Behavior Compliance Control for More Trustworthy Computation Outsourcing

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des Grades
Doktor Ingenieur (Dr.-Ing.)

von

Dipl.-Inform. Sami Alsouri

geboren in Al Mshairfeh, Jordanien.

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Referenten:	Prof. Dr. Stefan Katzenbeisser
	Technische Universität Darmstadt
	Prof. Dr. Eric Bodden
	Fraunhofer-Institut für Sichere Informationstechnologie
Tag der Einreichung:	20.06.2013
Tag der mündlichen Prüfung:	02.08.2013
Hochschulkennziffer:	D17

Darmstadt 2013

Abstract

Computation outsourcing has become a hot topic in both academic research and industry. This is because of the benefits accompanied with outsourcing, such as cost reduction, focusing on core businesses and possibility for benefiting from modern payment models like the pay-per-use model. Unfortunately, outsourcing to potentially untrusted third parties' hosting platforms requires a lot of trust. Clients need assurance that the intended code was loaded and executed, and that the application behaves correctly and trustworthy at runtime. That is, techniques from Trusted Computing which are used to allow issuing evidence about the execution of binaries and reporting it to a challenger are not sufficient. Challengers are more interested in evidence which allows detecting misbehavior while the outsourced computation is running on the hosting platform.

Another challenging issue is providing a secure data storage for collected evidence information. Such a secure data storage is provided by the Trusted Platform Module (TPM). In outsourcing scenarios where virtualizations technologies are applied, the use of virtual TPMs (vTPMs) comes into consideration. However, researcher identified some drawbacks and limitations of the use of TPMs. These problems include privacy and maintainability issues, problems with the sealing functionality and the high communication and management efforts. On the other hand, virtualizing TPMs, especially virutalizing the Platform Configuration Registers (PCRs), strikes against one of the core principles of Trusted Computing, namely the need for a hardware-based secure storage.

In this thesis, we propose different approaches and architectures which can be used to mitigate the problems above. In particular, in the first part of our thesis we propose an approach called Behavior Compliance Control (BCC) to defines architectures to describe how the behavior of such outsourced computations is captured and controlled as well as how to judge the compliance of it compared to a trusted behavior model. We present approaches for two abstraction levels; one on a program code level and the other is on the level of abstract executable business processes.

In the second part of this thesis, we propose approaches to solve the aforementioned problems related to TPMs and vTPMs, which are used as storage for evidence data collected as assurance for behavior compliance. In particular, we recognized that the use of the SHA-1 hash to measure system components requires maintenance of a large set of hashes of presumably trustworthy software; furthermore, during attestation, the full configuration of the platform is revealed. Thus, our approach shows how the use of chameleon hashes allows to mitigate the impact of these two problems. To increase the security of vTPM, we show in another approach how strength of hardware-based security can be gained in virtual PCRs by binding them to their corresponding hardware PCRs. We propose two approaches for such a binding. For this purpose, the first variant uses binary hash trees, whereas the other variant uses incremental hashing.

We further provide implementations of the proposed approach and evaluate their impact in practice. Furthermore, we empirically evaluate the relative efficacy of the different behavioral abstractions of BCC that we define based on different real world applications. In particular, we examined the feasibility, the effectiveness, the scalability and efficiency of the approach. To this end, we chose two kinds of applications, a web-based and a desktop application, performing different attacks on them, such as malicious input attach and SQL injection attack. The results show that such attacks can be detected so that the application of our approach can increase the protection against them.

Zusammenfassung

Auslagerung (Outsourcing) von Geschäftsprozessen ist ein heißes Thema geworden, sowohl in der akademischen Forschung als auch in der Industrie. Dies ist wegen der Vorteile, die das Outsourcing mit sich bringt, wie z.B. Kostenreduzierung, Fokussierung auf das Kerngeschäft und die Möglichkeit von modernen Zahlungsmodellen zu profitieren, wie z.B. das Pay-per-Use-Modell.

Leider ist das Outsourcing zu nicht notwendigerweise vertrauenswürdigen Hosting-Platform erfordert viel Vertrauen. Kunden brauchen die Gewissheit, dass der beabsichtigte Code nicht nur geladen und ausgeführt wird, sondern auch dass sich der Code zur Laufzeit richtig und wie gewünscht verhält. Das heißt, Techniken aus der Trusted Computing die angewendet werden, um Beweise über die Ausführung bestimmer Programme zu erstellen und zu einem Herrausforderer auszuliefern, sind nicht ausreichend. Viel mehr sind Herrausforderer daran interessiert, Missverhalten eines ausgelagerten Programms zu entdecken und entsprechend zu reagieren.

Ein weiteres relevantes Thema ist die Bereitstellung eines Laufzeit-sicheren Speichers, der zum Speichern von gesammelten Beweisdaten verwendet wird. Eine solche sichere Datenspeicherung wird bereitgestellt durch das Trusted Platform Module (TPM). In Outsourcing-Szenarien, in denen Virtualisierungstechnologien zum Einsatz kommen, werden virtuelle TPMs (vTPMs) benutzt um die Funktionalitäten eines wirklichen TPMs in virtualisierten Umgebungen zur Verfügung zu stellen. Jedoch haben Forscher einige Nachteile und Grenzen der Verwendung von TPM identifiziert. Zu diesen Problemen zählen Privatsphäre und Wartbarkeit Probleme, Probleme mit der Sealing-Funktionalität sowie der hohe Kommunikation und Management-Aufwand. Auf der anderen Seite, TPM- Virtualisierung, insbesondere Virtualisierung von PCRs (Plattform Configuration Register), stoßt gegen einen der wichtigsten Grundsätze der Trusted Computing, nämlich die Notwendigkeit für eine hardware-basierte sichere Aufbewahrung von Daten.

In dieser Arbeit präsentieren wir unterschiedliche Ansätze und Architekturen, die verwendet werden können, um die durch die oben genannten Probleme entstehenden Nachteile zu mildern. Im ersten Teil dieser Arbeit präsentieren wir einen Ansatz namens Behavior Compliance Control (BCC), die verschiedene Ansätze und Architekturen beinhaltet, die beschreiben, wie das Verhalten der ausgelagerten Berechnungen erfasst und gesteuert wird, sowie die Möglichkeit zur Beurteilung über die Übereinstimmung des registrierten Verhaltens mit einem vertrauenswürdigen Verhaltensmodell. Genauer, wir präsentieren Ansätze für zwei Abstraktionsebenen, einen auf eine Programm-Code-Ebene und einen anderen auf der Ebene der abstrakten ausführbaren Geschäftsprozesse.

Im zweiten Teil dieser Arbeit präsentieren wir unsere entwickelten Lösungen zu den oben genannten Problemen von TPMs und vTPMs. Wir haben festgestellt, dass die Verwendung von SHA-1 Hashfunktion zur Messung von Systemkomponenten zur Wartung von langen Listen von vertrauenswürdigen Software führt. Viel mehr wird die genaue Zusammensetzung der Konfiguration der Hosting-Platform bei der Ausführung vom Prozess der Remote Attestation bekanntgegeben. So zeigt unser Ansatz wie die Verwendung von Chamaleon Hashfunktionen es erlaubt, die Auswirkungen dieser beiden Probleme zu mildern. Auf der anderen Seite, um die Sicherheit der vTPMs zu erhöhen, zeigen wir in einem anderen Ansatz, wie die Stärke der Hardware-basierten Sicherheit für virtuelle PCRs durch Bindung ihrer Werte an die entsprechenden Hardware-PCRs zurückgewonnen werden kann. Wir entwickelten zwei Ansätze um eine solche Bindung zu realisieren; im ersten Ansatz verwenden wir binäre Hash-Bäume, während wir im zweiten Ansatz das inkrementelle Hashing nutzten.

Außerdem haben wir die vorgeschlagenen Ansätze prototypisch implementiert, um ihre Machbarkeit und Wirkung in der Praxis zu evaluieren. Darüber hinaus präsentieren wir eine empirische Bewertung der relativen Wirksamkeit der verschiedenen Verhaltens-Abstraktionen von BCC, die wir basierend auf reale Anwendungen erstellt haben. Insbesondere untersuchten wir die Machbarkeit, die Wirksamkeit, die Skalierbarkeit und Effizienz des BCC-Ansatzes. Zu diesem Zweck haben wir uns zwei Arten von Anwendungen ausgesucht, ein Web-basiertes und eine Desktop-Anwendung, auf die wir verschiedene Angriffe durchgefürt haben, wie z.B., Eingabe von bösartigen Inhalten und SQL-Injection-Angriffe. Die Ergebnisse zeigen, dass solche Angriffe mit Hilfe unseres Ansatzes erkannt werden können, so dass mehr Schutz gegen sie erreicht werden kann.

Acknowledgment

I am very thankful to my family, especially my parents, for their incredible efforts and for motivating me throughout the preparation of this thesis. They have been always supporting me and encouraging me with their best wishes. A special thank goes to my wife for her support and patience during the last years, also to my both daughters Nusaibah and Jumana.

Special thanks also go to my supervisor Stefan Katzenbeisser who gave me the opportunity to study and write this thesis under his guidance. I am very grateful to him for his support and assistance during the years in which I worked on this thesis. I also want to thank Eric Bodden for his valuable comments and for his co-supervision.

A well-deserved word of appreciation is also in order for Jan Sinschek, Andreas Sewe, Thomas Feller, Sunil Malipatlolla, Sebastian Biedermann and Özgür Dagdelen for the innumerable fruitful and enlightening discussions and for the research projects we conducted together.

I would like to make a special reference to the excellent working conditions I found at the Department of Computer Science at TU Darmstadt and the support I received from the Center for Advanced Security Research Darmstadt (CASED). I warmly thank all SECENG and CASED members for the motivating working atmosphere.

I would like to express my thanks to all those I did not mention and who supported me in any matter during the time in while I worked on my thesis.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

The past decade has seen an increasing interest in IT outsourcing as it promises many economic benefits, such as cost reduction, the possibility to focus on core capabilities as well as access to the provider's expertise and skills [69]. Another advantage, compared to in-house data processing, is that most providers charge on a pay-per-use basis which means that the customers do not have to pay for idle machines [110].

A concrete instantiation of this outsourcing scenario is Cloud Computing, in which concepts from virtualization technologies are applied. The core idea behind Cloud Computing is to provide enviroments for hosting both data and computation, i.e., clients machines must not be powerful to run complicated computations. In Cloud Computing, there are various delivery models, such as infrastructure as a service (IaaS), platform as a service (PaaS), and software as service (SaaS) [21], which provide different abstraction levels of hosting, i.e., the higher is the level the less the provider must care about. In IaaS, the hosting provider is responsible for providing only the infrastructure, which means that clients care about installing and configuring the platform and all upper levels. A step higher is the PaaS model, in which the provider provides — in addition to the infrastructure — the platform and leaves selecting and installing applications to the duties of the client. In SaaS, the hosting provider must provide all the previous services including software installation and in most cases its basis configurations.

At the same time, usage of service-oriented architectures (SOAs) has increased. Services in SOA deliver defined business functionalities and are clearly capsulated and loosely coupled entities [76]. Such services can be used in business processes which follow the SOA paradigm. In order to describe such abstract business processes and to specify the orchestration logic independent from the specific implementation of services, the Business Process Execution Language (BPEL) [63] has been established as an OASIS standard.

In recent years, not only outsourcing standard applications gains importance, but also outsourcing of business processes that are represented by workflows. Companies are able to use platforms provided by third parties as a remote runtime environment for their business processes, consisting of cross-linked services. For instance, in PaaS, a cloud provider hosts hardware, the operating system and the platform middleware (such as a business process execution engine, BP-Engine) and a database management system, and the customer delivers only the business process descriptions (e.g. BPEL processes) to the cloud provider.

However, outsourcing a computation to a hosting platform is not without risks: the client generally has no guarantee that the platform provider will execute the outsourced computation in the way intended [65]. An adversary (such as a malicious user or dedicated insider) in the could can compromise the outsourced application (e.g., document conversion server, a web server or business software) in a way that prevents it from fulfilling its security requirements. Furthermore, when outsourcing the more abstract business processes, customers are concerned about security problems that can arise after outsourcing. One of their major concerns is the adversary's possibility to manipulate the flow of the outsourced processes [79]. For example, consider a simple credit request process where either an extensive or a simple creditworthiness check is performed depending on the requested credit value. The former check is to be performed if the requested value is over 1000$, otherwise the latter check is performed. A customer who outsources this business process of credit risk assessment to a hosting platform must be assured that the correct activity (i.e., the correct flow) is executed.

Accordingly, we argue that hosting providers can provide clients such assurance by providing *trustworthy evidence about the runtime behavior* of the outsourced application. In this setting, it is important that the evidence can be efficiently checked, without re-doing most of the computations, as this would defeat the purpose of outsourcing. Also, such an approach needs to go without the need of manual step-by-step auditing, since this would require expensive manual labor and would be prone to human error. In response to this challenge, work has been done on trusted computing, wherein the integrity of a platform is assured via a process called integrity measurement. Load-time integrity measures [86], which check the program to be run before its execution and report its integrity back to the client, are insufficient in this context. For instance, in the scenario of outsourcing a document server, its code is unaltered and hence such measures would verify the integrity of the document server's code, providing a false impression of security. To effectively recognize attacks caused by malicious inputs or insufficient countermeasures, clients must be given a means to reliably validate that the outsourced application indeed executes as intended. However, the behavior of the application is not only influenced by its code, but also by input data driving computations and configuration files. As a consequence, identifying what applications run at a remote site is not sufficient to verify that outsourced code runs as desired.

In addition, to be able to verify the flow of an outsourced business process, trustworthy evidence about its execution is needed. That is, more transparency for the remote processes [110] is demanded such that manipulations of the flow of business processes can be detected, even if these changes take place in a provider's remote system. Delivering such evidence helps systems and process auditors to control the compliance of outsourced business processes with business or security objectives [64].

Another challenge in this area is how to securely provide a runtime-secure storage for such evidence. Trusted computing offers manners to securely store small amounts of data inside the so-called Platform Configuration Registers (PCRs) of the Trusted Computing Module (TPM). Unfortunately, the concept of one hardware TPM for every platform is not adequate in scenarios where multiple TPMs are needed on the same platform, such as virtualization scenarios. To solve this problem, the concept of virtual TPMs (vTPMs) [28] was proposed to allow the utilization of TPM functionalities, such that each virtualized system is associated to an isolated TPM instance. vTPMs are currently implemented in software. Current approaches for virtualizing TPMs [28, 90, 83] do not provide – to the best of our knowledge – hardware-based security for virtual PCRs (vPCRs), which is one of the main principles of trusted computing.

Another problem with trusted computing lies in the reporting services, called remote attestation. Research has identified several problems with the classical remote attestation process as specified by the Trusted Computing Group (TCG) [3]. These problems include privacy [33] and maintainability issues [86, 72], problems with the sealing functionality [82] and the accompanied high communication and management efforts [72].

To this end, we pursue in this thesis the following goals:

1. Developing an approach which helps to provide evidence of intended execution by capturing, controlling and verifying the compliance of the behavior of outsourced computations of different abstraction levels.

2. Extending existing runtime-secure storage architectures to be able to store all security-critical runtime information that are related the generated evidence, in a way which is more conform to the principles of cloud computing; the main focus will be the support for multi-tenancy and virtualization.

3. Providing proof-of-concept implementations for these architectures.

To fulfill these goals, we list in the following our contributions in this thesis:

Approaches to Behavior Compliance Control — Chapter 3

In Chapter 3, we present two different approaches to *behavior compliance control (BCC)* which allow hosting platforms to create trustworthy evidence about the behavior of outsourced computations, and —in turn— allow clients to decide on the compliance of this behavior with trusted and correct reference behavior. The first approach, called low-level BCC, consists of three phases as illustrated in Figure 1.1. In a learning phase, before outsourcing the application, the client learns the behavior of a program by running it locally (thus in a correct and trusted manner) using

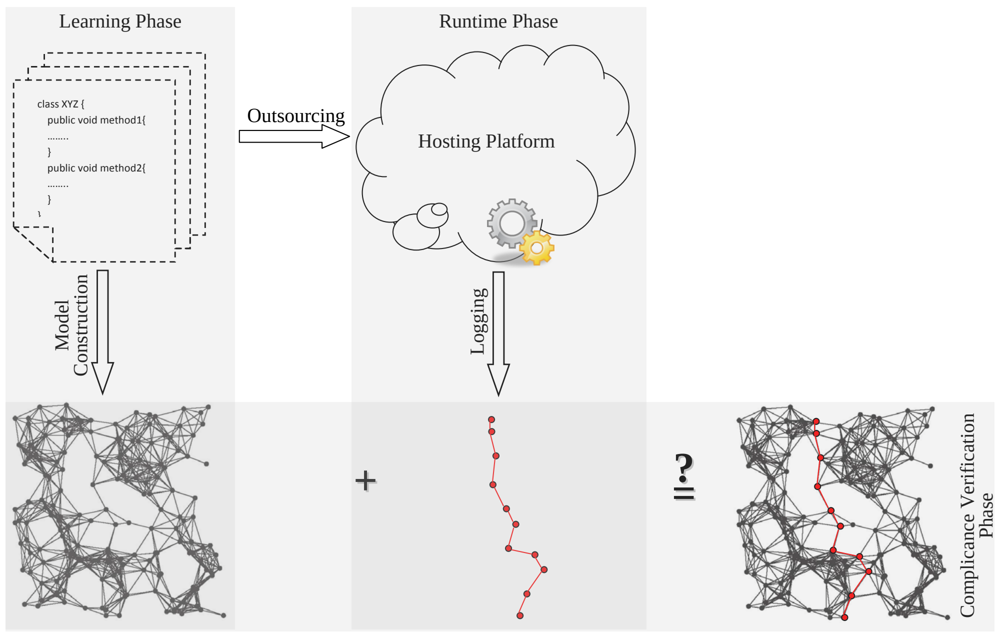


Figure 1.1: The three-phase approach to low-level behavior compliance control

a collection of representative inputs. This process results in execution profiles, from which one then computes a so-called application model. The model is considered to characterize the intended behavior of an application.

In the runtime phase, after the application has been outsourced to a hosting platform, there are two options to assure behavior compliance. First, the client can actively check model compliance at runtime, through an inserted inline reference monitor. As an alternative, the platform provider can use runtime monitoring techniques to log critical runtime information into a securely sealed storage, thus yielding trusted evidence on the application's actual behavior. Next, the client verifies if the observed program run, according to this evidence, complies with the application model learned in the training phase. If the run is found to be compliant, the outsourced computation is assumed to have executed correctly. Our approach builds on ideas from intrusion and anomaly detection [45, 49, 62, 93, 47, 44], but for the first time assesses to what extent such techniques can be used for behavior compliance control of outsourced computations, when combined with technologies for the trusted assessment of logs, e.g., secure storage.

To the best of our knowledge, previous works on anomaly detection all consider one particular way of abtracting from a program's behavior. But the nature of the behavior profiles used is critical to the success of our approach, and to anomaly-detection techniques in general. Hence, in this thesis, we take a more comprehensive approach and evaluated multiple possible representations of an application's behavior at different levels of abstraction. For the purpose of this thesis, we use function calls as an indicator of program behavior, as they are a natural abstraction of program behavior, and of wide applicability. To approximate the behavior of a program, we investigate three characterizations on different levels of granularity: function sets, call graphs and calling context trees. We define compliance of a program run as a subset relationship: a function call that is not covered by the application model marks deviating behavior.

We further present two architectural frameworks for compliance control, an active and a passive one. Given the application model, the active framework uses a reference monitor to verify the correctness of a program run just-in-time. The passive approach is similar to the integrity measurement approach of trusted computing, but aims at recording behavior rather than the integrity of the (binary) application code.

In the second part of this chapter, the second approach, called high-level behavior compliance control, is described. We consider the PaaS delivery model (sketched in Figure 1.2), where a customer provides a logging (attestation) policy in addition to the business process description, and expects from the hosting platform trustworthy evidence E about the correct execution of these processes. Accordingly, our main contribution is to provide an architecture which follows the "compliance by design" principle, allowing to remotely verify the correct execution of a business process.

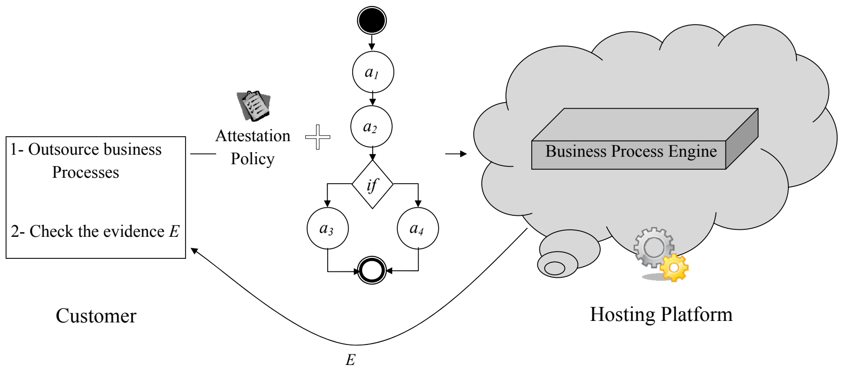


Figure 1.2: The PaaS delivery model in our scenario

Technically, we use remote attestation and the Trusted Platform Module (TPM), core technologies of Trusted Computing (TC). The combination of remote attestation with business processes has been poorly investigated so far. Unlike traditional attestation architectures, we provide a fine-granular and policy-based attestation architecture, where the attestation policy defines critical actions that need to be logged during the execution of the outsourced business process. After the process is completed, the client can request (during an attestation phase) a signed version of this log, which allows to verify correct process execution. The log can either be verified on the fly or stored for later use in case a dispute arises. Our architecture also works on multi-tenancy hosting platforms [56] and considers multi-instance processes.

Improving Existing Techniques to Build a Runtime-Secure Storage — Chapter 4

In this chapter, we deal with the aforementioned privacy, scalability and maintainability problems related to integrity measurement and remote attestation of trusted computing. More specific, remote attestation discloses full information about the software running on the attested platform, including details on the operating system and third-party software. This may be an unwanted privacy leak, as it allows for product discrimination (e.g., in a DRM context a party can force the use of a specific commercial software product before certain data is released, thereby limiting freedom of choice) or targeted attacks (e.g., if a party knows that someone runs a specifically vulnerable version of an operating system, dedicated attacks are possible). Thus, attestation methods are required that do not reveal the full configuration of the attested platform but nevertheless allow a challenger to gain confidence on its trustworthiness. The second major problem of TCG attestation is the scalability of Reference Measurement Lists [86]. The large number of software products and versions of operating systems makes maintenance of the lists cumbersome. For instance, [40] notes that a typical Windows installation loads about 200 drivers from a known set of more than 4 million, which is increasing continuously by more than 400 drivers a day. The large number of third-party applications aggravates the problem further. Scalability of the remote attestation process is sometimes seen as a major limiting factor for the success of trusted computing [72].

In this thesis, we propose novel attestation and integrity measurement techniques which use chameleon hashes or group signatures in the integrity measurement and attestation process. Even though this increases the computational complexity of the attestation process, we show that the presented mechanisms increase the scalability of remote attestation, while providing a fine-grained mechanism to protect privacy of the attested platform. One construction uses chameleon hashing [67], which allows grouping sets of software and hardware versions, representing them through one hash value. For instance, all products of a trusted software vendor or versions of the same software can be represented by one hash value. On the one hand, this reduces the management effort of maintaining Reference Measurement Lists (RMLs), and on the other hand increases privacy, as the challenger is not able to see any more the exact configuration of the attested platform, but only the installed software groups. At the same time, the challenger system can be assured that all running software comes from trusted software groups. We show that the proposed system can easily be integrated into an architecture similar to the TCG, with only minor modifications.

In the second part of this chapter, we propose an approach to bind vPCRs to hardware PCRs to gain strength of hardware-based security. More specifically, we provide two variants for this binding; the first uses binary hash trees [73] and the second uses the concept of incremental hashing [23, 51]. In the first variant all vPCRs of the same index – on a platform – are jointly hashed using binary hash trees. The root hash value is stored in the hardware PCR. In the second variant, we use the incremental hashing approach, so that an aggregated hash value can be stored in the hardware TPM chip.

Both approaches require the calculation of the hash tree or the incremental hash inside the TPM to guarantee the security of the hash result. Unfortunately, the current TPM specification does not provide interfaces for such operations. Thus, we propose some additions to TPM sepecifictions. While it is difficult to change deployed TPM chips, next-generation TPMs [104] will allow specifying required cryptographic functionalities; furthermore, the concept of reconfigurable TPM chips [39, 50, 43] allows the implementation of new functionalities with relative ease.

Proof-of-Concept Implementations — Chapter 5

To the approaches described in Chapter 3, we do provide two concrete instantiations, in the form of one implementation in the context of Java programs and one for BPEL business-process execution. The Java scenario is representative of white-box behavior profiles that are directly based on an application's internal execution behavior. The BPEL scenario acts on a higher level abstraction. It considers a client outsourcing business process definitions to be executed on a business process engine. Our implementation, based on BPEL and the ODE engine, can detect manipulations in the service orchestration specified in the business process definition.

Furthermore, we make a proposal for implementation of the approach detailed in Section 4.1 base on a Virtex5 FPGA platform and show that the application of both approaches can increase the security of virtual TPMs with reasonable overhead. In addition, we show that the proposed system described in Section 4.2 can easily be integrated into an architecture similar to the TCG, with only minor modifications. We have implemented the attestation process in a prototypical fashion and show that the approach is feasible in practice. Finally, we show that a very similar attestation technique can be implemented by group signatures instead of chameleon hashes as well.

An Analysis for Evaluating the Approaches — Chapter 6

Since the approach to the low-level BCC is considered as a main contribution of this thesis, we empirically evaluate the relative efficacy of the three different behavioral abstractions that we define based on different real world applications. We list some attacks, which can be performed by an adversary to harm an outsourced application, and describe how our approach can be applied to detect such attacks. We used our Java-based implementation to produce behavior profiles collected from these different outsourced open-source applications, fed with publicly available data as inputs. Our results show that using function sets and call graphs as behavior abstraction, it is possible to learn program behavior with low rates of false warnings. In addition, even in cases where false warnings do arise, we discuss how these can be used to the benefit of the approach by re-performing those computations in-house that our tool chain warned about (for example, in the private part of a hybrid cloud). This process allows the outsourcing company to identify false warnings as such, and hence allows the company to gradually refine the application profile, ultimately yielding an optimal profile with no manual intervention.

2 Preliminaries: Program Profiling, Cryptography and Trusted Computing

In this chapter, we give a background about the methods and techniques we use in our approaches we propose in Chapters 3 and 4. Since we propose in Section 3.1 an approach which uses methods from program analysis, we give in Section 2.1 a brief background about program analysis and program profiling and discuss their usage in research. In Section 2.2, we explore the area of business process management and automation. Section 2.3 gives a background about some cryptographic primitives we used in the development of our approaches we present in Section 4.2. Since various architectures we propose in this thesis rely on technologies from Trusted Computing and virtualization, we explain in Section 2.4 some principles and techniques on which both topics are built.

2.1 Program Analysis and Profiling

Program analysis aims at providing techniques which allow examining program code and reasoning over possible behaviors of it. One can distinguish between two main types of program analysis. While in static analysis a behavior model of the analyzed code is built without executing it, in dynamic analysis an execution of the code is necessary [41]. Moreover, static analysis usually uses an abstracted model (e.g., abstraction from program's input data) of program states which leads to loss of information about the program's behavior. On the contrary, dynamic analysis is precise because no abstraction needs to be done, i.e., during execution actual and exact runtime behavior is detected. Typically, dynamic analysis is used in areas like program optimization, program understanding and program testing, i.e., it is used mainly in software development lifecycle.

Program profiling is a form of dynamic program analysis used to analyze the dynamic program behavior using runtime information of the program. The most common use of profiling is to aid program optimization. Program or software optimization is the process of making the program/software run more efficiently or use fewer resources. Software profiling is also used for debugging and bug isolation, coverage testing, understanding program/architecture and examining memory or CPU usage.

There are two principal techniques for program profiling: instrumentation-based profiling and sampling-based profiling. While the former relies on augmenting a program with instrumentation code at points of interest, the latter uses sampling methods at defined time intervals to draw conclusions about the behavior of a program [77]. There is no perfect profiling techniques, i.e., each technique has its advantages and disadvantages.

Although instrumentation-based profiling is widly used, it still suffers from some drawbacks; the instrumentation method is intrusive and overhead-related, since instrumentation code must be inserted at numerous places of the program to be profiled. The instrumentation can be performed by a compiler, a profiling tool or – in case of executing byte code – by a virtual machine. In addition, by its nature, these techniques leads to increasing the size of code to be executed, which is in some cases unacceptable for memory-size critical systems. Another drawback is the fear of possible behavior changes of an instrumented program. Compared to the other profiling techniques, instrumentation at specific points can be considered at the same time as strength of the technique; selecting the instrumentation points leads to more profiling flexibility and accuracy. In summary, the most challenging issue is to control the tradeoff between accuracy and efficiency which is not trivial and application-related issue.

On the contrary, works like [111, 116] which use sampling techniques fail to give a complete and accurate overview about overall behavior of a system. This is because the fact that sampling uses timers which generate interrupts at defined time intervals, which could lead to missing information about the behavior of the program; predicting "good" time intervals is difficult for dynamically generated profiles. This time interval is then associated with a program construct such as a function body, a loop or a statement, depending on the program counter PC. The information loss is resulting from approximating the association of sampling intervals to a program construct [77]. Usually, hardware performance counters are used to obtain high frequency sampling with reasonable overhead. The higher is the frequency of the samples, the more profiling accuracy can be achieved. Another problem of this technique are execution interrupts caused by taking samples.

One of the data structure representations resulting by profiling a program is calling-context trees (CCT) [15], which associate a metric with a sequence of procedures called during execution. Keeping the context of calling avoids approximating a program's context dependent behavior.

