Ernesto Damiani

Claudio Agostino Ardagna

Nabil El Ioini

Open Source Systems Security Certification

Open Source Systems
Security Certification

Open Source Systems Security Certification

by

Ernesto Damiani
University of Milan, Milan
Italy

Claudio Agostino Ardagna
University of Milan, Milan
Italy

Nabil El Ioini
Free University of Bozen/Bolzano
Italy

Springer

Authors:
Ernesto Damiani
Università di Milano
Dipartimento di. Tecnologie dell'Informazione
via Bramante 65
26013 Crema, Italy
damiani@dti.unimi.it

Claudio Agostino Ardagna
Università di Milano
Dipartimento di. Tecnologie
dell'Informazione
via Bramante 65
26013 Crema, Italy
ardagna@dti.unimi.it

Nabil El Ioini
Free University of Bozen/Bolzano
Computer Science Faculty
Piazza Domenicani,3
39100 Bozen/Bolzano, Italy
nabil.elIoini@stud-inf.unibz.it

"There's no reason to treat software any differently from other products. Today Firestone can produce a tire with a single systemic flaw and they're liable, but Microsoft can produce an operating system with multiple systemic flaws discovered per week and not be liable. This makes no sense, and it's the primary reason security is so bad today".

Bruce Schneier, Cryptogram, 16/04/2002

Foreword

I predict that this book on the security certification of trusted open source software will be a best-seller in the software industry as both public – government and private sector software must rely more and more on the development and maintenance of trustworthy open source software. The need for this book material is urgent as adopters of open source solutions may currently lack control on some crucial security-related properties of deployed systems.

In this book Ernesto Damiani, a pioneer in secure information systems, Claudio Agostino Ardagna and Nabil El Ioini focus on certificates dealing with properties of software products. In recent years, the increased demand for certified IT products has driven the open source community to raise questions about the possibilities of certifying open source software. This book is aimed at open source developers and users interested in software security certification, as well as at security experts interested in bundling open source software within certified software products. By certifying individual security features, software suppliers give to their customers the possibility of choosing systems that best match their security needs. Throughout the book, the authors address the following questions:

- Which properties of a software system can be certified?
- How can we trust a certificate to be accurate, and how do we measure such trust?
- What can be done at software design time to establish an acceptable level of trust or, at run-time, to preserve or increase it?

Throughout the book, the authors build a solid case that the general solution to these challenges, more precisely barriers, is an intelligent, automated, and highly customizable software certification process, fully integrated into the open source development and maintenance. The authors focus on techniques for test and model-based certification, and on their application to the security properties of open source systems. The authors are particularly interested in helping the reader study how open source software lends itself to a combination of automatic test selection and formal verification methods. In

this context, the authors have, for example, investigated security evaluation and certification of open source software by looking at two major security evaluation standards as well as providing a partial application to two case studies. The authors' own 2008 research work on 'Mapping Linux security targets to existing test suites' mainly deals with describing and automating open-source certification process, an issue which is also addressed in this book.

The authors of this exciting and important new book find ways for the reader to go far beyond earlier works in trying to break down the above-mentioned long-standing barriers through more general customizable software certification processes. The authors first lay out the historical context by charting first attempts to create a standard for security certification of software, as a basis for later software certification processes, dating back to 1985, with the creation of the TCSEC standard in the U.S., commonly referred to as the Orange Book (USDoD 1985). The authors review how obtaining a TCSEC security certification is a complex process, which involves understanding the software product's architecture and development methodology, analyzing its users' security requirements, identifying and carrying out the corresponding testing.

Then a related attempt at Certification and Accreditation (C&A) implied by standards for Trusted Systems criteria of the U.S. Department of Defense, began in 1985 and led to DoD AIR FORCE SYSTEMS SECURITY INSTRUCTIONS 5024, September 1997. This was a customizable system certification process for DoD software and the trusted systems in which the software was deployed. The focus, however, was not software specific but for systems relying on computer and networked subsystems. It is important to notice that this DoD standard requires checklists for C&A, of the kind the authors mentioned above. As the authors point out it is well-known that DoD Trusted Systems C&A generally requires much more than testing. And a further related work is the DoD Information Technology Security Certification and Accreditation Process (DITSCAP) *https://infosec.navy.mil/Documents/top25.html*.

The authors of this new book articulate how in general DoD C&A customizable system certification process requires a combination of formal reading (static checklists), formal executing (dynamic test cases-checklists) and logical and mathematical proof of correctness, formal methods, just to verify (compliance) that a DoD system is trusted, but not necessarily assurance that it does so.

Within the enterprise systems, both government and private sector, that software must be secure and trusted, systems engineering development and maintenance compliance to the selected security requirements needs verification of correctness methods (a combination of formal reading (static checklists), formal executing (dynamic test cases-checklists) and logical and mathematical proof of correctness, formal methods). But for the same group of professionals, assurance for the selected requirements mandates continued conformance to the systems users' expectations.

Since the 1970's it has been well-known amongst security C&A professionals that there are certain certifying and accrediting security requirements compliance problems that are either sufficiently difficult as to be either not well understood or to be non-solvable. Let us consider a general problem wherein one is given an arbitrary set of security policy requirements P for a system S to be arbitrarily low risk C&A. Then is it possible to construct or prescribe a technology M (combined software, hardware and human resources), generally coined a protection mechanism, that *(1)* enforces a certified P against all threats or intruders to S with respect to P, and *(2)* certifies permission that all required information needed as a result of S for stakeholders will be allowed? Casually, *(1)* is the property of soundness for $M(P)$ and *(2)*, plus *(1)*, is the property of completeness for $M(P)$.

Within the professional literature used by this new book, there have been many characterizations of soundness (along with the absurd 'police state' where no communication may flow between any stakeholders), along with some, but lesser, degrees of completeness characterization. Completeness is much more difficult to characterize and prove than soundness. Both soundness and completeness have been worked using a spectrum of Verification and Validation (V&V) methods, including inspecting, testing and proving of correctness. The higher quality V&V work always tends to try to maximize the reductions of defects or errors through inspecting first, before submitting to follow on work with testing and formal methods (proof of correctness). If this protocol is not spelled out within a prescribed C&A requirements compliance approach, then the approach may possibly be suspect.

And, with so much of today's software grounded in open source software, an intelligent, automated, and highly customizable software certification process, fully integrated into the open source development and maintenance is mandatory within today's global systems where the complexity of ethnographic factors the correct identification and prevention of all threats and intruders becomes arbitrarily difficult, especially when a fair balance between soundness and correctness is required. This, along with an unknown mix of explicit and tacit knowledge embedded in P, S and M, makes guaranteeing C&A results either difficult or impossible. Therefore, clearly understanding the details of security certification of open source software under development and life cycle maintenance must always be viewed as ongoing.

Numerous important references have formed the bricks and mortar of this new book. Consider, for example, Richard Baskerville, "Information systems security design methods: implications for information systems development," *ACM Computing Surveys*, Vol. 25, No. 4, December 1993. This paper, based upon standard practice that includes standards for Trusted Systems criteria of the U.S. Department of Defense, beginning in 1985, specifies most common ways for collecting systems requirements, selecting subsets of priority requirements for further analysis, analyzing such subsets of defects and errors, and determining and assessing risks. The requirements, as units of analysis in this paper, can be general but the authors limited the focus to systems security

requirements, such that defects and errors are interpreted as possible threats based upon risks. *Inspection*, a general term in the paper including both the reading (static inspecting) of requirements, designs and codes and the executing (dynamic inspecting) of codes, is designed based upon both static and dynamic reading (static) checklists and executing (dynamic) checklists, the latter often coined testing (dynamic inspecting) using test cases (check lists when testing).

I very highly recommend this book on the security certification of trusted open source software to the software industry, both public – government and private sector software developers and maintainers, relying more and more on the development and maintenance of trustworthy open source software. The need for this book by Dr. Damiani, Dr. Ardagna and El Ioini is urgent as adopters of open source solutions may currently lack control on some crucial security-related properties of deployed systems.

Fairfax, August 2008   *David C. Rine*
                                                        Professor Emeritus and Consultant
                      Volgenau School of Information Technology and Engineering
                                                                George Mason University

Acknowledgements

Contents

Acronyms

AAM	Authentication and Authorization Model
ACL	Access Control List
BRR	Business Readiness Rating
BSD	Berkeley Software Distribution
CA	Certification Authority
CAPP	Controlled Access Protection Profile
CAS	Central Authentication Service
CC	Common Criteria ISO/IEC 15408
CCHIT	Certification Commission for Healthcare Information Technology
CERT	Carnegie Mellon Computer Emergency Response Team
CFA	Control Flow Analysis
CIM	Centralized Identity Management
CMMI	Capability Maturity Model Integration
COTS	Components-Off-the-Shelf
CTCPEC	Canadian Trusted Computer Product Evaluation Criteria
DAC	Discretionary Access Control
EAL	Evaluation Assurance Level
ETR	Evaluation Technical Report
FIMM	Full Identity Management Model
FM	Federated Model
FOCSE	Framework for OS Critical Systems Evaluation
FSA	Finite State Automaton
FSF	Free Software Foundation
GPL	General Public License
HFT	Health Information Technology
IETF	Internet Engineering Task Force
IPR	Intellectual Property Rights
ISO	International Standard Organization
ISO	International Organization of Standardization
ISO-SD 6	International Standard Organization SC 27 Standing Document 6

ISSO   Information Systems Security Organization
ITSEC   Information Technology Security Evaluation Criteria
JVM   Java Virtual Machine
LLS   Long-Lived software Systems
LOC   Lines Of Code
LSB   Linux Standard Base
LTP   Linux Test Project
LVM   Logical Volume Management
MAC   Mandatory Access Control
MCC   Model-Carrying Code
MIT   Massachusetts Institute of Technology
MPL   Mozilla Public License
NCSC   National Computer Security Center
NSA   National Security Agency
OIN   Open Invention Network
OS   Operating system
OSD   Open Source Definition
OSI   Open Source Initiative
OSMM   Open Source Maturity Model
OSP   Organizational Security Policy
OSS   Open Source Software
OSSTMM   Open Source Security Testing Methodology Manual
OVL   Open Virtual Lab
OWA   Ordered Weighted Average
PCC   Proof-Carrying Code
PGT   Proxy-Granting Ticket
PKI   Public Key Infrastructure
POSIX   Portable Operating System Interface
PP   Protection Profile
PT   Proxy Ticket
QSOS   Qualify and Select Open Source Software
RAV   Risk Assessment Value
RBAC   Role Based Access Control
RPM   Red Hat Package Manage
SAR   Security Assurance Requirement
SFP   Security Functional Policy
SFR   Security Functional Requirement
SLES   Suse Linux Enterprise Server
SMP   Symmetric Multi-Processing
SPICE   Software Process Improvement and Capability dEtermination ISO15504
SQuaRE   Software product Quality Requirements and Evaluation ISO25000
SSH   Secure Shell
SSL   Secure Socket Layer
SSO   Single Sign-On
ST   Security Target

SwA   Software Assurance
TCSEC   Trusted Computer System Evaluation Criteria
TFS   TOE Security Function
TGC   Ticket Granting Cookie
TLS   Transport Layer Security
TOE   Target Of Evaluation
USDoD   U.S. Department of Defense
UTM   Unified Threat Management
VM   Virtual Machine
VMM   Virtual Machine Monitor
VNL   Virtual Networking Labs
XSS   Cross-Site Scripting

Chapter 1
Introduction

Abstract Open source software (OSS) and components are playing an increasingly important role in enterprise ICT infrastructures; also, OSS operating system and database platforms are often bundled into complex hardware-software products. Still, OSS adopters often have only anecdotal evidence of some crucial non-functional properties of open source systems, including security-related ones. In this introduction, we provide some background on software certification and discuss how it can be used to represent and interchange the security properties of open source software.

1.1 Context and motivation

Software is everywhere in our life. It plays a major role in highly critical systems such as health care devices, cars and airplanes, telecommunication equipments and in many other appliances in our workplaces and in our homes. However, in spite of this pervasive presence of software in the modern society, many users express little trust in the correctness and reliability of applications they use every day, be such applications commercial or open source software. Disclaimers coming with some software licenses do not help to build user confidence, as they explicitly proclaim software programs to be "unfit to any specific purpose". Not surprisingly, software users' lack of trust has triggered an increasing public demand for improving the security of mission and safety-critical software. A huge interdisciplinary research effort is being devoted to finding methods for creating security, safety and dependability *certificates* to be delivered to users together with software systems.[1] This work is distinct but related to the ongoing one [14] to establish consistent regulation to ensure that software in critical systems meets minimum safety,

---

[1] Generally speaking, software certification may concern people (e.g., developers or testers) skills, the development process or the software product itself. Here we focus on the certification of software products.

security and reliability standards. In parallel to these research efforts, international governmental agencies and standards bodies have been busy devising regulations and guidelines on the development and deployment of certified software products. Software suppliers are also becoming interested in certification since under certain contractual obligations they may be liable for security breaches resulting in the loss of confidential information, as well as in business interruption. Software purchasers increasingly refuse full exclusion of liability in software supply contracts. An interesting situation occurs when software suppliers fail to address security threats for which countermeasures are known, especially when these threats have been explicitly mentioned by the purchasers. In some cases, software suppliers may be liable for all damage suffered as a result of the security breach; claims may even be made to the money needed to put the software purchaser in the same situation had the failure not occurred. Independent certification of a software product's security features can provide a way to reduce such liability, e.g. limiting insurance costs. Of course, different legal systems have different notions of liability, and the extent of any liability reduction due to software certification will depend on many factors, including where the organization using certified software is based. From the software purchaser's point of view, the certificate provides a useful independent appraisal of a software product's properties. From the software supplier's point of view, getting a prestigious certificate can be a powerful marketing tool, while failing to do so can be an early warning of something being wrong with the product. In this book, we focus on the technological and scientific aspects of generating certificates dealing with properties of open software products. The reader interested in the the legal aspects of liability issues involving software suppliers is referred to the seminal paper [4].

This book is aimed at open source developers and users interested in software security certification, as well as at security experts interested in bundling open source software within certified software products. Rather that providing ready-to-go recipes (which would be inevitably obsolete by the time the book reaches its audience), we will try to provide the reader with the basic notions and ideas needed to deal with open source certification. Throughout the book, we shall address the following questions:

- Which properties of a software system can be certified?
- How can we trust a certificate to be accurate, and how do we measure such trust?
- What can be done at software design time to establish an acceptable level of trust or, at run-time, to preserve or increase it?

Before trying to answer these questions, however, we need to agree on some basic terminology [5]. Let us start by reviewing some basic terms related to certification taken from the International Standard Organization (ISO) SC 27 Standing Document 6 (SD 6), a document well-known to security experts

worldwide as the *Glossary of IT Security Terminology*. ISO SD 6 provides the following definitions:

- *Accreditation*: Procedure by which an authoritative body gives formal recognition that a body or person is competent to carry out specific tasks (see ISO/IEC Guide 2), such as certification or evaluation.
- *Certification authority*: an entity trusted by one or more users to create and assign certificates.
- *Certification*: Procedure by which a third party gives written assurance that a product, process or service conforms to a specified requirement (see ISO/IEC Guide 2).
- *Certificate*: Declaration by a certification authority confirming that a statement about a software product is valid. Usually issued on the basis of the outcome of an evaluation.[2]
- *Evaluation*: Systematic assessment (e.g., by means of tests) of the extent to which a software product is capable of fulfilling a set of requirements.
- *Evaluation body* : Accredited lab carrying out evaluations. The evaluation outcome is a pre-requisite for issuing a certificate on the part of the certification authority.

Although we shall do our best to adhere to this standard terminology throughout the book, the above terms can be used in many practical situations, largely different from each other. Therefore, their meaning can depend on the context. An important aspect to be clarified, when software certification is discussed, is the distinction between *model-based* and *test-based* certification of software properties.

- *Model-based certificates* are formal proofs that an abstract model (e.g., a set of logic formulas, or a formal computational model such as a finite state automaton) representing a software system holds a given property. The model to be certified can be provided as a by-product of the software design process, or be reverse-engineered from the software code.
- *Test-based certificates* are evidence-based proofs that a test carried out on the software has given a certain result, which in turn shows (perhaps with a certain level of uncertainty) that a given property holds for that software. In particular, test-based certification of security-related properties is a complex process, identifying a set of high-level security properties and linking them to a suitable set of white- and black-box software tests.

Of course, other certification techniques do exist, including system simulation, code reviews and human sign offs; but they are less widespread than the two major approaches outlined above.

---

[2] Of course, the authority issuing the certificate is supposed to be operating in accordance with a well-specified standard procedure, such as ISO Guide 58, a.k.a., *Calibration and testing laboratory accreditation systems*.

Today, test-based security certification is mostly applied when customers need to have a reliable quantitative measure of the security level of their software products. By certifying individual security features, software suppliers give to their customers the possibility of choosing systems that best match their security needs. A wide range of (test-)certified software products exist, including operating systems, firewalls, and even smart cards. Only very few of those certified products are open source systems, because certification of open source is still a relatively new topic, and there are several challenges specifically related to open source certification. For instance, guaranteeing high reliability is difficult when a combination of diverse development techniques is used, as it is often the case for open source communities. In the remainder of the book, we (as well as, hopefully, the reader) are interested in studying if and how open source software lends itself to a combination of automatic test selection and formal verification methods. Also, we will discuss some open problems of the certification process. A major one is cost: certification effort can be huge, and it must be correctly estimated and kept under control. This can be done by reducing certification and re-certification times, as well as by dynamically linking artifacts to certificates.

