not available when our work was first published [83] in 2019. Today, efforts are being made to formalize such containers further on a higher level [68] and without focus on security and trust.

The work presented in Chapter 4 is best discussed in contrast to the prior work of Datta et al. in [36] and applications such as [11]. Prior work is based on a language to model networked code and included only constructs for *jumps*, i.e. for threads to execute new bits of code. Jumps are useful and sufficient for some analyses but they do not allow existing threads to explicitly create new ones. This is clearly limiting when complex systems are investigated. Threads can be added but only implicitly, e.g. by external manipulation of the current configuration. Consequently, adding threads is not the result of an action which can be discussed later or used to model systems. [36] concludes that a further analysis of systems using jumps only leads to no new insights which this thesis can confirm. Furthermore, LS^3 makes a radical cut by deliberately removing primitives for networking and agents. Such primitives have played only a minor role in prior work and hurt the integrity of the formal system [36, 31]. As a result, the domain specific language and model in Section 4.2 is more focused on the actual trusted computing system rather than adapting a logic for protocols to reason over mostly local system properties.

An important side-effect of externalizing and adding functionality carefully is that proofs in LS^3 can be embedded in existing proofs [36]. Performing the proofs is nevertheless tedious and a result of LS^3 sharing its logic with temporal

logics [2, 3, 5, 4] and protocol composition logics [35]. A much better alternative is needed to design and verify complex systems at scale. More practical alternatives to temporal logics could be linear logic or session type approaches [54, 21]. Exchanging the logic behind LS^3 with one which is supported by automated theorem provers has many advantages in terms of rigor and practicality. The thesis makes no contributions in this aspect. The focus on new actions in the language and support for system isolation in the operational semantics allowed this thesis to rapidly design new systems with LS^3 while ensuring that designs can also be verified. A fair judgement would be that this thesis improves on abstraction and modeling capabilities while it relies on a large background and formal foundations.

Another concern is LS^3's input language (i.e., the programming interface) draws from simple constructs of [94]. Similarly, LS^3's operational semantics are based on process-calculi developed in [94, 91, 92, 42, 43] which represent a common way to model transitions in a system for certain actions. In comparison, LS^3's language definitions are much simpler and only hint at certain types while a type system is missing entirely. Generally, a sophisticated type system could be used to further constrain certain actions in LS^3, which is expected to simplify verification efforts in the future.

Despite its focus on expressiveness, LS^3 lacks an important aspect: secure enclaves and trusted execution environments. Trusted execution using "attested execution secure co-processors" has been investigated and formally abstracted in [106] using the UC-Framework [22]. The treatment of computations using secure co-processors is exemplary but the requirements for the trusted environment in [126, 106, 30, 63] are vastly different from the work in this thesis. In LS^3 all software environments are initially treated as untrusted. Lastly, [106] does not offer a programming interface and does not consider potentially untrusted or adversary supplied code. In comparison, the work in Chapter 4 aims to prove a weaker integrity property but does so without relying on the heavy hardware assumptions of trusted execution environments.

The work by Rowe et al. in [114] presents another line of research which aims to establish certain trusted computing principles in very general setting. The formal model which is used to support the arguments helps to formally

express and prove some intuitive arguments. However, there is no way to model and evaluate existing systems in any detail. The models proposed in this thesis are considered superior for this reason. Nevertheless, the general principles which [114] aims to formally prove are intuitively sound. Prior to this, [58] have analyzed TPM based protocols using a formal automata model based on asynchronous product automata to emulate a TPM within an executable model and have suggested improvements. The result of this was a much more comprehensive TPM abstraction than the one offered in this thesis. In fact, this thesis makes little efforts to model the TPM (or a vTPM) closer and uses it as a primitive for secure systems instead. Another thorough modeling effort of the TPM is presented in [98] which aims to provide a verifiable reference implementation of the TPM. In summary, a consequence of treating the TPM as a primitive is that the TPM design itself remains unaffected in this thesis.

Finally, the verified security kernel seL4 [74] demonstrates the first successful formal verification of a trustworthy micro-kernel operating system. The rigor with which seL4's design and verification are treated is unparalleled at this scale. In fact, the proofs which support seL4's functional properties outweigh the its actual lines of code by several orders of magnitude. Verification at such a scale has lead to several innovations towards automated verification. In comparison, the work presented in this thesis follows the idea of formally verifying designs first and allows dealing with much more abstract states. Consequently, many of the functional properties (e.g., process isolation) of seL4 are taken for granted in the model of LS^3. Interestingly, [99] was able to prove abstract security properties on top of functional correctness proofs which is an interesting avenue for further research [51, 59]. A similar exploration is missing in this thesis but is still possible in future work.