Fore more accuracy and flexibility, CCT profiles for modern programing languages like Java are required to capture overall program execution on any standard, state-of-the-art Java Virtual Machines (JVM) and represent both inter-procedural and inter-procedural control flow. JP2 [88] is a profiler that produces accurate and complete calling-context trees on production JVMs. JP2 traces the entire call stack which to led a statement of method being executed. This thread-aware tool generate a single CCT for all threads executing in the JVM. As a result, the CCT can be serialized, e.g., as an XML structure. We use JP2 in the implementation of our approach proposed in Section 3.1. The implementation details about JP2 are explained in Section 5.1.

2.2 Business Processes

Business processes are complete and dynamically coordinated sets of collaborative and transactional activities that deliver one or more defined values and fulfill a specific goal. They gain more importance especially when they come in the context of Business Process Management (BPM). BPM includes methods and techniques for defining, designing, simulating, executing and monitoring both automated and non-automated business processes. The main focus of BPM lies in the phase of execution, which assumes the completion of the aforementioned pre-phases. Another important phase is the simulation, which helps managers and business analysts to simulate their processes and to get the information they need about processes. Simulation is usually used for, e.g., costs calculations, supporting the capacity or deployment planning, and prediction of performance data of changing environment situations. In Section 3.2 we see how simulation can be used in security-related issues as well.

One of the ways to automate processes is to use business process execution engines that execute the required steps of the process. Organizations and companies can use and combine new and existing services by the use of such engines. That is, business process engines are systems that manage and monitor processes while running in real time. It automates processes that are clearly defined in process templates or models. The execution engine then synchronizes the activities and interactions of the process model. It assigns activities to stakeholders according to the routing rules that are defined in the model. It escalates, delegates and manages the status of the workflow, and ensures that tasks are completely executed. In addition, it coordinates the interaction with other applications on the middleware and provides audit processes. An example for an open-source business process engine is the ODE Apache (Orchestration Director Engine) which executes business processes written following the WS-BPEL (Business Process Execution Language) standard. Besides Apache ODE, there are lots of commercial process engines which execute and manage BPEL processes such as the Oracle BPEL Process Manager[1], the BizTalk Server[2] developed by Microsoft, and the WebSphere Process Server[3] developed by IBM.

[1]http://www.oracle.com/technetwork/middleware/bpel
[2]http://www.microsoft.com/biztalk
[3]http://www-01.ibm.com/software/integration/wps/

WS-BPEL is an OASIS[4] standard executable language for specifying actions within business processes with web services. Although BPEL was introduced in 2002 by IBM, BEA Systems and Microsoft as a language used to describe the orchestration of web services, it is also wildly used in research, so that there are lots of scientific papers and articles which discuss strengths and weaknesses BPEL as well as its improvements. The description itself is also provided in the form of a web service and can be used as such. It is based upon the XML language and includes a number of ways in which business processes can be expressed. The goal of BPEL is to allow "abstract" programming. It should be noted that BPEL does not support direct human interaction. That is, BPEL processes only communicate with web services, i.e., the latter can act as an interface for human interaction. However, IBM has published a white paper in cooperation with SAP under the name BPEL4People [5] which is considered as an extension for BPEL providing the requirements for human interaction.

The description of processes is structured in blocks. In the block structuring, the control flow is expressed using basic activities such as `assign`, `invoke`, and `receive/reply`, or structured activities such as `sequence`, `if`, `foreach` or `pick`, which is similar to procedural programming languages. BPEL supports officially no graphical representation of its elements. However, there exist many extensions and partner-languages to support this. As an example for a BPEL process, we depict in Figure 2.1 a simple loan request process, which is graphically modeled using the BPEL Designer[5] plug-in for Eclipse.

In detail, the process begins by receiving the name of the customer applying for the loan. Afterwards, the activity "`prepareCustomerData`" of type `assign` is performed to prepare customer data for saving. After saving data, the engine begins to process the construct "`specifyIntensityCheck`" of type `switch`, which checks whether the requested amount is over a certain threshold. Depending on the requested value, a sequence of activities is performed; for high loan requests the web service "creditWorthinessCheckIntense" is called which performs intensive creditworthiness checks using the data of the requester, otherwise standard checks are performed. The process ends by sending the result of creditworthiness check to the caller. Note that this example process is used in sections where we describe our approach related to business processes (cf. Section 3.2).

[4]http://www.oasis-open.org/

[5]http://www.eclipse.org/bpel/

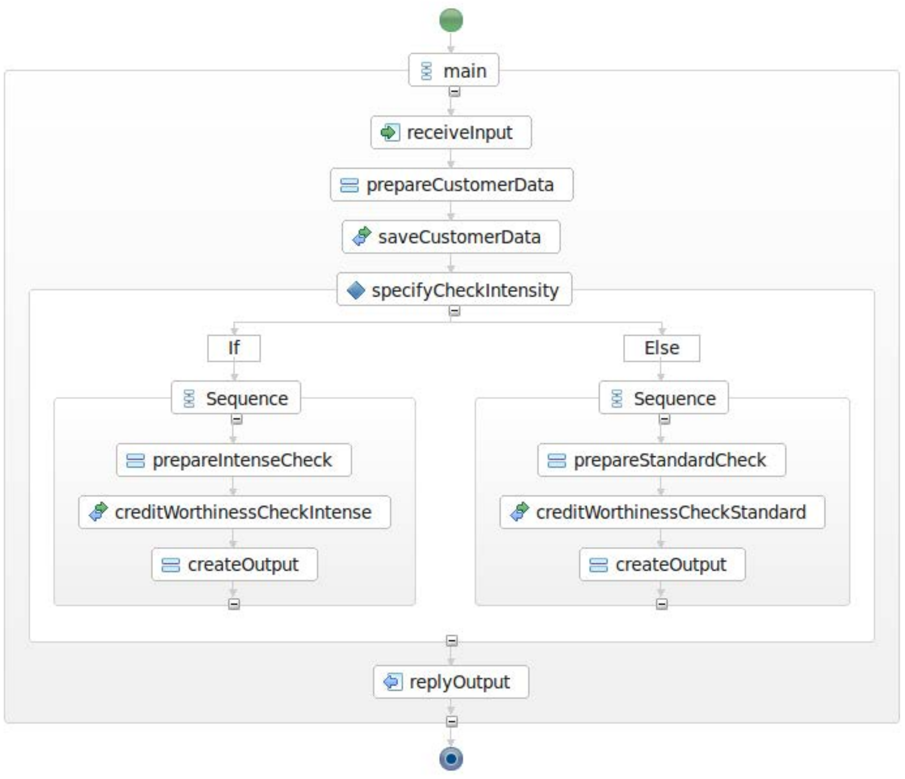


Figure 2.1: An example for a loan request process expressed in BPEL

2.3 Cryptographic Primitives

2.3.1 Chameleon Hashing

Unlike standard hash functions, chameleon hashes utilize a pair of public and private keys. Every party who knows the public key is able to compute the hash value on a given message. The possession of the private key enables collisions to be created. However, chameleon hash functions still provide collision-resistance against users who have no knowledge of the private key.

Maybe one of the most popular and important cryptographic primitives are hash functions. Hash functions are deterministic and low-cost functions. They take as input a bitstring of arbitrary length and output a value of fixed size. The special feature of hash functions are the following properties. It is easy to compute the output (hash value) for any given input string, but it is unfeasible[6] to find (a)

[6]Note that the adversary has actually a negligible probability to find such a input value due

an input string mapping to a given hash value (preimage-resistance) and (b) two distinct input strings mapping to the same hash value (collision-resistance). The feature of hash functions provide big advantages in many applications, e.g. digital signatures, hash tables, to identify files on peer-to-peer file sharing networks and a good deal more.

Chameleon hash functions are also hash functions, but unlike standard hash functions, they are assigned with a pair of public and private (trapdoor) information. Chameleon Hashing was introduced in [67] by Krawczyk and Rabin for the use in a signature scheme but basically they are non-interactive chameleon commitment schemes introduced by Brassard, Chaum and Crepeau [32].

A chameleon hash function is defined by a set of efficient (polynomial time) algorithms [20]:

Key Generation. The probabilistic key generation algorithm $\mathbf{Kg} : 1^\kappa \rightarrow (\mathbf{pk}, \mathbf{sk})$ takes as input a security parameter κ in unary form and outputs a pair of a public key **pk** and a private key (trapdoor) **sk**.

Hash. The deterministic hash algorithm $\mathbf{CH} : (\mathbf{pk}, m, r) \rightarrow h \in \{0,1\}^\tau$ takes as input a public key **pk**, a message m and an auxiliary random value r and outputs a hash h of length τ.

Forge. The deterministic forge algorithm $\mathbf{Forge} : (\mathbf{sk}, m, r) \rightarrow (m', r')$ takes as input the trapdoor **sk** corresponding to the public key **pk**, a message m and auxiliary parameter r. **Forge** computes a message m' and auxiliary parameter r' such that $(m, r) \neq (m', r')$ and $\mathbf{CH}(\mathbf{pk}, m, r) = h = \mathbf{CH}(\mathbf{pk}, m', r')$.

In contrast to standard hash functions, chameleon hashes are provided with the **Forge** algorithm. By this algorithm only the owner of the trapdoor (**sk**) can generate a different input message such that both inputs map to the same hash value. In some chameleon hashes the owner of the private information can even choose himself a new message m' and compute the auxiliary parameter r' to find a collision $\mathbf{CH}(\mathbf{pk}, m, r) = h = \mathbf{CH}(\mathbf{pk}, m', r')$. This is a powerful feature since anyone who knows the private information can map arbitrary messages to the same hash value.

We desire the following security properties to be fulfilled by a chameleon hash function (besides the standard property of collision resistance):

Semantic Security. For all pairs m, m', the values $\mathbf{CH}(\mathbf{pk}, m, r)$ and $\mathbf{CH}(\mathbf{pk}, m', r)$ are indistinguishable, i.e., $\mathbf{CH}(\mathbf{pk}, m, r)$ hides any information on m.

to the birthday paradox. It is assumed that in such cases we can ignore this probability.

Key Exposure Freeness. Key Exposure Freeness indicates that there exists no efficient algorithm able to retrieve the trapdoor from a given collision, even if it has access to a **Forge** oracle and is allowed polynomially many queries on inputs (m_i, r_i) of his choice.

Any chameleon hash function fulfilling the above definitions and security requirements can be used in our approach presented in Section 4.2; our particular choice of a chameleon hash is detailed in [20]. Next, we explain some details about our particular choice.

The scheme introduced in [20] takes in addition to a message m and an auxiliary parameter r a label $\mathcal{L}$ which is a arbitrary bitstring. We need a secure hash-and-encode scheme $\mathcal{C} : \{0,1\}^* \rightarrow \{0,\dots,2^{2\kappa-1}\}$ which we use to map the label $\mathcal{L}$ to an integer. In [20] it is recommended to use the EMSA-PSS encoding, defined in [25, 38]. The chameleon hash scheme $\mathcal{CHAM} = (\mathbf{Kg}, \mathbf{CH}, \mathbf{Forge})$ is constructed as follows:

Key Generation. Let τ and κ be security parameters. Let $\mathcal{H}$ be a collision resistant hash function, s.t. $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^\tau$. Choose randomly two positive κ-bit, distinct odd primes p, q. Let N be the RSA modulus $N = pq$, thus we get $\phi(N) = (p-1)(q-1)$. Furthermore, choose a positive integer e randomly, s.t. $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$. The RSA modulus N and the integer e form the public key. Next, determine the integer d which satisfies the congruence relation $de \equiv 1 \mod \phi(N)$. The private key consists of the values (p, q, d). Hence, the key generation algorithm **Kg** outputs $\mathbf{pk} = (N, e)$ and $\mathbf{sk} = (p, q, d)$.

Hashing. Given a message m, label $\mathcal{L}$ and randomness r, the hashing algorithm **CH** computes

$$\mathbf{CH}(\mathcal{L}, m, r) = J^{\mathcal{H}(m)} r^e \mod N,$$

where $J = \mathcal{C}(\mathcal{L})$.

Forge. Choose a message m' randomly (or alternatively take m' as input) and compute r' as

$$r' = r(J^d)^{\mathcal{H}(m) - \mathcal{H}(m')} \mod N.$$

The Forge algorithm **Forge** outputs (m', r').

2.3.2 Group Signatures

Group signatures were introduced by Chaum and van Heyst [35] and allow a member of a group to anonymously sign a message on behalf of the group. A group has a single group manager and can have several group members. Unlike standard digital signatures, signers of a group are issued individual signing keys $\mathbf{gsk}[i]$, while all members share a common group public key $\mathbf{gpk}$ such that their signatures can be verified without revealing which member of the group created the signature.

This provides anonymity. However, the group manager is assigned with a group manager secret key **gmsk** and is able to discover the signer (traceability).

Basically, a group signature scheme $\mathcal{GS} = (\mathbf{GKg}, \mathbf{GSig}, \mathbf{GVf}, \mathbf{Open})$ is defined by a set of efficient algorithms (for more details, we refer to [35] and [24]):

Group Key Generation. The probabilistic group key generation algorithm $\mathbf{GKg} : (1^\kappa, 1^n) \to (\mathbf{gpk}, \mathbf{gmsk}, \mathbf{gsk})$ takes as input the security parameter κ and the group size parameter n in unary form and outputs a tuple $(\mathbf{gpk}, \mathbf{gmsk}, \mathbf{gsk})$, where **gpk** is the group public key, **gmsk** is the group manager's secret key, and **gsk** is a vector of n secret signing keys. The group member $i \in \{1, \ldots, n\}$ is assigned the secret signing key $\mathbf{gsk}[i]$.

Group Signing. The probabilistic signing algorithm $\mathbf{GSig} : (\mathbf{gsk}[i], m) \to \sigma_i(m)$ takes as input a secret signing key $\mathbf{gsk}[i]$ and a message m and outputs a signature $\sigma_i(m)$ of m under $\mathbf{gsk}[i]$.

Group Signature Verification. The deterministic group signature verification algorithm $\mathbf{GVf} : (\mathbf{gpk}, m, \sigma) \to \{0, 1\}$ takes as input the group public key **gpk**, a message m and a signature σ and outputs 1 if and only if the signature σ is valid and was created by one of the group members. Otherwise, the algorithm returns 0.

Opening. The deterministic opening algorithm $\mathbf{Open} : (\mathbf{gmsk}, m, \sigma) \to \{i, \perp\}$, which takes as input a group manager secret key **gmsk**, a message m and a signature σ of m. It outputs an identity $i \in \{1, \ldots, n\}$ or the symbol $\perp$ for failure.

Join. A two-party protocol **Join** between the group manager and a user let the user become a new group member. The user's output is a membership certificate $cert_i$ and a membership secret $\mathbf{gsk}[i]$. After an successful execution of **Join** the signing secret $\mathbf{gsk}[i]$ is added to the vector of secret keys **gsk**.

The following is a list of security requirements which a group signature scheme should meet:

Correctness. A signature σ generated by a group member using the **GSig** algorithm must be recognized as valid (output is 1) by the **GVf** algorithm, i.e. $\forall m\ \forall i \in \{1, \ldots, n\}$
$\mathbf{GVf}(\mathbf{gpk}, m, \mathbf{GSig}(\mathbf{gsk}[i], m)) = 1$.

Unforgeability. No person other than group members is able to produce a signature to any message on behalf of the group.

Anonymity (or Untraceability). It must be computationally infeasible to trace the real identity of a signer from a valid group signature. Only a group manager using its group manager secret key has this ability.

Unlinkability. It must be computationally infeasible to determine whether two group signatures come from a same group member.

Exculpability (or No-framing). A coalition of group members or the group manager is able to produce a valid signature on behalf of another group member.

Traceability. A group manager must be able to determine the signer's identity of a given signature using the **Open** algorithm.

Coalition Resistance. Any given subset of group members, sharing their respective secrets, must be prevented to compute a valid group signature which is not openable by the group manager.

In order to allow revocation of users, we require an additional property:

Revocability. A signature produced using **GSig** by a revoked member must be rejected using **GVf**. Still, a signature produced by a valid group member must be accepted by the verification algorithm.

The group signature scheme $\mathcal{GS}$ above requires a fixed number of group members and we expect that the size and membership do not change after time. However, in our scenario it is desirable to add or remove a member of the group. Therefore, further properties are needed which imply adapted and redefined security properties as well as algorithms for administration of group members. Their security requirements are elaborately discussed in [26]. These groups offering the properties above are called dynamic groups.

2.4 Trusted Computing and Virtualization

2.4.1 Main Functionalities of Trusted Computing

Over the last decades, there were many efforts being made to increase security and to assure integrity of IT systems. To this end, since 1999 an IT consortium, known as Trusted Computing Group (TCG)[7], has been trying to define standards that provide more security and trustworthiness of IT systems.

A Trusted Computing System (TCS) is according to TCG composed of a Trusted Computing Platform (TCP) and a Trusted Operating System (TOS). TCP defines all extensions, methods and standards which are related to hardware or firmware

[7]http://www.trustedcomputinggroup.org/

of a system. These extensions starts by extending existing functionalities, e.g., CPU up to developing new devices and chips such as the Trusted Platform Module (TPM). On the contrary, TOS can be considered more complex, since it includes defining numerous security functionalities, concepts as well as trusted services and applications, which are responsible for, e.g., measuring and reporting the integrity of a system. It is worth mentioning that in addition to these two terms, the term Trusted Computing Base (TCB) plays a major role in a TCS, since it defines those components whose security and integrity influence the security of the whole system.

An architectural specification of TCP is defined by TCG in [101]. In this specification, there are two main components described; the tamper-resistant hardware module called TPM and the Root of Trust for Measurement (RTM).

Because hardware components provide generally more security than equivalent software implementations, one of the TCG goals was to realize the root of trust inside the hardware, namely using the TPM component which is soldered on the mainboard of a PC. RTM is realized through the measurements which are taken by the Core Root of Trust for Measurement (CRTM). The latter is normally implemented as an extension to BIOS, and therefore it is a part of the firmware of the system. Both the TPM and CRTM build the Trusted Computing Platform (TCP), and can be considered as root of trust for the overlying layers. That is, TCP consists of only hardware and firmware independently from the operating system and all other software running on the system. Root of trust is the basis of a chain of trust and therefore it can be seen as the start point of the functions of TCP. It is worth mentioning that the security and integrity of the root of trust can not be proven by the system itself, and it requires external evidence from a trusted authority.

Trusted Platform Module

The TPM is a microcontroller chip which can be placed on the motherboard of a computer system. The description of TPM and its functionalities can be found in many TCG specifications such as [102]. The specifications are addressed to both TPMs manufacturers such as Amtel and Infineon, and software developer who intend to use TPM functionalities.

Figure 2.2 illustrates the internal structure and the main units of a TPM. There are two types of units; functional and storage units. The functional units provide cryptographic functionalities like generating random numbers, calculating hashes and signing/verifying. The storage units can be divided into volatile and non-volatile storage. In addition, the TPM provides numerous commands (cf. the TCG commands specification [103]) to be used by devices and applications that need to communicate with the TPM and make use of its functionalities.

The SHA-1 engine generates a 160-bit hash value of a given input, which is generally used to update the Platform Configuration Registers (PCRs), especially

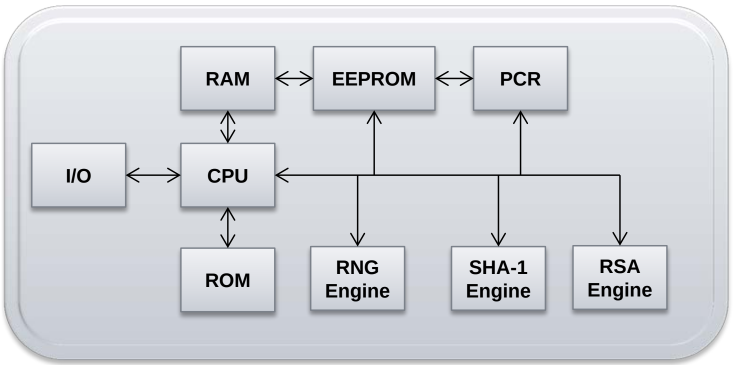


Figure 2.2: The internal structure of the TPM

at boot-time of the system. For the common use of SHA-1, it is not recommended to use this unit since a software implementation of this function is expected to be much faster than the implementation within a TPM.

The RSA-engine implements the well-known RSA algorithm. TCG recommends exclusively the use of 2048-bit keys. The engine is used for encryption and decryption as well as signing and verifying. An important usage area of the engine is to use it in building a secure hierarchy of cryptographic keys and encrypting small amounts of data to be stored outside the TPM. It is also used to generate RSA key pairs with the help of RNG-engine, which provides this unit with random numbers.

Another important unit of the TPM is the Endorsement Key (EK), which is a 2048-bit RSA key pair. It is important to note that EK is generated and placed into the TPM during production of the TPM and is kept in the non-volatile storage of the TPM. The EK has a certificate which is part of the platform credentials of a TCS, which guarantee for a challenger the presence of a TPM that is conform to TCG specifications. The private part of EK is known only to the TPM itself, and therefore it is assumed to be secure and can not be compromised or even read from any device or application outside the TPM. The EK and its certificate can be considered as machine identifier and therefore it is not used to sign messages that will be sent to challengers. To protect the identity privacy, the data can be signed using the Attestation Identity Key (AIK). The generation of AIKs is independent of EK, and thus, nothing is learned about the EK from data signed with AIKs.

The non-volatile storage is used mainly to store security-critical data which needs to be permanently stored in the TPM, such as the EK, Storage Root Key (SRK) and the Owner Authorization Secret. On the contrary, the volatile storage is used for, e.g., the Platform Configuration Registers (PCRs), the RSA key slots and authentication session handles. The PCRs are small data storages which are used to keep 160-bit SHA-1 hashes. The hashes are initialized and updated in the integrity measurement process defined by TCG (we will explain this later in more details). The TCG specification requires for TPM manufacturers the presence of at least 24 such registers. The lower indexes of these registers are reserved for keeping the measurements of the pre-boot process (indexes 0-7), while the higher indexes are reserved for measurements taken during the boot the process (indexes 8-15). Registers 17 to 23 can be used by the dynmaic operating system.

Trusted Software Stack

In addition to TCP, TCG defines specifications for supporting software called Trusted Software Stack (TSS). TSS is composed of different layers which provide standardized interfaces for applications to use the services of the TPM and to communicate with the rest of the platform and other remote platforms. Layering details of TSS and its functionalities can be found in the TSS specification [100]. In addition, the TPM Device Driver (TDD) are provided by TPM manufacturers to allow direct control over the TPM using the TPM Device Driver Library (TDDL) and its interface the TDDLI, i.e., the TSS uses the TDDL and TDDLI to communicate with the TPM. On the other hand, TSS provide some core services which define an interface for the overlying service provider (TCG Service Provider – TSP). These services include, for example, a context manager, key and credential manager and an event manager. For instance, an email software can make use of interfaces of TSP to get access to the functionalities of the TPM, e.g., encryption.

Integrity Measurement

One of the main goals of Trusted Computing is to assure the integrity of a platform. This is done by measuring every entity (such as BIOS, OS kernel and libraries, and application software) using the SHA-1 hash before its execution. The measurements are taken performing either the SHA-1 engine provided by TPM, or a software implementation of the SHA-1 algorithm. All measurements are securely stored by extending values in a particular PCR register by a hash chain. To allow the challenger to recompute the hash values, information about the measured entities is stored in form of a Stored Measurement Log (SML).

The following demonstrates how an entity A measures an entity B [37]:

- A measures the entity B. The result is B's hash value.
- Corresponding information about the entity B (such as entity name) and the hash value itself will be stored in the unsecured Stored Measurement Log (SML).
- The current PCR value is extended in the form of: ExtendedPCRValue := SHA-1(previous PCR value || hash value of entity B).

- A passes control to B.

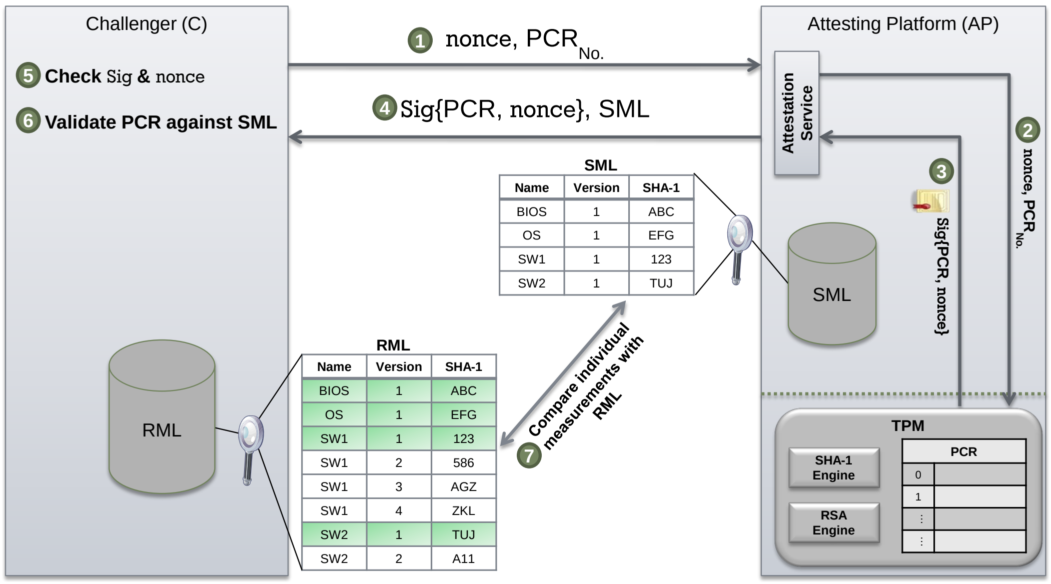
To prevent malicious software behavior, the TPM chip only allows to extend the PCR registers, so that PCRs can not be reset as long as the system is running. The only way to reset PCRs is to reboot the platform, i.e., when starting the system, all PCRs are empty at this time. As mentioned before, the basis of a chain of trust is a trusted component, i.e., in this case it is the CRTM. The CRTM acts as trust anchor for the measurements coming, and computes a hash value of itself and the static part of BIOS. After extending the hash value in the particular PCR as explained above, the CRTM passes control to the BIOS to measure hardware, option ROMs, and the operating system (OS) loader, then passes control to the OS loader. The hash values chain built is called Static Chain of Trust, i.e., this chain contains the hash values of the TCP. Building on the Static Chain of Trust, the Dynamic Chain of Trust contains all hash values taken by a Trusted-OS, such as Integrity Measurement Architecture (IMA) [86], or any overlying layer in the system such as middleware. IMA extends the ”measure-then-execute” principle followed throughout the boot process into the run-time of a system and therefore builds the Dynamic Chain of Trust.

The practical attestation framework IMA, which is an extension of the Linux kernel, was developed by IBM research [86]. IMA measures user-level executables, dynamically loaded libraries, kernel modules and shell scripts. The individual measurements are collected in a *Stored Measurement List* (SML) that represents the integrity history of the platform. Measurements are initiated by so-called *Measurement Agents*, which induce a measurement of a file, store the measurement in an ordered list into SML, and report the extension of SML to the TPM. Any measurement taken is also aggregated into the TPM PCR number 10. Thus, any measured software can not repudiate its existence.

Remote Attestation

Measuring the integrity of a platform builds only the basis to assess it. To this end, TCG defines a process called “Remote Attestation”, in which signed measurements can be released to third parties in order to judge the trustworthiness of the state of the attested platform.

As shown in Figure 2.3, the challenger creates a 160-bit *nonce* and sends it to the attested platform. The attestation service running on that host forwards the received nonce and the PCR number requested by the challenger to the TPM chip, which signs the data using the *TPM_Quote* function. After signing, the results are sent back to the attestation service. To protect identity privacy, only the *Attestation Identity Keys* (AIKs) can be used for the signing operation. The attestation service sends the signed data together with the SML back to the challenger. Using the



SML

Name	Version	SHA-1
BIOS	1	ABC
OS	1	EFG
SW1	1	123
SW2	1	TUJ

RML

Name	Version	SHA-1
BIOS	1	ABC
OS	1	EFG
SW1	1	123
SW1	2	586
SW1	3	AGZ
SW1	4	ZKL
SW2	1	TUJ
SW2	2	A11

Figure 2.3: The remote attestation process

corresponding public key AIK_{pub}, the challenger verifies the signature and the nonce, and re-computes the hash chain using the SML. If the re-computed hash value equals the signed PCR value, then SML is considered untampered. Finally, the challenger determines whether all measurements in SML can be found in the trusted Reference Measurement List (RML); in this case the attested platform is considered as trusted.

Sealing

The idea of sealing is to bind data of the local platform, or data sent to other platforms, on a a particular state of the recipient's system. A sealed message can be decrypted by the recipient only if he is, from the perspective of the transmitter, in a trusted state. If the recipient is in the desired state, then he will be able to decrypt the message sent to him. That is, the configuration state is used in the encryption and decryption process.

2.4.2 Cloud Computing and Virtualization

Cloud Computing depends on virtualization technologies. Several Virtual Machines (VMs)—each running a separate operating system (termed guest)—run on one hardware platform, controlled by a hypervisor. Hypervisors provide an environment to manage the VMs themselves and the native resources needed by each guest.

Basically, one can distinguish between two types of hypervisors: a type 1 hypervisor runs directly on the hardware, while a type 2 hypervisor runs on another operating system. Type 1 can, in turn, be classified into full virtualization (also called HVM) and paravirtualization (PVM). In full virtualization the hypervisor provides the guest operating systems with virtual hardware. This requires the usage of special hardware such as the Intel Virtualization Technology (VT-x) and AMD's AMD-V. On the contrary, PVM does not require special hardware, but substantial operating system modifications.

One of the widely used hypervisors that supports type 1 is Xen [22]. When booting a machine with Xen support, the first guest operating system, called in Xen terminology "domain 0" (dom0), boots automatically when the hypervisor boots and obtains special management privileges and direct access to all physical hardware by default. Any further guest operating systems are called "domain U" (domU).