1.2 Software certification

As we anticipated above, software certification relies on a wide range of formal and semi-formal techniques dealing with verification of software systems' reliability, safety and security properties. In principle, a certificate should contain all information necessary for an independent assessment of all properties claimed for a software artifact. Obviously, then, the exact content of a certificate will depend on the software artifact it is about and the specific property it claims to hold for that artifact. Certifiable artifacts include implementation-level ones (such as entire software systems, individual software components, or code fragments), but also analysis and design-level artifacts such as requirements, UML diagrams, component interfaces, test suites, individual test cases and others. A certificate has to represent all entities involved in the certification process, that is, *(i)* the certification authority making the assertion, *(ii)* the software artifact being certified and *(iii)* the property being asserted. Certificates for artifacts belonging to the same software system are not an unstructured collection; rather, they exhibit some structure according to the software system's architecture and granularity. If the software system under certification can be decomposed into a number of subsystems, and in turn each subsystem is composed of a number of components, then the certificate issued for the entire system depends hierarchically on the certificates of the subsystems and on the certificates of all involved components. This *certificate hierarchy* can be used for auditing and incremental re-certification. The certification authority can determine which certificates need to be inspected,

recomputed, or revalidated after an artifact or a certificate has been (or would be) modified.

Of course, much of the the value of a certificate is in the eye of the beholder - i.e., of the purchaser. What precisely this value is will largely depend on the customer's specific perspective and on its trust in the authority issuing the certificate.

1.2.1 Certification vs. standardization

The notion of certification is strictly intertwined with the one of standardization. Historically, open standards have been the result of an agreement between technology suppliers, customers and regulatory authorities at a national or international level. The emergence of open standards has been crucial to ensure interoperability among different brands of software products or services; the rapid development of the global Internet is probably the most studied example of a growth and innovation process enabled by the "bottom-up" open standardization process carried out within the *Internet Engineering Task Force* (IETF). Certification bodies operate in the middle between the requirements of the users and the industrial needs. In that sense, certificates can be seen as trusted information given to potential customers about products and services available on the market.

Certificates are closely related to open standards inasmuch they give information about a software product's properties to the potential customer and to the general public, facilitating a comparison among competing software products or services. Certified properties correspond to a set of requirements, which can coincide to those defining an open standard. For instance, the *Common Criteria* (see below, as well as the detailed discussion in Chapter 3) is a certification scheme where the security level of a software product is assessed according to a set of requirements defined in the international standard ISO/IEC 15408.

1.2.2 Certification authorities

In principle, any private or public body can issue certificates. In practice, certification authorities tend do be endorsed by governments [5]. For instance, the European Standard EN 45011 specifies requirements for certification authorities to be accredited by governments of EU member states. EN 45011 requirements on certification authorities mainly concerned how the no-profit

nature of the certification activity.[3] [4] A major task of a certification authority is the generation and interchange of (signed) certificates. In order to obtain a certificate, a software product has to go through an evaluation process, where a third party checks whether the product complies with a given set of requirements. More specifically, given a software artifact and a set of claimed properties the certification authority will attempt to execute tests or perform model-based reasoning in order to generate and sign a certificate, involving human agents when necessary. Also, the certification authority will carry out intelligent re-certification when some changes take place in the software artifacts to be certified. When computing re-certification, existing (sub-)certificates will be reused where possible, especially where *part-of* component hierarchies are available. Additional services managed by a certification authority may include:

- *Certificates revocation.* An individual certification step is no longer valid (e.g., a bug has been discovered in a test suite), leading to the revocation of all certificates which depend on it.
- *Certification history.* A certification authority should be able to provide a complete certification history with full information about all procedures followed, so that comprehensive audits can be carried out.

In general, creating and promoting a certification scheme and providing the generation and revocation services are distinct activities, which require different skills. Therefore, certification authorities can delegate suitable *evaluation bodies* who check compliance and provide the certification authority with evidence, based on which the certification authority issues (or does not issue) the certificate. Certification authorities have also some informal goals to achieve, including *(i)* defining the process (and resources) needed for certification and *(ii)* improving the market awareness on the meaning and scope of the certification program.

1.3 Software security certification

Historically, software certification has been more closely related to generic quality assurance than to security issues [8, 14]. Again using ISO terminology, quality is "the degree to which a set of inherent characteristics fulfills requirements" (ISO9001:2000), and the step of bringing in someone to certify the extent to which requirements are satisfied seems quite straightforward. However, some requirements are easier to specify than others, and tracing software quality requires an accurate definition of evaluation characteristics.

---

[3] In some cases, a government directly empowers an organization.

[4] In Germany, BSI in Germany is authorized by law to issue certificates and no accreditation is required for the BSI Certification Body. A similar situation occurs in France with DCSSI and its evaluation center (CESTI).

Quality certifications dealing with software products and the software development process have been investigated since the early days of Software Engineering. Both McCall's and Boehm's historical software quality models [3, 9] date back to the late Seventies, and significant contributions to the formalization of the general notion of software quality were given by industrial research already in the Eighties (Boeing Aerospace Company published their Quality Model for Software in 1985). Standardization of software quality requirements started in 1991, when the first *International Standard for Software Quality* was brought out as ISO9126 [12]; in 1994 the ISO9001 standard was issued. In recent years, increased understanding of software product and process quality requirements brought about the *Capability Maturity Model Integration* (CMMI) (CMMI version 1.0 was announced in 1997). Finally, in 1998 the *Software Process Improvement and Capability dEtermination* (SPICE) was issued as ISO15504 standard. With these building blocks firmly in place, work on software quality requirements has continued to this day; the first *Software product Quality Requirements and Evaluation* (SQuaRE) document was published in 2005 as ISO25000 and revised in 2007 [7, 13].

Like quality, the problem of software security is a time-honored one, and can be traced back to the introduction of computers themselves. However, in the early days of software engineering security was mainly an afterthought. In the last few years, the fast pace of the technological evolution of computing devices in terms of networking, speed and computing power, has dramatically affected the way we look at software security. Software security has become a great concern for many decision makers, who want to make sure that the software products they buy possess security features needed to counteract all known threats. Security-related certification is now one of the most important and challenging branch of software certification, especially for component-based architectures [1] where software coming from different sources must be integrated.

However, software security does not affect software systems only once they have been deployed; rather, it deals with the entire software life cycle, from its conception and development till its usage and maintenance. For our purposes, a key aspect is the damage that security threats can do to the perceived quality of software products. For many years, it has been a common mistake to underestimate the negative effects of threats on the acceptance of software products. In the last few years, interest in understanding and preventing security threats has been rising. According to Gary McGraw [11], the three pillars of software security are *applied risk management*, *software security best practices*, and *knowledge sharing*. In terms of applying risk management to software purchases, a major problem is that many software suppliers still consider security to be an additional feature, rather than a built-in property of their products. In case of a security failure, many software suppliers still hasten to release patches and quick fixes to the problem [10], rather than considering which security properties their system failed to deliver and why.

1.3.1 The state of the art

As we shall see throughout the book, obtaining independent security certification of software products is becoming the preferred choice of many software suppliers, especially those dealing with security-related software platforms, who need to reduce their liability and prove the effectiveness and the robustness of their products. However, a security certification is an expensive process in terms of time and effort. The first attempt to create a standard for security certification of software dates back to 1985, with the creation of the TCSEC standard in the U.S., commonly referred to as the *Orange Book* (USDoD 1985). Obtaining a TCSEC security certification is a complex process, which involves understanding the software product's architecture and development methodology, analyzing its users' security requirements, identifying and carrying out the corresponding testing. In the Nineties, the need for software security certification also emerged outside the U.S., leading to the creation of local security certifications, such as, ITSEC in Europe (ITSEC 1991) and CTCPEC in Canada (CSE 1993).

Since these national certifications were (and still are) totally independent from each other, the cost of certifying a software system at an international level has remained very high for a long time. Today, there is a lot of interest in defining security benchmarks and platforms to automate fully or partially the security certification process, speeding up products adoption and commercialization.

Security certifications may reduce insurance costs, limit liability and even pay off commercially: for instance, some governments afford advantages to CMMI and ISO 9001-certified companies, and the same is happening with security certifications. As we will see in the next chapters, cost is one of the key factors that have led to the creation of international standards for software security certification, leading to technical configuration standards for operating systems, network devices and applications which correspond to a set of desired security properties. We shall deal with these standard benchmarks throughout the book. Some of them are user-originated and reflect the consensus of expert users worldwide, while others have been defined by suppliers to meet the security requirements of specific market niches.

Correspondingly, a number of testing platforms and tools have been developed in order to manage the level of compliance of computer configurations with the properties corresponding to the technical settings defined in the benchmarks. Many problems, however, remain to be solved. Fast obsolescence of security certifications (see Chapter 9) is a major one: in order to prevent it, long-term monitoring system security is needed to ensure that benchmark security configurations remain in place over time and security properties continue to hold.

1.3.2 Changing scenarios

Security certification process must be able to follow fast and agile software development processes, including the ones relying on *Commercial Off-The-Shelf* (COTS) components. This is not a trivial task: if COTS products change every few months and the certification process requires a year, certified products may become obsolete by the time certificates are available. Often, users have to decide whether to adopt a non-certified recent release of a system, or settle for a previous version which completed the certification process.[5] In principle, many researchers agree that security certification should follow the embedded software model; that is, it should deal with software products and the environment in which they run at the same time [16]. Alternatively, a *divide-et-impera* approach would decouple the certification of the execution environment from the one of a given software product in the context of that environment.

Another major challenge is the rapidly changing landscape of security threats, which makes it difficult to maintain the set of desirable properties against which products must be certified. The validity of a security certificate over time is essentially associated with the software product's ability to repeal new attacks. While a general technique for security certificates' *recomputation* or *continuous monitoring* is still an open challenge, some certification authorities do offer a certified product monitoring activity (Chapter 7) which consists of regularly recomputing the certificate including properties considered desirable in order to repeal or prevent emerging threats. As long as the evaluation body in charge of this monitoring activity repeats this procedure each time a new threat emerges, the certificate is considered to be monitored.

1.4 Certifying Open source

An increasing number of software programs and applications are designated with the term *Open-Source Software* (OSS). We shall not try to define the term precisely in this introductory chapter, as those who employ it to describe a software product may refer to one or more of a number of diverse properties related to sourcing, licensing and development (see Chapter 5). Indeed, the term "open source" itself is prone to misuse; being descriptive, it cannot be protected as a trademark, so its actual meaning in terms of licensing needs has to be carefully specified in any given context. However, clearly specifying licensing issues is just a first step toward the needs of companies wishing to

---

[5] Today, no general methodology is available for computing incremental, reusable certificates. Research aimed at reusing the results of the previous evaluation may reduce considerably the evaluation costs of new versions of certified products.

bring OSS as a first-class citizen into critical products and appliances. Acknowledged benefits of open-source software include lower development and maintenance costs, as well as timely evolution to meet new requirements. OSS adoption for complex mission-critical applications, such as telecommunications and embedded systems, is still quantitatively and qualitatively less successful than it could be, even in presence of detailed adoption guidelines. Relatively few companies do not use open source products and technologies at all, but project managers working on complex systems still perceive barriers that limit their use, including support and license topics.

While complex software systems like mobile network services go through highly structured production processes, including stringent certification of performance, security and safety properties, OSS projects are functionality-driven, collaborative, rapidly updated and rapidly released (see Chapter 5), and pushed by volunteers working with a diversity of interests, skills, and hardware sets. Today, OSS integration into complex products is becoming increasingly dynamic, relying on mechanisms such as object wrapping/delegation and stubbing/remote invocation. This evolution, partly due to *Intellectual Property Rights* (IPR) concerns, has raised a number of issues about compositional properties of complex systems dynamically integrating (vs. statically bundling) OSS components and services. A major issue companies producing complex systems and services face in incorporating OSS into their products is lack of reliable knowledge on open source components and their impact on the overall properties of systems they are dynamically integrated in. Some OSS products are only useful as a source for developing requirements for internal software development, others can provide good components for development projects and, finally, some OSS can be put in enterprise production environments without hesitation. Also, OSS properties are asserted by different sources at different levels of trustworthiness, and some of them are left for the final adopter to discover. Many OSS products lack of stable and reliable information about functionalities and about non-functional properties crucial for their dynamic integration in complex systems, like security, dependability, safety, and code maturity. Often, when companies have to decide which OSS components to integrate in their products and how to do it, there is little time for long evaluations and for scanning a large technology landscape; even when detailed information is available, it is difficult to assemble the specific set of information needed to assess a specific (e.g., dynamic) integration technique. Two major sources of problems are:

- *Unsuccessful proof-of-concept*, due to the fact that a product including an OSS solution does not meet non-functional requirements about performance, code quality or security. Such failure can be systematic or happen only occasionally (e.g., under specific load conditions). For instance, a high-end network device can invoke the services of an open source TCP/IP component that is not certified w.r.t. certain ICMP malformed packet vulnerabilities. When the OSS component is used to receive

ICMP messages, the whole product will not meet the standards required by certification, even if the software component is not faulty.

- *Development of faulty, solution-driven requirements for final products*, that is, unwanted influence of "what is already available as OSS" over the company's product specifications. For instance, designing a network switch based on Linux standard support for packet switching may generate an unexpected performance ceiling, which can be overcome in peak traffic conditions by dynamically integrating switching hardware/software based on high-performance multiprocessor.

The impact of these issues can be effectively reduced by the use of virtualized environments for testing and designing the properties of new solutions dynamically including OSS, before inserting them in production systems. Virtualization techniques allow creating isolated or interconnected systems that totally simulate real systems. Virtual Networking Labs (VNL) has been proposed and developed exploiting virtualization frameworks [6] in the field of e-Learning, web-based services provision, and software testing. We shall deal with the role of virtualization in certifying software products in Chapter 8. Another important aspect is the one related to OSS IPR. Some initial steps have been made: for instance, the Open Invention Network (OIN) acquires patents and licenses them royalty-free to companies that agree not to enforce their own patents against the Linux operating system and certain Linux-related applications.

Today, OSS adopters and the open source community are raising questions about the possibilities of certifying open source software. On the one hand, as we said above, open source software development by large-scale communities exhibits a number of features that make the certification process more awkward than it is for traditionally developed proprietary software (Chapter 5). For instance, not all open source communities provide reliable test data, and even when they do, these data are seldom suitable for test-based certification needs. Also, only a small number of definitions of desirable security properties which can be certified via tests have been published for open source systems.

On the other hand, having a security assurance process leading to certification is in the best interests of the open source community. Indeed, the mere fact that a software product is open source does not limit its supplier's liability. Besides developers, potentially liable parties include systems integrators and software maintenance companies. This situation has fostered several attempts to understand the requirements of existing security certifications and apply them to open source software .

The IBM Linux® Technology Center has provided many contributions to the development community on the topic of Linux security. In a seminal paper by the IBM Linux group [15], the authors describe how they obtained Common Criteria security certification for Linux, the first open-source product to receive such certification. The IBM group has also been extending the range of Linux security properties to be certified. In March 2007, the same group presented a work [17] that explores the evolution, rationale, and development

of features to meet new profiles related to advanced access control models. We
have investigated security evaluation and certification of open source software
by looking at two major security evaluation standards as well as providing
a partial application to two case studies. Our own research work [2] mainly
deals with describing and automating open-source certification process, an
issue which is also addressed in this book.

1.5 Conclusions

Customer demand for independent security certification of software products
is increasing, as both software users and suppliers are usually not able to carry
out evaluations themselves. Also, independent security certificates may have
liability reduction effect and a commercial impact, especially for suppliers
taking part to international tenders of e-government products (e.g., smart
cards). However, a less expensive re-certification procedure is needed to make
certification affordable for small and medium enterprises. As we shall discuss
later in the book (see Chapter 7), "lightweight" certification schemes are also
gaining acceptance in the area of security-related software products such as
firewall and intrusion detection systems.

Open source security certification is a crucial step toward inserting OSS
integration in the critical path of the development of complex network and
hardware/software architectures. Providing techniques for compositional cer-
tification of security, dependability and safety properties across a number of
dynamic and static integration techniques, can pave the way to an exponen-
tial diffusion of open source paradigm. Well-designed certification schemes
can also help to configure safety-critical software products or services in or-
der to comply with external regulations.

References

1. A. Alvaro and E.S. de Almeida S.R. de Lemos Meira. Software component certi-
   fication: A survey. In *Proc. of 31st EUROMICRO Conference on Software En-
   gineering and Advanced Applications*, Porto, Portugal, August-September 2005.
2. C.A. Ardagna, E. Damiani, N. El Ioini, F. Frati, P. Giovannini, and R. Tchokpon.
   Mapping linux security targets to existing test suites. In *Proc. of The Fourth
   IFIP International Conference on Open Source Systems*, Milan, Italy, September
   2008.
3. B. Boehm, J. Brown, M. Lipow, and G. MacCleod. *Characteristics of Software
   Quality.* NY, American Elsevier, 1978.
4. D. Callaghan and C. O'Sullivan. Who should bear the cost of software bugs?
   Computer Law & Security Report, 21(1):56–60, February 2005.
5. C. Casper and A. Esterle. Information security certifications. a primer: Products,
   people, processes. Technical report, December 2007.