Bigraphical Trust Model

The work in Chapter 5 is best compared to prior work by Parno [105]. Parno makes the simple point that it may be impossible to establish trust in an unknown and therefore untrusted platform. During the investigation, the paper concludes that without prior knowledge or additional arrangements anyone

can be fooled into establishing trust in an untrusted computer. The TPM at that time did not offer an explicit way to prevent this. The model which is needed to convey this point is simple and involves only machines (i.e., physical platforms), TPMs, and agents which may be malicious. The work presented in Section 5.3, i.e., the formalization of the *Goldeneye* attack, is only possible if a more complex virtualization system and the virtual platform are considered. For this reason, the system model in Section 5.2 adopts several software system components as building blocks which are drawn from specifications such as [102]. In this model, abstract components needed enrichment such as recovering their ability to be composed. This turned a static component definition [102] into a functional model which can express arbitrarily complex systems (Section 5.3.2). Furthermore, the definition of trusted computing axioms (Table 5.2) and predicates (Table 5.1) is also more complex when compared to prior work [105]. Combining bigraphical structures with the predicate calculus in Section 5.3.2 is another contribution which for the first time allowed reasoning over security properties in bigraphs. In summary, the model and the analysis of Chapter 5 is much more sophisticated compared to other works because the targeted systems and the argument itself is more complex.

However, the solutions proposed in Section 5.4.2 will benefit from proving them effective using the formal model. Currently, the formal model can only be used for the analysis of commodity systems and does not have primitive for some of the proposed techniques (e.g., trusted execution environments). Lastly, bigraphs are currently studied in the context of category theory. This thesis had to use the expressiveness of Milner's bigraphs [93] conservatively in order to make them immediately useful as a modeling tool. Bigraphs as a formalism for modeling are applied elsewhere but today the work in Chapter 5 is the first to apply bigraphs in a computer security context. A consequence of the theoretical work in Section 5.3.2 is that relevant standardization groups are also considering exploiting the inherent correspondence between graphical representations (i.e., pictures) and concrete foundations. The refinement of current layer and block diagrams but also bridging gaps between informative descriptions and normative statements in [56] is a next step.

6.2 Future Work

The work presented in this thesis may be continued in several ways which include improvements to certain theoretical and practical contributions but also related research areas which were not in the scope of this thesis.

This thesis has not investigated system management, management controllers, orchestration and the so called *cloud* layers of systems. The introduction in Chapter 3 gives a brief overview of a cloud system and shows that cloud systems include several components which are not part of the virtualization system but provide important functionality and fidelity. Our formal system in Chapter 4 has constructs to run containers but is not at all concerned with their origin or how they are managed, shutdown, and migrated. In short, this thesis does not present insights into orchestration despite its focus on trusted virtualization systems. Orchestration of physical and virtual nodes is not only essential for operations but contributes significantly towards a cloud systems security, availability, and reliability. The work presented in [120] treats cloud systems as a form of distributed systems and is actively considering orchestration at the expense of the precision with which virtualization systems are treated in Chapters 3 to 5. Providing the same kind of precision and modeling is certainly an area of active and future research.

Another clear direction for future work is the application of the concepts developed in this thesis to specific hardware and software components. The thesis generally abstracts commodity hardware and systems and claims that it can be applied to a variety of them. Implementing the measurement architecture of Chapter 4 using Linux as the *container OS* with the popular Docker [39] engine for containers. This can be paired with the vTPM and Virtual Platform constructs proposed in Chapter 5 using the open-source trusted computing software stack and virtual TPM implementation [13]. The goal of such implementation would be to demonstrate the feasibility of the concepts, document the engineering effort, and evaluate time and space complexities of the implementations.

The inclusion of roots of trust other than the TPM for storage, reporting, or secure code execution such as secure enclaves and trusted execution environments in secure systems is already popular and may be a viable choice for future

virtual trusted platforms. Major virtualization providers and hardware vendors are already beginning to provide their own roots of trust implementation as a secure system on a chip (SoC) (e.g. Apple's Secure Enclave and T2 [8] or Google's Titan chip [103]. Furthermore, extended processor and SoC capabilities for trusted computing can help provide solutions which avoid extending the trusted computing base or software dependencies. Such solutions also support the basic trusted computing mechanisms discussed in this thesis. This allows development of portable architectures [56] which rely on vendor specific roots of trust and thus allow the construction of virtual trusted platforms using an even wider variety of physical platforms. Lastly, this thesis provides concepts for establishing secure and trusted *software* environments with hardware support. Silicon based alternatives intend to provide environments with very similar properties but with better hardware features and less software dependencies. There is an inherent similarity between the concepts for software systems, e.g., remote attestation and trusted boot, and the ones which which get embedded in hardware such that we expect useful synergy effects from a better hardware and software co-design.