In order to enjoy the benefits of TPMs in VMs, *Virtual TPMs (vTPMs)* were proposed to allow different (separated) TPM instances in each VM. For example, [28] proposes a system that enables the use of the functionalities of Trusted Computing (such as secure storage, cryptographic functions, etc.) for an unlimited number of VMs on a single hardware platform. All (software) vTPM instances are executed within a special VM, which provides an interface to manage and create these instances. To establish trust, the authors propose three different strategies of how the *Virtual Endorsement Key (vEK)* can be issued. One of the strategies is that the vEK is signed by an AIK of a physical TPM, so that the vTPM can request certificates for its vAIKs at a privacy CA. Another strategy is to sign the vAIK with an AIK of a physical TPM, and the third strategy is to use a local CA to issue a certificate for the vEK of the vTPM. In addition, a challenger can distinguish between a real TPM and a vTPM by a statement in the certificate. One important requirement for such vTPMs is the secure association of a vTPM instance to a VM instance. More information about the security requirements of vTPMs can be found in [28, 81, 80]. Another virtualization concept of TPMs was proposed by Strasser and Stamer [94]. The authors detailed in their approach a software-based TPM emulator to be used for research and developing purposes.

3 Behavior Compliance Control

Application security is one of the important issues discussed in both literature and practice over the last decades. A special topic in this area is security of outsourced computation. More specific, as motivated previously, one of the discussed problems in this thesis is verifying the behavior trustworthiness of outsourced applications to remote systems. In this chapter, we present in detail our proposed approaches and architectures which describe how to capture, control and judge the behavior compliance of such outsourced applications running on remote hosting platforms. Hereby we present approaches for two different abstraction levels; one is on the level of program code, and the other is on the level of abstract executable business processes which contain actions that fulfill a certain business goal. We intentionally present approaches on two different levels of abstraction to show the impact of such an abstraction on controlling the level of security and trust, effectiveness and performance, as well as the feasibility of each proposed approach.

In Section 3.1, we present an approach which is based on program profiling. We capture first the behavior of a target application using different profiling techniques and then build an application model, which is considered as a behavior benchmark for this application. Every program run is then compared to this model to verify the compliance of this run to the model. We call this approach *"Low-Level Behavior Compliance Control"*.

In a similar way, we present in Section 3.2 another approach for more abstract business processes, which we call *"High-Level Behavior Compliance Control"*.

Note that Sections 3.1 and 3.2 are entirely based on the publications [14, 12] respectively.

3.1 Low-Level Behavior Compliance Control Using Program Profiling

Assume that a client wishes to outsource some computation to a remote computing facility operated by a hosting platform. To do so, the client sends the application code to the service provider[1], e.g., using the platform-as-a-service model of Cloud Computing, or, in a software-as-a-service scenario, it licenses the required service.

[1]In this section, we use the terms "service provider" and "hosting platform" interchangeably.

At the remote facility, the code is executed on machines that are not under the virtualizing TPMs, especially virutalizing the Platform Con

guration Registers (PCRs), strikes against one of the core principles of Trusted Computing, namely the need for a hardware-based secure storge of the client. The client therefore wishes to get guarantees ensuring that the provider does indeed execute the code as-is, without any manipulation performed by the service provider or by any other adversary.

As described before in Figure 1.1, to give this guarantee we propose here a three-phase approach as follows:

Phase I: Learning Phase – Before outsourcing
: This phase captures the behavior of a representative amount of the target application's executions (profiles) using different techniques of program profiling (this phase must be done in a secure environment, e.g., in-house), and builds an application model out of all captured profiles. The model is considered trusted.

Phase II: Runtime Phase – At runtime
: The runtime program execution (trace) of the target application is logged in a secure manner.

Phase III: Compliance Checking – After execution
: Finally, the logged trace is compared to the well-known and trusted application model built in the learning phase in order to check its compliance.

Before outsourcing the target application, the client (or a trusted third party) determines (e.g., in-house) application profiles that describes a super set of program executions that are deemed to be representative for whole application behavior, and acceptable in terms of legal behavior. Each profile represents one single program run. The nature of those profiles is key to the success of the approach. We will discuss below different ways how such profiles can be structured (see Section 3.1.1). Out of all these profiles, an application model is created. This model represents all — from the point of view of this approach — allowed and acceptable program runs. Given such a model, the client can then characterize whether program runs executed remotely by the service provider comply with a model or not. If a program run does not comply with the model, this non-compliance signals to the client that the computation might not have been trustworthy, for instance because the software was executed with a sabotaged configuration setup, or because it was fed invalid inputs with the goal of exploiting a vulnerability in the application (potentially unknown to the client). Otherwise, the non-compliance is considered false positive.

In Chapter 4, we describe how to secure collected runtime information about runs executed on the side of the hosting platform, i.e., in the runtime phase of behavior compliance control (cf. Figure 1.1). We also describe how such a platform can be built and of which components it is composed. In this section, we first explain how to collect a behavior-characterizing model, i.e., in the learning phase, and in which ways such a model can be represented.

We assume that the application's model is constructed in a trusted manner by running an instrumented version of the target application using a set of test cases which fulfill an acceptable degree of behavior representativeness. This generates execution profiles, which we use to construct the model. Figure 1.1 presents the result of this learning phase as a graph (lower left). We then collect the runtime information in the same way as generating the application profiles. In a similar way, we define any given program run (logged at runtime phase) as compliant with this model if this run is covered by the model. In other words, the application model comprises *all* allowed behavior; every execution outside this model is considered a breach of compliance. That way, the model defines a safety property for the application, denoting the behavior that is anticipated and therefore considered safe. Note that this notion of model does *not* cover liveness properties, i.e., a model defined that way cannot demand that a program run *must* expose a certain behavior; an empty run will always be compliant to any model.

We call this approach *low-level* since it is applied on program's code level, which is considered low-level, compared to the other Behavior Compliance Control approach for more abstract business processes presented in Section 3.2.

3.1.1 How to Characterize Behavior

With the semantics of models defined by excluding behavior out of training sets, one of the major research questions in the area of behavior compliance control is how to actually classify behavior. Behavior can be classified in many ways, ranging from very coarse-grained characterizations to very fine-grained ones. The challenge in behavior compliance control is to find the right granularity in this spectrum. If the profiles are used to construct a model that is too coarse-grained then the model will be too permissive, which may cause malicious behavior to go undetected by our approach (false negatives). Conversely, if the chosen behavior model is too fine-grained, then it may suffer from overfitting, which may cause spurious false warnings (so-called false positives). Such false warnings are a burden on the client, since they require further inspection, which is, albeit fully automated, still time-consuming. In this section, we propose different characterizations of behavior at different levels of granularity, and to assess the usefulness of those abstractions in the context of behavior compliance control.

Previous works by others described behavior on different level of granularity. Sekar et al. [93] classify behavior using a quite coarse-grain black-box approach, in which finite-state automata resemble temporal orderings of observed system calls. Gao et al. [47] use a gray-box approach that also monitors system calls, but uses additional information from the runtime execution stack to build a so-called execution graph. Feng et al.'s approach [44], on the other hand, is a pure white-box approach, i.e., uses information internal to the executing program: the author's base their approach entirely on stack-trace information. All of the above approaches have their relative strengths and weaknesses. Hence, for this work, we decided to not restrict ourselves to a single mind set: instead of arbitrarily fixing one given classification of behavior, we decided to implement three white-box models on different levels of abstraction, and to compare their relative usefulness for the behavioral compliance control of outsourced applications. We chose white-box approaches because in general white-box approaches yield at least as much information as black-box approaches (which only monitor the application's input-output behavior). Many programs written for outsourcing scenarios (e.g., for the cloud) are written in managed-code languages such as Java, which are easy to instrument, and are therefore particularly amenable to such white-box approaches.

We regard function calls as a main ingredient for characterizing behavior. We have consequently evaluated three possible approximations of behavior by tracing which functions a program calls during its execution and in which contexts. Each approximation thereby induces a different kind of profile, which can then be used for our behavior compliance control approach. We distinguish profiles according to the amount of information that they contain (from least to most information):

- *Functions:* A set of functions F the program called during the execution.
- *Call graph:* A call graph, with nodes representing functions, and an edge from f to f' if f calls f' at least once during the execution.
- *Calling context tree:* A calling context tree (CCT) [15], with the root node representing the program's entry point and a node f' as child of node f if f calls f' in the same context at least once during its execution.

To illustrate these abstractions, consider the example program in Figure 3.1. In Figure 3.2 we show the corresponding instantiations of each of the different abstractions mentioned above: functions, call graph, and calling context tree. Figure 3.2a shows the "Functions" representation. Herein, the profile just consists of the set of all functions called during the program's execution. Figure 3.2b, on the other hand, shows the program's dynamic call graph. Note that in a call graph, every function, such as `bar`, is represented by exactly one node, no matter in how many different contexts the function is invoked. Figure 3.2c shows the program's calling context tree. In this representation, calling contexts are kept separate: because `bar` is called in two different places, once by `main` and once by `foo` (which is in turn called by `main`), it appears twice in the tree, just under the appropriate contexts.

```
public static void main(String args[]) {
    foo();
    for(int i = 0; i < args.length; i++)
        bar();
}

static void foo() {
    bar();
}

static void bar() { }
```

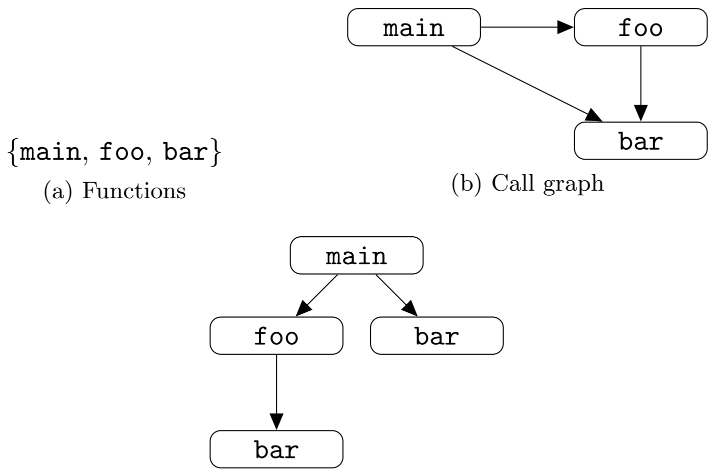
Figure 3.1: Example program

We chose these three different characterizations of behavior in a way that they would produce a total order with an increasing level of detail. It is easy to see that calling context trees contain more information than call graphs, and call graphs contain more information than the set of executed functions. Conversely, we can construct a call graph from a calling context tree simply by merging nodes labeled with the same function. Similarly, we can construct the set of all executed functions by a traversal of the call graph.

The fact that the three different abstractions form such a total order allows us to evaluate different characterizations of behavior at opposite ends of the granularity spectrum: The "Functions" representation is quite coarse-grained but can be computed very efficiently. Yet, by its nature it may have the tendency to yield false negatives, i.e., to miss attacks on the application. The calling context trees at the other end of the spectrum are very fine-grained. Their computation consumes more time, which may still pay off, though, because profiles based on calling context trees may identify more attacks. On the other hand, such profiles may suffer from false positives, i.e., false warnings. In Chapter 6, we present an extensive evaluation demonstrating how well these three different abstractions are suited to the task of behavior compliance control. Next, we provide a formalization of our three different abstractions.

Definition 1 (Function set)**.** *Let r be a monitored program run. Then the function set of r, denoted by* **functionSet**(r)*, is the smallest set fulfilling the following property: For any invocation $f \rightarrow f'$ of function f' from function f on r, it holds that $\{f, f'\} \subseteq$* **functionSet**(r).

Definition 2 (Call graph)**.** *A call graph is a directed graph (V, E) with V a set of nodes representing functions, and $E \subseteq V \times V$ a set of directed edges. Then the call graph of r,* **cg**(r)*, is a call graph that fulfills the following constraints. V is the smallest set such that for any invocation $f \rightarrow f'$ of function f' from function f on*



(c) Calling context tree

Figure 3.2: Three abstractions of the example program: Functions, call graph, and calling context tree

r, it holds that $\{f, f'\} \subseteq V$. E is the smallest subset of $V \times V$ such that for each such f, f' it holds that $(f, f') \in E$.

Definition 3 (Calling context tree)**.** *Let F be the set of all function identifiers. Then C_M, the set of all calling contexts over F, is defined as $C_M := F^+$. The set C_M is closed under concatenation: we define a concatenation function "$\cdot$" on calling contexts such that for any context $c \in C_M$ and function $f \in F$ it holds that $c \cdot f \in C_M$. A calling context tree is a tree (V, E) with $V \subseteq C_M$ a set of nodes representing calling contexts and $E \subseteq V \times V$ a parent-child relationship. We further demand that there exists a unique root node v_0 which has no parents, i.e., for which it holds that $\neg\exists v \in V$ s.th. $(v, v_0) \in E$. Let r be a monitored program run. Then* $\mathbf{cct}(r)$ *is a calling context tree for which the following holds. V is the smallest set such that for any invocation $c \to f$ of function f from within context c on r, it holds that $\{c, c \cdot f\} \subseteq V$. E is the smallest subset of $V \times V$ such that for each such f, c it holds that $(c, c \cdot f) \in E$.*

Note that all of the above definitions are oblivious to multiple threads. In particular, different threads do not induce different contexts in the calling context tree. Variants of the above abstractions that do distinguish between threads are straight-forward to define, but have not proven necessary to conduct our evaluation (cf. Chapter 6).

3.1.2 Profile and Model Generation

While, in principle, one could create all variants of application models described above by hand, this process would already be prohibitively cumbersome in the case of function sets and downright unrealistic for calling context trees. Therefore, a way to automatically generate these behaviour models is needed.

There are two basic options to generate function-call models automatically: statically or dynamically. In a static approach, one would use a static-analysis tool (such as Soot [106] or WALA [109]) to analyse the program's code without actually executing the program. We considered such an approach in the beginning but then quickly decided that it would be unsuitable for the purpose of behavior compliance control: Static code analysis typically abstracts from all input data, which means that the models found through static analysis tools would allow all behaviors the application could possibly exhibit on any (valid or invalid) input. For the purpose of behavior compliance control, such models would clearly be too coarse-grained; behavior compliance control is particularly useful in scenarios where attacks occur through invalid program inputs such as compromised configuration files.[2] It is therefore important that the models are, at least to some extent, sensitive to those inputs.

For behavior compliance control, we therefore opt for a dynamic approach that collects application profiles at runtime. The application is first instrumented with code that serializes a profile onto disk. Then the client test-runs the instrumented application using existing test cases. The client collects an execution profile for every test run. The application's final model for this training data is then defined as the union of all those individual profiles: in the case of Functions we use simple set union, while in the case of call graphs or calling context trees we define the union in the natural way, by computing the union over the graph, respectively tree, node and edge sets. We call the resulting profile the application's *model*. Since the union operation is associative, it does not matter whether the model is computed in a step-wise, iterative way, i.e., after each individual profile is collected, or if one instead computes the union once over all collected individual profiles.

Due to this property, when a model appears too restrictive, it can easily be expanded by joining the application's current model with new execution profiles. This is a fully automatic process. A narrowing conversion can only be done by hand. In the case where the model contains calling context trees, however, it suffices to delete a single node to cut off this node's entire transitive closure, which makes such a modification comparatively easy to perform, especially if proper tool support is provided to visualize and explore the calling context tree [74]. This step merely requires to identify a starting point for the "forbidden" feature in question.

[2]Attacks that rely on modifying code can be avoided through existing techniques like binary attestation.

When, as in our approach, the other abstractions (function sets and call graphs) are computed from calling context trees, the same narrowing conversion can be used to update those models as well. In the other case, where call graphs or function sets are created directly, an equivalent narrowing conversion must be applied on the respective abstraction instead.

Some properties (e.g., multi-user) of an application influence the behavior of an application in such a way that models generated by using different values of these properties differ much from each other. Consequently, it will not be useful to generate one model for all values of these properties. For example, an administrator has obviously totally different access rights to some modules of the application than a normal user. For this reason, clients should not share models among another even if the application is used jointly. This deficit is probably caused by joining all profiles using a simple union operation. Accordingly, this causes loss of information which has been gathered in profiles individually. Sharing could dilute the model, as for some client it could cause malicious behavior to go unnoticed if behavior similar to the malicious behavior ("similar" in the sense of what the profile characterizes) was observed during the other client's learning phase. This can be considered as a limitation of our approach.

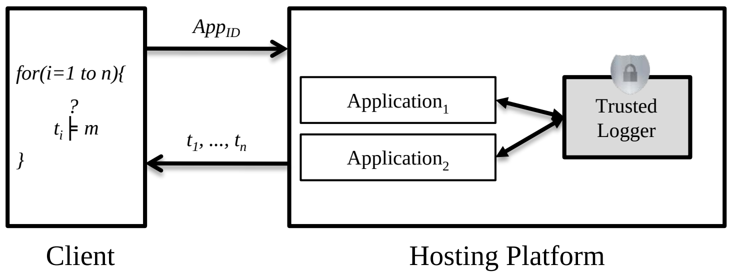
In particular, it should be noted that all approaches to anomaly detection, including our own one, are susceptible to mimicry attacks [96, 108], in which an attacker tries to cause behavior that is malicious, nevertheless mimics legal behavior in such a way that the malicious behavior remains undetected. This problem can be mitigated somewhat by keeping the application model undisclosed, but to the best of our knowledge no way to absolutely avert mimicry attacks is known to date.

3.1.3 Platform Reference Architecture

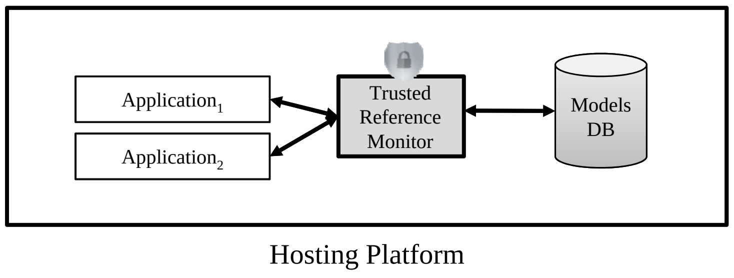
In this section, we present two possible variants of platform architectures for behavior compliance control. In the first architecture, the client is interested in verifying the behavior of his outsourced application *after* its execution. We call this variant *passive compliance control*. In the second variant, called *active compliance control*, the client observes the behavior of the outsourced application at runtime, which allows the client to react upon any detected untrustworthy behavior just in time.

Both variants require the presence of a trusted behavior measurement component (such as the Trusted Logger and the Trusted Reference Monitor explained below), as well as secure storage on the host that performs the computation. Assuring the integrity of these components is a complementary problem from behavior compliance control and may be achieved through several means.

For this reason, the architectures described in this section are meant to be generic. However, basing on technologies from Trusted Computing, e.g., integrity measurement and remote attestation, we will describe at the end of this section a concrete



(a) Passive compliance control



(b) Active compliance control

Figure 3.3: Architectures for passive and active compliance control

instantiation of the generic architectures that fulfills these requirements, including concrete mechanisms for verifying the integrity of collected profiles. An implemention of this instantiation is described in Sections 5.1 and 5.2.2.

Passive Compliance Control

First, we consider a platform architecture for passive behavior compliance control, which records the behavior of an outsourced application and reports it to the client in a trustworthy manner. Even though passive monitoring does not prevent untrustworthy behavior from happening, it helps the client to detect such behavior and take appropriate measures (such as legal actions against the service provider).

In summary, as described earlier, the client first computes an application model by running the software in a trusted environment, collecting profiles as described in Section 3.1.1. Subsequently, the client outsources the application to the host and executes it on the hosting platform. After execution, the host provides the client with evidence about the application behavior. The client finally verifies the evidence and decides on its trustworthiness. In the following, we detail the individual phases of this procedure.

- **Learning phase:** As described in Section 3.1.1, the client generates, in a secure manner, an application model m which characterizes the behavior of the target application, which has a unique application ID App_{ID}. Afterwards, the application is outsourced to the hosting platform. Note that this step is done in a trusted environment (for example in the private part of a hybrid cloud) and thus is assumed to be trustworthy.

- **Runtime phase:** Figure 3.3a shows the abstract platform architecture of the hosting platform. We assume the presence of trusted system measurement components, which assure the load-time integrity of the loaded applications and the trusted logger. The load-time integrity of the hosting platform can subsequently be verified by the client before outsourcing takes place. To this end, techniques such as integrity measurement and remote attestation proposed by the Trusted Computing Group can be used to assure the integrity of the platform and to securely report its state to a challenger (see Section 2.4.1).

  The trusted logger logs all events coming from the application itself (i.e., in case of using instrumented code) or from the middleware (e.g., the Java Virtual Machine), on which this application runs. As mentioned previously, a core task of the hosting platform is to generate trustworthy traces about the execution of the outsourced application. As the generated traces are security-critical, secure storage is required to assure their integrity.

- **Compliance verification phase:** As shown in Figure 3.3a, the client obtains the authenticated traces as recorded by the trusted logger component. Using a secure challenging and communication protocol the challenger sends the ID App_{ID}, of the application which was outsourced to hosting platform. The hosting platform retrieves, in a secure manner, all the traces $(t_1, \ldots, t_n)$ to all application executions corresponding to the sent App_{ID}, where n is the total number of these executions. Once the client has obtained the traces and verified their integrity, he compares the trace against the application model collected in the learning phase. We write $t \models m$ if the trace t corresponds to the model m and $t \not\models m$ otherwise. Whenever $t \not\models m$, this indicates that the trace diverged from the model and the remote execution is hence untrustworthy. In this case, the client computes the difference $t \backslash m$. This difference aids the client in determining the characteristics of the divergence, with the goal to determine if an actual attack has taken place.

  The difference $l \backslash m$ is very important to determine whether the divergence is a false-positive or an attack which has been detected. Nevertheless, this is not an easy job, since the client can either verify it manually, which is in some situations very time costly or even impossible, or automatically by re-executing the application using the same context data, which is again impossible in some cases (technically or even legally because of, e.g., license restrictions).

However, in case of re-executing the application in-house (for example, in the private part of a hybrid cloud), the client can record the resulting trace t'. We write $t \equiv t'$ if the trace t is equivalent to the trace t' and $t \not\equiv t'$ otherwise. Whenever $t \equiv l'$, the execution on the hosting platform is considered correct and trustworthy, and the divergence was a false positive. In this case, the client expands m by adding $t \backslash m$ to it. By doing so, m can be continuously improved to decrease the overall false positives rate. If otherwise $t \not\equiv t'$, the execution is considered untrustworthy. However, concrete implementations of the "$\models$", "$\backslash$" and "$\equiv$" operators depend on the kind of profile being used.

Note that, in contrast to intensive detection, where a very low false positive rate is imperative, our approach can tolerate higher rates. Essentially, the false-positive rate determines how much outsourced computation has to be re-done in-house (cf. hybrid cloud environment). Thus, a false-positive rate in the order of a few percent may well be acceptable, as still the bulk of the computation is performed by the hosting platform.

Active Compliance Control

While passive compliance control of program behavior is a good way to decide on its trustworthiness in retrospect, it cannot suppress undesired actions. To make this active, it is necessary to monitor the running application and react accordingly as soon as a deviation from the application model occurs.

In Figure 3.3b, we present an abstract hosting-platform architecture, assuming the presence of system integrity measurement components, which guarantee the load-time integrity of the reference monitor and the model database, which stores the different models of different applications. In addition, we assume that the Trusted Reference Monitor (TRM) is resistant against runtime attacks.

Unlike in passive compliance control, the client transfers the application's model m to the hosting platform in a secure manner. On the platform, m is stored in the model database. When the application executes, either the application itself or the application middleware (such as a virtual machine) notifies the TRM of every event characterizing the application's behavior. The TRM controls the application's behavior at each action verifying the compliance of that action with the reference model m. In case of non-compliance, the TRM acts appropriately according to a security policy.

The difficulty and efficiency of this approach depend on the chosen granularity of the application model. For function sets, suppression logic can be added to the program quite conveniently at calls to functions that are outside the allowed-function set. For a call graph, it is equally possible to disallow all calls that do not correspond to a call edge in the model. This, however, does not hold for virtual method calls in object-oriented programs. For such calls, the call target can often only be determined at runtime. In these cases, a potentially offensive callee must be made aware of its runtime caller in order to perceive a model violation. Monitoring compliance with a model based on calling context trees is difficult, as conformance depends on the entire stack configuration. A suitable implementation will hence use an external monitor to keep track of the callstack as it evolves. Such an approach is likely too overhead-prone to monitor regular applications, but it may be feasible when monitoring the behavior of more abstract applications such as automated business processes, where the model is usually very small and runtime monitoring therefore reasonably fast.

A Concrete Instantiation of the Reference Architecture

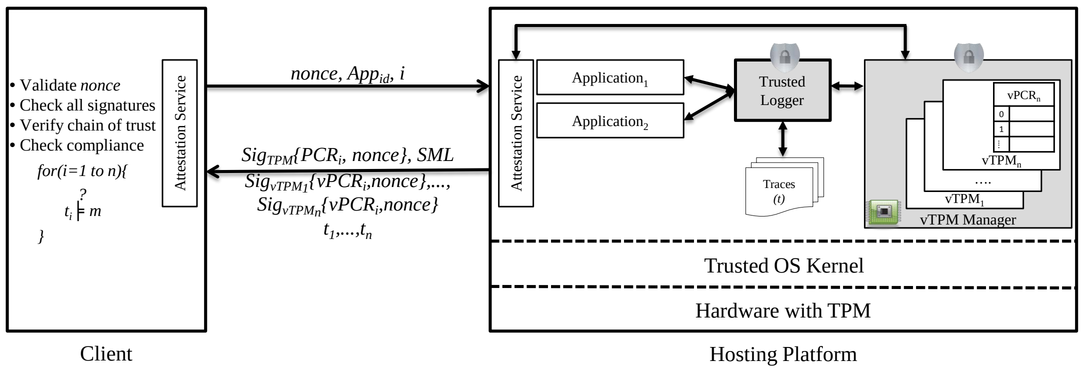


Figure 3.4: A concrete instantiation of the low-level behavior compliance control

Here, we propose an instantiation of the reference architecture for passive compliance control. However, most the components we use in this instantiation can be also used for active variant. Our particular choice is the use of functionalities of Trusted Computing to assure the integrity of both the platform and the generated profiles. For this, we use binary measurements to assure the integrity of each running component, and remote attestation to report the state of the platform and the profiles to the client. In the context of outsourced applications, the use of a single hardware TPM is not sufficient, since many applications are run on one platform, each of which can be executed many times. We hence use the concept of virtual TPMs, allowing to a assign a (unique) virtual TPM instance to each outsourced process. All vTPMs are managed by a vTPM manager, which provides an interface to create, manage and access vTPM instances; the vTPM manager is notified whenever an application instance is started. When notified, the manager creates a new vTPM instance and associates it with the application instance.

In the following, we explain the details of these components:

Setup: The setup is done as described in the previous section. The result of the learning phase is an application model m, which characterizes the behavior of the application. Afterwards, the application, whose ID is App_{id}, is outsourced to the hosting platform.

Behavior measurement: As shown in Figure 3.4, the use of TPM and the trusted OS kernel on the hosting platform assures integrity of Trusted Logger, i.e., by building a static and dynamic chain of trust. Using trusted boot, we build a chain of trust starting from the TPM chip up to the kernel, all target applications and finally the logging facility. The load-time integrity of the hosting platform can subsequently be verified by the client (before or even after outsourcing takes place) using classic binary remote attestation, as described below.

The Trusted Logger logs the events coming from the application itself (i.e., in case of using instrumented code) or from the middleware (e.g., the Java Virtual Machine, JVM), on which this application runs. As mentioned previously, a core task of the hosting platform is to generate trustworthy traces of the execution of the outsourced application. As the generated traces are security-critical, secure storage is required. To this end, we use facilities of TC as well, by recording a hash chain of all trace events in one fixed PCR register of the TPM chip.

To assure integrity of stored traces, a particular vPCR i is chosen to hold a hash chain of all recorded events (e.g., the set of nodes V in calling context trees or call graph). Whenever a new trace entry is generated, the vPCR i is extended by hashing the trace entry using SHA-1 and running the `TPM_Extend` command as described in the TPM specification [105]. The trace entry itself is stored in external (untrusted) storage.

Thus, after the outsourced application terminated, the vPCR register i of the vTPM associated to the application contains a (securely stored) hash chain of all recorded events; further, the trace t of all events is available on storage.

Attestation of the platform and the traces: In the remote attestation process, the client obtains the authenticated trace as recorded by the Trusted Logger component. The operation proceeds similar to TCG's remote attestation. As shown in Figure 3.4, the attestation service of the client sends a 160-bit *nonce*, the PCR index i, which is supposed to contain the hash chain, and the application ID App_{id}. The attestation service of the hosting platform's TPM then forwards the received information to the TPM, which in turn signs the desired PCR value and the *nonce*. The result is the signature Sig_{TPM}. In addition, the attestation request is forwarded to the vTPM manager, which in turn forwards it to all vTPM instances ($vTPM_1, ..., vTPM_n$) that correspond the traces $t_1, ..., t_n$ of the application whose ID is App_{id}, and where n is the total number of these instances. The result of this operation is a set of signatures $Sig_{vTPM_1}, ..., Sig_{vTPM_n}$.

All these values are sent then to the client, who verifies the correctness of the signatures and validates the PCR value against the SML, and vPCR values against the corresponding traces $t_1, \ldots, t_n$.

Compliance verification: Once the client has obtained the traces and verified their integrity, he compares the them against the application model collected in the learning phase, i.e., verifies whether $t \models m$. The verification is done as described above in compliance verification phase of the passive compliance control.

3.2 High-Level Behavior Compliance Control for Business Processes

In Section 3.1 we introduced our first approach for capturing and controlling behavior of outsourced applications. We described how our approach is applied on a lower level of abstraction, i.e., directly on program's code level. In this section, we introduce another approach to behavior compliance control, dedicated to the more abstract level of automated and executable business processes.

