PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Doctorado
en Informática

Formal analysis of security models for mobile devices, virtualization platforms, and domain name systems

Carlos Luna

2014

Formal analysis of security models for
mobile devices, virtualization platforms,
and domain name systems
Carlos Luna
ISSN 0797-6410
Tesis de Doctorado en Informática
Reporte Técnico RT 14-11
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 2014

PEDECIBA INFORMÁTICA

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

TESIS DE DOCTORADO EN INFORMÁTICA

Formal analysis of security models for mobile devices, virtualization platforms, and domain name systems

Carlos Luna
cluna@fing.edu.uy

Agosto de 2014

Director académico y de tesis: Gustavo Betarte

Tribunal

Nazareno Aguirre
Departamento de Computación, Facultad de Ciencias Exactas, Físico-Químicas y Naturales, Universidad Nacional de Río Cuarto, Argentina (Revisor)

Ana Bove
Department of Computing Science and Engineering, Chalmers University of Technology, Sweden / PEDECIBA Informática

Eduardo Grampín
Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay / PEDECIBA Informática

Alvaro Martín
Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay / PEDECIBA Informática (Presidente)

Tamara Rezk
INRIA Sophia Antipolis-Mediterranée, France (Revisora)

Formal analysis of security models for mobile devices,
virtualization platforms, and domain name systems
Luna, Carlos
ISSN 0797-6410
Tesis de Doctorado en Informática
Reporte Técnico RT ___-2014
PEDECIBA
Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, Agosto de 2014

Caminante, son tus huellas el camino y nada más;

caminante, no hay camino, se hace camino al andar...

Antonio Machado – Proverbios y Cantares; Campos de Castilla (1912)

Formal analysis of security models for mobile devices, virtualization platforms, and domain name systems

Abstract

In this thesis we investigate the security of security-critical applications, i.e. applications in which a failure may produce consequences that are unacceptable. We consider three areas: mobile devices, virtualization platforms, and domain name systems.

The Java Micro Edition platform defines the Mobile Information Device Profile (MIDP) to facilitate the development of applications for mobile devices, like cell phones and PDAs. In this work we first study and compare formally several variants of the security model specified by MIDP to access sensitive resources of a mobile device.

Hypervisors allow multiple guest operating systems to run on shared hardware, and offer a compelling means of improving the security and the flexibility of software systems. In this thesis we present a formalization of an idealized model of a hypervisor, and we establish (formally) that the hypervisor ensures strong isolation properties between the different operating systems, and guarantees that requests from guest operating systems are eventually attended. We show also that virtualized platforms are transparent, i.e. a guest operating system cannot distinguish whether it executes alone or together with other guest operating systems on the platform.

The Domain Name System Security Extensions (DNSSEC) is a suite of specifications that provides origin authentication and integrity assurance services for DNS data. We finally present a minimalistic specification of a DNSSEC model which provides the grounds needed to formally state and verify security properties concerning the chain of trust of the DNSSEC tree.

We develop all our formalizations in the Calculus of Inductive Constructions —formal language that combines a higher-order logic and a richly-typed functional programming language— using the Coq proof assistant.

Keywords: Formal modelling, Security properties, Coq proof assistant, JME-MIDP, Virtualization, DNSSEC.

Análisis formal de modelos de seguridad para dispositivos móviles, plataformas de virtualización y sistemas de nombres de dominio

Resumen

En esta tesis investigamos la seguridad de aplicaciones de seguridad críticas, es decir aplicaciones en las cuales una falla podría producir consecuencias inaceptables. Consideramos tres áreas: dispositivos móviles, plataformas de virtualización y sistemas de nombres de dominio.
La plataforma Java Micro Edition define el Perfil para Dispositivos de Información Móviles (MIDP) para facilitar el desarrollo de aplicaciones para dispositivos móviles, como teléfonos celulares y asistentes digitales personales. En este trabajo primero estudiamos y comparamos formalmente diversas variantes del modelo de seguridad especificado por MIDP para acceder a recursos sensibles de un dispositivo móvil.
Los hipervisores permiten que múltiples sistemas operativos se ejecuten en un hardware compartido y ofrecen un medio para establecer mejoras de seguridad y flexibilidad de sistemas de software. En esta tesis formalizamos un modelo de hipervisor y establecemos (formalmente) que el hipervisor asegura propiedades de aislamiento entre los diferentes sistemas operativos de la plataforma, y que las solicitudes de estos sistemas son atendidas siempre. Demostramos también que las plataformas virtualizadas son transparentes, es decir, que un sistema operativo no puede distinguir si ejecuta sólo en la plataforma o si lo hace junto con otros sistemas operativos.
Las Extensiones de Seguridad para el Sistema de Nombres de Dominio (DNSSEC) constituyen un conjunto de especificaciones que proporcionan servicios de aseguramiento de autenticación e integridad de origen de datos DNS. Finalmente, presentamos una especificación minimalista de un modelo de DNSSEC que proporciona los fundamentos necesarios para formalmente establecer y verificar propiedades de seguridad relacionadas con la cadena de confianza del árbol de DNSSEC.
Desarrollamos todas nuestras formalizaciones en el Cálculo de Construcciones Inductivas —lenguaje formal que combina una lógica de orden superior y un lenguaje de programación funcional tipado— utilizando el asistente de pruebas Coq.

Palabras clave: Modelos formales, Propiedades de seguridad, Asistente de pruebas Coq, JME-MIDP, Virtualización, DNSSEC.

Contents

List of Figures

List of Tables

Acknowledgments

En primer lugar, quiero agradecer a mi director académico y de tesis, *Gustavo Betarte (Gustun)*, quien me brindó todo el apoyo necesario para la realización de esta tesis. Su trabajo, guía y motivación constantes fueron claves para mi durante estos años. Mil gracias *Gustun*!

A *Gilles Barthe*, por empeñarse –junto con *Gustun*– en que concretara el doctorado. Haber trabajado y publicado junto a él, en temas de virtualización, ha sido para mi una experiencia muy valiosa, de la cual aprendí mucho. Gracias *Gilles* también por los comentarios sobre versiones preliminares de este documento.

A los coautores de los artículos (en Argentina y Uruguay) que constituyen la base de esta tesis. En particular, a *Santiago Zanella Béguelin*, *Ramin Roushani*, *Juan Manuel Crespo*, *Gustavo Mazeikis*, *Julio Pérez*, *Jesús Mauricio Chimento* y *Ezequiel Bazán*.

A *Juan Diego Campo*, un gran compañero de trabajo, con quien compartimos el camino del doctorado en este último tiempo, y muchas jornadas de trabajo en torno a dos proyectos de investigación. Su tesis será sin dudas memorable!

A los miembros del tribunal, y en particular a los revisores, por sus comentarios y correcciones. Asimismo, a los revisores anónimos de los artículos referidos.

Al *PEDECIBA Informática* y al *Instituto de Computación (InCo) de la Facultad de Ingeniería*, por darme la oportunidad.

A mis compañeras y compañeros de grupo; tanto en el área de *métodos formales* como de *seguridad informática*. De distintas maneras, contribuyeron con este trabajo.

A las y los docentes que influyeron en mi iniciación en el campo de la investigación. En particular, a *Jorge Aguirre*, *Gilles Barthe*, *Grabriel Baum*, *Gustavo Betarte*, *Cristina Cornes*, *Alberto Pardo*, *Nora Szasz* y *Alvaro Tasistro*.

A la Universidad Nacional de Río Cuarto y la Universidad Nacional de Rosario (Argentina); a la Universidad de la República y la Universidad ORT (Uruguay), por brindarme la posibilidad de participar en estos años de una vida académica, que tanto me apasiona.

A mis colegas y amigos, por acompañarme y aguantarme en este camino...

Finalmente, agradezco a mi familia (de aquí y de allá) el apoyo y la comprensión durante tanto tiempo! Dedico especialmente esta tesis a ellas: *Juli*, *Sofi*, *Mayru* (mis tres niñas) y *Jóse*.

Este trabajo fue financiado, parcialmente, por: i) STEVE: Security Through Verifiable Evidence (PDT 63/118, FCE 2006, DINACYT, Uruguay, 2007-2009); ii) VirtualCert: Towards a Certified Virtualization Platform (ANII-Clemente Estable, PR-FCE-2009-1-2568, Uruguay, 2010-2012); and iii) VirtualCert: Towards a Certified Virtualization Platform - Phase II (UDELAR-CSIC I+D, Uruguay, 2013-2015).

Chapter 1

Introduction

There are multiple definitions of the term safety-critical system. In particular, if a system failure could lead to consequences that are determined to be unacceptable, then the system is safety-critical. In essence, a system is safety-critical when we depend on it for our welfare.

In this thesis, we investigate the security of three areas of safety-critical applications:

1. *mobile devices*: with increasing capabilities in mobile devices (like cell phones and PDAs) and posterior consumer adoption, these devices have become an integral part of how people perform tasks in their works and personal lives. Unfortunately, the benefits of using mobile devices are sometimes counteracted by security risks;

2. *virtualization platforms*: hypervisors allow multiple operating systems to run in parallel on the same hardware, by ensuring isolation between their guest operating systems. In effect, hypervisors are increasingly used as a means to improve system flexibility and security; unfortunately, they are often vulnerable to attacks;

3. *domain name systems*: the domain name systems constitute distributed databases that provide support to a wide variety of network applications such as electronic mail, WWW, and remote login. The important and critical role of these systems in software systems and networks makes them a prime target for (formal) verification.

1.1 Security policies

In general, *security* (mainly) involves the combination of confidentiality, integrity and availability [1]. *Confidentiality* can be understood as the prevention of unauthorized disclosure of information; *integrity*, as the prevention of unauthorized modification of information; and *availability*, as the prevention of unauthorized withholding of information or resources [2]. However, this list is incomplete.

We distinguish between security models and security policies. The *security model* term could be interpreted as the representation of security systems confidentiality, integrity and availability requirements [3]. The more general usage of the term specifies a particular mechanism, such as access control, for enforcing (in particular) confidentiality, which has been adopted for computer security. Security issues arise in many different contexts; so many security models have been developed.

A *security policy* is a specification of the protection goals of a system. This specification determines the security properties that the system should possess. A security policy defines executions of a system that, for some reason, are considered unacceptable [4]. For instance, a security policy may affect:

- *access control*, and restrict the operations that can be performed on elements;
- *availability*, and restrict processes from denying others the use of a resource;
- *information flow*, and restrict what can be inferred about objects from observing system behavior.

To formulate a policy is necessary to describe the entities governed by the policy and set the rules that constitute the policy. This could be done informally in a natural language document. However, approaches based on natural languages fall short of providing the required security guarantees. Because of the ambiguity and imprecision of informal specification, it is difficult to verify the correctness of the system. To avoid these problems, formal security models are considered.

1.2 Security models

Security models play an important role in the design and evaluation of high assurance security systems. Their importance was already pointed out in the Anderson report [5]. The first model to appear was Bell-LaPadula [6], in 1973, in response to US Air Force concerns over the confidentiality of data in time-sharing mainframe systems.

In the last decades, some countries developed specific security evaluation standards [7]. This initiative opened the path to worldwide mutual recognition of security evaluation results. In this direction, new Common Criteria (CC) [8] was developed. CC defines seven levels of assurance for security systems (EAL 1–7). In order to obtain a higher assurance than EAL 4, developers require specification of a security model for security systems and verify security using formal methods. For example, the Gemalto and Trusted Logic companies obtained the highest level of certification (EAL 7) for their formalization of the security properties of the JavaCard platform [9, 10, 11], using the Coq proof assistant [12, 13].

State machines are a powerful tool used for modeling many aspects of computing systems. In particular, they can be employed to specify security models. The basic features of a state machine model are the concepts of state and state change. A *state* is a representation of the system under study at a given time, which should capture those system aspects relevant to our problem. Possible *state transitions* can be specified by a state transition function that defines the next state based on the current state and input. An output can also be produced.

If we want to analize a specific safety property of a system, like security, using a state machine model, we must first identify all the states that satisfy the property. Then we can to check if all state transitions *preserve* this property. If this is the case and if the system starts in an *initial state* that has this property, then we can prove by induction that the property will always be fulfilled. Thus, state machines can be used to enforce a collection of security policies on a system.

1.3 Reasoning about implementations and models

The increasingly important role of critical components in software systems (as hypervisors in virtual platforms or access control mechanisms in mobile devices) make them a prime target for formal verification. Indeed, several projects have set out to formally verify the correctness of critical system implementations, e.g. [14, 15, 16].

Reasoning about implementations provides the ultimate guarantee that deployed critical systems provide the expected properties. There are however significant hurdles with this approach, especially if one focuses on proving security properties rather than functional correctness. First, the complexity of formally proving non-trivial properties of implementations might be overwhelming in terms of the effort it requires; worse, the technology for verifying some classes of security properties may be underdeveloped: specifically, liveness properties are generally hard to prove, and there is currently no established method for verifying, using tools, security properties involving two system executions, a.k.a. 2-properties, for implementations, making their formal verification on large and complex programs exceedingly challenging. For instance, operating system isolation property is a 2-safety property [17, 18] that requires reasoning about two program executions. Second, many implementation details are orthogonal to the security properties to be established, and may complicate reasoning without improving the understanding of the essential features for guaranteeing important properties. Thus, there is a need for complementary approaches where verification is performed on idealized models that abstract away from the specifics of any particular implementation, and yet provide a realistic setting in which to explore the security issues that pertain to the realm of those critical systems.

1.4 Formal analysis of security models for critical systems

In this section we introduce the research lines —in the three domains of safety-critical applications listed above— and contributions of the thesis. The work was developed in the context of the following research projects [19]:

- ReSeCo: Reliability and Security of Distributed Software Components (STIC-AMSUD, 2006-2009);
- STEVE: Security Through Verifiable Evidence (PDT 63/118, FCE 2006, DINACYT, Uruguay, 2007-2009);
- Especificación Formal y Verificación de Sistemas Críticos (SeCyT-FCEIA ING266, UNR, Argentina, 2009-2010);
- VirtualCert: Towards a Certified Virtualization Platform (ANII-Clemente Estable, PR-FCE-2009-1-2568, Uruguay, 2010-2012);
- VirtualCert: Towards a Certified Virtualization Platform - Phase II (UDELAR-CSIC I+D, Uruguay, 2013-2015).

1.4.1 Mobile devices

Today, even entry-level mobile devices (e.g. cell phones) have access to sensitive personal data, are subscribed to paid services and can establish connections with external entities.

Users of such devices may, in addition, download and install applications from untrusted sites at their will. This flexibility comes at a cost, since any security breach may expose sensitive data, prevent the use of the device, or allow applications to perform actions that incur a charge for the user. It is therefore essential to provide an application security model that can be relied upon—the slightest vulnerability may imply huge losses due to the scale the technology has been deployed.

Java Micro Edition (JME) [20] is a version of the Java platform targeted at resource-constrained devices which comprises two kinds of components: configurations and profiles. A configuration is composed of a virtual machine and a set of APIs that provide the basic functionality for a particular category of devices. Profiles further determine the target technology by defining a set of higher level APIs built on top of an underlying configuration. This two-level architecture enhances portability and enables developers to deliver applications that run on a wide range of devices with similar capabilities. This work concerns the topmost level of the architecture which corresponds to the profile that defines the security model we formalize.

The Connected Limited Device Configuration (CLDC) [21] is a JME configuration designed for devices with slow processors, limited memory and intermittent connectivity. CLDC together with the Mobile Information Device Profile (MIDP) provides a complete JME runtime environment tailored for devices like cell phones and personal data assistants. MIDP defines an application life cycle, a security model and APIs that offer the functionality required by mobile applications, including networking, user interface, push activation and persistent local storage. Many mobile device manufacturers have adopted MIDP since the specification was made available. Nowadays, literally billions of MIDP enabled devices are deployed worldwide and the market acceptance of the specification is expected to continue to grow steadily.

The security model of MIDP has evolved since it was first introduced. In MIDP 1.0 [22] and MIDP 2.0 [23] the main goal is the protection of sensitive functions provided by the device. In MIDP 3.0 [24] the protection is extended to the resources of an application, which can be shared with other applications. Some of these features are incorporated in the Android security model [25, 26, 27] for mobile devices, as we will see in Section 2.6.

Contributions

The contributions of the work in this domain are manyfold:

1. We describe a formal specification of the MIDP 2.0 security model that provides an abstraction of the state of a device and security-related events that allows us to reason about the security properties of the platform where the model is deployed;

2. The security of the desktop edition of the Java platform (JSE) relies on two main modules: a *Security Manager* and an *Access Controller*. The Security Manager is responsible for enforcing a security policy declared to protect sensitive resources; it intercepts sensitive API calls and delivers permission and access requests to the Access Controller. The Access Controller determines whether the caller has the necessary rights to access resources. While in the case of the JSE platform, there exists a high-level specification of the Access Controller module (the basic mechamism is based on stack inspection [28]), no equivalent formal specification exists for JME. We illustrate the pertinence of the specification of the MIDP 2.0

security model that has been developed by specifying and proving the correctness of an *access control module* for the JME platform;

3. The latest version of MIDP, MIDP 3.0, introduces a new dimension in the security model at the application level based on the concept of authorizations, allowing delegation of rights between applications. We extend the formal specification developed for MIDP 2.0 to incorporate the changes introduced in MIDP 3.0, and show that this extension is conservative, in the sense that it preserves the security properties we proved for the previous model;

4. Besson, Duffay and Jensen [29] put forward an alternative access control model for mobile devices that generalizes the MIDP model by introducing permissions with multiplicites, extending the way permissions are granted by users and consumed by applications. One of the main outcomes of the work we report here is a general framework that sets up the basis for defining, analyzing and comparing access control models for mobile devices. In particular, this framework subsumes the security model of MIDP and several variations, including the model defined in [29].

1.4.2 Virtualization platforms

Hypervisors allow several operating systems to coexist on commodity hardware, and provide support for multiple applications to run seamlessly on the guest operating systems they manage. Moreover, hypervisors provide a means to guarantee that applications with different security policies can execute securely in parallel, by ensuring isolation between their guest operating systems. In effect, hypervisors are increasingly used as a means to improve system flexibility and security, and authors such as [30] already in 2007 predicted that their use would become ubiquitous in enterprise data centers and cloud computing.

The increasingly important role of hypervisors in software systems makes them a prime target for formal verification. Indeed, several projects have set out to formally verify the correctness of hypervisor implementations. One of the most prominent initiatives is the Microsoft Hyper-V verification project [14, 15, 31], which has made a number of impressive achievements towards the functional verification of the legacy implementation of the Hyper-V hypervisor, a large software component that combines C and assembly code (about 100 kLOC of C and 5kLOC of assembly). The overarching objective of the formal verification is to establish that a guest operating system cannot observe any difference between executing through the hypervisor or directly on the hardware. The other prominent initiative is the L4.verified project [16], which completed the formal verification of the seL4 microkernel, a general purpose operating system of the L4 family. The main thrust of the formal verification is to show that an implementation of the microkernel correctly refines an abstract specification. Subsequent machine-checked developments prove that seL4 enforces integrity, authority confinement [32] and intransitive non-interference [33]. The formalization does not model cache.

Reasoning about implementations provides the final guarantee that deployed hypervisors provide the expected properties. There are however, as mentioned previously, serious difficulties with this approach. For example, many implementation details are orthogonal to the security properties to be established, and may complicate reasoning

without improving the understanding of the essential features for guaranteeing isolation among guest operating systems. Therefore, there is a need for complementary approaches where verification is performed on idealized models that abstract away from the specifics of any particular hypervisor, and yet provide a realistic environment in which to investigate the security problems that pertain to the realm of hypervisors.

Contributions

The work presented here initiates such an approach by developing a minimalistic model of a hypervisor, and by formally proving that the hypervisor correctly enforces isolation between guest operating systems, and under mild hypotheses guarantees basic availability properties to guest operating systems. In order to achieve some reasonable level of tractability, our model is significantly simpler than the setting considered in the Microsoft Hyper-V verification project, it abstracts away many specifics of memory management such as shadow page tables (SPTs) and of the underlying hardware and runtime environment such as I/O devices. Instead, our model focuses on the aspects that are most relevant for isolation properties, namely read and write resources on machine addresses, and is sufficiently complete to allow us to reason about isolation properties. Specifically, we show that an operating system can only read and modify memory it owns, and a non-influence property [34] stating that the behavior of an operating system is not influenced by other operating systems. In addition, our model allows reasoning about availability; we prove, under reasonable conditions, that all requests of a guest operating system to the hypervisor are eventually attended, so that no guest operating system waits indefinitely for a pending request. Overall, our verification effort shows that the model is adequate to reason about safety properties (read and write isolation), 2-safety properties (OS isolation), and liveness properties (availability).

Additionally, in this work we present an implementation of a hypervisor in the programming language of Coq, and a proof that it realizes the axiomatic semantics, on an extended memory model with a formalization of the cache and Translation Lookaside Buffer (TLB). Although it remains idealized and far from a realistic hypervisor, the implementation arguably provides a useful mechanism for validating the axiomatic semantics. The implementation is total, in the sense that it computes for every state and action a new state or an error. Thus, soundness is proved with respect to an extended axiomatic semantics in which transitions may lead to errors. An important contribution in this part of the work is a proof that OS isolation remains valid for executions that may trigger errors.

Finally, we show that virtualized platforms are *transparent*, i.e. a guest operating system cannot distinguish whether it executes alone or together with other guest operating systems on the platform.

1.4.3 Domain name systems

The Domain Name System (DNS) [35, 36] constitutes a distributed database that provides support to a wide variety of network applications such as electronic mail, WWW, and remote login. The database is indexed by *domain names*. A domain name represents a path in a hierarchical tree structure, which in turn constitutes a *domain name space*. Each node of this tree is assigned a label, thus, a domain name is built as a sequence of labels separated by a dot, from a particular node up to the root of the tree.

A distinguishing feature of the design of DNS is that the administration of the system can be distributed among several (authoritative) name servers. A *zone* is a contiguous part of the domain name space that is managed by a set of authoritative name servers. Then, distribution is achieved by delegating part of a zone administration to a set of delegated sub-zones. DNS is a widely used scalable system, but it was not conceived with security concerns in mind, as it was designed to be a public database with no intentions to restrict access to information. Nowadays, a large amount of distributed applications make use of domain names. Confidence on the working of those aplications depends critically on the use of trusted data: fake information inside the system has been shown to lead to unexpected and potentially dangerous problems.

Already in the early 90's serious security flaws were discovered by Bellovin and eventually reported in [37]. Different types of security issues concerning the working of DNS have been discussed in the literature, see, for instance, [38, 37, 39, 40, 41, 42]. Identified vulnerabilities of DNS make it possible to launch different kinds of attacks, namely: *cache poisoning*, *client flooding*, *dynamic update vulnerability*, *information leakage* and *compromise of the DNS servers authoritative database* [43, 37, 44, 45, 46].

Domain Name System Security Extensions (DNSSEC) [47, 48, 49] is a suite of Internet Engineering Task Force (IETF) specifications for securing information provided by DNS. More specifically, this suite specifies a set of extensions to DNS which are oriented to provide mechanisms that support authentication and integrity of DNS data but not its availability or confidentiality. In particular, the security extensions were designed to protect resolvers from forged DNS data, such as the one generated by DNS cache poisoning [45, 45, 46], by digitally signing DNS data using public-key cryptography. The keys used to sign the information are authenticated via a chain of trust, starting with a set of verified public keys that belong to the DNS root zone, which is the trusted third party.

The DNSSEC standards were finally released in 2005 and a number of testbeds, pilot deployments, and services have been rolled out in the last few years [50, 51, 52, 53, 54, 55, 56]. In particular, the main objective of the OpenDNSSEC project [55] is to develop an open source software that manages the security of domain names on the Internet.

Contributions

The important and critical role of DNSSEC in software systems and networks makes it a prime target for formal analysis. This thesis presents the development of a minimalistic specification of a DNSSEC model, and yet provides a realistic setting in which to explore the security issues that pertain to the realm of DNS. The specification puts forward an abstract formulation of the behavior of the protocol and the corresponding security-related events, where security goals, such as the prevention of cache poisoning attacks, can be given a formal treatment. In particular, the formal model provides the grounds needed to formally state and verify security properties concerning the chain of trust generated along the DNSSEC tree.

1.4.4 Personal contributions

The thesis builds upon and extends a number of previously published papers:

1. Betarte, G., Luna, C., Zanella Béguelin, S.: A formal specification of the MIDP 2.0 security model. In: 4th International workshop on Formal Aspects in Security

and Trust, FAST 2006. Volume 4691 of Lecture Notes in Computer Science, Springer-Verlag (2006) 220–234;

2. Betarte, G., Luna, C., Roushani Oskui, R.: A certified access controller for JME-MIDP 2.0 enabled mobile devices. In: 2009 International Conference of the Chilean Computer Science Society, SCCC 2009, Los Alamitos, CA, USA, IEEE Computer Society (2009) 51–58;

3. Betarte, G., Luna, C., Mazeikis, G.: Formal specification and analysis of the MIDP 3.0 security model. In: 2009 International Conference of the Chilean Computer Science Society, SCCC 2009, Los Alamitos, CA, USA, IEEE Computer Society (2009) 59–66;

4. Betarte, G., Crespo, J., Luna, C.: A framework for the analysis of access control models for interactive mobile devices. In: Types for Proofs and Programs, International Conference, TYPES 2008. Volume 5497 of Lecture Notes in Computer Science, Springer-Verlag (2009) 49–63;

5. Barthe, G., Betarte, G., Campo, J., Luna, C.: Formally verifying isolation and availability in an idealized model of virtualization. In Butler, M., Schulte, W., eds.: Formal Methods 2011. Volume 6664 of Lecture Notes in Computer Science, Springer-Verlag (2011) 231–245;

6. Barthe, G., Betarte, G., Campo, J.D., Chimento, J.M., Luna, C.: Formally verified implementation of an idealized model of virtualization. In Matthes, R., Schubert, A., eds.: 19th International Conference on Types for Proofs and Programs (TYPES 2013). Volume 26 of Leibniz International Proceedings in Informatics (LIPIcs)., Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2014) 45–63;

7. Betarte, G., Eixarch, E. B., Luna, C. D.: A formal specification of the DNSSEC model. Electronic Communications of the European Association of Software Science and Technology (ECEASST) 48 (2011) 1–21.

In each of the papers that constitute the core of this thesis, I worked on the analysis of different security models considered, in the development of formal specifications, the formulation of relevant security properties, demonstrating many of these properties, and building certified implementations. I have also collaborated in the analysis of results and writing of those papers. Finally, I participated in the following works directly related to this thesis, but which are not part of it:

- Barthe, G., Betarte, G., Campo, J., Luna, C.: Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In: IEEE 25th Computer Security Foundations Symposium (2012) 186–197;

- Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. To appear in proceedings of the 21st ACM Conference on Computer and Communications Security (2014).

Throughout the development of the thesis, and in the context of research projects involved, I have officiated as (co)director of the final degree projects of the following

students (in Argentina and Uruguay): Santiago Zanella Béguelin, Ramin Roushani Oskui, Juan Manuel Crespo, Gustavo Mazeikis, Julio Pérez, Jesús Mauricio Chimento, and Ezequiel Bazán.

1.5 Formal language used

1.5.1 Coq

The Coq proof assistant [12, 13] is a free open source software that provides a (dependently typed) functional programming language and a reasoning framework based on higher order logic to perform proofs of programs. Coq allows developing mathematical facts. This includes defining objects (sets, lists, functions, programs); making statements (using basic predicates, logical connectives and quantifiers); and finally writing proofs. The Coq environment supports advanced notations, proof search and automation, and modular developments. It also provides program extraction towards languages like Ocaml and Haskell for execution of (certified) algorithms [57, 58]. These features are very useful to formalize and reason about complex specifications and programs.

As examples of its applicability, Coq has been used as a framework for formalizing programming environments and designing special platforms for software verification: Leroy and others developed in Coq a certified optimizing compiler for a large subset of the C programming language [59]; Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [60]. Also, as mentioned previously, the Gemalto and Trusted Logic companies obtained the level CC EAL 7 of certification for their formalization, developed in Coq, of the security properties of the JavaCard platform.

We developed our specifications in the Calculus of Inductive Constructions (CIC) [61, 62, 63] using Coq. The CIC is a type theory, in brief, a higher order logic in which the individuals are classified into a hierarchy of types. The types work very much as in strongly typed functional programming languages which means that there are basic elementary types, types defined by induction, like sequences and trees, and function types. An inductive type is defined by its constructors and its elements are obtained as finite combinations of these constructors. Data types are called “Sets” in the CIC (in Coq). When the requirement of finiteness is removed we obtain the possibility of defining infinite structures, called coinductive types, like infinite sequences. On top of this, a higher-order logic is available which serves to predicate on the various data types. The interpretation of the propositions is constructive, i.e. a proposition is defined by specifying what a proof of it is and a proposition is true if and only if a proof of it has been constructed. The type of propositions is called `Prop`.

1.5.2 Notation

We use standard notation for equality and logical connectives. Implication and universal quantification may be encoded in Coq using dependent product, while equality and the other connectives can be defined inductively. Anonymous predicates are introduced using lambda notation, e.g. $(\lambda n : nat.\ n = 0) : nat \to Prop$ is a predicate that when applied to n, is true iff n is zero.

We extensively use record types; a record type definition

$$R \stackrel{\text{def}}{=} [\![\ field_1 : T_1, \ldots, field_n : T_n\ ]\!] \tag{1.1}$$

generates a non-recursive inductive type with just one constructor, namely mkR, and projections functions $field_i : R \to T_i$. We write $\langle a_1, \ldots, a_n \rangle$ instead of $mkR\ a_1 \ldots a_n$ when the type R is obvious from the context. Application of projections functions is abbreviated using dot notation (i.e. $field_i\ r = r.field_i$). Field update of a record r is written as $r.[field_i := v]$; we also use simultaneous field update, which is defined in the usual way. For each field $field_i$ in a record type we define a binary relation $\equiv_{field_i}$ over objects of the type as

$$r_1 \equiv_{field_i} r_2 \stackrel{\text{def}}{=} \forall\ j, j \neq i \to r_1.field_j = r_2.field_j \tag{1.2}$$

We define an inductive relation I by giving introduction rules of the form

$$\frac{P_1 \cdots\ P_m}{I\ x_1 \ldots x_n}\ rule \tag{1.3}$$

where free occurrences of variables are universally quantified. Similarly, inductive types are defined by giving constructors in the following form

$$\begin{array}{rl} T \stackrel{def}{=} & \mid C_1 : A_{1,1} \to \ldots A_{1,n_1} \to T \\ & \vdots \\ & \mid C_m : A_{m,1} \to \ldots A_{m,n_m} \to T \end{array} \tag{1.4}$$

where $C_1 \ldots C_m$ are the constructors of T.

The following parametric (co)inductive types are assumed to be predefined:

- *option* A with constructors $None$: *option* A and $Some : A \to$ *option* A;
- finite lists over A, *list* A. The empty list is denoted by [] and the (infix) constructor that inserts an element a at the front of a list s is denoted by $a \triangleright s$. Finite snoc lists over A, *snocList* A, that is, lists that are constructed by inserting elements at the back, are also used. [] denotes the empty snoc list and $s \triangleleft\ a$ denotes the insertion of an element a at the back of the snoc list s. The symbol $\oplus$ stands for the concatenation operator on lists;
- infinite lists. The type of streams of type A is written $[A]_\infty$. Objects of type $[A]_\infty$ are constructed with the (infix) operator ::, hence $x :: xs$ is of type $[A]_\infty$ whenever x is of type A and xs is of type $[A]_\infty$. Given $s : [A]_\infty$ we let $s[i]$ denote the i-th element of s;
- sets of type A are defined as *set* A, where *set* is an inductive type encoding sets as lists. We use $\{a_0, \ldots, a_n\}_A$ to denote the set of elements $a_0, \ldots, a_n$ of type A. When the type A is obvious from the context, we write $\{a_0, \ldots, a_n\}$. Classical notation is used for set operations $(\cup, \cap, \setminus, \in, \subseteq)$.

Moreover, we make an extensive use of partial maps, and bounded partial maps. The type of partial maps from objects of type A into objects of type B is written $A\ \mapsto\ B$, and the type of partial maps from A to B whose domain is of size smaller or equal to

k (where k is a natural number) is written as $A \mapsto_k B$. Application of a map m on an object a of type A is denoted $m[a]$ and map update is written $m[a := b]$, where b overwrites the value, if any, associated to the key a.

In the formal constructions presented in Section 2.5 we use inductive types that have constructors tagged either as *valid* or *invalid*. In that section we adopt the convention that a type with the same name but prefixed with the *valid* keyword is the subtype consisting of terms derivable using only constructors tagged as *valid*.

1.6 Document outline

The remainder of this document is organized as follows. In Chapter 2 we study and compare formally several variants of the security model specified by MIDP to access sensitive resources of a device. In Chapter 3 we present the formalization of an idealized model of a hypervisor, and we establish (formally) that the hypervisor ensures strong isolation properties between the different operating systems, and guarantees that requests from guest operating systems are eventually attended. We then develop a certified implementation of a hypervisor —in the programming language of Coq— on an extended memory model with a formalization of the cache and TLB, and we analyze also security properties for the extended model with execution errors. In Chapter 4 we present a minimalistic specification of a DNSSEC model which provides the grounds needed to formally state and verify security properties concerning the chain of trust of the DNSSEC tree. Finally, the conclusions of the research are formulated in Chapter 5, along with some statistics on the formal developments in Coq and the possible lines of future work.

Chapter 2

Formal analysis of security models for mobile devices

2.1 Introduction

JME technology provides integral mechanisms that guarantee security properties for mobile devices, defining a security model which restricts access to sensitive functions for badly written or maliciously written applications.

The architecture of JME is based upon two layers above the virtual machine: the configuration layer and the profile layer. The Connected Limited Device Configuration (CLDC) is a minimum set of class libraries to be used in devices with low processors, limited working memory and capacity to establish low broad band communications. This configuration is complemented with the Mobile Information Device Profile (MIDP) to obtain a run-time environment suitable to mobile devices such as mobile phones, personal digital assistants and pagers. The profile also defines an application life cycle, a security model and APIs that provide the functionality required by mobile applications, such as networking, user interface, push activation, and local persistent storage.

2.1.1 MIDP security model

A security model at the platform level, based on sets of permissions to access the functions of the device, is defined in the first version of MIDP (version 1.0) and refined in the second version (version 2.0). In the third version (version 3.0) a new dimension to the model is introduced: the security at the application level. The security model at the application level is based on authorizations, so that an application can access the resources shared by another application.

In this section we describe the two security levels considered by MIDP in its different versions.

Security at the platform level

In the original MIDP 1.0 specification, any application not installed by the device manufacturer or a service provider runs in a sandbox that prohibits access to security sensitive APIs or functions of the device (e.g. push activation). Although this sandbox security model effectively prevents any rogue application from jeopardizing the security of the

device, it is excessively restrictive and does not allow many useful applications to be deployed after issuance.

In contrast, MIDP 2.0 introduces a new security model at the platform level based on the concept of protection domain. A protection domain specifies a set of permissions and the modes in which those permissions can be granted to applications bound to the domain. Each API or function on the device may define permissions in order to prevent it from being used without authorization.

Permissions can be either granted in an unconditional way, and used without requiring any intervention from the user, or in a conditional way, specifying the frequency with which user interaction will be required to renew the permission. More concretely, for each permission a protection domain specifies one of three possible modes:

- *blanket*: permission is granted for the whole application life cycle;
- *session*: permission is granted until the application is closed;
- *oneshot*: permission is granted for one single use.

