Formal Analysis of Security Models for Mobile Devices, Virtualization Platforms, and Domain Name Systems

Gustavo Betarte
Instituto de Computación, Facultad de Ingeniería, Universidad de la República,
Montevideo, Uruguay, 11300,
gustun@fing.edu.uy

and

Carlos Luna
Instituto de Computación, Facultad de Ingeniería, Universidad de la República,
Montevideo, Uruguay, 11300,
cluna@fing.edu.uy

Abstract

In this work 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. 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 work we present a formalization of an idealized model of a hypervisor. 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 introduce 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.

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 paper we provide a detailed account of the work presented in [1], where we have investigated 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 [2]. *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 [3]. 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 [4]. 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 [5]. 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 [6]. The first model to appear was Bell-LaPadula [7], 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 [8]. This initiative opened the path to worldwide mutual recognition of security evaluation results. In this direction, new Common Criteria (CC) [9] 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 [10, 11, 12], using the Coq proof assistant [13, 14].

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 check if all 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. [15, 16, 17].

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 [18, 19] 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 [1], summarized in this paper.

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 2.5.

Contributions

The contributions of the work in this domain are manyfold:

- 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;
- 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 mechanism 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;
- 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;
- 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 [15, 16, 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 [17], 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 fulfill 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 work 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.

1.5 Formal language used

The Coq proof assistant [13, 14] 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]. 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 [58]; Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [59]. 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) [60, 61] 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`.

1.6 Document outline

The remainder of this document is organized as follows. In Section 2 we study and compare several variants of the security model specified by MIDP to access sensitive resources of a device. In Section 3 we present the formalization of an idealized model of a hypervisor. 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 Section 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 Section 5.

2 Formal analysis of security models for mobile devices

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.

This section builds upon a number of previously published papers [62, 63, 64, 65]. The rest of the section is organized as follows: Section 2.1 presents the MIDP security model; 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. Furthermore, briefly describes a high level formal specification of an access controller for JME-MIDP 2.0 and an algorithm that has been proved correct with respect to that specification. Section 2.3 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 2.4 introduces 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. Finally, Section 2.5 discusses related work and Section 2.6 presents conclusions and future work. The full development is available for download at `http://www.fing.edu.uy/inco/grupos/gsi/sources/midp`.

2.1 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.

2.1.1 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; and *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 [66]. 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.

2.1.2 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

```
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: pYRJ8Q1u5ITBLxcAUzYXDKnmg...
```

Figure 1: A typical MIDlet descriptor, specifying the required and optional permission of suite for a suite with two applications

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: applications bound to a specific domain; applications from a certain vendor with a certified signature; applications from a certain vendor but whithout a certified signature; 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.

2.2 Formally verifying security properties of MIDP 2.0

In this section we overview the formalization of the MIDP 2.0 security model presented in [62].

2.2.1 Applications

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 1.

2.2.2 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 structure within the state.

We define a notion of valid state, through a predicate ($Valid$) on states, that captures essential properties of the platform. For instance, a property states that 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) [62].

2.2.3 Events

We define a set *Event* for those events that are relevant to our abstraction of the device state (Table 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.

Table 1: Security-related events

Name	**Description**
start	Start of session
terminate	End of session
request	Permission request
install	MIDlet suite installation
remove	MIDlet suite removal

The behaviour of the events is specified by their pre- and postconditions given by the predicates *Pre* and *Post* respectively. Preconditions are defined in terms of the device state while postconditions 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 [62].

2.2.4 One-step execution

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

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

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 $s \xhookrightarrow{e/r} s'$ may be read as "the execution of the event e in state s results in a new state s' and produces a response r".

2.2.5 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* 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,

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

2.2.6 Verification of security properties

This section is devoted to mention some relevant security properties of the model [1].

We call one-step invariant a property that remains true after the *execution* of every event if it is true before. We prove that the validity of the device state is a one-step invariant of our specification: for any $s\ s' : State$, $r : Response$ and $e : Event$, if $Valid\ s$ and $s \xhookrightarrow{e/r} s'$ hold, then $Valid\ s'$ also holds.

We call session invariant a step property that holds for the rest of a session once it is established in a step. We prove that state validity is a session invariant.

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 [62].

2.2.7 Executable specification

In the formalization described previously 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 show in [62] 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 [67] on operation and data refinement, and the more comprehensive works of Back and von Wright [68] and Morgan [69] on refinement calculus.

In [63], we developed 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. In this algorithm, 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.