In particular, we consider a concrete delivery model of Cloud Computing, called platform-as-a-service (PaaS) [21]. In PaaS, a cloud provider hosts hardware, the operating system, the platform middleware (such as a business process execution engine) and a database management system as well as other services. The client delivers the software using tools and/or libraries from the provider, controls software deployment and is able to modify its configuration settings.

As mentioned previously, our goal is to build a system architecture and to provide techniques which allow both, a hosting platform $\mathcal{H}$and a client $\mathcal{C}$, to take advantage of business process outsourcing without being concerned about malicious workflow behavior of outsourced business processes (attacker model is described in Section 3.2.4). Here, a client provides a logging (attestation) policy in addition to the business process description, and expects from the hosting platform trustworthy evidence about the correct execution of these processes. Before starting to discuss the details of our approach, we discuss in the following some definitions of a business process.

Hammer and Champy [60] define a business process as a collection of activities that take one or more kinds of input and create an output that is of value to the client. However, we need a more formal definition of the notion of business processes in order to use its terminology in the proposed approach. Since our approach deals with the flow of business processes, it is important to identify which elements in a business process can play a role in influencing the flow an executable business process and accordingly the correct execution of it.

We slightly modified the definition of [70] and define a BP is as a tuple:

Definition 4 (Business process). *$BP = \langle BP_{id}, A, M, F, T, D\rangle$, where*

- *BP_{id} is a unique ID of the process;*
- *A is a set of activities with one activity marked as the start activity a_0. In addition, there exist two disjoint subsets A_{int} and A_{ext} of A such that $A = A_{int} \cup A_{ext}$. A_{int} is the set of all activities which do not need an external service call (i.e., the computation is directly done inside the BP-Engine). On the contrary, A_{ext} contains all activities that do need external service calls;*
- *M is a set of possible process messages, where each message is composed of a set of variables;*
- *F is a set of connectors, all of type $\{XOR, OR, AND\}$;*
- *T is a transition function $U \to 2^U$, where U is a set of process elements and is defined as $U = A \cup F$; and*
- *D is a data transition function, where $D_{in} : A \to M$ defines input parameters and $D_{out} : A \to 2^M$ defines output parameters for each activity $a \in A$.*

This formal definition covers abstract processes and is not bound to a certain specification language. It can, however, be used to describe processes expressed in BPEL and is therefore used throughout this approach.

Every single execution of a BP is called a business process instance (PI). These PIs are run and managed using a business process engine, such as Apache ODE[3]. Every PI has a unique ID, denoted by PI_{id}, across all business processes executed on the BP-Engine. The engine is also needed to send and receive messages, handle data manipulation as well as in the communication with external services.

3.2.1 Overview of the System Architecture

As mentioned before, a client $\mathcal{C}$wants to be assured about the correct execution of his outsourced business processes running on the hosting platform $\mathcal{H}$. That is, $\mathcal{H}$ must provide trustworthy evidence about the integrity of his computing platform and the correct execution of the outsourced processes to $\mathcal{C}$. The architecture which allows issuing such evidence should fulfill the following four requirements:

[3]http://ode.apache.org

R_1 *Trustworthy system architecture from boot to the business process level.* The trustworthiness of the higher levels of a system requires the trustworthiness of all lower levels (from the operating system level down to the hardware level of the system). The whole system's trustworthiness must be based on a root-of-trust.

R_2 *Multi-tenancy support.* The system of the provider must be able to host business processes for many clients simultaneously in order to be cost-effective.

R_3 *Support of multi-instance business processes for BP engines.* In practice, process instances are created whenever abstract business processes are executed. For the business process' execution to be considered trustworthy, every process instance must be trustworthy, even when running simultaneously with others.

R_4 *Tenants privacy.* Each tenant should be able to verify his remote business processes without gaining information on business processes of other tenants that run on the same hardware platform.

Figure 3.5 shows the overall architecture of the client platform, the hosting platform, and the communication between them. In summary, the client outsources a business process with id BP_{id} to the hosting platform, which uses virtualization technologies and therefore has different domains. The hosting platform runs the BP on a BP-engine installed on a certain domain and logs critical actions during execution. At the same time, the client can "attest" the hosting platform after execution to obtain a (signed) log of the execution of his BP, which he can validate locally.

On the hosting platform, our approach leverages a hardware TPM to build a *chain of trust*, documenting the integrity of the computing platform. As in the standard integrity measurement of TC, each component measures the next one before passing control to it, and notifies the TPM to update the state of the platform, which is reflected in the PCR registers. Once the boot process is finished, control is passed to the hypervisor, which is responsible for loading the guest domains (Domain $1, \ldots, n$ in Figure 3.5). Since we consider the PaaS delivery model of Cloud Computing, we assume in our architecture that every tenant has his own guest domain. In order to securely store the measurement values of all components in a domain, including the operating system kernel, we use a *domain virtual TPM manager*. This vTPM manager provides an interface to control and manage vTPM instances, so that every domain is securely associated to a different domain vTPM instance. All created vTPMs and the vTPM manager run in a separated and secured domain (e.g., dom0 in Xen). All vTPMs and the vTPM manager are supposed to be trusted by all tenants and therefore belong to our Trusted Comput-

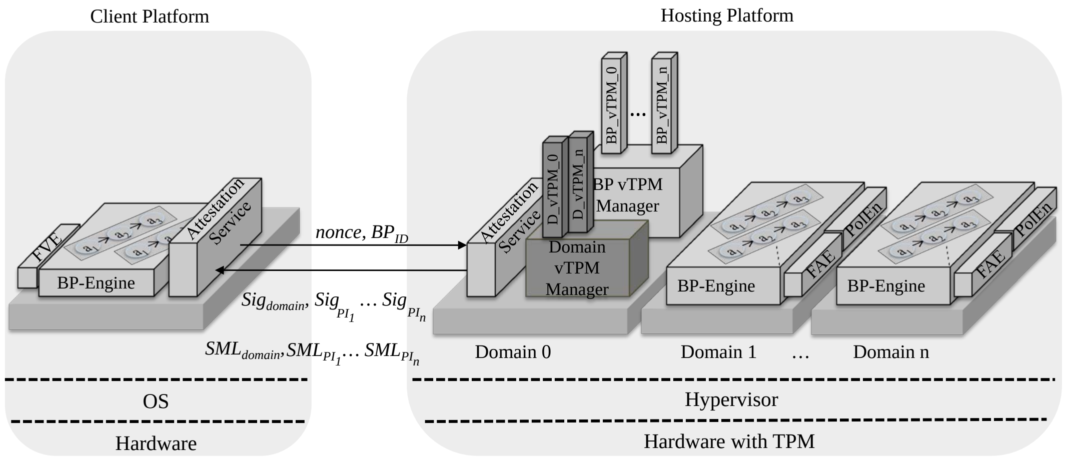


Figure 3.5: The hosting platform architecture

ing Base (TCB). Note that we propose different approaches in Chapter 4, which provide more efficient and scalable hardware-based security for virtual TPMs.

After passing control to the *trusted OS* installed in a domain, the chain of trust is extended to include all middleware applications and their extensions. The trusted OS maintains a measurement log, which contains all components loaded at runtime. We denote this log by SML_{domain}. Since we consider business processes written in a special specification language, our architecture includes a business process engine (BP-Engine). The BP-Engine is measured by the underlying level (e.g. the trusted OS), and its measured value is extended in the corresponding PCR of the domain vTPM instance. The BP-Engine is equipped with a *Policy Engine (PolEn)* and a *Flow Attesting Extension (FAE)*. The former is responsible for identifying the elements in a business process that need to be logged during execution, while the latter is responsible for measuring these elements and extending the measured values in the corresponding vTPM. Note that we assume that the FAE is trusted once its measurement corresponds to a well-known and trustworthy reference measurement. That means it is not prone to TOCTTOU attacks. This way, the chain of trust is extended to include the business process.

Regarding the client platform, we assume that the client trusts his own environment; his platform therefore does not require extra security mechanisms. It is only equipped with a BP-Engine, which is extended to have the *Flow Verification Extension (FVE)*. FVE can be used to verify the (signed) logs provided by the hosting platform. More details about the FVE and the communication between the client platform and the hosting platform are given in Section 3.2.3.

3.2.2 Integrity Measurement of Business Processes on the Hosting Platform

This subsection details which duties need to be carried out by $\mathcal{H}$ to prove flow correctness of executed business processes. We identify hereby two main duties; the first is to provide a mechanism to log sensitive activities of business processes, and the second is to provide secure storage for the logged information.

Providing secure storage. Since we consider multi-instance business processes (as required by R_3 of Section 3.2.1), the use of a single hardware TPM, which has only 24 PCR registers [6], is not adequate. In fact, each PI requires an independent PCR register, to which hashes of all executed sensitive activities are extended. If we consider thousands of PIs, it is therefore not possible to store the log information of these PIs in the PCRs of a single hardware TPM. One can theoretically store the states of all PIs using only one PCR register by extending all their measurements using the SHA-1 hash. However, this would result in problems later. Remember that in the remote attestation process, $\mathcal{C}$ has to recalculate the quoted PCR value using the SML. Consequentially, all logged information of all executed PIs on a single host would be needed during verification of one of them.

For example, let a_1 and a_2 be two executed activities within a process instance PI, and a'_1, a'_2 be within PI'. Let the BP-Engine execute the activities as follows: a_1, a'_1, a_2, a'_2. Later, to verify the SML against the quoted PCR hash value in the attestation process using one PCR, it is necessary to involve PI' in the verification of PI and vice versa, which makes the verification impossible, since the challenger does not have the reference values (i.e. RML) to verify the BPL entries against it.

Another problem of having only one hardware TPM is caused by multi-tenancy. As required by R_2, the hosting platform executes business processes for many tenants. That is, the quoted PCR and the SML would have to be sent to different tenants, and each of them obtains information about the business processes of the other. This would clearly violate the privacy of the clients.

Thus, we have to separate the log information in a manner so that every PI has an independent log. Accordingly, it must be possible to calculate the state of each PI separately. To solve this problem, we propose using a second vTPM manager, called *BP vTPM Manager*, which also controls and manages vTPM instances. Every BP vTPM instance is associated to one business process instance and stores its measured values. We refer to Section 5.2.2 for more details about the construction and implementation of the vTPM manager.

The logging approach. The second duty of $\mathcal{H}$ is to provide a logging mechanism which convinces $\mathcal{C}$ of the trustworthiness of executed business processes. For this purpose, we propose the use of a target/actual flow comparison. This comparison forms the core security guarantee about the correct execution of every PI. Hereby we distinguish between two types of flows, f_{target} and f_{actual}. While the target flow f_{target} specifies the expected and allowed flow of the business process, the actual flow f_{actual} specifies the flow executed by $\mathcal{H}$.

The target/actual flow comparison can deliver correct results only if FAE can log all information of the BP which influences both types of flows. To identify these elements, we give a look at Definition 4. We identify that only the set M influences f_{target}. That is, the attestation policy of f_{target} should, in this case, include logging every modification to any variable in M. f_{actual} is specified by logging the executed activities, denoted by A_{exe}, at runtime. Similarly, the attestation policy of f_{actual} contains every executed activity. On the contrary, the BP_{id} element trivially does not influence both flows. The sets F, T and D are also irrelevant for both flows. However, they are still required for performing the comparison at $\mathcal{C}$.

Whenever a PI is created, the FAE is notified and a BP vTPM instance is created by the BP vTPM manager in the root domain and associated to the new PI. Also a new $SML_{PI_{id}}$ is created in the guest domain. The FAE collaborates with the extension PolEn to identify which elements must be logged. That is, for the aforementioned policies, $SML_{PI_{id}}$ contains log entries about all activities, as well as every data manipulation that occurred to the set of messages M. All these log entries are stored in the corresponding vTPM instance by hashing the BP specification part of the logged entry and extending the old state using SHA-1. This way, it will be possible to recreate both flows (f_{actual} and f_{target}) using the $SML_{PI_{id}}$.

3.2.3 Attestation and Verification of Executed Business Processes

$\mathcal{C}$ can start the verification of a BP at any time after the BP was executed on $\mathcal{H}$. That is, different BP vTPM instances and different $SML_{PI_{id}}$ must have been created as described in Section 3.2.2. The verification is divided into three phases; the remote attestation, the signature and log verification, and the flow verification phase.

The remote attestation phase: The goal of this phase is to securely report the state of all executed business processes to the relevant client. The attestation process starts by sending a 160-bit nonce and the BP_{id}, in which $\mathcal{C}$ is interested, to $\mathcal{H}$; more specifically to the attestation service running on $\mathcal{H}$ (see Figure 3.5). The attestation service forwards the request to both vTPM managers. The domain vTPM manager forwards the request again to the vTPM instance associated to the client's domain. The nonce and the value of the requested PCR register are then signed by the vTPM. This signature is denoted by Sig_{domain}. Similarly, the BP vTPM manager receives the nonce and BP_{id}. Since a BP might execute multiple

PIs, where every PI has his own vTPM, the manager first retrieves all vTPMs corresponding to this BP_{id}. Again, the nonce and the requested PCR values are signed by all vTPMs instances, which correspond to the sent BP_{id}. Similar to the $SML_{PI_{id}}$, we denote these signatures by $Sig_{PI_{id}}$.

The attestation service now sends these signatures (Sig_{domain} and $Sig_{PI_{1,\dots,n}}$), the SML_{domain} and the set $SML_{PI_{1,\dots,n}}$ to $\mathcal{C}$, which starts verifying the signatures and logs as described below.

The signature and log verification phase: After receiving the response, $\mathcal{C}$ needs to verify the evidence delivered by $\mathcal{H}$. To this end, $\mathcal{C}$ first verifies the signature Sig_{domain} using the corresponding $vAIK_{pub}$. Afterwards, $\mathcal{C}$ recomputes the state of his own guest domain using SML_{domain}. If the resulting state equals the signed PCR value, the SML_{domain} can be considered untampered. Similarly, $\mathcal{C}$ verifies the signatures $Sig_{PI_1}, \dots, Sig_{PI_n}$ and the logs $SML_{PI_1}, \dots, SML_{PI_n}$ as explained before.

Note that all signatures and logs can be verified on the fly or stored for later use in case a dispute arises.

If all aforementioned steps finished successfully, $\mathcal{C}$ can start verifying the correctness of the flow as explained in the next phase.

The flow verification phase: In this phase, $\mathcal{C}$ decides whether a specified PI was executed correctly. More specifically, if f_{target} of a PI equals f_{actual}, then the execution of this PI is verified correct. As established in R_3 of the previous section, to consider a BP as trusted, every PI of it must be verified correct.

The f_{actual} of a PI can be regenerated using the set of activities A_{exe} logged in the $SML_{PI_{id}}$. However, to generate f_{target}, FVE simulates its execution on a BP-Engine using the messages M and the data manipulation that occurred within the PI, which can be also retrieved from $SML_{PI_{id}}$.

The simulation of the BPs must fulfill two conditions. First, the simulation must not re-execute activities which need external service calls (A_{ext}). Second, the simulation must be performed with acceptable overhead such that $\mathcal{C}$ still can benefit the outsourcing. This is an applicable scenario especially in hybrid clouds.

To avoid calling external activities in the simulation process, FVE uses a transformation function $\mathcal{T}$. The goal is here to transform every activity which needs an external service call to another one which can be directly computed on the BP-Engine using the input and output data of each executed external activity. Note that all this information is logged in $SML_{PI_{id}}$. That is, after the transformation we obtain a new business process description BP$'$ which can be directly executed on BP-Engine without external calls.

Now, the execution of BP′ can be done very fast without the need of extensive resources and without mentionable overhead. Also the BP-Engine used for the simulation does not need to provide all the functionalities which are provided by a standard one (such as external calls components, big database management systems, etc.).

After executing BP′, FVE compares the resulting f_{target} with the regenerated f_{actual}. If both flows are equivalent, the execution of the PI is considered trusted.

Our approach considers only deterministic business processes. That is, for BPs which show non-deterministic behavior, there might exist multiple "correct" paths. For those parts (e.g., flow block in BPEL), FVE can accept any valid path.

Requirements Revisited

In the following we argue that our architecture fulfills the requirements defined in Section 3.2.1:

- R_1: The use of a hardware-based root of trust and secure boot guarantees the trustworthiness of the most lower level. Based on it, the chain of trust is extended to include all higher levels.

- R_2: The use of vTPMs allows the fulfillment of the multi-tenancy requirement. Each tenant has his own domain vTPM instance.

- R_3: We support muti-instance business processes by the use of the BP vTPM manager.

- R_4: Tenants privacy is fulfilled by the secure isolation of vTPMs. As described in [28], the use of a 4-byte vTPM instance identifier to each packet carrying a TPM command identifies to which vTPM the command should be delivered. The vTPM instance number is perpended in dom0, so that compromised guest domains can not forge packets in order to get access to another tenant's vTPM. In addition, since all BP vTPM commands come through the domain vTPM manager, it is not possible for a client to get access to another's BP vTPM.

3.2.4 Attacker Model

We assume the presence of an attacker $\mathcal{A}$that is not capable of modifying outsourced business process descriptions during transfer between $\mathcal{C}$ and $\mathcal{H}$ in undetectable way. This can easily be achieved using an authenticated channel between $\mathcal{C}$ and $\mathcal{H}$.

In addition, we assume that $\mathcal{A}$ is not capable of making any hardware attack in the hosting platform, especially those attacks which are targeted to any hardware security chips (e.g., TPM), and therefore it can be trusted by the client. Note, this is a common assumption when using hardware security chips.

We also assume that once a component is measured, the attacker $\mathcal{A}$ is not capable of performing runtime attacks on the measured component. That is, $\mathcal{A}$ is not capable of modifying its behavior at runtime after performing the measurement at load-time (TOCTTOU attacks). For example, once Flow Attesting Extension (FAE) is measured and its value is considered trustworthy, we assume that it behaves correctly and trustworthy. However, this is not assumed for BP orchestration component (i.e., the component which uses the transition function T to decide where the flow goes next). That means, we assume that an adversary (e.g., malicious administrator) can manipulate the orchestration of the activities involved in this BP.

3.3 Related Work

In this section, we present the related literature we reviewed to both, the low-level and the high-level BCC.

For low-level BCC, there has been a significant amount of previous work on automated property inference [78, 98, 97, 42] and anomaly detection [61, 52] on many different levels, both static and dynamic, all with their relative strengths and weaknesses. Many of those approaches could be integrated into our generic architecture defined in Section 3.1.3. We decided to define our own set of three behavior abstractions because this setup would allow us to evaluate the relative properties of those abstractions. Our approach extends all previous approaches to anomaly detection by allowing anomalies to be identified in a distributed but trustworthy manner.

Our approach is not the first to capture program behavior in terms of calling-context information. Ammons et al. [15] show how to generate context-sensitive performance profiles efficiently, using hardware performance counters. Dynamic sandboxing, proposed by Inoue et al. [62], shows similarities with behavior compliance control. Like behavior compliance control, dynamic sandboxing relies on dedicated training runs to determine a set of legal behaviors. However, Inoue et al. only consider profiles at function granularity and validate them in two very limited scenarios; in particular, they do not provide a detailed, quantitative evaluation and do not consider a broader applicability of dynamic sandboxing beyond runtime monitoring.

Our approach builds on ideas from intrusion detection. In the mid-nineties, Forrest et al. [45] addressed an important problem in intrusion detection, the definition of what they call “self”, in other words a system’s normal behavior. The authors propose a method to define “self” for privileged Unix processes by recording short sequences of system calls. Behaviors that deviate from these patterns are flagged as anomalous and considered untrustworthy. Giffin et al. [49] developed an approach for validating remote system calls of mobile code outsourced from a local machine to a (potentially untrusted) client machine to detect potential manipulations. The approach uses statistical analysis to construct a behavior model from the user's binary program. During remote job execution, all system calls arriving at the local machine are checked against the model.

None of those approaches considers the scenario of behavior compliance control in outsourcing scenarios and consequently the authors did not discuss the security of the hosting platform as we do in Chapter 4. In addition, all approaches are black-box approaches (in addition to other similar works mentioned in Section 3.1.1). Compared to our approach, this gives them the advantage of being independent of any programming language or compiler. On the other hand, white-box approaches such as ours yield higher flexibility (as they can obtain more information) and finer granularity [4].

Our active approach to behavior compliance control shares some properties with state-based runtime monitoring, first introduced by Schneider [92], whereby desired security properties are formulated in terms of a finite-state automaton. During execution of the program, a trace of events is generated and checked against the automaton expressing safety properties. The main difference to behavior compliance control is that security automata usually need to be coded manually, which imposes a significant burden on programmers. While there are approaches to inferring automata from a program's code or execution trace (e.g. [16]), those approaches are often limited and have not yet been shown to scale in practice. Our approach, on the other hand, extracts behavior fully automatically.

Other authors have proposed enforcement architectures to control access to data objects distributed to remote systems [115]. Such architectures control how outsourced applications can access outsourced objects at runtime, assuming that these applications are trusted after verifying their load-time integrity. As we discussed before, behavior compliance control goes well beyond such load-time based measures.

Trusted Computing allows to remotely attest the integrity of computing platforms. Behavior compliance control goes beyond binary attestation by not only considering the integrity of the application's code at load-time, but its actual runtime behavior. Gu et al. [54] propose an approach to remote attestation that can be seen as complementary to ours. Behavior compliance control is focused on assessing the compliance of a single application's execution to its model. Gu et al.'s

[4]Note that if we apply our approach on Java applications, our approach can be then considered as black-box approach by using load-time transformation of Java bytecode. Nevertheless, the fine-granularity and the flexibility are still kept.

approach, on the other hand, rather focuses on system-wide attestation; the authors attest behavior by measuring the ways in which different processes call each other. In an approach called Semantic Attestation, Vivek et al. [58] propose to use a trusted virtual machine for remote attestation. The core idea is that such a trusted virtual machine is capable of performing code analysis and runtime monitoring. In the approach, the appropriate property checkers need to be programmed manually, though. This is in stark difference to behavior compliance control, in which application models are automatically generated from legal executions. In more recent work, Gu et al. [53] propose an architecture to attest the execution of single mission-critical subroutines of an outsourced application. The authors use the debug facilities of certain CPUs to track the execution of a specific function. The execution of the function is then transferred to a secure environment prepared by a secure kernel.

Finally, some effort has been spent on the construction of schemes for verifiable computation [48, 27], which aim at outsourcing computations to a third party, while offering a proof of correctness for the result. At the moment, these constructions are rather impractical and cannot cope with side-effects of the program execution.

A related work to the high-level BCC is the one proposed by [9]. The authors proposed a set of formal definitions to describe the behavior of business processes and introduced a framework for their attestation based on the functionalities of TC. The authors also propose a high-level and abstract description of the verification of the attested business processes. However, it is not clear how the approach proves a correct execution of a business process, since the verification mechanism they propose is very abstract. In addition, the authors did not consider the business processes executed by a BP-Engine, and accordingly the did not consider and discuss the problem of multi-instance and multi-tenant business processes.

WS-Attestation [113] proposes an extension to WS-Trust [75] to define the interaction between the entities service provider and service consumer. The authors introduced three models to describe the communication between these entities. However, they basically discussed the communication issues and did not discuss the execution of business processes on Cloud Computing environments. Also the approach proposed in [10] is only a specification of the models proposed in WS-Attestation.

Anstett et al. [17] investigated the requirements and challenges of outsourcing BPEL processes in a cloud environment considering different delivery models. However, they did not provide a solution architecture for the investigated challenges and requirements.

The authors in [114] proposed a first step towards a general framework to enforce usage control in ubiquitous computing environments. Finally, Sailer et al. [85] aim at providing an attestation architecture that can protect against firewall-bypassing and information-leaking of confidential data. In our work we consider a completely different scenario.

4 A Runtime-Secure Storage

In Chapter 3, we presented our approaches for capturing and controlling behavior of outsourced computations. We also presented techniques which assure a secure and trustworthy storage of recorded information (especially information recorded at runtime). To this end, we used technologies from Trusted Computing to prove and verify the state's trustworthiness of the hosting platform, and also to construct a manager for virtual TPMs (vTPMs) as detailed in Sections 3.1 and 3.2. This manager provides interfaces to create and manage vTPMs in a secure manner. In this chapter, we present novel approaches for building more secure runtime-secure storage, and to improve some main functionalities of Trusted Computing.

More particular, current vTPM approaches strike against one of the important principles of Trusted Computing, namely the hardware-based security. For this reason, we propose two approaches to gain strength of hardware-based security for virtual PCRs by binding them to their corresponding hardware PCRs. This is done in Section 4.1.

In addition, one of the central aims of Trusted Computing is to provide the ability to attest that a remote platform is in a certain trustworthy state. While in principle this functionality can be achieved by the remote attestation process as standardized by the Trusted Computing Group, problems like privacy and scalability make it difficult to realize in practice: In particular, the use of SHA-1 hash to measure system components requires maintenance of a large set of hashes of presumably trustworthy software; furthermore, during attestation, the full configuration of the platform is revealed. In Section 4.2, we show how chameleon hashes and group signatures can be used to mitigate of these problems.

Note that Sections 4.1 and 4.2 are based on our works detailed in [13, 11] respectively.

4.1 Hardware-based Security for vTPMs

Virtual TPMs (vTPMs), which are software-based implementations, were constructed to offer the functionalities of hardware TPMs for contexts where one TPM is not enough, e.g., virtualized environments. The main problem of them is that one looses the strength of the hardware-based implementation of TPMs, which is actually one of the main principles set by the TCG when designing TPMs. Especially, storing sensitive and security-critical data about the state of a platform or

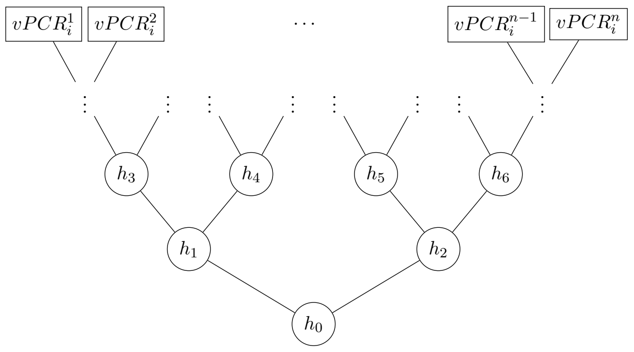


Figure 4.1: Sample Binary Hash Tree

a software in virtual PCRs (vPCRs) is problematic. To provide hardware-based security for virtual PCRs, we propose in this section two different approaches. The first uses the well-known binary hash trees and the second uses incremental hashing. Both approaches are based on the idea of binding all virtual PCRs with a specific index to the hardware PCR of the same index. That is, for conventional hardware PCRs, it exists exactly 24 hash trees. That way, any manipulation of a virtual PCR can be detected by the help of the value of its corresponding hardware PCR.

4.1.1 Hash Tree Based Binding

To bind vPCRs to hardware PCRs, we imagined the vPCRs running on a platform as leafs and the hardware PCRs as a root. To bind values stored in leafs to the root value, we decided to use cryptographic hash trees as shown in Figure 4.1. The root of each hash tree is stored in the corresponding register of the TPM. Currently available TPMs are not capable of handling hash trees, therefore we present an efficient approach to enable hash tree support for TPMs, proposing an extension to the TCG's standard. In the following, we explain our approach using three phases; the setup phase, the integrity measurement phase, and finally the remote attestation phase.

Setup Phase. We construct the hash tree in the following way: The leaves at the top of the tree present all vPCRs of a specific index i of all existing vTPMs $(1, \ldots, n)$ on a platform. For instance, $vPCR_{10}^1$ indicates the vPCR number 10 of the vTPM number 1. To increase efficiency, we propose using hash trees of fixed height l. That is, with $l = 10$, one can run 1024 vTPMs on the same platform bound to a single hardware TPM. This number is probably enough for single platforms (e.g., servers), in case of using isolated vTPMs for virtual machines. Nodes further down

Algorithm 1: TPM_Update_Leaf_Init

```
Input: old vPCR value $vPCR_{old}$,
       new vPCR value $vPCR_{new}$,
       hardware PCR index $i$,
       height of the tree $l$
Output: $OK$ or $error$
if $c_i \neq 0$ then                // a hash tree execution is running
    return error;
else
    $c_i \leftarrow l$              // initialize counter with tree height
    $tmp_{old} \leftarrow vPCR_{old}$;
    $tmp_{new} \leftarrow vPCR_{new}$;
    return OK;
```

in the tree are the hashes of their respective child nodes. Figure 4.1 illustrates this process; h_0 represents the accumulated vPCR values (root hash node) that will be stored in the hardware TPM; h_0 is obtained by combining the hashes h_1 and h_2, i.e.,

$$h_0 \Leftarrow hash(h_1||h_2),$$

where $||$ indicates the concatenation operation. Similar to h_0, all intermediate hashes are computed. That way, the calculation of h_0 depends on the calculation of the leaves and all intermediate nodes in the hash tree. Consequentially, any manipulation to one of the leaves can be detected.

Integrity Measurement. Once a vPCR value needs to be updated, the vTPM is notified about the new measurement and the new value of the vTPM is bound to the corresponding hardware PCR as explained in the setup phase. In addition, the SML of this vTPM is also updated.

More specific, the TSS notifies the underlying hardware TPM by starting the procedure depicted in Algorithm 1, sending the old vPCR value $vPCR_{old}$, the new vPCR value $vPCR_{new}$, the height of the hash tree l and the PCR index i of the hardware TPM that needs to be updated. This algorithm is then executed inside the hardware TPM, which in turn stores these provided values in temporary registers in the volatile storage. The algorithm returns OK if and only if the process was successfully finished and there is no hash tree updating process currently running for this PCR_i. As a summary, the goal of executing Algorithm 1 is to initialize the re-calculation process of the new root value.