In Chapter 3 we propose *User-Centered Attestation* by reviewing prior work and proposals [79] as a set of principles with an outlook on how to design and implement it. User-Centered Attestation is a novel attestation system which aims at establishing trust in individual virtualization systems. While we solve the issue of trustworthy measurement architectures and constructing virtual trusted platforms in Chapters 4 and 5, details of the remote attestation procedure are still open issues.

The work presented in Chapter 4 introduces both LS^3 as a formal system and an *Enhanced Integrity Measurement Architecture* (EIMA). While the overall design goal of including containers as virtual environments and recording them separately was achieved, several alternatives and improvements are possible. One prominent investigation would consider a vTPM assigned to each container and hosting a vTPM on the container OS. As mentioned earlier, in a hybrid setting this would imply nesting virtual TPMs and virtual platforms which is not well researched.

When EIMA is deployed, the PCRs used to record the container OS are regular static PCRs. Dynamic PCRs are used to record containerized applications. Consequently, containers can be spun up, migrated, and destructed rapidly without making static PCR values brittle. The two PCR kinds are currently linked and investigating a new TPM command for this purpose is an interesting proposal, especially when a vTPM is used.

Lastly, EIMA needs support in a remote attestation protocol. EIMA is an integrity measurement architecture and acts as a root of trust for measurements (RTM) and with the (v)TPM as the tool for reporting, suitable protocols which use integrity information locally or remotely need to be developed. While only minimal changes to the configuration of existing protocols are expected, a deeper investigation is still needed.

LS^3 includes several new commands and system model properties to support reasoning over stratified and isolated system compartments. Prior work has a much stronger focus more on the formal system. Thus, combining existing work in theorem proving and formal foundations for secure systems with model and semantics of LS^3 may yield several benefits in terms of applicability, expressiveness, and overall rigor.

The work in Chapter 5 is heavily influenced by Parno's prior work [105]. This thesis manages to surpass a large body of prior work in several ways: Most notably, the sophisticated model which allows reasoning over complex trusted computing systems and architectures. However, the work presented in Chapter 5 lacks an application of the graphical and formal model to the proposed solutions in Section 5.4.2. Certain solutions can be modeled and represented graphically which in turn allows us to use trusted computing axioms directly. Other solutions will require modeling further secure hardware capabilities which is a definitive item for future work.

From a theoretical standpoint, the formal model in Section 5.3.2 utilizes only a small subset of the expressiveness of bigraphs [93]. All systems which are investigated are abstracted as snapshots or static images of otherwise dynamic systems. However, bigraphs are intended to model both static and dynamic properties and further analysis which treats software components more like mobile agents is needed. A promising avenue for such an investigation is

the combination of LS^3's actions with the bigraphical model and reasoning capabilities in Section 5.3.2. LS^3 supports constructs to create new nodes (e.g., containers and apps) and our bigraphical system in Chapter 5 supports reasoning over the resulting graph. To our knowledge, such a system would be uniquely practical and would come furnished with concrete foundations. Vice versa, the inclusion of containers in the bigraphical model Chapter 5 is a necessary next step. Containers are currently omitted from our analysis as related work does not concern containers — yet. However, containers are already a part of an update to the system model and fit well into the system and formal model.

So far the association between vTPM an VM's or *virtual trusted platform integrity* has been discussed as a unique problem of the virtual platform. In hindsight, platform integrity is not a problem unique to the virtual platform: manufacturers and OS vendors are increasingly concerned about adversaries tampering with CPU, TPM, and storage combinations to circumvent soft- and hardware data protection tools. One way to address this concern is to establish session secrets between components based on their unique identities. This concept is named *link encryption* and can provide confidentiality on BUS communications and a form of authentication between components. Future work will have to discuss cryptographic approaches to platform integrity and explore some of these options in depth for virtual platforms.

As discussed in Section 6.1.1, this work has made notable contributions towards a *modern* virtual trusted platform by (1) placing it into a distributed systems context, (2) including containerization in a canonical and trustworthy way, and (3) discussing trust establishment and virtual platform integrity in great depth. Future work on a virtual trusted platform must be concerned with rooting virtual platforms in physical ones which may be related to virtualizing the TPM [14]. The theoretical and practical aspects for a solution are unclear but there is a clear security benefit and commercial interest. Virtual platforms which are *bound* to physical ones benefit from (i) additional security guarantees and assumption about the physical platform, (ii) cannot be cloned uncontrolled or run on downgraded physical platforms, and (iii) certain secrets can enjoy the protection of the hardware roots of trust. From a commercial perspective, binding virtual platforms to physical ones enables a plethora of deployment

models where vendors can issue a virtual platform tailored for a customer's physical platforms, improve software licensing options, and protect virtual solutions from being reproduced illegally.