2.3 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. 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 [64], the formalization described in Section 2.2 is extended so as to model security at the application level, introduced in Section 2.1. The extended device state (with allowed and denied access authorizations), a new condition for state validity, and the authorization event (that models the access authorization request from a MIDlet suite to the shared resources of the active suite) 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. On the other hand, an algorithm for the access authorization of applications was developed and its correctness proven. Additionally, certified algorithms have been developed for application installation and communication between applications (IMC protocol) in MIDP 3.0 [70, 71].

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. 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 [1].

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 (when the vendor name is used 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.

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.

2.4 A framework for defining and comparing access control policies

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, based on [65, 1], 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 a multiplicity to a permission, 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 [72, 73].

One of the objectives of the work reported in [65, 1], 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 Related work

Some effort has been put into the evaluation of the security model for MIDP 2.0; Kolsi and Virtanen [74] and Debbabi et al. [75] 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 [73, 76, 72, 77]. 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, 78] 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, based on [65, 1], 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, 78].

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 [79], the authors propose an entity-relationship model for the Android permission scheme [80]. 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 [62]. The abstract operation set considered in [79] does not include permission request/revoke operations presents in MIDP. In [81], 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 [82], which also details the Android security model and analyzes some of the more recent works that formalize different aspects of this model [83].

Language-based access control has been investigated for some idealised program models, see *e.g.* [84, 85, 86]. 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.

2.6 Summary and future work

We have provided the first verifiable formalization of the MIDP 2.0 security model, according to [62], 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 [1]. 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.

Future 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 [82, 87, 88, 89, 90, 91, 92], which focuses on the analysis of the permission system in general, and in the scheme of permission re-delegation, in particular [93].

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. This allows 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 [15, 16] and L4.verified [17] 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 [18, 19] 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.

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 [94], 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 [95]. 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 rest of the section

This section builds upon the previously published papers [96, 97]. In Section 3.2 we briefly present 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. In Section 3.7 we discuss the isolation theorems for the extended model with execution errors and a proof of transparency. Finally, Section 3.8 considers related work and Section 3.9 summarizes the conclusions and future work. The formal development is available at `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

3.2 The model

In this section we briefly present and discuss some aspects of 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 2, 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

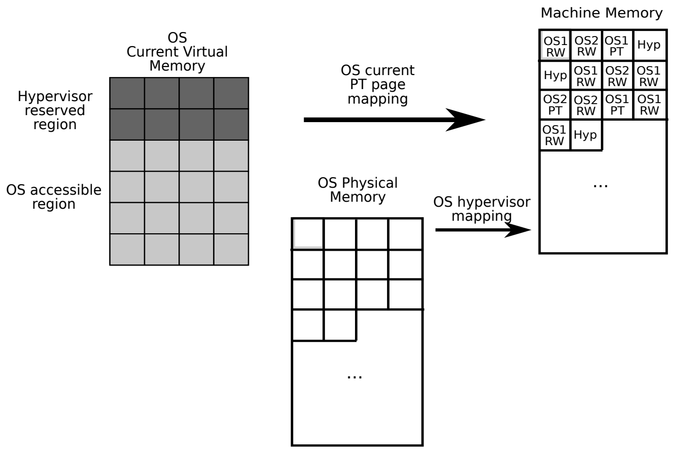

Figure 2: Memory model of the platform

used 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 [95] 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

read va	A guest OS reads virtual address *va*.
write va val	A guest OS writes value *val* in virtual address *va*.
switch o	The hypervisor sets *o* to be the active OS.
hcall c	An untrusted OS requires privileged service *c* to be executed by the hypervisor.
new 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 OS *o*, where *pa* translates to *ma* for *o*.
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.
pin o pa t	The memory page that corresponds to physical address *pa* (for untrusted OS *o*) is registered and classified with type *t*.

Table 2: Platform actions

with read/write (RW) 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 *pin* and *unpin* [1].

3.2.2 Formalizing states

The states of the platform are modeled by a record (*State*) with six components: 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 previously.

We define a notion of valid state (*valid_state*) that captures essential properties of the platform. For instance, two properties states that: 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; and 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.

3.2.3 Actions and executions

Table 2 summarises a small subset –due to space restrictions– of the actions specified in the model [1], 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.

The behaviour of actions is specified by a precondition *Pre* and by a postcondition *Post*. The execution of an action is specified by the $\hookrightarrow$ relation:

$$\frac{\mathit{valid\_state}(s) \quad \mathit{Pre}\ s\ a \quad \mathit{Post}\ s\ a\ s'}{s \overset{a}{\hookrightarrow} s'}$$

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 $s \overset{a}{\hookrightarrow} s'$ may be read as *the execution of the action a in a valid state s results in a new state* s'.

One-step execution preserves valid states. 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.