After providing the hardware TPM with the old and new vPCR values, re-

Algorithm 2: TPM_Update_Leaf

```
Input: hardware PCR index $i$, $sibling$
Output: updated hardware PCR value $PCR'_i$ or $error$
$tmp_{old} = hash(tmp_{old}||sibling)$;
$tmp_{new} = hash(tmp_{new}||sibling)$;
$c_i \leftarrow c_i - 1$;                     // reduce the hash tree level by 1
if $c_i = 0$ then                              // root of tree reached?
    if $tmp_{old} = PCR_i$ then
        $PCR_i \leftarrow tmp_{new}$;
        return $PCR_i$;
    else
        return $error$;                        // the hash tree is tampered
else
    return $c_i$;          // update calculation runs, return tree level
```

calculation of the hash tree is required as shown in Algorithm 2. Since the hash tree is located outside the TPM and the algorithm must be provided with all siblings located in the way to the root of the hash tree, Algorithm 2 must be called $l - 1$ times, and provided at each time with the correct sibling of the current hash tree level. First, the TPM is provided with the sibling of the leaf (i.e., the vPCR) and hashes the old and the new value of the vPCR with its sibling. The same process is repeated until the root is reached (i.e., the tree height equals 0), otherwise the current hash tree level is returned. If the old vPCR value equals the value stored in PCR_i, the hash tree is untampered and the newly calculated root can be stored in PCR_i, otherwise an *error* is returned, indicating a potential software attack aiming at manipulating the PCR values. It does not matter who calls Algorithm 2 in the hardware TPM to provide the siblings values, more important is the provision of the correct values to calculate a correct root value, which equals PCR_i. That is, an attacker which calls Algorithm 2 after the Algorithm 1 was called, would have to deliver a collision to PCR_i in order to successfully perform an attack on the TPM in order to update the root value to another selected one. This is assumed to be hard when using a collision-resistant hash function.

Note that it would be possible to provide the TPM with all required siblings (from a leaf to the root) at once. Although this would reduce the communication overhead with the TPM, it would require at the same time the presence of enough temporary storage for all these values, which could be a problem for resource constraint TPM implementations.

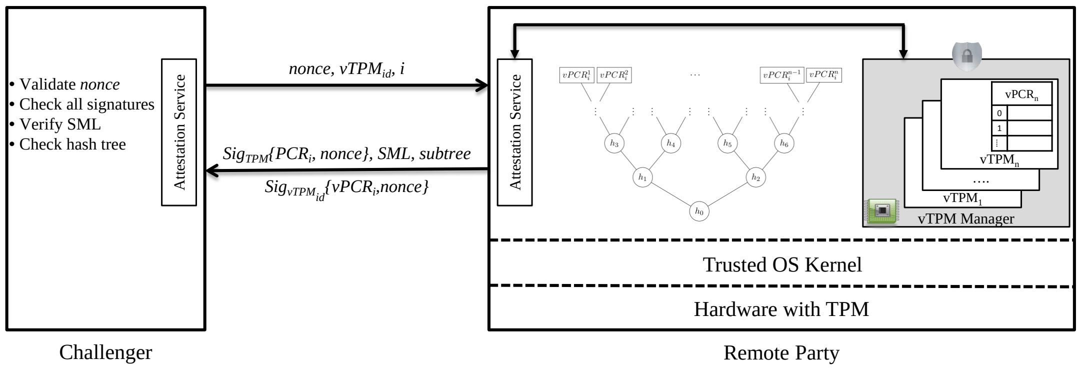


Figure 4.2: The remote attestation process using hash trees

Remote Attestation. The remote attestation process is very similar to the one described by TCG, with one more difference, which is verifying the hash tree. In detail, we assume that the challenger knows the vTPM's ID, in which he is interested. That is, the attestation service of the challenger sends a *nonce*, the $vTPM_{id}$ and PCR's index i, to the remote party as shown in Figure 4.2. The attestation service of the remote party receives the request and forwards it to the vTPM manager, which controls and manages vTPM instances. Afterwards, the vTPM manager retrieves the correct vTPM instance, which in turn signs the *nonce* together with the value of $vPCR_i$ by using a vAIK of this particular vTPM. The result is denoted by $Sig_{vTPM_{id}}(vPCR_i, nonce)$. The *nonce* is then forwarded to the hardware TPM, which also signs the *nonce* and the value of the requested PCR (i.e., the root node of the hash tree), which results in the signature $Sig_{TPM}(PCR_i, nonce)$. These signatures and the SML are finally sent to the challenger (see Figure 4.2). In addition, a subtree, which contains all siblings located in the way from the a vPCR to the root, has to be sent to the challenger. The challenger verifies the signatures and the SML. In addition, the challenger re-calculates the hash tree as described above. If the signed root value equals the re-calculated value (which means that the vPCR is untampered), and the signed value can be considered trusted.

4.1.2 Incremental Hash Based Binding

Incremental hashing is another efficient way to aggregate hash values of messages that change over time. More specific, an incremental hash function produces an updated hash value of a modified message faster than recomputing the hash from scratch. In summary, incremental hashing provides a collision-free hash function f for which the following is true. Let $x = x_1 \dots x_n$ be some input, viewed as a sequence of blocks, and say block i is modified to x'_i. Let $x' = x'_1 \dots x'_i \dots x'_n$ be the new message. Then given $f(x), i, x_i, x'_i$, it is easy to compute $f(x')$ [23]. We can apply this paradigm to the idea of binding vPCR values to a single hardware PCR value to have the following mapping:

- $f(x)$ represents the old hardware PCR value,
- i represents the vTPM instance number k,
- x_i represents the old value of the vPCR,
- x'_i represents the new value of the vPCR, and
- $f(x')$ represents the new accumulated value stored in the hardware PCR.

To this end, we propose here an approach which uses incremental hashing to aggregate all vPCR values of a platform and update the aggregated value after every extend operation performed on any vTPM on the platform.

Algorithm 3 details the hash update procedure based on the incremental hashing scheme of [23], which defines different combining operation. Our particular choice is the modular multiplication MuHASH. To comply to TCG standards, we slightly modify the input parameter of MuHASH, such that the updated hash will include the history (PCR_i) of all measurements.

Algorithm 3: TPM_Increment_Hash

```
**Input**: vPCR instance number $k$, old hash-value $vPCR_{old}$, new hash-value
        $vPCR_{new}$
**Output**: updated hardware PCR value $PCR'_i$
$tmp_i = mod\_div(PCR_i, hash(k||vPCR_{old}));$
$PCR'_i = mod\_mult(tmp_i, hash(k||vPCR_{new}||PCR_i));$
**return** $PCR'_i$;
```

Setup Phase. The incremental hash-based binding approach presented herein can be used with an arbitrary number of vTPMs. Adding and removing PCR values of vTPMs is done by multiplying/dividing the corresponding hash values with the aggregated hardware PCR value. In the setup phase, to bind all vPCRs of a vTPM to the value of the hardware PCR_i, all corresponding $vPCR_i$ of each vTPM instance k are combined according to the following equation, where m is the prime modulus, i is the index of the PCR register and n is the total instance number of all vTPMs existing on the same platform:

$$PCR_i \Leftarrow \prod_{k=1}^{n} hash(k||vPCR_i^k) \; mod \; m$$

Integrity Measurement. For continuous integrity measurement, the update of a PCR value is performed according to Algorithm 3. First, the algorithm removes the old vPCR value by dividing the current PCR value by the old vPCR's hash value (see *mod_div*). The result is then stored in a temporary value tmp_i. Afterwards, to add the new vPCR value to the accumlated value, the algorithm multiplies tmp_i by the new hash value (see *mod_mult*). Note that PCR_i is included in the updated value PCR_i'. This is very important to do in order to avoid resetting a PCR value and to keep track of the update history of a PCR.

Remote Attestation. The verification of the remote attestation process has to include the integrity verification of the incremental hash. In addition to the SML provided by a TSS of a vTPM, an SML for the incremental hash updates is provided. As defined by TCG, a challenger first verifies all signatures and the SML of the vTPM. In addition, the challenger uses the SML of the hardware TPM, which has all incremental hash updates, to verify the integrity of vTPM itself.

4.1.3 Attacker Model

We assume the presence of an attacker which is not capable of performing physical attacks on the hardware root-of-trust, the TPM. That is, the TPM is utilized according the specifications of TCG.

We consider in this approach those platforms which use virtualization technologies and therefore vTPMs to secure virtualized machines (VMs), or platforms which use vTPMs to secure instances of applications (see Sections 3.1 and 3.2). Here, we assume that all measurement units running on this platform (also in virtualized machines) are supposed to be not attackable by the attacker, and therefore they belong to the Trusted Software Stack (TSS). That is, all integrity measurements of running software and hardware are supposed to be correct, and reported to the corresponding vTPM to be stored in a vPCR.

However, the vPCRs themselves are still assumed attackable, even if they run in a separated environment. That is, the attacker is capable of modifying a selected vPCR to any arbitrary value. Since the hash tree is stored outside the TPM, it is also assumed attackable. However, the root value of it is stored inside the hardware TPM and therefore is supposed to be secure. Hereby the goal of the attacker is to load and execute malicious software in one or more VM and modifying specific vPCR values (after reporting the value of the malicious software) to another, for the challenger, trusted value.

4.1.4 Related Work

Unfortunately, the current specification of the TPM does not support hardware-based security for systems using virtualization and cloud computing technologies. Though there exist in literature designs supporting resource constrained embedded systems [43] and arbitrary number of virtual TPMs [90], they do not address the above problem. Virtual TPMs in these approaches belong therefore to the TCB of a platform.

The concept of hash trees has been used in many different contexts. In the area of Trusted Computing, hash trees were applied in [112] to protect memory regions using the region block size and the number of memory updates as parameters for the hash tree. Schmidt et al. [91] used hash trees during the integrity measurement process to create tree-formed measurements, in which the measured components represent the leaves and the PCR values represent the roots. The goal of this work was to allow detecting the position of a possible manipulation of an SML, which was possible in case of using linear ordered measurements (like in TCG standard) only by checking the integrity value of each entry in the SML. Another work applied the concept of hash trees in TC is the one presented by Sarmenta et al. [89]. The objective of the authors was to create very large number of virtual monotonic counters on an untrusted machine with a TPM. The virtual counters can be then used to detect illegitimate modifications to shared data objects (including replay attacks and forking attacks) [107]. The authors proposed for this the use of additional TPM commands in order to calculate hash tree node and root values in a secure manner. However, we apply hash trees in our approach in a completely other context, specifically, to bind virtual PCRs to hardware PCRs, which is a security problem of virtual TPMs.

A conventional TPM is implemented, in general, as an Application Specific Integrated Circuit (ASIC) [105], and therefore cannot be updated after deployment. However, there exist approaches in literature for supporting a flexible update of cryptographic algorithms on the TPM using the reconfiguration technology such as Field Programmable Gate Array (FPGA) as proposed by Malipatlolla et al. in [95]. Further, Feller et al. [43] porposed a novel architecture for a TPM utilized to provide intelectual property (IP) protection and to design a trustworthy embedded system.

4.2 Group-Based Attestation: Enhancing the Privacy and Maintainability

Throughout this thesis, we often used technologies from Trusted Computing to assure a platform's integrity. However, research has identified some problems related with Trusted Computing. These problems include privacy and maintenance issues as well as sealing and communication difficulties.

After clearly explaining some problems of Trusted Computing, we approach, in this section, the mentioned problems by proposing three novel attestation techniques, which are based on either chameleon hashes or group signatures. More information about these two cryptographic systems can be found in Section 2.3

The first and second techniques allow balancing configuration privacy with the control precision of the attestation process and substantially decrease the overhead for maintaining RMLs, while the third one provides more flexibility for the challenger in control precision but offers no privacy advantage when compared with the TCG attestation[1].

4.2.1 Attestation Problems & Related Work

Integrity measurement according to the TCG specification seems to be a promising way to check trustworthiness of systems. However, the suggested remote attestation process has several shortcomings. In the following we list some problems which are related to the approaches we present later in this section:

- *Privacy.* We can distinguish between identity privacy (IP) and configuration privacy (CP). IP focuses on providing anonymity for the attested platform. This problem can be solved by Direct Anonymous Attestation (DAA) [33, 34, 36]. On the other hand, CP is concerned with keeping configuration details of an attested platform secret, since its disclosure may lead to privacy violations. Still, the challenger system must be assured that the attested platform indeed is in a trustworthy state. Our proposed approaches focus on providing CP. Note that CP and IP are orthogonal problems, i.e., our solution can be used in conjunction with mechanisms that guarantee IP.
- *Discrimination and targeted attacks.* By using remote attestation, product discrimination may be possible. For example, in the context of DRM environments, large operating system vendors and content providers could collaborate and force usage of specific proprietary software, which restricts the freedom of choice. Furthermore, an adversary could leverage the precise configuration of the attested platform and perform a specific targeted attack [68].
- *Maintainability.* A further drawback lies in the maintenance of Reference Measurement Lists [86]. The TCG attestation requires the challenger to

[1]Note that the proposed approaches can be applied in the architecture proposed in Chapter 3. We use in this section the term “software” as an example for integrity measurement, since this is the standard in Trusted Computing. However, the approaches work also on any kind of data (such as business process descriptions and nodes of a calling context trees).

maintain a Reference Measurement List (RML), which contains hashes of all trustworthy software, to validate the received measurements. Consequently, software updates or patches require distribution of new hash values. For this reason, the management overhead increases to a point where attestation becomes impractical. Consequently, keeping these RML lists up-to-date involves high management and communication efforts.

- *Sealing.* Besides remote attestation, TCG offers the ability to seal data to the configuration of a specific platform. Again, any software update or configuration change can lead to a completely new platform configuration state and consequently hinder unsealing [82].

Sadeghi and Stüble [82] approached the above mentioned problems by the introduction of Property-based Attestation (PBA). By applying PBA, the attested platform proves that it fulfills certain semantic security requirements, called "properties". This way, the concrete configuration of a platform does not need to be disclosed. However, PBA requires an extension of TPM or alternatively a Trusted Third Party along with a Trusted Attestation Service, which is responsible for translations between properties and software. Semantic attestation [59] verifies that the behavior of a platform fulfills given particular high-level properties. WS-Attestation proposed by Yoshihama et al. [113] employs PCR obfuscation to hide software versions; however, maintainability remains a problem [9].

In [55], a model-driven remote attestation was proposed providing evidence of trustworthiness by behavior compliances. However, these proposals involve changes in operating system design or even changes to the trusted hardware module. On the contrary our proposals only add a new cryptographic primitive (chameleon hash function) in trusted kernel or introduce a PKI considering group signatures.

4.2.2 Chameleon Attestation I

The goal of the challenger in a remote attestation process is to decide about the trustworthiness of the attested platform. A honest challenger is not concerned, in fact, to know the detailed configuration of the attested platform, i.e., the specific versions of hardware and software (see configuration privacy problem above). That is, the challenger would be satisfied, if it is also possible for him to decide about the trustworthiness of the platform without knowing the details of its configuration.

Other problems, like maintaining a huge database of RML entries (considering millions of software and software versions) and sealing's invalidity of updated software, are also a big challenge. Our basic idea to solve all these problems is building "legal" *software groups.*

In particular, we recognized that the use SHA-1 hashes leads to constructing different chains of trust. Assume that $SW_{v.1}$ is executed on the attested platform,

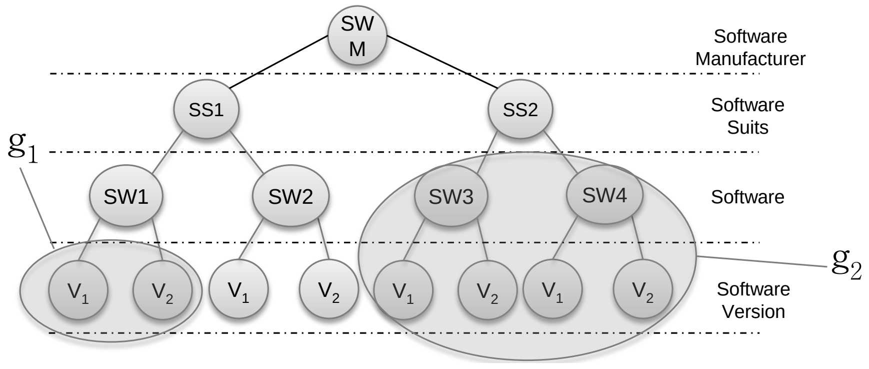


Figure 4.3: Building software groups of different granularities

we then would get a chain of trust (S_1) as follows:

$$S_1 = \mathit{SHA\text{-}1}(\mathit{SHA\text{-}1}(\mathit{SHA\text{-}1}(\ldots || \underline{\mathit{SHA\text{-}1}(SW1_{v.1})})||\mathit{SHA\text{-}1}(SW2_{v.1}))|| \ldots)$$

Now, assume that $SW_{v.1}$ is updated to another version ($SW_{v.2}$), we would get another chain of trust (S_2) like the following:

$$S_2 = \mathit{SHA\text{-}1}(\mathit{SHA\text{-}1}(\mathit{SHA\text{-}1}(\ldots || \underline{\mathit{SHA\text{-}1}(SW1_{v.2})})||\mathit{SHA\text{-}1}(SW2_{v.1}))|| \ldots)$$

Consequently, because of the different SHA-1 values of both versions, $S_1 \neq S_2$. But, we need is a technique which allows the statement $S_1 = S_2$, in such a way that the challenger in the attestation processes can be convinced that the change in chain of trust is because of a “legal” update and not a software manipulation.

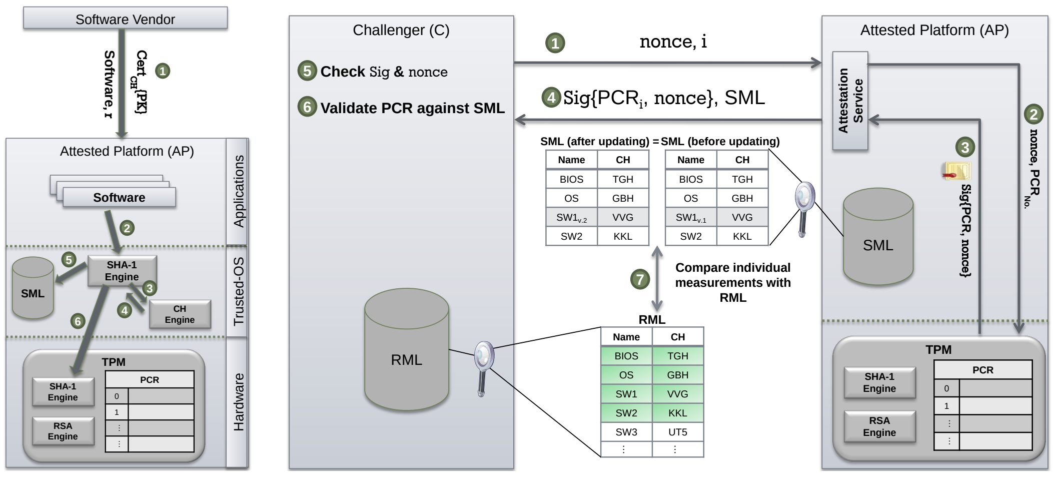
In this section we describe a novel remote attestation approach, which makes it possible for the challenger to decide on the trustworthiness of the attested platform, without knowing its detailed configuration. The assumptions listed in [86] about the attacker model are also the basis of our approach. In particular, we assume that once a measurement is stored in an RML, the corresponding software is considered trusted; additional security mechanisms must be in place to secure the integrity of the RML (this is out of scope of this work).

Our approach is based on the the concept of software grouping; that is, according to the precise scenario, these groups may e.g. contain all software products of the same vendor, compatible software products (see g_2 in Figure 4.3) or all versions of one specific software (see g_1 in Figure 4.3). We design the attestation process in such a way that we assign the same hash value to all members of a software group. To achieve this, we make use of a chameleon hash function. As mentioned in Section 2.3.1, any party who knows the public key **pk** is able to compute the hash value for a given message. In contrast, only the trusted instance holding the private key **sk** can create collisions. Based on the idea of software groups sharing the same hash value, we describe in the following a novel remote attestation process we call *Chameleon Attestation I*.

Setup phase: For each group, a trusted instance (such as a software vendor) runs the key generation algorithm **Kg** to obtain a public/private key pair $(\mathbf{pk}, \mathbf{sk})$. When establishing a new software group, the software vendor picks for the first product contained in the new software group a random r and makes it available to the attested platform by delivering it with the software. Furthermore, he hashes the program's code c of the software with the chameleon hash to obtain $h = \mathbf{CH}(\mathbf{pk}, m, r)$; for performance reasons the SHA-1 hash value of c is taken as m. The obtained chameleon hash is made public in a trusted RML. Subsequently, to add a new software c' to the same software group, he uses the algorithm **Forge** to find a new r' so that $\mathbf{CH}(\mathbf{pk}, m', r') = h$ and distributes the new r' alongside the software. Again m' is the SHA-1 hash of c' and is taken as input message. Step 1 in Figure 4.4a shows the parameters distributed to the attested platform by a software vendor.

Integrity measurement: On the attested platform, the operation proceeds in a similar way as in the original integrity measurement process, see Figure 4.4a. In particular, the software is first hashed using SHA-1 (step 2). Subsequently, the attested platform computes in step 3 the chameleon hash value h of the software using the public key **pk** and the random value r distributed alongside the software. Since the PCRs in the TPM accept only a 160-bit message to be extended to a particular register, the chameleon hash value is hashed again using SHA-1 in step 4 and the corresponding information is stored in the SML in step 5. The resulting value is finally extended to a PCR register (step 6). If we assume that groups of software are built upon software versions, chameleon hash value of $SW1_{v.1}$ equals the chameleon hash value of $SW1_{v.2}$, e.g., after updating the software (see SML in Figure 4.4b).

Remote attestation: The attestation process of Chameleon Attestation I is very similar to the standard TCG attestation process. In step 1 in Figure 4.4b, the challenger sends a *nonce* and the PCR index i, whose content has to be signed by the TPM. In step 2, the Attestation Service forwards the request to the TPM, and in step 3 the TPM signs the desired PCR value and the *nonce*, and sends them back to the Attestation Service. In step 4, the attested platform sends the SML containing the chameleon hash values instead of SHA-1 values. In steps 5-7 the challenger verifies the signature, validates the PCRs values against SML, and checks the trustworthiness of the sent measurements. Only if SML contains trustworthy measurements the attested platform is considered trusted. This way, a malicious challenger is no longer capable of disclosing



SML (after updating) = SML (before updating)

Name	CH
BIOS	TGH
OS	GBH
$SW1_{v.2}$	VVG
SW2	KKL

Name	CH
BIOS	TGH
OS	GBH
$SW1_{v.1}$	VVG
SW2	KKL

RML

Name	CH
BIOS	TGH
OS	GBH
SW1	VVG
SW2	KKL
SW3	UT5
⋮	⋮

(a) Integrity measurement (b) Attestation process

Figure 4.4: Integrity measurement and the process of Chameleon Attestation I

the specific configuration (e.g., a specific software version) of the platform.

Chameleon Attestation I is flexible in the sense that the granularity of software groups can be easily chosen to balance privacy and control precision (see Figure 4.3): If more privacy is desired, then larger software groups may be formed; on the other hand, if distinction between different software versions is an issue, smaller groups can be maintained. Note that the decision of how granular a group is, can be made only by the software vendor. Without modifying the TPM, Chameleon Attestation I supports only the *static chain of trust*, since the TPM itself does not provide functionalities to calculate chameleon hashes.

In Section 5.4.1, we provide a full implementation of Chameleon Attestation I, and in Section 5.4.3 we discuss in detail some experimental results and the consequences of using Chameleon Attestation.

4.2.3 Group Signatures Based Attestation

An alternative approach to improve the remote attestation process in terms of privacy and maintainability is possible by applying digital signatures, in particular group signatures. This requires the following modifications to the integrity measurement architecture:

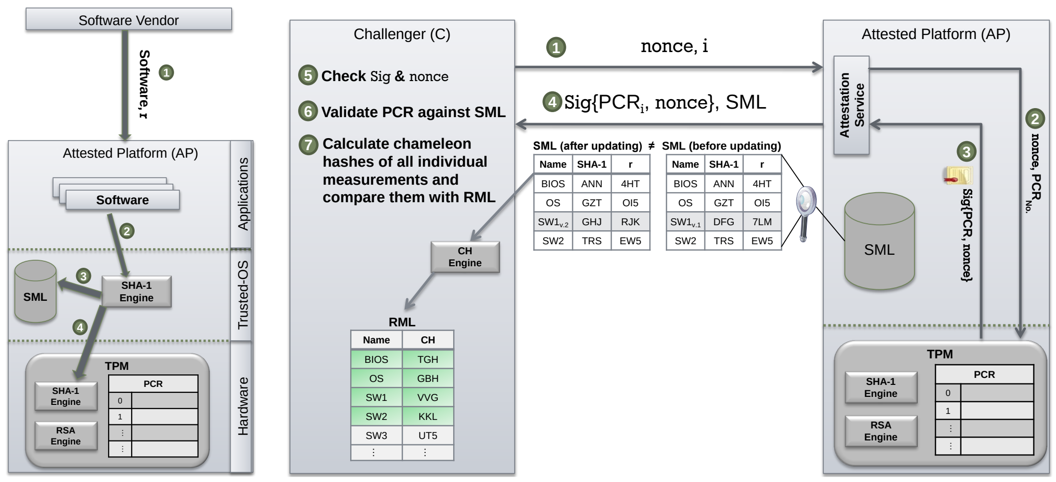
Setup phase: We again use the concept of software groups. This time, we use group signatures; each software in the software group has its own private signature key $\mathbf{gsk}[i]$, while all share a common verification key $\mathbf{gpk}$. Whenever a new product or an update of software is published, the software is first hashed with SHA1 to obtain $h = \text{SHA-1}(c)$, where c is the code of the software. Then, the hash value h is signed by the private key $\mathbf{gsk}[i]$, i.e., $\sigma = \mathbf{GSig}(\mathbf{gsk}[i], h)$. The public verification key and the signature is distributed alongside the software. Furthermore, the public keys of all trusted software groups are stored in the RML.

Integrity measurement: Whenever a software is loaded, it is hashed with SHA-1 and its signature is checked with the included public key using the group signature verification algorithm $\mathbf{GVf}$. If the signature is valid, the attesting platform hashes the public key and extends the particular PCR with the hash value of the public key of the verified software (instead of the hash value of the software). Afterwards, a corresponding entry containing the name of the software group and its public key $\mathbf{gpk}$ is stored in the Stored Measurement Log (SML). If any failure occurs, similar to the process of IMA, the corresponding PCR is set to an invalid state.

Remote attestation: The remote attestation works exactly as described in Section 2.4.1 up to the point where the challenger receives the answer from the attested platform. Then, the challenger verifies the signed PCR and his chosen nonce, validates the hash chain of the PCR against the public keys contained in the SML and checks their presence in the trusted RML. If all checks succeed, the system is considered trustworthy.

Using group signatures instead of chameleon hashes provides some advantages. While in Chameleon Attestation I a revocation of chameleon hash value requires the revocation of all group members, using group signatures allows the revocation of specific members of the group without the need to revoke the whole group. A second advantage lies in the ability of fitting a group signature hierarchy to an organization structure. That is, every product realm or series could have its own private key, while verification is performed with one single public key.

On the other hand, Chameleon Attestation I outperforms group signature based attestation in terms of performance. Chameleon Attestation needs to only compute hash value for measured software whereas group signatures need to verify signatures. While fast group signature schemes (like [31]) need about six exponentiations for signing and verification, chameleon hash functions require much less computations. For instance, our particular choice of a chameleon hash detailed in [20] performs only two exponentiations. To the best of our knowledge there exists no group signature which require less than three exponentiations.



(a) Integrity measurement

(b) Attestation process

Figure 4.5: Integrity measurement and the process of Chameleon Attestation II

4.2.4 Chameleon Attestation II

The remote attestation proposed above can be used to mitigate the privacy problem. However, there is a tradeoff between privacy and *control precision* of the approach: as the challenger is only able to see the software groups running on the attested system, the challenger cannot distinguish individual software versions any more: Assume a software vendor has developed a product $SW_{v.1}$ which is later updated to $SW_{v.2}$ because of disclosed security vulnerabilities. By applying the technique mentioned above, a challenger cannot distinguish platforms where $SW_{v.1}$ or $SW_{v.2}$ is run. When using Chameleon Attestation I we lose the possibility to efficiently revoke certain members of a software group. A software vendor can only declare the old chameleon hash value for the group as invalid and publish a new one. However, this requires an update to the challenger's RML. That is, revocation in this context means revocation of the whole software group with all of its members and not revocation of a certain member or even a subgroup.

In this section we show how chameleon hashes can be used to reduce the management overhead of maintaining large RMLs in scenarios where configuration privacy is not an issue. Instead of computing chameleon hashes on the attested platform, we can move this calculation to the challenger side. As in the system described in Section 4.2.2, the manufacturer picks one chameleon hash for each software group, publishes the hash value of each group in an RML, and sends alongside the software random values r required to compute the chameleon hash. On the attested system, the standard integrity measurement process is performed (in which SHA-1 hashes of loaded executables are stored into PCRs), except that the random values r required to compute the chameleon hashes and the SHA-1 hashes are both saved in the SML. The remote attestation process proceeds as in the standard TCG attestation, i.e., the challenger receives the signed PCR values. Subsequently, the challenger verifies the signed PCR and his chosen nonce and validates the contents of the PCR against the SML containing all SHA-1 values. Finally, for each entry in SML, the chameleon hash is computed to build software groups and validated against the RML. Figure 4.5 depicts the steps performed in the integrity measurement and attestation process.

Applying Chameleon Attestation II makes revocation of specific software group members easier. Unlike Chameleon Attestation I and group signatures based attestation, the challenger himself can refuse untrusted software versions by simply validating the SHA-1 values of these members against blacklists of revoked or untrusted group members. This leads to more flexibility for the challenger and gives him a tradeoff between maintainability and control precision.