6. E. Damiani, M. Anisetti, V. Bellandi, A. Colombo, M. Cremonini, F. Frati, J.T. Hounsou, and D. Rebeccani. Learning computer networking on open para-virtual labs. *IEEE Transaction On Education*, 50:302–311, November 2007.

7. Internal Document ISO/IEC JTC1/SC7. *ISO25010-CD JTC1/SC7, Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE)*, 2007. State: CD Commision Draft.

8. B. Kitchenham and S.L. Pfleeger. Software quality: the elusive target. *IEEE Software*, 13(1):12–21, Jan 1996.

9. J. McCall, P. Richards, and G. Walters. Factors in software quality, vol 1,2, & 3. Technical report, November 1977.

10. G. McGraw. Managing software security risks. *IEEE Computer*, 35(4):99–101, March 2002.

11. G. McGraw. Software security: Building security in. In *Proc of the 17th International Symposium on Software Reliability Engineering (ISSRE 2006)*, November 2006.

12. International Standardization Organization. *ISO9126:1991 JTC1/SC7, Information Technology Software Product Quality*, 1991.

13. International Standardization Organization. *ISO25000:2005 JTC1/SC7, Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE)*, 2005.

14. D. Parnas. Certification of software: What, how, and how confidently. In *International Workshop on Software Certification*, Hamilton, ON, Canada, August 2006. Invited Talk.

15. K.S. Shankar and H. Kurth. Certifying open source: The linux experience. *IEEE Security & Privacy*, 2(6):28–33, November-December 2004.

16. J. Voas and K. Miller. Software certification services: Encouraging trust and reasonable expectations. *IT Professional*, 8:39–44, September-October 2006.

17. G. Wilson, K. Weidner, and L. Salem. Extending linux for multi-level security. In *SELinux Symposium*, 2007.

Chapter 2
Basic Notions on Access Control

Abstract Security certifications deal with security-related properties and features of software products. In this chapter we introduce some basic concepts on software systems features related to security and dependability. Such features are often designated with the acronym *CIA*: they include *Confidentiality* features, ensuring that only authorized users can read data managed by the software system, *Integrity* features, ensuring that only authorized users can can modify the software system's resources, and *Availability* features, ensuring that only authorized users can use the software system. In this chapter we shall introduce some concepts underlying some software security features which are an important target for the software certification process. Specifically, we shall focus on access control subsystems, which are among the most widespread security features added to operating systems, middleware and application software platforms. Such subsystems will be among the targets of certification discussed in this book.

2.1 Introduction

The software security field is a relatively recent one;[1] but today software security is so wide notion that has come to mean different things to different people. For our purposes, we shall loosely define it as the set of techniques aimed at designing, implementing and configuring software so that it will function as intended, even under attack.

Security features, such as access control systems [6], are added to software products in order to enforce some desired security properties. Although the notion of specific security-related features and subsystems is an important one, it is much easier for such features to protect a software product that

---

[1] According to Gary McGraw [4], academic degrees on software security appeared as recently as 2001. Our own online degree course on Systems and Network Security at the University of Milan (http://cdlonline.unimi.it) was started in 2003.

is fault-free than one internally riddled with faults which may cause failures which prevent security features from doing their job.

Our notion of software security is therefore twofold: firstly, software security is about protecting "plain" software systems by adding to them a number of specific security features; secondly, software security is about programming techniques for developing secure software systems, designing and coding them to withstand (rather than prevent) attacks. The former notion of software security follows naturally from a system-centric approach, where access to software systems "from outside" must be controlled and regulated. The second notion, instead, relies on the developers' insight and focuses on methods for identifying and correcting dangerous faults which may be present in the software itself.

Most modern software systems do include some security features, but adding such features does not guarantee security *per se*. Indeed, software security is a system-wide issue that includes both adding security features (such as an access control facility) and achieving security-by-coding (e.g., via robust coding techniques that make attacks more difficult).

We can further clarify this distinction via a simple example. Let us assume that a web server has a buffer overflow fault [2], and that we want to prevent a remote attacker from overflowing the buffer by sending to the server an oversize HTTP `GET` request. A way to prevent this buffer overflow attack could be adding a software feature to the web server, a `monitor` function that observes HTTP requests as they arrive over port 80, and drops them if they are bigger than a pre-set threshold. Another way to achieve the same result consists in fixing the web server source code to eliminate the buffer overflow fault altogether. Clearly, the latter approach can only be adopted when the server's source code is available and its operation is well understood.

In most organizations, software security is managed by system administrators who set up and maintain security features such as access control and intrusion detection systems, firewalls and perimeter defenses, as well as antivirus engines. Usually, these system administrators are not programmers, and their approach is more oriented to adding security features to protect a faulty software than to correcting the faults in the software that allow attackers to take advantage of it.

Furthermore, software development projects (like any other type of project) have schedules and deadlines to respect for each step of the development process. The pressure put on development teams make many developers care for little else than making the software work [2], neglecting verification of security properties.

As we shall see, security certifications try to deal simultaneously with both aspects of software security: they certify the presence of some security features as well as the outcome of testing their functionality. From this point of view security is an emergent property of the entire software system rather than

---

[2] The reader who is unaware of what a buffer overflow fault is can skip this example and come back to it after reading Chapter 4.

a set of individual requirements. This is an important reason why software security must be part of a full lifecycle-based approach to software development. In this chapter we shall introduce some concepts underlying software security features which are an important target for the security certification process. Specifically, we shall focus on access control subsystems, which are among the most important security features added to operating systems, middleware and application software platforms. Such subsystems will be among the targets of certification we shall discuss in the next chapters.

2.2 Access Control

Access Control (AC) is the ability to allow or deny the use of resources. In other words, access control decides who *subject* is authorized to perform certain operations on a given *object* and who is not. The notion of controlling access is independent of the nature of objects and subjects; objects can be physical resources (such as a conference room, to which only registered participants should be admitted) or digital ones (for example, a private image file on a computer, which only certain users should be able to display), while subjects can be human users or software entities.

Generally speaking, computer users are subject to access control from the moment they turn on their machines, even if they do not realize it. On a computer system, electronic credentials are often used to identify subjects, and the access control system grants or denies access based on the credential presented. To prevent credentials being shared or passed around, a two-factor authentication can be used. Where a second factor (besides the credentials) is needed for access. The second factor can be a PIN, or a biometric input. Often the factors to be made available by a subject for gaining access to an object are described as

- something you have, such as a credential,
- something you know, e.g. a secret PIN, or
- something you are, typically a fingerprint/eye scan or another biometric input.

Access control techniques are sometimes categorized as either *discretionary* or *non-discretionary*. The three most widely recognized models are Discretionary Access Control (DAC) , Mandatory Access Control (MAC), and Role Based Access Control (RBAC). MAC and RBAC are both non-discretionary.

2.2.1 Discretionary Access Control

The Trusted Computer System Evaluation Criteria (TCSEC) [5] defines the
Discretionary Access Control (DAC) model "a means of restricting access to
objects based on the identity of subjects and/or groups to which they belong.
DAC is *discretionary* in the sense that a subject with a certain access per-
mission is capable of passing on that permission (perhaps indirectly) to any
other subject (unless explicitly restrained from doing so by mandatory access
control)". The Orange Book also states that under DAC "an individual user,
or program operating on the user's behalf, is allowed to specify explicitly
the types of access other users (or programs executing on their behalf) may
have to information under the user's control". DAC provides a very impor-
tant security property: subjects to have full control over their own objects.
Subjects can manipulate their objects in a variety of ways based on their
authorizations.

Authorization determines which *actions* a subject can do on the system;
the semantics of actions depends on the nature of the objects involved. For in-
stance, within an operating systems platform, actions are variations or exten-
sions of three basic types of access: Read (R), where the subject can read file
contents or list directory contents, Write (W), where the subject can change
the name or the contents of a file or directory, and Execute (X), where, if
the object is a program, the subject can cause it to be run. As we shall
see below, these permissions are implemented differently in systems based
on Discretionary Access Control (DAC) than in Mandatory Access Control
(MAC).

Throughout the book, we shall repeatedly mention DAC, because many
operating systems today, including Unix and Unix look-alikes like Linux, use
DAC as part of their file protection mechanisms. DAC has also been adopted
for controlling access to networking devices such as routers and switches.
DAC's high flexibility enables users to have full control to specify the privi-
leges assigned to each object.

A widely used framework for implementing DAC is the Matrix access
model. Formalized by Harisson, Ruzzo, and Ullman (HRU) [3], the Matrix
access model provides a conceptual abstraction which specifies the access
rights that each subject s has over each object o. The Matrix access model
is composed of rows representing the subjects, and columns representing the
objects. Each intersection cell between a subject s and an object o represents
the access rights for s over o. In other words, the access rights specified by
the cell ensure what type of access a subject has over an object (e.g. write ac-
cess, read access, execution access). An example of an access matrix is shown
in Figure 2.1. In this example, the access rights that are considered are (R)
read, (W) write, and (X) execute. An empty cell means that a specific subject
s has no right on a specific object o, conversely if a cell is assigned all the
access rights, it means that a specific subject s has full control over a specific
object o.

	O1	O2	O3	On
S1	R W X		R	W
S2	W	R W	R W X	
S3	R W X	R W	R	W
Sn		W	R X	R W

Fig. 2.1: Access matrix

2.2.1.1 Access Control Lists

Access Control Lists (ACLs) are data structures widely used to implement both discretionary and mandatory access control models. Often, operating systems implement DAC by maintaining lists of access permissions to be attached to system resources. Each object's access list includes the subjects allowed to access the object and the actions they are allowed to execute on the object. For example, the entry (`Bob`, `read`) on the ACL for a file `foo.dat` gives to subject Bob permission to read the file. In an ACL-based implementation of DAC, when a subject tries to execute an action on an object, the system first checks the object's list to see whether it includes an entry authorizing such action. if not, the action is denied.[3]

ACLs have been implemented in different ways in various operating systems, although a partial standardization was attempted in the (later withdrawn) POSIX security drafts .1e and .2c, still known as POSIX ACLs. In practice, ACLs can be visualized as tables, containing entries that specify individual subjects or group permissions to access system objects, such as processes, or a file. These entries are called `access control entries` (ACEs) within Microsoft Windows, Linux and Mac OS X operating systems.

2.2.2 *Mandatory Access Control*

While in DAC subjects are in charge of setting and passing around access rights, the *Mandatory Access Control* (MAC) model enforces access control based on rules defined by a central authority [6]. This feature makes MAC more suitable to setup enterprise-wide security policies spanning entire organizations. MAC has been defined by the TCSEC [5] as "a means of restricting

---

[3] A key issue in the efficient implementation of ACLs is how they are represented, indexed and modified. We will not deal with this issue in detail here, even if such implementation details may well be a target for a software security certification.

access to objects based on the *sensitivity* (as represented by a label) of the information contained in the objects and the formal authorization (i.e., *clearance*) of subjects to access information of such sensitivity".

In MAC access control system all subjects and objects are assigned different sensitivity labels to prevent subjects from accessing unauthorized information. The sensitivity labels assigned to the subjects indicate their level of trust, whereas the sensitivity labels assigned to the objects indicate the security clearance a subject needs to have acquired to access them. Generally speaking, for a subject to be able to access an object, its sensitivity level must be at least equal or higher than the objects' sensitivity level.

One of the most common mandatory policies is *Multilevel Security* (MLS), which introduces the notion of classification of subjects and objects and uses a *classification lattice*. Each class of the lattice is composed of two entities, namely:

- Security Level (L): which a hierarchical set of elements representing level of data sensitivity

  Examples:

  Top Secret (TS), Secret (S), Confidential (C), Unclassified (U)

  $$TS > S > C > U \tag{2.1}$$

  Essential (E), Significant (S), Not important (NI)

  $$E > S > NI \tag{2.2}$$

- Categories (C): a set of non-hierarchical elements that represent the different areas within the system

  Example:

  Accounting, Administrative, Management etc.

By combining the elements of the two components one obtains a partial order operator on security classes, traditionally called `dominates`. The `dominates` operator represents the relationship between each pair (L, C) with the rest of the pairs. The `dominates` relation (whose notation is the sign $\succeq$) is defined as follows:

$$(L1, C1) \succeq (L1, C2) \iff L1 \geq L2 \land C1 \supseteq C1 \tag{2.3}$$

To better understand the relation among the different classes defined in Eq. 2.3, let's examine the structure created by the `dominates` relation $\succeq$. It is a lattice, often called *classification lattice*, which combines together the

Security Classes (SC) and the relations between them. Being a lattice, the classification structure satisfies the following properties:

Reflexivity of $\succeq$ $\forall x \in SC : x \succeq x$
Transitivity of $\succeq$ $\forall x, y, z \in SC : x \succeq y, y \succeq z \Longrightarrow x \succeq z$
Antisymmetry of $\succeq$ $\forall x, y \in SC : x \succeq y, y \succeq x \Longrightarrow x = y$
Least upper bound (lub) $\forall x, y \in SC : \exists! z \in SC$

- $z \succeq x$ and $y \succeq z$
- $\forall t \in SC : t \succeq x$ and $t \succeq y \Longrightarrow t \succeq z$

Greatest lower bound (glb) $\forall x, y \in SC : \exists! z \in SC$

- $x \succeq z$ and $y \succeq z$
- $\forall t \in SC : x \succeq t$ and $y \succeq t \Longrightarrow z \succeq t$

Figure 2.2 shows an example of a classification lattice with two security levels: Classified (C), Unclassified (U), and two categories: Management and Finance. By looking closely to the figure we can notice that the sensitivity and the importance of the information increases as we move upward in the lattice. For instance, starting from the bottom of the lattice, we have Unclassified information U that does not belong to any category, and by moving along any of the two side lines we find unclassified information. However, this time the information belongs to a specific category, which gives this information more importance. If we move upward in the lattice we get to more important classified information.
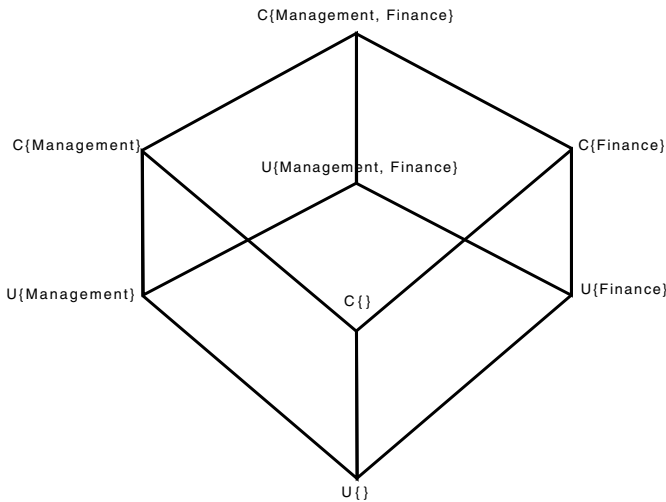
Fig. 2.2: Classification Lattice

Two well-known security models that use a mandatory access control approach are the *Bell-LaPadula*, and the *Biba* model.

2.2.2.1 Bell-LaPadula Model

The Bell-LaPadula (BLP) model defines a mandatory security policy based on information secrecy. It addresses the issue of confidential access to classified information by controlling the information flow between security classes.

BLP partitions the organizational domain into *security levels* and assigns a *security label* to each level. Furthermore, each subject and object within the system need to be associated with one of those predefined levels and therefore associated with a security label. The security labels denominations may change from one organization to another, based on the terminology used in each organization. For instance in the context of military organizations, it would make sense to have labels like *Top secret, Secret, Classified*, while in a business organization we may find labels of the type *Board Only* which specifies the type of information that only the board of directors can access and *Managerial* which specifies the information that can be accessed by managers and so on.