Chapter 7

Conclusion

Virtualization poses an interesting issue for specifications towards trustworthy systems as the *trust* placed originally only in hardware components needs to be extended to reporting and measurement mechanisms in upper layers and virtual environments. While approaches towards trust establishment exist, their semantics are ambiguous and an appraiser has to decide whether a virtualization system or upper layers are trusted without much guidance or support in reasoning for such a decision. Furthermore, existing attestation approaches imply a particular topology, connectivity, and capability that does not reflect decentralized systems and prevents clients from establishing trust in their software systems. A *User-Centered Attestation*, as a novel attestation system, encompasses these concerns and proposes a strategy for specifying and synthesizing suitable trust establishment mechanisms and inspired further research and contributions towards open and collaborative trusted computing systems.

We present the design and verification of an enhanced integrity measurement architecture (EIMA). EIMA addresses the trustworthiness and constrained disclosure of integrity measurements for containerized systems which allows recording and reporting domain specific measurements to clients. Our development and design was aided by a precise formal model of stratified systems. The formal model was conceived by adding required constructs for domains and domain specific measurements to an existing and established formal model.

The additions allowed a proof of a trustworthy measurement architecture from system boot all the way up to containerized applications. The proof of EIMA has also shown the security properties of the commonly used Integrity Measurement Architecture (IMA) [117]. However, EIMA does not reveal information about a target container to other untrusted tenants on the same system. Future work will have to address more detailed interactions between otherwise isolated domains and sub-domains via system call interfaces and privileged administrator commands. The verification of properties which were deferred to the TPM is an important task and a suitable remote attestation protocol is still needed.

Our work in LS^3 relies on either a TPM or a vTPM to record and report integrity measurements of software running in a container or other virtual environment. We found that creating a "trusted" virtual platform is necessary for a variety of secure cloud computing scenarios. Ideally, we should be able to use secure hardware such as the TPM to bootstrap trust in a virtualization system and utilize a virtual TPM as a (virtual) root of trust for a virtual machine. However, our model and analyses reveal that the process of extending trust from the TPM all the way up into a VM is vulnerable to a *Goldeneye* attack. The *Goldeneye* attack emphasizes the importance of a trusted association between VM and vTPM. The association between a system and its root of trust is typically not considered on a real machine but we emphasize that it needs special considerations in case of a VM and a vTPM. We demonstrated attack vectors present in recent papers and proposed solutions which could be implemented today. Current systems rely on assuming an invariant about the association between VMs and vTPMs. We suggest and discuss several solutions including constraining interfaces, changes to measurement architectures, and vTPM architectures. A clean solution in the future might be to provide architectural support for roots of trust for software in virtual environments.

The discussion of the thesis reveals that our work achieves major goals and makes significant contributions towards a *modern* virtual trusted platform. Virtual environments are placed in the perspective of a distributed system which is necessary as large amounts of services and infrastructure are migrated to virtual platforms. To support the attestation of software in virtual environments as

well as the virtualization system, we design EIMA and verify it against specified security properties. Lastly, our systematic analysis reveals that the integrity of a virtual platform is an important security property and that solutions need to be implemented in order to detect or prevent tampering with virtual trusted platforms.

References

[1] Akram, R. N. and Ko, R. K. L. (2014). Digital trust - trusted computing and beyond: A position paper. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 884–892.

[2] Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

[3] Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123 – 154.

[4] Allen, J. F. and Ferguson, G. (1994). Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4:531–579.

[5] Allen, J. F. and Hayes, P. J. (1989). Moments and points in an interval-based temporal logic. In *Contexts in context. In Contexts in Knowledge Representation and Natural Language, Proceedings of the 1997 AAAI Spring Symposium, Menlo Park, Calif*, pages 225–238. AAAI Press.

[6] Almorsy, M., Grundy, J. C., and Mueller, I. (2016). An analysis of the cloud computing security problem. *CoRR*, abs/1609.01107.

[7] Amazon (2020). Improving Security with Cloud Computing: Six Advantages of Cloud Security. https://aws.amazon.com/blogs/publicsector/improving-security-with-cloud-computing-six-advantages-of-cloud-security/. Accessed: 2020-01-28.

[8] Apple Inc. (2020). About the Apple T2 Security Chip. https://support.apple.com/en-us/HT208862. Accessed: 2020-01-7.