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

$$s_0 \overset{a_0}{\hookrightarrow} s_1 \overset{a_1}{\hookrightarrow} s_2 \overset{a_2}{\hookrightarrow} s_3 \ldots$$

such that every execution step $s_i \overset{a_i}{\hookrightarrow} s_{i+1}$ is valid.

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.

3.3.1 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.

3.3.2 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.

3.3.3 OS Isolation

OS isolation is a 2-safety property [18, 19], cast in terms of two executions of the system, and is closely related to the non-influence property studied by Oheimb and co-workers [34, 98]. 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* between states w.r.t. an operating system *osi* ($\equiv_{osi}$). For instance, two conditions which are part of this relation of equivalence are: 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; and 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. Formally, OS isolation states that *osi*-equivalence is preserved under execution of any action, and is formalized as a "step-consistent" unwinding lemma, see [99].

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

All isolation properties extend to traces, using coinductive reasoning principles [1]. In particular, the extension of OS isolation to traces establishes a non-influence property [34].

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.

3.5 Extension of the model with cache and TLB

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 [1, 100], we present an extension of the idealized model of virtualization, introduced in Section 3.2, that features cache and TLB. We provide a model of a VIVT cache and TLB (as in Xen on ARM [101]), for a write-through policy. Figure 3 shows a diagram of the extended memory model of the platform. 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.

3.6 Verified implementation

In [97], 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. 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.

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

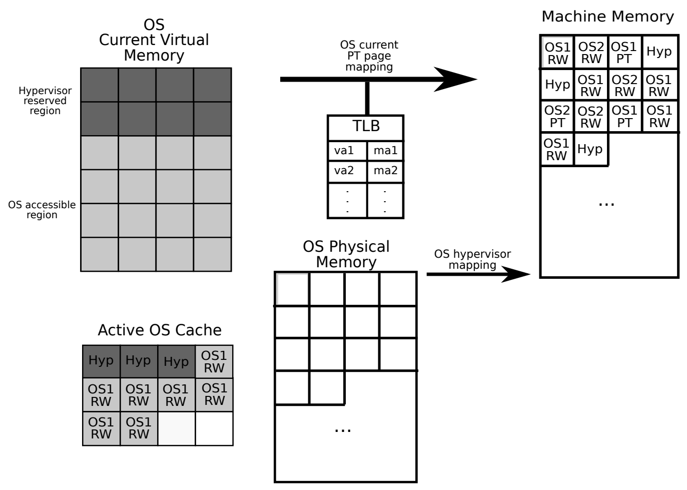

Figure 3: Memory model of the platform with cache and TLB

execution is the corresponding sequence of memory states (the trace of execution) obtained while executing the sequence of actions.

In [102], we derive two certified hypervisor implementations, using the extraction mechanism of Coq, in functional languages Haskell and OCaml.

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 [97], we extend the proof of OS isolation from Section 3.3.3, yielding modifications in some key technical definitions and lemmas, so that it accounts for errors in execution traces for the model extended with cache and TLB. In particular, 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. OS isolation formally establishes that, two traces are equivalent w.r.t an OS *osi*, if they have the same set of actions w.r.t. *osi* and if their initial states are *osi*-equivalent [1].

In addition, we present, in [97], 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. The transparency property can then be expressed as: given an OS *osi* and any valid trace of the system, the erasure of the trace is another trace *osi*-equivalent to the original one [1].

Finally, 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. On the other hand, we have fixed a *write-through* policy for the main memory. However, we ensure 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 [103].

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 [104, 105].

The Microsoft Hyper-V verification project focuses on proving the functional correctness of the deployed implementation of the Hyper-V hypervisor [15, 16] or of a simplified, baby, implementation [106]. 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 [107] and devices [108].

The L4.verified project [17] 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 [109, 33]. The formalization does not model cache.

Recently, the Verve project [110] 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 [111], 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 [112, 113, 114] 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 [115] 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, 98] 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* [116] show how to extract a functional implementation from an inductive specification in the Coq proof assistant. Similar approaches exist for Isabelle, see e.g. [117]. Earlier, alternative approaches such as [118, 119] aim to provide reasoning principles for executable specifications.

3.9 Summary and future work

We have presented a formalization of an idealized model of a hypervisor —initially introduced in [96]— 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, including proofs, and forms a suitable basis for reasoning about hypervisors.

Additionally, we have enhanced the idealized model of virtualization considered in [96] 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 [96], 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. Some advances in these lines were published in [100, 103].

4 A formal specification of the DNSSEC model

4.1 DNS and DNSSEC

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.

4.2 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; and *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 [120, 35, 36].

In an early work, Cheung and Levitt [121] 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; and 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.