5 Implementation of the Approaches

This chapter describes the implementation of the proposed approaches in this thesis. It provides a guidance to the technologies and techniques which can be used to implement them, and at the same time fulfill the requirements established in previous sections.

In particular, we present in Section 5.1 a Java-based implementation of the low-level BCC proposed in Section 3.1. To this end, we use a profiler, called JP2, which is an open source calling-context-tree profiler for Java. Section 5.2 provides, first, the implementation of the trusted components which generate the chain of trust used in Trusted Computing, and second the implementation details of the vTPM manager used in previous sections. Section 5.3 describes the implementation of the high-level BCC presented in Section 3.2. Here, we describe how the Apache ODE BPEL engine is used to implement the approach. In addition, in Section 5.4 we implement the approaches proposed in Section 4.2.

Note that our implementations in this chapter serve only as proof-of-concept implementations and can be improved in case of real production. In addition, the implementations are – in general – based on two different configuration settings, which are listed in the following table.

Component	Brand/Model/Version	
	Setting 1	Setting 2
CPU	AMD Phenom II X2 555	Intel Core 2 Duo T9600, 2.8 Ghz
Hard drive	500GB	250GB SATA 7200 rpm
Memory	4GB DDR3	4GB SDRAM
TPM	Siemens TPM 1.2	Intel iTPM
Boot loader	GRUB 0.97 with TrustedGRUB v. 1.1.5	GRUB with TrustedGRUB v. 1.1.3
Kernel	Linux Kernel v. 2.6.32.5 with IMA	Linux Kernel v. 2.6.27.38 with IMA
OS	Debian 6.0 linux (squeeze)	Fedora 10
TSS	tpm4java	jTSS

Table 5.1: Platform configurations for hardware and software

5.1 A Java Implementation of the Low-Level BCC

Profile Generation

To generate execution profiles, we use JP2, an open source calling-context-tree profiler for Java [88, 87]. This light-weight profiler consists of a small Java agent, which instruments the profiled application at load time, and an accompanying tool to instrument the Java runtime library ahead of time. This combination enables us to generate execution profiles which cover not only the application but also the Java runtime library itself. Moreover, JP2's profiles cover not only methods that have a bytecode representation but also method calls made in either direction across the bytecode-native code boundary. The following details are specific to a Java-based setting:

Virtual machine-based execution: The Java platform allows for easy load-time transformation of code. Hence, to introduce a runtime monitor, a client does not need to instrument his application in house. Instead, the application can be transformed remotely, by a custom class loader [71] or transformation agent. Such instrumentation is performed on the level of bytecode and requires no access to source code. JP2 does exactly this.

Generated code: The same class-loader mechanism that makes it easy to introduce a runtime monitor at load time also makes it possible to generate classes at runtime. Such classes frequently bear a randomized name, and that name must be canonicalized to ensure that the same method, up to renaming, can be reliably identified across program runs. To that end, we integrated the hashing facility from TamiFlex [29] with the calling-context-tree profiler described next.

Recursion: When using the Calling Context Tree abstraction, recursive calls can cause the profile to grow very large. One way to address this would be to "fold" those sub-trees in the CCT that exhibit a recursive structure. The generated profiles would hereby be bounded. However, what exactly counts as recursion in a language with dynamic dispatch is not obvious: Do only calls to the same target method count or also calls to a different target method of the same call site? Calls of the latter kind are frequent, e.g., when operating on objects structured using the Composite pattern [46]. Moreover, mutual recursion or, more generally, larger cycles of calls could be considered recursive as well and maybe thus subject to folding. For the purpose of this thesis we restrict the discussion to the straight-forward calling context tree abstraction produced by JP2 and do not fold recursive calls; thus, the tree structure mirrors the entire computation.

In our current implementation, we always collect full calling context trees, even if we are just interested in call graphs or function sets. Call graphs are computed from a CCT by merging nodes with the same name, and method sets are computed by a simple exhaustive search through the call graph. This is sufficient to judge the feasibility, effectiveness and scalability of the approach. Trivially, the efficiency of the last two abstractions would be strongly influenced because of the extensive logging of CCTs. For more efficiency, a realistic implementation would record only the information required for the chosen behavior characterization.

JP2 supports profiling multi-threaded applications, i.e., it manages the access and communication of different threads to the same CCT. In addition, it uses a modified version of the Java class `Thread`[1]. This modified version includes additional fields (e.g., `threadLocalCCTNode`) which help to get some runtime information about the status of the CCT being generated. These fields facilitate tracking the generated CCT profiles. Moreover, in order to be able to create CCTs, it is necessary to instrument Java classes which are going to be executed. For this, JP2 instruments Java code by adding new instructions that, e.g., calculate the number of execution times of each method, and set information about the caller and callee.

When starting profiling, JP2 creates a root node (called `callingContextTree`) which is considered as caller of all its children, i.e., callees. Figure 5.1 shows the CCT of the example program from Section 3.1.1 (cf. Figure 3.1) written in XML format, which is one of the available dump formats provided by JP2. Every node is identified by its class name and method name. In addition, the number of execution times of a method is also stored (cf. the XML tag `executionCount` in the example).

Model Creation

We implemented a profile management tool with a graphical user interface as shown in Figure 5.2. The tool creates a model by merging a selected number of profiles, which are assumed to represent the behavior of the outsourced application. More specific, the tags `callsite` and `mathod` are the most relevant ones when creating the model.

To merge, the tool has to traverse all selected profiles. Since the profiles may be of different sizes, we used two different APIs, the DOM and StAX, to read and write content of XMLs. The DOM interface is used for profiles whose size is quite small. Advantage of this technique is that one can navigate/read to any node of the loaded profiles, because data is available in the memory. On the contrary, we used the StAX interface for larger files. Using StAX does not lead to loading the whole profile into memory, i.e., nodes are read once StAX asks to continue to next event.

[1]http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html

```
<?xml version="1.0" encoding="UTF-8"?>
<callingContextTree xmlns="http://jp-profiler.origo.ethz.ch/xmlns/calling-context-tree">
+ <method declaringClass="Ljava/lang/reflect/AccessibleObject;" name="setAccessible" params="Z" return="V">

        ⋮

- <method declaringClass="LMain;" name="main" params="[Ljava/lang/String;" return="V">
    +   <executionCount>
    +   <executedInstructions>
    +   <callsite instruction="0">
    -   <callsite instruction="1">
        -   <method declaringClass="LMain;" name="foo" params="" return="V">
            +   <executionCount>
            +   <executedInstructions>
            -   <callsite instruction="1">
                -   <method declaringClass="LMain;" name="bar" params="" return="V">
                    - <executionCount>
                            1
                      </executionCount>
                    + <executedInstructions>
                    + <basicBlock startInstruction="1" endInstruction="1">
            + <basicBlock startInstruction="1" endInstruction="2">
    - <callsite instruction="5">
        - <method declaringClass="LMain;" name="bar" params="" return="V">
            - <executionCount>
                    10
                </executionCount>
            + <executedInstructions>
            + <basicBlock startInstruction="1" endInstruction="1">

        ⋮

    ⋮

</callingContextTree>
```

Figure 5.1: Example CCT

Furthermore, we used the tool Graphiz[2] and Cytoscape[3] to visualize the compliance check of the profiles for more usability in case of manually checking execution paths by humans. Figures 5.3a and 5.3b show example results of an executed compliance check. The red paths represent non-compliant paths, whereas blue paths represent compliant paths. The black paths represent paths which are subset of the model but not of profile.

Compliance Control

We implement the "$\models$" operator from Section 3.1.3 by simply checking whether the calling context tree, call graph or function set collected on the server is a sub-tree, sub-graph or sub-set of the respective application model. For the "$\equiv$" we define that $l \equiv l'$ if the respective trees or graphs are isomorphic, or in the case of function sets if they are equal. We store calling context trees and call graphs in a normalized fashion that allows us to decide $l \equiv l'$ in time linear in the size of the operands.

[2]http://www.graphviz.org/
[3]http://www.cytoscape.org/

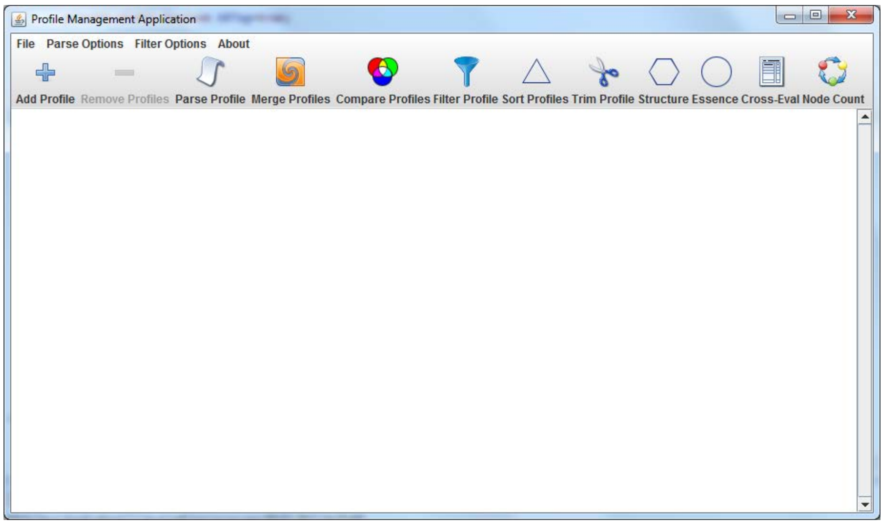


Figure 5.2: The profile management tool

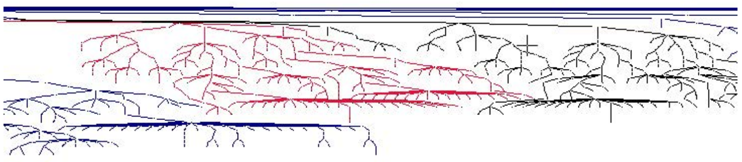
5.2 Building Chain of Trust and Runtime-Secure Storage

5.2.1 Building the Chain of Trust

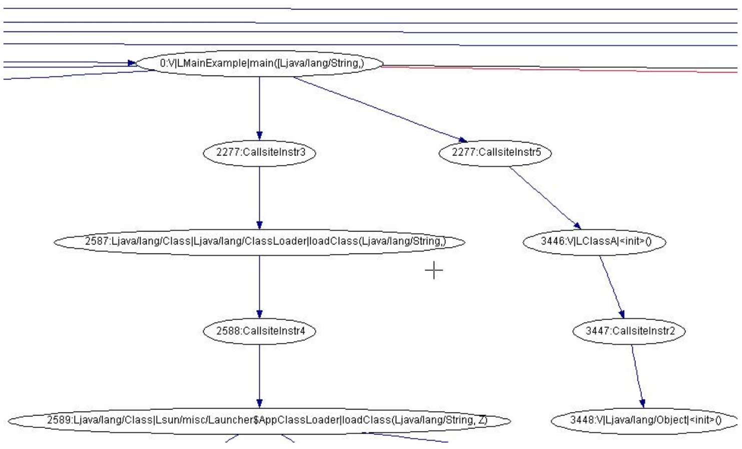
Our particular choice to instantiate the integrity measurement components and to build a chain of trust relies on concepts of Trusted Computing. Table 5.1 shows the exact settings of the selected software and hardware to build our implementation platform.

As specified by Trusted Computing, a root of trust must be implemented in hardware chips, called TPM. Our platform is therefore equipped with an Intel iTPM chip. During booting, SRTM (Static Root of Trust for Measurement) BIOS measurements are made using SHA-1 and the values are stored in the `binary_bios_measurements` file located in the directory `/sys/kernel/security/tpm0/`. Simultaneously, these values are extended into specific PCRs in the TPM. After measuring BIOS, the chain of trust will include a trusted boot loader alongside the regular boot loader. We used, for this, GRUB [1] together with its extension TrustedGRUB [4], which extends the chain of trust by measuring and extending the operating system, which is loaded through the bootloader. This enables the possibility to attest the booted system configuration and to verify whether it is indeed the intended system configuration and that it has not been manipulated or exchanged by malicious software or attackers (cf. pre-kernel measurements).

Afterwards, we used the Integrity Measurement Architecture (IMA) [84] as an extension for the Kernel to allow measuring integrity of different kinds of executables, such as libraries, scripts, running software, hardware drivers and Kernel mod-



(a) A bird's eye view of the compliance check


0:V|LMainExample|main([Ljava/lang/String,)
2277:CallsiteInstr3
2277:CallsiteInstr5
2587:Ljava/lang/Class|Ljava/lang/ClassLoader|loadClass(Ljava/lang/String,)
3446:V|LClassA|<init>()
2588:CallsiteInstr4
3447:CallsiteInstr2
2589:Ljava/lang/Class|Lsun/misc/Launcher$AppClassLoader|loadClass(Ljava/lang/String, Z)
3448:V|Ljava/lang/Object|<init>()

(b) A zoomed-in perspective of the compliance check

Figure 5.3: Architectures for passive and active compliance control

ules (post-boot measurements) on the platform. That is, IMA extends the chain of trust to include all executables running on the platform. To this end, it creates and updates a special SML, in which all measurements are stored. These measurements mirror – in case of no manipulations – a specific PCR value in the TPM, which is by default number 10. IMA creates the directory `/sys/kernel/security/ima/`, in which the file `binary_runtime_measurements` is included. All IMA measurements are stored in this file as list entries. The first measurement list entry is the `boot_aggregate` which consists of a SHA-1 hash of the contents of the first eight PCRs (0 . . . 7). An example of the SML created and updated by IMA is shown in Table 5.2, where measurements hooks are the places from which a measure call is issued. We refer to [86] for more information about the implementation details of IMA. In summary, the aforementioned implementation details guarantee creating a chain of trust from booting up to applications' *load-time*. In Section 5.2.2 we explain how we can provide an implementation which extends the chain of trust to include applications *runtime* information.

Entry #	SHA-1 Measurement	Measurement Hook	File Name	Type
000	EDF6DBA0609E5B3919E27F9EB6234259936BE6DA	ima-init	boot-aggregate	aggregate PCRs 0-7
001	3698483DE04931CB92E8681D7B9DE8DBFF90DA8A	mmap-file	init	executable
⋮	⋮	⋮	⋮	⋮
020	6D223FC8527ED9ED091177C93D8B85BDDE20646F	mmap-file	grep	executable
021	B4DAFEAB11FD01553844F800FB28C08E31FB1568	bash-script	cham_cal	bash command file
⋮	⋮	⋮	⋮	⋮

Table 5.2: An example of post-boot measurements made by IMA

5.2.2 Implementation of the vTPM Manager

Here, we explain how we can build a runtime-secure storage. As discussed before, we decided to use technologies from Trusted Computing, which we consider as basis for building such a storage.

In particular, we need to extend the chain of trust explained in the previous section to include not only hash values of the executable binaries, but also information coming from running midlleware or software at runtime. To this end, we used vTPMs, which are managed by an underlying vTPM manager. Remember that the use of one hardware TPM is not sufficient in scenarios where for each application instance a TPM instance is needed (see Sections 3.1.3 and 3.2.1).

We implemented a vTPM manager in Java as a singleton proxy to create and manage vTPM instances. That is, it offers functionalities which can be used by applications and attestation services which has demand for connecting to a TPM, querying its state and capabilities as well as for sending TPM commands and receiving the corresponding responses. In Table 5.3, we list some important commands which have been implemented within the vTPM manager.

Implementing vTPMs is based on the software-based TPM emulator (version 7.1) proposed by Strasser and Stamer [94] which is a powerful testing and debugging tool that can also be used for educational purposes. It is composed of three main parts: a user-space daemon (tpmd) that implements the actual TPM emulator, a TPM device driver library (tddl) as the regular interface to access the emulator, and a kernel module (tpmd_dev) that provides the character device `/dev/tpm` for low-level compatibility with TPM device drivers [8]. The emulator offers the possibility for command-line configuration using input parameters. Our vTPM manager uses this feature to create instances of vTPMs. To this end, the manager uses the method `Runtime.getRuntime().exec(cmd)`, which returns an instance of the class `java.lang.Process`. The parameter `cmd` contains the given command which has to be performed, e.g., `/usr/local/bin/tpmd -u /tmp/tpmd/socket:123`.

Command	Description
`vTPM_Instance_Create`	Starts an instance of the TPM emulator and creates UDS connection to this instance. It also initializes the vTPM instance by executing the `TPM_TakeOwnership` command, which is important when performing the remote attestation process in a later step
`vTPM_Instance_Extend`	Given a vTPM instance ID, the manager executes the same command of the vTPM instance having the given ID
`vTPM_Instance_Destroy`	Given a vTPM instance ID, the manager destroys an afore created vTPM instance having the given ID, stops its corresponding process, and closes its UDS connection
`vTPM_Instance_Quote`	Given a vTPM instance ID, the manager executes and `TPM_Quote` command of the vTPM instance having the given ID, which quotes the data structure `TPM_Quote_Info` (see [99] for more details about data structures in TPMs)
`vTPM_Instance_PCRRead`	Given a vTPM instance ID, the manager provides non-cryptographic reporting over the content of a named PCR of a vTPM instance having the given ID

Table 5.3: Examples from the vTPM manager commands list

In order for the manager to communicate with vTPMs, we decided to use Unix Domain Socket (UDS), which is an endpoint used for bidirectional inter-process communication to exchange data between processes in the same host. That is, the parameter `u` in the last command-line is needed to define the socket required for exchanging data between the manager and the vTPM. This way, it is possible to create a communication channel for every vTPM instance, which guarantees for the manager to select the correct vTPM instance to communicate with. Since our manager is implemented in Java, we used the library Java Unix Domain Socket (JUDS, version 0.93) [4].

We used a modified version of tpm4java [2] as a TSS, which is a software specification that provides a standard API for accessing the functions of the TPM. tpm4java is developed for establishing connection to only one TPM. Since we need to establish connections to different vTPMs, we conducted some code modifica-

[4]http://code.google.com/p/juds/

tions to tpm4java. To this end, we removed the singleton implementation of it. Since communication with the TPM is typically handled by the TCG device driver library (TDDL), and its interface is defined by the TSS (TCG Software Stack) specification, we wrote a new constructor which allows taking the type of connection as a parameter. In addition, we implemented a new connection type (i.e., TDDL), which we called UDS-TDDL. Conequently, this new type allowed communication with vTPMs through sockets. This way, the manager can manage an arbitrary number of vTPMs.

5.3 Adoption of the High-Level BCC to Apache ODE

In this section we describe how we implemented our approach proposed in Section 3.2. We first introduce the setup of the platform and then describe the implementation of each component of the architecture.

5.3.1 Platform Architecture Setup

Since our proposed architecture from Section 3.2.1 relies on technologies from Trusted Computing, the construction of $\mathcal{H}$ needs to guarantee generating a chain of trust. This is done using the configurations setting 1 in Table 5.1 and the implementation described in Section 5.2.1.

Since $\mathcal{H}$ relies on technologies from Cloud Computing and therefore on virtualization technologies, the kernel of the platform is equipped with Xen 4.0.1 extension and serves as main operating system. This kernel is a Xen dom0 kernel and supports starting hardware virtual guest machines (HVM) on the same computer with the help of the Xen hypervisor. As domU guest domain we used a standard Debian 6.0 linux (squeeze) with kernel 2.6.38.2. In addition, we use IMA to allow measuring kernel modules and software running on all guest domains.

As a BP-Engine we selected the Apache ODE (Orchestration Director Engine)[5] (version 1.3.4), which is a Java-based web application that executes business processes written in Business Process Execution Language (BPEL) [7]. The engine is responsible to communicate with services, sending and receiving messages, handling data manipulation and error recovery as described by the BPEL definition. That is, the engine is capable of performing inner-computation (i.e., internally performing data manipulation functions) and outer-computation (i.e., through calling external services). The ODE engine runs on an apache tomcat[6] servlet container in version 6.0.32. We assume that each tenant has its own BP-Engine, which runs on a separate domU domain.

[5]http://ode.apache.org/
[6]http://tomcat.apache.org/

To extend the chain of trust to include the ODE application, we implemented an extension to the apache tomcat which measures all loaded Java classes of the ODE engine using the SHA-1 hash. More specifically, we extended the `WebAppClassLoader` of the package `org.apache.catalina.loader` in tomcat, which is responsible for loading Java classes of deployed web applications. All measured classes extend a specific PCR register in the vTPM associated to the guest domain.

The proposed architecture suggests the use of two different vTPM managers. The first one is used to offer TPM functionalities for all measurements related to guest domains up to the BP-Engine, which we call in the architecture domain vTPM manager. The second vTPM manager is responsible for offering TPM functionalities for measurements taken within then BP-Engine. As a domain vTPM manager we used the approach described by [28], whereas we used the implementation described in Section 5.2.2 for the BP vTPM manager. Both vTPM managers run in dom0.

To assure that each tenant can access only the vTPMs associated to his own business processes, we make these vTPMs also domain-specific by associating the domain id to the BP vTPM upon creation. To specify the domain id of requests coming to the BP vTPM, we use the vTPM manager as a proxy, i.e., all requests to create and destroy BP vTPMs, as well as extending PCR values, come from the domain vTPM manager. This is important to fulfill R_3 (see Section 3.2.1 for more details). Since the virtualized domains are independent, we used for the communication between the FAE and the vTPM manager the Remote Method Invocation (RMI) interface.

In addition, all SML types are represented as XML documents without storing them on the hard disk. This mitigates the overhead resulting from the frequent logging. A manager of these logs is also implemented in Java.

5.3.2 Implementation of the Flow Attestation Extension

The FAE is implemented as a plugin for Apache ODE and extends it `BpelEventListener`, which implements the well-known *observer pattern*. ODE provides several types of execution events, such as process creation and deletion as well as activity start and end events. That is, whenever such an event occurs the FAE is informed about it and can react accordingly.

The implementation is designed in a way such that every business process instance is associated to vTPM instance. In addition, we implemented an SML manager in Java to create and manage the SMLs of each PI, which are stored in XML format.

In detail, the FAE uses the event's type `instanceLifecycle` to inform the BP vTPM manager about creation and termination of processes (by calling `vTPM_Instance_Create`). Depending on the sent command the manager, in turn, creates or destroys the corresponding vTPM instances where for simplicity the PI_{id} is used as a vTPM instance ID. At the same time, a corresponding $SML_{PI_{id}}$ is created by the log manager.

```
<?xml version="1.0" encoding="UTF-8"?>
<TrustedOdeLog InstanceID="251" ProcessID="LoanRequestService-1">

        ⋮

    <ActivityExecStartEvent activityName="creditWorthinessCheckStandard" activityType="Invoke" />

    <VariableModificationEvent varName="LoanServicesPLResponse">
      <message>
        <parameters>
           <checkCreditWorthinessResponse>
             <return>true</return>
           </checkCreditWorthinessResponse>
          </parameters>
        </message>
    </VariableModificationEvent>

    <ActivityExecEndEvent activityName="creditWorthinessCheckStandard" activityType="Invoke" />

        ⋮

</TrustedOdeLog>
```

Figure 5.4: Example SML

Another event's type we used is `activityLifecycle`, which gives information about starting, execution's termination and failing of activities. The FAE uses this type, e.g., to extend the vTPM, which corresponds to PI, with the help of the command `vTPM_Instance_Extend` offered by the BP vTPM manager. To this end, FAE retrieves the BPEL description of the activity to be extended, hashes it and finally sends the hash value to the vTPM manager. A corresponding SML entry is also added to $SML_{PI_{id}}$.

The approach also specifies recording data handling, since it is required – in a later step – to recreate the flows f_{actual} and f_{target}. To do this, we make use of the event's type `dataHandling`, which informs about any data modification occurs to any variable during execution. The data modifications are also reported to the corresponding vTPM and an SML entry is added to $SML_{PI_{id}}$.

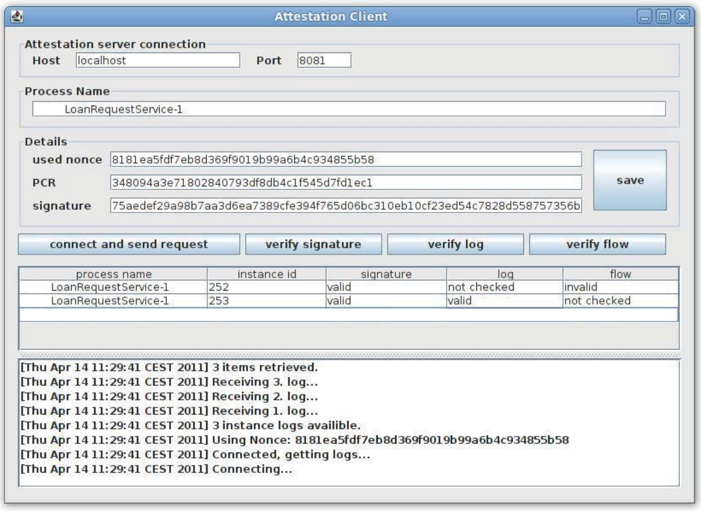
Figure 5.4 shows an digest from $SML_{PI_{id}}$ of the *loanRequestService* example presented in Section 2.2. Note that all measurements are reported to the vTPM manager using SSH tunneling for Java RMI[7].

5.3.3 Implementation of the Attestation Services

The attestation service module works as a proxy to transmitt requests of attestation from the client $\mathcal{C}$ to the hosting platform $\mathcal{H}$. The former is called attestation client and the latter is called attestation server.

For the attestation client we developed a user interface, which allows starting an attestation process for selected business processes. Figure 5.5 shows a screen

[7]http://www.javaranch.com/journal/2003/10/rmi-ssh_p1.html



Attestation server connection

Host: localhost Port: 8081

Process Name: LoanRequestService-1

Details

used nonce: 8181ea5fdf7eb8d369f9019b99a6b4c934855b58

PCR: 348094a3e71802840793df8db4c1f545d7fd1ec1

signature: 75aedef29a98b7aa3d6ea7389cfe394f765d06bc310eb10cf23ed54c7828d558757356b

process name	instance id	signature	log	flow
LoanRequestService-1	252	valid	not checked	invalid
LoanRequestService-1	253	valid	valid	not checked

```
[Thu Apr 14 11:29:41 CEST 2011] 3 items retrieved.
[Thu Apr 14 11:29:41 CEST 2011] Receiving 3. log...
[Thu Apr 14 11:29:41 CEST 2011] Receiving 2. log...
[Thu Apr 14 11:29:41 CEST 2011] Receiving 1. log...
[Thu Apr 14 11:29:41 CEST 2011] 3 instance logs availible.
[Thu Apr 14 11:29:41 CEST 2011] Using Nonce: 8181ea5fdf7eb8d369f9019b99a6b4c934855b58
[Thu Apr 14 11:29:41 CEST 2011] Connected, getting logs...
[Thu Apr 14 11:29:41 CEST 2011] Connecting...
```

Figure 5.5: Attestation client's user interface

shot of the interface depicting the elements used in the attestation process, e.g., the unique identifier of a business process and *nonce*.

The attestation server is implemented using multi-threading in Java, so that multiple attestation processes can be done simultaneously. After receiving the attestation request form the client, the attestation server retrieves all $SML_{PI_{id}}$ which belong to the business process sent in the request. This is done with the help of the log manager described previously. In addition, the attestation service uses the command `vTPM_Instance_Quote` of the vTPM manager to quote the requested PCRs from all vTPMs which correspond to the found PIs. Finally, the attestation service sends back the response to the client as shown in Table 5.4, where "*" signals a variable size and is defined directly before. The row numbers 2-8 are repeated as much as the number of fetched instances.

After receiving the response from the attestation server, the attestation client reconstructs the data structures `TPM_Quote_Info` and `TPM_Composite_Hash` as specified by [99]. This is necessary to allow the attestation client to verify the nonce, the signatures and all SMLs ($SML_{PI_1} \ldots SML_{PI_n}$) as described in Section 3.2.1. As shown in Figure 5.5, the client has the possibility to verify the flow of any SML in list as described in Section 5.3.4.

5.3.4 Implementation of the Flow Verification Extension

As described in 3.2.1, we use an actual/target comparison to decide on the correctness of a business process' flow. Data stored in $SML_{PI_{id}}$ are transformed, on

	Size (byte)	Format	Description
1	4	integer	number of fetched instances
2	20	binary	PCR's value
3	4	integer	SML's size
4	*	xml	SMLs
5	4	integer	signature's size
6	*	binary	signature
7	4	integer	size of the public key part of AIK
8	*	binary	public key part of AIK

Table 5.4: Attestation-server's response to the client

the one hand, directly into f_{actual}, and, on the other hand, indirectly into f_{target}. More specifically, we use data stored as `VariableModificationEvent` as shown in Figure 5.4 to simulate a BPEL process.

In addition, a transformation function $\mathcal{T}$ is used to avoid calling external services A_{ext}. Our implementation of $\mathcal{T}$ is done using a BPEL-to-BPEL transformation. For a given PI, the FVE transforms every BPEL activity of type `invoke` into an activity of type `assign`. The values used for the `assign` activity are taken from the corresponding $SML_{PI_{id}}$. By doing so, the execution of the new resulting process BP′ will not result in calling external activities, and allows local simulation.

Our implementation of $\mathcal{T}$ considers loops and conditional statements blocks (e.g., if statements), and transforms `invokes` within these blocks according to number of iterations and branches taken during the execution of a PI. In case of iterations, the implementation of $\mathcal{T}$ ignores unnecessary `assign` transformations.

After finishing the *transformation* process, which results in, first, a new business process definition BP′ (i.e., f_{target}) that contains only internal activities, and, second, the f_{actual}, FVE starts the *simulation* process by executing BP′ on the BP-Engine of $\mathcal{C}$. FVE is implemented as an extension of Apache ODE. During execution, the *flow verifier* checks the steps of the business process (i.e., the steps of f_{target}). It compares every activity of f_{target} with the one that was actually executed in f_{actual}. If both are equal, this step is verified valid. For those parts of BP, which behave non-deterministic (such as `flow` blocks in BPEL), the flow verifier accepts any of the flows within the block.