[9] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

[10] Azab, A. M., Ning, P., Sezer, E. C., and Zhang, X. (2009). Hima: A hypervisor-based integrity measurement agent. In *2009 Annual Computer Security Applications Conference*, pages 461–470, San Juan, USA. ACSA.

[11] Bai, G., Hao, J., Wu, J., Liu, Y., Liang, Z., and Martin, A. (2014). Trustfound: Towards a formal foundation for model checking trusted computing platforms. In Jones, C., Pihlajasaari, P., and Sun, J., editors, *FM 2014: Formal Methods*, pages 110–126, Cham. Springer International Publishing.

[12] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA. ACM.

[13] Berger, S. (2020). Software TPM. https://github.com/stefanberger/swtpm. Accessed: 2020-01-6.

[14] Berger, S. et al. (2006). vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA. USENIX Association.

[15] Beth, T., Borcherding, M., and Klein, B. (1994). Valuation of trust in open networks. In *Proceedings of the Third European Symposium on Research in Computer Security*, ESORICS '94, pages 3–18, London, UK, UK. Springer-Verlag.

[16] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA. ACM.

[17] Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Arvind, and Devadas, S. (2018). Mi6: Secure enclaves in a speculative out-of-order processor.

[18] Bräuner, T. (2011). *Hybrid Logic and its Proof-Theory*. Springer-Verlag, Dordecht, Germany.

[19] Brickell, E., Camenisch, J., and Chen, L. (2004). Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 132–145, New York, NY, USA. ACM.

[20] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD. USENIX Association.

[21] Caires, L. and Pfenning, F. (2010). Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, pages 222–236. Springer.

[22] Canetti, R. (2001). Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 136–145.

[23] Chen, C., Raj, H., Saroiu, S., and Wolman, A. (2014). ctpm: A cloud TPM for cross-device trusted applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA. USENIX Association.

[24] Chen, L., El Kassem, N., Lehmann, A., and Lyubashevsky, V. (2019). A framework for efficient lattice-based daa. In *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race*, CYSARM'19, pages 23–34, New York, NY, USA. ACM.

[25] Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. In Takagi, T. and Peyrin, T., editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham. Springer International Publishing.

[26] Chiang, M. and Zhang, T. (2016). Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864.

[27] Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68.

[28] Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., and Sniffen, B. (2011). Principles of remote attestation. *Int. J. Inf. Secur.*, 10(2):63–81.

[29] Conforti, G. et al. (2005). Spatial logics for bigraphs. In *Automata, Languages and Programming*, pages 766–778, Berlin, Heidelberg. Springer Berlin Heidelberg.

[30] Costan, V., Lebedev, I., and Devadas, S. (2016). Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX. USENIX Association.

[31] Cremers, C. (2008). On the protocol composition logic pcl. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 66–76.

[32] Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. (2006). Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 79–88, New York, NY, USA. ACM.

[33] Dall, F., De Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., and Yarom, Y. (2018). Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):171–191.

[34] Danev, B., Masti, R. J., Karame, G. O., and Capkun, S. (2011). Enabling secure vm-vtpm migration in private clouds. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 187–196, New York, NY, USA. ACM.

[35] Datta, A., Derek, A., Mitchell, J. C., and Roy, A. (2007). Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172:311–358.

[36] Datta, A., Franklin, J., Garg, D., and Kaynar, D. (2009). A logic of secure systems and its application to trusted computing. In *2009 30th IEEE Symposium on Security and Privacy*, pages 221–236.

[37] Devriese, D., Birkedal, L., and Piessens, F. (2016). Reasoning about object capabilities with logical relations and effect parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 147–162.

[38] Dijkstra, E. W. (1979). Programming considered as a human activity. In Yourdon, E. N., editor, *Classics in Software Engineering*, pages 1–9. Yourdon Press, Upper Saddle River, NJ, USA.

[39] Docker Inc. (2020). Docker homepage. https://www.docker.com. Accessed: 2020-01-15.

[40] England, P. and Loeser, J. (2008). Para-virtualized tpm sharing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Trust '08, pages 119–132, Berlin, Heidelberg. Springer-Verlag.

[41] Fan, J. and Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.

[42] Felleisen, M., Friedman, D. P., Kohlbecker, E. E., and Duba, B. F. (1986). Reasoning with continuations. In *Proc. First Symposium on Logic in Computer Science*, pages 131–141.

[43] Felleisen, M. and Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271.

[44] Firesmith, D. (2020a). Virtualization via Containers. https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html. Accessed: 2020-01-20.

[45] Firesmith, D. (2020b). Virtualization via Virtual Machines. https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-virtual-machines.html. Accessed: 2020-01-20.