4.3 Formalization of the DNSSEC model

In [122, 1], 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. 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. First, we define the set of states and a notion of valid state. Then we define the security-related events as state transformers and provide a formal definition of their execution. Finally, we present in the thesis [1] some of the verified properties. The full development is available for download at `http://www.fing.edu.uy/inco/grupos/gsi/sources/dnssec.`

4.3.1 States

To reason about the DNSSEC security system most details of the state may be abstracted. In the formal specification, states are modeled by a record type ($State$) with components to represent, in particular:

- the involved DNS servers;
- the set of publicly known trusted servers;
- the set of keys for every server;
- the delegations issued by each server;
- the fathers of a server;
- the authoritative view and the cache view of every server;
- the expected answers to the already performed queries and the corresponding pending submissions for each server.

In [122, 1], we define also a notion of valid state ($Valid$) 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].

4.3.2 Events

The working of the system is modelled in terms of the execution of events. An event (of type *Event*) is understood as an action that transforms the state of the system. We only show here a brief description of three distinguished events:

- $Server\_ZSK\_rollover$ performs a rollover of the zone key of a given server;
- *Receive_Response* receives a response from a given server; and
- *RR_TimeOut* indicates that the TTL of a given set of resource records that share the same domain name, type and class has expired.

The behavior of the events is specified by their pre- and postconditions. 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 $Post$.

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

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

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_pre*. However, if $Pre\ s\ e$ is not satisfied, then the state s remains unchanged and the system answer is the error message determined by a relation $ErrorMsg$, involving error codes (of type $ErrorCode$) [122, 1].

4.3.3 Verification of security properties

In [1], 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: 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.

We 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. In particular, the instantiation of proposition that states the invariance of valid state 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.

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 set of resource records that share the same domain name, type and class (RRset) within the zone must be signed to generate the new RRSIG records (signatures over RRsets made using private key). 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 [122, 1].

4.4 Summary and future work

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. In particular, 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 [1] 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, and forms a suitable basis for reasoning about DNSSEC.

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 [123].

5 Conclusion

The increasingly important role of critical components in software systems make them a prime target for formal verification. Indeed, several projects have set out to formally verify the correctness of critical system implementations. 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. 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.

Our work shows that it is feasible to formally analyze models of safety-critical applications, using specifications based on state machines, 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.

In this paper we present independent formalizations for the considered applications. Future work is to develop a framework in Coq (a Coq library) to specify and reason about state machines, that allow represent complex critical systems.

References

[1] C. Luna, “Formal analysis of security models for mobile devices, virtualization platforms, and domain name systems,” Ph.D. dissertation, PEDECIBA Informática (TR 14-11), Facultad de Ingeniería, Universidad de la República, Uruguay, 2014, available: http://www.fing.edu.uy/inco/pedeciba/bibliote/tesis/tesisd-luna.pdf.

[2] R. Housley, W. Ford, W. Polk, and D. Solo, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile,” United States, 1999. [Online]. Available: http://portal.acm.org/citation.cfm?id=RFC2459

[3] D. Gollmann, *Computer Security (3. ed.).* Wiley, 2011.

[4] J. Mclean, “Security models,” in *Encyclopedia of Software Engineering.* Wiley and Sons, 1994.

[5] F. B. Schneider, “Enforceable security policies,” *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000. [Online]. Available: http://doi.acm.org/10.1145/353323.353382

[6] J. P. Anderson, “Computer Security technology planning study,” Deputy for Command and Management System, USA, Tech. Rep., 1972. [Online]. Available: http://csrc.nist.gov/publications/history/ande72.pdf

[7] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE Corp., Bedford, MA, Tech. Rep. MTR-2547, Vol. 1, 1973.

[8] *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense, Dec. 1985, dOD 5200.28-STD (supersedes CSC-STD-001-83).

[9] *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model, Version 3.1.*, CCMB-2012, Sep. 2012.

[10] B. Chetali and Q.-H. Nguyen, "About the world-first smart card certificate with eal7 formal assurances," Slides 9th ICCC, Jeju, Korea, Sep. 2008, available: www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf.

[11] G. Betarte, E. Giménez, C. Loiseaux, and B. Chetali, "FORMAVIE: Formal Modeling and Verification of the Java Card 2.1.1 Security Architecture," in *Proceedings of eSmart'02*, 2002.

[12] J. Andronick, "Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation," Ph.D. dissertation, Université Paris-Sud, 2006.