The set of permissions effectively granted to an application is determined by the permissions declared in its descriptor, the protection domain it is bound to, and the answers received from the user to previous requests. A permission is granted to an application if it is declared in its application descriptor and either its protection domain unconditionally grants the permission, or the user was previously asked to grant the permission and its authorization remains valid.

The MIDP 2.0 model distinguishes between untrusted and trusted applications. An untrusted application is one whose origin and integrity can not be determined. These applications, also called unsigned, are bound to a protection domain with permissions equivalent to those in a MIDP 1.0 sandbox. On the other hand, a trusted application is identified and verified by means of cryptographic signatures and certificates based on the X.509 Public Key Infrastructure [64]. These signed applications can be bound to more specific and permissive protection domains. Thus, this model enables applications developed by trusted third parties to be downloaded and installed after issuance of the device without compromising its security.

Security at the application level

In MIDP 3.0 applications which are bound to different protection domains may share data, components, and events. The *Record Management System* (RMS) API specifies methods that make it possible for record stores from one application to be shared with other applications. With the *Inter-MIDlet Communication* (IMC) protocol an application can access a shareable component from another one running in a different execution environment. Applications that register listeners or register themselves will only be launched in response to events triggered by authorized applications.

By restricting access to shareable resources of an application, such as data, runtime IMC communication and events, the MIDP 3.0 security model introduces a new dimension at the application level. This new dimension introduces the concept of *access authorization*, which enables an application to individually control access requests from other applications.

An application that intends to restrict access to its shareable resources must declare access authorizations in its application descriptor. There are four different types of access

authorization declarations involving one or more of the following application credentials: domain name, vendor name, and signing certificates. These four types apply respectively to:

1. applications bound to a specific domain;
2. applications from a certain vendor with a certified signature;
3. applications from a certain vendor but whithout a certified signature;
4. applications signed under a given certificate.

When an application tries to access the shareable resources of another application, its credentials are compared with those required by the access authorization declaration. If there is a mismatch the application level access is denied.

2.1.2 Contents of the chapter

This chapter builds upon and extends a number of previously published papers [65, 66, 67, 68]. The rest of the chapter is organized as follows: Section 2.2 overviews the formalization of the MIDP 2.0 security model, presents some of its verified properties and proposes a methodology to refine the specification and obtain an executable prototype. Section 2.3 describes a high level formal specification of an access controller for JME-MIDP 2.0, and presents an algorithm that has been proved correct with respect to that specification. Section 2.4 develops extensions of the formal specification presented in Section 2.2 concerning the changes introduced by MIDP 3.0, and puts forward some weaknesses of the security mechanisms introduced in that (latest) version of the profile. Section 2.5 presents a framework suitable for defining, analyzing, and comparing the access control policies that can be enforced by (variants of) the security models considered in this work, and to prove desirable properties they should satisfy. Then Section 2.6 discusses related work and finally Section 2.7 concludes with a summary of the chapter.

2.1.3 Formalization

The full development is available for download at

`http://www.fing.edu.uy/inco/grupos/gsi/sources/midp`

2.2 Formally verifying security properties of MIDP 2.0

In this section we overview the formalization of the MIDP 2.0 security model presented in [65]. We begin, in Section 2.2.1, by introducing some types and constants used in the remainder of the formalization, we then define the set of valid device states and security-related events, give a transition semantics for events based on pre- and post-conditions and define the concept of a session. Section 2.2.2 reports on some security properties that we have verified, while Section 2.2.3 presents and discusses a proposal for constructing a refinement of the formal specification to obtain an executable prototype.

```
MIDlet-Jar-URL: http://www.foo.com/foo.jar
MIDlet-Jar-Size: 90210
MIDlet-Name: Organizer
MIDlet-Vendor: Foo Software
MIDlet-Version: 2.1
MIDlet-1: Alarm, alarm.png, organizer.Alarm
MIDlet-2: Agenda, agenda.png, organizer.Agenda
MIDlet-Permissions: javax.microedition.io.Connector.socket,
    javax.microedition.io.PushRegistry
MIDlet-Permissions-Opt: javax.microedition.io.Connector.http
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
MIDlet-Certificate-1-1: MIICgDCCAekCBEH11wYwDQ...
MIDlet-Jar-RSA-SHA1: pYRJ8Qlu5ITBLxcAUzYXDKnmg...
```

Figure 2.1: A typical MIDlet descriptor, specifying the required and optional permission of suite for a suite with two applications

2.2.1 Formalization of the MIDP 2.0 security model

Sets and constants

In MIDP, applications (usually called MIDlets) are packaged and distributed as suites. A suite may contain one or more MIDlets and is distributed as two files, an application descriptor file and an archive file that contains the actual Java classes and resources. A suite that uses protected APIs or functions should declare the corresponding permissions in its descriptor either as required for its correct functioning or as optional. The content of a typical MIDlet descriptor is shown in Figure 2.1.

Let $Permission$ be the total set of permissions defined by every protected API or function on the device and $Domain$ the set of all protection domains. Let us introduce, as a way of referring to individual MIDlet suites, the set $SuiteID$ of valid suite identifiers. We will represent a descriptor as a record composed of two predicates, *required* and *optional*, that identify respectively the set of permissions declared as required and those declared as optional by the corresponding suite,

$$Descriptor \stackrel{\text{def}}{=} [\![\begin{array}{l} required : Permission \to Prop, \\ optional : Permission \to Prop \end{array}]\!] \tag{2.1}$$

A record type is used to represent an installed suite, with fields for its identifier, associated protection domain and descriptor,

$$Suite \stackrel{\text{def}}{=} [\![\begin{array}{l} id : SuiteID, \\ domain : Domain, \\ descriptor : Descriptor \end{array}]\!] \tag{2.2}$$

Permissions may be granted by the user to an active MIDlet suite in either of three modes, for a single use (oneshot), as long as the suite is running (session), or as long as the suite remains installed (blanket). Let $Mode$ be the enumerated set of user interaction modes $\{oneshot, session, blanket\}$ and $\leq_{\mathrm{m}}$ an order relation such that

$$oneshot \leq_{\mathrm{m}} session \leq_{\mathrm{m}} blanket \tag{2.3}$$

We will assume for the rest of the formalization that the security policy of the protection domains on the device is an anonymous constant of type

$$Policy \stackrel{\text{def}}{=} [\![\quad \begin{array}{ll} allow & : Domain \to Permission \to Prop, \\ user & : Domain \to Permission \to Mode \to Prop \end{array}]\!] \tag{2.4}$$

which for each domain specifies at most one mode for a given permission,

$$\begin{array}{l} (\forall\ d\ p, allow\ d\ p \to \forall\ m, \neg user\ d\ p\ m) \wedge \\ (\forall\ d\ p\ m, user\ d\ p\ m \to \neg allow\ d\ p\ \wedge \forall\ m', user\ d\ p\ m' \to m = m') \end{array} \tag{2.5}$$

and such that $allow\ d\ p$ holds when domain d unconditionally grants the permission p and $user\ d\ p\ m$ holds when domain d grants permission p with explicit user authorization and maximum allowable mode m (w.r.t. $\leq_{\mathrm{m}}$). The permissions effectively granted to a MIDlet suite are the intersection of the permissions requested in its descriptor with the union of the permissions given unconditionally by its domain and those given explicitly by the user.

Device state

To reason about the MIDP 2.0 security model most details of the device state may be abstracted; it is sufficient to specify the set of installed suites, the permissions granted or revoked to them and the currently active suite in case there is one. The active suite, and the permissions granted or revoked to it for the session are grouped into a record structure

$$SessionInfo \stackrel{\text{def}}{=} [\![\quad \begin{array}{l} id : SuiteID, \\ granted : Permission \to Prop, \\ revoked : Permission \to Prop \end{array}]\!] \tag{2.6}$$

The abstract device state is described as a record of type

$$State \stackrel{\text{def}}{=} [\![\quad \begin{array}{l} suite : Suite \to Prop, \\ session : option\ SessionInfo, \\ granted : SuiteID \to Permission \to Prop, \\ revoked : SuiteID \to Permission \to Prop \end{array}]\!] \tag{2.7}$$

where *suite* is the characteristic predicate of the set of installed suites.

Example 2.2.1. *Consider a MIDlet that periodically connects to a webmail service using HTTPS when possible or HTTP otherwise, and alerts the user whenever they have new mail. The suite containing this MIDlet should declare in its descriptor as required permissions p_push, for accessing the PushRegistry (for timer-based activation), and p_http, for using the HTTP protocol API. It should also declare as optional the permission p_https for accessing the HTTPS protocol API. Suppose that upon installation, the suite (whose identifier is id) is recognized as trusted and is thus bound to a protection domain dom that allows access to the PushRegistry API unconditionally but by default requests user authorization for opening every HTTP or HTTPS connection. Suppose also that the domain allows the user to grant the MIDlet the permission for opening further connections as long as the suite remains installed. Then, the security policy satisfies:*

$$\begin{array}{l} allow\ dom\ p\_push \wedge \\ user\ dom\ p\_http\ blanket \wedge \\ user\ dom\ p\_https\ blanket \end{array} \tag{2.8}$$

If st is the state of the device, the suite is represented by some ms : Suite such that st.suite ms and ms.id = id hold. Its descriptor ms.descriptor satisfies

$$\begin{array}{l} ms.descriptor.required\ p\_push \wedge \\ ms.descriptor.required\ p\_http \wedge \\ ms.descriptor.optional\ p\_https \end{array} \tag{2.9}$$

The MIDlet will have unlimited access to the PushRegistry applet, but will have to request user authorization every time it makes a new connection. The user may choose at any time to authorize further connections by granting the corresponding permission in blanket mode, thus avoiding being asked for authorization each time the applet communicates with the webmail service.

The remainder of this subsection enumerates the conditions that must hold for an element $s : State$ in order to represent a valid state for a device.

1. A MIDlet suite can be installed and bound to a protection domain only if the set of permissions declared as required in its descriptor are a subset of the permissions the domain offers (with or without user authorization). This compatibility relation between $des : Descriptor$ and $dom : Domain$ can be stated formally as follows,

$$\begin{array}{l} des \wr dom \stackrel{\text{def}}{=} \\ \quad \forall\ p : Permission, des.required\ p \rightarrow \\ \quad\quad allow\ dom\ p\ \vee \exists\ m : Mode, user\ dom\ p\ m \end{array} \tag{2.10}$$

   Every installed suite must be compatible with its associated protection domain,

$$\begin{array}{l} SuiteCompatible \stackrel{\text{def}}{=} \\ \quad \forall\ ms : Suite, s.suite\ ms \rightarrow \\ \quad\quad ms.descriptor \wr ms.domain \end{array} \tag{2.11}$$

2. Whenever there exists a running session, the suite identifier in $s.session$ must correspond to an installed suite,

$$\begin{array}{l} CurrentInstalled \stackrel{\text{def}}{=} \\ \quad \forall\ ses : SessionInfo, s.session = Some\ ses \rightarrow \\ \quad\quad \exists\ ms : Suite, s.suite\ ms \wedge ms.id = ses.id \end{array} \tag{2.12}$$

3. The set of permissions granted for the session must be a subset of the permissions requested in the application descriptor of the active suite. In addition, the associated protection domain policy must allow those permissions to be granted at least in *session* mode,

$$\begin{array}{l} ValidSessionGranted \stackrel{\text{def}}{=} \\ \quad \forall\ ses : SessionInfo, s.session = Some\ ses \rightarrow \\ \quad\quad \forall\ p : Permission, ses.granted\ p \rightarrow \\ \quad\quad\quad \forall\ ms : Suite, s.suite\ ms \rightarrow ms.id = ses.id \rightarrow \\ \quad\quad\quad\quad (ms.descriptor.required\ p\ \vee ms.descriptor.optional\ p) \wedge \\ \quad\quad\quad\quad \exists\ m : Mode, user\ ms.domain\ p\ m\ \wedge session \leq_{\mathrm{m}} m \end{array} \tag{2.13}$$

Table 2.1: Security-related events

Name	Description	Type
start	Start of session	$SuiteID \to Event$
terminate	End of session	$Event$
request	Permission request	$Permission \to option\ UserAnswer \to Event$
install	MIDlet suite installation	$SuiteID \to Descriptor \to Domain \to Event$
remove	MIDlet suite removal	$SuiteID \to Event$

4. Every installed suite shall have a unique identifier,

$$\begin{aligned} &UniqueSuiteID \stackrel{\text{def}}{=} \\ &\quad \forall\ ms_1\ ms_2 : Suite, s.suite\ ms_1 \to s.suite\ ms_2 \to \\ &\qquad ms_1.id = ms_2.id \to ms_1 = ms_2 \end{aligned} \tag{2.14}$$

5. For every installed suite with identifier *id*, the predicate *s.granted id* should be valid with respect to its descriptor and associated protection domain,

$$\begin{aligned} &ValidGranted \stackrel{\text{def}}{=} \\ &\quad \forall\ (p : Permission)(id : SuiteID), s.granted\ id\ p \to \\ &\qquad \forall\ ms : Suite, s.suite\ ms \to ms.sid = id \to \\ &\qquad\quad (ms.descriptor.required\ p\ \vee ms.descriptor.optional\ p) \wedge \\ &\qquad\quad user\ ms.domain\ p\ blanket \end{aligned} \tag{2.15}$$

6. A granted permission shall not be revoked at the same time and viceversa. This condition is formalized by the predicate $ValidGrantedRevoked$.

Events

We define a set *Event* for those events that are relevant to our abstraction of the device state (Table 2.1). The user may be presented with the choice between accepting or refusing an authorization request, specifying the period of time their choice remains valid. The outcome of a user interaction is represented using the type $UserAnswer$ with constructors

$$ua\_allow, ua\_deny : Mode \to UserAnswer \tag{2.16}$$

The behaviour of the events is specified by their pre- and postconditions given by the predicates *Pre* and *Post* respectively. Preconditions (Table 2.2) are defined in terms of the device state while postconditions (Table 2.3) are defined in terms of the before and after states and an optional response which is only meaningful for the *request* event and indicates whether the requested operation is authorized,

$$\begin{aligned} &Pre : State \to Event \to Prop \\ &Post : State \to State \to option\ Response \to Event \to Prop \end{aligned} \tag{2.17}$$

Example 2.2.2. *Consider an event representing a permission request for which the user denies the authorization. Such an event can only occur when the active suite has*

Table 2.2: Event preconditions

$$
\begin{array}{l}
Pre\ s\ (start\ id) \stackrel{\text{def}}{=} \\
\quad s.session = None\ \wedge \exists\ ms : Suite, s.suite\ ms \wedge ms.id = id \\
\\
Pre\ s\ terminate \stackrel{\text{def}}{=} \\
\quad s.session \neq None \\
\\
Pre\ s\ (install\ id\ des\ dom) \stackrel{\text{def}}{=} \\
\quad des \wr dom \wedge \forall\ ms : Suite, s.suite\ ms \rightarrow ms.id \neq id. \\
\\
Pre\ s\ (remove\ id) \stackrel{\text{def}}{=} \\
\quad (\forall\ ses : SessionInfo, s.session = Some\ ses\ \rightarrow ses.id \neq id) \wedge \\
\quad \exists\ ms : Suite, s.suite\ ms \wedge ms.id = id
\end{array}
$$

Table 2.3: Event postconditions

$$
\begin{array}{l}
Post\ s\ s'\ r\ (start\ id) \stackrel{\text{def}}{=} \\
\quad r = None\ \wedge s \equiv_{session} s'\ \wedge\ \exists\ ses', s'.session = Some\ ses'\ \wedge ses'.id = id \wedge \\
\quad \forall\ p : Permission, \neg ses'.granted\ p\ \wedge \neg ses'.revoked\ p \\
\\
Post\ s\ s'\ r\ terminate \stackrel{\text{def}}{=} \\
\quad r = None\ \wedge s \equiv_{session} s'\ \wedge s'.session = None \\
\\
Post\ s\ s'\ r\ (install\ id\ des\ dom) \stackrel{\text{def}}{=} \\
\quad r = None\ \wedge\ (\forall\ ms : Suite, s.suite\ ms \rightarrow s'.suite\ ms) \wedge \\
\quad (\forall\ ms : Suite, s'.suite\ ms \rightarrow s.suite\ ms\ \vee ms = \langle id, dom, des \rangle) \wedge \\
\quad s'.suite\ \langle id, dom, des \rangle\ \wedge s'.session = s.session \wedge \\
\quad (\forall\ p : Permission, \neg s'.granted\ id\ p \wedge \neg s'.revoked\ id\ p) \wedge \\
\quad (\forall\ id_1 : SuiteID, id_1 \neq id \rightarrow \\
\qquad s'.granted\ id_1 = s.granted\ id_1\ \wedge s'.revoked\ id_1 = s.revoked\ id_1) \\
\\
Post\ s\ s'\ r\ (remove\ id) \stackrel{\text{def}}{=} \\
\quad r = None\ \wedge s \equiv_{suite} s' \wedge \\
\quad (\forall\ ms : Suite, s.suite\ ms \rightarrow ms.id \neq id \rightarrow s'.suite\ ms) \wedge \\
\quad (\forall\ ms : Suite, s'.suite\ ms \rightarrow s.suite\ ms \wedge ms.id \neq id)
\end{array}
$$

$$
\begin{aligned}
&Pre\ s\ (request\ p\ None) :=\\
&\quad \exists\ ses, s.session = Some\ ses\ \wedge\ \forall\ ms, s.suite\ ms \rightarrow ms.sid = ses.sid \rightarrow\\
&\qquad (ms.descriptor.required\ p\ \vee\ ms.descriptor.optional\ p) \rightarrow\\
&\qquad \forall\ m, user\ ms.domain\ p\ m \rightarrow\\
&\qquad\quad ses.granted\ p\ \vee\ ses.revoked\ p\ \vee\\
&\qquad\quad s.granted\ ses.sid\ p \vee\ s.revoked\ ses.sid\ p\\
&Post\ s\ s'\ r\ (request\ p\ None) :=\\
&\quad s = s'\ \wedge \forall\ ses\ ms, s.session = Some\ ses \rightarrow s.suite\ ms \rightarrow ms.sid = ses.sid \rightarrow\\
&\qquad (\neg ms.descriptor.required\ p \rightarrow \neg ms.descriptor.optional\ p \rightarrow\\
&\qquad\quad r = Some\ denied) \wedge\\
&\qquad (ms.descriptor.required\ p\ \vee ms.descriptor.optional\ p \rightarrow\\
&\qquad\quad (allow\ ms.domain\ p \rightarrow r = Some\ allowed) \wedge\\
&\qquad\quad (ses.granted\ p \rightarrow r = Some\ allowed) \wedge\\
&\qquad\quad (s.granted\ ses.sid\ p \rightarrow r = Some\ allowed) \wedge\\
&\qquad\quad (ses.revoked\ p \rightarrow r = Some\ denied) \wedge\\
&\qquad\quad (s.revoked\ ses.sid\ p \rightarrow r = Some\ denied))
\end{aligned}
$$

Figure 2.2: Specification of event *request* when user intervention is not required

declared the requested permission in its descriptor and is bound to a protection domain that specifies a user interaction mode for that permission (otherwise, the request would be immediately accepted or rejected). Furthermore, the requested permission must not have been revoked or granted for the rest of the session or the rest of the suite's life,

$$
\begin{aligned}
&Pre\ s\ (request\ p\ (Some\ (ua\_deny\ m))) \stackrel{def}{=}\\
&\quad \exists\ ses : SessionInfo, s.session = Some\ ses\ \wedge\\
&\qquad \forall\ ms : Suite, s.suite\ ms \rightarrow ms.id = ses.id \rightarrow\\
&\qquad\quad (ms.descriptor.required\ p\ \vee ms.descriptor.optional\ p) \wedge\\
&\qquad\quad (\exists\ m_1 : Mode, user\ ms.domain\ p\ m_1) \wedge\\
&\qquad\quad \neg ses.granted\ p\ \wedge\ \neg ses.revoked\ p \wedge\\
&\qquad\quad \neg s.granted\ ses.id\ p\ \wedge \neg s.revoked\ ses.id\ p
\end{aligned}
\tag{2.18}
$$

When $m = session$, the user revokes the permission for the whole session, therefore, the response denies the permission and the state is updated accordingly,

$$
\begin{aligned}
&Post\ s\ s'\ r\ (request\ p\ (Some\ (ua\_deny\ session))) \stackrel{def}{=}\\
&\quad r = Some\ denied\ \wedge s \equiv_{session} s'\ \wedge\\
&\quad \forall\ ses : SessionInfo, s.session = Some\ ses \rightarrow\\
&\qquad \exists\ ses' : SessionInfo,\\
&\qquad\quad s'.session = Some\ ses'\ \wedge ses' \equiv_{revoked} ses\ \wedge ses'.revoked\ p\ \wedge\\
&\qquad\quad (\forall\ q : Permission, q \neq p \rightarrow ses'.revoked\ q = ses.revoked\ q)
\end{aligned}
\tag{2.19}
$$

The full specification of event *request* is divided into three cases: i) when user intervention is not required (Figure 2.2). The permission must have been previously granted or revoked by the rest of the session or indefinitely; ii) when intervention is required and the user accepts the request (Figure 2.3). In this case, the state changes if the user gives permission to the suite by session or indefinitely; and iii) when intervention

$$
\begin{aligned}
&\mathit{Pre}\ s\ (\mathit{request}\ p\ (\mathit{Some}\ (\mathit{ua\_allow}\ m))) :=\\
&\quad \exists\ \mathit{ses}, s.\mathit{session} = \mathit{Some}\ \mathit{ses}\ \wedge\ \forall\ \mathit{ms}, s.\mathit{suite}\ \mathit{ms} \rightarrow \mathit{ms.sid} = \mathit{ses.sid} \rightarrow\\
&\qquad (\mathit{ms.descriptor.required}\ p\ \vee\ \mathit{ms.descriptor.optional}\ p) \wedge\\
&\qquad \neg \mathit{ses.granted}\ p\ \wedge\ \neg \mathit{ses.revoked}\ p \wedge\\
&\qquad \neg s.\mathit{granted}\ \mathit{ses.sid}\ p\ \wedge\ \neg s.\mathit{revoked}\ \mathit{ses.sid}\ p \wedge\\
&\qquad (\exists\ m', \mathit{user}\ \mathit{ms.domain}\ p\ m') \wedge\\
&\qquad (\forall\ m', \mathit{user}\ \mathit{ms.domain}\ p\ m' \rightarrow m \leq_m m')\\
&\mathit{Post}\ s\ s'\ r\ (\mathit{request}\ (\mathit{Some}\ (\mathit{ua\_allow}\ \mathit{oneshot}))) := r = \mathit{Some}\ \mathit{allowed}\ \wedge\ s = s'\\
&\mathit{Post}\ s\ s'\ r\ (\mathit{request}\ (\mathit{Some}\ (\mathit{ua\_allow}\ \mathit{session}))) :=\\
&\quad r = \mathit{Some}\ \mathit{allowed}\ \wedge\ s \equiv_{\mathit{session}} s' \wedge\\
&\quad \forall\ \mathit{ses}, s.\mathit{session} = \mathit{Some}\ \mathit{ses} \rightarrow\\
&\qquad \exists\ \mathit{ses}', s'.\mathit{session} = \mathit{Some}\ \mathit{ses}'\ \wedge\ \mathit{ses}' \equiv_{\mathit{granted}} \mathit{ses}\ \wedge\ \mathit{ses}'.\mathit{granted}\ p \wedge\\
&\qquad (\forall\ q, q \neq p \rightarrow \mathit{ses}'.\mathit{granted}\ q = \mathit{ses.granted}\ q)\\
&\mathit{Post}\ s\ s'\ r\ (\mathit{request}\ (\mathit{Some}\ (\mathit{ua\_allow}\ \mathit{blanket}))) :=\\
&\quad r = \mathit{Some}\ \mathit{allowed}\ \wedge\ s \equiv_{\mathit{granted}} s'\ \wedge\ \forall\ \mathit{ses}, s.\mathit{session} = \mathit{Some}\ \mathit{ses} \rightarrow\\
&\qquad s'.\mathit{granted}\ \mathit{ses.sid}\ p \wedge\\
&\qquad (\forall\ q, q \neq p \rightarrow s'.\mathit{granted}\ \mathit{ses.sid}\ q = s.\mathit{granted}\ \mathit{ses.sid}\ q) \wedge\\
&\qquad (\forall\ \mathit{id}, \mathit{id} \neq \mathit{ses.sid} \rightarrow s'.\mathit{granted}\ \mathit{id} = s.\mathit{granted}\ \mathit{id})
\end{aligned}
$$

Figure 2.3: Specification of event *request* when intervention is required and the user accepts the request

$$
\begin{aligned}
&\mathit{Pre}\ s\ (\mathit{request}\ p\ (\mathit{Some}\ (\mathit{ua\_deny}\ m))) :=\\
&\quad \exists\ \mathit{ses}, s.\mathit{session} = \mathit{Some}\ \mathit{ses}\ \wedge\ \forall\ \mathit{ms}, s.\mathit{suite}\ \mathit{ms} \rightarrow \mathit{ms.sid} = \mathit{ses.sid} \rightarrow\\
&\qquad (\mathit{ms.descriptor.required}\ p\ \vee\ \mathit{ms.descriptor.optional}\ p) \wedge\\
&\qquad \neg \mathit{ses.granted}\ p\ \wedge\ \neg \mathit{ses.revoked}\ p \wedge\\
&\qquad \neg s.\mathit{granted}\ \mathit{ses.sid}\ p\ \wedge\ \neg s.\mathit{revoked}\ \mathit{ses.sid}\ p \wedge\\
&\qquad (\exists\ m', \mathit{user}\ \mathit{ms.domain}\ p\ m')\\
&\mathit{Post}\ s\ s'\ r\ (\mathit{request}\ (\mathit{Some}\ (\mathit{ua\_deny}\ \mathit{oneshot}))) := r = \mathit{Some}\ \mathit{denied}\ \wedge\ s = s'\\
&\mathit{Post}\ s\ s'\ r\ (\mathit{request}\ (\mathit{Some}\ (\mathit{ua\_deny}\ \mathit{session}))) :=\\
&\quad r = \mathit{Some}\ \mathit{denied}\ \wedge\ s \equiv_{\mathit{session}} s' \wedge\\
&\quad \forall\ \mathit{ses}, s.\mathit{session} = \mathit{Some}\ \mathit{ses} \rightarrow\\
&\qquad \exists\ \mathit{ses}', s'.\mathit{session} = \mathit{Some}\ \mathit{ses}'\ \wedge\ \mathit{ses}' \equiv_{\mathit{revoked}} \mathit{ses}\ \wedge\ \mathit{ses}'.\mathit{revoked}\ p \wedge\\
&\qquad (\forall\ q, q \neq p \rightarrow \mathit{ses}'.\mathit{revoked}\ q = \mathit{ses.revoked}\ q)\\
&\mathit{Post}\ s\ s'\ r\ (\mathit{request}\ (\mathit{Some}\ (\mathit{ua\_deny}\ \mathit{blanket}))) :=\\
&\quad r = \mathit{Some}\ \mathit{denied}\ \wedge\ s \equiv_{\mathit{revoked}} s'\ \wedge\ \forall\ \mathit{ses}, s.\mathit{session} = \mathit{Some}\ \mathit{ses} \rightarrow\\
&\qquad s'.\mathit{revoked}\ \mathit{ses.sid}\ p \wedge\\
&\qquad (\forall\ q, q \neq p \rightarrow s'.\mathit{revoked}\ \mathit{ses.sid}\ q = s.\mathit{revoked}\ \mathit{ses.sid}\ q) \wedge\\
&\qquad (\forall\ \mathit{id}, \mathit{id} \neq \mathit{ses.sid} \rightarrow s'.\mathit{revoked}\ \mathit{id} = s.\mathit{revoked}\ \mathit{id})
\end{aligned}
$$

Figure 2.4: Specification of event *request* when intervention is required and the user denies the request

$$s_0 \xrightarrow{start\ id/r_1} s_1 \xrightarrow{e_2/r_2} s_2 \xrightarrow{e_3/r_3} \cdots \xrightarrow{e_{n-1}/r_{n-1}} s_{n-1} \xrightarrow{terminate/r_n} s_n$$

Figure 2.5: A session for a suite with identifier *id*

is required and the user denies the request (Figure 2.4). The state changes if the user denies permission to the suite by session or indefinitely.

One-step execution

The behavioural specification of the execution of an event is given by the $\hookrightarrow$ relation with the following introduction rules:

$$\frac{\neg Pre\ s\ e}{s \xrightarrow{e/None} s}\ npre \qquad \frac{Pre\ s\ e \quad Post\ s\ s'\ r\ e}{s \xrightarrow{e/r} s'}\ pre \tag{2.20}$$

Whenever an event occurs for which the precondition does not hold, the state must remain unchanged. Otherwise, the state may change in such a way that the event postcondition is established. The notation $s \xrightarrow{e/r} s'$ may be read as "the execution of the event e in state s results in a new state s' and produces a response r".

Sessions

A session is the period of time spanning from a successful *start* event to a *terminate* event, in which a single suite remains active. A session for a suite with identifier *id* (Figure 2.5) is determined by an initial state s_0 and a sequence of steps $\langle e_i, s_i, r_i \rangle$ $(i = 1, \ldots, n)$ such that the following conditions hold,

- $e_1 = start\ id$;
- $Pre\ s_0\ e_1$;
- $\forall\ i \in \{2, \ldots, n-1\}, e_i \neq terminate$;
- $e_n = terminate$;
- $\forall\ i \in \{1, \ldots, n\}, s_{i-1} \xrightarrow{e_i/r_i} s_i$.

A partial session is a *session* for which the *terminate* event has not yet been elicited; it is defined inductively by the following rules,

$$\frac{Pre\ s_0\ (start\ id) \quad s_0 \xrightarrow{start\ id/r_1} s_1}{PSession\ s_0\ ([\,] \lhd \langle start\ id, s_1, r_1 \rangle)}\ psession\_start \tag{2.21}$$

$$\frac{PSession\ s_0\ (ss \lhd last) \quad e \neq terminate \quad last.s \xrightarrow{e/r} s'}{PSession\ s_0\ (ss \lhd last \lhd \langle e, s', r \rangle)}\ psession\_app \tag{2.22}$$

Now, sessions can be easily defined as follows,

$$\frac{PSession\ s_0\ (ss \lhd last) \quad last.s \xrightarrow{terminate/r} s'}{Session\ s_0\ (ss \lhd last \lhd \langle terminate, s', r \rangle)}\ session\_terminate \tag{2.23}$$

2.2.2 Verification of security properties

This section is devoted to establishing relevant security properties of the model. Proofs are merely outlined; however, all proofs have been formalized in Coq and are available as part of the full specification.

An invariant of one-step execution

We call one-step invariant a property that remains true after the *execution* of every event if it is true before. We show next that the validity of the device state, as defined in Section 2.2.1, is a one-step invariant of our specification.

Theorem 2.2.3. *Let $Valid$ be a predicate over State defined as the conjunction of the validity conditions in Section 2.2.1. For any $s\ s' : State$, $r : option\ Response$ and $e : Event$, if $Valid\ s$ and $s \xrightarrow{e/r} s'$ hold, then $Valid\ s'$ also holds.*

Proof. By case analysis on $s \xrightarrow{e/r} s'$. When $Pre\ s\ e$ does not hold, $s = s'$ and s' is valid because s is valid. Otherwise, $Post\ s\ s'\ r\ e$ must hold and we proceed by case analysis on e. We will only show the case $request\ p\ (Some\ (ua\_deny\ session))$, obtained after further case analysis on a when $e = request\ p\ (Some\ a)$.

The postcondition (2.19) entails that $s \equiv_{session} s'$, that the session remains active, and that $ses' \equiv_{revoked} ses$. Therefore, the set of installed suites remains unchanged ($s'.suite = s.suite$), the set of permissions granted for the session does not change ($ses'.granted = ses.granted$) and neither does the set of permissions granted or revoked in *blanket* mode ($s'.granted = s.granted$, $s'.revoked = s.revoked$). From these equalities, every validity condition of the state s' except $ValidGrantedRevoked\ s'$ follows immediately from the validity of s. We next prove $ValidGrantedRevoked\ s'$.

We know from the postcondition of the event that

$$\forall\ q, q \neq p \rightarrow ses'.revoked\ q = ses.revoked\ q \tag{2.24}$$

Let q be any permission. If $q \neq p$, then from (2.24) follows $ses'.revoked\ q = ses.revoked\ q$ and because q was not granted and revoked simultaneously before the event, neither it is afterwards. If $q = p$, then we know from the precondition (2.18) that p was not granted before and thus it is not granted afterwards. This proves $ValidGrantedRevoked\ s'$ and together with the previous results, $Valid\ s'$. □

Session invariants

We call session invariant a step property that holds for the rest of a session once it is established in a step. Let P be a predicate over T, we define $all\ P$ as an inductive predicate over $snocList\ T$ by the following rules:

$$\frac{}{all\ P\ [\,]}\ all\_nil \qquad \frac{all\ P\ ss \quad P\ s}{all\ P\ (ss \triangleleft s)}\ all\_snoc \tag{2.25}$$

Theorem 2.2.4. *Let s_0 be a valid state and ss a partial session starting from s_0, then every state in ss is valid,*

$$all\ (\lambda\ step\ .\ Valid\ step.s)\ ss \tag{2.26}$$

Proof. By induction on the structure of $PSession\ s_0\ ss$.

- When constructed using *psession_start*, ss has the form $[\,] \lhd \langle start\ id, s_1,\ r_1 \rangle$ and $s_0 \xrightarrow{start\ id/r_1} s_1$ holds. We must prove

$$all\ (\lambda\ step\ .\ Valid\ step.s)\ ([\,] \lhd \langle start\ id, s_1,\ r_1 \rangle) \tag{2.27}$$

  By applying *all_app* and then *all_nil* the goal is simplified to $Valid\ s_1$ and is proved from $s_0 \xrightarrow{start\ id/r_1} s_1$ and $Valid\ s_0$ by applying Theorem 2.2.3.

- When it is constructed using *psession_app*, ss has the form $ss_1 \lhd last \lhd \langle e, s', r \rangle$ and $last.s \xrightarrow{e/r} s'$ holds. The induction hypothesis is

$$all\ (\lambda\ step\ .\ Valid\ step.s)\ (ss_1 \lhd last) \tag{2.28}$$

  and we must prove $all\ (\lambda\ step\ .\ Valid\ step.s)\ (ss_1 \lhd last \lhd \langle e, s', r \rangle)$. By applying *all_app* and then (2.28) the goal is simplified to $Valid\ s'$. From (2.28) we know that $last.s$ is a valid state. The goal is proved from $last.s \xrightarrow{e/r} s'$ and $Valid\ last.s$ by applying Theorem 2.2.3.

□

The above theorem may be easily extended from partial sessions to sessions using Theorem 2.2.3 one more time. State validity is just a particular property that is true for a partial session once it is established, the result can be generalized for other properties as shown in the following lemma.

Lemma 2.2.5. *For any property P of a step satisfying*

$$\begin{aligned} &\forall\ (s\ s' : State)(r\ r' : option\ Response)(e\ e' : Event), \\ &\quad e' \neq terminate \rightarrow s \xrightarrow{e'/r'} s' \rightarrow P\ \langle e, s, r \rangle \rightarrow P\ \langle e', s', r' \rangle\ , \end{aligned} \tag{2.29}$$

if $PSession\ s_0\ (ss \lhd step \oplus ss_1)$ and $P\ step$, then $all\ P\ ss_1$ holds.