A major goal of BLP is preventing the information to flow to lower or incompatible security classes [6]. This is done by enforcing three properties:

- **The Simple Security Property :** A subject can only have read access to objects of lower or equal security level (no read-up). For instance looking at the example in Figure 2.3, we can see that Subject *S1* can read both objects *O2* and *O3*, since they belong to a lower security level, the same for subject *S2*, it can read object *O2* since it belongs to the same security level, whereas it is not authorized to read object *O1*, since it belongs to a higher security level
- **The *-property (star-property):** A subject can only have write access to objects of higher or equal security level (i.e, no *write-down* is allowed). The example shown in Figure 2.4 depicts this property, where subject *S2* can write both in object *O1* and *O2* since they belong respectively to a higher and to the same security level, while subject *S1* can write only to object *O1* since it belongs to the same security level.
- **The Discretionary Security Property :** This property allows a subject to grant access to another subject, maintaining the rules imposed by the MAC. So subjects can only grant accesses for the objects over which they have the necessary authorizations that satisfy the MAC rules.
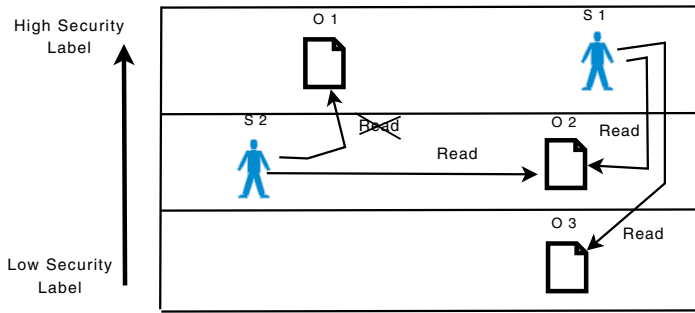
Fig. 2.3: BLP simple security property
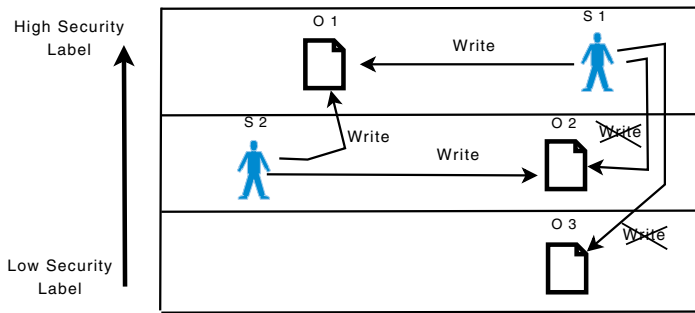
Fig. 2.4: BLP *-security property

2.2.2.2 Biba Model

Whereas the main goal of BLP is to ensure confidentiality, Biba's similar mechanism aims to ensure integrity. The basic idea behind Biba is to prevent low-integrity information to flow to higher-integrity objects, while allowing the opposite. Very often, the Biba model has been referred to as the reverse of the BLP model, since it exactly reverses the rules defined by the BLP. Before discussing the properties of the Biba model and looking to some examples, we once again remind the reader that the approach used by Biba is explicitly based on integrity rather than confidentiality or disclosure of information as in BLP. As mentioned earlier integrity refers to the ability of altering data objects. So the model prevents high integrity objects from being altered or changed by an unauthorized subject.

In a military environment, the Biba model will allow a subject with a high security label to insert mission-critical information like a military mission's target and its coordinates, whereas subjects with lower security labels can only read this information. This way the integrity of the system can be preserved more efficiently since the flow of information goes only from higher to lower levels [1]. The Biba model guarantees the following properties:

- **Simple Integrity Property :** A subject can only have read access to objects of higher or equal security level (no read-down).
- **Integrity *-property (star-property):** A subject can only have write access to objects of lower or equal security level (no write-up).

2.2.3 Role Based Access Control

Instead of providing access rights at user or group level as in DAC, *Role-Based Access Control (RBAC)* uses the roles assigned to users as a criteria to allow or deny access to system resources. Typically, RBAC models define users as individuals with specific roles and responsibilities within an organization. This relationship creates a natural mapping based on the organization's structure, and thus assigns access rights to each role rather than to each individual. The relation between roles and individuals is many-to-many, like the one between roles and system resources. A specific role may include one or more individuals, and at the same time a specific role can have access rights to one or more resources. Figure 2.5 depicts these relations.

The main idea behind RBAC is to group privileges in order to give organizations more control in enforcing their specific security polices. In other words, within an organization, the access control system deals with the job function of individuals rather than with their real identities. When permitting or denying access to a specific resource, the access right is intended to permit or deny access to a specific role and not to a specific individual.

Within the RBAC, a role is defined as "a set of actions and responsibilities associated with a particular working activity" [6]. The scope or the extent of the role can be limited to specific task or mission to accomplish, for instance *processing a client request, or preparing a report*, or it can concern a user's job like for instance *manager, director*.

2.3 Conclusions

Today's information systems and platforms include security features aimed at protecting confidentiality and integrity of digital resources. Controlling access to resources across an entire organization is a major security challenge. Access control models address this challenge by defining a set of abstractions capable of expressing access policy statements for a wide variety of information resources and devices. Access control systems are among the most widespread security features added to operating systems, middleware and application software platforms; therefore, certification of their security properties is of paramount importance. Many operating systems today, including Unix and Unix look-alikes like Linux, use DAC as part of their protection

Users

Role 1   Role 2
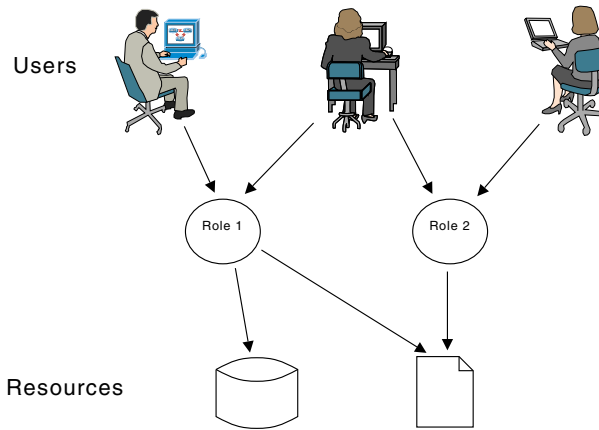
Resources

Fig. 2.5: Role-based access control users-roles-resources relations

mechanisms. DAC has also been adopted for controlling access to networking devices such as routers and switches. DAC's high flexibility enables resource owners to specify privileges assigned to each object.

References

1. E.G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall PTR, 1994.
2. M.G. Graff and K.R. Van Wyk. *Secure Coding: Principles and Practices*. O'Reilly, 2003.
3. M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.
4. G. McGraw. From the ground up: The dimacs software security workshop. In *IEEE Security and Privacy*, volume 1, pages 59–66, March 2003.
5. USA Department of Defense. *DEPARTMENT OF DEFENSE TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA*. USA Department of Defence, 1985.
6. P. Samarati and S. De Capitani di Vimercati. *Foundations of Security Analysis and Design*, chapter Access Control: Policies, Models, and Mechanisms, pages 137–196. Springer Berlin / Heidelberg, 2001.

Chapter 3
Test based security certifications

Abstract Test-based certifications are a powerful tool for software users to assess the extent to which a software product satisfies their security requirements. To exploit the full potential of this idea, it is necessary to have a shared way to describe the security properties to be tested, as well as to define the tests to be carried out and the testing environment itself. Since the early days of the US Department of Defense's *Trusted Computer System Evaluation Criteria (TCSEC)* [17], also known as *Orange Book*, several attempts have been made to standardize such descriptions, especially for operating system platforms; the trend has been toward increasing expressive power and complexity. A major result of this standardization activity is the *Common Criteria* (CC) , an ISO standard that defines a general test-based framework to specify, design and evaluate the security properties of IT products [12]. The introduction of CC came after almost two decades of joint work of many international organizations aimed at creating a common security standard that can be recognized at an international level. The rationale behind CC is to formalize the language used by customers, developers and security evaluators to have the same understanding when security requirements are specified and evaluated. In this chapter, after a brief primer to software testing, we will focus on test-based security certification, looking at the milestones that have led to the introduction of the CC standard; then, the general conceptual model underlying the CC standard will be explained.

3.1 Basic Notions on Software Testing

Software certification (see Chapter 1) is aimed at generating certificates demonstrating non-functional properties of software systems, such as the ones linked to dependability, security and safety. While the notion of interoperable software certificates as metadata items is a relatively new one, certification techniques build on well-known software validation and verification

techniques. Therefore, an in-depth discussion of test-based security certification requires some background knowledge on the software testing process. This section will provide such a background, focusing on the notion of *risk-oriented* testing. Experienced readers may safely skip it. Our main reference here is the excellent book by Paul Ammann and Jeff Offutt [1] and its rich bibliography, as well as the recent book by Pezzè and Young [19].[1]

Informally, software development can be defined as the process of designing (and implementing) a software product which meets some (functional and non-functional) user requirements. In turn, software testing can be defined as the process of validating a software product's functionality, and, even more importantly from our point of view, of verifying that the software has all the desired non-functional properties (performance, robustness, security and the like) its users expect. Hopefully, the testing process will also reveal the software product's defects.

In practice, software testing is performed as an iterative process: in each iteration, some tests are designed and executed to reveal software problems, and the detected problems are fixed. One should not expect miracles from this procedure: software testing can reveal the presence of failures, but it cannot provide proof of their absence.[2]

In this book, we distinguish between test-based certification as proceeding from testing, and model-based certification as proceeding from abstract models of the software such as automata or graphs. Some researchers, including Ammann and Offutt [1] justifiably argue that testing is also driven by abstract models, which can be developed to provide a *black-box* or a *white-box* view. Occasionally, *black-box* testing is carried out by professional testers who do not need to be acquainted with the code. The main purpose of black-box testing is to assess the extent to which functional and non-functional user requirements are satisfied;[3] each requirement is checked by a set of *test cases.*[4] The most important component of a test case is the *test case value*, that is, the input values fed into the software under test during a test execution of it. To evaluate the results of a black-box test case, we must know in advance its *expected results* or, in other words, the result that will be produced executing the test if (and only if) the program satisfies its requirements. Strictly speaking, a test case is the combination of test case values and expected results, plus two additional components: *prefix* and *postfix* values. Prefix values are inputs that, while not technically test values, are necessary to put the software into the appropriate state to feed it with the actual test case values. Postfix values are inputs that need to be sent to the software after the test

---

[1] We will follow references [1, 19] in the remaining of this chapter.

[2] The original statement is "But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." E. W. Dijkstra, "The Humble Programmer", Turing Award Lecture, 1972, Communications of the ACM, 15(10).

[3] Black-box tests can also be used to discover failures, such as crashes.

[4] Test-related literature often uses the term test set to indicate a set of test cases.

case values have been fed into it (e.g., for displaying results or terminating the test execution). For the sake of conciseness, throughout this book we will loosely use the term *test case* as a synonym of "test case value". Determining a software product's expected behavior may well be easy for some toy function (what should a `float squareroot(int )` function output when fed with a test case value of *4*?), but it is a much harder nut to crack for more complex programs (what should an image-enhancement software output when fed with a test-case value like `sampleimage.gif`?)

White-box tests have a very different aim: to ensure that some aspects of some code property are exercised in the hope that this will reveal any errors not otherwise detected. Also, white-box test can be used to verify coding standards; for instance, those barring the interchangeable use of pointer arithmetic and array reference in C. Therefore, they may consist of code walkthroughs rather than of sample executions.[5] In *white-box* testing , tests are designed and validated by developers, or at least by people who know well the code of the program under test.

Both white- and black-box testing can be *risk-oriented*. Risk orientation simply means that when some pre-defined constraints to testing are specified, (e.g., ones related to maximum test cost and available time for carrying out the tests) both types of tests can be prioritized so that risks for the software user are minimized as far as possible [10]. From the perspective of a software user, we may define the risk of running a software product in a very simple way, as follows:

$$R = \sum_{\{p\}} P_p I_p \qquad (3.1)$$

where P_p is the probability of occurrence of a software failure p, and I is the resulting impact of that problem, expressed in dollars of damage or profit loss. Note that implicitly we are also defining the risk connected to each specific problem p, that is, $P_p I_p$. The summation of Equation (3.1) ranges over the entire set of problems the software may have. In general, not all of these problems are known or can be forecasted at development time.

Of course, by using Equation (3.1) we are making the additional assumption that software failures are independent from each other. Although this is in general not the case, and often conditional probabilities must be used for correctly quantifying total risk, Equation (3.1) is good enough for our present purposes, that is, for illustrating the risks related to software delivery. Curtailing the test on a problem or forgoing them altogether means that the risk associated to that problem, not covered by the testing, will remain when the software is used. Risk analysis consists in the identification of software problems. It can be used for:

---

[5] Of course, in many practical situations walkthroughs would not be acceptable as a replacement for white-box execution tests.

- *goal oriented testing*: software failures having different risks will be covered by test procedures of different depths (and cost);
- *prioritized (depth-first) testing*: software areas with a higher risk receive a higher priority in the testing process.
- *comprehensive (width-first) testing*: software testing will be carried out to cover all rest categories at a pre-set minimum depth.

Risk-based testing consists of analyzing the software code and deriving a test plan focusing on software modules most likely to experience a failure that would have the highest impact. Both the impact and probability of each failure must be assessed before risk-based test planning can proceed. The impact I_p of the risk equation depends on the specific nature of the application and can be determined by domain analysis. For instance, failures in mission critical and safety critical modules may have assigned an higher impact than failures in the rest of the code[6]. Impact assessment requires a thorough understanding of failure costs, which is again highly domain-dependent. Estimating the likelihood of failures P_p means determining how likely it is that each component of the overall software product will fail. It has been proven that code that is more complex has a higher incidence of errors or problems [20]. For example, *cyclomatic complexity* is a well-known criterion for ranking the complexity of source code [16]. Therefore, in procedural languages like C the prioritization of modules according to failures probability can be done simply by sorting them by their cyclomatic complexity. Using cyclomatic complexity rankings for estimating failure probabilities, together with impact estimates from domain knowledge, analysts can pinpoint which software modules should get the highest testing effort [7].

A fundamental way towards the reduction of product risks is the finding and removal of errors in a software product. In the remainder of the chapter we will connect the idea of removing errors with the one of certifying software properties.

3.1.1 Types of Software Testing

Modern software products are often developed following processes composed of pre-defined activities.[8] The test process is also composed of several activities: during *test planning*, the test objects (e.g., the software functions to be tested) are identified. In *test case investigation*, the test cases for these test

---

[6] From a different point of view, software project management assigns highest impact to failures in modules on the project's *critical path*

[7] Of course, different programming languages and paradigms may use different complexity metrics than cyclomatic complexity for estimating failure probabilities.

[8] Such processes do include agile ones. We shall discuss how testing is carried out within community-based development process typical of open source software in Chapter 6.

objects are created and described in detail. Finally, during *test execution*, test scripts are executed to feed the test cases to the software under test.

Most software products have modular architectures, and testing can be done at different levels of granularity [19]:

- *Unit Testing*: tests the implementation of individual coding units.
- *Module Testing*: tests the detailed design of software modules
- *Integration Testing* tests each subsystem's design.
- *System Testing*: tests the entire system's architecture.
- *Acceptance Testing*: tests the extent to which the entire system meets requirements.

We are particularly interested in acceptance tests, because they are often related to non-functional requirements, that is, to properties the software product should possess in order to be acceptable for the customer. In principle, acceptance tests could be written by the end user herself; in practice, they are often co-designed with the customer, so that passing them with flying colors is guaranteed to satisfy both the software supplier and the purchaser that the software has all the desired properties. In a sense, obtaining the customer sign-off of a set of non-functional properties can be seen as equivalent to providing the same customer with a certificate of the same properties, signed by an authority she trusts.

Each of the above types of software testing involves one or more testing techniques, each of which is applied for different purposes, and requires the design and execution of specific test cases. Software testing techniques include [1]:

- *Functionality testing,* to verify the proper functionality of the software under test, including its meeting business requirements, correctly performing algorithms and providing the expected user experience.
- *Forced error testing,* to try extreme test cases (oversized and malformed inputs, erratic user behavior, etc.) in order to break (and fix) the software during testing, preventing customers from doing so in production.[9]
- *Compatibility testing,* to ensure that software is compatible with various operating systems platforms and other software packages.
- *Performance testing,* to see how the software product under test behaves under specific levels of load. Performance testing includes *stress testing* to see how the system performs under extreme conditions, such as a very large number of simultaneous users.
- *Regression testing,* to ensure that code added to address a specific problem did not introduce new problems.
- *Scalability testing,* a form of performance testing which aims to ensure that the software will function well as the number of users and size of databases increase.

---

[9] This can also be seen as a form of stress testing. See below.

- *Usability and accessibility testing,* to ensure that the software is accessible to all categories of users (including for example visually impaired ones,) and is intuitive to use. This kind of testing is traditionally the bailiwick of human computer interaction [5].
- *Security testing,* to make sure that valuable and sensitive data cannot be accessed inappropriately or compromised under concerted attack.

Of course the above list is not exhaustive: there are several other types of testing, including one very relevant to our purposes: *regulatory-compliance testing*, which verifies whether a software product complies with the regulations of the industry where it will be used (e.g., for the software controlling a manufacturing tool, one may test whether the user interface can be easily operated without taking off protection gloves). These regulations, and the corresponding testing procedure may vary depending on the type of software and application domain.

Since this book focuses on security certification, it is important to highlight the strict relationship we would like to establish between testing and certification. A key concept goes under the acronym *IV&V*, which stands for *Independent Verification and Validation*. Here, the term *independent* means that the tests on a software product are performed by non-developers; indeed, sometimes the IV&V team is recruited within the same project, other times within the same company developing the software. Conceptually, however, the step between this and outsourcing IV&V to an independently accredited lab is a small one.

It is also important to clarify the relation between security certification and security testing, which is less strict than one might expect. Security testing verifies all software features, from the password checking to channel encryption to making sure that information is not disclosed to unauthorized users. It also includes testing for known vulnerabilities, e.g., assessing how a software product responds to hostile attacks like feeding a software with so much input data that an *input buffer overflow* failure is produced, and the machine on which the software is run can be taken over by the attacker. Buffer overflows are by far the most commonly exploited bug on Linux.