[13] The Coq Development Team, *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012. [Online]. Available: coq.inria.fr

[14] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. Springer-Verlag, 2004. [Online]. Available: www.labri.fr/publications/l3a/2004/BC04

[15] E. Cohen, "Validating the microsoft hypervisor," in *FM '06*, ser. LNCS, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Springer, 2006, vol. 4085, pp. 81–81. [Online]. Available: http://dx.doi.org/10.1007/11813040_6

[16] D. Leinenbach and T. Santen, "Verifying the microsoft hyper-v hypervisor with vcc," in *FM 2009*, ser. LNCS, A. Cavalcanti and D. Dams, Eds., vol. 5850. Springer, 2009, pp. 806–809.

[17] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," *Communications of the ACM (CACM)*, vol. 53, no. 6, pp. 107–115, Jun. 2010.

[18] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Proceedings of SAS'05*, ser. Lecture Notes in Computer Science, C. Hankin and I. Siveroni, Eds., vol. 3672. Springer-Verlag, 2005, pp. 352–367.

[19] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[20] *Java Platform Micro Edition.*, Sun Microsystems, Inc., Last accessed: March 2013, available: //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," Sun Microsystems, Inc., Tech. Rep., 2000.

[23] JSR 118 Expert Group, "Mobile information device profile for java 2 micro edition. version 2.0," Sun Microsystems, Inc. and Motorola, Inc., Tech. Rep., 2002.

[24] JSR 271 Expert Group, "Mobile information device profile for java micro edition. version 3.0," Motorola, Inc., Tech. Rep., 2009.

[25] *Android project*, Open Handset Alliance, Last accessed: March 2013, available: //source.android.com/.

[26] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, pp. 50–57, 2009.

[27] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *IEEE Security and Privacy*, vol. 8, no. 2, pp. 35–44, 2010.

[28] D. S. Wallach and E. W. Felten, "Understanding java stack inspection," in *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998, pp. 52–63.

[29] F. Besson, G. Dufay, and T. Jensen, "A formal model of access control for mobile interactive devices," in *11th European Symposium on Research in Computer Security (ESORICS'06), LNCS 4189*, 2006, pp. 110–126.

[30] T. Garfinkel and A. Warfield, "What virtualization can do for security," *;login: The USENIX Magazine*, vol. 32, Dec. 2007.

[31] E. Cohen, W. J. Paul, and S. Schmaltz, "Theory of multi core hypervisor verification," in *SOFSEM*, ser. Lecture Notes in Computer Science, P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. R. Nawrocki, and H. Sack, Eds., vol. 7741. Springer, 2013, pp. 1–27.

[32] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *2nd Conference on Interactive Theorem Proving*, Nijmegen, The Netherlands, Aug 2011.

[33] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. G., and G. Klein, "sel4: from general purpose to a proof of information flow enforcement," in *IEEE Symposium on Security and Privacy*, 2013, pp. 415–429.

[34] D. v. Oheimb, "Information flow control revisited: Noninfluence = Noninterference + Nonleakage," in *Computer Security – ESORICS 2004*, ser. LNCS, P. Samarati, P. Ryan, D. Gollmann, and R. Molva, Eds., vol. 3193. Springer, 2004, pp. 225–243.

[35] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034 (Standard), Internet Engineering Task Force, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. [Online]. Available: www.ietf.org/rfc/rfc1034.txt

[36] ——, "Domain names - implementation and specification," RFC 1035 (Standard), Internet Engineering Task Force, Nov. 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. [Online]. Available: http://www.ietf.org/rfc/rfc1035.txt

[37] S. M. Bellovin, "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, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267591.1267609

[38] ——, "Security problems in the tcp/ip protocol suite," *SIGCOMM Comput. Commun. Rev.*, vol. 19, pp. 32–48, April 1989. [Online]. Available: http://doi.acm.org/10.1145/378444.378449

[39] C. Cert/c. (2001) Cert advisory ca-2001-02 multiple vulnerabilities in bind. Available: http://www.cert.org/advisories/CA-2001-02.html.

[40] E. Gavron, "RFC 1535: A security problem and proposed correction with widely deployed DNS software," Oct. 1993, status: INFORMATIONAL. [Online]. Available: ftp://ftp.internic.net/rfc/rfc1535.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1535.txt

[41] Purdue University. Dept. of Computer Sciences and Schuba, C.L., *Addressing weaknesses in the domain name system protocol*, ser. CSD-TR / Computer Sciences Department, Purdue University. Purdue University, Dept. of Computer Sciences, 1994, no. n.º 28. [Online]. Available: http://books.google.com/books?id=QDHDPgAACAAJ

[42] P. Vixie, "Dns and bind security issues," in *Proceedings of the 5th conference on USENIX UNIX Security Symposium - Volume 5*. Berkeley, CA, USA: USENIX Association, 1995, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267591.1267610

[43] D. Davidowicz, "Domain Name System (DNS) Security," 1999, available: //compsec101.antibozo.net/papers/dnssec/dnssec.html.