Perhaps a more interesting property is a guarantee of the proper enforcement of revocation. We prove that once a permission is revoked by the user for the rest of a session, any further request for the same permission in the same session is refused.

Lemma 2.2.6. *The following property satisfies* (2.29)*,*

$$(\lambda\ step\ .\ \exists\ ses, step.s.session = Some\ ses \wedge ses.revoked\ p) \tag{2.30}$$

Theorem 2.2.7. *For any permission p, if $PSession\ s_0\ (ss \lhd step \lhd step_1 \oplus ss_1)$, $step_1.e = request\ p\ (Some\ (ua\_deny\ session))$ and $Pre\ step.s\ step_1.e$, then*

$$all\ (\lambda\ step\ .\ \forall\ o, step.e = request\ p\ o \rightarrow step.r \neq Some\ allowed)\ ss_1 \tag{2.31}$$

Proof. Since $Post\ step.s\ step_1.s\ step_1.r\ step_1.e$ must hold, p is revoked for the session in $step_1.s$. From Lemmas 2.2.5 and 2.2.6, p remains revoked for the rest of the session. Let $e = request\ p\ o$ be an event in a step $step_2$ in ss_1. We know that p is revoked for the session in the state before $step_2.s$. If the precondition for e does not hold in the state before[1], then $step_2.r = None$. Otherwise, e must be $request\ p\ None$ and its postcondition entails $step_2.r = Some\ denied$. □

[1] Actually, it holds only when $o = None$.

2.2.3 Refinement

In the formalization described in the previous sections it has been specified the behaviour of events implicitly as a binary relation on states instead of explicitly as a state transformer. Moreover, the described formalization is higher-order because, for instance, predicates are used to represent part of the device state and the transition semantics of events is given as a relation on states. The most evident consequence of this choice is that the resulting specification is not executable. What is more, the program extraction mechanism provided by Coq to extract programs from specifications cannot be used in this case. However, had we constructed a more concrete specification at first, we would have had to take arbitrary design decisions from the beginning, unnecessarily restricting the allowable implementations and complicating the verification of properties of the security model.

We will show in the rest of this section that it is feasible to obtain an executable specification from our abstract specification. The methodology we propose produces also a proof that the former is a refinement of the latter, thus guaranteeing soundness of the entire process. The methodology is inspired by the work of Spivey [69] on operation and data refinement, and the more comprehensive works of Back and von Wright [70] and Morgan [71] on refinement calculus.

Executable specification

In order to construct an executable specification it is first necessary to choose a concrete representation for every object in the original specification not directly implementable in a functional language. In particular, the transition relation that defines the behaviour of events implicitly by means of their pre- and postconditions must be refined to a function that deterministically computes the outcome of an event. At this point, it is unavoidable to take some arbitrary decisions about the exact representation to use. For example, a decidable predicate P on A might be represented as a function from A to a type isomorphic to *bool*, as an exhaustive list of the elements of A that satisfy the predicate (when P has finite support), or in some other equally expressive way. For every type T in the abstract specification, we will denote its concrete model as $\overline{T}$. Let $a : A$ and $\overline{a} : \overline{A}$; we will indicate that $\overline{a}$ is a refinement of a as $a \sqsubseteq \overline{a}$.

In our case, every predicate to be refined is decidable and is satisfied only by a finite subset of elements in its domain (they are all characteristic predicates of finite sets). Let P be one of such predicates on a set A and let l be a list of elements of $\overline{A}$; we will say that l refines P whenever

$$\begin{array}{l}(\forall\ a, P\ a \rightarrow \exists\ \overline{a}, \overline{a} \in l\ \wedge\ a \sqsubseteq \overline{a}) \wedge \\ (\forall\ \overline{a}, \overline{a} \in l \rightarrow \exists\ a, P\ a\ \wedge\ a \sqsubseteq \overline{a})\end{array} \tag{2.32}$$

where $x \in l$ means that there exists at least one occurrence of x in l. When A and $\overline{A}$ coincide, $\sqsubseteq$ is the equality relation on A, and the condition (2.32) simplifies to $\forall\ a, P\ a \leftrightarrow a \in l$. Let $a : A$ and $\overline{a} : \overline{A}$ be such that $a \sqsubseteq \overline{a}$; we define

$$\begin{array}{lcl}(None : option\ A) & \sqsubseteq & (None : option\ \overline{A}) \\ Some\ a & \sqsubseteq & Some\ \overline{a}\end{array} \tag{2.33}$$

The above concrete representations can be used to obtain a concrete model for the

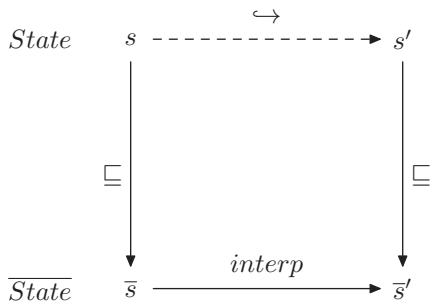

Figure 2.6: Simulation relation between *interp* and the relation $\hookrightarrow$. Given states s, $\overline{s}$ and events e, $\overline{e}$ such that $s \sqsubseteq \overline{s}$ and $e \sqsubseteq \overline{e}$, for every state $\overline{s}'$ and response r computed by *interp* there must exist a corresponding abstract state s' refined by $\overline{s}'$ reachable from s by the $\hookrightarrow$ relation with the same response

device state and the security-related events:

$$\begin{array}{lll} \overline{State} := [\![& suite & : list\ \overline{Suite}, \\ & session & : option\ \overline{SessionInfo}, \\ & granted & : SuiteID \to list\ Permission, \\ & revoked & : SuiteID \to list\ Permission\ ]\!] \end{array} \tag{2.34}$$

$$\begin{array}{lcl} \overline{start} & : & SuiteID \to \overline{Event} \\ \overline{terminate} & : & \overline{Event} \\ \overline{request} & : & Permission \to option\ UserAnswer \to \overline{Event} \\ \overline{install} & : & SuiteID \to \overline{Descriptor} \to Domain \to \overline{Event} \\ \overline{remove} & : & SuiteID \to \overline{Event} \end{array} \tag{2.35}$$

The refinement relation $\sqsubseteq$ can be naturally extended to states, events and the rest of the types used in the formalization.

Soundness

Having chosen a concrete representation for the objects in the specification, everything is set for specifying the behaviour of events as a function

$$interp : \overline{State} \to \overline{Event} \to \overline{State} \times (option\ Response) \tag{2.36}$$

The soundness of the *interp* function w.r.t. the transition relation $\hookrightarrow$ is given by the following simulation condition, illustrated in Figure 2.6.

$$\begin{array}{l} \forall\ (s : State)\ \left(\overline{s} : \overline{State}\right)\ (e : Event)\ \left(\overline{e} : \overline{Event}\right)\ (r : option\ Response), \\ \quad s \sqsubseteq \overline{s} \to e \sqsubseteq \overline{e} \to \\ \quad \text{let}\ \ (\overline{s}', r) := interp\ \overline{s}\ \overline{e}\ \text{in}\ \exists\ s' : State, s' \sqsubseteq \overline{s}'\ \wedge\ s \xhookrightarrow{e/r} s' \end{array} \tag{2.37}$$

It can be shown that the refinement relation $\sqsubseteq$ on states satisfies

$$\forall\ \overline{s} : \overline{State}, \exists\ s : State, s \sqsubseteq \overline{s} \tag{2.38}$$

Thus, the existential quantifier in (2.37) may be replaced by a universal quantifier to obtain the stronger (but sometimes easier to prove) condition:

$$\begin{array}{l} \forall\ (s\ s' : State)\ \left(\overline{s} : \overline{State}\right)\ (e : Event)\ \left(\overline{e} : \overline{Event}\right)\ (r : option\ Response), \\ \quad s \sqsubseteq \overline{s} \rightarrow e \sqsubseteq \overline{e} \rightarrow \\ \quad \text{let}\ (\overline{s}', r) := interp\ \overline{s}\ \overline{e}\ \text{in}\ s' \sqsubseteq \overline{s}'\ \rightarrow s \xrightarrow{e/r} s' \end{array} \tag{2.39}$$

With a function *interp* satisfying either (2.37) or (2.39) and a concrete initial state $\overline{s_0}$ that refines an initial abstract state s_0, the Coq program extraction mechanism can be used to produce an executable prototype of the MIDP 2.0 security model in a functional language such as OCaml, Haskell or Scheme.

2.3 A certified access controller for MIDP 2.0

This section reports work, based on [66], concerning the development of a high level formal specification of an Access Control module (AC) of JME - MIDP 2.0. In particular, a certified algorithm that satisfies the proposed specification of an AC is described.

2.3.1 Formalization of the access control module

The specification of the module has been defined as a conservative extension of the model presented in Section 2.2.1. The set $methodID$ of (method) identifiers that compose a MIDlet suite and the set $functionID$ of functions (or APIs) in a mobile device are introduced. The definition of a *Suite* is extended with a predicate that characterizes the methods that belong to that suite:

$$\begin{array}{lll} Suite \stackrel{\text{def}}{=} [\![& id & : SuiteID, \\ & domain & : Domain, \\ & descriptor & : Descriptor, \\ & methodid & : methodID \rightarrow Prop\ ]\!] \end{array} \tag{2.40}$$

Then, the state of the device is extended with the following predicates: one that specifies the functions or APIs registered in the device ($function$), one that describes the sensitive functions registered in the device ($functionSensitive$), and one that models the association of a permission to a sensitive function ($funcperm$). Formally,

$$\begin{array}{lll} State \stackrel{\text{def}}{=} [\![& suite & : Suite \rightarrow Prop, \\ & session & : option\ SessionInfo, \\ & granted & : SuiteID \rightarrow Permission \rightarrow Prop, \\ & revoked & : SuiteID \rightarrow Permission \rightarrow Prop, \\ & function & : functionID \rightarrow Prop, \\ & functionSensitive & : functionID \rightarrow Prop, \\ & funcperm & : functionID \rightarrow Permission \rightarrow Prop\ ]\!] \end{array} \tag{2.41}$$

A state now is said to be a valid one if it satisfies the conditions described in Section 2.2.1 and the following ones (for details see [66]):

- $MIDletsAtLeastOneMethod$: every installed MIDlet has at least one method;
- $fAtLeastOnePerm$: every installed sensitive function necessarily has associated a permission;

- $fPermInstalledSens$: if a function has associated a permission then that function must be installed in the device and is sensitive;
- $fUniquePermission$: no function has associated more than one permission;
- $methodInOnlyOneMidlet$: every method identifier is unique, even in distinct midlets;
- $permStateCoherence$: if one permission was granted (revoked) to a MIDlet for the rest of its life cycle, then neither could have been revoked (granted), for the rest of its life cycle, nor could have been granted or revoked for the rest of the session. Likewise, if one permission was granted (revoked) for the rest of the session, then neither it could have been revoked (granted), for the rest of the session, nor could have been granted or revoked for the rest of its life cycle life; and,
- $policyCompatible$: if the security policy does not mention any association between the protection domain of a suite and a certain permission then that permission can not be registered as granted or revoked to that suite in the device.

As to the events, the type $Event$ is now extended with a new constructor, $call$,

$$call : methodID \to functionID \to Event \tag{2.42}$$

which represents a method call to a function of the device. The behaviour of this kind of event is formalized using the pre- and postconditions explained in what follows. Let *suite* be a middlet suite, *meth* be a method, *func* be a function invoked by that method, and *dom* be the protection domain associated to *suite*. Then, the precondition of *call*

$$\begin{array}{l} Pre\ s\ (call\ meth\ func) \stackrel{\text{def}}{=} \\ \quad PreCall\ meth\ \wedge\ (s.functionSensitive\ func\ \to \\ \quad\quad \exists\ perm : Permission,\ s.funcperm\ func\ perm\ \wedge \\ \quad\quad\quad PreRequestNone\ perm\ \wedge\ \forall\ (ua : UserAnswer)\ (m : Mode), \\ \quad\quad\quad\quad (ua = uaAllow\ m \to\ PreRequestUserAllow\ perm\ m)\ \wedge \\ \quad\quad\quad\quad (ua = uaDeny\ m \to\ PreRequestUserDeny\ perm)) \end{array} \tag{2.43}$$

requires in the first place ($PreCall$) that it should exist an active session, that *suite* is the active suite and *meth* belongs to it. Then, if $func$ is a sensitive function, there must exist a permission *perm* associated to the function such that *perm* has been declared as required or optional inside the descriptor of *suite*. In the case no user intervention is required, the precondition specifies ($PreRequestNone$) that the security policy either unconditionally allows *dom* to access $func$, or *dom* specifies a user interaction mode for *perm*, and the user has granted a permission which is still valid. If user intervention is required and the user denies access, the precondition establishes ($PreRequestUserDeny$) that *dom* specifies a user interaction mode for *perm*, and the user has not granted any permission that is still valid. If user intervention is required and the user allows the access the precondition additionally requires ($PreRequestUserAllow$) the user authorization mode to be less, according to the order defined on permission modes, than the maximum allowed mode specified in the security policy.

The postcondition of the *call* event has been formally defined in Coq as follows

$$
\begin{array}{l}
Pos\ s\ s'\ r\ (call\ meth\ func) \stackrel{\text{def}}{=} \\
\quad (s.functionSensitive\ func\ \rightarrow\ \forall\ perm : Permission, \\
\qquad s.funcperm\ func\ perm \rightarrow\ PosRequestNone\ perm\ \vee \\
\qquad\quad \exists\ (ua : UserAnswer)\ (m : Mode), \\
\qquad\qquad (ua = uaAllow\ m\ \wedge\ PosRequestUserAllow\ perm\ m)\ \vee \\
\qquad\qquad (ua = uaDeny\ m\ \wedge\ PosRequestUserDeny\ perm\ m))\ \wedge \\
\neg s.functionSensitive\ func \rightarrow s = s'\ \wedge\ r = Some\ allowed
\end{array}
\tag{2.44}
$$

In the case no user intervention is required, the postcondition ($PosRequestNone$) specifies the response of the access controller when a *call* event is executed, namely, if the user has previously allowed (denied) authorization and it is still valid then the module will respond allowing (denying) the access to the function. If the security policy specifies that the access should be allowed unconditionally, then the module will respond allowing permission.

In the case that is required user intervention and the user grants the permission, the specification establishes ($PosRequestUserAllow$) that the response of the controller should be to allow the access. Similarly, if the permission is granted at session level (*session* mode) or for the whole life cycle of the application (*blanket* mode), this authorization should be registered in the device. If it is granted in *oneshot* mode, then the entire state of the device remains unchanged and the controller just reacts granting the access. $PosRequestUserDeny$ is similar to $PosRequestUserAllow$ but the controller shall deny the access and the update will be on the fields that register the access denial (depending on whether the denial is for the rest of the session or for the rest of the life cycle of the suite).

2.3.2 Verification of security properties

The invariants satisfied by the core model, which are described in Section 2.2.2, are preserved in the extended model, including those regarding the validity of the state of the device. The proof, for instance, that the execution of any event preserves the compatibility property of installed MIDlet suites described in (2.11), follows by first proving that the *call* event does not modify the state of the device and then proceeding as shown in Section 2.2.2. In what follows only two of the properties, *permStateCoherence* and *policyCompatible*, described in Section 2.3.1 are formally stated (Lemmas 2.3.1 and 2.3.2, respectively) and hints are provided on how it was proved they are satisfied by (in particular) a *call* event execution. For the complete set of analyzed properties, including their proofs, the interested reader is referred to [66, 72].

Lemma 2.3.1.

$$
\begin{array}{l}
\forall (s\ s' : State)(e : Event)(r : option\ Response)\ Valid\ s \rightarrow s \xrightarrow{e/r} s' \rightarrow \\
\forall (sid : SuiteID)(p : Permission)(ses : SessionInfo), \\
\quad s'.session = Some\ ses \rightarrow ses.sid = sid \rightarrow \\
\quad (s'.granted\ sid\ p\ \rightarrow \neg s'.revoked\ sid\ p\ \wedge \neg ses.granted\ p\ \wedge \neg ses.revoked\ p)\ \wedge \\
\quad (s'.revoked\ sid\ p\ \rightarrow \neg s'.granted\ sid\ p\ \wedge \neg ses.granted\ p\ \wedge \neg ses.revoked\ p)\ \wedge \\
\quad (ses.granted\ p\ \rightarrow \neg s'.granted\ sid\ p\ \wedge \neg s'.revoked\ sid\ p\ \wedge \neg ses.revoked\ p)\ \wedge \\
\quad (ses.revoked\ p\ \rightarrow \neg s'.granted\ sid\ p\ \wedge \neg s'.revoked\ sid\ p\ \wedge \neg ses.granted\ p)
\end{array}
\tag{2.45}
$$

Proof. Each of the four subproperties (propositions in the conjunction formula) is proved following a similar strategy, after the execution of an event (particularly a *call*). In the case that no user intervention is required, either because is established by the security policy or because the invoked function is not sensitive, the proof follows directly because the state of the device remains unchanged. In the case that user intervention is required, the proof proceeds by performing case analysis on the response provided by the user to the permission request (allow or deny in oneshot, session and blanket modes). □

Lemma 2.3.2.

$$\begin{aligned}
&\forall (s\ s' : State)(e : Event)(r : option\ Response)\ Valid\ s \rightarrow s \xrightarrow{e/r} s' \rightarrow \\
&\forall (sid : SuiteID)(p : Permission)(ses : SessionInfo), \\
&\quad s'.session = Some\ ses \rightarrow ses.sid = sid \rightarrow \\
&\quad \forall\ ms : Suite,\ s'.suite\ ms \rightarrow ms.sid = sid \rightarrow \\
&\quad (\neg policy.allow\ ms.domain\ p\ \wedge \neg \exists\ m : Mode,\ policy.user\ ms.domain\ p\ m) \rightarrow \\
&\quad \neg s'.granted\ s\ sid\ p\ \wedge \neg s'.revoked\ s\ sid\ p\ \wedge \neg ses.granted\ p\ \wedge \neg ses.revoked\ p
\end{aligned} \tag{2.46}$$

Proof. In the case that user intervention is not required in the execution of an event (particularly a *call*), the proof follows directly because the state of the device remains unchanged. Otherwise, the proof follows by contradiction, because a request for user intervention would contradict the hypothesis that there is no relation in the security policy between the permission and the protection domain of the active suite. □

2.3.3 A certified access controller

The high level formalization that has been described in the previous sections is appropriate and general: it is a good setting to reason about the properties of the MIDP 2.0 security model and, in particular, about the behavior of the access control module without conditioning possible implementations. However, in order to construct and certify an algorithm that satisfies the access control behavior that has been specified, first the high level model has been refined into a more computationally oriented one and then the soundness of the refinement process has been proved. Then, the algorithm was defined using the functional language of Coq and a theorem that establishes that the obtained algorithm satisfies the concrete specification of the access controller has been formally proved with the help of the proof assistant.

Model refinement

Following the methodology described in Section 2.2.3, the refinement of the high level specification has been engineered by providing a representation for both the state of the device and the events that can be directly implementable using a high level functional language. In addition to that, the relation which specifies the behavior of the events has been refined down to a deterministic function that computes explicitly the state transformation specified by those events. For the refinement of a decidable predicate P over a finite set A, it has been adopted the following approach: if the intended meaning of P is to characterize a set, then it shall be represented using lists of elements of type A, as in programming languages. If the intended meaning is that of a decision procedure, it shall be represented as a boolean function over type A. To illustrate how the components

of the concrete model look like, it follows the definition of the set $\overline{State}$, which should be grasped as the concrete representation of the state of the device.

$$\begin{array}{lll} \overline{State} \stackrel{\text{def}}{=} [\![& \overline{suite} & : list\ \overline{Suite}, \\ & \overline{session} & : option\ \overline{SessionInfo}, \\ & \overline{granted} & : SuiteID \to Permission \to bool, \\ & \overline{revoked} & : SuiteID \to Permission \to bool, \\ & \overline{function} & : functionID \to bool, \\ & \overline{functionSensitive} & : functionID \to bool, \\ & \overline{funcperm} & : functionID \to Permission\ ]\!] \end{array} \tag{2.47}$$

The algorithm

Before proceeding to describe the algorithm, it is worth recalling that each method inherits or gets the protection domain that has been associated to the MIDlet suite to which the method belongs. Therefore, every method of a middlet suite is related to the permissions that the protection domain embodies.

The algorithm has been built as a quite direct implementation of the specified behavior of the access controller in the concrete model. Basically, when a method invokes a protected function or API the access controller checks that the method has the corresponding permission and that the response from the user, if required, permits the access. If this checkup is successful, the access is granted; otherwise, the action is denied and a security exception is launched. The complete definition of the algorithm in Coq can be found in the Coq repository specified in Section 2.1.3; here it is only included a pseudocode version. It can be noticed that its structure is that of a case expression where the guards of the branches express properties involving some of the following components: the state, the policy of the device, the method and the invoked function.

The certification

The certification of the algorithm that implements a correct access controller reduces to prove the (correctness) theorem that establishes the following property: if the precondition of the *call* event is satisfied by a given state, then the state resulting from executing the algorithm satisfies its corresponding postcondition. Formally this is stated as follows:

Theorem 2.3.3.

$$\begin{array}{l} \forall\ (s\ s' : \overline{State})\ (policy : \overline{Policy})\ (ua : UserAnswer)\ (m : methodID) \\ \quad (f : functionID)\ (r : option\ Response), \\ \qquad \overline{Pre}\ s\ (call\ m\ f) \to \\ \qquad accessCtrlExe\ s\ policy\ ua\ m\ f\ = (s', r) \to \\ \qquad \overline{Pos}\ s\ s'\ r\ (call\ m\ f) \end{array} \tag{2.48}$$

The proof follows directly the structure of the algorithm and is constructed by performing case analysis on the conditions of the branches, by resorting to the use of several properties of the model, in particular those related to the validity of the state of the device.

Algorithm 1

accessCtrlExe $(s : \overline{State})$ $(policy : \overline{Policy})$ $(ua : UserAnswer)$ $(m : methodID)$
$(f : functionID) : \overline{State} \times (option\ Response) :=$
$(* \ current\ MIDlet\ suite\ in\ s \ *)$
$let\ ms := getMIDletSuiteInSession\ s\ in$
$(* \ permission\ associated\ to\ the\ function\ f \ *)$
$let\ p := functionToPermission\ s\ f\ in$
$(* \ m\ is\ not\ a\ method\ in\ ms \ *)$
$case\ (methInCurrSuite\ s\ m) = false : (s, None)$
$(* \ f\ is\ not\ a\ sensitive\ function\ registered\ in\ the\ device \ *)$
$case\ (s.\overline{functionSensitive}\ f = false) : (s, Some\ allowed)$
$(* \ p\ is\ not\ a\ permission\ declared\ in\ the\ descriptor\ of\ ms \ *)$
$case\ (ms.\overline{descriptor}.\overline{required}\ p \vee ms.\overline{descriptor}.\overline{optional}\ p) = false : (s, Some\ denied)$
$(* \ the\ permission\ p\ has\ been\ granted\ (revoked)\ previously\ in\ blanket\ mode \ *)$
$case\ s.\overline{granted}\ s.\overline{session}.\overline{sid}\ p = true : (s, Some\ allowed)$
$case\ s.\overline{revoked}\ s.\overline{session}.\overline{sid}\ p = true : (s, Some\ denied)$
$(* \ the\ permission\ p\ has\ been\ granted\ (revoked)\ previously\ in\ session\ mode$
$in\ the\ current\ session \ *)$
$case\ s.\overline{session}.granted\ p = true : (s, Some\ allowed)$
$case\ s.\overline{session}.revoked\ p = true : (s, Some\ denied)$
$(* \ the\ policy\ grants\ unconditionally\ the\ permission\ p\ to\ MIDlet\ ms \ *)$
$case\ policy.\overline{allow}\ ms.\overline{domain}\ p = true : (s, Some\ allowed)$
$(* \ the\ protection\ domain\ specifies\ interaction\ with\ the\ user\ for\ permission\ p,$
$which\ does\ not\ have\ a\ valid\ authorization\ (unauthorization)\ in\ s \ *)$
$case\ (policy.\overline{user}\ ms.\overline{domain}\ p\ oneshot \vee policy.\overline{user}\ ms.\overline{domain}\ p\ session \vee$
$policy.\overline{user}\ ms.\overline{domain}\ p\ blanket) = true :$
$(* \ the\ permission\ p\ is\ allowed\ (denied)\ in\ one\ of\ three\ modes.\ In\ session\ and$
$blanket\ modes,\ the\ authorization\ (unauthoritization)\ is\ registered\ in\ the\ state \ *)$
{
$case\ ua = uaAllow\ oneshot : (s, Some\ allowed)$
$case\ ua = uaAllow\ session : (s\ except\ s.\overline{session}.\overline{granted}\ p = true, Some\ allowed)$
$case\ ua = uaAllow\ blanket : (s\ except\ s.\overline{granted}\ (s.\overline{session}.\overline{sid})\ p = true, Some\ allowed)$
$case\ ua = uaDeny\ oneshot : (s, Some\ denied)$
$case\ ua = uaDeny\ session : (s\ except\ s.\overline{session}.\overline{revoked}\ p = true, Some\ denied)$
$case\ ua = uaDeny\ blanket : (s\ except\ s.\overline{revoked}\ (s.\overline{session}.\overline{sid})\ p = true, Some\ denied))$
}
$(* \ the\ protection\ domain\ does\ not\ grant\ permission\ p\ unconditionally\ and$
$does\ not\ specify\ any\ user\ interaction\ for\ it \ *)$
$case\ (policy.\overline{user}\ ms.\overline{domain}\ p\ oneshot \vee policy.\overline{user}\ ms.\overline{domain}\ p\ session \vee$
$policy.\overline{user}\ ms.\overline{domain}\ p\ blanket) = false : (s, Some\ denied)$

2.4 Formally verifying security properties of MIDP 3.0

Two important enhancements in MIDP 3.0 (informal) specification are Inter-MIDlet Communication (IMC) and Events. In particular, the IMC protocol enables MIDP 3.0 applications to communicate and collaborate among themselves. The MIDP 3.0 Event framework is a publish and subscribe architecture that allows MIDlets to push data to all authorized subscribers. MIDP 3.0 provides, in particular, the following capabilities: i) enable and specify proper behavior for MIDlets, such as: allow multiple concurrent MIDlets, and allow inter-MIDlet communications (direct communication, and indirect using events); ii) enable shared libraries for MIDlets; and iii) increase functionality in all areas, including: secure RMS stores, removable/remote RMS stores, IPv6, and multiple network interfaces per device.

In this section the formalization described in Section 2.2 is extended so as to model security at the application level, introduced in Section 2.1.1. The extended device state, a new condition for state validity, and the authorization event are presented. The validity of the security properties already proved for MIDP 2.0, and the new properties related with MIDP 3.0 are examined. The content of this section is based on [67].

2.4.1 Security at the application level

New data types are introduced: $Vendor$ denotes the names of the MIDlet suite vendors, $Certificate$ represents the set of public key certificates, and $Signature$ the signature of the MIDlet suite archive.

The application descriptor is extended with the vendor name and, optionally, its public key certificate and signature. A MIDlet suite declares access authorization to each set of suites involved. *DomainAuthorization* authorizes access for all MIDlet suites from a specific domain. *SignerAuthorization* authorizes access to a set of suites signed by a given signer certificate. $VendorUnsignedAuthorization$ gives access to the set of unsigned vendor suites, while *vendorSignedAuthorization* indicate access authorization to the set of MIDlet suites signed by the specified signing certificate of a vendor.

$$
\begin{array}{lll}
Descriptor \stackrel{\text{def}}{=} [\![& required & : Permission \to Prop, \\
& optional & : Permission \to Prop, \\
& vendor & : Vendor, \\
& jarSignature & : option\ Signature, \\
& signerCertificate & : option\ Certificate, \\
& domainAuthorization & : Domain \to Prop, \\
& signerAuthorization & : Certificate \to Prop, \\
& vendorUnsignedAuthorization & : Vendor \to Prop, \\
& vendorSignedAuthorization & : Vendor \to \\
& & Certificate \to Prop\ ]\!]
\end{array}
\tag{2.49}
$$

2.4.2 New device state

In order to represent allowed and denied access authorizations, the device state is also extended. Allowed access authorizations are represented with the *authorized* field while the denied ones are represented with the *unauthorized* one.

$$
\begin{array}{lll}
State \stackrel{\text{def}}{=} [\![& suite & : SuiteID \to Prop, \\
& session & : option\ SessionInfo, \\
& granted & : SuiteID \to Permission \to Prop, \\
& revoked & : SuiteID \to Permission \to\ Prop, \\
& authorized & : SuiteID \to SuiteID \to Prop, \\
& unauthorized & : SuiteID \to SuiteID \to Prop\ ]\!]
\end{array}
\tag{2.50}
$$

A validity condition is added to the device state regarding access authorizations. $ValidAuthorization$ establishes that a MIDlet suite can not be simultaneoussly authorized and unauthorized by another suite.

$$
\begin{array}{l}
ValidAuthorization \stackrel{\text{def}}{=} \forall\ idGrn\ idReq : SuiteID, \\
\quad s.authorizedidGrnidReq \to \neg s.unauthorizedidGrnidReq
\end{array}
\tag{2.51}
$$

2.4.3 Application level access request

A new event called *authorization* models the access authorization request from a MIDlet suite to the shared resources of the active suite.

$$
authorization : SuiteID \to Event \tag{2.52}
$$

The precondition for this event establishes that in the state s an active suite must exist, and the identifier of the requesting MIDlet suite $idReq$ belongs to an installed MIDlet suite.

$$
\begin{array}{l}
Pre\ s\ (authorization\ idReq) \stackrel{\text{def}}{=} \\
\quad \forall\ ses : SessionInfo, s.session = Some\ ses\ \to \\
\qquad \exists\ msReq : Suite, s.suite\ msReq\ \wedge\ msReq.id = idReq
\end{array}
\tag{2.53}
$$

The postcondition establishes the possible device answers and the relation between the previous state If the requesting MIDlet suite $idReq$ was already authorized, the access is granted and the device state remains unchanged. If the requesting MIDlet suite $idReq$ was not already authorized but there is a match between its credentials and any of the access authorizations declared by the active MIDlet suite, the access is also granted. This time the device state is modified since $idReq$ becomes part of the set of authorized MIDlet suites of the active suite.

$$
\begin{array}{l}
Post\ s\ s'\ (Some\ allowed)\ (authorization\ idReq)\ \stackrel{\text{def}}{=} \\
\quad \forall\ ses : SessionInfo, s.session = Some\ ses \to \\
\qquad (s.authorized\ ses.id\ idReq\ \wedge\ s = s') \vee \\
\qquad (\neg s.unauthorized\ ses.id\ idReq\ \wedge AccessAuthorization\ s\ ses.id\ idReq\ \wedge \\
\qquad\quad s \equiv_{authorized} s'\ \wedge\ s'.authorized\ ses.id\ idReq)
\end{array}
\tag{2.54}
$$

If the requesting MIDlet suite $idReq$ was already unauthorized, the access is denied. If $idReq$ was not authorized and there is a mismatch between its credentials and the access authorizations declared by the active MIDlet suite, the access is denied as well. In this opportunity, the state is also modified since $idReq$ is added to the set of unauthorized MIDlet suites of the active suite.

$$\begin{aligned}
&Post\ s\ s'\ (Some\ denied)\ (authorization\ idReq) \stackrel{\text{def}}{=} \\
&\quad \forall\ ses : SessionInfo, s.session = Some\ ses \rightarrow \\
&\qquad (s.unauthorized\ ses.id\ idReq\ \wedge\ s = s') \vee \\
&\qquad (\neg s.authorized\ ses.id\ idReq\ \wedge \neg AccessAuthorization\ s\ ses.id\ idReq\ \wedge \\
&\qquad\quad s \equiv_{unauthorized} s'\ \wedge\ s'.unauthorized\ ses.id\ idReq)
\end{aligned} \tag{2.55}$$

The predicate *AccessAuthorization* represents the comparison, in a given state, between the access authorizations declared by the first MIDlet suite and the credentials of the second one

$$AccessAuthorization : State \rightarrow SuiteID \rightarrow SuiteID \rightarrow Prop \tag{2.56}$$

An authorization is granted when at least one of the following conditions is verified:

- $IsDomainAuthorized$: the protection domain of the applicant suite $msReq$ satisfies the $domainAuthorization$ predicate of the $msGrn$ suite that shares the resource

$$msGrn.descriptor.domainAuthorization\ (msReq.domain) \tag{2.57}$$

- $IsVendorSignedAuthorized$: the vendor and the certificate of the applicant suite $msReq$ satisfy the $vendorSignedAuthorization$ predicate of the $msGrn$ suite that shares the resource

$$\begin{aligned}
&msGrn.descriptor.vendorSignedAuthorization\ (msReq.vendor \\
&\quad msReq.signerCertificate)
\end{aligned} \tag{2.58}$$

- $IsSignerAuthorized$: the certificate of the applicant suite $msReq$ satisfies the $signerAuthorization$ predicate of the $msGrn$ suite that shares the resource

$$msGrn.descriptor.signerAuthorization\ (msReq.signerCertificate) \tag{2.59}$$

- $IsVendorUnsignedAuthorized$: the vendor's name of the applicant suite $msReq$ satisfies the $vendorUnsignedAuthorization$ predicate of the $msGrn$ suite that shares the resource

$$msGrn.descriptor.vendorUnsignedAuthorization\ (msReq.vendor) \tag{2.60}$$

2.4.4 Conservative extension of security properties

Several changes are introduced in the formalization of the security model of MIDP 2.0 to incorporate the new aspects of MIDP 3.0. In the first place, the device state is modified so as to keep a record of the authorized and unauthorized MIDlet suites. In the second place, the access authorization is introduced as a new event. Finally, a new condition for the state validity is added.

In this new context, it is essential to prove that the device state, extended with the record of authorized and unauthorized MIDlet suites, is still valid after any event execution, even the authorization request. The state validity property must be considered with the new condition added.

Lemma 2.4.1 shows that the resulting state of executing any event in a valid initial state is still a valid state.

Lemma 2.4.1. *Let $ExtValid$ be a predicate over State defined as the conjunction of the of the predicates $Valid$ and $ValidAuthorizationvalidity$. For every $s\ s' : State$, $r :$ option Response, and $e : Event$, if $ExtValid\ s$ and $s \xrightarrow{e/r} s'$ hold, then $ExtValid\ s'$ also holds.*

Proof. The proof follows by case analysis on $s \xrightarrow{e/r} s'$. When $Pre\ s\ e$ does not hold, $s = s'$. From this equality and $ExtValid\ s$ then $ExtValid\ s'$. Otherwise, $Post\ s\ s'\ r\ e$ must hold and we proceed by case analysis on e. Only the event *authorization idReq* is considered here, in particular when the authorization is granted. Case analysis in the postcondition (2.54) of the event is carried out. In the first case, if the suite is authorized in the state s, the postcondition establishes $s = s'$. Being $ExtValid\ s$ then $ExtValid\ s'$ holds. In the second case, if the suite is unauthorized but there is a match between its credentials and any of the access authorizations declared by the active MIDlet suite in the state s, the postcondition establishes $s \equiv_{authorized} s'$. From the previous equivalence and $Valid\ s$, follows $Valid\ s'$. $ValidAuthorization\ s'$ remains to be proven. The equivalence $s \equiv_{authorized} s'$ preserves the unauthorized suites in the state s'; therefore, the suite is not unauthorized in the state s'. Since the suite is authorized in s', $ValidAuthorization\ s'$ holds. □

The state validity property is also invariant during a session. Lemma 2.4.2 shows that result.