5.3.5 Performance Evaluation

This section presents some performance tests of our implementation, see Table 5.5. All measurements are performed on the software and hardware configurations listed in setting 1 of Table 5.1.

The tests number 5 and 6 of Table 5.5 are performed for the credit risk assessment

#	What is tested?	# Experiments	Time average
1	Creating a vTPM instance	100	1.01 seconds
2	TPM extending	10000	400 μs
3	SHA-1 hashing (512 bytes)	100000	13 μs
4	SHA-1 hashing (1 kb)	100000	16 μs
5	BP completion with TPM	100	1.07 seconds
6	BP completion without TPM	100	0.121 seconds

Table 5.5: The performance analysis of our implementation

business process we introduced in Section 2.2. The SML_{PI} of this process includes 13 activity start and 13 activity end log entries, as well as 9 variable modifications.

As shown in the table, the most expensive operation is the vTPM instance creation. However, this operation is done only once for every PI. The most frequent operation is number 2 in the table, which takes in average 400 μs for every activity start and finish, as well as every data modification.

The execution of business processes usually includes calls to external services over a communication network; these calls cause a significant performance penalty. Thus, the small overhead resulting after applying our approach (as shown in Table 5.5) will not significantly influence the execution of such a process, when compared to usual network latencies. This shows that our approach is indeed applicable in practice.

5.4 Implementation of Group-Based Attestation

In this section we describe the changes we made to the Linux system for the implementation of both variants of Chameleon Attestation as proposed in Sections 4.2.2 and 4.2.4. Out implementation is based on the software and hardware configurations listed in setting 2, Table 5.1. Building a chain of trust as required in our approaches is done using the implementation described in Section 5.2.1.

Since we used as a TSS the Java based jTSS, it was necessary to make some modifications on it, because jTSS supports only one measurement log, namely only for pre-kernel measurements. To this end, we modified it to also support reading the measurement log created by IMA (i.e., post-boot measurements). For the remote attestation process, we implemented a Java based server and client. jTSS is used by the server to access the functions of the TPM such as reading PCR registers, signing PCR content, etc. The client also uses the functionalities provided by jTSS to verify signatures and recompute PCR contents. In addition, a MySQL database management system was used on the client side to store the Reference Measurement List (RML). The remote attestation process is performed from both, the client and the server, as specified in Sections 4.2.2 and 4.2.4.

For both approaches, it is assumed that a trusted party is responsible for creating attestation groups. We implemented the algorithm **Forge** (cf. Section 2.3.1) in order to create collisions for group members with the help of the private key part **sk**, using some functionalities provided by the openSSL library[8]. In addition, the implementation used the same library to generate all parameters needed to calculate the chameleon hash value (i.e., executing the **CH** function).

5.4.1 Implementation of Chameleon Attestation I

For the first variant described in Section 4.2.2, it is necessary to calculate our chosen chameleon hash function described in [20], denoted as **CH**, on the attested platform. For that reason, we implemented an extension for IMA such that the **CH** value is calculated after measuring every executable. We assume that the parameters required to calculate **CH** are delivered with the executable and are accessible by our extension. Note that to allow accessing these parameters for first measurements performed by IMA, we added the file containing these parameters to `initrd`, which contains all files that have to be executed by the kernel during the boot process.

We first created a special measurement list SML_{CH} which contains the chameleon hashes of measured executables. These measurements should mirror – in case of no manipulations – the value of PCR number 11. To this end, we added the option `IMA_MEASURE_PCR_IDX_CH` in the configuration file `/security/ima/Kconfig`. The implementation of SML_{CH} is done using two files stored in same directory as the original SML created by IMA. They also have the same names plus the postfix "`_ch`". SML_{CH} contains – as in the original SML of IMA – the SHA-1 hash value of chameleon hashes, the PCR index and the name of the measured executable. Remember that, in our approach, chameleon hashes are hashed again using SHA-1 to produce a 160-bit long message that is needed for extension to a particular PCR. We also modified the standard SML to store the public **CH** parameters J, r, e and N. In particular, in order to store these parameters we extended the struct `ima_measure_entry`.

Afterwards, to read these parameters again from SML, we implemented a new function in the file `/security/ima/ima_main.c`, which is called from the following functions: `ima_do_measure_file` and `ima_do_measure_memory`. To calculate the **CH** value, we created a new function in the file `/ima/ima_main.c`, which also stores the resulting **CH** value in SML_{CH} and the SHA-1 value in standard SML. Note that the standard ML is used only for internal purposes, whereas the SML_{CH} is sent to the challenger during the attestation process. For the implementation of **CH** we used a slightly changed version of the RSA patch for avr32linux, such

[8]http://www.openssl.org

that all functionalities needed by the patch can be called from `/ima/ima_main.c`.

A modification to the function `ima_extend` was also necessary, such that it expects another parameter containing the chameleon hash value, in order to extend the PCR specified for storing the chain of chameleon hashes.

Group signature based attestation. The implementation of the group signature based attestation proposed in Section 4.2.3 can be achieved in a similar way as the Chameleon Attestation I. The verification of a signature must be placed at the position where the calculation of chameleon hashes is performed in Chameleon Attestation I. In addition, the public keys of software vendors have to be stored in the struct `ima_measure_entry` instead of chameleon hash values.

5.4.2 Implementation of Chameleon Attestation II

In the second variant described in Section 4.2.4 we need to calculate the chameleon hash on the platform of the challenger. We thus modify the measurement process in a way that the parameters J, r, e and N are added to SML, as in Chameleon Attestation I. Furthermore, we extended the package `iaik.tc.tss.impl.java.tcs.evenmgr` of jTSS such that the new chameleon hash parameters can be read from SML in addition to SHA-1 values. To calculate the chameleon hash on the challenger side, we modified the server such that the SHA-1 values and the corresponding new parameters can be delivered to the challenger. We implemented the RSA based chameleon hash function using OpenSSL on the side of the challenger to enable it to calculate the hash value and verify it against the RML.

5.4.3 Experimental Results

To make this approach measurable, we conducted some experiments to see the impact of our approach to the problems of integrity measurement and remote attestation discussed in Section 4.2.1. The experiments show that Chameleon Attestation significantly reduces the number of the reference measurements required to decide the trustworthiness of the attested system. Subsequently, we discuss the performance of our approach.

Maintainability

As mentioned before, keeping millions of RML entries up-to-date is a big challenge when using the classical TCG's remote attestation. A solution for this problem should answer the following research question: to what extent do the number of RML entries and its growth ratio influence the management, communication and maintenance effort of fetching RML entries and keeping them up-to-date?

Packages	Measure-ment	Fresh installation	Update	Total	Statistics	
					% Fresh installation	% Update
kernel	TCG	1,820	1,816	3,636	50.1 %	49.9 %
	CA	1	0	1	100.0 %	0 %
samba-common	TCG	18	15	33	54.5 %	45.5 %
	CA	1	0	1	100.0 %	0.0 %
samba	TCG	24	26	50	48.0 %	52.0 %
	CA	1	0	1	100.0 %	0.0 %
httpd (Apache)	TCG	71	72	143	49.7 %	50.3 %
	CA	1	0	1	100.0 %	0.0 %
⋮	⋮	⋮	⋮	⋮	⋮	⋮
All	TCG	8,268	5,448	13,716	60.3 %	39.7 %
	CA	981	37	1.018	96.3 %	3.7 %
	ratio	8.5:1	147:1	13.5:1		10.7:1

Table 5.6: Reduction of measurements in RML

To give an answer to this question, we first created an RML by measuring a fresh installation of Fedora 10 (kernel version 2.4.27.5), but neglecting the content of two folders: the folder `/var/` which contains variable data that can be changed or deleted at runtime, and the folder `/usr/share/` which contains the architecture-independent data. Since it is difficult in retrospect to group packages by manufacturer (because the package manager of Fedora does not store information about the author/manufacturer of a package), we grouped software products by packages and assigned each file in a package its appropriate random r. Table 5.6 shows that we need 8,268 different entries in RML for the fresh installation when we employ classic TCG attestation (one for each file). In the contrast, we need to store merely 981 measurements in the RML by applying our approach (one for each package in case of grouping by packages). Moreover, grouping measurements by software manufacturer decreases the number of entries in RML sharply to reach the number of trusted manufacturer. This shows how our approach can be used to reduce the number of entries in the RML. Consequently, on the one hand, there will be no need for large storage capacities for RMLS, and on the other hand, fetching an entry out of all these entries will be much easier.

To test the management overhead when updating packages, we performed another experiment by updating the Linux distribution and its installed packages to newer versions. For instance, the kernel is updated from version 2.6.27.5 to 2.6.27.41, the package samba-common from 3.2.4 to 3.2.15, the package samba from 3.2.4 to

3.2.15, and the package httpd from 2.2.10 to 2.2.14. Table 5.6 shows that in case of using the classic TCG attestation 1,816 new SHA-1 measurements (49.9 % of the total measurements for the kernel) have to be distributed and published in RMLs. Conversely, by employing Chameleon Attestation no new measurements have to be distributed or published. For the overall distribution and its installed packages, we need to update only 37 chameleon hashes rather than 5,448. These hashes mainly account for newly added packages.

The previous numbers hold only for one updating process, neglecting at the same time all in-between updates, which must have been performed. If we have a look at the kernel versions, we find that the update was from version 2.6.27.5 to 2.6.27.41, which means that there were lots of versions missing and consequently lots of updating required. Thus, the management and communication effort is significantly reduced.

Another issue which increases the number of possible versions of one measurement is considering numerous Linux distributions. Normally, Linux distributors compile program packages themselves, which leads to the fact that there will exist many SHA-1 measurements of the same file (one for each distributor). This is a real problem since the RML size, the management and communication overhead will be multiplied by a factor equaling the number of distributors. However, employing our approach will absolutely solve this problem, since the chameleon hash of files located in different distributions stays the same. Figure 5.6 shows the significant impact difference between applying SHA-1 and chameleon hashes in RMLs, when updating all packages of ten different distributions, assuming that each distribution contains the same packages, each of which contains the same number of files. From the figure, we can see that the number of RML entries increases significantly by increasing the number of distributors when using SHA-1 hashes, while the same number stays almost constant when using chameleon hashes.

Privacy

The configuration privacy of the attested platform is substantially enhanced by the use of Chameleon Attestation I: the challenger can decide on the trustworthiness of the attested platform without knowing the exact details of the configuration.

However, since there is a tradeoff between privacy and control precision, the scheme can be applied on different granularities: depending on the choice of the manufacturer, software groups may encompass different versions of individual files, packages, software systems or even software of a specific vendor (see Figure 5.7). The higher the level, the more privacy can be protected; on the downside, less information on the platform is available, i.e., the control precision is lower. Our approach can be easily combined with other identity privacy approaches, such as a Privacy CA and DAA.

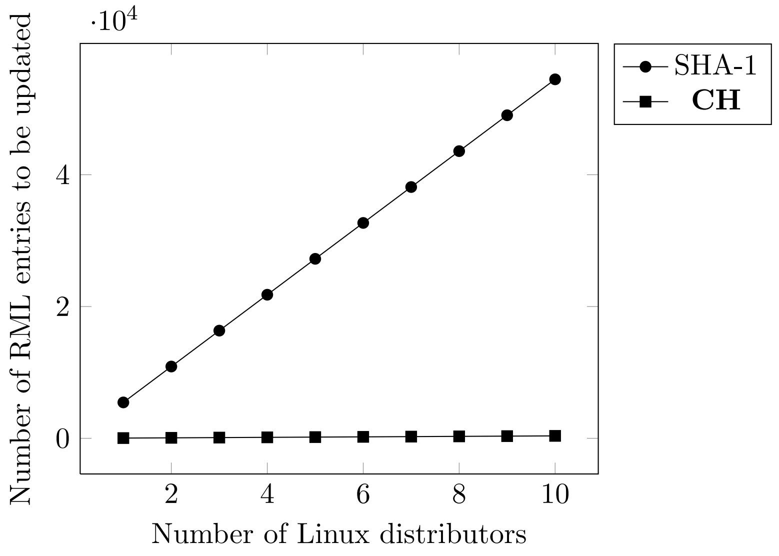


Figure 5.6: The impact of distributions' number on maintainability of RMLs

A potential problem by using levels of privacy granularities lies in the inability of the challenger to clearly identify loading sequences of security components. For instance, if the grouping is done on the level of software manufacturers, so it will be not possible for a challenger to know whether, e.g., an anti-virus program is loaded before a firewall program, assuming they both belong to the same manufacturer. A possible solution for this problem is by using dynamic levels of granularity, i.e., for specific security-critical components a lower level of granularity is chosen. Another solution is the complete renunciation of configuration privacy by the use of Chameleon Attestation II, in which grouping is done by the challenger himself.

Another problem associated with TCG's remote attestation is discrimination. Using chameleon attestation may mitigate this problem by building a much higher level of granularity, namely by using groups of software consortium. However, we believe this is theoretically feasible but practically difficult.

Sealing

In a similar manner, the sealing problem can be avoided, since different versions of the a software will have the same chameleon hash value, i.e., data can be bound to this value without risking data unavailability when updating to the next version.

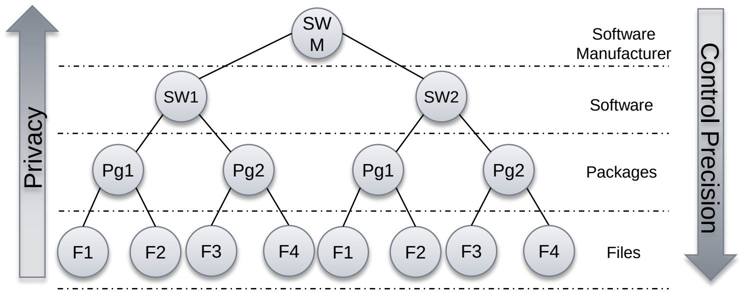


Figure 5.7: Granularity levels of privacy and control precision

Performance evaluation

Public key cryptography is often associated with performance overhead. Since we use such public key schemes in the integrity measurement, we evaluated the performance of Chameleon Attestation by measuring the timing difference compared to the standard TCG measurement process. Our experiments were performed on the platform whose hardware and software configurations listed in setting 2, Table 5.1.

We used bootchart[9] to determine the boot time of a standard kernel, a kernel with IMA, and a kernel with **CH**. While a standard kernel takes 30s to finish booting, a kernel with IMA takes 33s and a kernel with **CH** takes 44s. However, booting is a process which does not take place regularly, especially for server systems. More interesting is the additional required time for every executable content, since they give more insight into the performance.

The calculation of **CH** in Chameleon Attestation I (cf. Section 4.2.2) is performed in the kernel space and requires 4,674 μs, while the calculation of **CH** in Chameleon Attestation II (cf. Section 4.2.4) is done in the user space and requires 896 μs, i.e., the fifth of the time needed for the first variant. Note that these measurements are taken for a 160-bit input, which is the SHA-1 value of a measured file. The calculation of collisions takes 899 μs in the user space. All measurements were taken using the function `gettimeofday` in both the kernel space and the user space. We neglected the overhead of executing this function which was less than 1 μs. Note that all measurements we present here aim at giving a gross overview on the overhead of applying public-key schemes – instead of SHA-1 only – in the integrity measurement process. We expect that significant performance improvements can be obtained using highly optimized code also in kernel space.

We conducted some experiments to evaluate the overhead resulting from applying Chameleon Attestation. The results are shown in Table 5.7 and depict that measuring a 2 byte file using SHA-1 takes 1.4 μs, while measuring the same

[9]http://www.bootchart.org

file and subsequently extending the TPM PCRs (SHA-1 + extend) takes approximately 9,972 μs. However, since these measurements are already a few years old, the values should now be smaller. Table 5.7 also illustrates the performance of **CH**

Measurement method	2 byte	1 KB	1 MB
SHA-1	1.4 μs	20 μs	18,312 μs
SHA-1 + **CH**	4,675 μs	4,694 μs	22,986 μs
CH fraction	99.8 %	99.6 %	20.3 %
SHA-1 + extend	9,972 μs	9,989 μs	28,281 μs
SHA-1 + **CH** + extend	14,646 μs	14,663 μs	32,955 μs
CH fraction	31.9 %	31.9 %	14.2 %

Table 5.7: Performance of **CH** depending on SHA-1 and different file sizes

in the measurement process. The performance values of **CH** given in the table are the average of 1000 measurements. Obviously, the size of the measured files influences the required time significantly. For instance, the calculation of SHA-1 of a 1 KB file takes approx. 20.1 μs, while measuring a 1 MB file takes 18,312.3 μs $\approx$ 18.3 ms. Note that the time required to compute **CH** is almost constant, as it is only applied to a SHA-1 value. Table 5.7 also gives timing measurements for the whole process of computing the SHA-1 and chameleon hashes and extending the PCR register with the newly created hashes. The measurements show that for a file of 1 MB 14.2% of the total time required to extend a particular PCR is taken for computing the **CH** value. This percentage falls further when larger files are executed. The table also shows clearly that the most time is taken for extending, i.e., communicating with the TPM.

Thus, we believe that Chameleon Attestation is still applicable and its overhead is reasonable compared to the communication overhead resulting from requests and responses sent and received to/from the TPM.

6 A Security Evaluation for Low-Level BCC Based on Real Applications

To be able to judge the low-level approach proposed in Section 3.1, we provide, in this chapter, a detailed security evaluation using practical applications.

We evaluated hereby over four applications performing different attacks on them. The goal was to measure to what extent can we detect these attacks and with which costs. To this end, we set the following four main research questions:

RQ1 (Feasibility): In the learning phase, do the collected profiles converge to a stable model with low false-positive rates?

RQ2 (Effectiveness): To what extent is the application model able to discriminate between legal and illegal behavior?

RQ3 (Scalability): Is the profile size independent of the application's runtime?

RQ4 (Efficiency): Can our approach be implemented efficient enough to induce a sufficiently low runtime overhead?

6.1 Malicious-Input Attack on Document Manipulation Applications

Malicious inputs represent one of the serious threats facing nowadays applications, especially online application which has public interfaces and can be fed with data from anybody. Attacks using malicious inputs try to tamper with specific parts of an application in order to bypass the application's security mechanisms. Common attacks which use malicious inputs are command insertion, cross site scripting and buffer overflows. In the following, we evaluate such attacks using three Java applications.

6.1.1 General Experimental Setup

In order to evaluate the above attack, we had to opt for applications for which we would be able to obtain large sets of abnormal/malicious as well as legal/harmless inputs. We chose the following subjects:

1. *Apache pdfbox:* A PDF manipulation framework [18].
2. *POI-HSLF:* A Java API to extract data from PowerPoint documents [19].
3. *POI-HWPF:* A Java API to extract data from Word documents [19].

All applications operate on popular file types (Adobe PDF, Microsoft PowerPoint `.ppt`, and Microsoft Word `.doc`), all of which can be obtained in large numbers from the web. Moreover, all three file types are well-known attack vectors. For the PDF file type there further exist repositories of malicious inputs, which serve us to simulate possible manipulations by the cloud provider (details below).

6.1.2 RQ1: Feasibility

For behavior compliance control to be feasible, it must be possible to automatically generate a useful application model from only a number of representative inputs small enough not to be prohibitive. Moreover, the generated application models must yield false positive rates low enough for the approach to pay off. Remember that any false positive induces increased computation cost in, e.g., a private cloud, in case of re-performing the application in a fully trusted environment.

For our evaluation, we used the top 1,000 results of a Google search for `filetype:pdf`, `filetype:ppt`, and `filetype:doc`, respectively. The resulting corpus of inputs allowed us to generate application models from various numbers of input documents. We believe those 3,000 documents to be harmless, legal documents.[1] Therefore, if a model classified any run as abnormal that was induced by one of those inputs, we count this classification as a false positive.

To generate the application models, we first used the JP2 profiler (cf. Section 3.1.2) to obtain a calling context tree for each of the applications and inputs. From the resulting CCTs, we then derived both dynamic call-graph and function-set profiles. This ensures that, for a given input document, all three abstractions of the application's behavior are consistent.

We then used ten-fold cross-validation [66] to determine the false-positive rate that can be expected of the collected models. For each of the three file types, the 1,000 profiles were first divided into ten subsets of 100 profiles each. Then, each profile from one of the subsets was checked for compliance with application models derived from an increasing number of (randomly chosen) profiles in the other nine subsets, up to all 900 profiles in the end. Every compliance check yields either the answer "compliant" or an anomaly warning. Since we consider our training set

[1]This is because Google has put in place filters to remove invalid or potentially malicious documents from its search index. In fact, we tried to find malicious documents using Google but failed.

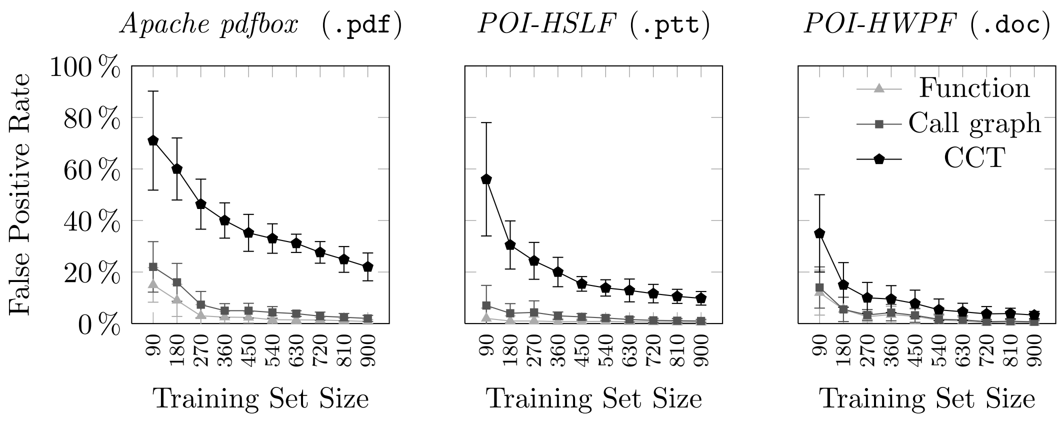


Figure 6.1: False positive rate for differently-sized training sets (arithmetic mean ± standard deviation of 10 training sets each).

to only contain compliant input documents, we consider all warnings to be false positives.

Figure 6.1 shows the resulting false positive rates, averaged over the 10 subsets, for various training set sizes. Because we used ten-fold cross-validation, at most 900 out of the 1000 available inputs were used for model generation. As Figure 6.1 shows, for both the Function and Call Graph abstractions it suffices to use only a few hundred inputs for model generation to obtain a model with a false-positive rates below 5%. Using the calling-context-tree (CCT) abstraction, however, requires a larger number of inputs to achieve low false-positive rates. Even when using 900 inputs to generate the application model, an average of about 22%, 10%, and 3%, respectively, of the remaining 100 profiles are deemed non-compliant. We also observe that at least for the Calling Context Tree abstraction the false-positive rates very much depend on the program under evaluation.

6.1.3 RQ2: Effectiveness

To increase trustworthiness, behavior compliance control must be able to detect abnormal execution behavior. For the purpose of our evaluation we consider an execution to be abnormal if it executes on an abnormal program input. In reality, there could be other sources of abnormality such as glitches in the hardware or execution environment. We obtained abnormal inputs from two distinct sources: from dedicated repositories of malicious inputs for the file types in question and from applying fuzzing techniques to legal inputs.

To simulate a targeted attack by a third party, we have used a set of 118 PDFs

	Exploits	Fuzzed		
		Apache pdfbox (.pdf)	*POI-HSLF* (.ppt)	*POI-HWPF* (.doc)
Functions	11 %	83 %	100 %	100 %
Call graphs	34 %	89 %	100 %	100 %
CCTs	100 %	97 %	100 %	100 %

Table 6.1: Percentage of inputs (exploits or fuzzed) detected as illegal.

that have previously been used in exploits.[2] For this experiment, we used application models computed by including all 1,000 profiles for PDF file type. As Table 6.1 shows, all abnormal executions were classified correctly when using the Calling Context Tree abstraction. When using the more coarse-grained Call Graph and Function abstractions, however, only 34 % respectively 11 % of inputs were classified correctly. We therefore conclude that it is essential to use information-rich profiles to detect targeted attacks reliably. This is the main trade-off at the heart of this paper: increased trust requires an increase investment to counter-balance the increased rate of false positives caused by such information-rich profiles.

As we were unable to obtain a similarly large number of malicious PowerPoint and Word documents to simulate a targeted attack, we commenced on a best-effort basis and resorted to fuzzing techniques to simulate an untargeted attack or a problem caused by a faulty data transmission. For each file type, we randomly picked 100 documents from of our corpus of legal documents and applied simple fuzzing techniques to them.[3] This process yields 100 documents each which we define to be abnormal inputs. For each of these inputs we then ran the corresponding application and compared its behavior, abstracted as Functions, Call Graph, or Calling Context Tree, with the application model of legal inputs used before.

Table 6.1 shows the percentage of fuzzed inputs that were successfully detected as illegal. As these results show, false negatives created by this simple fuzzing algorithm are easy to recognize. It follows that abnormal program runs induced by inputs corrupted in this manner will most likely be detected using behavior compliance control; the abstraction chosen (Functions, Call Graph, Calling Context Tree) has little influence on the detection rate. Those observations hold for the particular fuzzing approach we consider. More targeted fuzzing approaches, taking

[2] Test data taken from http://contagiodump.blogspot.com/2010/08/malicious-documents-archive-for.html (Collection 3).

[3] 10 random single-byte changes beyond the first 1024 bytes of data; the latter avoids corrupting the main document header, a case that is particularly easy to identify as abnormal.

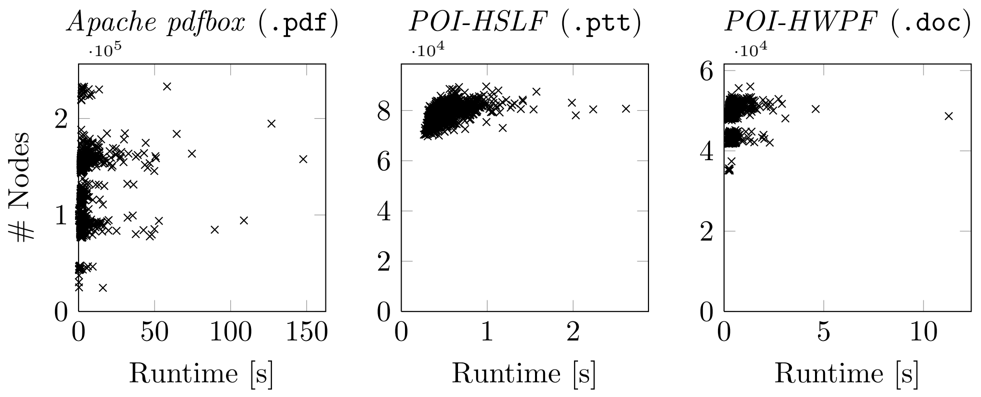


Figure 6.2: Relation between application runtime and model size, measured in number of calling context tree nodes.

advantage of the input document's internal structure, may be harder to recognize, but from a security perspective would probably also be less capable of exploiting a vulnerability in the outsourced application.

6.1.4 RQ3: Scalability

For behavior compliance control to pay off, checking for compliance must be affordable, and must scale to large, long-running applications. We thus evaluate whether the size of the model correlates with the runtime of the application. If this were the case, the compliance check could be as expensive as re-performing the actual outsourced computation, hence defeating the purpose of outsourcing.

For the Function and Call Graph abstractions it is immediately obvious that no such correlation can exist. This is because the number of functions, and consequently the number of call-graph edges, is statically bounded. For the Calling Context Tree abstraction, however, this is not the case. In particular, the use of recursion can cause an application's calling context tree to any size.[4] Figure 6.2 visualizes the relation between application runtime (with CCT logging enabled) and the number of nodes in the resulting CCT profile. Interestingly, in our benchmark longer-running applications do *not* induce significantly larger profiles; thus, our approach scales well over time.

6.1.5 RQ4: Efficiency

We comment on the runtime overhead caused by the instrumentation necessary for profile generation and on the overhead induced by using securely sealed storage.

[4]In practice, the virtual machine's maximum stack size does impose a (large) limit.

Efficiency is not the focus of our research. For the experiments mentioned above, we used a setup as described in Section 3.1.1: we collected calling context trees in all cases, and in a second step computed call graphs and method sets based on the collected trees. This procedure is inefficient. In a real-world setting one would rather opt for a customized instrumentation that emits the respective representation directly, as this can safe a significant amount of execution time. While computing full CCTs will generally incur a significant runtime overhead (10 times or more), one can bring overheads down to under 5% by using probabilistic calling context trees [30]. Such probabilistic CCTs appear quite sufficient for our purposes, and we plan to evaluate their utility in future work. Method sets and call graphs are statically bounded and can therefore be indexed ahead-of-time, which makes instrumentation possible that produces little to no observable runtime overhead [57]. We thereby conclude that sufficiently efficient implementations are possible given the state of the art in dynamic program analysis. While such implementations are outside the scope of this paper, we plan to investigate them in future work.

We measured the runtime cost of our runtime-secure storage on a machine equipped with an AMD Phenom II X2 555 processor and 4 GiB RAM under GNU/Linux (kernel 2.6.32) and the TPM emulator version 0.7.2. Our tests show that the most expensive operation is to create a vTPM instance, which takes 1 second on average. However, this operation is only invoked once, at application startup time. The overhead is caused by the expensive `TPM_TakeOwnership` operation, which creates the Storage Root Key (SRK) key-pair.

The average total cost of storing a CCT profile depends on the average node number. For pdfbox, POI-HSLF and POI-HWPF those are 120,850, 78,568 and 48,239 respectively. Hashing the unique identifier (8 bytes) of every node takes about 6 μs. The instruction `TPM_Extend`, which extends a PCR register with a hash, takes 400 μs. That is, we estimate the overhead of securely storing a full CCT profile for pdfbox, POI-HSLF and POI-HWPF at about 50, 32 and 20 seconds respectively. When using the more coarse-grained Call Graph abstraction, only an average 5,313, 2,338 resp. 2,289 nodes must be stored for pdfbox, POI-HSLF and POI-HWPF respectively, lasting approximately 3.1, 1.95 and 1.93 seconds. The most efficient abstraction are Functions. The overhead for Functions is 2.1, 1.53 and 1.52 seconds for 2,577, 1,301 and 1,281 functions respectively.