Buffer overflows are the result of putting more data into a programs buffer or input device than is defined/allowed for in the program. A simple example of a C program susceptible to buffer overflow is given below:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
char buf[20];
if(argc < 2)
{
```

```
printf("Usage: \%s <echo>\n", argv[0]);
exit(0);
}
strcpy(buf, argv[1]);
printf("You typed: \%s\n", buf);
return 0;
}
```

The program (we will call `vultest`) copies the string `argv[1]` passed to it on the command line into an internal buffer whose address is `buf`. The internal buffer holds at most 20 chars, so if we pass a longer string on the command line we expect something bad to happen. What we get is the following behavior:

```
M-10:~ edamiani\$  ./vultest aaaaaaaaaaaaaaaaaaaaaaaa
<echo> aaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

The segmentation fault error message says that the buffer was filled with more data than it can hold. More specifically, `vultest` wrote too much data on the program stack, which was supposed to hold the arguments passed to `strcpy` (i.e., the `buf` array). These data overwrote the return address to the caller code, which is stored in the stack below the local variables; when `strcpy` tried to jump back to the caller using this return address, it found an illegal address in its stead (most likely the ASCII code for `aa`, `0x6565`) and tried to use it as the target for a jump, causing a violation. Suppose now the input string, rather than a sequence of `a` chars, contains the address of some code supplied by the attacker e.g., the one of a `sh` shell. This would give to the attacker the opportunity to seize `vultest`'s execution environment. There is an additional difficulty here: the attacker's code (let's call it `attack`) must be stored at a valid memory address on the system. In other words, the attacker has to know how to store her code in a valid address, obtain the address, and pass it (instead of `aa`) to `vultest` to overflow the buffer. For instance, the attacker can write `attack`, translate it into hexadecimal figures and load these figures into an array; then, he can use a memory copy system call like `memcpy` to copy the content of this array to some valid location in memory[10] keeping the address to pass it to `vultest`.[11]

---

[10] We will discuss `memcpy` in some detail in the next Chapter

[11] Some pre-processing (e.g., format conversion) is required before the address can be passed to `vultest`.

3.1.2 Automation of Test Activities

All types of software testing described above can be performed manually or automated.

Manual software testing usually consists of having a human tester feed test cases to the software interface, and performing other types of manual interaction, including trying to hack the software. There are some areas for which manual software testing is most appropriate (or, rather, is the only possibility), such as exploratory security testing where testers do not execute a pre-defined script, but rather explore the application and use their experience and intuition to identify vulnerabilities. Manual software testing has two major disadvantages: first of all, it requires a huge effort and remarkable skills on the part of the tester. Secondly, it is not fully repeatable: manual exploratory tests tend to show high variability of results depending on who is performing them.

Automated software testing can be seen as a way of accelerating the software testing process while minimizing the variability of results. Automation is based on test scripts that can be run automatically, repetitively, and through several iterations of the test process. Automated software testing can help to minimize the variability of results, speed up the testing process, increase test coverage, and ultimately provide greater confidence in the quality of the software being tested. Many testing tasks that defied automation in the past have now become candidates for such automation due to advances in technology. For example, generating test cases that satisfy a set of functional requirements was typically a hard problem that required intervention from an experienced test engineer. Today, however, many tools are available, both commercial and open source, that automate this task to varying degrees, e.g., generating Unix shell scripts to feed input files into the program under test and produce a report. We shall mention again these testing platforms in Chapter 6. The interested readers are however referred to [19].

3.1.3 Fault Terminology

The remainder of this section presents three terms that are important in software certification as well as in testing and will be used later in this chapter. Like terms about certification, terms about testing are defined by standard glossaries such as the IEEE Standard Glossary of Software Engineering Terminology, DOD-STD-2167A and MIL-STD-498 from the US Department of Defense. We focus on the types of problems that using a software product may involve. Throughout the book, following [1], we shall adopt the following definitions of software *fault*, *error*, and *failure*.

Definition 3.1 (Software Fault). A static defect in the software.

Definition 3.2 (Software Error). An incorrect internal state that is the manifestation of some fault.

Definition 3.3 (Software Failure). External, incorrect behavior with respect to the requirements or other description of the expected behavior.

In order to clarify the definitions of fault, error, and failure, we shall use a revised version of an example presented in [1]. Namely, let us consider the following C program:

```
int digitcount (char s[])
{
  int digit = 0;
  int i;
  for (i = 1; s[i] != '\0'; i++)
      if(s[i] >= '0' && s[i] <= '9')
          digit++;
  return digit;
}
```

The *software fault* in the above sample function is of course the instruction `for`, where the function starts counting digit characters at index 1 instead of 0, as would be correct for C character arrays.

For example, using the test case values [a, b, 0] and [0, 7, c] we notice that `digitcount`([a, b, 0]) correctly evaluates to 1, while the test case `digitcount`([0, 7, c]) incorrectly gives the same result. Note that only the latter test execution of `digitcount` results in a *software failure*, although the faulty instruction is executed the same number of times in both cases. Also, both the execution with failure and the one without it involve a more elusive concept, the one of *software error*. In order to fully understand it, we need to execute our faulty function stepwise, meaning that we consider its state (i.e., the content of the function's local variables).

The state of `digitcount` consists of three memory locations, containing values for the variables `s`, `digit`, `i`. For the first test case execution, the state at the first iteration of the loop is (`s` = [a, b, 0], `digit` = 0, `i` = 1). This execution state is a software error, because the value of the variable `i` should be zero on the first iteration. However, since the value of variable `digit` is (purely by chance!) correct, this time the error does not cause a failure. In the second test case execution, the error state is (`s` = [0, 7, c], `digit` = 0, `i` = 1). Here the error propagates to the variable `digit` and causes a failure. Now, it is clear that when a software product contains a fault, not all test cases will ensure that the corresponding error will cause a failure, how it would be desirable in order to reveal and fix the fault itself. In addition, even when a failure does occur, it is may be very difficult to trace it back to the fault which caused it.

3.1.4 Test Coverage

Since the term "certification" has the same root as the term "certainty", one might be tempted to think that black-box tests can provide conclusive evidence that a software product holds (or does not hold) a given property. For instance, can we use black-box testing to prove to a software purchaser's satisfaction that the execution of the software product she is buying will never require more than 1 MByte of user memory?

Unfortunately, this need for conclusive evidence clashes with a theoretical limitation of software testing. Even for our simple `digitcount` function, using 8-bit ASCII character coding and excluding arrays of characters longer than 256 characters, one would require 2^{64} executions to run all possible test cases.

We may understand better how the identification of the "right" test cases is carried out via the notion of *coverage criteria*. In practice, coverage criteria correspond to properties of test cases. The tester tries to select test cases showing the whole range of their properties values, that is, providing the maximum coverage.

Let us briefly examine some types of coverage that are relevant to our purposes. A classic coverage criterion is to execute all `if` alternatives (i.e., cover all decisions) in the program. This criterion is called *branch coverage*. The corresponding property of test cases we are interested in is which `if` selectors (if any) they trigger. To satisfy this criterion, the tester will chose test cases so that each of them causes the execution of one or more (non-overlapping) branches controlled by `if` selectors in the program. Ideally, the tester will obtain a test set which will cause the execution of all the program's branches, achieving *full branch coverage*. An analogous line of reasoning leads us to the notion of *full statement coverage* criterion, that is, a set of test cases which causes the execution of all the program's statements.

Coverage criteria can be related to one another, in terms of a relation called *subsumption* [11]. A coverage criterion C_1 subsumes C_2 if (and only if) every test set that satisfies criterion C_1 also satisfies C_2. In the case of branch and statement coverage, it is easy to see that if a test set covers every branch in a program, then the same test set is guaranteed to have covered all statements as well. In other words, the branch coverage criterion subsumes the statement coverage criterion.[12]

It is important to realize that some coverage criteria cannot be satisfied at all. For instance in the case of the following C function:

```
int digitcount (char s[])
{
 int i;
 int digit = 0;
```

_____

[12] Our intuition may tell us that if one coverage criterion subsumes another, it should reveal more faults. However, this intuition is not supported by any theoretical result [23].

```
    if (digit) i=0;
    for (i = 1; s[i] != '\0'; i++)
        if (s[i] >= '0' && s[i] <= '9')
            digit++;
    return digit;
}
```

there is no set of test cases ensuring full statement coverage, due to the presence of *dead code*: the statement i = 0; can never be reached, regardless of the input.

One may think to find an algorithm to decide whether such a test set exists or not; unfortunately, there can be no algorithm for deciding whether an arbitrary program can get full coverage with respect to an arbitrarily chosen set of coverage criteria, even though some partial solutions (i.e., for special classes of programs and/or criteria) have been proposed (see Chapter 4). In other words, achieving 100% coverage for any set of coverage criteria is impossible in practice, and there is no way to design a test set that will detect all faults.[13].

Something can be done, anyway: coverage criteria can be used either to generate test case values or to validate randomly generated or manually picked ones. Both problems are in general (i.e., when the criteria are arbitrary) undecidable; the latter technique, however, is the one adopted in practice, because the validation problem turns out to be tractable much more often than the generation one.[14] There is however a drawback: validating randomly chosen test cases will allow us to assess the extent to which a given test set provides coverage, but leave us clueless on how to increase it. In terms of commercial automated test tools, a *test case generator* is a tool that automatically creates test case values. A validator tool that takes a test set and performs its coverage analysis with respect to some criterion. Both types of tools are available as commercial and open source products. Some well-known tools include *xUnit* (JUnit, CPPUnit, NUnit and so on), *IBM Rational Functional Tester*, *WinRunner*, *DejaGnu*, *SMARTS*, *QADirector*, *Test Manager*, *T-Plan Professional*, and *Automated Test Designer* (ATD).

3.2 Test-based Security Certification

For a long time, testing in general and security testing in particular have been internal processes of software suppliers. Software purchasers had prac-

---

[13] The result that finding all failures in a program is undecidable is due to Howden [13]

[14] More precisely, given a criterion checking whether some existing test cases satisfy, it is feasible far more often than it is possible to generate tests for that criterion starting from scratch.

tically no way to obtain an independent appraisal of a software product's security prior to buying it. Often, disclaimers coming with software products would exclude any guarantee, expressed or implied, of any security or dependability property. This situation is now simply unacceptable for organizations purchasing safety and mission-critical systems. Generally speaking, security certification standards have been devised to provide purchasers with some guarantee of the security properties of their software.

Intuitively, the security certification process of a software product should reveal all the problems and faults of the product's security features, which could lead to vulnerability to attacks. In the early days of security certifications software vendors would limit themselves to asserting (and testing) the presence and functionality of security features, often as a part of non-functional requirements elicitation, and left it to the user to make the link between the support of a given security feature and the corresponding security property. For instance, early certificates would state that a given software system supported Access Control Lists (ACL) on data resources, leaving it to the user to make the connection between the ACL mechanism and the specific category of discretionary access control policies she was interested in.
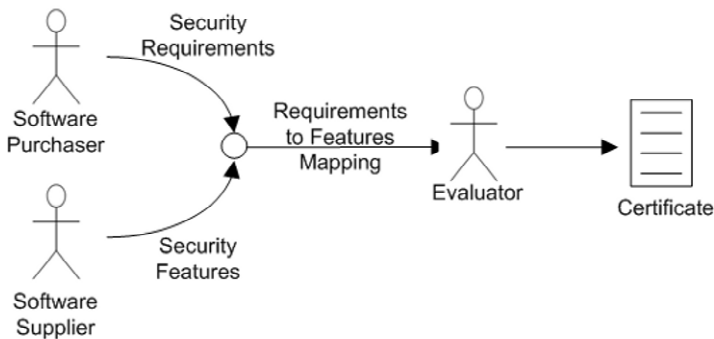
Fig. 3.1: The conceptual model for test-based security certification

The software certification process has greatly evolved along the years. Figure 3.1 shows an abstract conceptual model for today's test-based security certification. The first phase of the security certification process is providing a mapping between the security properties (in terms of security requirements) the software purchaser is interested in and the security features the software vendor has included in the product. Once a mapping has been specified between a set of security properties and the corresponding security features, test-based security certifications provide test-backed proof that: *(i)* the software product under certification actually possesses the required features and *(ii)* such features perform exactly their intended functionalities and nothing else. If the test process is carried out in a controlled environment and by

a trusted evaluation body, e.g., as mandated by an internationally accepted *assurance* standard, this proof is usually acceptable enough to the software purchaser.

In the next chapters, we shall use the term "assurance" to refer to all activities necessary to provide enough confidence that a software product will satisfy its users' security requirements. In other words, security standards specify which security requirements a product should satisfy, while assurance standards specify how to collect and provide the evidence that it does. We shall elaborate further on this issue in Section 3.3. Also, in this chapter we shall use the term "proof" quite loosely: security feature testing (like any other software test) can never be exhaustive, and the mapping between features and properties may or may not have been formalized and proved. We shall come back to these problems when dealing with model-based security certification in Chapter 4.

In the remainder of this chapter, we shall briefly review some early security certification standards adopted in the US, in Canada and in Europe. These early mechanisms are still with us, providing much of the vocabulary shared by software purchasers, vendors and security auditors.

3.2.1 The Trusted Computer System Evaluation Criteria (TCSEC) standard

As one of the biggest software buyers worldwide, the U.S Department of Defense (DoD) has been understandably keen on addressing the issue of standardizing security certifications . The U.S. Department of Defense's *National Computer Security Center* (NCSC) has sponsored the introduction of what is known as the *Trusted Computer System Evaluation Criteria* (TCSEC) or *Orange Book*. Originally, the Orange Book was devised as a way of standardizing security requirements coming from both the government and the industry [21]; although it was originally written with military systems and applications in mind, its security classifications have been broadly used within the entire computer industry. According to its proposers, the Orange Book was created with the following basic objectives in mind [17].

1. Provide sound security guidelines for manufacturers to be able to build products that satisfy the security requirements of applications.
2. Define a systematic way to (qualitatively) measure the level of trust provided by computer systems for secure, classified and other sensitive information.
3. Allow software purchasers to specify their own security requirements, rather than take or leave fixed sets of security features defined by suppliers.

The classifications in the Orange Book provide a useful shorthand for the basic security features of operating systems.[15]

In the course of time, the NCSC has published different "interpretations" of the Orange Book . These interpretations have clarified the Orange Book requirements with respect to specific families of operating system components. For example, the NCSC's Trusted Network Interpretation of the Orange Book, also known as *Red Book*, is an interpretation of Orange Book security requirements as they apply to the networking components of a secure operating system. The Red Book does not change the Orange book original requirements; it simply indicates how a networking system should operate to meet them. Interpretations come in several colors: in the same way as the Red Book is an interpretation of the Orange Book for network systems, there is a Blue Book interpreting the Orange Book for subsystem components, and other books for other component families. The NCSC has made available a complete set of Orange Book interpretations (the so-called *Rainbow Series*), to assist software vendors in ensuring that their systems comply with Orange Book requirements.

Orange Book security certification goes one step further with respect to the situation where suppliers agreed on non-functional requirements about the presence of some security features. Provisions that need to be present for considering a system to be "secure" are mapped to specific *security requirements*, which must be provided by purchasers. To help software users in defining their own security requirements, the Orange Book introduced four fundamental *requirement types*, which are derived from the objectives stated above [17]: *Policy, Accountability, Assurance* and *Documentation.*

Despite promoting a novel view in supporting software purchasers' own security requirements, the Orange Book ended up with severe limitations in many aspects, which prevented its generalized adoption. These limitations are mainly due to the Book's lack of flexibility in expressing security requirements and in mapping them to security features. The original users of the Orange Book were military and governmental organizations having very specific security requirements, mostly related to preventing disclosure of classified data. This brought families of security requirements which are of paramount importance for business applications (e.g., requirements related to availability and integrity) to be insufficiently considered, and the Orange Book failed to adapt to the security requirements of a wider general market [2].

Today, the Orange Book's main legacy consists in its classification of software products (mainly operating systems) into pre-set categories based on the security requirements they satisfy (and, correspondingly, on the features they possess). TCSEC categories are identified by labels forming an ordinal

---

[15] While the Orange Book security categories have played an important role in filling the communication gap between vendors, evaluators, and customers, the Orange Book itself [17] is notoriously difficult to read. Also, the Orange Book is not readily available to non-US parties, which has made a full understanding of TCSEC security ratings rather difficult to achieve for security experts outside the US.

scale, allowing for qualitative assessment of security and system comparison. Verifying the actual functionality of security features via suitable tests is left to an external evauation body, who should be neither the software supplier nor the purchaser. The Orange Book provides some high-level guidelines for testing security requirements, but does not mandate a specific test process or laboratory setting. Also, software products are only listed on the NCSC's Evaluated Products List after a long evaluation process culminating in a Final Evaluation Report from NCSC. It is important to remark that the system submitted to NCSC for certification can include additional modules (e.g., hardware ones) specifically added to comply with TCSEC requirements. This means that users will have to install this additional hardware or software in order to meet the security requirements.

3.2.1.1 TCSEC categories and requirements

The Orange Book security categories range from D (Minimal Protection) to A (Verified Protection). To be classified in a given category, a software system must provide all the security features corresponding to that category. Namely, categories are defined as follows (see also Table 3.1).

Category	Class	Comment
D - Minimal Protection	-	Category D includes any system that does not comply with any other category, or has failed to receive a higher classification.
C - Discretionary Protection	C1 - Discretionary Security Protection	Category C applies to *Trusted Computer Bases* (TCBs) with optional object (i.e., file, directory, devices etc.) protection.
	C2 - Controlled Access Protection	
B - Mandatory Protection	B1 - Labelled Security Protection	Category B specifies that the TCB protection systems should be mandatory, not discretionary.
	B2 - Structured Protection	
	B3 - Security Domains	
A - Verified Protection	A1 - Verified Protection	Category A is characterized by the use of formal security verification methods to assure
	A1 and above	that the mandatory and discretionary security controls are correctly employed.