Lemma 2.4.2. *Let ss be a session which starts in a valid state. Every state of ss is valid:*

$$all\ (\lambda\ step\ .\ ExtValid\ step.s)\ ss \tag{2.61}$$

Device state invariants, such as state validity, are useful to analyze other relevant properties of the security model. For instance, having proved that nonauthorization is an invariant, if a MIDlet suite has been unauthorized in a session, then any further access authorization request of the same MIDlet suite will be denied. These two last properties are presented in the next two lemmas.

Lemma 2.4.3. *Given an unauthorized MIDlet suite identified by sidReq, and a session ss generated from a valid initial state, the following expression is a session invariant:*

$$\begin{aligned} &all\ (\lambda\ step\ .\ \exists\ ses,\\ &\quad step.s.session = Some\ ses\ \wedge\ step.s.unauthorized\ ses.id\ sidReq)\ ss \end{aligned} \tag{2.62}$$

Lemma 2.4.4. *Given a MIDlet suite identified by sidReq, and a session $(ss \lhd step_i \lhd step_{i+1}) \lhd ss'$ generated from a valid initial state, such that*

$$\begin{aligned} &step_{i+1}.e = authorization\ sidReq\ \wedge\ step_{i+1}.r = Some\ denied\ \wedge\\ &Pre\ step_i.s\ step_{i+1} \end{aligned} \tag{2.63}$$

Then,

$$all\ (\lambda\ step\ .\ step.e = authorization\ sidReq \rightarrow step.r = Some\ denied)\ ss' \tag{2.64}$$

The lemmas stated were formalized and proven in the Coq proof assistant. An algorithm for the access authorization of applications was developed and its correctness proven [73].

Additionally, certified algorithms we have developed for application installation and communication between applications (IMC protocol) in MIDP 3.0, as part of final degree projects [74, 75].

2.4.5 Weaknesses of the security mechanisms of MIDP 3.0

MIDP 3.0 allows component-based programming. An application can access a shareable component running in a different execution environment through thin client APIs, like the IMC protocol previously mentioned. The shareable component handles requests from applications following the authorization-based security policy. As described in Section 2.1.1, this policy considers authorization access declarations and the credentials shown by the potential client applications, to grant or deny access.

The platform and application security levels were conceived as independent and complementary frameworks, however the (unsafe) interplay of some of the defined security mechanisms may lead to provoke (unexpected) violations to security policies. In what follows some potential weaknesses of MIDP 3.0 are put forward and a plausible (unsecure) scenario is discussed.

In the first place, while at the platform level permissions are defined for each sensitive function of the device, at the application level access authorization declarations do not distinguish between different shareable components from the same MIDlet suite. In this way, once its credentials have been validated, a client application may have access to all the components shared by another application.

In the second place, despite of the fact that permissions are granted in various modes (*oneshot*, *session*, *blanket*), access authorization is allowed exclusively in a permanent manner.

Finally, at the platform level the bounding of an application to a protection domain is based on the X.509 Public Key Infrastructure. At the application level the same infrastructure is used to verify the integrity and authenticity of applications, except for one case. The declaration shown in (2.49) with the predicate $vendorUnsignedAuthorization$, has the vendor name as the unique credential needed to grant access authorization.

Considering the previously established elements, the following scenario is feasible. Let A be an application which is bound to a certain protection domain and having a shareable component. The shareable component uses a sensitive function f of the device (granted by the protection domain) in order to implement its service. This application A declares access authorizations for unsigned applications from certain vendor V. Let B be an unsigned application which is bound to a different protection domain and that has been denied permission to access the sensitive function f. Now, if the application B is capable of providing just the vendor name V as credential, it would be granted permanent access to the shareable component. Thereby, B shall be able to access the sensitive function f of the device. This is a clear example of an unwanted behavior, where a sensitive function is accessed by an application without its permission. The next section shows the code of an unwanted access in MIDP 3.0.

Two observations are drawn from the previous scenario. On the one hand, a stronger access authorization declaration is necessary. If declarations based only on the vendor name are left aside, all the remaining ones demand the integrity and authenticity of signatures and certificates. This will result in a more reliable security model.

On the other hand, to avoid circumventing the security at the platform level two aspects should be considered. The first one is to declare the permissions of the protection domain exposed by a shareable component. The second aspect is to extend the security policy by conceding those permissions to sensitive functions when being accessed by applications through shareable resources.

Considering the previously mentioned observations the formalization of the security

model of MIDP 3.0 could be extended. This could be done by defining new conditions of the device state according to the security policy described above. It should also establish new pre and postconditions of the access authorization request.

An Unwanted Access Example in MIDP 3.0

This example illustrates a possible scenario where an unstrusted application manages to access, presenting a false credential, a resource shared by another application.

The RMS is a sensitive resource of the device that allows one to create and to access information persisted in a record store. Regardless of the protection domain to which they are linked, all applications have permissions to invoke the methods of the RMS.

In MIDP 3.0 the application that is owner of a record store can share it with other applications and restrict access to it using the authorization mechanism described in this document. Provided by the RMS API, the *openRecordStore* method allows one to create and to access a record store. Depending on the purpose, each application uses a different signature of the same method.

An application must use the following signature to create and share a record store in a restricted way. In particular, using the $AUTHMODE\_APPLEVEL$ value on the third parameter of the method it is activated the access control mechanism based on authorizations:

```
public static javax.microedition.rms.RecordStore openRecordStore(
       String recordStoreName, boolean createIfNecessary, int authmode, boolean writable)
```

A bank account store named $BankAccountsDb$, say, with security at the application level can be created as follows:

```
rsPrv = RecordStore.openRecordStore ("BankAccountsDb", true, AUTHMODE_APPLEVEL, false);
```

To restrict access, the application must include access authorizations in the descriptor. The access declaration by vendor shown below authorizes all the applications made by the same vendor $TrustyVendor$ that provides the application.

```
MIDlet-Name: TrustyMIDlet
MIDlet-Version: 1.0
MIDlet-Vendor: TrustyVendor
MIDlet-Access-Authorization-1: vendor;TrustyVendor
```

An application that intends to access the shared resource must use the signature shown below of the *openRecordStore* method. It also requires to know the name of the shared resource and some data of the provider application, namely, its vendor and public name:

```
public static javax.microedition.rms.RecordStore openRecordStore(
       String recordStoreName, String vendorName, String suiteName)
```

Now, let us suppose that an authorized third party seeks to obtain bank account information. For that, it develops an application without signatures or digital certificates, using as vendor's name that of the owner of the record store. A fragment of the descriptor could be defined as follows:

```
MIDlet-Name: MaliciousMIDlet
MIDlet-Version: 1.0
MIDlet-Vendor: TrustyVendor
```

Not being signed, the malicious application is installed and linked to the *Unidentified Third Party Protection Domain*. This protects the platform from the application performing an unauthorized access to sensitive resources of the device. However, this does not prevent access to the shared bank accounts store. The following Java code allows the untrusted application to open the record store and then to proceed to obtain, using the method *getRecord*, and to process, using the procedure *processRecord*, sensitive information:

```
// open the shared store
rsCns = RecordStore.openRecordStore( "BankAccountsDb", "TrustyVendor", "TrustyMIDlet" );

// proceed to read the records of the store
int nextID = rs.getNextRecordID();
byte[] data = null;
for( int id = 0; id < nextID; ++id ){
        try {
        int size = rs.getRecordSize( id );           // obtain the size of the registry
        if( data == null || data.length < size )
                data = new byte[ size ];
                rs.getRecord( id, data, 0 );         // obtain the registry
                processRecord( rs, id, data, size ); // process the registry
        } catch( InvalidRecordIDException e ){
        } catch( RecordStoreException e ){
                handleError( rs, id, e ); }
}
```

2.5 A framework for defining and comparing access control policies

An alternative model has been proposed that extends MIDP's by introducing permissions with multiplicities and adding flexibility to the way in which permissions are granted by the user of the device and used by the applications running on it. This section presents a framework, formalized using Coq, suitable for defining and comparing the access control policies that can be enforced by (variants of) those security models and to prove desirable properties they should satisfy. The content of this section is based on [68].

2.5.1 An alternative security model for mobile devices

In [29], a security model for interactive mobile devices is put forward which can be grasped as an extension of that of MIDP. The work presented in this section has focused on developing a formal model for studying, in particular, interactive user querying mechanisms for permission granting for application execution on mobile devices. Like in the MIDP case, the notion of permission is central to this model. A generalisation of the one-shot permission described above is proposed that consists in associating to a permission a multiplicity which states how many times that permission can be used.

The proposed model has two basic constructs for manipulating permissions: **grant** and **consume**. The grant construct models the interactive querying of the user, asking

whether he grants a particular permission with a certain multiplicity. The consume construct models the access to a sensitive function which is protected by the security police, and therefore requires (consumes) permissions.

A semantics of the model constructs is proposed as well as a logic for reasoning on properties of the execution flow of programs using those constructs. The basic security property the logic allows one to prove is that a program will never attempt to access a resource for which it does not have a permission. The authors also provide a static analysis that makes it possible to verify that a particular combination of the grant-consume constructs does not violate that security property. For developing that kind of analysis the constructs are integrated into a program model based on control-flow graphs. This model has also been used in previous work on modelling access control for Java, see for instance [76, 77].

One of the objectives of the work that is being reported here, has been to build a framework which would provide a formal setting to define the permission models defined by MIDP and the one presented in [29] (and variants of it) in an uniform way and to perform a formal analysis and comparison of those models. This framework, which is formally defined using the CIC, adopts, with variations, most of the security and programming constructions defined in [29]. In particular it has been modified so as to be parameterized by permission granting policies, while in the original work this relation is fixed.

2.5.2 A framework for access control modeling

This section introduces the formal setting used to define the security concepts that constitute the basis of certain access control mechanisms, to proceed then to describe how those mechanisms are used to define the permission granting models which are object of analysis of this work.

Permissions

Every (controlled) resource of the device is given a type. Let $ResType$ be the set of types of resources. If rt is a resource type, $Resources\ rt$ and $Actions\ rt$ define the set of resources of type rt available on the device and the actions that can performed over them, respectively. The permissions of a resource type are defined as follows:

$$\begin{aligned} PermRes \quad & (rt : ResType) \stackrel{def}{=} \\ | \quad & valid : list\ (Resources\ rt) \to list\ (Actions\ rt) \to PermRes\ rt \\ | \quad & invalid : PermRes\ rt \end{aligned} \tag{2.65}$$

That is, given a resource type rt, an object of type $PermRes\ rt$ is a set (represented by a list) of actions and resources over rt, or the constant $invalid$. A relation $\sqsubseteq_{PermRes}$ is defined by applying set inclusion component-wise. This relation defines a lattice structure where $invalid$ is the bottom element $\bot_{PermRes}$ and $\sqcup_{PermRes}$ a lub operator which is obtained applying set union component-wise.

As already mentioned, a notion of multiplicity of granted permission is introduced in [29]. A multiplicity is defined to be either a natural number, a special value ∞ that

denotes irrestricted permission, or an error value $\perp$. A type Mul is defined:

$$\begin{array}{rl} Mul \stackrel{def}{=} & \mid \perp \ : Mul \\ & \mid val : nat \rightarrow Mul \\ & \mid \infty \ : Mul \end{array} \tag{2.66}$$

It is straightforward to see that a lattice can be constructed over Mul with $\perp$ and ∞ as the bottom and top elements, respectively. The obvious extensions of functions and predicates defined over naturals to functions and predicates over Mul, such as $\sqsubseteq_{Mul}$, $+_{Mul}$, $-_{Mul}$, $pred_{Mul}$, are also defined.

An accumulated permission for a resource type is comprised of two components: the set of resources and actions allowed and a multiplicity. One such permission (of resource type rt) is then grasped as an object of the following record type:

$$PermMul\ (rt : ResType) \stackrel{def}{=} [\![\ permRes : PermRes, rtmul : Mul\ ]\!] \tag{2.67}$$

The lattice of permissions of a resource type can be obtained by defining the order $\sqsubseteq_{PermMul}$:

$$\begin{array}{rcl} pm_1 \sqsubseteq_{PermMul} pm_2 & \stackrel{def}{=} & pm_1.permRes \sqsubseteq_{PermRes} pm_2.permRes \\ & \wedge & pm_1.mul \sqsubseteq_{Mul} pm_2.mul \end{array} \tag{2.68}$$

where pm_1 and pm_2 are objects of type $PermMul\ rt$. Now, the permission state of the device is defined. One such state is ultimately a mapping that associates a permission to each resource type. Therefore, it is defined as the following dependent function type:

$$Perm \stackrel{def}{=} \forall\ (rt : ResType), PermMul\ rt \tag{2.69}$$

It is said that two permissions p_1 and p_2 are (extensionally) equal if for every resource type rt it holds that $p_1\ rt\ = p_2\ rt$.

An order $\sqsubseteq_{Perm}$ can be defined as the product-wise extension of $\sqsubseteq_{PermMul}$ as follows:

$$p_1 \sqsubseteq_{Perm} p_2 \stackrel{def}{=} \forall\ (rt : ResType), (p_1\ rt) \sqsubseteq_{PermMul} (p_2\ rt) \tag{2.70}$$

In order to model the operations that affect the state of the permissions an *update* function is introduced:

$$update\ (p : Perm)(rt : ResType)(pres : PermRes\ rt)(m : Mul) : Perm \tag{2.71}$$

The intended (and formalized) behaviour of this function is that of a usual store updating operator: the permission state remains unchanged for every resource type different from rt, and for rt yields $\langle pres, m\rangle$. This behavior is captured by the following two lemmas:

Lemma 2.5.1.

$$\begin{array}{l} \forall\ (p : Perm)(rt : ResType)(pres : PermRes\ rt)(m : Mul), \\ \quad (update\ p\ rt\ pres\ m)\ rt = \langle pres, m\rangle \end{array} \tag{2.72}$$

Lemma 2.5.2.

$$\begin{array}{l} \forall\ (p : Perm)(rt\ rt' : ResType)(pres : PermRes\ rt)(m : Mul), \\ \quad rt <> rt' \rightarrow (update\ p\ rt\ pres\ m)\ rt' = p\ rt' \end{array} \tag{2.73}$$

If rt is a resource type and p a permission state, then the following inductive relation $Error$ is defined

$$\frac{(p\ rt).permRes = invalid\ rt}{Error\ p} \qquad \frac{(p\ rt).mul = \bot}{Error\ p} \tag{2.74}$$

The intuition is that an error situation may occur when either there is an attempt to perform an action over a resource of type rt and no valid permission is associated to it (first rule) or when there are no granted permissions for that resource (second rule).

Programs

A program in, among others, [29, 78] is represented by a control-flow graph that captures the manipulations of permissions and the handling of method calls and returns as well as exceptions.

A control-flow graph is a tuple $G = (NO, EX, KD, TG, CG, EG, n_0)$ where:

- NO is the set of nodes of the graph (one for each instruction),
- EX is the set of exceptions,
- KD is a function of type $KD : NO \to Instr$ that associates each node to an instruction,
- $TG : NO \to NO \to Prop$ is the propositional function that characterizes the set of intra-procedural edges (i.e. $n_1\ TG\ n_2$ if control can be transferred from instruction at node n_1 to instruction at node n_2 within the currect procedure),
- CG is the set of inter-procedural edges (which can be used to capture dynamic method calls),
- $EG : EX \to NO \to NO \to Prop$ are the intra-procedural exception edges,
- $n_0 : NO$ is the graph entry node.

The instructions are formally defined in the framework by means of the following inductive type:

$$\begin{aligned} Instr \stackrel{def}{=} & \\ & \mid\ Grant : \forall\ (rt : ResType), validPermRes\ rt \to MulValid \to Instr \\ & \mid\ Consume : \forall\ (rt : ResType), validPermRes\ rt \to Instr \\ & \mid\ Call : Instr \\ & \mid\ Return : Instr \\ & \mid\ Throw : EX \to Instr \end{aligned} \tag{2.75}$$

where $MulValid$ is the type of valid multiplicities, that is, different from the multiplicity $\bot$. The definition of the operational semantics of programs strongly depends on those of the permission granting and consumption mechanisms. They are briefly discussed and described in what follows.

In [29] two variants are discussed concerning the effect of the update operation after a permission has been granted: either the permissions before the update instruction are

discarded or they are accumulated. At a first sight these *permission granting policies* have advantages and drawbacks. Furthermore, independently of this particular discussion, it is at this point that the permission model proposed by the authors introduces a generalization with respect to that of MIDP: the multiplicity of a permission. One of the main objectives of the work presented here has been to design a framework that would make it possible to provide a uniform setting where those different permissions models could be formally defined and compared. To that end, the constructions defined to provide semantics to the computational behaviour of the programs as well as to reason over that behaviour have been parameterized by permission granting policies. One such parameter shall be formally represented by an object of the following type:

$$\begin{aligned} grantPolicy \stackrel{def}{=} \; & \forall\ (rt : ResType), \\ & validPermRes\ rt \to NZMulValid \to Perm \to Perm \end{aligned} \tag{2.76}$$

where an object of type $NZMulValid$ is a valid multiplicity constructed with a non-zero natural.

As to the consumption of permissions, the following is the definition of the consume operation:

$$\begin{aligned} consume \quad & (rt : ResType)(pr : validPermRes\ rt)(p : Perm) : Perm \stackrel{def}{=} \\ & if\ (pr\ \sqsubseteq_{PermRes}\ (p\ rt).permRes) \\ & then\ update\ p\ rt\ (p\ rt).permRes\ (pred_{Mul}\ (p\ rt).mul) \\ & else\ update\ p\ rt\ (invalid\ rt)\ (pred_{Mul}\ (p\ rt).mul) \end{aligned} \tag{2.77}$$

The consume operation is monotonic on permissions. This is stated (and proved) in the following lemma:

Lemma 2.5.3.

$$\begin{aligned} & \forall\ (rt : ResType)(pr : validPermRes\ rt)(p\ p' : Perm), \\ & \quad p \sqsubseteq_{Perm} p' \to (consume\ rt\ pr\ p) \sqsubseteq_{Perm} (consume\ rt\ pr\ p') \end{aligned} \tag{2.78}$$

Following [29] the small-step operational semantics of a control-flow graph has been defined basically as a relation that defines transitions between states consisting of a standard control-flow stack of nodes enriched with the permissions held at that point in the execution. This definition has been extended by making it depend on a permission granting policy g. Formally, it has been defined as an inductive propositional function $\rightsquigarrow_g$ whose rules are depicted in Figure 2.7. An important property of this semantics is that it is non-intrusive, that is to say, the permission state does not interfere with execution. In other words, a transition will not be blocked by the absence of permissions. This is formally stated, and proved, in the following lemma:

Lemma 2.5.4.

$$\begin{aligned} & \forall\ (g : grantPolicy)(s\ s' : list\ NO)(ex\ ex' : option\ EX)(p\ p' : Perm), \\ & \quad s\ ex\ p\ \rightsquigarrow_g s'\ ex'\ p' \to \forall\ (p : Perm), (\exists\ (p' : Perm), s\ ex\ p\ \rightsquigarrow_g\ s'\ ex'\ p') \end{aligned} \tag{2.79}$$

$$\frac{KD\ n = Grant\ rt\ pr\ m \quad TG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright s)\ None\ (g\ rt\ pr\ m\ p)}$$

$$\frac{KD\ n = Consume\ rt\ pr \quad TG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright s)\ None\ (consume\ rt\ pr\ p)}$$

$$\frac{KD\ n = Call \quad CG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright n \triangleright s)\ None\ p} \qquad \frac{KD\ r = Return \quad TG\ n\ n'}{(r \triangleright n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright s)\ None\ p}$$

$$\frac{KD\ n = Throw\ ex \quad EG\ ex\ n\ h}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (h \triangleright s)\ None\ p} \qquad \frac{KD\ n = Throw\ ex \quad \forall\ (h : NO), \neg EG\ ex\ n\ h}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n \triangleright s)\ (Some\ ex)\ p}$$

$$\frac{\forall\ (h : NO), \neg EG\ ex\ n\ h}{(t \triangleright n \triangleright s)\ (Some\ ex)\ p\ \rightsquigarrow_g\ (n \triangleright s)\ (Some\ ex)\ p} \qquad \frac{EG\ ex\ n\ h}{(t \triangleright n \triangleright s)\ (Some\ ex)\ p\ \rightsquigarrow_g\ (h \triangleright s)\ None\ p}$$

Figure 2.7: Semantics of instructions

Traces

In [29] global results on the execution of programs are expressed on traces, which in turn are defined in terms of the operational semantics described above (instantiated for a particular grant policy) as follows: *a partial trace of a control-flow graph is a sequence (of type snocList* $(NO, option\ EX)$*) of nodes*

$$[] \lhd \langle n_0, None\rangle \lhd \langle n_1, e_1\rangle \lhd \cdots \lhd \langle n_k, e_k\rangle \tag{2.80}$$

such that for all $0 \leq i < k$ *there exists* $\rho, \rho' \epsilon\ Perm$, $s, s' \epsilon\ (list\ NO)$ *and verifying* $n_i \triangleright s, e_i, \rho \rightsquigarrow n_{i+1} \triangleright s', e_{i+1}, \rho'$.

The stacks s and s' in the above definition are existentially quantified because they are not defined to be components of the elements of a trace. This quantification, however, induces a loss of information w.r.t. the operational semantics. An example should clarify this situation. Consider the control-flow graph:

$$\begin{aligned}
&NO = \{A, B, C, D\}\\
&EX = \{\}\\
&KD = \{(A, Return), (B, x), (C, Consume\ rt\ y), (D, Return)\}\\
&TG = \{(B, C), (C, D)\}\\
&CG = \{\}\\
&EG = \{\}\\
&n_0 = A
\end{aligned}$$

where $x : Instr$, $rt : ResType$, $y : validPermRes\ rt$, and with initial permission $p_{init} = \lambda\ (rt : ResType)\ .\ \langle (valid\ rt\ []\ []), (val\ 0)\rangle$. Figure 2.8 depicts the control-flow graph in question. From this definition it can be noticed that $[] \lhd \langle A, None\rangle$ is the only admissible trace yielding a valid permission state. According to the definition of partial trace stated above, the object $([] \lhd \langle A, None\rangle \lhd \langle C, None\rangle \lhd \langle D, None\rangle)$ is admitted as a partial trace of the defined control-flow graph. This trace can be built using the transition rules for the *Consume* and *Return* instructions (see Figure 2.7). However, this latter trace yields an error situation, because the transition from node C to node D attempts to consume an unavailable permission.

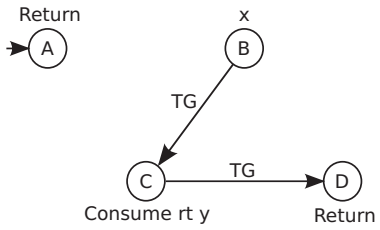

Figure 2.8: Control-flow graph example

The definition of program execution traces that are proposed in the framework presented here remedies the situation described above by including the node stack as a component of the elements of the trace. This is formally represented by the following type:

$$Trace \stackrel{def}{=} snocList \; [\![\; noT : NO, stT : list \; NO, exT : option \; EX \;]\!] \tag{2.81}$$

The notion of parameterized partial trace is then inductively defined over elements of type $Trace$ as follows:

$$\overline{PTrace_g \; []} \qquad \overline{PTrace_g([] \lhd \langle n_0, [], None \rangle)} \tag{2.82}$$

$$\frac{PTrace_g \; (tr \lhd \langle n, s, ex \rangle) \quad \exists \; (p \; p' : Perm), n \rhd s \; ex \; p \rightsquigarrow_g n' \rhd s' \; ex' \; p'}{PTrace_g \; (tr \lhd \langle n, s, ex \rangle \lhd \langle n', s', ex' \rangle)} \tag{2.83}$$

Let tr be a trace and g be a grant policy, if $PTrace_g \; tr$ holds then it shall be said that tr is a valid trace according to g.

Given a trace tr and a grant policy g, the function

$$PermsOf_g : Perm \to Trace \to Perm \tag{2.84}$$

computes the permission state resulting from the execution of the program that tr represents:

$$\begin{array}{l} PermsOf_g(p_{init} : Perm)(tr : Trace) : Perm \stackrel{def}{=} \\ match \; tr \; with \\ \quad | \; [] \Rightarrow p_{init} \\ \quad | \; tr' \lhd e \Rightarrow match \; KD \; e.noT \; with \\ \qquad | \; Consume \; rt \; pr \Rightarrow consume \; rt \; pr \; (PermsOf_g \; p_{init} \; tr') \\ \qquad | \; Grant \; rt \; pr \; m \Rightarrow g \; rt \; pr \; m \; (PermsOf_g \; p_{init} \; tr') \\ \qquad | \; \_ \; \Rightarrow PermsOf_g \; p_{init} \; tr' \\ \quad end \\ end \end{array} \tag{2.85}$$

Finally, given a grant policy g, a trace is said to be safe if none of its prefixes yields a faulty permission state:

$$\begin{array}{l} Safe_g(tr : Trace)(p_{init} : Perm) \stackrel{def}{=} \\ \qquad \forall \; tr' : Trace, (prefix \; tr' \; tr) \to \neg Error(PermsOf_g \; p_{init} \; tr') \end{array} \tag{2.86}$$

2.5.3 Permission grant policies

Two kinds of grant policies are analysed in [29]: given a resource type rt, one of the policies establishes that when a new permission is granted to resources of rt, all previous granted permissions are overwritten. This policy is called here $grant_{ow}$. The other policy, called here $grant_{ac}$, establishes that new granted permissions for rt are accumulated with the ones previously obtained for that resource type. These policies are formally defined as follows:

$$\begin{aligned} grant_{ow} : \; & grantPolicy \stackrel{def}{=} \\ & \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\ & update\ p\ rt\ pr\ m \end{aligned} \tag{2.87}$$

$$\begin{aligned} grant_{ac} : \; & grantPolicy \stackrel{def}{=} \\ & \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\ & update\ p\ rt\ (pr \sqcup_{PermRes} (p\ rt).permRes)\ (m +_{mul} (p\ rt).mul) \end{aligned} \tag{2.88}$$

The permission modes defined by MIDP are also defined below as grant policies. The $grant_{bk}$ term represents the blanket permission mode, which specifies unrestricted access to a given resource type. The one-shot permission mode, which specifies a single access to a given resource type, is represented by the term $grant_{os}$.

$$\begin{aligned} grant_{bk} : \; & grantPolicy \stackrel{def}{=} \\ & \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\ & update\ p\ rt\ (pr \sqcup_{PermRes} (p\ rt).permRes)\ \infty \end{aligned} \tag{2.89}$$

$$\begin{aligned} grant_{os} : \; & grantPolicy \stackrel{def}{=} \\ & \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\ & update\ p\ rt\ pr\ 1 \end{aligned} \tag{2.90}$$

It should be noticed that both the allowed mode and the session permission mode specified by MIDP 2.0 can be modeled as a blanket grant policy. In the first case, the granted permission would hold for the rest of the life cycle of the application to which is granted the permission and, in the second case, the scope would be that of a session during which that application is active.

In order to perform a comparative analysis of grant policies of the kind of the ones just defined, the following relation is defined:

$$\begin{aligned} g_1 \sqsubseteq_g g_2 \stackrel{def}{=} \; & \forall\ (tr : Trace)(p : Perm), \\ & Ptrace_{g_1}\ tr \rightarrow Safe_{g_1}\ p\ tr \rightarrow Safe_{g_2}\ p\ tr \end{aligned} \tag{2.91}$$

This order establishes that given a control-flow graph, for every valid trace of the graph according to g_1 and every initial set of permissions it holds that if the trace is safe by granting the permissions using g_1 as policy, then it must also be safe if the permissions are granted using the policy g_2. Intuitively, g_1 yields a more restrictive permission model.

The following lemma states that the order relation between permission states preserves error situations. It can also be proved that $\sqsubseteq_g$ is a partial order (reflexive, transitive and antisymmetric). These results shall be of help when relating the grant policies described so far.

Lemma 2.5.5.

$$\forall\ (p_1\ p_2 : Perm), Error\ p_1 \rightarrow p_1 \sqsubseteq_{Perm} p_2 \rightarrow Error\ p_2 \tag{2.92}$$

The relation $\sqsubseteq_g$ defines a lattice structure with the policies $grant_{os}$ and $grant_{bk}$ as the bottom and top elements respectively.

The following theorem states a sufficient condition (a criterion) to prove that two permission granting policies, say g_1 and g_2, are in the order relation ($g_1 \sqsubseteq_g g_2$):

1. the error situations that arise using g_2 as a policy are also error situations if g_1 is used, and

2. if a grant policy g_1 is applied, then every permission available at the end of a trace is also available if g_2 is used instead of g_1.

This theorem is important in order to compare different security policies.

Theorem 2.5.6.

$$\begin{array}{l}\forall\ (g_1\ g_2 : grantPolicy)\\ (H_{errors} : \forall\ (rt : ResType)\ (pr : validPermRes\ rt)\ (m : NZMulValid)\\ \qquad\qquad (p : Perm), Error(g_2\ rt\ pr\ m\ p) \rightarrow Error(g_1\ rt\ pr\ m\ p))\\ (H_{perms} : \forall\ (p : Perm)\ (tr : Trace),\\ \qquad\qquad (PermsOf_{g_1}\ p\ tr) \sqsubseteq_{Perm} (PermsOf_{g_2}\ p\ tr)),\\ g_1 \sqsubseteq_g g_2\end{array} \tag{2.93}$$

Proof. The proof proceeds by induction over $(PTrace_{g_1}\ tr)$, which is obtained after unfolding $g_1 \sqsubseteq_g g_2$. If the trace tr is empty, then the theorem holds trivially. In the case the trace is a singleton node, the proof uses hypothesis H_{errors} and proceeds by doing case analysis on the instruction type associated with that node; the interesting case corresponds to the *Grant* instruction, since *consume* is monotonic w.r.t. $\sqsubseteq_{Perm}$ and the rest of the instructions do not affect the permission state.

The inductive step follows basically from the Lemma 2.5.5, the hypothesis H_{perms}, and the induction hypothesis. □

2.5.4 Relating permission grant policies

Using the formal setting defined so far it is now possible to state and prove a theorem that establishes how the four policies described in the previous section are related according to the order relation $\sqsubseteq_g$.

Theorem 2.5.7.

$$grant_{os} \sqsubseteq_g grant_{ow} \sqsubseteq_g grant_{ac} \sqsubseteq_g grant_{bk} \tag{2.94}$$

Proof. The proof proceeds first by proving the three inequalities $grant_{os} \sqsubseteq_g grant_{ow}$, $grant_{ow} \sqsubseteq_g grant_{ac}$ and $grant_{ac} \sqsubseteq_g grant_{bk}$, and then applying the transitivity of the order $\sqsubseteq_g$. Each inequality is proved applying the theorem that establishes the sufficient conditions to prove that two grant policies are in the order relation (Theorem 2.5.6), and following a similar strategy. Here it shall be presented in detail the proof of the first inequality, indications on how to proceed for the remaining two cases shall also be provided.

The application of the Theorem 2.5.6 to prove $grant_{os} \sqsubseteq_g grant_{ow}$ generates in turn the following proof obligations:

$$\forall\ (rt : ResType)\ (pr : validPermRes\ rt)\ (m : NZMulValid)\ (p : Perm),\\ Error(g_2\ rt\ pr\ m\ p) \rightarrow Error(g_1\ rt\ pr\ m\ p)) \tag{2.95}$$

$$\forall\ (p : Perm)\ (tr : Trace), (PermsOf_{g_1}\ p\ tr) \sqsubseteq_{Perm} (PermsOf_{g_2}\ p\ tr)) \tag{2.96}$$

The proof of (2.95) proceeds by first applying the Lemma 2.5.5. This leads to have to prove that $(grant_{os}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p)$. Unfolding the definition of $grant_{ow}$ and $grant_{os}$, and applying the lemmas that characterize the function *update*, we have to prove $\langle pr, 1\rangle \sqsubseteq_{PermMul} \langle pr, m\rangle$ and $(p\ rt) \sqsubseteq_{PermMul} (p\ rt)$. The latter follows directly because $\sqsubseteq_{PermMul}$ is reflexive. As to the former, as $m : NZMulValid$ so the least number it can be is 1, in which case, since $\sqsubseteq_{PermMul}$ is reflexive, the obligation is discharged.

For (2.96), the proof proceeds by induction on tr:

- $tr = []$, the inequality simplifies to $p \sqsubseteq_{Perm} p$ and since $\sqsubseteq_{Perm}$ is reflexive, this obligation is discharged.
- $tr = tr' \triangleleft \langle n, st, ex\rangle$, the proof proceeds by case analysis on $KD\ n$. The relevant cases are *Grant* and *Consume*, since the rest of the instructions do not affect the permission state. The *Consume* case is straightforward since the function *consume* is monotonic, and by induction hypothesis it is known that:

  $$(PermsOf_{grant_{os}}\ p\ tr') \sqsubseteq_{Perm} (PermsOf_{grant_{ow}}\ p\ tr') \tag{2.97}$$

  The *Grant* case is proved using transitivity of $\sqsubseteq_{Perm}$, the induction hypothesis and the following two lemmas:

  – $\forall\ (rt : ResType)(pr : validPermRes\ rt)(m : NZMulValid)(p : Perm),$
  $(grant_{os}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p)$
  – $\forall\ (rt : ResType)(pr : validPermRes\ rt)(m : NZMulValid)(p\ p' : Perm),$
  $p \sqsubseteq_{Perm} p' \rightarrow (grant_{ow}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p').$

  (2.98)

The structure of the proof of the two remaining inequalities are quite similar to the one just described above. In both cases the bulk of the proof reduces to proving auxiliary lemmas similar to the ones of the proof obligation (2.96) for the involved grant policies. □

This theorem and its proof provide a formal evidence that, in the first place, of the four policies, MIDP's one-shot is the most restrictive policy and MIDP's blanket is the most permissive one. In addition to that, these two policies have been formally related with the permission grant policies defined in [29]. Furthermore, the theorem also formally relates these two latter granting policies, showing that the accumulative one is more permissive than the overwriting one.

The difference between accumulating permissions and overwriting permissions is subtle. The problem with accumulating permissions is that at any program point to approximate the permissions available for a given resource type it has to be considered all

the consumptions and all the permissions granted for that resource type. Whereas in the overwriting grant policy it is enough to consider the last grant operation and the subsequent consume operations. This suggests that a static permission analysis might be simpler using the overwriting grant policy.

2.6 Related work

This work builds upon and extends a number of previously published papers [65, 66, 67, 68]. In what follows we discuss work related with the results that have been presented in the previous sections.

Some effort has been put into the evaluation of the security model for MIDP 2.0; Kolsi and Virtanen [79] and Debbabi et al. [80] analyze the application security model, spot vulnerabilities in various implementations and suggest improvements to the specification. Although these works report on the detection of security holes, they do not intend to prove their absence. The formalizations we overview in this article, however, provide a formal basis for the verification of the model and the understanding of its intricacies.

Various articles analyze access control in Java and C♯, see for instance [78, 77, 81, 76, 82]. All these works have mainly focused on the stack inspection mechanism and the notion of granting permissions to code through privileged method calls. The access control check procedures in MIDP do not involve a stack walk. While in the case of the JSE platform there exists a high-level specification of the access controller module (the basic mechanism is based on stack inspection [28]), no equivalent specification exists for JME. In this work, we illustrate the pertinence of the specification of the MIDP (version 2.0) security model that has been developed by specifying and proving the correctness of an access control module for the JME platform.