[46] Floyd, R. W. (1993). *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht.

[47] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009.

[48] Gallery, E. and Mitchell, C. J. (2009). Trusted computing: Security and applications. *Cryptologia*, 33(3):217–245.

[49] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., and Riviere, E. (2015). Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42.

[50] Gasser, M., Goldstein, A., Kaufman, C., and Lampson, B. (1989). The digital distributed system security architecture. In *National Computer Security Conf., NIST/NCSC, Baltimore*, volume 12, pages 305–319. National Institute of Standards and Technology.

[51] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. (2018). A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27.

[52] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA. ACM.

[53] Gentzen, G. (1935). Untersuchungen über das logische schliessen i+ii. *Mathematische Zeitschrift*, 39:176–210.

[54] Girard, J.-Y. (1987). Linear logic. *Theoretical computer science*, 50(1):1–101.

[55] Gligor, V. (2014). Dancing with the adversary: a tale of wimps and giants. In *Cambridge International Workshop on Security Protocols*, pages 100–115. Springer.

[56] Group, T. C. (2011). *Virtualized Trusted Platform Architecture Specification*. Trusted Computing Group. Rev. 1.26.

[57] Group, T. C. (2018). Trusted platform module 2.0 library specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/. Accessed: 2018-10-31.

[58] Gürgens, S., Rudolph, C., Scheuermann, D., Atts, M., and Plaga, R. (2007). Security evaluation of scenarios based on the tcg's tpm specification. In *Proceedings of the 12th European Conference on Research in Computer Security*, ESORICS'07, pages 438–453, Berlin, Heidelberg. Springer-Verlag.

[59] Heiser, G., Klein, G., and Murray, T. (2019). Can we prove time protection? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 23–29, New York, NY, USA. ACM.

[60] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[61] Huang, J. and Fox, M. S. (2006). An ontology of trust: formal semantics and transitivity. In *ICEC*.

[62] IBM (2020). Benefits of Cloud Computing. https://www.ibm.com/cloud/learn/benefits-of-cloud-computing. Accessed: 2020-01-28.

[63] Intel (2016). Intel software guard extensions (sgx). https://software.intel.com/en-us/sgx. Accessed: 2018-10-31.

[64] ISO (2015a). Trusted platform module library. ISO ISO/IEC 11889-1:2015, International Organization for Standardization, Geneva, Switzerland.

[65] ISO (2015b). Trusted platform module library. ISO ISO/IEC 11889-1:2015, International Organization for Standardization, Geneva, Switzerland.

[66] ISO (2018). ISO/IEC np 27070 information technology – security techniques – security requirements for establishing virtualized roots of trust. https://www.iso.org/standard/56571.html. Accessed: 2018-10-31.

[67] ISO (2020). ISO/IEC 27070 — information technology — security techniques — security requirements for virtualized roots of trust [draft]. https://www.iso27001security.com/html/27070.html. Accessed: 2020-01-20.

[68] Jangda, A., Pinckney, D., Baxter, S., Devore-McDonald, B., Spitzer, J., Brun, Y., and Guha, A. (2019). Formal foundations of serverless computing. *arXiv preprint arXiv:1902.05870*.

[69] Jia, L., Sen, S., Garg, D., and Datta, A. (2015). A logic of programs with interface-confined code. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 512–525.

[70] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. (2019). Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv e-prints*, page arXiv:1902.03383.

[71] Juglaret, Y., Hritcu, C., Amorim, A. A. D., Eng, B., and Pierce, B. C. (2016). Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 45–60.

[72] Kamvar, S. D., Schlosser, M. T., and Garcia-Molina, H. (2003). The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 640–651, New York, NY, USA. ACM.

[73] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. (2014). Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372.

[74] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA. ACM.

[75] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[76] Kreps, D. M., Milgrom, P., Roberts, J., and Wilson, R. (1982). Rational cooperation in the finitely repeated prisoners' dilemma. *Journal of Economic theory*, 27(2):245–252.

[77] Lampson, B. W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615.

[78] Lampson, B. W. (1984). Hints for computer system design. *IEEE Software*, 1(1):11.

[79] Lauer, H. and Kuntze, N. (2016). Hypervisor-based attestation of virtual environments. In *Advanced and Trusted Computing (ATC), 2016 Intl IEEE Conferences*, pages 333–340, Toulouse. IEEE, IEEE.

[80] Lauer, H., Rudolph, C., and Nepal, S. (2018). User-centered attestation for layered and decentralized systems. In *Network and Distributed Systems Security (NDSS) Symposium 2018, Workshop on Decentralized IoT Security and Standards (DISS), 18-21 February 2018, San Diego, CA, USA*. ISOC.