[44] D. Atkins and R. Austein, "Threat analysis of the domain name system," in *DNS. RFC 3833, Internet Engineering Task Force*, 2004.

[45] A. Herzberg and H. Shulman, "Security of patched dns," *CoRR*, vol. abs/1205.5190, 2012.

[46] S. Son and V. Shmatikov, "The hitchhiker's guide to dns cache poisoning," in *SecureComm*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, S. Jajodia and J. Zhou, Eds., vol. 50. Springer, 2010, pp. 466–483.

[47] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," RFC 4033 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFC 6014. [Online]. Available: http://www.ietf.org/rfc/rfc4033.txt

[48] ——, "Resource Records for the DNS Security Extensions," RFC 4034 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 4470, 6014. [Online]. Available: http://www.ietf.org/rfc/rfc4034.txt

[49] ——, "Protocol Modifications for the DNS Security Extensions," RFC 4035 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 4470, 6014. [Online]. Available: http://www.ietf.org/rfc/rfc4035.txt

[50] T. I. I. Foundation. (2011) .se top level domain. Available: www.iis.se/.

[51] IANA. (2011) Interim trust-anchor repository. Available: //itar.iana.org/.

[52] Internet Research Lab, UCLA CS Department. (2011) The secspider dnssec monitoring project. Available: //secspider.cs.ucla.edu/.

[53] Nominet. (2011) Nominet dnssec testbed. Available: www.nominet.org.uk/registrars/DNSSEC/.

[54] P.I.R. PIR, "Org Top Level Domain," 2011, available: www.iana.org/domains.

[55] R. Arends, R. Bellgrim, A. Dalitz, J. A. Dickinson, J. Jansen, S. Lloyd, M. Mekking, S. Morris, R. Post, Y. Schaeffer, J. Schlyter, and P. Wallstrm. (Last accessed: April 2014) The opendnssec project. Available: www.test.org/doe/.

[56] H. Yang, E. Osterweil, D. Massey, S. Lu, and L. Zhang, "Deploying cryptography in internet-scale systems: A case study on dnssec," *IEEE Trans. Dependable Sec. Comput.*, vol. 8, no. 5, pp. 656–669, 2011.

[57] P. Letouzey, "Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq," Ph.D. dissertation, Université Paris-Sud, Jul. 2004.

[58] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, pp. 107–115, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[59] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," *SIGPLAN Not.*, vol. 44, no. 1, pp. 90–101, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594834.1480894

[60] T. Coquand and G. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, no. 2-3, pp. 95–120, Feb. 1988. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(88)90005-3

[61] C. Paulin-Mohring, "Inductive definitions in the system coq - rules and properties," in *1st Int. Conf. on Typed Lambda Calculi and Applications*, ser. LNCS, M. Bezem and J. F. Groote, Eds., vol. 664. Springer-Verlag, 1993, pp. 328–345.

[62] S. Zanella Béguelin, G. Betarte, and C. Luna, "A formal specification of the MIDP 2.0 security model," in *4th International workshop on Formal Aspects in Security and Trust, FAST 2006*, ser. Lecture Notes in Computer Science, vol. 4691. Springer, 2006, pp. 220–234.

[63] R. Roushani Oskui, G. Betarte, and C. Luna, "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, pp. 51–58.

[64] G. Mazeikis, G. Betarte, and C. Luna, "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, pp. 59–66.

[65] J. Crespo, G. Betarte, and C. Luna, "A framework for the analysis of access control models for interactive mobile devices," in *Types for Proofs and Programs, International Conference, TYPES 2008*, ser. Lecture Notes in Computer Science, vol. 5497. Springer, 2009, pp. 49–63.

[66] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," United States, 1999. [Online]. Available: http://portal.acm.org/citation.cfm?id=RFC2459

[67] J. M. Spivey, *The Z notation: a reference manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[68] R. J. Back and J. Wright, *Refinement Calculus: A Systematic Introduction (Texts in Computer Science).* Springer, April 1998.

[69] C. Morgan, *Programming from specifications.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[70] J. Forte, "Formalización del protocolo de comunicación entre aplicaciones para dispositivos móviles java," Master's thesis, FCEIA, Universidad Nacional de Rosario, Argentina, Oct. 2013.

[71] C. Prince, "Análisis formal de la instalación de aplicaciones en midp 3.0," Master's thesis, FCEIA, Universidad Nacional de Rosario, Argentina, 2014.

[72] T. Jensen, D. L. Métayer, and T. Thorn, "Verification of control flow based security properties," in *Proc. of the 20th IEEE Symp. on Security and Privacy.* New York: IEEE Computer Society, may 1999, pp. 89–103.