Table 3.1: Orange book categories and classes

- *D - Minimal Protection.* This category includes any system that does not comply with any other category, or has failed to receive a higher classification. D-level certification is very rare.
- *C - Discretionary Protection.* Discretionary protection applies to *Trusted Computer Bases* (TCBs) with optional object (i.e., file, directory, devices and the like) protection.

  - *C1 - Discretionary Security Protection.* This category includes systems whose users are all on the same security level; however, the systems have provisions for Discretionary Access Control by providing separation of users and data, as for example Access Control Lists (ACLs) or User/Group/World protection. An example of C1 requirements is shown in Table 3.2. C1 certification is quite rare and has been used for earlier versions of Unix.
  - *C2 - Controlled Access Protection.* This category has the same features as C1, except for the addition of object protection on a single-user basis, e.g., through an ACL or a Trustee database. C2 is more fine-grained of C1 and makes users individually accountable for their actions. An example of C2 requirements is shown in Table 3.2. C2 is one of the most common certifications. Some of the Operating Systems using C2 certification are: VMS, IBM OS/400, Windows NT, Novell NetWare 4.11, Oracle 7, DG AOS/VS II.

Class	Requirements
C1	Username and Password protection and secure authorisations database (ADB)
	Protected operating system and system operations mode
	Periodic integrity checking of TCB
	Tested security features with no obvious bypasses
	Documentation for User Security
	Documentation for Systems Administration Security
	Documentation for Security Testing
	TCB design documentation
C2	Authorisation for access may only be assigned by authorised users
	Object reuse protection (i.e., to avoid reallocation of secure deleted objects)
	Mandatory identification and authorisation procedures for users, e.g., Username/Password
	Full auditing of security events (i.e., date/time, event, user, success/failure, terminal ID)
	Protected system mode of operation
	Added protection for authorisation and audit data
	Documentation as C1 plus information on examining audit information

Table 3.2: C classes' requirements

- *B - Mandatory Protection.* It specifies that the TCB protection systems should be mandatory, not discretionary. A major requirement here is the protection of the integrity of sensitivity labels and their adoption to enforce a set of mandatory access control rules.

  – *B1 - Labelled Security Protection.* B1 systems require all the features required for class C2 and the requirements provided in Table 3.3. Some of the operating systems and environments using B1 certification include: HP-UX BLS, Cray Research Trusted Unicos 8.0, Digital SEVMS, Harris CS/SX, SGI Trusted IRIX, DEC ULTRIX, Trusted Oracle 7.
  – *B2 - Structured Protection.* B2 systems require all the features required for class B1 and the requirements provided in Table 3.3. Some of the systems using B2 certification are: Honeywell Multics, Cryptek VSLAN, Trusted XENIX 4.0.
  – *B3 - Security Domains.* B3 systems require all the features required for class B2 and the requirements provided in Table 3.3. The only B3-certified OS is Getronics/Wang Federal XTS-300.

- *A - Verified Protection.* It is the highest security category. Category A is characterized by the use of formal security verification methods to assure that the mandatory and discretionary access control models correctly protect classified or other sensitive information stored or processed by the system [17]. TCB must meet the security requirements in all aspects of design, development and implementation.

  – *A1 - Verified Protection.* A1 systems require all the features required for class B3 and formal methods and proof of integrity of TCB. The following are the only A1-certified systems: Boeing MLS LAN, Gemini Trusted Network Processor, Honeywell SCOMP. All of them are network components rather than operating systems.
  – *A1 and above.* The Orange Book mentions future provisions for security levels higher than A2, although these have never been formally defined.

3.2.1.2 A closer look

We now take a closer look to the security features corresponding to the Orange Book categories, in order to spell out the main innovation of the TCSEC approach, that is, the mapping between security requirements and security features. We shall focus on C2-level security, which is a requirement of many U.S. government installations.[16] Here, we shall consider four of the most important requirements of TCSEC C2-level security:

---

[16] In the European Union, government agencies usually refer to ITSEC categories, introduced in the next Section. The corresponding ITSEC rating is E3.

Class	Requirements
B1	Mandatory security and access labeling of all objects, e.g., files, processes, devices and so on
	Label integrity checking (e.g., maintenance of sensitivity labels when data is exported)
	Auditing of labelled objects
	Mandatory access control for all operations
	Ability to specify security level printed on human-readable output (e.g., printers)
	Ability to specify security level on any machine-readable output
	Enhanced auditing
	Enhanced protection of Operating System
	Improved documentation
B2	Notification of security level changes affecting interactive users
	Hierarchical device labels
	Mandatory access over all objects and devices
	Trusted path communications between user and system
	Tracking down of covert storage channels
	Tighter system operations mode into multilevel independent units
	Covert channel analysis
	Improved security testing
	Formal models of TCB
	Version, update and patch analysis and auditing
B3	ACLs additionally based on groups and identifiers
	Trusted path access and authentication
	Automatic security analysis
	TCB models more formal
	Auditing of security auditing events
	Trusted recovery after system down and relevant documentation
	Zero design flaws in TCB, and minimum implementation flaws

Table 3.3: B classes' requirements

- *Discretionary Access Control* (also in C1). The owner of a resource, such as a file, must be able to control access to it.
- *Secure Object Reuse* (C2 specific). The operating system must protect data stored in memory for one process so that it is not randomly reused by other processes. For example, operating systems must swipe per-process memory after use, including kernel-level data structures, so that (user and kernel) per-process data cannot be peeked after the memory has been freed.[17]
- *Identification and Authentication* (C2 specific). Each user must uniquely identify himself or herself when logging onto the system. After login, the system must be able to use this unique identification to keep track the user's activities. In many operating systems, this is achieved by typing a

---

[17] This requirement applies to the entire memory hierarchy, including disk storage: after a file has been deleted, users must no longer be able to access the file's content. This requires some protection to be applied when the disk space formerly used by a file is re-allocated, e.g., for use by another file.

username and password pair before being allowed access. We shall further explore this requirement in Chapter 5.

- *Auditing* (C2 specific). System administrators must be able to audit the actions of individual users, as well as all security-related events. Access to this audit data must be limited to authorized administrators.

Let us now describe the mapping between these requirements and the security features.

- *Discretionary Access Control.* From a system management perspective, the discretionary access control requirement involves the presence and functionality of a number of security features. For example, a mechanism such as Access Control Lists must be in place for the system administrator to control which users have access rights to which system resources, including files, directories, servers, printers, and applications. Rights must be definable on each resource basis and managed centrally from any single location. A user-group management tool must also be present, through which the system administrator can specify group memberships and other user account parameters.

- *Secure Object Reuse.* This requirement involves the presence and functionality of a number of security features corresponding to the different levels of the memory hierarchy. When a program accesses data, those data are placed in main memory, from where they can be swapped to disk by the virtual memory mechanism. This means that at the level of main memory, two security features are required to satisfy this requirement: *(i)* a *Memory Management Unit (MMU)* level mechanism ensuring the protection of data in the machine's physical memory, so that only authorized programs can access them, and *(ii)* a swap partition protection memory to ensure that no process can access the disk portion hosting the virtual memory used by another process. When these two mechanisms are in place and work correctly, it is impossible for a rogue application to take advantage of another application's data.

- *Identification and Authentication.* A simple password-based log-on procedure may suffice, provided it uses a system-level encryption of passwords so that they are never passed over the wire. This encryption prevents unauthorized discovery of a user's password through eavesdropping.[18]

- *Auditing.* An encrypted log feature must exist supporting logging of all security related events such as user access to files, directories, printers and other resources and log-on attempts. A simple symmetric key encryption mechanism is sufficient to guarantee the secure access to logs mentioned by the requirement.

The mapping described above underlies the entire certification process . When checking a security requirement, the evaluator checks that all the cor-

---

[18] However, more complex system can be used to satisfy other requirements as well. We shall further explore these mechanisms in Chapter 5.

responding security features are in place and test them (according to some pre-set test guidelines) to verify they are working correctly. If all tests succeed, the product certifiably satisfies the requirements and gets the corresponding certificate. It is important to understand that there are a number of other requirements, such as the usability of the security features, that the TCSEC guidelines do not directly address. For example, the fact that a software product has achieved the C2 certification guarantees that it includes a security feature (e.g., an ACL-based one) capable of controlling which users have access rights to which resources; but such a feature can be extremely awkward to use without a GUI. Again, as far as user accounts and group memberships are concerned, having checked the presence and functionality of a bare-bones security feature for managing users and groups will not guarantee the possibility of displaying log-in times, account expirations, and other related parameters which will substantially increase the feature usability. These additional requirements are not covered by certifications and therefore remain part of the negotiation between software system vendors and purchasers.

3.2.2 CTCPEC

The Canadian Trusted Computer Product Evaluation Criteria or the CTCPEC was proposed as a revised, "demilitarized" version of the US Orange Book. The CTCPEC goal was to define a wider set of types to accommodate diverse security requirements. The original TCSEC security requirement types were extended to deal with software *integrity, assurance, accountability, confidentiality* and *availability* as well as the original types defined by the Orange Book [2]. In other words, the CTCPEC addressed the commercial market demands by supporting a richer classification of security requirements and more expressive mapping of these requirements to security features. Understandably, and regardless of the efforts to ensure backward compatibility, the wider scope of the CTCPEC caused a growing incompatibility with the Orange Book [2]. This incompatibility, in turn, became a major driver toward a unified, international security certification standard.

3.2.3 ITSEC

At the same time when Canada was trying to define its new security certification standard, on the other side of the Atlantic several countries like France, Germany, UK and the Netherlands started working together to develop a certification designed to satisfy the security needs of the European industry. Learning from the US experience, the Information Technology Security Evaluation Criteria (ITSEC) authors tried also to define a set of goals

to overcome the limitations of the Orange Book as well as defining new goals that fit in the European context. Version 1.2 of the ITSEC standard was released in 1991 and is still used today. The major goals of the ITSEC as described in [15] are:

- *Generality*. ITSEC certification criteria are not limited to any category of software products.
- *Interoperability*. ITSEC ensures compatibility with the national catalogs of security evaluation criteria used by each European country, and definition of mappings from these national catalogs to ITSEC.
- *Neutrality*. ITSEC is supported by third parties, taking a neutral role between software.
- *Scalability*. ITSEC contains guidelines for testing security features, aimed at achieving full automatization of the certification process.

3.3 The Common Criteria : A General Model for Test-based Certification

The work done within ITSEC identified two major extension areas to test-based security certification techniques. The first area regarded increasing the expressive power of the security requirements, extending and formalizing their type systems and specifying matching from the requirement to the feature space. The second extension area dealt with automation of feature testing. Both issues were addressed by a standardization group including representatives of the US, Canada and European Union, the latter with the exception of Italy (the group included neither Japan nor Australia). This joint effort, started in 1993 under the label *Common Criteria for IT Security* (CC) became an ISO standard (ISO/IEC 15408) in 1998. The final version of ISO/IEC 15408 was released in 1999. The *Common Criteria* (CC) certification standard has been defined to fulfill the needs of an international standard for affordable software security certification . Common Criteria provides an unified process and a flexible framework to specify, design, and evaluate the security properties of IT products [12].

A major goal of CC evaluation is to certify that the security policies claimed by the developers are correctly enforced by the security functions of the product under evaluation. The Common Criteria model tries to capture all the security aspects of the product and uses the term *Target Of Evaluation* (TOE) for the technological product under evaluation. A TOE can be software, firmware, hardware or a combination of the three [6]; also, it can be a subsystem rather than an entire software system. In this case only the sub-system defined as the TOE will be certified and not the entire product. Figure 3.2 depicts the general model of the CC evaluation.
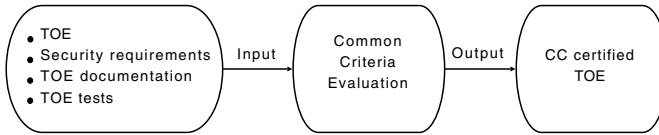
Fig. 3.2: The General Model of the CC evaluation process

In the following of this section, we start by surveying the CC terminology and rationale; in Chapter 6, we will present a detailed case study.

3.3.1 CC components

The CC defines a set of components and specifies the way they interact with each other. CC components can be divided into three categories.

- Catalogs of reusable security functional and assurance requirements.
- Evaluation Assurance Levels (EALs) , specifying the assurance level used in the certification process (from 1 to 7).
- The Protection Profile (PP) and specification of Security Target (ST) , describing respectively the security requirements and the security features of the product to be certified.

It is important to remark that CC Protection Profiles and specifications of Security Targets are themselves *software artifacts*, potentially to be published and exchanged among suppliers, purchasers and independent third parties such as evaluators.

3.3.1.1 The Protection Profile

The introduction of the *Protection Profile* (PP) is an important innovation of the CC, inasmuch it allows groups or consortia of software purchasers to define and share their own sets of security requirements. Of course, PPs do not mandate how (i.e., via which features) these requirements must be implemented; rather, they contain high-level descriptions of users' needs. Also, PPs are not written in a formal or controlled language; indeed, when comparing the PP structure defined in CC part 1 [6] to the publicly available instances of PPs, one may notice that even their structure changes slightly. To get a better idea of what a real PP looks like, we shall use examples taken from real-world PPs, considering their structure rather than the standard one defined by [6]. The general structure of a PP contains the sections showed in Figure 3.3 and discussed below.

1. PP introduction

   - The PP identification
   - PP overview
   - Conventions and document organization
   - Terms and keywords
   - EAL

2. TOE description
3. Security problem definition

   - Threats
   - Organizational security policies
   - Assumptions

4. Security objectives

   - Security objectives of the TOE
   - Security objectives of the TOE environment
   - Security objectives rationale

5. Security requirements

   - Security functional requirements
   - Security Assurance requirements
   - Security requirements rationale

6. PP rationale

Fig. 3.3: PP general structure

The *PP introduction* section provides general information about the PP, allowing it to be registered through the Protection Profile registry, searched and shared. This section includes the *PP identification*, that is, a descriptive information to identify, catalogue, register, and cross-reference a PP. The *PP overview* describes the scope of the PP and provides the necessary information for customers to decide if a particular PP is appropriate for their needs. The two other subsections of the PP introduction section (conventions and document organization, and terms and keywords) help the reader to understand how the PP document is organized and provide basic definitions of any domain specific-term used in the document. Furthermore, the *EAL* for which the PP claims conformance is also mentioned [18]. Figure 3.4 shows the introduction of a well-known PP , the Controlled Access Protection Profile (CAPP) specifying desirable high-level security requirements related to discretionary access control.

The *TOE description* describes the software product from the customer point of view, which includes the purpose of the TOE, the security functionalities needed and the intended operational environment. Other information concerning some technical details could be added such as cryptographic requirements, remote access requirements, and so on. Figure 3.5 shows an ex-

---

Example of PP Introduction section.

- *PP identification*:
    - *Title*: Controlled Access Protection Profile (CAPP)
    - *Registration*: Information Systems Security Organization (ISSO)
    - *Keywords*: access control, discretionary access control, general-purpose operating system, information protection

- *PP overview*: The Common Criteria (CC) Controlled Access Protection Profile , hereafter called CAPP , specifies a set of security functional and assurance requirements for Information Technology (IT) products. CAPP-conformant products support access controls that are capable of enforcing access limitations on individual users and data objects. CAPP-conformant products also provide an audit capability which records the security-relevant events which occur within the system.
- *Conventions*: This document is organized based on Annex B of Part 1 of the Common Criteria .There are several deviations in the organization of this profile. First, rather than being a separate section, the application notes have been integrated with requirements and indicated as notes. Likewise, the rationale has been
- *Terms*:
    - A user is an individual who attempts to invoke a service offered by the TOE.
    - An authorized user is a user who has been properly identified and authenticated. These users are considered to be legitimate users of the TOE.
    - An authorized administrator is an authorized user who has been granted the authority to manage the TOE. These users are expected to use this authority only in the manner prescribed by the guidance given them.
    - The Discretionary Access Control policy, also referred to as DAC, is the basic policy that a CAPP conformant TOE enforces over users and resources.

- *EAL*

Fig. 3.4: Example of the PP introduction section [4]

ample of a TOE description regarding a general purpose operating system including some *Components-Off-the-Shelf* (COTS) provided by third parties.

The *security problem definition* section describes the expected operational environment of the TOE. More specifically, it defines the known security threats, the security assumptions and the organizational security policies. It is important to notice that it is not mandatory to have statements for all the three subsections. In other words, there may well be cases in which there are no assumptions, or no organizational policies to speak of. However, defining the security threats in a clear and unambiguous way is important, because it makes the construction of the PP easier.

The threats subsection describes the potential threats that may put at risk the TOE assets. In other words this subsection states what we want to protect the TOE from, including violations by system administrators, hackers,

TOE Description. This protection profile specifies requirements for multilevel general-purpose, multi-user, COTS operating systems together with the underlying hardware for use in National Security Systems. Such operating systems are typically employed in a networked office automation environment containing file systems, printing services, network services and data archival services and can host other applications (e.g., mail, databases). This profile does not specify any security characteristics of security-hardened devices (e.g., guards, firewalls) that provide environment protection at network boundaries. When this TOE is used in composition with other products to make up a larger national security system, the boundary protection must provide the appropriate security mechanisms, cryptographic strengths and assurances as approved by NSA to ensure adequate protection for the security and integrity of this TOE and the information it protects.