[81] Lauer, H., Rudolph, C., and Nepal, S. (2020). Design and analysis of a modern virtual trusted platform. *TBD*, pages 1–30.

[82] Lauer, H., Rudolph, C., Nepal, S., and Sakzad, A. (2019a). Bootstrapping trust in a "trusted" virtualized platform. In *Workshop on Cyber-Security Arms Race (CSYARM) 2019, The 26th ACM Conference on Computer and Communications Security (CCS), 15 November 2019, London, UK*. ACM.

[83] Lauer, H., Sakzad, A., Rudolph, C., and Nepal, S. (2019b). A logic for secure stratified systems and its application to containerized systems. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications*, pages 1–8, Rotorua, New Zealand. IEEE.

[84] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[85] Liu, L. and Shi, W. (2010). Trust and reputation management. *IEEE Internet Computing*, 14(5):10–13.

[86] Marti, S. and Garcia-Molina, H. (2006). Taxonomy of trust: Categorizing p2p reputation systems. *Comput. Netw.*, 50(4):472–484.

[87] Martin, A. (2008). The ten-page introduction to trusted computing.

[88] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. (2010). Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA. IEEE Computer Society.

[89] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. (2008). Flicker: An execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328.

[90] Microsoft Corporation (2020). Trusted Platform Module Technology Overview. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview. Accessed: 2020-01-14.

[91] Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer.

[92] Milner, R. (1999). *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[93] Milner, R. (2009). *The Space and Motion of Communicating Agents*. Cambridge University Press, New York, NY, USA, 1st edition.

[94] Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[95] Mitchell, C. (2005). *Trusted Computing*, volume 6. Iet.

[96] Moghimi, D., Sunar, B., Eisenbarth, T., and Heninger, N. (2020). Tpm-fail: TPM meets timing and lattice attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA. USENIX Association.

[97] Morabito, R. (2017). Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850.

[98] Mukhamedov, A., Gordon, A. D., and Ryan, M. (2009). Towards a verified reference implementation of a trusted platform module. In *International Workshop on Security Protocols*, pages 69–81. Springer.

[99] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., and Klein, G. (2013). sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429.

[100] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. (2017). Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 252–269, New York, NY, USA. ACM.

[101] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73.

[102] NIAP (2016). Protection profile for virtualization. Protection Profile 1, National Information Assurance Partnership, Maryland, USA.

[103] OpenTitan (2020). OpenTitan github hompage. https://github.com/lowRISC/opentitan. Accessed: 2020-01-7.

[104] Orbaek, P. and Palsberg, J. (1997). Trust in the lambda-calculus. *J. Funct. Program.*, 7(6):557–591.

[105] Parno, B. (2008). Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 9:1–9:6, Berkeley, CA, USA. USENIX Association.

[106] Pass, R., Shi, E., and Tramer, F. (2017). Formal abstractions for attested execution secure processors. In *EUROCRYPT (1)*, pages 260–289. Springer.

[107] Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421.

[108] Popovic, K. and Hocenski, Z. (2010). Cloud computing security issues and challenges. In *The 33rd International Convention MIPRO*, pages 344–349.

[109] Raj, H., Saroiu, S., Wolman, A., Aigner, R., Cox, J., England, P., Fenner, C., Kinshumann, K., Loeser, J., Mattoon, D., Nystrom, M., Robinson, D., Spiger, R., Thom, S., and Wooten, D. (2016). ftpm: A software-only implementation of a TPM chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, Austin, TX. USENIX Association.

[110] Raza, A. et al. (2019). Unikernels: The next stage of linux's dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 7–13, New York, NY, USA. ACM.

[111] Rein, A. (2017). Drive: Dynamic runtime integrity verification and evaluation. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 728–742, New York, NY, USA. ACM.

[112] Richardson, M., Agrawal, R., and Domingos, P. (2003). Trust management for the semantic web. In *Proceedings of the Second International Conference on Semantic Web Conference*, LNCS-ISWC'03, pages 351–368, Berlin, Heidelberg. Springer-Verlag.

[113] Roscher, S., Boenisch, V., Lee, J., Zeisberg, D., and Schweflinghaus, J. (2018). Integrating solutions on ibm z with secure service container. *IBM J. Res. Dev.*, 62(2-3):3:1–3:6.

[114] Rowe, P. D. (2016). *Bundling Evidence for Layered Attestation*, pages 119–139. Springer International Publishing, Cham.

[115] Ruan, A. and Martin, A. (2011). Repcloud: Achieving fine-grained cloud tcb attestation with reputation systems. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, pages 3–14, New York, NY, USA. ACM.