Besson, Duffay and Jensen [29, 83] have put forward an alternative access control model for mobile devices that generalizes the MIDP model by introducing permissions with multiplicites, extending the way permissions are granted by users and consumed by applications. One of the main outcomes of the work we report in the present paper is a general framework that sets up the basis for defining, analyzing and comparing access control models for mobile devices. In particular, this framework subsumes the security model of MIDP and several variations, including the model defined in [29, 83].

Android [25] is an open platform for mobile devices developed by the Open Handset Alliance led by Google, Inc. Focusing on security, Android combines two levels, Linux system and application framework level, of enforcement [26, 27]. At the Linux system level, Android is a multi-process system. The Android security model resembles a multi-user server, rather than the sandbox model found on JME platform. At the application framework level, Android provides fine control through Inter-Component Communication reference monitor, that provides Mandatory Access Control enforcement on how applications access the components. There have been several analysis done on the security of the Android system, but few of them pay attention to the formal aspect of the permission enforcing framework. In [84, 85], the authors propose an entity-relationship model for the Android permission scheme [86]. This work —the first formalization of the permission scheme which is enforced by the Android framework— builds a state-based formal model and provides a behavioral specification, based in turn on the specification developed in [65] and described in Section 2.3. The abstract operation set considered in [84, 85] does not include permission request/revoke operations presents in MIDP. In

[87], Chaudhuri presents a core language to describe Android applications, and to reason about their dataflow security properties. The paper introduces a type system for security in this language. The system exploits the access control mechanisms already provided by Android. Furthermore, [26] reports a logic-based tool, Kirin, for determining whether the permissions declared by an application satisfy a certain safety invariant. A formal comparison between both JME-MIDP and Android security models is an interesting further work. However, we develop a first informal analysis in the technical report [88], which also details the Android security model and analyzes some of the more recent works that formalize different aspects of this model.

Language-based access control has been investigated for some idealised program models, see *e.g.* [89, 90, 91]. These works make use of static analysis for verifying that resources are accessed according to access control policies specified and that no security violations will occur at run-time. They do not study, though, specific language primitives for implementing particular access control models.

2.7 Summary

The Java Micro Edition platform defines the Mobile Information Device Profile (MIDP) to facilitate the development of applications for mobile devices. In this chapter, we have studied and compared formally several variants of the security model specified by MIDP to access sensitive resources of a device. Our contributions are manyfold. First, we have described a formal specification of the MIDP 2.0 security model that has been developed using proof assistant Coq. This formalization provides an abstraction of the state of a device and security-related events that allows to reason about the security properties of the platform where the model is deployed. Then, we have illustrated the pertinence of this formalization by specifying and proving the correctness of an access control module. The latest version of MIDP (3.0), introduces a new dimension in the security model at the application level. We have extended the formal specification developed for MIDP 2.0 to incorporate the changes introduced in MIDP 3.0, and we have showed that this extension is conservative, in the sense that it preserves the security properties we proved for the previous model. We have spotted and discuss here some weaknesses introduced by the security mechanisms of version 3.0. In [29, 83] it has been proposed an alternative access control model for mobile devices that generalizes the MIDP model by introducing permissions with multiplicities, extending the way permissions are granted and consumed. Finally, we have presented a general framework in the Coq proof assistant that allows one to define and compare access control policies that can be enforced by (variants of) this generalized security model.

Chapter 3

Formally verifying security properties in an idealized model of virtualization

3.1 Introduction

Virtualization is a prominent technology that allows high-integrity, safety-critical, systems and untrusted, non-critical, systems to coexist securely on the same platform and efficiently share its resources. To achieve the strong security guarantees requested by these application scenarios, virtualization platforms impose a strict control on the interactions between their guest systems. While this control theoretically guarantees isolation between guest systems, implementation errors and side-channels often lead to breaches of confidentiality, allowing a malicious guest system to obtain secret information, such as a cryptographic key, about another guest system.

Over the last few years, there have been significant efforts to prove that virtualization platforms deliver the expected, strong, isolation properties between operating systems. The most prominent efforts in this direction are within the Hyper-V [14, 15] and L4.verified [16] projects, which aim to derive strong guarantees for concrete implementations: more specifically, Murray *et al* [33] recently presented a machine-checked information flow security proof for the seL4 microkernel. In comparison with the Hyper-V and L4.verified projects, our proofs are based on an axiomatization of the semantics of a hypervisor, and abstract away many details from the implementation; on the other hand, our model integrates caches and Translation Lookaside Buffers (TLBs), two security relevant components that are not considered in these works.

There are three important isolation properties for virtualization platforms. On the one hand, read and write isolation respectively state that guest operating systems cannot read and write on memory that they do not own. On the other hand, OS isolation states that the behavior of a guest operating system does not depend on the previous behavior and does not affect the future behavior of other operating systems. In contrast to read and write isolation, which are safety properties and can be proved with deductive verification methods, OS isolation is a 2-safety property [17, 18] that requires reasoning about two program executions. Unfortunately, the technology for verifying 2-safety properties is not fully mature, making their formal verification on large and complex programs exceedingly challenging.

3.1.1 A primer on virtualization

This section provides a primer on virtualization, focusing on the elements that are most relevant for our formal model, which we develop in Section 3.2.

Virtualization is a technique used to run on the same physical machine multiple operating systems, called *guest operating systems*. The hypervisor, or Virtual Machine Monitor [92], is a thin layer of software that manages the shared resources (e.g. CPU, system memory, I/O devices). It allows guest operating systems to access these resources by providing them an abstraction of the physical machine on which they run. One of the most important features of a virtualization platform is that its OSs run isolated from one another. In order to guarantee isolation and to keep control of the platform, a hypervisor makes use of the different execution modes of a modern CPU: the hypervisor itself and trusted guest OSs run in supervisor mode, in which all CPU instructions are available; while untrusted guest operating systems will run in user mode in which privileged instructions cannot be executed.

Historically there have been two different styles of virtualization: *full virtualization* and *paravirtualization*. In the first one, each virtual machine is an exact duplicate of the underlying hardware, making it possible to run unmodified operating systems on top of it. When an attempt to execute a privileged instruction by the OS is detected the hardware raises a trap that is captured by the hypervisor and then it emulates the instruction behavior. In the paravirtualization approach, each virtual machine is a simplified version of the physical architecture. The guest (untrusted) operating systems must then be modified to run in user CPU mode, changing privileged instructions to hypercalls, i.e. calls to the hypervisor. A hypercall interface allows OSs to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page table updates, in which the hypervisor validates and applies a list of updates, returning control to the calling OS when this is completed.

In this work, we focus on the memory management policy of a paravirtualization style hypervisor, based on the Xen virtualization platform [93]. Several features of the platform are not yet modeled (e.g. I/O devices, interruption system, or the possibility to execute on multi-cores), and are left as future work.

3.1.2 Contents of the chapter

This Chapter builds upon and extends the previously published papers [94, 95]. In Section 3.2 we develop our formal model. Isolation properties are considered in Section 3.3, whereas availability is discussed in Section 3.4. Section 3.5 presents an extension of the memory model with cache and TLB. Section 3.6 describes the executable semantics of the hypervisor and outlines the proof of correctness of the implementation. In Section 3.7 we present the isolation theorems for the extended model with execution errors and a proof of transparency. Section 3.8 considers related work, and finally Section 3.9 concludes with a summary of the chapter.

3.1.3 Formalization

The formal development is available at

`http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`

and can be verified using the Coq proof assistant.

3.2 The model

In this section we present and discuss the formal specification of the idealized model. We first introduce the set of states, and the set of actions; the latter include both operations of the hypervisor and of the guest operating systems. The semantics of each action is specified by a precondition and a postcondition. Then, we introduce a notion of valid state and show that state validity is preserved by execution. Finally, we define execution traces.

3.2.1 Informal overview of the memory model

The most important component of the state is the memory model, which we proceed to describe. As illustrated in Figure 3.1, the memory model involves three types of addressing modes and two address mappings: the machine address is the real machine memory; the physical memory is used by the guest OS, and the virtual memory is used by the applications running on an operating system. The *virtual memory* is the one used

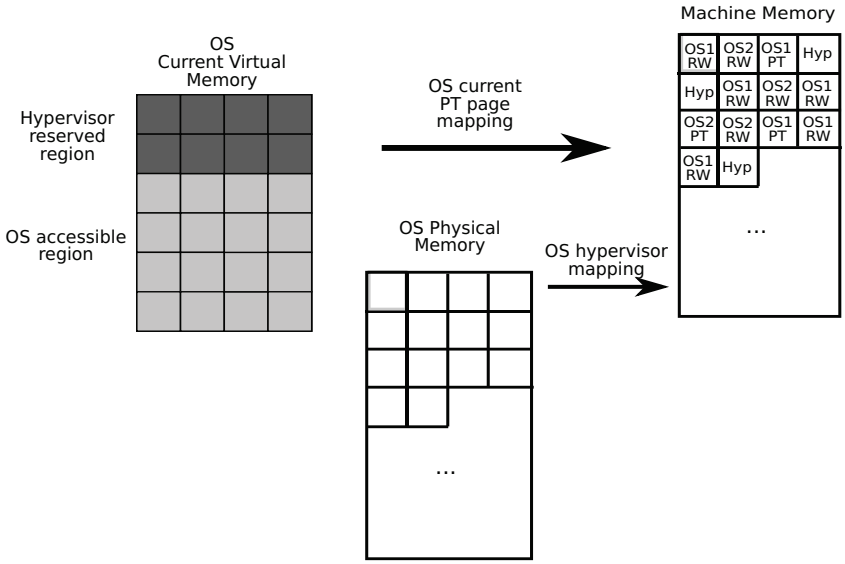

Figure 3.1: Memory model of the platform

by applications running on OSs. Each OS stores a partial mapping of virtual addresses to machine addresses. This will allow us to represent the translation of the virtual addresses of the applications executing in the OS into real hardware addresses. Moreover, each OS has a designated portion of its virtual address space (usually abbreviated VAS) that is reserved for the hypervisor to attend hypercalls. We say that a virtual address *va* is *accessible* by the OS if it belongs to the virtual address space of the OS which is not reserved for the hypervisor. We denote the type of virtual addresses by *vadd*.

The *physical memory* is the one addressed by the kernel of the guest OS. In the Xen [93] platform, this is the type of addresses that the hypervisor exposes to the domains (the untrusted guest OSs in our model). The type of physical addresses is written *padd*.

The *machine memory* is the real machine memory. A mechanism of page classification was introduced in order to cover concepts from certain virtualization platforms, in particular Xen. The model considers that each machine address that appears in a memory mapping corresponds to a memory page. Each page has at most one unique owner, a particular OS or the hypervisor, and is classified either as a data page with read/write access or as a page table, where the mappings between virtual and machine addresses reside. It is required to register (and classify) a page before being able to use or map it. The type of machine addresses is written *madd*.

As to the mappings, each OS has an associated collection of page tables (one for each application executing on the OS) that map virtual addresses into machine addresses. When executed, the applications use virtual addresses, therefore on context switch the current page table of the OS must change so that the currently executing application may be able to refer to its own address space. Neither applications nor untrusted OSs have permission to read or write page tables, because these actions can only be performed in supervisor mode. Every memory address accessed by an OS needs to be associated to a virtual address. The model must guarantee the correctness of those mappings, namely, that every machine address mapped in a page table of an OS is owned by it.

The mapping that associates, for each OS, machine addresses to physical ones is, in our model, maintained by the hypervisor. This mapping might be treated differently by each specific virtualization platform. There are platforms in which this mapping is public and the OS is allowed to manage machine addresses. The physical-to-machine address mapping is modified by the actions `page_pin` and `page_unpin`, as shall be described in Section 3.2.3.

3.2.2 Formalizing states

The platform state consists of a collection of components that we now proceed to describe.

Operating systems

We start from a type *os_ident* of identifiers for guest operating systems, and a predicate *trusted_os* indicating whether a guest operating system is trusted. The state contains information about each guest OS current page table, which is a physical address, and information on whether it has a hypercall pending to be resolved. Formally the information is captured by a mapping *oss_map* that associates OS identifiers with objects of type *os*, where

$$\begin{aligned} os &\stackrel{\text{def}}{=} [\![\ curr\_page : padd, \\ &\qquad hcall : option\ Hyper\_call\ ]\!] \\ oss\_map &\stackrel{\text{def}}{=} os\_ident \ \mapsto \ os \end{aligned} \tag{3.1}$$

Execution modes

Most hardware architectures distinguish at least two execution modes, namely *user mode* (usr) and *supervisor mode* (svc). These modes are used as a protection mechanism, where *privileged* instructions are only allowed to be executed in supervisor mode. In our model, untrusted OSs execute in user mode while trusted ones and the hypervisor execute in supervisor mode. When an untrusted OS needs to execute a privileged operation, it

requests the hypervisor to do it on its behalf. Execution modes are formalized by the enumerated type *exec_mode*, where

$$exec\_mode \stackrel{\text{def}}{=} usr \mid svc \tag{3.2}$$

Moreover, there is a single active OS in each state. After requesting the hypervisor to execute some service, the active guest OS will turn in processor execution mode *waiting* until the service is completed and the execution control returned, switching then its execution mode to *running*. Active OS execution mode is formalized by the type

$$os\_activity \stackrel{\text{def}}{=} running \mid waiting \tag{3.3}$$

Memory mappings

The mapping, that given an OS returns the corresponding mapping from physical to machine addresses, is formalized as an object of the type *hypervisor_map*, where

$$hypervisor\_map \stackrel{\text{def}}{=} os\_ident \ \mapsto \ (padd \mapsto \ madd) \tag{3.4}$$

The real platform memory is formalized as a mapping that associates to a machine address a page, thus

$$system\_memory \stackrel{\text{def}}{=} madd \mapsto \ page \tag{3.5}$$

A page consists of a page content and a reference to the page owner. Page contents can be either (readable/writable) values, an OS page table or nothing; note that a page might have been created without having been initialized, hence the use of option types. Page owners can be the hypervisor, a guest OS or none. Formally:

$$\begin{array}{lll} content & \stackrel{\text{def}}{=} & RW \ (v : option \ Value) \\ & & \mid \ PT \ (va\_to\_ma : vadd \ \mapsto \ madd) \\ & & \mid \ Other \\ page\_owner & \stackrel{\text{def}}{=} & Hyp \\ & & \mid \ Os \ (osi : os\_ident) \\ & & \mid \ No\_Owner \\ page & \stackrel{\text{def}}{=} & [\![\ page\_content : content, \\ & & \ \ page\_owned\_by : page\_owner \ ]\!] \end{array} \tag{3.6}$$

States

The states of the platform are modeled by a record with six components:

$$\begin{array}{lll} State \stackrel{\text{def}}{=} [\![& active\_os & : os\_ident, \\ & aos\_exec\_mode & : exec\_mode, \\ & aos\_activity & : os\_activity, \\ & oss & : oss\_map, \\ & hypervisor & : hypervisor\_map, \\ & memory & : system\_memory \ ]\!] \end{array} \tag{3.7}$$

$$
\begin{array}{l}
Pre\ s\ (\texttt{read}\ va) \stackrel{\text{def}}{=} \\
\quad os\_accessible(va)\ \wedge\ s.aos\_activity\ =\ running\ \wedge \\
\quad \exists\ ma : madd,\ va\_mapped\_to\_ma(s, va, ma)\ \wedge \\
\quad is\_RW((s.memory[ma]).page\_content) \\
Post\ s\ (\texttt{read}\ va)\ s' \stackrel{\text{def}}{=} s\ =\ s'
\end{array}
$$

$$
\begin{array}{l}
Pre\ s\ (\texttt{write}\ va\ val) \stackrel{\text{def}}{=}\ Pre\ s\ (\texttt{read}\ va) \\
Post\ s\ (\texttt{write}\ va\ val)\ s' \stackrel{\text{def}}{=} \\
\quad \exists\ ma : madd,\ va\_mapped\_to\_ma(s, va, ma)\ \wedge \\
\quad s' = s \cdot [memory := (s.memory[ma := \langle RW(Some\ val), s.active\_os\rangle])]
\end{array}
$$

$$
\begin{array}{l}
Pre\ s\ (\texttt{chmod}) \stackrel{\text{def}}{=} \\
\quad s.aos\_activity\ =\ waiting\ \wedge\ (s.oss[s.active\_os]).hcall\ =\ None \\
Post\ s\ (\texttt{chmod})\ s' \stackrel{\text{def}}{=} \\
\quad trusted\_os(s.active\_os)\ \wedge \\
\qquad s' = s \cdot \left[\begin{array}{rcl} aos\_exec\_mode & := & svc, \\ aos\_activity & := & running \end{array}\right] \vee \\
\quad \neg\ trusted\_os(s.active\_os)\ \wedge \\
\qquad s' = s \cdot \left[\begin{array}{rcl} aos\_exec\_mode & := & usr, \\ aos\_activity & := & running \end{array}\right]
\end{array}
$$

$$
\begin{array}{l}
Pre\ s\ (\texttt{page\_pin\_untr}\ o\ pa\ t) \stackrel{\text{def}}{=} \\
\quad \neg\ trusted\_os(o)\ \wedge\ s.aos\_activity\ =\ waiting\ \wedge \\
\quad (s.oss[o]).hcall\ =\ Some\ (Hyperv\_call\_pin(pa, t))\ \wedge \\
\quad physical\_address\_not\_allocated(s.hypervisor[o], pa)\ \wedge \\
\quad \exists\ ma : madd,\ memory\_available(s.memory, ma) \\
Post\ s\ (\texttt{page\_pin\_untr}\ o\ pa\ t)\ s' \stackrel{\text{def}}{=} \\
\quad \exists\ ma : madd, memory\_available(s.memory, ma)\ \wedge \\
\quad s' = s \cdot \left[\begin{array}{rcl} oss & := & s.oss[o := \langle None, (s.oss[o]).curr\_page\rangle], \\ hypervisor & := & s.hypervisor[o, pa := ma], \\ memory & := & s.memory[ma := newpage(t, o)] \end{array}\right]
\end{array}
$$

Figure 3.2: Formal specification of actions semantics

The component *active_os* indicates which is the active operating system, and the components *aos_exec_mode* and *aos_activity* the corresponding execution and processor mode. *oss* stores the information of the guest operating systems of the platform. Finally, the components *hypervisor* and *memory* are the mappings used to formalize the memory model described in the previous section.

We define the predicate $os\_accessible(va)$, that holds if va belongs to the set of virtual addresses accessible by any OS.

Valid state

We define a notion of valid state that captures essential properties of the platform. Formally, the predicate *valid_state* holds on state s if s satisfies the following properties:

- a trusted OS has no pending hypercalls;
- if the active OS is in running mode then no hypercall requested by it is pending;

- if the hypervisor or a trusted OS is running the processor must be in supervisor mode;
- if an untrusted OS is running the processor must be in user mode;
- the hypervisor maps an OS physical address to a machine address owned by that same OS. This mapping is also injective;
- all page tables of an OS o map accessible virtual addresses to pages owned by o and not accessible ones to pages owned by the hypervisor;
- the current page table of any OS is owned by that OS;
- any machine address ma which is associated to a virtual address in a page table has a corresponding pre-image, which is a physical address, in the hypervisor mapping.

All properties have a straightforward interpretation in our model. For example, the first property is captured by the proposition:

$$\forall\ osi : os\_ident,\ trusted\_os(osi) \rightarrow (s.oss[osi]).hcall\ =\ None \tag{3.8}$$

Valid states are invariant under execution, as shall be shown later.

3.2.3 Actions

Table 3.1 summarises the set of the actions specified in the model, and their effects. Actions can be classified as follows:

- hypervisor calls `new`, `delete`, `pin`, `unpin` and `lswitch`;
- change of the active OS by the hypervisor (`switch`);
- access, from an OS or the hypervisor, to memory pages (`read` and `write`);
- update of page tables by the hypervisor on demand of an untrusted OS or by a trusted OS directly (`new` and `delete`);
- changes of the execution mode (`chmod`, `ret_ctrl`);
- changes in the hypervisor memory mapping (`pin` and `unpin`), which are performed by the hypervisor on demand of an untrusted OS or by a trusted OS directly. These actions model (de)allocation of resources.

Actions Semantics

The behaviour of actions is specified by a precondition Pre and by a postcondition $Post$ of respective types:

$$\begin{aligned} &Pre : State \rightarrow Action \rightarrow Prop \\ &Post : State \rightarrow Action \rightarrow State \rightarrow Prop \end{aligned} \tag{3.9}$$

Figure 3.2 provides the axiomatic semantics of some relevant actions, namely, `read`, `write`, `chmod` and `page_pin_untr` (the names of the auxiliary predicates used should be self-explanatory). Notice that what is specified is the effect the execution of an action

`read` *va*	A guest OS reads virtual address *va*.
`read_hyper` *va*	The hypervisor reads virtual address *va*.
`write` *va val*	A guest OS writes value *val* in virtual address *va*.
`write_hyper` *va val*	The hypervisor writes value *val* in virtual address *va*.
`new_tr` *va pa*	The virtual address *va* is mapped to the machine address *ma* in the memory mapping of the trusted active OS, where *pa* translates to *ma* for the active OS.
`new_untr` *o va pa*	The hypervisor adds (on behalf of the OS *o*) a new ordered pair (mapping virtual address *va* to the machine address *ma*) to the current memory mapping of the untrusted OS *o*, where *pa* translates to *ma* for *o*.
`new_hyper` *va ma*	The hypervisor adds a new ordered pair to the current memory mapping of the active OS (mapping virtual address *va* to the machine address *ma*) for his own purposes.
`del_tr` *va*	The trusted active OS deletes the ordered pair that maps virtual address *va* from its memory mapping.
`del_untr` *o va*	The hypervisor deletes (on behalf of the *o* OS) the ordered pair that maps virtual address *va* from the current memory mapping of *o*.
`del_hyper` *va*	The hypervisor deletes (for its own purposes) the ordered pair that maps virtual address *va* from the current memory mapping of the active OS.
`switch` *o*	The hypervisor sets *o* to be the active OS.
`lswitch_tr` *pa*	The trusted active OS changes its current memory mapping to be the one located at physical address *pa*. This action corresponds to a traditional context switch by the active OS.
`lswitch_untr` *o pa*	The hypervisor changes the current memory mapping of the untrusted active OS, to be the one located at physical address *pa*.
`silent`	Represents the silent action (the system does not advertise any effects).
`hcall` *c*	An untrusted OS requires privileged service *c* to be executed by the hypervisor.
`ret_ctrl`	Returns the execution control to the hypervisor.
`chmod`	The hypervisor changes the execution mode from supervisor to user mode, if the active OS is untrusted, and gives to it the execution control.
`page_pin_tr` *pa t*	The memory page that corresponds to physical address *pa* (for the active OS) is registered and classified with type *t*.
`page_pin_untr` *o pa t*	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.
`page_unpin_tr` *pa*	The memory page that corresponds to physical address *pa* (for the active OS) is un-registered.
`page_unpin_untr` *o pa*	The memory page that corresponds to physical address *pa* (for the untrusted OS *o*) is un-registered.

Table 3.1: Platform actions

has on the state of the platform. In particular, the action `read` does not return the accessed value.

The precondition of the action `read` *va* requires that *va* is accessible by the active OS, that there exists a machine address *ma* to which *va* is mapped, that the active OS is running and that the page indexed by the machine address *ma* is readable/writable. The postcondition requires the execution of this action to keep the state unchanged.

The precondition of the action `write` is identical to that of the action `read`. The postcondition establishes that the state after the execution of the action only differs in the value (*val*) of the page associated to *ma*, which is owned by the active OS. The precondition of the action `chmod` requires that there must not be a pending hypercall for the active OS. The postcondition establishes that after the execution of the action, if the active OS is a trusted one, then the effect on the state is to change its execution mode to supervisor mode. Otherwise, the execution mode is set to user mode. In both cases, the processor mode is set to *running*.

The execution of the action `page_pin_untr` requires, in the first place, that the hypervisor is running and that the active OS is untrusted. In addition to that, the OS *o* must be waiting for an hypercall to *pin* the physical address *pa* of type *t*, *pa* must not be already allocated and there must be machine memory available. The effect of the action is to create and allocate at machine address *ma* a new page of type *t* whose owner is the OS *o* and bind, in the hypervisor mapping, the physical address *pa* to *ma*. The rest of the state remains unchanged.

One-step execution

The execution of an action is specified by the $\hookrightarrow$ relation:

$$\frac{valid\_state(s) \quad Pre\ s\ a \quad Post\ s\ a\ s'}{s \xhookrightarrow{a} s'} \tag{3.10}$$

Whenever an action occurs for which the precondition holds, the (valid) state may change in such a way that the action postcondition is established. The notation $s \xhookrightarrow{a} s'$ may be read as *the execution of the action a in a valid state s results in a new state* s'. Note that this definition of execution does not consider the cases where the preconditions of the actions are not fulfilled.

Invariance of valid state

One-step execution preserves valid states, that is to say, the state resulting from the execution of an action is also a valid one.

Lemma 3.2.1.

$$\forall\ (s\ s' : State)\ (a : Action),\ s \xhookrightarrow{a} s' \ \rightarrow\ valid\_state(s') \tag{3.11}$$

Platform state invariants, such as state validity, are useful to analyze other relevant properties of the model. In particular, the results presented in this work are obtained from valid states of the platform.

3.2.4 Traces

Isolation properties are eventually expressed on execution traces, rather than execution steps; likewise, availability properties are formalized as fairness properties stating that something good will eventually happen in an execution traces. Thus, our formalization includes a definition of execution traces and proof principles to reason about them.

Informally, an execution trace is defined as a stream (an infinite list) of states that are related by the transition relation $\hookrightarrow$, i.e. an object of the form

$$s_0 \xhookrightarrow{a_0} s_1 \xhookrightarrow{a_1} s_2 \xhookrightarrow{a_2} s_3 \ldots \tag{3.12}$$

such that every execution step $s_i \xhookrightarrow{a_i} s_{i+1}$ is valid. Formally, an execution trace is defined as a stream t of pairs of states and actions, such that for every $i \geq 0$, $s[i] \xhookrightarrow{a[i]} s[i+1]$, where $t[i] = \langle s[i], a[i] \rangle$ and $t[i+1] = \langle s[i+1], a[i+1] \rangle$. We let $Trace$ define the type of traces.

State properties are lifted to properties on pairs of states and actions in the obvious way. Moreover, state properties can be lifted to properties on traces; formally, each predicate P on states can be lifted to predicates $\Box\ P$ (read always P) and $\Diamond\ P$ (read eventually P) in a temporal logic as Computation Tree Logic (CTL) [96, 97], formalized in the CIC (in Coq) [98]. The former $\Box\ P$ is defined co-inductively defined by the clause $\Box(P, s :: t)$ iff $P(s)$ and $\Box(P, t)$, whereas the latter $\Diamond\ P$ is defined inductively by the clauses $\Diamond(P, s :: t)$ iff $P(s)$ or $\Diamond(P, t)$; each modality has an associated reasoning principle attached to its definition. Similar modalities can be defined for relations, and can be used to express isolation properties. In particular, given a relation R on states, and two traces t_1 and t_2, we have $\Box(R, t_1, t_2)$ iff $R(t_1[i], t_2[i])$ for all i.

3.3 Isolation properties

We formally establish that the hypervisor enforces strong isolation properties: an operating system can only read and modify memory that it owns, and its behavior is independent of the state of other operating systems. The properties are established for a single step of execution, and then extended to traces.

Read isolation

Read isolation captures the intuition that no OS can read memory that does not belong to it. Formally, read isolation states that the execution of a `read` va action requires that va is mapped to a machine address ma that belongs to the active OS current memory mapping, and that is owned by the active OS.

Lemma 3.3.1.

$$\begin{array}{l} \forall\ (s\ s' : State)\ (va : vadd), \\ s \xhookrightarrow{read\ va} s'\ \rightarrow\ \exists\ ma : madd,\ va\_mapped\_to\_ma(s, va, ma)\ \wedge \\ \exists\ pg : page,\ pg = s.memory[ma]\ \wedge\ pg.page\_owned\_by = s.active\_os \end{array} \tag{3.13}$$

The property is proved by inspection of the pre and postcondition for the `read` action, using the definition of valid state.

Write Isolation

Write isolation captures the intuition that an OS cannot modify memory that it does not own. Formally, write isolation states that, unless the hypervisor is running, the execution of any action will at most modify memory pages owned by the active OS or it will allocate a new page for that OS.

Lemma 3.3.2.

$$\begin{aligned}&\forall\ (s\ s' : State)\ (a : Action)\ (ma : madd),\\ &s \xrightarrow{a} s'\ \rightarrow\ \neg hyper\_running(s)\ \rightarrow\\ &s'.memory[ma] = s.memory[ma] \vee owner\_or\_free(s.memory, ma, s.active\_os)\end{aligned} \tag{3.14}$$

where *hyper_running* and *owner_or_free* respectively denote that the hypervisor is running, and that the owner of the given machine address is either the given OS or it is free.

The property is proved by case analysis on the action executed. The relevant cases are the actions that are performed by the active OS and that modify the memory; for each such action, the property follows from its pre and postconditions, and from the definition of valid state.

OS Isolation

OS isolation is a 2-safety property [17, 18], cast in terms of two executions of the system, and is closely related to the non-influence property studied by Oheimb and coworkers [34, 99].

OS isolation captures the intuition that the behavior of any OS does not depend on other OSs states, and is expressed using the notion of *equivalence* w.r.t. an operating system *osi*. Formally, two states s and s' are *osi-equivalent*, denoted $s \equiv_{osi} s'$, if the following conditions are satisfied:

- *osi* is the active OS in both states and the processor mode is the same, or the active OS is different to *osi* in both states;
- *osi* has the same hypercall in both states, or no hypercall in both states;
- the current page tables of *osi* are the same in both states;
- all page table mappings of *osi* that maps a virtual address to a RW page in one state, must map that address to a page with the same content in the other;
- the hypervisor mappings of *osi* in both states are such that if a given physical address maps to some RW page, it must map to a page with the same content on the other state.

All conditions have a straightforward interpretation in our model. The first condition is formalized by the proposition:

$$\begin{aligned}&(s.active\_os = osi \wedge s'.active\_os = osi \wedge s.aos\_activity = s'.aos\_activity) \vee\\ &(s.active\_os \neq osi \wedge s'.active\_os \neq osi)\end{aligned} \tag{3.15}$$

The following two conditions are captured by the proposition:

$$s.oss[osi] = s'.oss[osi] \tag{3.16}$$

Finally, the last two conditions are formally represented by the sentence:

$$(os\_equivalent\_PT\_pages\ s\ s'\ osi) \wedge (os\_equivalent\_hyper\_mapping\ s\ s'\ osi) \tag{3.17}$$

where *os_equivalent_PT_pages* and *os_equivalent_hyper_mapping* are formally defined in `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`. In particular, *os_equivalent_hyper_mapping* s s' *osi* holds iff $s \sim_{osi}^{hyp} s'$ and $s' \sim_{osi}^{hyp} s$, where: $s_1 \sim_{osi}^{hyp} s_2$ iff

$$\begin{array}{l} \forall\ (pa : padd)\ (pg : page), \\ (\exists\ (ma : madd), s_1.hypervisor[osi][pa] = ma \wedge s_1.memory[ma] = pg) \rightarrow \\ is\_RW(pg.page\_content) \rightarrow \\ \exists\ (ma' : madd), s_2.hypervisor[osi][pa] = ma' \wedge s_2.memory[ma'] = pg \end{array} \tag{3.18}$$

Note that we cannot require that memory contents be the same in both states for them to be *osi*-equivalent, because on a `page_pin` action, the hypervisor can assign an arbitrary (free) machine address to the OS, so we consider *osi*-equivalence without taking into account the actual value of the machine addresses assigned. In particular, two *osi*-equivalent states can have different page table memory pages, which contain mappings from virtual to arbitrary machine addresses, but such that the contents of such an arbitrary machine address be the same on both states, if it corresponds to a RW page. This definition bears some similarity with notions of indistinguishable states used for reasoning about non-interference in object-oriented languages [100].

OS isolation states that *osi*-equivalence is preserved under execution of any action, and is formalized as a "step-consistent" unwinding lemma, see [101].

Lemma 3.3.3.

$$\begin{array}{l} \forall\ (s_1\ s_1'\ s_2\ s_2' : State)\ (a : Action)\ (osi : os\_ident), \\ s_1 \equiv_{osi} s_2\ \rightarrow s_1 \overset{a}{\hookrightarrow} s_1'\ \rightarrow\ s_2 \overset{a}{\hookrightarrow} s_2'\ \rightarrow\ s_1' \equiv_{osi} s_2' \end{array} \tag{3.19}$$

The proof of OS isolation relies on write isolation, on Lemma 3.3.4, and on an isolation lemma for the case where *osi* is the active OS of both states s_1 and s_2.

The next lemma formalizes a "locally preserves" unwinding lemma in the style of [101], stating that the *osi*-component of a state is not modified when another operating system is executing.

Lemma 3.3.4.

$$\begin{array}{l} \forall\ (s\ s' : State)\ (a : Action)\ (osi : os\_ident), \\ \neg\ os\_action(s, a, osi)\ \rightarrow\ s \overset{a}{\hookrightarrow} s'\ \rightarrow\ s \equiv_{osi} s' \end{array} \tag{3.20}$$

where $os\_action(s, a, osi)$ holds if, in the state s, *osi* is the active and running OS and therefore is executing action a, or otherwise the hypervisor is executing the action a on behalf of *osi*.

Extensions to traces

All isolation properties extend to traces, using coinductive reasoning principles. In particular, the extension of OS isolation to traces establishes a non-influence property [34]. Formally, we define for each operating system *osi* a predicate *same_os_actions* stating that two steps have the same set of actions w.r.t. *osi*: concretely, if for all i the actions in $t_1[i]$ and $t_2[i]$ are the same *os_action* for *osi*, or both are arbitrary actions not related to *osi*, then $same\_os\_actions(osi, t_1, t_2)$ holds.

Lemma 3.3.5.

$$\begin{array}{l}\forall\ (t_1\ t_2 : Trace)\ (osi : os\_ident),\\ same\_os\_actions(osi, t_1, t_2) \rightarrow (t_1[0] \equiv_{osi} t_2[0]) \rightarrow\\ \Box(\equiv_{osi}, t_1, t_2)\end{array} \tag{3.21}$$

For technical reasons related to the treatment of coinductive definitions in Coq (specifically the need for corecursive definitions to be productive), our formalization of non-influence departs from common definitions of non-interference and non-influence, which rely on a purge function that eliminates the actions that are not related to *osi*. One can however define an erasure function *erase* that replaces actions that are not related to *osi* by `silent` actions, and prove for all traces t that

$$\Box(\equiv_{osi}, t, erase(osi, t)) \tag{3.22}$$

3.4 Availability

An essential property of virtualization platforms is that all guest operating systems are given access to the resources they need to proceed with their execution. In this section, we establish a strong fairness property, showing that if the hypervisor only performs `chmod` actions whenever no hypercall is pending, then no OS blocks indefinitely waiting for its hypercalls to be attended. The assumption on the hypervisor is satisfied by all reasonable implementations of the hypervisor; one possible implementation that would satisfy this restriction is an eager hypervisor which attends hypercalls as soon as it receives them and then chooses an operating system to run next. If this is the case, then when the `chmod` action is executed, no hypercalls are pending on the whole platform.

Formally, the assumption on the hypervisor is modelled by considering a restricted set of execution traces in which the initial state has no hypercall pending, and in `chmod` actions can only be performed whenever no hypercall is pending. Then, the strong fairness property states that: if the hypervisor returns control to guest operating systems infinitely often, then infinitely often there is no pending hypervisor call.