Fig. 3.5: Example of PP TOE description [4]

unauthorized users, and so forth. Table 3.4 shows some examples of threats definitions suitable for an operating system.

Threat	Description
T.ADMIN_ERROR	An administrator may incorrectly install or configure the TOE resulting in ineffective security mechanisms.
T.ADMIN_ROGUE	An authorized administrator's intentions may become malicious resulting in user or TSF data being compromised.
T.SPOOFING	A malicious user, process, or external IT entity may misrepresent itself as the TOE to obtain authentication data.

Table 3.4: Threats definition [4]

Organizational Security Policies (OSPs) are the set of roles, rules and procedures adopted by the organizations using the TOE to protect its assets. OSPs can be defined either by the organization that controls the operational environment of the TOE or by external regulatory bodies [6]. Table 3.5 shows a fragment of an OSP definition specifying the administration roles which will be involved in setting the TOE access control policies, and some separation constraints on them.

During the certification process, some *assumptions* will inevitably have to be made, purely and simply because it is almost impossible to adopt the same set of requirements for all customers. When software purchasers write a PP, they need to take in consideration their specific needs which may change among different groups of customers, even regarding the same product. Focusing on the operating system example, we might well have two groups of customers who define two different PPs for the same operating system. One group may assume that the operating system will include features capable of enforcing access control to classified information, while the other group

OSP	description
P.ACCOUNTABILITY	The users of the TOE shall be held accountable for their actions within the TOE.
P.AUTHORIZED_USERS	Only those users who have been authorized to access the information within the TOE may access the TOE.
P.ROLES	The TOE shall provide multiple administrative roles for secure administration of the TOE. These roles shall be separate and distinct from each other.

Table 3.5: PP Organizational security policies from [4]

may assume that access will be regulated by suitable organizational security policies. Table 3.6 illustrates some assumptions [18].

Assumptions	Description
A.LOCATE	The processing resources of the TOE will be located within controlled access facilities which will prevent unauthorized physical access.
A.PROTECT	The TOE hardware and software critical to security policy enforcement will be protected from unauthorized physical modification.
A.MANAGE	There will be one or more competent individuals assigned to manage the TOE and the security of the information it contains.

Table 3.6: PP assumptions from [18]

Based on the security problems defined in the previous sections of the PP, the *security objectives* section provides a set of concise statements as responses to those issues. Every problem definition must be adequately addressed by one or more security objectives. Determining the security objectives is a crucial step in PP construction, since it consists the base for defining the testing activities to satisfy those objectives, and because testing without clear objectives may lead to waste of time and effort. The PP includes also a mapping between the security objectives and security problems to help the evaluator to recognize the relations between the different security objectives and their corresponding security problems. Some security objectives which address some of the security problems mentioned above are shown in Table 3.7. Table 3.8, instead, shows the mapping between the security objectives and the corresponding security problems.

The *security objectives rationale* is usually a short description of how the security objectives will address security problems. It can be either written as a separate subsection or embedded in the mapping table. Having it embedded in the table showing the mapping (security problems → security objectives) makes it much easier to understand. For instance if we take the mapping

Security objective	Description
Objectives to counter Threats	
O.ADMIN_GUIDANCE	The TOE will provide administrators with the necessary information for secure management of the TOE.
O.ADMIN_ROLE	The TOE will provide administrator role to isolate administrative actions
Objectives to enforce OSP	
O.AUDIT_GENERATION	The TOE will provide administrators with the necessary information for secure management of the TOE.
O.ACCESS	The TOE will ensure that users gain only authorized access to it and to resources that it controls.
Objectives to uphold assumptions	
O.PHYSICAL	Those responsible for the TOE must ensure that those parts of the TOE critical to security policy are protected from physical attack which might compromise IT security objectives.
O.INSTALL	Those responsible for the TOE must ensure that the TOE is delivered, installed, managed, and operated in a manner which maintains IT security objectives.

Table 3.7: Examples of Security objectives [18]

Security objective	Security problem
Threats	
O.ADMIN_GUIDANCE	T.ADMIN_ERROR
O.ADMIN_ROLE	T.ADMIN_ROGUE
OSP	
O.AUDIT_GENERATION	P.ACCOUNTABILITY
O.ACCESS	P.AUTHORIZED_USERS
Assumptions	
O.PHYSICAL	A.LOCATE
O.INSTALL	A.MANAGE

Table 3.8: Mapping security objectives to security problem definitions [18]

between the threat O.ADMIN_ROLE and the objective T.ADMIN_ROGUE, the standard document [4] defines the rationale as in Figure 3.6.

The *Security Requirements* section represents the core part of the PP document, that is, the one dealing with the desired security properties. To be able to assess or evaluate the security level of a TOE, a PP needs to define a set of requirements that would allow the evaluator to know which software features should be tested. Compared to the other PP sections, the security requirements section is usually much larger. The reason behind that security requirements need to be described clearly, including all the needed details to avoid any ambiguous interpretation. The CC certification defines two types of

TOE Description. It is important to limit the functionality of administrative roles. If the intentions of an individual in an administrative role become malicious, O.ADMIN_ROLE mitigates this threat by isolating the administrative actions within that role and limiting the functions available to that individual. This objective presumes that separate individuals will be assigned separate distinct roles with no overlap of allowed operations among the roles. Separate roles include an authorized administrator and a cryptographic administrator.

Fig. 3.6: Rationale of the mapping between O.ADMIN_ROLE and T.ADMIN_ROGUE [4]

security requirements: *Security Functional Requirements* (SFRs) and *Security Assurance Requirements* (SARs) .

SFRs define the requirements that the security features of the product under evaluation should satisfy. In other words SFRs specify how the TOE should work to preserve its expected behavior. The CC standard includes a predefined extendable catalogue of security functional requirements "that are known and agreed to be of value by the CC part 2 authors" [8]. However, the SFRs are flexible and can be extended for particular scenarios. The CC authors have divided the set of the SFRs into four hierarchies depicted in Figure 3.7 (i.e., *Classes, Families, Components and Elements*), each one providing more fine-grained security requirements [12]. For instance, in our
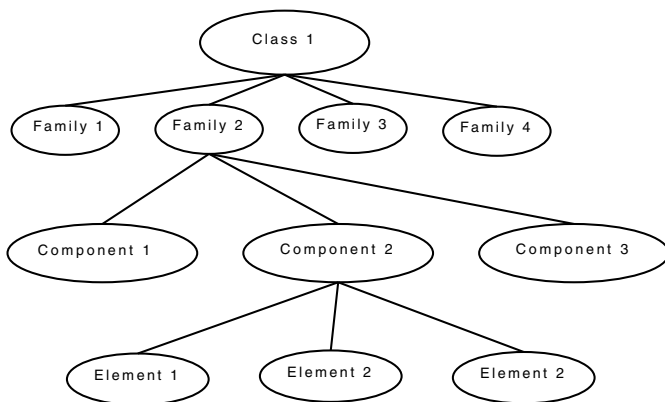
Fig. 3.7: The hierarchical structure of the SFRs

operating system example it is important to limit the functionality of administrative roles. If the intentions of an individual in an administrative role become malicious, O.ADMIN_ROLE mitigates this threat by isolating the administrative actions within that role and limiting the functions available to that individual. This objective presumes that separate individuals will be assigned separate distinct roles with no overlap of allowed operations between

the roles. Separate roles may include an authorized administrator and an encryption administrator, as well as provisions for enforcing *division of labor* between the two. Figure 3.8 shows an example of SFRs.

FAU_GEN.2.1 The TSF shall be able to associate each auditable event with the identity of the user that caused the event.

- Application Note: There are some auditable events which may not be associated with a user, such as failed login attempts. It is acceptable that such events do not include a user identity. In the case of failed login attempts it is also acceptable not to record the attempted identity in cases where that attempted identity could be misdirected authentication data; for example when the user may have been out of sync and typed a password in place of a user identifier.
- Rationale: O.AUDITING calls for individual accountability (i.e., "TOE users") whenever security-relevant actions occur. This component requires every auditable event to be associated with an individual user.

Fig. 3.8: An example of SFRs [4]

In the example in Figure 3.8, the name of the SFRs is represented by the standard notation used by CC, namely (`classes`, `families`, `components` and `elements`). The first letter `F` indicates that this is a `Functional requirement`. The two following letters indicate the requirement class (`AU = Security audit`); the next three letters represent the family name (`GEN = Security audit data generation`); the first digit represents the component number and the second digit indicates the element number. The application note and the rationale provide some details which are specific to this particular PP, to help in interpreting the requirements correctly.

The SARs describe some practical ways to check the effectiveness of the security features of the product under evaluation [22]. The SARs catalogue includes predefined requirements focusing on different phases of the product life cycle such as development, configuration management, testing and so forth. SARs specify the actions deemed necessary to provide enough confidence that the software product will satisfy the security requirements, that is, how to investigate the efficiency of the security functions for the required level of security [8]. Figure 3.9 shows an example of SARs.

Syntactically, SARs follow the same notation standard introduced for SFRs. However, they have an additional letter at the end called the `action element type`. Since the assurance elements are the most fine-grained entities used by the CC , dividing them to even smaller entities may not lead to significant results. For this reason, the standard defines three different action types that identify each of the assurance elements [9].

- *Developer action elements* (letter D): identify the tasks that shall be performed by the developer.

- **ADO_DEL.1.1D** The developer shall document procedures for delivery of the TOE or parts of it to the user.
- **ADO_DEL.1.2D** The developer shall use the delivery procedures.
- **ADO_DEL.1.1C** The delivery documentation shall describe all procedures that are necessary to maintain security when distributing versions of the TOE to a user's site.

Fig. 3.9: An example of SARs [4]

- *Content and presentation of evidence elements* (letter C): describe the required evidence and what the evidence shall show.
- *Evaluator action elements* (letter E): identify the tasks that shall be performed by the evaluator.

Finally the *PP Rationale* is the section where the mapping between security problems and security objectives, and the mapping between SFRs and security objectives are formalized. Also, the PP rationale discusses how threats will be addressed. This is a section where more details could be added to help understand how the TOE shall meet the stated security objectives. Further details can be added concerning SFRs and SARs classes, families, components and elements to help the evaluator to fully understand how the CC components should be applied [9].

3.3.1.2 Security Target

The *Security Target* (ST) is a security specification of a software product. ST contains the security requirements of a given software product, to be achieved by a set of specific security functions. Unlike PPs, STs are implementation dependent, which means that it specifies the implementation details about each SFR. The content of the ST is depicted in Figure 3.10.

The ST is a basis for agreement between the developers, evaluators and, where appropriate, users on the TOE security properties and on the scope of the evaluation. A ST can be derived from a given PP by instantiation; in general, each ST corresponds to a particular PP definition. A ST may then claim conformance to a PP by providing the implementation details concerning the security requirements defined by that PP [14]. Also, ST may augment the requirements derived from the PP. Indeed, there might be cases where there is no PP that matches the security properties of a specific product. In this case, the product developer can still create its own ST without claiming conformance to any PP [14].

An important aspect of ST requirements specification is the definition of the *threats* and *security objectives* of the TOE and its environment. Threats identify situations that could compromise the system assets, while *security*

1. ST introduction

    - ST reference
    - TOE reference
    - TOE overview
    - TOE description

2. Conformance claims

    - CC conformance claim
    - PP claim, Package claim
    - Conformance rationale

3. Security problem definition

    - Threats
    - Organizational security policies
    - Assumptions

4. Security objectives

    - Security objectives for TOE
    - Security objectives for the operational environment
    - Security objectives rationale

5. Extended components definition

    - Extended components definition

6. Security requirements

    - Security functional requirements
    - Security Assurance requirements
    - Security requirements rationale

7. TOE summary specification

    - TOE summary specification

Fig. 3.10: Content of a Security Target

objectives contain all the statements about the intents to counter identified threats and/or satisfy identified organization security policies and assumptions. Based on threats and security objectives, the ST defines the *security requirements* that the TOE security features need to satisfy to achieve the security objectives.

To help establishing an association between the components of the CC and the CC certification process, Figure 3.11 shows where the CC components are used during the CC process. First, software developers need to decide whether their ST will claim conformance to any PP, if yes the specified PP will be included in the certification documents and the ST will be validated against it. Additionally, a set of other documents including design documentation, customer guidance, configuration management and testing must be made
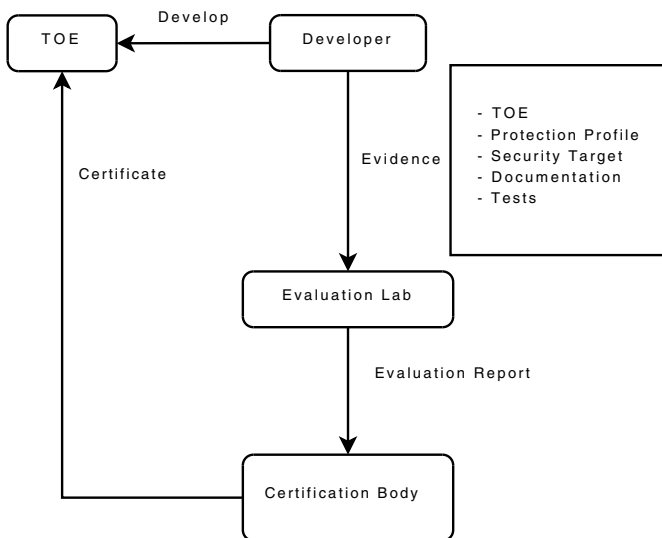
Fig. 3.11: CC process scheme

available to the evaluation body. The role of CC evaluation body is to inspect and analyze all evidence provided by developers.

A fundamental aspect that was taken into consideration when defining the CC is the ability to repeat and reproduce the evaluation results. For this reason, the CC authors have defined an additional document (Common Certification Methodology) that specifies the minimum set of actions to be performed by an evaluator during a CC evaluation [7].

3.3.1.3 Evaluation Assurance Levels

The Common Criteria standard defines seven hierarchical Evaluation Assurance Levels (EAL), which balance the desired level of security and the cost of deploying the corresponding degree of assurance [3]. EAL identifies different sets of security assurance requirements, which are shown in Table 3.9. In case the predefined requirements do not match the level of assurance required, the EAL might be augmented by adding additional assurance requirements.

An example of EALs achieved by operating systems and other software products are given in Figure 3.10.

EAL	Description
EAL1	Functionally tested (black-box testing)
EAL2	Structurally tested
EAL3	Methodologically tested and checked
EAL4	Methodologically designed, tested and reviewed
EAL5	Semiformally designed and tested
EAL6	Semiformally verified design and tested
EAL7	Formally verified design and tested

Table 3.9: Evaluation Assurance Levels

Product	Description
Apple	No evaluations
Linux	EAL 2, for Red Hat Enterprise Linux 3, February 2004
Linux	EAL 3+, for SuSE Linux Enterprise Server V8, Service Pack 3, RC4, January 2004
Solaris	EAL 4, for Solaris 8, April 2003
Solaris	EAL 4, for Trusted Solaris 8, March 2004
Windows	EAL 4+, for Windows 2000 Professional, Server, and Advanced Server with SP3 and Q326886, October 2002

Table 3.10: An example of Evaluation Assurance Levels achieved by software products

3.4 Conclusions

Test-based security certification approaches can provide some confidence (although no certainty) that a software product will preserve its properties of data confidentiality, integrity and availability even under hostile conditions. Many high technology, safety-critical products like aircrafts, include large distributed software systems. In such safety-critical systems, each computational node must be certified to perform its functions safely. As more and more safety-critical and mission-critical software systems communicate with other systems, malicious attempts to subvert those communications multiply, and security concerns become increasingly important. The Common Criteria defines seven Evaluation Assurance Levels (EALs) 1 (low) through 7 (high). The threat level and the value of the information jointly determine the appropriate level of confidence in both the correctness of the security functionality (EAL level) and the extent of the security functionality, specified in a Protection Profile. The consequences of some information being compromised may range from negligible effects to severe damage.

It is important to remark that security certification standards are strictly related to safety ones. The DO-178B standard for safety, like the Common Criteria standard for security, is mostly concerned with program correctness. The difference lies in the fact that DO-178B addresses post-certification quality assurance, while the Common Criteria covers topics such as vulnerability, user documentation and software delivery. DO-178B defines Level A through Level E.; there is no safety impact if Level E software fails, while if Level A

software fails, the safety impact is catastrophic. Another characteristic common to both safety and security is that earning certification is much more difficult, risky, and therefore expensive if the certification was not an original design goal. This occurs when certification requirements are extended as the result of revised policies or regulations. Certifications are expensive, although open source tools that analyze source code for faults are becoming available.