Our results show that the cost of secure storage becomes an issue with CCTs but appears low enough for the other two abstractions. In any case, note that storage can be performed asynchronously on a separate processor core (or even a set of those).

6.2 SQL Injection Attack on Web Applications

Web applications constitute a huge part of modern applications, applied in different scenarios, e.g., social networks, online auctions, and all kinds of cloud services constitute popular examples of online services complying with this paradigm. At the same time threats and risks of disclosing sensitive data and running software on unintended way arise. The Open Web Application Security Project (OWASP)[5] is an open community, which helps organizations to develop, purchase, and maintain applications that can be trusted. One of the main projects of OWASP is the Top 10 project, whose goal is to raise awareness about application security by identifying some of the most critical risks facing organizations.

In this section, we evaluate our approach against the most popular attack list in the ten[6] top risks identified by OWASP, namely the injection flaws attack. As a test application we chose the `InsecureWebApp`[7] application, which is a web application that includes common web application vulnerabilities, deployed on the Apache Tomcat Server. We did so to easily find exploits in web applications so that we can evaluate our approach. The application is supposed to simulate some simplified functionalities know from accounting software, such as payments management. It has also a simple user management functionality, which manages access to the application and its functionalities. The application is equipped with an SQL DBMS, namely the HyperSQL (HSQL), which is an SQL relational database engine written in Java.

6.2.1 RQ1: Feasibility

First we created different profiles by through clicking all visible hyperlinks in the application, which is considered as the legal usage of this web application. Some hyperlinks were click multiple times, resulting in calling all `.jsp` pages of the web application. This is was achievable since the considered application is composed of small number of such pages. The result of these tests were 35 profiles, which were considered as legal runs of the application. Note that these profiles were created using only the CCT abstraction. To create the model, we randomly divided the profiles into five subsets containing seven profiles each. Then, each profile from one of the subsets was checked for compliance with application models derived from an increasing number of (randomly chosen) profiles in the other four subsets, up to all 28 profiles in the end. As shown in Figure 6.3, the average of the false positive rate at the end is about 24%.

[5]https://www.owasp.org/

[6]https://www.owasp.org/index.php/Top_10_2013-Top_10

[7]https://www.owasp.org/index.php/Category:OWASP_Insecure_Web_App_Project

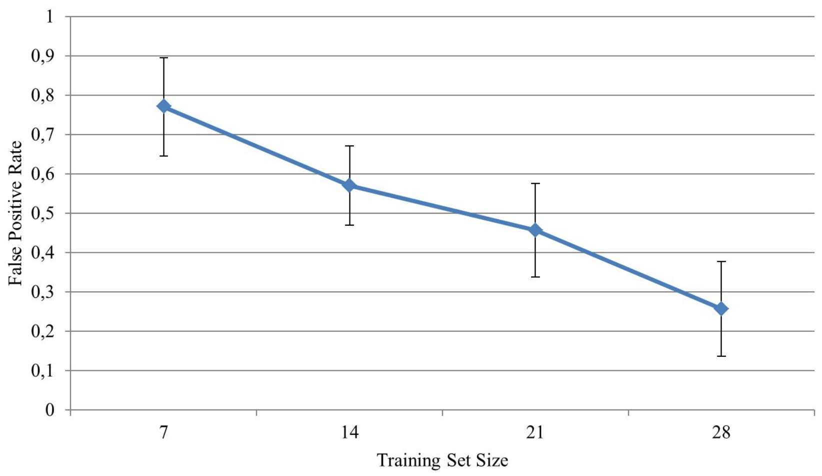


Figure 6.3: False positive rates during creating the model of `InsecureWebApp`

6.2.2 RQ2: Effectiveness

To test the effectiveness of applying our approach on `InsecureWebApp`, we performed multiple SQL injection attacks[8] on it. Since the selected application is prone to this kind of attacks, it was known that such attacks will end successfully. The challenge was to evaluate the ability of our approach to detect such attacks.

Our first attack was to try getting administrator rights on the application, by bypassing the login functionality. In particular, in the field specified for user name we put the administrator user name which is "admin", and in the password field we put following SQL statement:

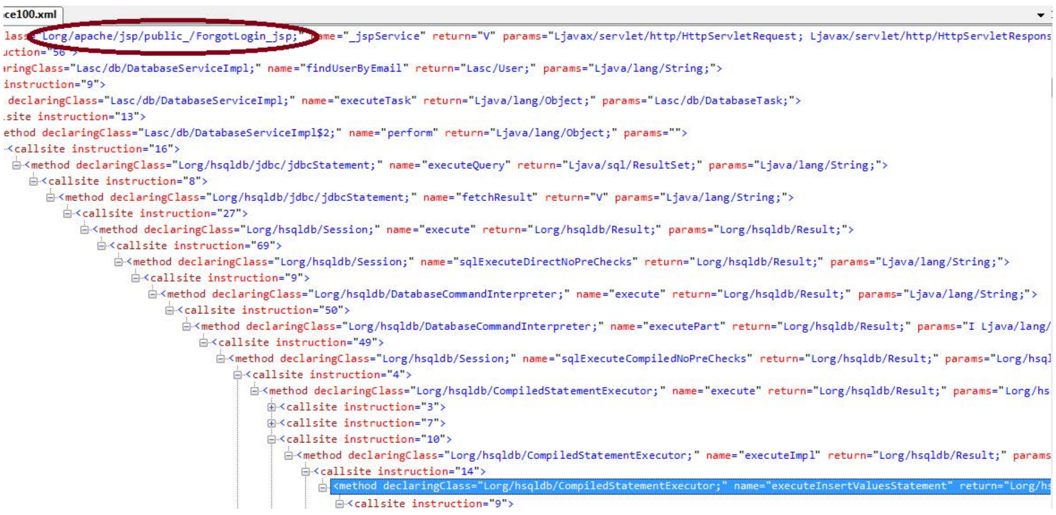
```
' OR 1=1/*
```

Since the application was vulnerable to this attack, the result was successful and access as administrator was granted. Our second attack was to try inserting a new database entry in the "user" table. For this, we determined that the field specified for the resending forgotten passwords is prone to this attack. Thus, we put the following statement to add a new user:

```
x'; INSERT INTO user VALUES (6,'Jeffrey','blakas','Andreas
        Papadreu', 'blakas@BCChacker.com',1); --
```

The attack here also was successful and a new user was added to the table. The next attack is similar to the previous attack. The attack lies in trying to update a user data entry in the "user" table of the database. The following SQL statement was used and performed successfully so the user data was updated:

[8]http://www.unixwiz.net/techtips/sql-injection.html



```
ce100.xml
lass="Lorg/apache/jsp/public_/ForgotLogin_jsp;" name="_jspService" return="V" params="Ljavax/servlet/http/HttpServletRequest; Ljavax/servlet/http/HttpServletRespons
uction="56">
aringClass="Lasc/db/DatabaseServiceImpl;" name="findUserByEmail" return="Lasc/User;" params="Ljava/lang/String;">
instruction="9">
 declaringClass="Lasc/db/DatabaseServiceImpl;" name="executeTask" return="Ljava/lang/Object;" params="Lasc/db/DatabaseTask;">
.site instruction="13">
ethod declaringClass="Lasc/db/DatabaseServiceImpl$2;" name="perform" return="Ljava/lang/Object;" params="">
<callsite instruction="16">
 ⊟<method declaringClass="Lorg/hsqldb/jdbc/jdbcStatement;" name="executeQuery" return="Ljava/sql/ResultSet;" params="Ljava/lang/String;">
   ⊟<callsite instruction="8">
     ⊟<method declaringClass="Lorg/hsqldb/jdbc/jdbcStatement;" name="fetchResult" return="V" params="Ljava/lang/String;">
       ⊟<callsite instruction="27">
         ⊟<method declaringClass="Lorg/hsqldb/Session;" name="execute" return="Lorg/hsqldb/Result;" params="Lorg/hsqldb/Result;">
           ⊟<callsite instruction="69">
             ⊟<method declaringClass="Lorg/hsqldb/Session;" name="sqlExecuteDirectNoPreChecks" return="Lorg/hsqldb/Result;" params="Ljava/lang/String;">
               ⊟<callsite instruction="9">
                 ⊟<method declaringClass="Lorg/hsqldb/DatabaseCommandInterpreter;" name="execute" return="Lorg/hsqldb/Result;" params="Ljava/lang/String;">
                   ⊟<callsite instruction="50">
                     ⊟<method declaringClass="Lorg/hsqldb/DatabaseCommandInterpreter;" name="executePart" return="Lorg/hsqldb/Result;" params="I Ljava/lang/
                       ⊟<callsite instruction="49">
                         ⊟<method declaringClass="Lorg/hsqldb/Session;" name="sqlExecuteCompiledNoPreChecks" return="Lorg/hsqldb/Result;" params="Lorg/hsql
                           ⊟<callsite instruction="4">
                             ⊟<method declaringClass="Lorg/hsqldb/CompiledStatementExecutor;" name="execute" return="Lorg/hsqldb/Result;" params="Lorg/hs
                               ⊞<callsite instruction="3">
                               ⊞<callsite instruction="7">
                               ⊟<callsite instruction="10">
                                 ⊟<method declaringClass="Lorg/hsqldb/CompiledStatementExecutor;" name="executeImpl" return="Lorg/hsqldb/Result;" params
                                   ⊟<callsite instruction="14">
                                     ⊟<method declaringClass="Lorg/hsqldb/CompiledStatementExecutor;" name="executeInsertValuesStatement" return="Lorg/hs
                                       ⊟<callsite instruction="9">
```

Figure 6.4: The trace after executing the insert attack

```
x'; UPDATE user SET email = 'hacker@trojan.net' WHERE
                    email = 'chris@mail.com
```

Afterwards, we started the application together with our extension implemention of BCC and got interesting results. All the previous attacks were detected successfully. The reason for detecting such attacks was because of calling methods in wrong contexts. For example, when performing the attack where a new user is added to the "user" table, a method called ``executeInsertValuesStatement'' is called in a context which is not part of model at this place. Figure 6.4 shows the place of this call in trace resulting after performing the attack. According to the evaluation results, the effectiveness of our approach against such attacks was high.

6.2.3 RQ2: Scalability

The answer of the question correlation between the profile size and the application's runtime for web applications is not trivial. This is because different possible definitions of a profile. We previously defined a profile as a single run of the application. However, in web applications there is an interaction between a client and server, which means that a single run can vary from requesting a single static page to executing a complex computation. In our test application, we tried to cover all possible functionalities. In our evaluation, instead of considering every request/response as profile, we rather randomly collected multiple request/response transactions in one profile to see their impact over time on the size of the profile, i.e., scalability. As shown in Figure 6.5, our results show that the size of the profiles is independent from its running time, i.e., the resulting model scales well over time. Note that

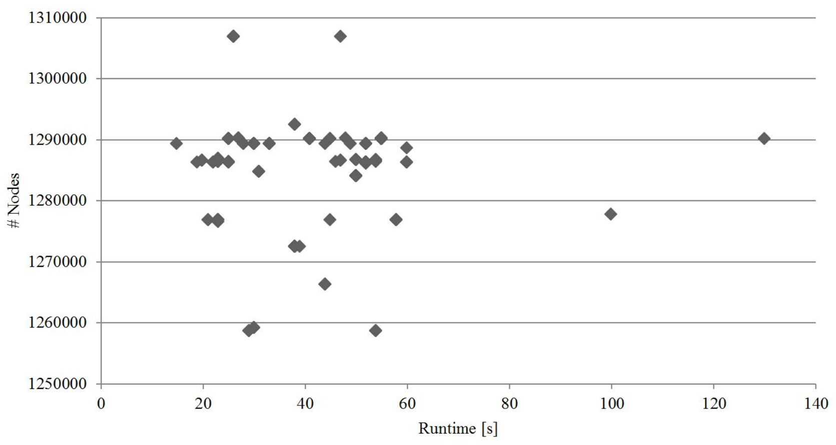


Figure 6.5: The relation between model size and the runtime of `InsecureWebApp`

the number of nodes in this figure contains also the method calls caused by the the container of the web application, namely the Apache Tomcat server.

6.2.4 RQ2: Efficiency

The evaluation results of the efficiency when applying our approach to detect this attack show similar results to the ones presented in Section 6.1.5. For this reason we refer to that section for more details.

7 Summary and Conclusion

This work presented approaches an techniques which allow capturing, controlling the behavior of outsourced computations as well as judging on their compliance. In addition, we presented in this work how one can securely store collected behavior data at runtime.

The Behavior Compliance Control (BCC) approach goes beyond load-time based systems for compliance control by considering the application's runtime behavior. This allows the client outsourcing the application to detect attacks in which the application's code remains unaltered at load-time. Hereby we presented an approach on program code level and another approach which is applied for automated executable business processes. The BCC approach is fully automatic, allowing clients to describe their application's reference behavior by executing test runs in-house, resulting in a so-called application model. We conducted an empirical evaluation trying to test the efficacy of our approach in practice. For example, we evaluated the efficacy of our approach against malicious input attack using more than 3,000 inputs to three different applications. We showed that behavior compliance control can learn reference behavior quickly, resulting in application models that yield few false warnings, but nevertheless successfully identify many attacks through malicious program inputs. Our evaluation showed, though, that some abstractions of program behavior are better suited towards yielding few false warnings, while others are better suited to yielding few missed attacks. We hence conclude that the ideal abstraction of behavior, combining both properties, lies somewhere in between the abstractions we considered so far, opening an interesting avenue for future research.

In addition, we have considered some of the security problems that arise when outsourcing business processes to remote systems. In particular, we considered a scenario where a hosting platform must provide trustworthy evidence about the enforcement of mechanisms which guarantee the security of the platform. We showed how the hosting platform can provide guarantees about the correct execution path of outsourced business processes as described by the customer, plus guarantees about the execution of the corresponding service code. We proposed an architecture to provide such guarantees based on Trusted Computing technologies. In addition, the architecture provides a solution for multi-tenancy hosting platforms considering at the same time multi-instance processes. We also described how such an architecture can be implemented using standard Trusted Computing implementations, the Java virtual machine and the ODE business process engine.

To secure collected behavior information, we proposed techniques and architectures which provide runtime-secure storage based on technologies of Trusted Computing. However, we recognized that some drawbacks and limitations of using standard techniques of Trusted Computing can negatively influence the security and efficiency of the approach. To this end, we proposed some improvements to the main functionalities of Trusted Computing, more specific, to the integrity measurement and remote attestation processes. In particular, we have considered the problem of privacy and scalability in remote attestation, as standardized by the Trusted Computing Group. In particular, the use of SHA-1 hashes to measure the integrity of programs and system components creates a large management overhead; in addition, remote attestation causes privacy problems, as the full state of the system is disclosed. To mitigate these problems we proposed Chameleon Attestation, where we can assign a single hash value to sets of trusted software. By a prototypical implementation we show that the performance overhead of using public-key operations in the attestation process is acceptable.

Another contribution of this work was to provide hardware-based security to the virtual TPMs, used to create the runtime-secure storage, by binding them to a single hardware TPM. For this, two novel approaches, hash tree based binding and incremental hash based binding, have been proposed.

Bibliography

[1] GNU GRUB - GNU Project - Free Software Foundation (FSF). `http://www.gnu.org/software/grub/`.

[2] The tpm4java Library. `http://sourceforge.net/projects/tpm4java/`.

[3] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`.

[4] TrustedGRUB Extension to the GRUB Bootloader. `http://sourceforge.net/projects/trustedgrub/`.

[5] Oasis ws-bpel extension for people (bpel4people), June 2007.

[6] TPM main specification version 1.2, level 2 revision 103, July 2007.

[7] Web services business process execution language 2.0, Apr. 2007.

[8] Software-based tpm emulator, Feb. 2013.

[9] M. Alam, M. Nauman, X. Zhang, T. Ali, and P. C. Hung. Behavioral attestation for business processes. In *Web Services, IEEE International Conference on*, pages 343–350, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[10] M. Alam, X. Zhang, M. Nauman, and T. Ali. Behavioral attestation for web services (BA4WS). In *Proceedings of the 2008 ACM Workshop on Secure Web Services*, pages 21–28, Alexandria, Virginia, USA, 2008. ACM.

[11] S. Alsouri, O. Dagdelen, and S. Katzenbeisser. Group-based attestation: enhancing privacy and management in remote attestation. In *Proceedings of the 3rd international conference on Trust and trustworthy computing*, TRUST'10, pages 63–77, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] S. Alsouri, S. Katzenbeisser, and S. Biedermann. Trustable outsourcing of business processes to cloud computing environments. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 280 –284, sept. 2011.

[13] S. Alsouri, S. Malipatlolla, T. Feller, and S. Katzenbeisser. Hardware-based security for virtual trusted platform modules. *ArXiv e-prints*, Aug. 2013.

[14] S. Alsouri, J. Sinschek, A. Sewe, E. Bodden, M. Mezini, and S. Katzenbeisser. Dynamic anomaly detection for more trustworthy outsourced computation. In *Proceedings of the 15th international conference on Information Security*, ISC'12, pages 168–187, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the 10th Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1997.

[16] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[17] T. Anstett, F. Leymann, R. Mietzner, and S. Strauch. Towards BPEL in the cloud: Exploiting different delivery models for the execution of business processes. In *Proceedings of the 2009 Congress on Services - I*, pages 670–677. IEEE Computer Society, 2009.

[18] Apache Software Foundation. The Apache Java PDF Library (PDFbox). `http://pdfbox.apache.org/`.

[19] Apache Software Foundation. The Java API for Microsoft Documents (Apache POI). `http://poi.apache.org/`.

[20] G. Ateniese and B. de Medeiros. On the key exposure problem in chameleon hashes. In *Security in Communication Networks*, pages 165–179. 2005.

[21] F. Aymerich, G. Fenu, and S. Surcis. An approach to a Cloud Computing network. In *Applications of Digital Information and Web Technologies, 2008. ICADIWT 2008. First International Conference on the*, pages 113–118. IEEE, 2008.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[23] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: incrementality at reduced cost. In *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, EUROCRYPT'97, pages 163–192, Berlin, Heidelberg, 1997. Springer-Verlag.

[24] M. Bellare, D. Micciancio, and B. Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *Advances in Cryptology – EUROCRYPT 2003*, page 644. 2003.

[25] M. Bellare and P. Rogaway. Pss: Provably secure encoding method for digital signatures. *IEEE P1363a*, 1998.

[26] M. Bellare, H. Shi, and C. Zhang. Foundations of group signatures: The case of dynamic groups. In *Topics in Cryptology – CT-RSA 2005*, pages 136–153. 2005.

[27] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, pages 111–131, 2011.

[28] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Vancouver, B.C., Canada, 2006. USENIX Association.

[29] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, 2011.

[30] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 97–112, 2007.

[31] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *Advances in Cryptology CRYPTO 2004*, pages 41–55. 2004.

[32] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, Oct. 1988.

[33] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 132–145, Washington DC, USA, 2004. ACM.

[34] E. Brickell and J. Li. Enhanced privacy ID: a direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, pages 21–30, Alexandria, Virginia, USA, 2007. ACM.

[35] D. Chaum and E. van Heyst. Group signatures. In *Advances in Cryptology – EUROCRYPT 91*, pages 257–265. 1991.

[36] X. Chen and D. Feng. A new direct anonymous attestation scheme from bilinear maps. In *Young Computer Scientists, International Conference for*, pages 2308–2313, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[37] S. Choi, J. Han, and S. Jun. Improvement on tcg attestation and its implication for drm. In *Proceedings of the 2007 international conference on Computational science and its applications - Volume Part I*, ICCSA'07, pages 912–925, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] R. L. R. Crypt. Std: Emsapss – pkcs♯1 v2.1., 2002.

[39] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf. Reconfigurable trusted computing in hardware. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 15–20, New York, NY, USA, 2007. ACM.

[40] P. England. Practical techniques for operating system attestation. In *Trusted Computing - Challenges and Applications*, pages 1–13. 2008.

[41] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[42] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[43] T. Feller, S. Malipatlolla, D. Meister, and S. A. Huss. TinyTPM: A Lightweight Module aimed to IP Protection and Trusted Embedded Platforms. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST 2011)*, June 2011.

[44] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 62–75, Washington, DC, USA, 2003. IEEE Computer Society.

[45] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120 –128, may 1996.

[46] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[47] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 318–329, New York, NY, USA, 2004. ACM.

[48] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Technique (CRYPTO)*, 2010.

[49] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *In 11th USENIX Security Symposium*, pages 61–79, 2002.

[50] B. Glas, A. Klimm, O. Sander, K. Müller-Glaser, and J. Becker. A system architecture for reconfigurable trusted platforms. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 541–544, New York, NY, USA, 2008. ACM.

[51] B.-M. Goi, M. U. Siddiqi, and H.-T. Chuah. Incremental hash function based on pair chaining & modular arithmetic combining. In *Proceedings of the Second International Conference on Cryptology in India: Progress in Cryptology*, INDOCRYPT '01, pages 50–61. Springer-Verlag, 2001.

[52] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proc. of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130, 2010.

[53] L. Gu, Y. Cheng, X. Ding, R. H. Deng, Y. Guo, and W. Shao. Remote attestation on function execution. In *Proceedings of the 1st International Conference on Trusted Systems (INTRUST)*, pages 60–72, 2010.

[54] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei. Remote attestation on program execution. In *Proceedings of the 3rd Workshop on Scalable Trusted Computing (STC)*, pages 11–20, 2008.

[55] L. Gu, X. Ding, R. H. Deng, Y. Zou, B. Xie, W. Shao, and H. Mei. Model-Driven remote attestation: Attesting remote system from behavioral aspect. In *Young Computer Scientists, International Conference for*, volume 0, pages 2347–2353, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[56] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native Multi-Tenancy application development and management. In *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, pages 551–558, Tokyo, Japan, 2007.

[57] T. Gutzmann and W. Löwe. Custom-made instrumentation based on static analysis. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA '11, pages 18–23, New York, NY, USA, 2011. ACM.

[58] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium*, pages 3–20, 2004.

[59] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 3–3, San Jose, California, 2004. USENIX Association.

[60] M. Hammer and J. Champy. *Reengineering the Corporation: A Manifesto for Business Revolution.* HarperBusiness, May 1994.

[61] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.

[62] H. Inoue and S. Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the 2002 Workshop on New Security Paradigms (NSPW)*, pages 52–60, 2002.

[63] D. Jordan and J. Evdemon. Web services business process execution language 2.0, OASIS standard, 2007.

[64] K. Julisch, C. Suter, T. Woitalla, and O. Zimmermann. Compliance by design - bridging the chasm between auditors and it architects. *Computers & Security*, In Press, Corrected Proof, 2011.

[65] Y. Karabulut, F. Kerschbaum, F. Massacci, P. Robinson, and A. Yautsiukhin. Security and trust in IT business outsourcing: a manifesto. *Electronic Notes in Theoretical Computer Science*, 179:47–58, July 2007.

[66] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1137–1143, 1995.

[67] H. Krawczyk and T. Rabin. Chameleon hashing and signatures. In *Proceedings of the Network and Distributed System Security Symposium*, pages 143–154. The Internet Society, 2000.

[68] U. Kühn, M. Selhorst, and C. Stüble. Realizing property-based attestation and sealing with commonly available hard- and software. In *STC '07: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, pages 50–57, New York, NY, USA, 2007. ACM.

[69] M. C. Lacity, S. A. Khan, and L. P. Willcocks. A review of the IT outsourcing literature: Insights for practice. *The Journal of Strategic Information Systems*, 18(3):130–146, Sept. 2009.

[70] A. Lazovik and H. Ludwig. Managing process customizability and customization: Model, language and process. In *IN: PROCEEDINGS OF WISE*, 2007.

[71] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 36–44, 1998.

[72] J. Lyle and A. Martin. On the feasibility of remote attestation for web services. In *2009 International Conference on Computational Science and Engineering*, pages 283–288, Vancouver, BC, Canada, 2009.

[73] R. C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford, CA, USA, 1979. AAI8001972.

[74] P. Moret, W. Binder, A. Villazón, D. Ansaloni, and A. Heydarnoori. Visualizing and exploring profiles with calling context ring charts. *Softw. Pract. Exper.*, 40(9):825–847, Aug. 2010.

[75] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist. WS-Trust 1.3, 2007.

[76] M. P. Papazoglou and W. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[77] R. V. Peri, S. Jinturkar, and L. Fajardo. A novel technique for profiling programs in embedded systems. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1999.

[78] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proc. of the 24th International Conference on Automated Software Engineering (ASE)*, pages 371–382, 2009.

[79] R. Ramer and S. E. Goodman. Global sourcing of IT services and information security: Prudence before playing. *Communications of the Association for Information Systems*, 20(1), Dec. 2007.

[80] A. Sadeghi. Trusted computing: special aspects and challenges. In *Proceedings of the 34th conference on Current trends in theory and practice of computer science*, pages 98–117. Springer-Verlag, 2008.

[81] A. Sadeghi, C. Stble, and M. Winandy. Property-Based TPM virtualization. In *Proceedings of the 11th international conference on Information Security*, pages 1–16, Taipei, Taiwan, 2008. Springer-Verlag.

[82] A. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. pages 67–77, Nova Scotia, Canada, 2004. ACM.

[83] A.-R. Sadeghi, C. Stüble, and M. Winandy. Property-based tpm virtualization. In *Proceedings of the 11th international conference on Information Security*, ISC '08, pages 1–16, Berlin, Heidelberg, 2008. Springer-Verlag.

[84] R. Sailer. Integrity measurement architecture (IMA). http://researcher.watson.ibm.com/researcher/view_project.php?id=2851.

[85] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, Washington DC, USA, 2004. ACM.

[86] R. Sailer, X. Zhang, T. Jaeger, and van Doorn Leendert. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004. USENIX Association.

[87] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 2012.

[88] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java virtual machine. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011.

[89] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a tpm without

a trusted os. In *Proceedings of the first ACM workshop on Scalable trusted computing*, STC '06, pages 27–42. ACM, 2006.

[90] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik. Tpm virtualization: Building a general framework. In N. Pohlmann and H. Reimer, editors, *Trusted Computing*, pages 43–56. Vieweg+Teubner, 2008.

[91] A. U. Schmidt, A. Leicher, Y. Shah, and I. Cha. Tree-formed verification data for trusted platforms. *CoRR*, abs/1007.0642, 2010.

[92] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[93] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, Washington, DC, USA, 2001. IEEE Computer Society.

[94] M. Strasser and H. Stamer. A software-based trusted platform module emulator. In *Proceedings of the 1st Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications (Trust)*, pages 33–47, 2008.

[95] M. Sunil, F. Thomas, S. Abdulhadi, A. Tolga, and S. A. Huss. A novel architecture for a secure update of cryptographic engines on trusted platform module. In *IEEE International Conference on Field Programmable Technoology (FPT)*, Dec. 2011.

[96] K. M. C. Tan, J. McHugh, and K. S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Revised Papers from the 5th International Workshop on Information Hiding*, pages 1–17, London, UK, UK, 2003. Springer-Verlag.

[97] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. of the 24th International Conference on Automated Software Engineering (ASE)*, pages 283–294, 2009.

[98] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. of the 31st International Conference on Software Engineering (ICSE)*, pages 496–506, 2009.

[99] Trusted Computing Group. Tpm main part 2 tpm structures.

[100] Trusted Computing Group. TCG software stack (TSS) specification version 1.2, Mar. 2007.

[101] Trusted Computing Group. Tcg specification architecture overview 1.4, Aug. 2007.

[102] Trusted Computing Group. TPM main specification version 1.2, level 2 revision 103, July 2007.

[103] Trusted Computing Group. TPM main specification version 1.2, level 2 revision 116, part 3 - commands, 2011.

[104] I. Trusted Computing Group. Summary Of Features Under Consideration For The Next Generation Of TPM, 2009.

[105] Trusted Computing Group, Inc. *TPM Main Specification Level 2 Version 1.2*, march 2011. Revision 116.

[106] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[107] M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 41–48. ACM, 2007.

[108] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[109] T.J. Watson Libraries for Analysis (WALA). `http://wala.fs.net/`.

[110] C. Weinhardt, A. Anandasivam, B. Blau, N. Borissov, T. Meinl, W. Michalk, and J. Stößer. Cloud computing - a classification, business models, and research directions. *Business & Information Systems Engineering*, 1(5):391–399, 2009.

[111] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87. ACM, 2000.

[112] D. Williams and E. G. Sirer. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *Network Computing and Applications, 2004. (NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 383 – 388, aug.-1 sept. 2004.

[113] S. Yoshihama, T. Ebringer, M. Nakamura, S. Munetoh, and H. Maruyama. WS-Attestation: efficient and Fine-Grained remote attestation on web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 743–750. IEEE Computer Society, 2005.

[114] X. Zhang, J. Seifert, and R. Sandhu. Security enforcement model for distributed usage control. In *Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (sutc 2008)*, pages 10–18. IEEE Computer Society, 2008.

[115] X. Zhang, J.-P. Seifert, and R. Sandhu. Security enforcement model for distributed usage control. In *Proceedings of the Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, pages 10 –18, 2008.

[116] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *ACM SIGPLAN Notices*, volume 41, pages 263–271. ACM, 2006.

Wissenschaftlicher Werdegang

Oktober 2001 – Januar 2009
Studium der Informatik (Diplom) an der Technischen Universität Darmstadt

März 2009 – August 2013
Promotion an der Technischen Universität Darmstadt unter Leitung von Prof. Dr. Stefan Katzenbeisser

März 2009 – Februar 2013
Stipendiat an Center for Advanced Security Research Darmstadt - CASED