Lemma 3.4.1.

$$\begin{array}{l}\forall\ (t : Trace), \neg\ hcall(t[0]) \rightarrow\\ \Box(chmod\_nohcall, t) \rightarrow\\ \Box(\Diamond\ \neg\ hyper\_running, t) \rightarrow\\ \Box(\Diamond\ \neg\ hcall, t)\end{array} \tag{3.23}$$

where *hcall* and *chmod_nohcall* respectively denote that there is an hypercall pending and that `chmod` actions only arise when no hypercall is pending.

The proof of the strong fairness property proceeds by co-induction and relies on showing that $\neg\ hyper\_running(s) \rightarrow \neg\ hcall(s)$ is an invariant of all traces that satisfy the hypothesis of the lemma.

Note that our strong fairness property is independent of the scheduler: in particular, the hypothesis $\Box(\Diamond\ \neg\ hyper\_running, t)$ does not guarantee that each operating system will be able to execute infinitely often. Further restricting the implementation of the hypervisor so as to guarantee that the hypervisor is fair to each guest operating system is left for future work.

3.5 Extension of the model with cache and TLB

In this section, we present an extension of the idealized model of virtualization, introduced in Section 3.2, that features cache and TLB. We show the new formulation of the set of state and the semantics of the actions of state transformers. We start by providing some background on cache managment, according to [102].

3.5.1 Cache management

In order to reduce the overhead necessary to access memory values, CPUs use faster storage devices called caches to store a copy of the data from main memory. When the processor needs to read a memory value, it fetches the value from the cache if it is present (cache hit), or from memory if it is not (cache miss). In systems with virtual memory, the TLB is analogously used to speed up the translation from virtual to physical memory addresses. In the event of a TLB hit, the physical address corresponding to a given virtual address is obtained directly, instead of searching for it in the process page table (which resides in main memory).

The cache may potentially be accessed either with a physical address (physically indexed cache) or a virtual address (virtually indexed cache). There are, in turn, two variants of virtually indexed caches: those where the entries are tagged with the virtual address (virtually indexed, virtually tagged or VIVT cache) and those which are tagged with the corresponding physical address (virtually indexed, physically tagged or VIPT cache).

There exist several alternatives policies for implementing cache content management, in particular concerning the update and replacement of cache information. A replacement policy is one that specifies the criteria used to remove a value when the cache is full and a new value needs to be stored. Among the most common policies we find those that specify that the value to be removed is either the least recently used value (LRU), the most recently used (MRU) or the least frequently used (LFU). In our model we specify an abstract replacement policy which can be refined to any of the three policies just described. A write policy specifies how the modification of a cache entry impacts the memory structures: a *write-through* policy, for instance, requires that values are written both in the cache and in the main memory. A *write-back* policy, on the other side, requires that values are only modified in the cache and marked dirty, and updates to main memory are performed when a dirty entry is removed from the cache.

In this work we provide a model of a VIVT cache and TLB (as in Xen on ARM [103]), for a write-through policy.

3.5.2 States

Figure 3.3 shows a diagram of the extended memory model of the platform. It involves the already presented three types of addresses. The figure also shows the cache and the TLB. The cache is indexed with virtual addresses and holds a (partial) copy of the memory pages. The TLB is used in conjunction with the current page table of the active OS to map virtual to machine addresses.

The type of states of the platform is defined as an extension of the set *State* of Section 3.2.2 including two additional components that model the cache and the TLB. Formally,

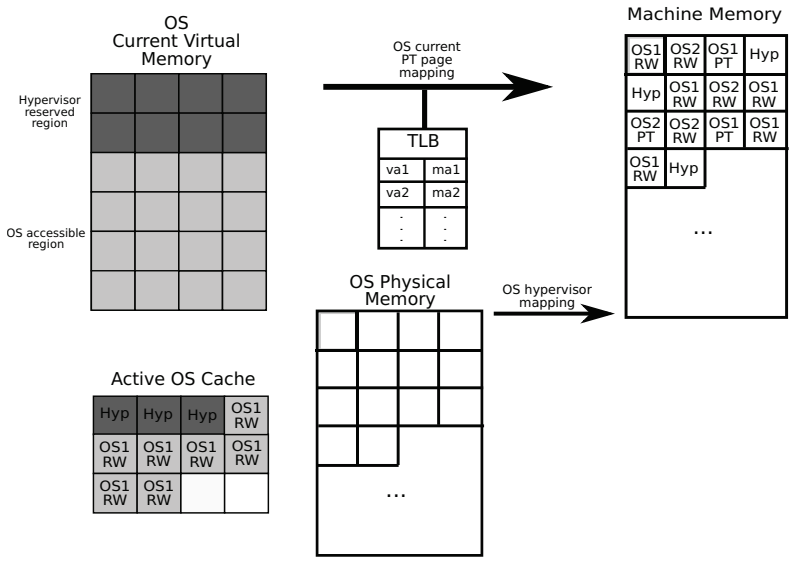

Figure 3.3: Memory model of the platform with cache and TLB

$$
\begin{aligned}
State \stackrel{\text{def}}{=} [\![\ & oss && : oss\_map,\\
& active\_os && : os\_ident,\\
& mode && : exec\_mode,\\
& activity && : os\_activity,\\
& hypervisor && : hypervisor\_map,\\
& memory && : machine\_memory,\\
& cache && : cache\_vivt,\\
& tlb && : tlb\_struct\ ]\!]
\end{aligned}
\tag{3.24}
$$

Thus, the remaining two components of the state, the *cache* and the *tlb*, are modelled as partial maps, whose domains are bounded in size with positive fixed constants *size_cache* and *size_tlb*:

$$
\begin{aligned}
cache\_vivt &\stackrel{\text{def}}{=} vadd \ \mapsto_{size\_cache}\ page\\
tlb\_struct &\stackrel{\text{def}}{=} vadd \ \mapsto_{size\_tlb}\ madd
\end{aligned}
\tag{3.25}
$$

The notion of valid state is formalized by a predicate *valid_state* that captures essential properties of the platform and extends the one presented in Section 3.2.2. The following are examples of properties verified by valid states that relate to cache and TLB: i) the size of the cache and TLB mappings does not exceed their bound; ii) if *pg* is associated to *va* in the data cache, then *va* must be translated to some machine address *ma* in the active memory mapping, and $s.memory[ma] = pg$; iii) if *va* is translated into *ma* according to the TLB, then the machine address *ma* is associated to *va* in the active memory mapping; where the active memory mapping is the current memory mapping of the active OS. Notice that these conditions are necessary to ensure the main memory has been updated according to the write-through policy.

3.5.3 Actions semantics

As in Section 3.2.3, the behaviour of actions in the extended model is specified by a precondition *Pre* and by a postcondition *Post*. For instance, Figure 3.4 provides the axiomatic semantics of the `write` action.

$$
\begin{array}{l}
Pre\ s\ (\texttt{write}\ va\ val) \stackrel{\text{def}}{=} \\
\quad \exists\ (ma : madd), \\
\quad va\_mapped\_to\_ma(s, va, ma) \wedge \\
\quad is\_RW(s.memory[ma].page\_content) \wedge \\
\quad s.activity = running \\
\quad os\_accessible(va) \wedge \\
\\
Post\ s\ (\texttt{write}\ va\ val)\ s' \stackrel{\text{def}}{=} \\
\quad \exists\ (ma : madd), \\
\quad va\_mapped\_to\_ma(s, va, ma) \wedge \\
\quad \texttt{let}\ pg\ :=\ s.memory[ma]\ \texttt{in} \\
\quad \texttt{let}\ new\_pg\ :=\ \langle RW(Some\ val), pg.page\_owned\_by\rangle\ \texttt{in} \\
\quad s' = s \cdot \left[\begin{array}{rcl} memory & := & s.memory[ma := new\_pg], \\ cache & := & cache\_add(fix\_cache\_syn(s, s.cache, ma), va, new\_pg), \\ tlb & := & tlb\_add(s.tlb, va, ma) \end{array} \right]
\end{array}
$$

Figure 3.4: Axiomatic specification of action `write`

The precondition of the action `write` *va val* says that there exists a machine address *ma* such that *va* is associated to it (*va_mapped_to_ma*) and that the page associated to it in the memory is readable/writable (*is_RW*); that the guest OS activity must be running; and that *va* must be accessible by the active guest OS (*os_accessible*). Its postcondition sets up that the only variations in the state after executing this action can be produced in the value of the page associated to *ma* in memory, and in the values stored in the cache and the TLB. It is not hard to see that, as the cache uses a write-through policy, both the memory and the cache are updated when a write is performed. A cache c_2 is the result of updating a cache c_1 with a pair *va* and *pg*, written $c_2 = cache\_add(c_1, va, pg)$, iff

$$
\begin{array}{l}
pg\ =\ c_2[va] \wedge \\
\forall\ (va' : vadd)\ (pg' : page),\ va\ \neq\ va'\ \rightarrow\ pg'\ =\ c_2[va']\ \rightarrow\ pg'\ =\ c_1[va']
\end{array}
\tag{3.26}
$$

The definition of $c_2 = tlb\_add(c_1, va, ma)$ is analogous. Moreover, in order to avoid aliasing problems we fix synonyms before adding a new entry into the cache using the function fix_cache_syn. The result of $fix\_cache\_syn(s, c_1, ma)$ is a cache c_2 whose indexes (virtual addresses) are translated to machine addresses ma' which differ from ma in s. We recall that we are modeling a VIVT cache.

Action	Failure	Error Code
`write` *va val*	$s.aos\_activity \neq running$	$wrong\_os\_activity$
	$\neg\ va\_mapped\_to\_ma(s, va, ma)$	$invalid\_vadd$
	$\neg\ os\_accessible(va)$	$no\_access\_va\_os$
	$\neg\ is\_RW(s.memory[ma].page\_content)$	$wrong\_page\_type$
`new_tr` *va pa*	$s.aos\_activity \neq running$	$wrong\_os\_activity$
	$\neg\ os\_accessible(va)$	$no\_access\_va\_os$
	$\neg\ trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ page\_of\_OS(s.active\_os, pa, ma)$	$wrong\_owner$
`lswitch_untr` *osi pa*	$s.aos\_activity \neq waiting$	$wrong\_os\_activity$
	$trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ is\_PT(s.memory[ma].page\_content)$	$wrong\_page\_type$
	$\neg\ lswitch\_hypercall(s.oss[osi].hcall)$	$wrong\_pending\_hcall$
	$\neg\ page\_of\_OS(s.activeos, pa, ma)$	$wrong\_owner$
`page_unpin_untr` *osi pa*	$s.aos\_activity \neq waiting$	$wrong\_os\_activity$
	$trusted\_os(osi)$	$os\_trust\_failure$
	$\neg\ page\_unpin\_hypercall(s.oss[osi].hcall)$	$wrong\_pending\_hcall$
	$\neg\ pa\_not\_curr\_page(s, s.oss, pa)$	$wrong\_curr\_page\_add$
	$s.hypervisor[osi][pa] \neq\ ma$	$invalid\_madd$
	$\neg\ no\_va\_mapped\_to\_ma(s, osi, ma)$	$invalid\_vadd$

Table 3.2: Preconditions and error codes

3.6 Verified implementation

In this section we present an extension of the model with execution errors. We describe the executable specification and show that it constitutes a correct implementation of the behavior specified by the idealized model, with cache and TLB.

3.6.1 Error management

There can be attempts to execute an action on a state that does not verify the precondition of that action. In the presence of one such situation the system answers with a corresponding error code. These error codes are defined in our model by the enumerated type $ErrorCode$.

We define the relation between an error code and the unfulfilled precondition of an action with the predicate $ErrorMsg$. Formally,

$$ErrorMsg : State \to Action \to ErrorCode \to Prop \tag{3.27}$$

where $ErrorMsg\ s\ a\ ec$ means that the execution of the action a in the state s generates the error ec. In Table 3.2 we show some examples about error codes associated to unverified preconditions of some actions of our model. Notice that in the case of the `write` action, for instance, to each of the propositions that compose the precondition of that action there corresponds an element of $ErrorCode$ that indicates the failure of the state s to satisfy that proposition.

Executions with error management

Executing an action a over a state s produces a new state s' and a corresponding answer r (denoted $s \xrightarrow{a/r} s'$), where the relation between the former state and the new one is

given by the postcondition relation $Post$.

$$\frac{valid\_state(s) \qquad Pre(s,a) \qquad Post(s,a,s')}{s \xrightarrow{a/ok} s'}$$
$$\frac{valid\_state(s) \qquad ErrorMsg(s,a,ec)}{s \xrightarrow{a/error\ ec} s} \tag{3.28}$$

Whenever an action occurs for which the precondition holds, the (valid) state may change in such a way that the action postcondition is established. The notation $s \xrightarrow{a/ok} s'$ may be read as *the execution of the action a in a valid state s results in a new state s′*. However, if the precondition is not satisfied, then the state s remains unchanged and the system answer is the error message determined by the relation $ErrorMsg$.

Formally, the possible answers of the system are defined by the following type:

$$\begin{aligned} Response \stackrel{\text{def}}{=} & \ ok : Response \\ & \mid error : ErrorCode \rightarrow Response \end{aligned} \tag{3.29}$$

where *ok* is the answer resulting from a successful execution of an action.

One-step execution with error management preserves valid states, that is to say, the state resulting from the execution of an action is also a valid one.

Lemma 3.6.1.

$$\begin{aligned} & \forall\ (s\ s' : State)(a : Action)(r : Response), \\ & valid\_state(s)\ \rightarrow s \xrightarrow{a/r} s' \rightarrow valid\_state(s') \end{aligned} \tag{3.30}$$

3.6.2 Executable specification

The executable specification of the hypervisor has been written using the Coq proof assistant and it ultimately amounts to the definition of functions that implement action execution. The functions have been defined so as to conform to the axiomatic specification of action execution as provided by the idealized model. The implementation of the hypervisor consists of a set of `Coq` functions, such that for every predicate involved in the axiomatic specification of action execution there exists a function which stands for the functional counterpart of that predicate. An important characteristics of our formalization is that the definition of state that is used for defining the executable semantics of the hypervisor is exactly the same as the one introduced in the idealized model. This simplifies the formal proof of soundness between the inductive and the functional semantics of the hypervisor. The execution of the virtualization platform consists of a (potentially infinite) sequence of action executions starting in an (initial) platform state. The output of the execution is the corresponding sequence of memory states (the trace of execution) obtained while executing the sequence of actions.

Action execution

The execution of actions has been implemented as a *step* function, that given a memory state s and an action a invokes the function that implements the execution of a in s,

which in turn returns an object *res* of type *Result*:

$$Result \stackrel{\text{def}}{=} [\![\ resp\ :\ Response, st\ :\ State\ ]\!] \tag{3.31}$$

where *res.resp* is either an error code *ec*, if the precondition of the actions does not hold in state *s*, or otherwise the value *ok*, and the state *res.st* represents the execution effect. The *step* function acts basically as an action dispatcher. Figure 3.5, which shows the structure of the dispatcher, details the branch corresponding to the dispatching of action `write`, which is the action we shall use along this section to illustrate the working of the implementation. The functions invoked in the branches, like $write\_safe$,

Definition *step s a* :=
  match *a* **with**
    | ... ⇒ ...
    | *Write va val* ⇒ $write\_safe(s, va, val)$
    | ... ⇒ ...
  end.

Figure 3.5: The *step* function

are state transformers whose definition follows this pattern: first it is checked whether the precondition of the action is satisfied in state *s*, and then, if that is the case, the function that implements the execution of the action is invoked, otherwise, the state *s*, unchanged, is returned along with an appropriate response.

In Figure 3.6 we show the definition of the function that implements the execution of the `write` action. The Coq code of this function, together with that of the remaining functions, can be found in [104].

Definition $write\_safe\ (s : state)\ (va : vadd)\ (val : value) : Result$:=
  match $write\_pre(s, va, val)$ **with**
    | *Some ec* ⇒ $\langle error(ec), s\rangle$
    | *None* ⇒ $\langle ok, write\_post(s, va, val)\rangle$
  end.

Figure 3.6: Execution of `write` action

The function *write_pre* is defined as the nested validation of each of the properties of the precondition (see Figure 3.7). The function *write_post*, shown in Figure 3.8, implements the expected behavior of the `write` action: when a new value has to be written in a certain virtual address *va*, first it must be checked whether *va* is in the cache (i.e. is an index of the cache). If that is the case, then the function updates both the cache and the memory, because it implements a write-through policy. Otherwise, i.e. if the virtual address *va* is not already in the cache, the machine address associated to *va* has to be determined in order to write the new value in memory. First, the TLB is inspected to check whether *va* has already been translated. If there is a translation of *va* in the TLB, then the machine address is used to update the memory and the new entry $\langle va,\ new\_pg\rangle$ is added to the cache. If there is no translation of *va* in the TLB, then the corresponding machine address has to be recovered using the current page table of

```
Definition write_pre (s : state) (va : vadd) (val : value) : option ErrorCode :=
  match get_os_ma(s, va) with
    | None ⇒ Some invalid_vadd
    | Some ma
      ⇒ match page_type(s.memory, ma) with
          | Some RW
            ⇒ match aos_activity(s) with
                | Waiting ⇒ Some wrong_os_activity
                | Running
                  ⇒ if vadd_accessible(s, va)
                      then None
                      else Some no_access_va_os
                end
          | _ ⇒ Some wrong_page_type
        end
  end.
```

Figure 3.7: Validation of `write` action precondition

the active guest OS. Once that translation has been found, the memory is updated, the new entry $\langle va,\ new\_pg \rangle$ is added to the cache and the corresponding translation of va is added to the TLB.

Cache and TLB update

In the axiomatic semantics of cache and TLB management the replacement policy has been left abstract. For the execution semantics we have chosen to implement a simple FIFO replacement mechanism. However, this behavior is encapsulated in the definition of the functions $fcache\_add$ and $ftlb\_add$, which implement cache and TLB replacement, respectively. Therefore, for the implementation of an alternative replacement policy it suffices to modify correspondingly these two functions leaving the rest of the code unchanged.

Figure 3.9 shows the definition of the $fcache\_add$ function: first, it is checked whether the virtual address va is the index of an entry of the cache c (map_valid_index). If this is the case, it suffices to perform a simple update of c with the page pg (caches are implemented as bounded maps of virtual addresses to machine addresses). Otherwise, the behaviour of the function depends on whether c has room for a new entry or it is full (is_full_cache). If c is full, the cache update, and entry eviction, is handled using the FIFO replacement algorithm ($fifo\_replace$). If there is room left for a new entry, then c must be updated following the FIFO replacement algorithm guidelines for adding new entries in the cache ($fifo\_add$).

The definitions of the replacement and update function for the TLB are analogous.

3.6.3 Soundness

We proceed now to outline the proof that the executable specification of the hypervisor correctly implements the axiomatic model. It has been formally stated as a soundness theorem and verified using the Coq proof assistant.

Theorem 3.6.2.

$$\begin{array}{l} \forall\ (s : State)\ (a : Action),\ valid\_state(s) \rightarrow \\ s \xrightarrow{a/step(s,a).resp} step(s,a).st \end{array} \tag{3.32}$$

The proof of this theorem follows by, in the first place, performing a case analysis on $Pre(s, a)$ (this predicate is decidable) and then: if $Pre(s, a)$ applying Lemma 3.6.3; otherwise applying Lemma 3.6.5.

Lemma 3.6.3.

$$\begin{array}{l} \forall\ (s : State)\ (a : Action), \\ valid\_state(s) \rightarrow\ Pre(s,a)\ \rightarrow \\ s \xrightarrow{a/ok} step(s,a).st\ \wedge\ step(s,a).resp = ok \end{array} \tag{3.33}$$

The proof of Lemma 3.6.3 proceeds by applying functional induction on $step(s, a)$ and then by providing the corresponding proof of soundness of the function that implements the execution of each action. Thus, in the case of the action `write` we have stated and proved Lemma 3.6.4. This lemma, in turn, follows by performing a case analysis on the result of applying the function *write_pre* on s and the action: if the result is an error code then the thesis follows by contradiction. Otherwise, it follows by the correctness of the function *write_post*.

Lemma 3.6.4.

$$\begin{array}{l} \forall\ (s : State)\ (va : vadd)\ (val : value), \\ valid\_state(s) \rightarrow\ Pre(s, (\texttt{write}\ va\ val))\ \rightarrow \\ Post(s, (\texttt{write}\ va\ val), write\_post(s, va, val)) \end{array} \tag{3.34}$$

As to Lemma 3.6.5, the proof also proceeds by first applying functional induction on $step(s, a)$. Then, for each action a, it is shown that if $\neg Pre(s, a)$ the execution of the function that implements that action yields the values returned by the branch corresponding to the case that the function that validates the precondition of the action a in state s fails, i.e., an error code ec and the (unchanged) state s.

Lemma 3.6.5.

$$\begin{array}{l} \forall\ (s : State)\ (a : Action), \\ valid\_state(s) \rightarrow\ \neg Pre(s,a) \rightarrow \\ \exists\ (ec : ErrorCode), step(s,a).st = s\ \wedge \\ step(s,a).resp = error(ec)\ \wedge\ ErrorMsg(s,a,ec) \end{array} \tag{3.35}$$

```
Definition write_post (s : state) (va : vadd) (val : value) : state :=
  match s.cache[va] with
  | Value old_pg ⇒
    let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
    let val_ma := va_mapped_to_ma_system(s, va) in
    match val_ma with
    | Value ma ⇒
      s · [ mem := s.memory[ma := new_pg],
            cache := fcache_add(fix_cache_syn(s, s.cache, ma), va, new_pg) ]
    | Error _ ⇒ s
    end
  | Error _ ⇒
    match s.tlb[va] with
    | Value ma ⇒
      match s.memory[ma] with
      | Value old_pg ⇒
        let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
        s · [ mem := s.memory[ma := new_pg],
              cache := fcache_add(fix_cache_syn(s, s.cache, ma), va, new_pg) ]
      | Error _ ⇒ s
      end
    | Error _ ⇒
      match va_mapped_to_ma_currentPT(s, va) with
      | Value ma ⇒
        match s.memory[ma] with
        | Value old_pg ⇒
          let new_pg := Page (RW_c (Some val)) (page_owned_by old_pg) in
          s · [ mem := s.memory[ma := new_pg],
                cache := fcache_add(fix_cache_syn(s, s.cache, ma), va, new_pg),
                tlb := ftlb_add(s.tlb, va, ma) ]
        | Error _ ⇒ s
        end
      | Error _ ⇒ s
      end end end.
```

Figure 3.8: Effect of `write` execution

```
Definition fcache_add (c : cache_vivt) (va : vadd) (pg : page) : cache_vivt :=
  if map_valid_index(c, va)
  then map_add(c, va, pg)
  else if is_full_cache(c)
       then fifo_replace(c, va, pg)
       else fifo_add(c, va, pg).
```

Figure 3.9: Cache update

3.7 Isolation and transparency in the extended model

Isolation theorems ensure that the virtualization platform protects guest operating systems against each other, in the sense that a malicious operating system cannot gain information about another victim operating system executing on the same platform. In this section we extend the proof of OS isolation from Section 3.3, yielding modifications in some key technical definitions and lemmas below, so that it accounts for errors in execution traces for the model extended with cache and TLB.

Finally, we present a machine-checked proof of transparency. Transparency states that the virtualization platform is a correct abstraction of the underlying hardware, in the sense that a guest operating system cannot distinguish whether it executes alone or together with other systems. Transparency is a 2-safety property; its formulation involves an erasure function which removes all the components of the states that do not belong to some fixed operating system. We define an appropriate erasure, establish its fundamental properties, and finally derive transparency.

3.7.1 OS isolation

We have mechanically verified three isolation properties:

1. read isolation: no OS can read memory that does not belong to it;
2. write isolation: an OS cannot modify memory that it does not own;
3. OS isolation: the behavior of an OS does not depend on other OSs states.

Read and write isolation are safety properties, whereas OS isolation is a 2-property, i.e. it is cast in terms of two executions of the system. We focus on OS isolation, which is by far the most challenging property.

Informally, as presented in Section 3.3, OS isolation states that starting from states with the same information for an operating system *osi*, *osi* cannot distinguish between the two traces, as long as it executes the same actions in both. This captures the idea that the execution of *osi* does not depend on the state or behaviour of the other systems, even in the presence of erroneous executions.

Note that there is one particular error (the out_of_memory error in [104]) that can in principle influence the execution of an operating system, if during its execution the platform runs out of memory. Since we are specifically interested in modelling observations on states (and the cache, in particular), we treat this error as transparent for the executing operating system, and only make sure it does not modify the state. This is consistent with what usually happens in real implementations, where there are no data leaks from the victims when the platform runs out of memory, and the only information an attacker learns is the total memory consumption of the other operating systems in the platform. Additionally it is possible, in this case, to assign to each guest OS a fixed pool of memory from which to allocate, so whether allocation succeeds or fails for one OS doesn't depend on what any other guest OS does.

To formalize OS isolation we use a notion of state equivalence w.r.t. an operating system *osi*. The definition of *osi-equivalence* ($\equiv_{osi}$), coincides with the one used in [94] and presented in Section 3.3; in particular, it does not mention the cache and the TLB. However, one can prove that it entails some form of cache equivalence and TLB equivalence on valid states. Formally, we define two valid states s_1 and s_2 to be cache

equivalent for *osi*, written $s_1 \equiv_{osi}^{cache} s_2$, iff either *osi* is not active in both states, or it is active and the caches hold equal values for all accessible virtual addresses *va* that are in the domain of the cache of both states, i.e. for all virtual address *va* and pages p_1 and p_2:

$$os\_accessible(va) \ \rightarrow \ s_1.cache[va] = p_1 \ \rightarrow \ s_2.cache[va] = p_2 \ \rightarrow \ p_1 = p_2 \tag{3.36}$$

Note that we do not require that the domains of both caches coincide, as it would invalidate the following lemma.

Lemma 3.7.1.

$$\begin{array}{l} \forall\ (s_1\ s_2 : State)\ (osi : os\_ident), \\ valid\_state(s_1) \ \rightarrow \ valid\_state(s_2) \ \rightarrow \ s_1 \equiv_{osi} s_2 \ \rightarrow \ s_1 \equiv_{osi}^{cache} s_2 \end{array} \tag{3.37}$$

The notion of TLB equivalence is defined in a similar way. We say that two valid states s_1 and s_2 are TLB equivalent for *osi*, written $s_1 \equiv_{osi}^{tlb} s_2$, iff either *osi* is not active in both states, or it is active and for all accessible virtual addresses *va* that are in the domain of the TLB of both states, if the machine address $s_1.tlb[va]$ holds a page with RW memory content, then if *va* appears in $s_2.tlb$, it holds the same page, i.e. for all machine addresses ma_1 and ma_2, and page *pg*:

$$\begin{array}{l} s_1.tlb[va] = ma_1 \ \rightarrow \ s_1.memory[ma_1] = pg \ \rightarrow \\ is\_RW(pg.page\_content) \ \rightarrow \\ s_2.tlb[va] = ma_2 \ \rightarrow \ s_2.memory[ma_2] = pg \end{array} \tag{3.38}$$

and conversely. We have:

Lemma 3.7.2.

$$\begin{array}{l} \forall\ (s_1\ s_2 : State)\ (osi : os\_ident), \\ valid\_state(s_1) \ \rightarrow \ valid\_state(s_2) \ \rightarrow \ s_1 \equiv_{osi} s_2 \ \rightarrow \ s_1 \equiv_{osi}^{tlb} s_2 \end{array} \tag{3.39}$$

We write $s_1 \equiv_{osi}^{cache,tlb} s_2$ as a shorthand for $s_1 \equiv_{osi} s_2 \ \wedge \ s_1 \equiv_{osi}^{cache} s_2 \ \wedge \ s_1 \equiv_{osi}^{tlb} s_2$. We can now generalize the unwinding lemmas of Section 3.3. The first lemma states that equivalence is preserved by the execution of all actions that do not generate errors.

Lemma 3.7.3.

$$\begin{array}{l} \forall\ (s_1\ s_1'\ s_2\ s_2' : State)\ (a : Action)\ (osi : os\_ident), \\ s_1 \equiv_{osi} s_2 \ \rightarrow \ os\_action(s_1, a, osi) \rightarrow \ os\_action(s_2, a, osi) \rightarrow \\ s_1 \xrightarrow{a/ok} s_1' \ \rightarrow \ s_2 \xrightarrow{a/ok} s_2' \ \rightarrow \ s_1' \equiv_{osi}^{cache,tlb} s_2' \end{array} \tag{3.40}$$

where $os\_action(s, a, osi)$ denotes that action *a* is an action successfully executed by the OS *osi* in the state *s*; in particular, its execution does not cause an error. Note that an execution that fails does not generate a change in the system state.

The second lemma states that execution does not alter the state of non-active OSs, or active OS if it performs an execution that fails.

Lemma 3.7.4.

$$\begin{array}{l} \forall\ (s\ s' : State)\ (a : Action)\ (r : Response)\ (osi : os\_ident), \\ \neg\ os\_action(s, a, osi) \ \rightarrow \ s \xrightarrow{a/r} s' \ \rightarrow \ s \equiv_{osi}^{cache,tlb} s' \end{array} \tag{3.41}$$

3.7.2 OS isolation in execution traces

The extension to traces of the relation one-step execution with error management is defined as follows: an execution trace is defined as a stream (an infinite list) of states that are related by the transition relation $\xrightarrow{a/r}$, i.e. an object of the form

$$s_0 \xrightarrow{a_0/r_0} s_1 \xrightarrow{a_1/r_1} s_2 \xrightarrow{a_2/r_2} s_3 \ldots \tag{3.42}$$

In the sequel, we use $s \xrightarrow{a/r} t$ to denote the trace obtained by prepending the valid execution step $s \xrightarrow{a/r} t[0]$ to a trace t. We let $Trace$ define the type of these traces. Isolation properties are eventually expressed on execution traces, rather than execution steps.

Non-influencing execution (errors)

Using the unwinding lemmas previously presented, one can establish a non-influence result in the style of [34].

We define for each operating system *osi* a predicate *same_os_actions* stating that two traces have the same set of actions w.r.t. *osi*; so that two traces are related iff they perform the same valid *osi*-actions. Then we define two traces t_1 and t_2 to be *osi-equivalent*, written $t_1 \approx_{osi,cache,tlb} t_2$, co-inductively by the following rules:

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad \neg\ os\_action(s, a, osi)}{(s \xrightarrow{a/r} t_1) \approx_{osi,cache,tlb} t_2}$$

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad \neg\ os\_action(s, a, osi)}{t_1 \approx_{osi,cache,tlb} (s \xrightarrow{a/r} t_2)} \tag{3.43}$$

$$\frac{t_1 \approx_{osi,cache,tlb} t_2 \quad os\_action(s_1, a, osi) \quad os\_action(s_2, a, osi) \quad s_1 \equiv_{osi}^{cache,tlb} s_2}{(s_1 \xrightarrow{a/ok} t_1) \approx_{osi,cache,tlb} (s_2 \xrightarrow{a/ok} t_2)}$$

Theorem 3.7.5.

$$\begin{array}{l} \forall\ (t_1\ t_2 : Trace)\ (osi : os\_ident), \\ same\_os\_actions(osi, t_1, t_2) \rightarrow (t_1[0] \equiv_{osi} t_2[0]) \rightarrow t_1 \approx_{osi,cache,tlb} t_2 \end{array} \tag{3.44}$$

OS isolation formally establishes that two traces are *osi*-equivalent if they have the same set of *osi*-actions and if their initial states are *osi*-equivalent. The proof of OS isolation is based on co-induction principles and on the previous unwinding lemmas. Note that the definition of *osi*-equivalent traces conveniently generalizes the notion used in Section 3.3, by allowing related traces to differ in the number of actions executed by other OSs, and extends considering executions with error management. In particular, Theorem 3.7.5 states that the OS isolation property introduced in Section 3.3 is also valid in the context of executions that include error handling, considering that an *osi*-action is an action successfully executed by the OS *osi*.

Though it is left as future work, it is interesting to comment on the validity of isolation properties under other policies. On the one hand, the replacement policy for the cache and the TLB is left abstract in our model, so any reasonable algorithm will preserve these properties (as embodied e.g. in the definition of *cache_add* in Section 3.5.3). On

the other hand, we have fixed a *write-through* policy for the main memory: this policy entails that updates to memory pages are done simultaneously to the cache and main memory, and we have used throughout the development the invariant property that cache data is included in the memory. This inclusion property will not hold if we were to use a *write-back* policy, in which written entries are marked dirty and updates to main memory are done when a page is removed from the cache. We believe that it remains possible to prove strong isolation properties under the *write-back* policy, since page values, even if different in memory, will be equal if we consider the cache and memory together.

Finally, the flushing policy is assumed to be a total flush on switch and local switch execution. An alternative would be to tag cache (and TLB) entries with the virtual spaces allowed to access the entry. This will not have as much impact on the current model as the write policy, though changes will need to be done to the cache definition to include the tags. Isolation properties would still hold, given correct semantics of access control of cache entries.

3.7.3 Transparency

Functional correctness of a hypervisor is conveniently formalized as a transparency property, stating that, under some specific conditions, a guest operating system cannot distinguish whether it runs on the virtualization platform or operates directly on hardware. In this section, we prove a variant of this property suitable for paravirtualization: namely, a guest OS is unable to distinguish between executing together with other OSs and executing alone on the platform. More precisely, the proved properties establish that given a guest OS *osi*, the (concurrent) execution of other guest OSs does not interfere with the execution of *osi*.

Intuitively, our formulation of transparency states that for any operating system *osi*, any execution trace is *osi*-equivalent to its *osi*-erased trace, in which all state components not related to *osi* have been removed, and all actions not performed by *osi*, or by the hypervisor on behalf of *osi*, have been turned into `silent` actions. This approach is similar, but somewhat stronger than OS isolation, as it requires that after erasing all non-*osi* data, the execution is still valid; i.e. that state validity and action semantics are maintained.

More precisely, the erasure $s\backslash_{osi}$ of a state s is obtained by removing all state components that do not belong to *osi*, and it satisfies the following properties: i) *osi* is the only OS in the erased environment; ii) if *osi* was not active on s, the activity of the erased state is *waiting*, otherwise it is not changed; iii) hypervisor mappings on the erased state should only be defined for *osi* and be the same as in s; iv) all memory pages whose owner is not *osi* (including hypervisor owned ones) are freed; v) *osi* RW pages remain unchanged; vi) non accessible virtual address (i.e. owned by the hypervisor) are removed from all *osi* PT pages; vii) if *osi* is active in s, all non accessible virtual addresses are removed from the cache and the TLB; viii) if *osi* is not active in s, the cache and the TLB get flushed.

Note that if *osi* is not active, no cache (or TLB) entry will belong to *osi*, so we erase all of the entries, flushing the cache. If *osi* is active, cache entries will be owned either by *osi* or by the hypervisor; in the second case, we erase them by removing all entries corresponding to non accessible virtual addresses. The result of these transformations is that only data from *osi* stays in the erased cache in all cases.

$$Pre\ s\ \texttt{silent} \stackrel{\text{def}}{=} True$$
$$Post\ s\ \texttt{silent}\ s' \stackrel{\text{def}}{=}$$
$$s' = s.[cache := \star, tlb := \star] \wedge\ valid\_cache(s') \wedge\ valid\_tlb(s')$$

Figure 3.10: Formal specification of the action `silent`

The erasure of a valid state is also valid:

Lemma 3.7.6.

$$\forall\ (s : State),\ valid\_state(s)\ \rightarrow\ valid\_state(s\backslash_{osi}) \tag{3.45}$$

Moreover, the erasure of a state is equivalent to the original state. Since *osi* is always the active OS in $s\backslash_{osi}$, we must consider a weaker notion, denoted $\stackrel{w}{\equiv}_{osi}$, that is identical to $\equiv_{osi}$ except that it does not require that the two states have the same active OS.