[73] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn, "Model checking security properties of control flow graphs," *J. Comput. Secur.*, vol. 9, pp. 217–250, January 2001.

[74] O. Kolsi, "Midp 2.0 security enhancements," in *In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS04) - Track 9 - Volume 9*, 2004, pp. 287–294.

[75] M. D. Mohamed, M. Saleh, C. Talhi, and S. Zhioua, "Security analysis of wireless java," in *In Proceedings of the 3rd Annual Conference on Privacy, Security and Trust, PST05*, 2005, pp. 1–11.

[76] C. Fournet and A. D. Gordon, "Stack inspection: Theory and variants," *ACM Trans. Program. Lang. Syst.*, vol. 25, pp. 360–399, May 2003.

[77] B. mo Chang, "Static check analysis for java stack inspection," *ACM SIGPLAN Notices*, vol. 41, 2006.

[78] F. Besson, G. Dufay, T. Jensen, and D. Pichardie, "Verifying resource access control on mobile interactive devices," *J. Comput. Secur.*, vol. 18, pp. 971–998, September 2010.

[79] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "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, pp. 944–951.

[80] *Android developers*, Open Handset Alliance, Last accessed: April 2014, available: developer.android.com/index.html.

[81] A. Chaudhuri, "Language-based security on android," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS '09. New York, NY, USA: ACM, 2009, pp. 1–7.

[82] A. Romano and C. Luna, "Descripción y análisis del modelo de seguridad de Android," PEDECIBA Informática, Technical Report RT 13-08, 2013. [Online]. Available: http://www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR1308.pdf

[83] G. Betarte, J. D. Campo, C. D. Luna, and A. Romano, "Verifying android's permission model," in *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. Leucker, C. Rueda, and F. D. Valencia, Eds., vol. 9399. Springer, 2015, pp. 485–504. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-25150-9_28

[84] A. Igarashi and N. Kobayashi, "Resource usage analysis," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 2, pp. 264–313, 2005.

[85] M. Bartoletti, P. Degano, and G. L. Ferrari, "History-based access control with local policies," in *In Proceedings of FOSSACS 2005.* Springer, 2005, pp. 316–332.

[86] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula, "Enforcing resource bounds via static verification of dynamic checks," *ACM Trans. Program. Lang. Syst.*, vol. 29, August 2007.

[87] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "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*, ser. SOCIALCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 944–951. [Online]. Available: http://dx.doi.org/10.1109/SocialCom.2010.140

[88] M. J. May and K. Bhargavan, “Towards unified authorization for android,” in *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ser. ESSoS’13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 42–57. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36563-8_4

[89] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, “Modeling and enhancing android’s permission system,” in *ESORICS*, ser. Lecture Notes in Computer Science, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459. Springer, 2012, pp. 1–18.

[90] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS ’11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[91] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in android,” in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’11. New York, NY, USA: ACM, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[92] A. Armando, G. Costa, and A. Merlo, “Formal modeling and reasoning about the android security framework,” in *TGC’12: 7th International Symposium on Trustworthy Global Computing*, ser. Lecture Notes in Computer Science, C. Palamidessi and M. Ryan, Eds., vol. 8191. Springer Berlin Heidelberg, 2013, pp. 64–81. [Online]. Available: http://www.ai-lab.it/armando/pub/tgc2012.pdf

[93] A. Romano, “Descripción y análisis formal del modelo de seguridad de android,” Master’s thesis, Universidad Nacional de Rosario, Tech. Rep., 2014, available: www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es.

[94] R. P. Goldberg, “Survey of virtual machine research,” *IEEE Computer Magazine*, vol. 7, pp. 34–45, June 1974.

[95] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 164–177.

[96] G. Barthe, G. Betarte, J. Campo, and C. Luna, “Formally verifying isolation and availability in an idealized model of virtualization,” in *Formal Methods*, ser. LNCS, M. Butler and W. Schulte, Eds., vol. 6664. Springer-Verlag, 2011.

[97] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna, “Formally Verified Implementation of an Idealized Model of Virtualization,” in *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Matthes and A. Schubert, Eds., vol. 26. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 45–63. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4625

[98] D. v. Oheimb, V. Lotz, and G. Walter, “Analyzing SLE 88 memory management security using Interacting State Machines,” *International Journal of Information Security*, vol. 4, no. 3, pp. 155–171, 2005.

[99] J. M. Rushby, “Noninterference, Transitivity, and Channel-Control Security Policies,” SRI International, Tech. Rep. CSL-92-02, 1992.