[116] Ryan, M. D. (2013). Cloud computing security: The scientific challenge, and a survey of solutions. *Journal of Systems and Software*, 86(9):2263 – 2268.

[117] Sailer, R., Zhang, X., Jaeger, T., and Van Doorn, L. (2004). Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238.

[118] Salehi, A., Lauer, H., Grobler, M., Rudolph, C., and Sakzad, A. (2020). Access control, key and database management, and trust for emerging wireless body area networks in healthcare application. *IEEE Journal of Biomedical and Health Informatics*, pages 1–15.

[119] Santos, N., Rodrigues, R., Gummadi, K. P., and Saroiu, S. (2012). Policy-sealed data: A new abstraction for building trusted cloud services. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, Bellevue, WA. USENIX.

[120] Schear, N., Cable, II, P. T., Moyer, T. M., Richard, B., and Rudd, R. (2016). Bootstrapping and maintaining trust in the cloud. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 65–77, New York, NY, USA. ACM.

[121] Shi, E., Perrig, A., and Doorn, L. V. (2005). Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 154–168, Washington, DC, USA. IEEE Computer Society.

[122] Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646.

[123] Shi, W. and Dustdar, S. (2016). The promise of edge computing. *Computer*, 49(5):78–81.

[124] Sinha, A., Jia, L., England, P., and Lorch, J. R. (2014). Continuous tamper-proof logging using tpm 2.0. In *Proceedings of the 7th International Conference on Trust and Trustworthy Computing - Volume 8564*, pages 19–36, New York, NY, USA. Springer-Verlag New York, Inc.

[125] Son, J., Koo, S., Choi, J., Choi, S.-j., Baek, S., Jeon, G., Park, J.-H., and Kim, H. (2017). Quantitative analysis of measurement overhead for integrity verification. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1528–1533, New York, NY, USA. ACM.

[126] Subramanyan, P., Sinha, R., Lebedev, I., Devadas, S., and Seshia, S. A. (2017). A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2435–2450, New York, NY, USA. ACM.

[127] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[128] Thompson, K. (1984). Reflections on trusting trust. *Commun. ACM*, 27(8):761–763.

[129] Trusted Computing Group (2013). *Trusted Multi-Tenant Infrastructure Reference Framework*. Trusted Computing Group. Rev. 1.

[130] Trusted Computing Group (2014). *TCG EK Credential Profile*. Accessed: 2019-05-01.

[131] Trusted Computing Group (2018). *Implicit Idendity Based Device Attestation*. Trusted Computing Group. Rev. .93.

[132] Trusted Computing Group (2020a). Trusted Computing Group (homepage). https://trustedcomputinggroup.org. Accessed: 2020-01-30.

[133] Trusted Computing Group (2020b). Trusted Network Connect Resources. https://trustedcomputinggroup.org/work-groups/trusted-network-communications/tnc-resources/. Accessed: 2020-01-22.

[134] Trusted Computing Group (2020c). Virtualized Platform Working Group (VP WG). https://trustedcomputinggroup.org/work-groups/virtualized-platform/. Accessed: 2020-01-30.

[135] U.S.A. National Computer Security Center (1985). *Computer Security Requirements: Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments (The Orange Book)*. DoD Computer Security Center.

[136] Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F. D., and Piessens, F. (2019). A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1741–1758, New York, NY, USA. ACM.

[137] Wadler, P. (2015). Propositions as types. *Commun. ACM*, 58(12):75–84.

[138] Xen Project (2015). Xen vTPM Wiki. https://wiki.xenproject.org/wiki/Virtual_Trusted_Platform_Module_(vTPM). Accessed: 2020-01-14.

[139] Xiu, D. and Liu, Z. (2005). *A Formal Definition for Trust in Distributed Systems*, pages 482–489. Springer Berlin Heidelberg, Berlin, Heidelberg.

[140] Yap, J. Y. and Tomlinson, A. (2013). Para-virtualizing the trusted platform module: An enterprise framework based on version 2.0 specification. In *Proceedings of the 5th International Conference on Trusted Systems - Volume 8292*, INTRUST 2013, pages 1–16, Berlin, Heidelberg. Springer-Verlag.

[141] Yi, S., Li, C., and Li, Q. (2015). A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, pages 37–42, New York, NY, USA. ACM.

[142] Zhang, F., Chen, J., Chen, H., and Zang, B. (2011). Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA. ACM.

[143] Zhou, Z., Yu, M., and Gligor, V. D. (2014). Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323. IEEE.

[144] Zissis, D. and Lekkas, D. (2012). Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583 – 592.