Lemma 3.7.7.

$$\forall\ (s : State),\ valid\_state(s)\ \rightarrow\ s\ \stackrel{w}{\equiv}_{osi}\ s\backslash_{osi} \tag{3.46}$$

Next, erasure must be extended to actions. We silence all actions that are executed by OSs different from *osi*, and by the hypervisor for itself or for one of those other OSs. All other actions (those executed directly by *osi* or by the hypervisor on behalf of *osi*) remain unchanged. We use the $a\backslash_{osi}$ notation for the erasure of an action. Erasure preserves valid step executions.

Lemma 3.7.8.

$$\forall(s\ s' : State)(a : Action),\ s \xhookrightarrow{a} s'\ \rightarrow\ s\backslash_{osi} \xhookrightarrow{a\backslash_{osi}} s'\backslash_{osi} \tag{3.47}$$

The proof of the lemma hinges upon the "loose" semantics `silent` action. Informally, the semantics allows for the cache and the TLB to be different in the pre and post states, only requiring that both are still valid, according to the explanation provided in Section 3.5.2. The formal axiomatic semantics is presented in Figure 3.10; for the sake of readbility, we use non-deterministic field update $r \cdot [f := \star]$ in the rule—the Coq formalization introduces existential quantification instead.

As the erasure of the cache removes every hypervisor entry, a full cache in s will not necessarily be full in $s\backslash_{osi}$. Moreover, a silenced action that adds an entry to the cache in the original execution (a `read_hyper` action, for instance) will remove another entry to make space. In the erased execution, the initial state will not be full, but after the effects of `silent`, the resulting state will have less entries in the cache. This is not an issue, however, because the valid state invariance and Lemma 3.7.6 imply that caches in both states will be consistent: all memory accesses by *osi* will obtain the same data either from the cache or from main memory.

Lemmas 3.7.6 and 3.7.8 entail that the erasure of a trace is another trace. The desired transparency property can then be expressed as: given an *osi* and any valid trace of the system, the erasure of the trace is another trace *osi*-equivalent to the original one. Formally, weak equivalence $\stackrel{w}{\approx}_{osi}$ on traces is defined as a straightforward generalization of weak equivalence $\stackrel{w}{\equiv}_{osi}$ on states.

Theorem 3.7.9.

$$\forall\ (osi : os\_ident)\ (t : Trace),\ t\ \overset{w}{\approx}_{osi}\ t\backslash_{osi} \tag{3.48}$$

3.8 Related work

Thanks to recent advances in verification technology, it is now becoming feasible to verify formally realistic specifications and implementations of operating systems. A recent account of existing efforts can be found in the surveys [105, 106].

The Microsoft Hyper-V verification project focuses on proving the functional correctness of the deployed implementation of the Hyper-V hypervisor [14, 15] or of a simplified, baby, implementation [107]. Using VCC, an automated verifier for annotated C code, these works aim to prove that the hypervisor correctly simulates the execution of the guest operating systems, in the sense that the latter cannot observe any difference from executing on their own on a standard platform. At a more specific level, these works provide a detailed account of many components that are not considered in our work, including page tables [108] and devices [109].

The L4.verified project [16] focuses on proving that the functional correctness of an implementation of seL4, a microkernel whose main application is as a hypervisor running paravirtualized Linux. The implementation consists of approximately 9kLOC of C and 600 lines of assembler, and has been shown to be a valid refinement of a very detailed abstract model that considers for example page tables and I/O devices. Subsequent machine-checked developments prove that seL4 enforces integrity, authority confinement [32] and intransitive non-interference [110, 33]. The formalization does not model cache.

Recently, the Verve project [111] has initiated the development of a new operating system whose type safety and memory safety has been verified using a combination of type systems and Hoare logic. Outside these projects, several projects have implemented small hypervisors, to reduce the Trusted Computing Base, or with formal verification in mind [112], but we are not aware of any completed proof of functional correctness or security.

Our work is also related to formal verification of isolation properties for separation kernels. Earlier works on separation kernels [113, 114, 115] formalize a simpler model where memory is partitioned a priori. In contrast, our model allows the partition to evolve and comprises three types of addressing modes and is close to those of virtualization platforms, where memory requested by the OSs is dynamicaly allocated from a common memory pool. Dealing with this kind of memory management adds significant complications in isolation proofs.

Our work is also inspired by earlier efforts to prove isolation for smartcard platforms. Andronick, Chetali and Ly [116] use the Coq proof assistant to establish that the JavaCard firewall mechanism ensures isolation properties between contexts—sets of applications that trust each other. Oheimb and co-workers [34, 99] independently verify isolation properties for Infineon SLE 88 using the Isabelle proof assistant. In particular, their work formalizes a notion of non-influence that is closely related to our isolation properties.

Moving away from OS verification, many works have addressed the problem of relating inductively defined relations and executable functions, in particular in the context of

programming language semantics. For instance, Tollitte *et al* [117] show how to extract a functional implementation from an inductive specification in the Coq proof assistant. Similar approaches exist for Isabelle, see e.g. [118]. Earlier, alternative approaches such as [119, 120] aim to provide reasoning principles for executable specifications.

3.9 Summary

Hypervisors allow multiple guest operating systems to run on shared hardware, and offer a compelling means of improving the security and the flexibility of software systems. In this chapter we have formalized in the Coq proof assistant an idealized model of a hypervisor, and we have established (formally) that the hypervisor ensures strong isolation properties between the different operating systems, and guarantees that requests from guest operating systems are eventually attended.

In contrast to most prominent projects on operating systems verification, where such guarantees are proved directly on concrete implementations of hypervisors, the machine-checked model of virtualization presented in this work abstracts away most implementations issues and specifies the effects of hypervisor actions axiomatically, in terms of preconditions and postconditions. Unfortunately, seemingly innocuous implementation issues are often relevant for security. Incorporating the treatment of errors into the model has been therefore an important step towards strengthening the isolation theorems proved initially. In this chapter, we also have extended our earlier model with errors, for a memory system with cache and TLB, and we have proved that isolation theorems still apply. In addition, we have provided an executable specification of the hypervisor, and we have proved that it correctly implements the axiomatic model. The executable specification constitutes a first step towards a more realistic implementation of a hypervisor, and provides a useful tool for validating the axiomatic semantics developed. In [104] we derive two certified hypervisor implementations, using the extraction mechanism of Coq, in functional languages Haskell and OCaml.

Finally, we have showed that virtualized platforms are transparent, i.e. a guest operating system cannot distinguish whether it executes alone or together with other guest operating systems on the platform.

Chapter 4

A formal specification of the DNSSEC model

4.1 Introduction

The Domain Name System (DNS) [35, 36] constitutes a distributed database that provides support to a wide variety of network applications. The database is indexed by *domain names*. A domain name represents a path in a hierarchical tree structure, which in turn constitutes a *domain name space*. Each node of this tree is assigned a label, thus, a domain name is built as a sequence of labels separated by a dot, from a particular node up to the root of the tree.

A distinguishing feature of the design of DNS is that the administration of the system can be distributed among several (authoritative) name servers. A *zone* is a contiguous part of the domain name space that is managed by a set of authoritative name servers. Then, distribution is achieved by delegating part of a zone administration to a set of delegated sub-zones. DNS is a widely used scalable system, but it was not conceived with security concerns in mind, as it was designed to be a public database with no intentions to restrict access to information. Nowadays, a large amount of distributed applications make use of domain names. Confidence on the working of those aplications depends critically on the use of trusted data: fake information inside the system has been shown to lead to unexpected and potentially dangerous problems.

Already in the early 90's serious security flaws were discovered by Bellovin and eventually reported in [37]. Different types of security issues concerning the working of DNS have been discussed in the literature [38, 37, 39, 40, 41, 42]. Identified vulnerabilities of DNS make it possible to launch different kinds of attacks, such as *cache poisoning* [45, 46].

DNSSEC [47, 48, 49] is a suite of Internet Engineering Task Force (IETF) specifications for securing information provided by DNS. More specifically, this suite specifies a set of extensions to DNS which are oriented to provide mechanisms that support authentication and integrity of DNS data but not its availability or confidentiality. In particular, the security extensions were designed to protect resolvers from forged DNS data, such as the one generated by DNS cache poisoning, by digitally signing DNS data using public-key cryptography. The keys used to sign the information are authenticated via a chain of trust, starting with a set of verified public keys that belong to the DNS root zone, which is the trusted third party.

4.1.1 A primer on the vulnerabilities of DNS

In this section we provide some background on DNS, its vulnerabilities and the security extensions specified in DNSSEC.

The administration of DNS can be distributed among several (authoritative) name servers. DNS defines two types of server:

- *Nameservers*: which are authoritative servers that manage data of a contiguous part of the domain name space and where the master files reside. For redundancy sake, primary and secondary nameservers are provided for each zone.
- *Resolvers*: which are standard programs used to interact with the nameservers to extract information in response to client requests.

The process of obtaining information from DNS is called *name resolution*. Each server is initialized with the contact information of some authoritative servers of the root zone. Moreover, the root servers know how to contact the authoritative name servers of second level (e.g. the domains com, net, edu). Second level servers know the information of third level servers, and so on. By these means, the requestor can proceed refining the sought response by walking the tree structure, contacting different servers and getting "closer" to the answer after each referral.

Servers store the results of previous queries in their caches in order to speed up the resolution process, but as the mapping of names evolves, cached data has a limited time of validity. Authoritative servers attach, for instance, a Time To Live atribute (TTL) to this stored data, indicating when it should be removed from the cache. For more details concerning the working of DNS we refer to [121, 35, 36].

In an early work, Cheung and Levitt [122] discuss security issues of DNS and provide formal basis to model and prove security mechanisms that can be added to the system to prevent two specific problems:

- *Failure to authenticate DNS responses*: The message authentication mechanism used by DNS is weak. DNS checks if a received response matches a previously asked query only by checking if the Id attached in the query's header matches with the Id of the pointed response. In this way, if an attacker can predict the used query Id and his answer reaches before the real one does (as if a name server receives multiple responses for its query, it uses the first one), it will be able to send a forged response.
- *Cache poisoning attacks*: An attacker can take advantage of the lack of authentication mechanisms to intentionally formulating misleading information, injecting bogus information into some server DNS cache. Having cached this information, the cheated DNS server is likely to get a Denial of Service (DoS), if the attacker sends a negative response that could actually be resolved; or in the worst case, the attacker can masquerade as a trusted entity, and then be able to intercept, analyze and intentionally corrupt the communication.

Cache poisoning attacks exploits a flaw in DNS, namely, the weak mechanisms used to ensure the authentication of data origin. As one of the goals of the DNS security extensions was to solve these vulnerabilities, we have put special focus in developing a formal specification that allows us to reason on the effectiveness of those extensions regarding impersonation and cache poisoning attacks.

The extensions introduced by DNSSEC to improve the security of DNS require the combined application of mechanisms that make it possible to i) sign data, using public key cryptography, within zones ii) generate a chain of trust along the DNSSEC tree iii) perform key exchange within parent-child zones, and regular key rollover routines. The security extensions provide origin authentication and integrity assurance services for DNS data, including mechanisms for public key distribution and authenticated denial of existence of DNS data. However, respecting the principle assumed in the design of DNS that all data in the system must be visible, DNSSEC is not designed to provide confidentiality [47].

The specification of the security extensions does not prevent the interaction of secure name servers and resolvers with non-secure ones. However, any communication that involves an unsecure server results in the loss of all DNSSEC security related capabilities. For this reason, in our model it is assumed that every server is security aware.

4.1.2 Contents of the chapter

We present here a minimalistic specification of a DNSSEC model which provides the grounds needed to formally state and verify security properties concerning the chain of trust of the DNSSEC tree. The specification puts forward an abstract formulation of the behavior of the protocol and the corresponding security-related events, where security goals, such as the prevention of cache poisoning attacks, can be given a formal treatment. This chapter builds upon the previously published paper [123].

The rest of the chapter is organized as follows. In Section 4.2 we develop the elements that are most relevant for our formal model. Section 4.3 presents some of the verified properties and outlines the corresponding proofs. Finally, Section 4.4 concludes with a summary of the chapter.

4.1.3 Formalization

The full development is available for download at

`http://www.fing.edu.uy/inco/grupos/gsi/sources/dnssec`

4.2 Formalization of the DNSSEC model

In this section, we present an abstract model of DNSSEC. First, we introduce auxiliary definitions, to proceed then to define the set of states and a notion of valid state. Finally, we define the semantics of security-related events as state transformers and provide a formal definition of their execution.

4.2.1 Formalizing states

The state of a DNSSEC system consists of a collection of components that we now proceed to describe. Servers distributed globally are represented as objects of an abstract type **Process**.

Secure resource records

A Resource Record (RR) is the basic unit of information used in DNS. A structure of this kind is defined by six fields: i) the field $NAME$ defines the domain name that applies to the given RR, ii) the field $TYPE$ indicates the type of resource record. Figure 4.1 depicts the most common ones, iii) the field $CLASS$ is used to identify the protocol group to which the record belongs. In the sequel we shall only make use of the class IN, which is the one of interest to people using TCP/IP software, since it stands for *Internet*, iv) TTL stands for Time To Live; it is primarily used by resolvers, and specifies how long a resource record should be cached before discarding it, v) the field $RDLENGTH$ is an unsigned integer that specifies the length of the RDATA field, vi) the field $RDATA$ contains the data of the resource record. The data field is defined differently for each type and class of data.

Type	*Description*
A	Internet Address
CNAME	Canonical Name (nickname pointer)
HINFO	Host Information
MX	Mail Exchanger
NS	Name Server
PTR	Pointer
SOA	Start Of Authority

Figure 4.1: Types of Resource Records

Formally, a resource record is defined as an object of the following record type:

$$\begin{aligned} \mathbf{RR} \stackrel{\text{def}}{=} [\![\quad & Rdname : DName, \\ & Rtype : RRType, \\ & RClass : RRClass, \\ & Rttl : TTL, \\ & RDL : RDLength, \\ & Rrdata : RData \ ]\!] \end{aligned} \tag{4.1}$$

A set of resource records that share the same domain name, type and class is formalized as an object of the following type:

$$\mathbf{RRset} \stackrel{\text{def}}{=} set\ RR \tag{4.2}$$

To implement the proposed security extensions, DNSSEC introduces additional security-related resource records: i) for origin authentication DNSSEC provides a hierarchical public key infrastructure (PKI), which allows resolvers to obtain the DNSSEC key of a zone and use it for authenticating signed data belonging to that zone. To support this PKI, three resource records were introduced: RRSIG, DNSKEY, and DS, ii) for assuring integrity of data each zone signs all the RRsets over it is authoritative. In every transmission, RRSIGs are transmitted along with the replied RRsets, and by these means, when a transmission is received, data integrity can be verified, iii) for authenticated denial of existence, Next Secure (NSEC) resource records are provided. They list all of the existent RRs belonging to an owner name within an authoritative

zone, making it possible to verify the non-existence of a RR, by comparing against the RR list of its owner name. Figure 4.2 describes the security oriented new resource records.

RR	Description
RRSIG	Signature over RRset made using private key
DNSKEY	Public key, needed for verifying a RRSIG
DS	Delegation Signer, pointer for building chains of trust. The Parent DNSKEY signs the Parent DS, Parent DS signs the Child DNSKEY, and so forth, providing a mechanism to verify origin integrity from a domain name up to the root servers
NSEC	Used to provide an authenticated non-existence of data. Indicates which name is the next one in the zone and which type codes are available for the current name

Figure 4.2: Resource records introduced by DNSSEC

The type of a resource record, which includes the specific types defined by the DNSSEC specifications, is defined by the following enumerated type:

$$\begin{aligned} \mathbf{RRType} \stackrel{\text{def}}{=} \; & A \\ & \mid PTR \\ & \mid NS \\ & \mid CNAME \\ & \mid MX \\ & \mid SOA \\ & \mid HINFO \\ & \mid \mathbf{RRSIG} \\ & \mid \mathbf{DNSKEY} \\ & \mid \mathbf{DS} \\ & \mid \mathbf{NSEC} \end{aligned} \tag{4.3}$$

We formalize the notion of secure (set of) RR by the following two types:

$$\mathbf{SecRR} \stackrel{\text{def}}{=} [\![\; RR' : RR, \; Rsign : RR \;]\!] \tag{4.4}$$

$$\mathbf{SecRRset} \stackrel{\text{def}}{=} [\![\; RRset' : RRset, \; RRsign : RR \;]\!] \tag{4.5}$$

where $Rsign$ and $RRsign$ contain, respectively, the signature corresponding to RR' and to $RRset'$.

The distributed database

DNS manages a distributes database, which is indexed by a tuple (dname, type, class) of type Idx:

$$\mathbf{Idx} \stackrel{\text{def}}{=} [\![\; Idname : DName, \; Itype : RRType, \; IClass : RRClass \;]\!] \tag{4.6}$$

The range of this database is a set of secure RRs. A DNS database is thus formalzed as an object of the function type:

$$\mathbf{DbMap} \stackrel{\text{def}}{=} Idx \rightarrow SecRRset \tag{4.7}$$

DNS message

A DNS(SEC) message consists of a header and four additional sections [43], as illustrated in Figure 4.3. The *HEADER* contains an identification (Id) field, which is used to match

Figure 4.3: DNS Message Format

an answer to its corresponding query. The *QUESTION* section consists of a target domain name (*QNAME*), a type (*TYPE*), and a class (*QCLASS*). A query to find out, for instance, the IP address of the host *h1.fing.edu.uy* will have *QNAME=h1.fing.edu.uy*, *QTYPE=A* and *QCLASS=IN*. The *ANSWER* section contains RRs which directly answer the query. The *AUTHORITY* section carries RRs which describe other authoritative server (e.g. may contain RRs to refer the querier to other name servers during the resolution process). The *ADDITIONAL* section carries RRs that may be helpful for using RRs information of other sections (e.g. may contain **A** RRs to provide the IP address for the RRs listed in the authority section).

We define the $DNSMessage$ type as follows:

$$\begin{aligned} \mathbf{DNSMessage} \stackrel{\text{def}}{=} [\![\quad & Hdr : Header, \\ & Q : Idx, \\ & Ans : SecRRset, \\ & Auth : SecRRset, \\ & Additional : SecRRset \ ]\!] \end{aligned} \tag{4.8}$$

Pending submissions

Pending submissions, that is to say, requests or answers, possibly triggered by a received answer, which are waiting to be delivered, are defined as objects of the following type:

$$\begin{aligned} \mathbf{SendQR} \stackrel{\text{def}}{=} [\![\quad & SendQ : set\ InfoMsg, \\ & SendR : set\ InfoMsg \ ]\!] \end{aligned} \tag{4.9}$$

An object of the type $InfoMsg$ represents the required information needed to be sent.

$$\begin{aligned} \mathbf{InfoMsg} \stackrel{\text{def}}{=} [\![\quad & MFrom : Process, \\ & MTo : Process, \\ & MMsg : DNSMessage \ ]\!] \end{aligned} \tag{4.10}$$

System keys

DNSSEC security is based in the use of public key cryptography. Each DNS server owns two keys, namely, a Zone Signing Key (ZSK) and a Key Signing Key (KSK). A ZSK key is used to sign every RRset within a zone, generating the RRSIG records. A KSK key is used especifically to sign the DNSKEY RRset and generate its corresponding RRSIG. To access the system's DNSKEYS, the following record have been defined:

$$\mathbf{keySet} \stackrel{\text{def}}{=} [\![\quad key : RR, \\ \quad ksign : RR \,]\!] \tag{4.11}$$

where *key* is a DNSKEY and *ksign* its corresponding RRSIG signature.

Each server has its corresponding pair of keys:

$$\mathbf{keys} \stackrel{\text{def}}{=} [\![\quad zsk : keySet, \\ \quad ksk : keySet \,]\!] \tag{4.12}$$

Delegations

A delegation describes a father-child relationship between DNSSEC servers. As shown in Figure 4.2, it helps building the DNSSEC chain of trust. To model delegation in our specification we have defined the following function:

$$\mathbf{DSpDB} \stackrel{\text{def}}{=} Process \to DSp \tag{4.13}$$

where:

$$\mathbf{DSp} \stackrel{\text{def}}{=} [\![\quad Tsrv : Process, \\ \quad rrDS : SecRR \,]\!] \tag{4.14}$$

In words, we have defined a function that maps Parent servers with its corresponding Childs to whom they trust, along with the digest RR to prove it. This is an essential part of the model, as it will allow us to reason over the effectiveness of the chain of trust.

System state

To reason about the DNSSEC security system most details of the state may be abstracted. States are modeled by a record type with nine components:

$$\begin{aligned} \mathbf{State} \stackrel{\text{def}}{=} [\![\quad & Servers : set\ Process, \\ & TrustedServers : set\ Process, \\ & ServerKeys : Process \to keys, \\ & Delegations : Process \to DSpDB, \\ & Parents : Process \to DSp, \\ & viewAuth : Process \to DbMap, \\ & viewCache : Process \to DbMap, \\ & PendingQueries : Process \to set\ Header, \\ & SendBuffer : Process \to SendQR \,]\!] \end{aligned} \tag{4.15}$$

The component *Servers* indicates the involved DNS servers, whereas that the components $TrustedServers$ and $ServerKeys$ represent set of publicly known trusted servers

and the set of keys for every server, respectively. The delegations issued by each server and the fathers of a server are indicated by the components $Delegations$ and $Parents$. The components $viewAuth$ and $viewCache$ are used to represent the authoritative view and the cache view of every server, $PendingQueries$ is a function that maps a server with the expected answers to the already performed queries, and $SendBuffer$ is a buffer with the corresponding pending submissions for each server.

Valid state

We define a notion of valid state that captures essential security properties of the system, and more particularly of the DNSSEC specification provided by the "DNSSEC protocol document set", which refers to the three documents that form the core of the DNS security extensions: *"DNS Security Introduction and Requirements"* [47], *"Resource Records for DNS Security Extensions"* [48], and *"Protocol Modifications for the DNS Security Extensions"* [49].

We say that the predicate $Valid$ holds on state s if s satisfies the following properties:

1. *"Every RRset within the view of a server must be signed by its corresponding RRSIG signature"*, which can be stated formally as follows,

$$\begin{aligned} \textbf{RR\_integrity}\ s \stackrel{\text{def}}{=}\ & \forall\ srv{:}Process,\ srv \in s.Servers \rightarrow \\ & signedView\ s\ srv\ (s.viewAuth\ srv) \wedge \\ & signedView\ s\ srv\ (s.viewCache\ srv) \end{aligned} \tag{4.16}$$

   where $signedView$ verifies that every RRset of a sever's zone is correctly signed by its corresponding RRSIG.

2. *"Every server's Zone keys must be signed by its corresponding Key Signing Key (KSK)"*,

$$\begin{aligned} \textbf{ZSK\_integrity}\ s \stackrel{\text{def}}{=}\ & \forall\ (srv{:}Process),\ srv \in s.Servers \rightarrow \\ & verifySign\ (s.ServerKeys\ srv).ksk.key \\ & \{(s.ServerKeys\ srv).ksk.key\} \cup \\ & \quad \{(s.ServerKeys\ srv).zsk.key\} \\ & (s.ServerKeys\ srv).zsk.ksign \end{aligned} \tag{4.17}$$

   where $verifySign$ checks that the RRSIG of a RRset has been effectively generated from a given key.

3. *"Every server must be publicly known as trusted, or be verified by the corresponding digest in its father's zone"*,

$$\begin{aligned} \textbf{KSK\_integrity}\ s \stackrel{\text{def}}{=}\ & \forall\ (srv{:}Process),\ srvH \in s.TrustedServers \vee \\ & checkDigest\ s\ srvH\ (s.Parents\ srvH).rrDS.RR' \end{aligned} \tag{4.18}$$

   where $checkDigest$ validates if a DS record in the Parent zone really contains a digest of its Child keys.

We thus formally define the $Valid$ predicate over $State$ as the conjunction of the previous validity conditions:

$$\begin{aligned} \textbf{Valid}\ s \stackrel{\text{def}}{=}\ & RR\_integrity\ s \wedge \\ & ZSK\_integrity\ s \wedge \\ & KSK\_integrity\ s \end{aligned} \tag{4.19}$$

4.2.2 Events

The working of the system is modelled in terms of the execution of events. An event is understood as an action that transforms the state of the system. Next we present the syntax, and specify the semantics and execution of events.

Syntax

The syntax and signature for every event relevant to our abstract model of DNSSEC is formalized by the non-recursive inductive set $Event$:

$$\begin{aligned} \textbf{\textit{Event}} \stackrel{\text{def}}{=}\ & \\ & \mid\ Add\_Server : Process \to keys \to DSpDB \to DSp \to DbMap \to \\ & \quad DbMap \to Event \\ & \mid\ Delete\_Server : Process \to Event \\ & \mid\ Add\_RRset : Process \to Idx \to RRset \to Event \\ & \mid\ Delete\_RRset : Process \to Idx \to Event \\ & \mid\ Server\_ZSK\_rollover : Process \to RR \to Event \\ & \mid\ Server\_KSK\_rollover : Process \to RR \to RR \to Event \\ & \mid\ Add\_TrustedServer : Process \to Event \\ & \mid\ Del\_TrustedServer : Process \to Event \\ & \mid\ Send\_Query : Process \to Process \to DNSMessage \to Event \\ & \mid\ Receive\_Query : Process \to Process \to DNSMessage \to Event \\ & \mid\ Receive\_Response : Process \to Process \to DNSMessage \to Event \\ & \mid\ Send\_PendingQ : Process \to Event \\ & \mid\ Send\_PendingR : Process \to Event \\ & \mid\ RR\_TimeOut : Process \to Idx \to RR \to Event \end{aligned} \tag{4.20}$$

A brief description of each event is shown in Table 4.1. In the next section we present the formal semantics for a small subset of these events.

Semantics

The behavior of the events is specified by their pre- and postconditions, which are given by the predicates Pre and $Post$ respectively,

$$\begin{aligned} & Pre : State \to Event \to Prop \\ & Post : State \to State \to Event \to Prop \end{aligned} \tag{4.21}$$

Preconditions are defined in terms of the system state while postconditions are defined in terms of the before and after states. Due to space constraints, we only present here the formal specification of three distinguished events, namely, $Server\_ZSK\_rollover$, $Receive\_Response$ and $RR\_TimeOut$. In the specification of the postconditions we

Table 4.1: DNSSEC model events

Name	Description
Add_Server	Creates a new server in the domain name system
Delete_Server	Deletes a server from the domain name system
Add_RRset	Inserts a new RRset in the authoritative view of a given server
Delete_RRset	Deletes a RRset from the authoritative view of a given server
Server_ZSK_rollover	Performs a rollover of the zone key of a given server
Server_KSK_rollover	Performs a rollover of the KSK of a given server
Add_TrustedServer	Indicates the reliable publication of a server known as trusted
Del_TrustedServer	Indicates that a server should no longer be consider as trusted
Send_Query	Sends a specific query to a given server
Receive_Response	Receives a response from a given server
Send_PendingQ	Sends pending queries
Send_PendingR	Sends pending responses
RR_TimeOut	Indicates that the TTL of a given RRset has expired

have omitted all fields of the state that remain invariant when executing the corresponding event. The names of the auxiliary functions and predicates used should be self-explanatory.

- ***Server_ZSK_rollover srv rrzsk*** This event performs the rollover of the ZSK key for a specific server *srv*. For this event to take place it is needed that *srv* exists in the system and *rrzsk* should be a ZSK key, that is to say,a key which is identified as a *Zone Signing key* by the information of its RRDATA.

$$\begin{aligned} &\textbf{\textit{Pre s (Server\_ZSK\_rollover srv rrzsk)}} \stackrel{\text{def}}{=} \\ &\quad isServer\ s\ srv \wedge \\ &\quad isZSK\ rrzsk \end{aligned} \tag{4.22}$$

$$\begin{aligned} &\textbf{\textit{Pos s s' (Server\_ZSK\_rollover srv rrzsk)}} \stackrel{\text{def}}{=} \\ &\quad (s'.ServerKeys\ srv).zsk.key = rrzsk \\ &\quad \wedge\ (s'.ServerKeys\ srv).zsk.ksign = \\ &\qquad sign\ (s.ServerKeys\ srv).ksk.key \\ &\qquad\quad (\{(s.ServerKeys\ srv).ksk.key\} \cup \{rrzsk\}) \\ &\quad \wedge\ \forall\ (i{:}Idx),\ (s'.viewAuth\ srv\ i).RRsign = \\ &\qquad sign\ rrzsk\ (s.viewAuth\ srv\ i).RRset' \end{aligned} \tag{4.23}$$

  The postcondition states that when this event executes successfully the key *rrzsk* is stored as the new ZSK, its signature is calculated, using *sign*, and stored as its corresponding RRSIG. In addition to that, all the signatures for the resource records within the authoritative view of the server *srv* are re-calculated.

- ***Receive_Response srv_from srv_to m*** This event models the processing of an answer message *m*. For this operation to succeed, servers *srv_from* and *srv_to* must belong to the set of servers of the system and the received response should be an answer to a previously submitted query delivered by the server *srv_to*.

$$\begin{aligned}&\textbf{\textit{Pre s (Receive\_Response srv\_from srv\_to m)}} \stackrel{\text{def}}{=}\\ &\quad isServer\ s\ srv\_from\\ &\quad \wedge\ isServer\ s\ srv\_to\\ &\quad \wedge\ m.Hdr \in (s.PendingQueries\ srv\_to)\end{aligned} \tag{4.24}$$

$$\begin{aligned}&\textbf{\textit{Pos s s' (Receive\_Response srv\_from srv\_to m)}} \stackrel{\text{def}}{=}\\ &\quad if\ (nodata\_newrefer\ m)\\ &\qquad (s'.SendBuffer\ srv\_to).SendQ = (s.SendBuffer\ srv\_to).SendQ \cup\\ &\qquad\quad \{(makeResp\ srv\_to\ (next\ m.Additional.RRset')\ m)\}\\ &\quad elseif\ (verifySign\ (s.ServerKeys\ srv\_to).zsk.key\ m.Ans.RRset'\\ &\qquad m.Ans.RRsign)\\ &\qquad ((s'.viewCache\ srv\_to)\ m.Q).RRset' = m.Ans.RRset'\\ &\qquad \wedge\ ((s'.viewCache\ srv\_to)\ m.Q).RRsign = m.Ans.RRsign\\ &\qquad \wedge\ s'.PendingQueries\ srv\_to = (s.PendingQueries\ srv\_to) \setminus \{m.Hdr\}\end{aligned} \tag{4.25}$$

The postcondition establishes the conditions required to be satisfied for a given resource record to be accepted and allocated in the cache view of a server. In the case that the received answer does not contain information in its auth section, but it does contain a *closer* server to refer, then the message will be re-sent to that server. On the contrary, if the received message contains data in its auth section, and the received RRsets as well as their corresponding received signatures are verified by the sender zsk key, then the RRsets and their signatures are added to the server's cache view and, as the query has been successfully replied, the message is removed from the *PendingQueries* set. It should be noticed that a correct execution of this event would prevent a cache poisoning like the one discussed in Section 4.1.1. In Section 4.3 we will skecth the proof that the state remains valid after the execution of this event.

- ***RR_TimeOut srv i rr*** Runs on the expiration of the TTL of a resource record rr for a given server srv. The operation precondition will be verified when srv belongs to the set of servers of the system and the TTL of rr has expired due to timeout of either the resource record or its corresponding signature.

$$\begin{aligned}&\textbf{\textit{Pre s (RR\_TimeOut srv i rr)}} \stackrel{\text{def}}{=}\\ &\quad isServer\ s\ srv\ \wedge\ isTimeout\ s\ srv\ i\ rr\\ &\quad \wedge\ (rr\ \in\ (s.viewCache\ srv\ i).RRset' \vee\\ &\qquad rr\ \in (s.viewAuth\ srv\ i).RRset')\end{aligned} \tag{4.26}$$

$$\begin{aligned}&\textbf{\textit{Pos s s' (RR\_TimeOut srv i rr)}} \stackrel{\text{def}}{=}\\ &\quad if\ (rr\ \in (s.viewAuth\ srv\ i).RRset')\\ &\qquad ((s'.viewAuth\ srv)\ i).RRset' = ((s.viewAuth\ srv)\ i).RRset' \setminus \{rr\}\\ &\qquad \wedge\ ((s'.viewAuth\ srv)\ i).RRsign =\\ &\qquad\quad sign\ (s.ServerKeys\ srv).zsk.key\ ((s'.viewAuth\ srv)\ i).RRset'\\ &\quad else\ ((s'.viewCache\ srv)\ i).RRset' = ((s.viewCache\ srv)\ i).RRset'\{rr\}\\ &\qquad \wedge\ ((s'.viewCache\ srv)\ i).RRsign =\\ &\qquad\quad sign\ (s.ServerKeys\ srv).zsk.key\ ((s'.viewCache\ srv)\ i).RRset'\end{aligned} \tag{4.27}$$

The postcondition of *RR_TimeOut* states that the expired resource record and its signature are removed from the correponding authoritative or cache view.

Errors

When an event is executed and a precondition is not valid, the system answers with a corresponding error code. These error codes are defined by the following enumerated type:

$$\begin{aligned} \textbf{\textit{ErrorCode}} \stackrel{\text{def}}{=} \quad & server\_not\_exists \\ & \mid invalid\_zsk \\ & \mid query\_not\_asked \\ & \mid rrset\_not\_exists \\ & \mid rr\_not\_timeout \\ & \mid ... \end{aligned} \tag{4.28}$$

Table 4.2 specifies the error code associated to unverified preconditions of the three events whose semantics was presented in Section 4.2.2. In our model this is formalized as a relation $ErrorMsg$ between valid states of the system and error messages.

Table 4.2: Error messages

Server_ZSK_rollover srv rrzsk	
$\neg\ isServer\ s\ srv$	server_not_exists
$\neg\ isZSK\ rrzsk$	invalid_zsk
Receive_Response srv_from srv_to m	
$\neg\ isServer\ s\ srv\_from \vee\ \neg\ isServer\ s\ srv\_to$	server_not_exists
$\neg\ m.Hdr\ \in\ s.PendingQueries\ srv\_to$	query_not_asked
RR_TimeOut srv i rr	
$\neg\ isServer\ s\ srv$	server_not_exists
$rr\ \notin\ (s.viewCache\ srv\ i).RRset'$ $\vee\ rr\ \notin\ (s.viewAuth\ srv\ i).RRset'$	rrset_not_exists
$\neg isTimeout\ s\ srv\ i\ rr$	rr_not_timeout

One-step execution

Executing an event e over a state s produces a new state s' and a corresponding answer r (denoted $s \xhookrightarrow{e/ans} s'$), where the relation between the former state and the new one is given by the postcondition relationship $Post$.

$$\frac{Pre\ s\ e \qquad Post\ s\ s'\ e}{s \xhookrightarrow{e/ok} s'}\ \textbf{\textit{exec\_pre}} \tag{4.29}$$

$$\frac{\neg\ Pre\ s\ e \qquad \exists\ ec{:}ErrorCode,\ ErrorMsg\ s\ e\ ec \wedge ans = error\ ec}{s \xhookrightarrow{e/ans} s}\ \textbf{\textit{exec\_npre}} \tag{4.30}$$

If the precondition $Pre\ s\ e$ is satisfied, then the resulting state s' and the corresponding answer ans are the ones described by the relation $exec$. However, if $Pre\ s\ e$ is not satisfied, then the state s remains unchanged and the system answer is the error message determined by the relation $ErrorMsg$. Formally, the possible answers of the system are defined by the following type:

$$\begin{array}{rcl} \textbf{\textit{answer}} & \stackrel{\text{def}}{=} & ok : answer \\ & | & error : ErrorCode \rightarrow answer \end{array} \tag{4.31}$$

where ok is the answer resulting from a successful execution of an event.