[100] G. Barthe, G. Betarte, J. Campo, and C. Luna, “Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization,” in *IEEE 25th Computer Security Foundations Symposium (CSF)*, 2012, pp. 186–197.

[101] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, “Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones,” in *5th IEEE Consumer and Communications Networking Conference*, 2008.

[102] The VirtualCert project, “Supporting Coq formalization,” 2013, available: www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php.

[103] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, “System-level non-interference for constant-time cryptography,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: ACM, 2014, pp. 1267–1279. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660283

[104] G. Klein, “Operating system verification – an overview,” *Sādhanā*, vol. 34, no. 1, pp. 27–69, 2009.

[105] Z. Shao, “Certified software,” *Commun. ACM*, vol. 53, no. 12, pp. 56–66, 2010.

[106] E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova, “Automated verification of a small hypervisor,” in *Verified Software: Theories, Tools, Experiments*, ser. LNCS, G. Leavens, P. OHearn, and S. Rajamani, Eds. Springer, 2010, vol. 6217, pp. 40–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15057-9_3

[107] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. Paul, “Verifying shadow page table algorithms,” in *Formal Methods in Computer-Aided Design, 10th International Conference (FMCAD’10)*, R. Bloem and N. Sharygina, Eds. Switzerland: IEEE CS, 2010.

[108] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban, “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)*, ser. LNCS, vol. 6217. Edinburgh: Springer, 2010, pp. 71–85. [Online]. Available: http://www-wjp.cs.uni-saarland.de/publikationen/APST10.pdf

[109] D. Elkaduwe, G. Klein, and K. Elphinstone, “Verified protection model of the sel4 microkernel,” in *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE ’08. Springer, 2008, pp. 99–114. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87873-5_11

[110] J. Yang and C. Hawblitzel, “Safe to the last instruction: automated verification of a type-safe operating system,” *Commun. ACM*, vol. 54, no. 12, pp. 123–131, 2011.

[111] H. Tews, T. Weber, E. Poll, and M. v. Eekelen, “Formal Nova interface specification,” Radboud University Nijmegen, Tech. Rep. ICIS–R08011, May 2008, robin deliverable D12.

[112] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, “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*, ser. CCS ’06. NY, USA: ACM, 2006, pp. 346–355. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180448

[113] D. Greve, M. Wilding, and W. M. V. Eet, “A separation kernel formal security policy,” in *Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[114] W. Martin, P. White, F. Taylor, and A. Goldberg, “Formal construction of the mathematically analyzed separation kernel,” in *The Fifteenth IEEE International Conference on Automated Software Engineering*, 2000.

[115] J. Andronick, B. Chetali, and O. Ly, “Using Coq to Verify Java Card Applet Isolation Properties,” in *International Conference on Theorem Proving in Higher Order Logics (TPHOLs’03)*, ser. LNCS, D. A. Basin and B. Wolff, Eds., vol. 2758. Springer-Verlag, 2003, pp. 335–351.

[116] P.-N. Tollitte, D. Delahaye, and C. Dubois, “Producing certified functional code from inductive specifications,” in *CPP*, ser. LNCS, C. Hawblitzel and D. Miller, Eds., vol. 7679. Springer, 2012, pp. 76–91.

[117] S. Berghofer, L. Bulwahn, and F. Haftmann, “Turning inductive into equational specifications,” in *TPHOLs*, ser. LNCS, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674. Springer, 2009, pp. 131–146.

[118] A. Balaa and Y. Bertot, “Fix-point equations for well-founded recursion in type theory,” in *TPHOLs*, ser. Lecture Notes in Computer Science, M. Aagaard and J. Harrison, Eds., vol. 1869. Springer, 2000, pp. 1–16.

[119] G. Barthe, J. Forest, D. Pichardie, and V. Rusu, “Defining and reasoning about recursive functions: A practical tool for the coq proof assistant,” in *FLOPS*, ser. LNCS, M. Hagiya and P. Wadler, Eds., vol. 3945. Springer, 2006, pp. 114–129.

[120] C. Liu and P. Albitz, *DNS and BIND - help for system administrators: covers BIND 9.3 (5. ed.)*. O'Reilly, 2006.

[121] S. Cheung and K. N. Levitt, "A formal-specification based approach for protecting the domain name system," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2000, pp. 25–28.

[122] E. B. Eixarch, G. Betarte, and C. D. Luna, "A formal specification of the dnssec model," *ECEASST*, vol. 48, 2011.

[123] S. Weiler and J. Ihren, "Minimally Covering NSEC Records and DNSSEC On-line Signing," RFC 4470 (Proposed Standard), Internet Engineering Task Force, Apr. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4470.txt