4.3 Verification of security properties

In this section we discuss two relevant properties of the model that have been formally stated and verified. We first concentrate on the proof that one-step execution preserves the validity of states. We sketch the proof of this property and show that the notion of valid state embodies necessary conditions to prove the objective security properties. Then, we show, formally, that to have this invariance result is not enough to ensure consistency of the chain of trust.

4.3.1 An invariant of one-step execution

A one-step execution invariant is a property that if it holds for the state before the execution of any event it remains valid for the state resulting from that execution. We show that the validity of the model state, as defined in Section 4.2.1, is a one-step invariant of our specification.

Proposition 4.3.1. *For any $s\ s' : State$, $ans : answer$ and $e : Event$, if $Valid\ s$ and $s \xhookrightarrow{e/ans} s'$ hold, then $Valid\ s'$ also holds.*

(4.32)

Proof. The proof of this proposition proceeds by case analysis on the execution $s \xhookrightarrow{e/ans} s'$: i) if the precondition $Pre\ s\ e$ is not satisfied, then the specification of executions establishes that the state s is not modified and that $s = s'$, so s' is trivially valid because s is valid, ii) otherwise, $Post\ s\ s'\ r\ e$ must hold and then we proceed by case analysis on the event e.

We shall here discuss in detail the proof argument for the case the event e is of the form $Receive\_Response\ srv\_from\ srv\_to\ m$. The complete formal proof of this and each of the remaining cases can be found in the accompanying formalization.

The proof starts by analyzing the type of response obtained. In the case the answer has authoritative content we proceed by checking whether the signature of the received RRset is verified. Then we have two possible cases:

1. If the received answer does not contain data, then, as specified in its postconditions, when executing *Receive_Response*, the component $viewCache$ of the state s' will remain invariant, moreover the only component modified in s' will be $SendBuffer$ which is not verified in the validity property, so we can state that s' remain valid.

2. On the contrary, if the answer contains data in its *Auth* section, we must analyze if the received *RRset* is verified by its corresponding signature *RRsig*: i) if the signature is not verified, the received answer will be discarded and no component of the state will be modified, so the property will remain satisfied for s', ii) if the signature is correctly verified and so the *RRset* is added to the *viewCache* view of the server *srv_to*, the resultant state s' will stay valid, as *viewCache* is the only modified component and, by adding verified RRset and RR signature, it remains correctly signed.

□

Observe that the instantiation of Proposition 4.3.1 to the case in which the executed event is *Receive_Response* reflects the (informal) security requirement that no man-in-the-middle can masquerade as a trusted entity in order to provide answers with fake resource records. By proving the correct execution of this event, we are providing (formally verified) evidence that if a classical DNS attacker sends a malicious resource record to a secure server that record shall be discarded as it will not be verified by its corresponding signature, which, in turn, has been created by its father within the chain of trust. This is clearly a security improvement regarding DNS, because the reception mechanism of this system is what makes it a potential victim of cache poisoning attacks.

4.3.2 Compromising the chain of trust

It is important to notice that the conditions specified for a state of our model to be a valid one constitute an almost straightforward interpretation of the (security) recommendations laid down in relevant DNSSEC RFCs like, for instance, [47, 48, 49]. We can show, however, that despite the validity of states is preserved by execution this does not necessarily guarantee that the chain of trust remains valid. In particular, analyzing the conditions required for the rollover of a zone key, which in our model is specified as part of the semantics of the event *Server_ZSK_rollover*, we have detected a small inconsistency concerning the data of the system. Namely, for a resource record to be discarded from a view either authoritative or cache, it is only required that its corresponding TTL is recached. Now, a rollover of a zone key might be needed to be executed, for instance, in the case the server's zone is compromised, even if the TTL of the key has not expired. Consequently, every RRset within the zone must be signed to generate the new RRSIG records. Therefore, every DNS server that contains these, just re-signed, records inside its cache view will become inconsistent. This issue could not be detected during the verification of the invariance of the event *Server_ZSK_rollover* because the specification of the DNSSEC protocol mandates for a resource record to be discarded from the zone file only in if its TTL has expired.

This problem was independently discovered and pointed out by Bau and Mitchell in [124], where they study the security properties of a restricted model of DNSSEC using model checking techniques. We adhere to their recommendation that the resolver logic specified in the corresponding RFC [47] should be strengthened so as to prevent the identified problem.

We now proceed to provide a formal argument that the (incomplete) current specification of the zone key rollover procedure may facilitate the occurrence of inconsistent chains of trust. DNSSEC provides no mechanisms to validate a cached resource record

faced to another signature in its attestation chain. We shall show that if for some reason it is performed a rollover of the zone key for a given server (srv) which has not yet arrived to its TTL a resolver ($srvOldCache$) caching the corresponding resource record (identified by its index a) signed with the old zone key may keep cached that resource record for the entire signature validity period, turning the previous valid chain of trust into an inconsistent one.

We assume that the server $srvOldCache$ has information in its cache view corresponding to the authoritative zone of the server srv, i.e. for an index a:

$$((s.viewAuth\ srv)\ a).RRsign = ((s.viewCache\ srvOldCache)\ a).RRsign \tag{4.33}$$

Now, let us consider a correct execution of the event $Server\_ZSK\_rollover$. Thus, for arbitrary $s' : State$ and $rrzsk : RR$, the following predicates are verified:

$$Valid\ s \tag{4.34}$$
$$Pre\ s\ (Server\_ZSK\_rollover\ srv\ rrzsk) \tag{4.35}$$
$$Post\ s\ s'\ (Server\_ZSK\_rollover\ srv\ rrzsk) \tag{4.36}$$

Therefore, when performing a rollover of the zone key zsk for the server srv, the following inconsistency will take place:

$$((s'.viewAuth\ srv)\ a).RRsign \neq\ ((s'.viewCache\ srvOldCache)\ a).RRsign \tag{4.37}$$

The proof will be performed by reduction to the absurd, assuming that the mentioned inconsistency in fact does not occur, considering as valid:

$$((s'.viewAuth\ srv)\ a).RRsign = ((s'.viewCache\ srvOldCache)\ a).RRsign \tag{4.38}$$

We know from the postcondition of the event $Server\_ZSK\_rollover$ that:

$$(s'.ServerKeys\ srv).zsk.key = rrzsk \tag{4.39}$$
$$\forall\ i : Idx,\ ((s'.viewAuth\ srv)\ i).RRsign = sign\ rrzsk\ (s.viewAuth\ srv\ i).RRset' \tag{4.40}$$
$$s'.viewCache = s.viewCache \tag{4.41}$$

Rewriting (4.41) and then (4.33) in (4.38) we have that:

$$((s'.viewAuth\ srv)\ a).RRsign = ((s.viewAuth\ srv)\ a).RRsign \tag{4.42}$$

Now, considering that if a given record RR is signed with two different zsk keys, different RRSIG records are obtained, we arrive to:

$$((s'.viewAuth\ srv)\ a).RRsign \neq\ ((s.viewAuth\ srv)\ a).RRsign \tag{4.43}$$

The propositions (4.42) and (4.43) reflects an absurd. This allow us to conclude that (4.37) holds.

4.4 Summary

The Domain Name System Security Extensions (DNSSEC) is a suite of specifications that provide origin authentication and integrity assurance services for DNS data. In particular, DNSSEC was designed to protect resolvers from forged DNS data, such as the one generated by DNS cache poisoning. In this chapter we have presented a minimalistic specification of a DNSSEC model which provides the grounds needed to formally state and verify security properties concerning the chain of trust of the DNSSEC tree. The model, which has been formalized and verified using the Coq proof assistant, specifies an abstract formulation of the behavior of the protocol and the corresponding security-related events, where security goals can be given a formal treatment.

Chapter 5

Conclusion and future work

Our work shows that it is feasible to analyze formally models of safety-critical applications, and is part of a trend to build and analyze realistic models of mobile devices, operating systems, hypervisors, and other kinds of complex critical systems. The Coq proof assistant is a useful tool when sophisticated algorithms and specifications are involved and also as a general framework to design special platforms for the verification of critical systems.

We present now the conclusions and describe future work for each of the areas of safety-critical applications addressed in this work.

5.1 Security models for mobile devices

We have provided the first verifiable formalization of the MIDP 2.0 security model, according to [65], and have also constructed the proofs of several important properties that should be satisfied by any implementation that fulfills its specification. Our formalization is detailed enough to study how other mechanisms interact with the security model, for instance, the interference between the security rules that control access to the device resources, and mechanisms such as application installation. We have also proposed a refinement methodology that might be used to obtain a sound executable prototype of the security model.

Moreover, a high-level formalization of the JME-MIDP 2.0 Access Control module also has been developed. This specification assumes that the security policy of the device is static, that there exists at most one active suite in every state of the device, and that all the methods of a suite share the same protection domain. The obtained model, however, can be easily extended so as to consider multiple active suites as well as to specify a finer relation allowing to express that a method is bound to a protection domain, and then that two different methods of the same suite may be bound to different protection domains. With the objective of obtaining a certified executable algorithm of the access controller, the high-level specification has been refined into a executable equivalent one, and an algorithm has been constructed that is proved to satisfy that latter specification. The formal specification and the obtained derived code of the algorithm contribute to the understanding of the working of such an important component of the security model of that platform.

Additionally, it has also been presented an extension of the formalization of MIDP 2.0 security model which considers the changes introduced in version 3.0 of MIDP. In

particular, a new dimension of security is represented: the security at the application level. This extended specification preserves the security properties verified for MIDP 2.0 and enables the research of new security properties for MIDP 3.0. In this way, the formalization is updated keeping its validity as a useful tool to analyze the security model of MIDP at both, the platform and the application level. Some weaknesses introduced by the informal specification of version of MIDP 3.0 are also discussed, in particular those regarding the interplaying of the mechanisms for enforcing security at the application and at the platform level. They reflect potential weaknesses of implementations which satisfy the informal specification [24].

Finally, we have also built a framework that provides a uniform setting to define and formally analyze access control models which incorporate interactive permission requesting/granting mechanisms. In particular, the work presented here has focused on two distinguished permission models: the one defined by version 2.0 (and 3.0) of MIDP and the one defined by Besson et al. in [29]. A characterization of both models in terms of a formal definition of grant policy has also been provided. Another kind of permission policies can also be expressed in the framework. In particular, it can be adapted to introduce a notion of permission revocation, a permission mode not considered in MIDP. A revoke can be modeled in the permission overwriting approach, for instance, by assigning a zero multiplicity to a resource type. In the accumulative approach, revocation might be modeled using negative multiplicities. To introduce revocations, in turn, enables, without further changes to the framework, to model a notion of permission scope. One such scope would be grasped as the session interval delimited by an activation and a revocation of that permission.

The formal development is about 20kLOC of Coq (see Figure 5.1).

MIDP 2.0 formalization	3.2k
AC for MIDP 2.0	8k
MIDP 3.0 formalization	5.6k
Framework	3.2k
Total	20k

Figure 5.1: LOC of Coq development – security for mobile devices

Further work is the study and specification, using the formal setting provided by the framework, of algorithms for enforcing the security policies derived from different sort of permission models to control the access to sensitive resources of the devices. Moreover, one main objective is to extend the framework so as to be able to construct certified prototypes from the formal definitions of those algorithms. Finally, an exhaustive formal comparison between both JME-MIDP and Android security models is proposed as further work. We have begun developing a formal specification of the Android security model in Coq, considering [88, 125, 126, 127, 128, 129, 130, 131, 132, 133], which focuses on the analysis of the permission system in general, and in the scheme of permission re-delegation, in particular [134].

5.2 A model of virtualization

We have presented a formalization of an idealized model of a hypervisor —initially introduced in [94]— and established within this model important security properties

that are expected from virtualization platforms. In particular, our verification effort showed that the model is adequate to reason about safety properties (read and write isolation), 2-safety properties (OS isolation), and liveness properties (availability). The basic formal development is about 20kLOC of Coq (see Figure 5.2), including proofs, and forms a suitable basis for reasoning about hypervisors.

Model and basic lemmas	4.8k
Valid state invariance	8.0k
Read and write isolation	0.6k
OS Isolation and lemmas	6.0k
Traces, safety and availability	1.0k
Total	20.4k

Figure 5.2: LOC of Coq development – model of virtualization

Additionally, we have enhanced the idealized model of virtualization considered in [94] with an explicit treatment of errors, and showed that OS isolation is preserved in this setting. Moreover we have implemented an executable specification that realizes the axiomatic semantics used in [94], for the model extended with cache and TLB. Finally, we have proved that in our idealized model the virtualization platform is transparent for guest operating systems in the sense that they cannot tell whether they execute alone or together with other systems on the platform. The formal development for this extension is about 15 kLOC of Coq, where 8k correspond to the verified executable specification and 7k to the proofs on the extended model with errors.

There are several directions for future work: one immediate direction is to complete our formalization with a proof of correctness of the hypervisor, as in the Hyper-V verification project. We also intend to enrich our model with devices and interrupts, and to give sufficient conditions for isolation properties to remain valid in this extended setting. Another immediate direction is to prove availability properties on an implementation of the hypervisor. Additionally, we intend to implement and analyze alternative executable semantics for different models of cache and policies. Finally, it is of interest to understand how to adapt our models to other virtualization paradigms such as full virtualization and microvisors.

5.3 The DNSSEC model

We have developed an abstract model of DNS that incorporates the security extensions defined by the DNSSEC specification suite. We have established and proved the security properties required to be satisfied by the operational behaviour of an implementation of this version of DNS, which is specified in our model by an abstract state and the events that represent the working of the system. In particular, this result provides a formal means to assess the effectiveness of a (correct) deployment of the security requirements specified by DNSSEC to prevent cache poisoning attacks.

In addition to that, we have identified an exploitable vulnerability that, according to our understanding, emerges as a flaw of the specification in question. As shown in Section 4.3, the conditions that must be verified in the case a rollover of a zone key must be performed, as specified in the RFC 4033 [47], do not suffice to ensure the validity of

the chain of trust. We have sketched in this article a formal proof that shows how the chain of trust can be compromised if only the expiration time of a key is considered as the cause to perform the rollover procedure. This property is established as a lemma in the formalization available in `http://www.fing.edu.uy/inco/grupos/gsi/sources/dnssec`.

The formal development is about 5kLOC of Coq (see Figure 5.3), including proofs, and forms a suitable basis for reasoning about DNSSEC.

Model and basic lemmas	2k
Valid state invariance	3k
Proof of inconsistency	0.2k
Total	5.2k

Figure 5.3: LOC of Coq development – DNSSEC formalization

There are several directions for future work. One immediate direction is to extend our specification and to establish results concerning the impact of introducing resource records of type *NSEC*. An NSEC record points to the next valid name in the zone file and is used to provide proof of non-existence of names within a zone. Through repeated queries that return NSEC records it is possible to retrieve all of the names in the zone, a process commonly called *walking* the zone. This side effect of the NSEC architecture subverts policies frequently implemented by zone owners which forbid zone transfers by arbitrary clients. We see as an interesting and challenging task to specify ways of preventing zone walking by constructing NSEC records that cover fewer names [135].

Bibliography

[1] Housley, R., Ford, W., Polk, W., Solo, D.: Internet X.509 Public Key Infrastructure Certificate and CRL Profile (1999)

[2] Gollmann, D.: Computer Security (3. ed.). Wiley (2011)

[3] Mclean, J.: Security models. In: Encyclopedia of Software Engineering, Wiley and Sons (1994)

[4] Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1) (2000) 30–50

[5] Anderson, J.P.: Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA (1972)

[6] Bell, D.E., LaPadula, L.J.: Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA (1973)

[7] Department of Defense: Department of Defense Trusted Computer System Evaluation Criteria. (1985) DOD 5200.28-STD (supersedes CSC-STD-001-83).

[8] CCMB-2012: Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model, Version 3.1. (2012)

[9] Chetali, B., Nguyen, Q.H.: About the world-first smart card certificate with eal7 formal assurances. Slides 9th ICCC, Jeju, Korea (2008) Available at: `www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf`.

[10] Betarte, G., Giménez, E., Loiseaux, C., Chetali, B.: FORMAVIE: Formal Modeling and Verification of the Java Card 2.1.1 Security Architecture. In: Proceedings of eSmart'02. (2002)

[11] Andronick, J.: Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation. PhD thesis, Université Paris-Sud (2006)

[12] The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.4. (2012)

[13] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer-Verlag (2004)

[14] Cohen, E.: Validating the microsoft hypervisor. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM '06. Volume 4085 of LNCS. Springer (2006) 81–81

[15] Leinenbach, D., Santen, T.: Verifying the microsoft hyper-v hypervisor with vcc. In Cavalcanti, A., Dams, D., eds.: FM 2009. Volume 5850 of LNCS., Springer (2009) 806–809

[16] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM (CACM) **53**(6) (2010) 107–115

[17] Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In Hankin, C., Siveroni, I., eds.: Proceedings of SAS'05. Volume 3672 of Lecture Notes in Computer Science., Springer-Verlag (2005) 352–367

[18] Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6) (2010) 1157–1210

[19] Grupo de Seguridad Informática: Projects: STEVE, ReSeCo, VirtualCert, and VirtualCert - phase II (2013) Available at: `www.fing.edu.uy/inco/grupos/gsi`.

[20] Sun Microsystems, Inc.: Java Platform Micro Edition. (Last accessed: March 2013) Available at: `//java.sun.com/javame/index.jsp`.

[21] JSR 139 Expert Group: Connected Limited Device Configuration. Version 1.1. Sun Microsystems, Inc. and Motorola, Inc. (2003)

[22] JSR 37 Expert Group: Mobile information device profile for java 2 micro edition. version 1.0. Technical report, Sun Microsystems, Inc. (2000)

[23] JSR 118 Expert Group: Mobile information device profile for java 2 micro edition. version 2.0. Technical report, Sun Microsystems, Inc. and Motorola, Inc. (2002)

[24] JSR 271 Expert Group: Mobile information device profile for java micro edition. version 3.0. Technical report, Motorola, Inc. (2009)

[25] Open Handset Alliance: Android project. (Last accessed: March 2013) Available at: `//source.android.com/`.

[26] Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Security and Privacy **7** (2009) 50–57

[27] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google android: A comprehensive security assessment. IEEE Security and Privacy **8**(2) (2010) 35–44

[28] Wallach, D.S., Felten, E.W.: Understanding java stack inspection. In: Proceedings of the 1998 IEEE Symposium on Security and Privacy. (1998) 52–63

[29] Besson, F., Dufay, G., Jensen, T.: A formal model of access control for mobile interactive devices. In: 11th European Symposium on Research in Computer Security (ESORICS'06), LNCS 4189. (2006) 110–126

[30] Garfinkel, T., Warfield, A.: What virtualization can do for security. ;login: The USENIX Magazine **32** (2007)

[31] Cohen, E., Paul, W.J., Schmaltz, S.: Theory of multi core hypervisor verification. In van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J.R., Sack, H., eds.: SOFSEM. Volume 7741 of Lecture Notes in Computer Science., Springer (2013) 1–27

[32] Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 enforces integrity. In: 2nd Conference on Interactive Theorem Proving, Nijmegen, The Netherlands (2011)

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

[34] Oheimb, D.v.: Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In Samarati, P., Ryan, P., Gollmann, D., Molva, R., eds.: Computer Security – ESORICS 2004. Volume 3193 of LNCS., Springer (2004) 225–243

[35] Mockapetris, P.: Domain names - concepts and facilities. RFC 1034 (Standard) (1987) Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[36] Mockapetris, P.: Domain names - implementation and specification. RFC 1035 (Standard) (1987) Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[37] Bellovin, S.M.: Using the Domain Name System for System Break-ins. In: Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5, Berkeley, CA, USA, USENIX Association (1995) 18–18

[38] Bellovin, S.M.: Security problems in the tcp/ip protocol suite. SIGCOMM Comput. Commun. Rev. **19** (1989) 32–48

[39] Cert/c, C.: Cert advisory ca-2001-02 multiple vulnerabilities in bind (2001) Available at: `http://www.cert.org/advisories/CA-2001-02.html`.

[40] Gavron, E.: RFC 1535: A security problem and proposed correction with widely deployed DNS software (1993) Status: INFORMATIONAL.

[41] Purdue University. Dept. of Computer Sciences and Schuba, C.L.: Addressing weaknesses in the domain name system protocol. Number n.º 28 in CSD-TR / Computer Sciences Department, Purdue University. Purdue University, Dept. of Computer Sciences (1994)

[42] Vixie, P.: Dns and bind security issues. In: Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5, Berkeley, CA, USA, USENIX Association (1995) 19–19

[43] Davidowicz, D.: Domain Name System (DNS) Security (1999) Available at: `//compsec101.antibozo.net/papers/dnssec/dnssec.html`.

[44] Atkins, D., Austein, R.: Threat analysis of the domain name system. In: DNS. RFC 3833, Internet Engineering Task Force. (2004)

[45] Herzberg, A., Shulman, H.: Security of patched dns. CoRR **abs/1205.5190** (2012)

[46] Son, S., Shmatikov, V.: The hitchhiker's guide to dns cache poisoning. In Jajodia, S., Zhou, J., eds.: SecureComm. Volume 50 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering., Springer (2010) 466–483

[47] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard) (2005) Updated by RFC 6014.

[48] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard) (2005) Updated by RFCs 4470, 6014.

[49] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard) (2005) Updated by RFCs 4470, 6014.

[50] T. I. I. Foundation: .se top level domain (2011) Available at: `www.iis.se/`.

[51] IANA: Interim trust-anchor repository (2011) Available at: `//itar.iana.org/`.

[52] Internet Research Lab, UCLA CS Department: The secspider dnssec monitoring project (2011) Available at: `//secspider.cs.ucla.edu/`.

[53] Nominet: Nominet dnssec testbed (2011) Available at: `www.nominet.org.uk/registrars/DNSSEC/`.

[54] P.I.R. PIR: Org Top Level Domain (2011) Available at: `www.iana.org/domains`.

[55] Arends, R., Bellgrim, R., Dalitz, A., Dickinson, J.A., Jansen, J., Lloyd, S., Mekking, M., Morris, S., Post, R., Schaeffer, Y., Schlyter, J., Wallstrm, P.: The opendnssec project (Last accessed: April 2014) Available at: `www.test.org/doe/`.

[56] Yang, H., Osterweil, E., Massey, D., Lu, S., Zhang, L.: Deploying cryptography in internet-scale systems: A case study on dnssec. IEEE Trans. Dependable Sec. Comput. **8**(5) (2011) 656–669

[57] Letouzey, P.: A New Extraction for Coq. In Geuvers, H., Wiedijk, F., eds.: Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Volume 2646 of Lecture Notes in Computer Science., Springer-Verlag (2003)

[58] Letouzey, P.: Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq. PhD thesis, Université Paris-Sud (2004)

[59] Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52** (2009) 107–115

[60] Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. SIGPLAN Not. **44**(1) (2009) 90–101

[61] Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2-3) (1988) 95–120

[62] Coquand, T., Paulin-Mohring, C.: Inductively defined types. In Martin-Löf, P., Mints, G., eds.: COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988. Volume 417 of Lecture Notes in Computer Science., Springer-Verlag (1990) 50–66

[63] Paulin-Mohring, C.: Inductive definitions in the system coq - rules and properties. In Bezem, M., Groote, J.F., eds.: 1st Int. Conf. on Typed Lambda Calculi and Applications. Volume 664 of LNCS., Springer-Verlag (1993) 328–345

[64] Housley, R., Ford, W., Polk, W., Solo, D.: Internet X.509 Public Key Infrastructure Certificate and CRL Profile (1999)

[65] Zanella Béguelin, S., Betarte, G., Luna, C.: A formal specification of the MIDP 2.0 security model. In: 4th International workshop on Formal Aspects in Security and Trust, FAST 2006. Volume 4691 of Lecture Notes in Computer Science., Springer (2006) 220–234

[66] Roushani Oskui, R., Betarte, G., Luna, C.: A certified access controller for jme-midp 2.0 enabled mobile devices. In: 2009 International Conference of the Chilean Computer Science Society, SCCC 2009, Los Alamitos, CA, USA, IEEE Computer Society (2009) 51–58

[67] Mazeikis, G., Betarte, G., Luna, C.: Formal specification and analysis of the MIDP 3.0 security model. In: 2009 International Conference of the Chilean Computer Science Society, SCCC 2009, Los Alamitos, CA, USA, IEEE Computer Society (2009) 59–66

[68] Crespo, J., Betarte, G., Luna, C.: A framework for the analysis of access control models for interactive mobile devices. In: Types for Proofs and Programs, International Conference, TYPES 2008. Volume 5497 of Lecture Notes in Computer Science., Springer (2009) 49–63

[69] Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)

[70] Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction (Texts in Computer Science). Springer (1998)

[71] Morgan, C.: Programming from specifications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)

[72] Roushani Oskui, R.: Especificación formal en coq del módulo de control de acceso de MIDP 2.0 para dispositivos móviles interactivos. Technical report, Master's thesis, Universidad Nacional de Rosario (2008)

[73] Mazeikis, G., Luna, C.: Autorización de acceso en midp 3.0. In: V Congreso Iberoamericano de Seguridad Informática, CIBSI 2009. (2009) 283–297

[74] Forte, J.: Formalización del protocolo de comunicación entre aplicaciones para dispositivos móviles java. Master's thesis, FCEIA, Universidad Nacional de Rosario, Argentina (2013)

[75] Prince, C.: Análisis formal de la instalación de aplicaciones en midp 3.0. Master's thesis, FCEIA, Universidad Nacional de Rosario, Argentina (2014)

[76] Jensen, T., Métayer, D.L., Thorn, T.: Verification of control flow based security properties. In: Proc. of the 20th IEEE Symp. on Security and Privacy, New York: IEEE Computer Society (1999) 89–103

[77] Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. J. Comput. Secur. **9** (2001) 217–250

[78] Bartoletti, M., Degano, P., Ferrari, G.: Static analysis for stack inspection. In: Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science. Volume 54., Elsevier (2001) 2001

[79] Kolsi, O.: Midp 2.0 security enhancements. In: In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS04) - Track 9 - Volume 9. (2004) 287–294

[80] Mohamed, M.D., Saleh, M., Talhi, C., Zhioua, S.: Security analysis of wireless java. In: In Proceedings of the 3rd Annual Conference on Privacy, Security and Trust, PST05. (2005) 1–11

[81] Fournet, C., Gordon, A.D.: Stack inspection: Theory and variants. ACM Trans. Program. Lang. Syst. **25** (2003) 360–399

[82] mo Chang, B.: Static check analysis for java stack inspection. ACM SIGPLAN Notices **41** (2006)

[83] Besson, F., Dufay, G., Jensen, T., Pichardie, D.: Verifying resource access control on mobile interactive devices. J. Comput. Secur. **18** (2010) 971–998

[84] Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: Towards formal analysis of the permission-based security model for android. In: Proceedings of the 2009 Fifth International Conference on Wireless and Mobile Communications. ICWMC '09, Washington, DC, USA, IEEE Computer Society (2009) 87–92

[85] Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the Android framework. In: 2010 IEEE 2nd International Conference on Social Computing, Los Alamitos, CA, USA, IEEE Computer Society (2010) 944–951

[86] Open Handset Alliance: Android developers. (Last accessed: April 2014) Available at: `developer.android.com/index.html`.

[87] Chaudhuri, A.: Language-based security on android. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security. PLAS '09, New York, NY, USA, ACM (2009) 1–7

[88] Romano, A., Luna, C.: Descripción y análisis del modelo de seguridad de Android. Technical Report RT 13-08, PEDECIBA Informática (2013)

[89] Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. **27**(2) (2005) 264–313

[90] Bartoletti, M., Degano, P., Ferrari, G.L.: History-based access control with local policies. In: In Proceedings of FOSSACS 2005, Springer (2005) 316–332

[91] Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.C.: Enforcing resource bounds via static verification of dynamic checks. ACM Trans. Program. Lang. Syst. **29** (2007)

[92] Goldberg, R.P.: Survey of virtual machine research. IEEE Computer Magazine **7** (1974) 34–45

[93] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (2003) 164–177

[94] Barthe, G., Betarte, G., Campo, J., Luna, C.: Formally verifying isolation and availability in an idealized model of virtualization. In Butler, M., Schulte, W., eds.: Formal Methods. Volume 6664 of LNCS., Springer-Verlag (2011)

[95] Barthe, G., Betarte, G., Campo, J.D., Chimento, J.M., Luna, C.: Formally Verified Implementation of an Idealized Model of Virtualization. In Matthes, R., Schubert, A., eds.: 19th International Conference on Types for Proofs and Programs (TYPES 2013). Volume 26 of Leibniz International Proceedings in Informatics (LIPIcs)., Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2014) 45–63

[96] Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, London, UK, Springer-Verlag (1982) 52–71

[97] van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. Elsevier and MIT Press (1990)

[98] Luna, C.: Computation tree logic for reactive systems and timed computation tree logic for real time systems. Contributions to the Coq system, Universidad de la República, Uruguay (2000)

[99] Oheimb, D.v., Lotz, V., Walter, G.: Analyzing SLE 88 memory management security using Interacting State Machines. International Journal of Information Security **4**(3) (2005) 155–171

[100] Banerjee, A., Naumann, D.: Stack-based access control for secure information flow. Journal of Functional Programming **15** (2005) 131–177 Special Issue on Language-Based Security.

[101] Rushby, J.M.: Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International (1992)

[102] Barthe, G., Betarte, G., Campo, J., Luna, C.: Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In: IEEE 25th Computer Security Foundations Symposium (CSF). (2012) 186–197

[103] Hwang, J.Y., Suh, S.B., Heo, S.K., Park, C.J., Ryu, J.M., Park, S.Y., Kim, C.R.: Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In: 5th IEEE Consumer and Communications Networking Conference. (2008)

[104] The VirtualCert project: Supporting Coq formalization (2013) Available at: `www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

[105] Klein, G.: Operating system verification – an overview. Sādhanā **34**(1) (2009) 27–69

[106] Shao, Z.: Certified software. Commun. ACM **53**(12) (2010) 56–66

[107] Alkassar, E., Hillebrand, M., Paul, W., Petrova, E.: Automated verification of a small hypervisor. In Leavens, G., OHearn, P., Rajamani, S., eds.: Verified Software: Theories, Tools, Experiments. Volume 6217 of LNCS. Springer (2010) 40–54

[108] Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W.: Verifying shadow page table algorithms. In Bloem, R., Sharygina, N., eds.: Formal Methods in Computer-Aided Design, 10th International Conference (FMCAD'10), Switzerland, IEEE CS (2010)

[109] Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive verification of an os microkernel: Inline assembly, memory consumption, concurrent devices. In: Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10). Volume 6217 of LNCS., Edinburgh, Springer (2010) 71–85

[110] Elkaduwe, D., Klein, G., Elphinstone, K.: Verified protection model of the sel4 microkernel. In: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments. VSTTE '08, Springer (2008) 99–114

[111] Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. Commun. ACM **54**(12) (2011) 123–131

[112] Tews, H., Weber, T., Poll, E., Eekelen, M.v.: Formal Nova interface specification. Technical Report ICIS–R08011, Radboud University Nijmegen (2008) Robin deliverable D12.

[113] Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: Formal specification and verification of data separation in a separation kernel for an embedded system.

In: Proceedings of the 13th ACM conference on Computer and communications security. CCS '06, NY, USA, ACM (2006) 346–355

[114] Greve, D., Wilding, M., Eet, W.M.V.: A separation kernel formal security policy. In: Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications. (2003)

[115] Martin, W., White, P., Taylor, F., Goldberg, A.: Formal construction of the mathematically analyzed separation kernel. In: The Fifteenth IEEE International Conference on Automated Software Engineering. (2000)

[116] Andronick, J., Chetali, B., Ly, O.: Using Coq to Verify Java Card Applet Isolation Properties. In Basin, D.A., Wolff, B., eds.: International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03). Volume 2758 of LNCS., Springer-Verlag (2003) 335–351

[117] Tollitte, P.N., Delahaye, D., Dubois, C.: Producing certified functional code from inductive specifications. In Hawblitzel, C., Miller, D., eds.: CPP. Volume 7679 of LNCS., Springer (2012) 76–91

[118] Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: TPHOLs. Volume 5674 of LNCS., Springer (2009) 131–146

[119] Balaa, A., Bertot, Y.: Fix-point equations for well-founded recursion in type theory. In Aagaard, M., Harrison, J., eds.: TPHOLs. Volume 1869 of Lecture Notes in Computer Science., Springer (2000) 1–16

[120] Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In Hagiya, M., Wadler, P., eds.: FLOPS. Volume 3945 of LNCS., Springer (2006) 114–129

[121] Liu, C., Albitz, P.: DNS and BIND - help for system administrators: covers BIND 9.3 (5. ed.). O'Reilly (2006)

[122] Cheung, S., Levitt, K.N.: A formal-specification based approach for protecting the domain name system. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks. (2000) 25–28

[123] Eixarch, E.B., Betarte, G., Luna, C.D.: A formal specification of the dnssec model. ECEASST **48** (2011)

[124] Bau, J., Mitchell, J.C.: A security evaluation of dnssec with nsec3. IACR Cryptology ePrint Archive **2010** (2010) 115

[125] Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the android framework. In: Proceedings of the 2010 IEEE Second International Conference on Social Computing. SOCIALCOM '10, Washington, DC, USA, IEEE Computer Society (2010) 944–951

[126] May, M.J., Bhargavan, K.: Towards unified authorization for android. In: Proceedings of the 5th International Conference on Engineering Secure Software and Systems. ESSoS'13, Berlin, Heidelberg, Springer-Verlag (2013) 42–57

[127] Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing android's permission system. In Foresti, S., Yung, M., Martinelli, F., eds.: ESORICS. Volume 7459 of Lecture Notes in Computer Science., Springer (2012) 1–18

[128] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: Attacks and defenses. In: Proceedings of the 20th USENIX Conference on Security. SEC'11, Berkeley, CA, USA, USENIX Association (2011) 22–22

[129] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS '11, New York, NY, USA, ACM (2011) 627–638

[130] Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Proceedings of the 13th International Conference on Information Security. ISC'10, Berlin, Heidelberg, Springer-Verlag (2011) 346–360

[131] Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. MobiSys '11, New York, NY, USA, ACM (2011) 239–252

[132] Bugliesi, M., Calzavara, S., Spanò, A.: Lintent: Towards security type-checking of android applications. In Beyer, D., Boreale, M., eds.: FMOODS/FORTE. Volume 7892 of Lecture Notes in Computer Science., Springer (2013) 289–304

[133] Armando, A., Costa, G., Merlo, A.: Formal modeling and reasoning about the android security framework. In Palamidessi, C., Ryan, M., eds.: TGC'12: 7th International Symposium on Trustworthy Global Computing. Volume 8191 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 64–81

[134] Romano, A.: Descripción y análisis formal del modelo de seguridad de android. Technical report, Master's thesis, Universidad Nacional de Rosario (2014) Available at: `www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es`.

[135] Weiler, S., Ihren, J.: Minimally Covering NSEC Records and DNSSEC On-line Signing. RFC 4470 (Proposed Standard) (2006)