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.
2. E. Mate Bacic. The canadian trusted computer product evaluation criteria. In *Proc. of the Sixth Annual Computer Security Applications Conference*, Tucson, AZ, USA, December 1990.
3. K. Caplan and J.L. Sanders. Building an international security standard. *IEEE Educational Activities Department*, 22(3):29–34, March 1999.
4. Information Assurance Directorate. *US Government Protection Profile for Multilevel Operating Systems in Medium Robustness Environments*, 2007.
5. A.J. Dix, J.E. Finlay, G.D. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, 2004.
6. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model*, 2006.
7. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Evaluation methodology*, 2007.
8. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components*, 2007.
9. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components*, 2007.
10. P.G. Frankl, R.G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transaction on Software Engineering*, 24(8):586–601, August 1998.
11. P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
12. D.S. Herrmann. *Using the Common Criteria for IT security evaluation*. Auerbach Publications, 2002.
13. W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.
14. ISO/IEC. *Guide for the production of Protection Profiles and Security Targets*, 2004.
15. C. Jahl. The information technology security evaluation criteria. In *Proc. of the 13th International Conference on Software Engineering*, Austin, TX, USA, May 1991.
16. Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1(2):308–320, 1976.
17. USA Department of Defense. *DEPARTMENT OF DEFENSE TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA*. USA Department of Defence, 1985.

18. Information Systems Security Organization. *Controlled Access Protection Profile version 1.d*, 1999.
19. M. Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley.
20. S.L. Pfleeger and J.D. Palmer. Software estimation for object oriented systems. In *Fall International Function Point Users Group Conference*, San Antonio, Texas, October 1990.
21. D. Russell and G.T. Gangemi. *Computer Security Basics*. O'REILLY, 1991.
22. K.S. Shankar, O. Kirch, and E. Ratliff. Achieving capp/eal3+ security certification for linux. In *Proc. of the Linux Symposium*, volume 2, page 18, 2004.
23. H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, April 1996.

Chapter 4
Formal methods for software verification

Abstract The growing importance of software in every aspect of our life has fostered the development of techniques aimed at certifying that a given software product has a particular property. This is especially important in critical application areas such as health care and telecommunications, where software security certification can improve a software product's appeal and reduce users and adopters concern over the risks created by software faults. In this chapter, we shall deal with a wide range of formal and semi-formal techniques used for verifying software systems' reliability, safety and security properties. A central notion is the one of a *certificate* i.e. a metadata item containing all information necessary for an independent assessment of all properties claimed for a software artifact. Here we focus on the notion of *model-based certification*, that is, on providing formal proofs that an abstract model (e.g., a set of logic formulas, or a formal computational model, such as a finite state automaton), representing a software system, has a particular property. We start by laying out some of the work that has been done in the context of formal method verification, including in particular the areas of model checking, static analysis, and security-by-contract. Then, we go on discuss the formal methods that have been used for analyzing/certifying large-scale, C-based open source software.

4.1 Introduction

Software systems are trending towards increased size and increasingly complex architectures. This is making it more and more difficult to achieve full assurance of a software product's non-functional properties by means of test-based techniques alone (see Chapter 3). An important alternative option is to use *formal methods*, that is, techniques based on logics, set theory and algebra for the specification of software systems models and the verification of the models' properties. The use of formal methods has become widespread,

especially during the early phases of the development process. Indeed, an abstract model of a software system can be used to understand if the software under development satisfies a given set of functional requirements and guarantees certain non-functional properties. Also, the increasing number of reports of security-relevant faults in software shows that the problem of verifying security-related properties cannot be ignored, especially in the development of high-integrity systems where safety and security are paramount.

There are several case studies proving the applicability of formal methods to security certification [17, 30]. Some practitioners are, however, still reluctant to adopt formal methods. This reluctance is mainly due to a lack of theoretical understanding, and to the misconceived perception that formal techniques are difficult to learn and apply. The detection and the prevention of faults is indeed one of the main motivations for using formal methods. Verifying a formal system specification can help to detect many design flaws; furthermore, if the specification is given in an executable language, it may also be exploited to simulate the execution of the system, making the verification of properties easier (*early prototyping*). Over the last few years software verification using formal methods has become an active research area. Special attention is being given to the verification of concurrent and parallel programs, in which testing often fails to find faults that are revealed only through the use of very specific test cases or timing windows. However, the problem of using models for checking a software product's memory-related non-functional properties, also known as *pointer analysis*, remains to be solved. To understand why, let us consider a slight variation of the `digitcount` function we introduced in Chapter 3.

```
int digitcount(char* s)
{
 int digit = 0;
 int i;
 for (i = 1; *(s+i) != '\0'; i++)
     if ('0' <= *(s+i) <= '9')
         digit++;
 return digit;
}
```

While this coding of `digitcount` is functionally equivalent to the one of Chapter 3, switching the type of the input argument from the array of characters of the original version (whose maximum size can be defined in the calling program) to a pointer to char - that is, an address pointing to a memory area whose maximum size is not known - has interesting effects. Assuming that the calling program had originally limited the size of arrays passed to `digitcount` to 256, the number of states (defined, as before, as the possible contents of the function's local variables) of our new version of the sample function may have increased dramatically as a result of this alternative coding. In this example, the state space may change in size but remains

anyway finite; in general, because of the dynamic and unbounded nature of
C memory handling primitives (including allocations, deallocations, referenc-
ing, dereferencing etc.), models representing C programs with pointers must
take into account an infinite state space. With infinite state spaces,[1] exhaus-
tive searches are no longer possible, and checking the properties of models
involving recursive pointer types may lead to undecidability [21].

4.2 Formal methods for software verification

Let us start with a survey on the categories of formal methods that are of
interest for security certification. Program verification techniques fall into
three broad categories: The first category involves non-formal or partially
formal methods such as testing, which we have discussed in some detail in
Chapter 3; the second category, known as *model checking*, involves formal
verification of software systems with respect to specifications expressed in a
logical framework, either using state space analysis or theorem proving; the
third category includes classic *static program analysis* techniques. We shall
now provide a brief introduction to both these categories.

4.2.1 Model Checking

Model checking [12] is a formal verification approach for detecting behavioral
anomalies (including safety, reliability and security-related ones) of software
systems based on suitable models of such systems. Model checking produces
valuable results, uncovering software faults that might otherwise go unde-
tected. Sometimes it has been extremely successful: in 1998, the SPIN model-
checker was used to verify the plan execution module in NASA's DEEP
SPACE 1 mission and discovered five previously unknown concurrency er-
rors. However, model checking is not a panacea. Indeed, there are still several
barriers to its successful integration into software development processes. In
particular, model checking is hamstrung by scalability issues; also, there is
still a gap between model checking concepts and notations and the models
used by engineers to design large-scale systems.

Let us focus on the scalability problem, which is a major obstacle to using
model checking to verify (and certify) the security properties of software
products. Methods and tools to aid design and analysis of concurrent and
distributed software are often based on some form of a state reachability
analysis, which has the advantage of being conceptually simple. Basically,
the verifier states the non-functional properties she would like the program

---

[1] In some cases, there are ways of representing infinite state spaces finitely, but this
would take us well outside the scope of this book.

to possess; then, by means of a model checker tool, she searches the program state space looking for *error states*, where the specified properties do not hold. If some error states are detected, the verifier removes the faults which made them reachable, and repeats the procedure. The reader may detect a certain likeness to the testing process: indeed, one can never be sure that all error states have been eliminated. Usually, state analysis is not performed directly on the code; rather, one represents the program to be verified as a state transition system, where states are values of variables, and transitions are the instructions of the program. Fig. 4.1 shows the transition system for the instruction `for (int i = 1; *(s + i) != 'n0'; i++)` in our `digitcount` example.
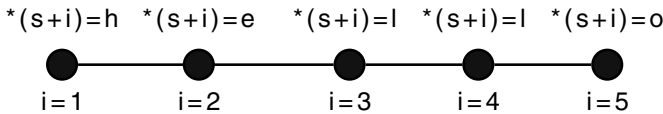

Fig. 4.1: Transition system of a `digicount` instruction

It is easy to see where the problem lies: the number of states may proliferate even for relatively simple programs, making the model checking approach computationally very expensive.

However, space search algorithms allowing more than 10^{20} states have been available for several years now, and today's model-checkers can easily manage millions of state variables. Also, a number of techniques have been developed to prevent state space explosion and to enable formal verification of realistic programs and designs. Here, we will only recall an important method in state space reduction, namely *abstraction*. Abstraction techniques reduce the state space of a software system by mapping the set of states of the actual system into an abstract, and much smaller, set of states in a way that preserves all relevant system behaviors. *Predicate abstraction* [20] is one of the most popular methods for systematic reduction of program state-spaces. It abstracts program data by only keeping track of certain predicates on the data, rather than of the data themselves. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting Boolean program is an over-approximation of the original program.

In practice, the verifier starts with a coarse abstraction of the system to be checked, and checks this abstraction for errors. If checking reports an unrealistic error-trace, the error-trace is used to refine the abstract program, and the process proceeds iteratively until no spurious error traces can be found. The actual steps of this iterative process follow an abstract-verify-refine paradigm.

Automated theorem proving [15] is a well-developed subfield of automated reasoning that aims at proving mathematical theorems by means of a computer program. Commercial use of automated theorem proving is mostly concentrated in integrated circuit design and verification, although many significant problems have been solved using theorem proving. Some of the fields where theorem proving has been successfully used are mathematics, software creation and verification, and knowledge based systems.

When assessing the correctness of the program, two distinct approaches using properties are in use, namely *pre/post-condition* and *invariant assertion*. Large programs should specify pre- and post-conditions for every one of their major procedures.[2] A pre-condition is a logical formula whose value is either the Boolean *true* or *false*. It is a statement that a given routine should not be entered unless its pre-condition is guaranteed[3]. Obviously, a pre-condition does not involve variables that are local to the procedure; rather, it may involve relevant global variables, and the procedure's input arguments.

Ideally, a pre-condition is a Boolean expression, possibly using the for-all ($\forall$) or there-exists ($\exists$) quantifiers.

The post-condition of a procedure is also a Boolean formula which describes the outcome of the procedure itself. The results described in a post-condition are accomplished only if the procedure is called while its pre-condition is satisfied. The post-condition talks about all relevant global variables and the input arguments and relates them to the output results. Like a pre-condition, a post-condition is a strictly Boolean expression, possibly using the for-all ($\forall$) or there-exists ($\exists$) quantifiers. Pre/post-condition -based approaches to software verification formulate the software correctness problem as checking the relationship between the pre-condition Boolean formula that is assumed to hold at the beginning of program execution and the post-condition formula that should hold at the end of program execution.

The pre/post-condition based verification process goes as follows: again, the verifier states the non-functional properties she would like the program to possess. Then, she formulates relevant pre-conditions and post-conditions known to hold for smaller parts of the program, and uses these properties and additional axioms to derive the desired non-functional properties.

One might legitimately ask where do these axioms come from and how pre- and post-conditions can be formulated. The prototype of each procedure - or better, its *signature* - is an important source of pre- and post-conditions. When a procedure declares a formal parameter as passed by value, this means that the actual argument will remain unchanged at the end of the procedure: a basic post-condition. Also, procedures may modify global variables whose names are not listed as actual arguments. It is possible to generate pre- and

---

[2] In this section we use the word "procedure" as a generic term for both procedures and functions. A function returns a value to the caller but has no side effects on the caller local memory, whereas the purpose of a procedure is exactly to have such a side-effect.

[3] If the procedure is entered anyway, its behavior is unpredictable

post-conditions by stating the global variables' values before and after a call to the procedure. As far as the axioms are concerned, they simply express typical pre-conditions and post-conditions of the programming language's instructions. To further clarify this issue, let us use the elegant notation of *Hoare triplets* [18], as follows: $\{P\}S\{Q\}$, where P is a precondition, S a statement and Q a postcondition.

The meaning of a Hoare triplet is the one suggested by intuition: if P holds before S is executed, then after the execution of S is executed, Q holds. For example, $\{a > b\}$ `while (a > b) a − −`$\{a = b\}$ $\{(a, b : int)\}$ means that if the integer `a` is greater than `b`, the loop `while (a > b) a − −` will make them equal. Let us now state the simplest possible axiom, the one corresponding to the assignment instruction:

$$\{Q(e/x)\}x = e\{Q\} \tag{4.1}$$

The axiom states the (rather intuitive) fact that if we have a pre-condition which is true if e is substituted for x, and we execute the assignment `x = e`, the same formula will hold as a post-condition. For example, if we want to prove that $\{i = 0\}$ `i = i + 1` $\{i > 0\}$, we apply the assignment axiom to obtain $\{i + 1 = 1\}$ `i = i + 1`$\{i + 1 > 1\}$, from which the thesis is derived by simple arithmetics.

Verification becomes more difficult when we consider loops, a basic control structure in nearly all programming languages. Loops are executed repeatedly, and each iteration may involve a different set of pre- and post-conditions. Therefore, verification needs to focus on assertions which remain constant between iterations rather than on pre- and post-conditions. These assertions are known as *loop invariants*, and remain true throughout the loop. To clarify this concept, let us consider the code fragment below, which computes the minimum in an array of positive integers:

```
min = 0;
int j;
for (j = 0; j <= n; j++)
{
 if(s[j] < min)
    min=s[j];
}
```

The invariant of this loop is that, at any iteration, $s[k] < min$ for $k = 0, 1, \ldots, j$.[4] To prove that an assertion of interest still holds after a loop terminates, the verifier must start by proving that the loop does indeed terminate. The verifier needs to identify an invariant and use it together with axioms to derive the theorem that the desired assertion is true after last iteration. It is interesting to remark that identifying pre-conditions, post-

---

[4] The invariant holds even for $j = k = 0$, since the array is made of positive integers.

conditions and invariants is useful even if the formal verification process is not carried out. Consider the following code:

```
int digitcount (char s[])
{
 if (!precond(s))
     return -1;
 /* .. rest of the digitcount code.. */
}
```

Here, the function `digitcount` is doing something unexpected: it is checking its own precondition. If an input parameter violates the pre-condition, that is, `precond(s)` is false, the function returns a value that is outside its expected range of return values, in this example -1. This precaution increases the robustness of the function, preventing error conditions due to malformed inputs. [5]

A key difference between the model checking approach to software verification and the theorem proving one we just explained, is that theorem provers do not need to exhaustively visit the whole program state space in order to verify properties, since the constraints are on states and not on instances of states. Thus, theorem provers can reason about infinite state spaces and state spaces involving complex datatypes and recursion.

A major drawback of theorem provers is that they require a great deal of user expertise and effort: although theorem provers are supposed to support fully automated analysis, only in restricted cases is an acceptable level of automation is provided. This is mainly due to the fact that, depending on the underlying logic, the problem of deciding the validity of a theorem varies from trivial to impossible. For the case of propositional logic, the problem is decidable but *NP*-complete, and hence only exponential-time algorithms are believed to exist for general proofs. For the first order predicate calculus, the theorem prover could even end up in non-termination. In practice, the theoretical results require a human to be in the loop, to derive non-trivial theorems and to guide the theorem prover in its search for a proof. Despite these theoretical limits, practical theorem provers can solve many hard software verification problems.

4.2.2 Static Analysis

Static program analysis aims to retrieve valuable information about a program by analyzing its code. Static analysis of programs is a proven technology

---

[5] Strictly speaking, here `precond(x)` is not a proper precondition of `digitcount`, because `digitcount` is executed whether `precond(x)` is true or not; but the programmer can now be confident that the "real" `digitcount` code will be executed only if `precond(s)` evaluates to true.

in the context of the implementation and optimization of compilers and interpreters. The Syntactic analysis carried out by compilers is a first step in this direction: many faults due to typing mistakes can be tracked by modifying the C syntax specification on which the compiler is based to generate appropriate warning messages. Let us consider the following code fragment:

```
int digit = 0;
if(*s = '\0')
    return digit;
else
    /* rest of the digitcount code */
```

It is easy to see that this code always returns 0; this is due to the programmer erroneously using an assignment instead of a comparison operator. The C syntax analyzer can be modified to generate a warning whenever an assignment statement appears in a conditional expression (where a comparison would be expected). In the modified syntax specification, the compiler action upon detecting an assignment in a conditional expression is specified as "print a warning message".

In recent years static analysis techniques have been applied to novel areas such as software validation, software re-engineering, and verification of computer and network security. Giving a way of statically verifying a security property has, in principle, the advantage of making the checking of the property more efficient; moreover it allows the writing of programs which are secure-by-construction (e.g., when the performed analysis is proved to imply some behavioural security properties). As most non-trivial properties of the run-time behaviour of a program are either undecidable or *NP*-hard, it is not possible to detect them accurately, and some form of approximation is needed. In general, we expect static analysis to produce a possibly larger set of possibilities than what will ever happen during execution of the program. From a practical perspective, however, static analysis is not a replacement for testing nor can it completely eliminate manual review. Static analysis is effective only when operating on the source code of programs, whereas code consumers typically deal with binary code which makes it difficult (if not impossible) for them to statically verify whether the code satisfies their policy. This is not the case with open source, though; and this remark alone would be sufficient to make static analysis an important topic for our purposes. But there is more: static analysis has a proven record of effectiveness for dealing with security-related faults. Often attackers do not even bother to find new faults but try and exploit well-known ones, such as buffer overflow, which could have been detected and removed using static code analysis. When performing static analysis of a program, the verifier uses tools like `lint` and `splint`, which perform error checking of C source, to scan the program's source code for various vulnerabilities. Such scanning involves two steps: *control flow* and *data flow* analysis.

Control Flow Analysis (CFA) [24] is an application of Abstract Interpretation technologies. The purpose of CFA is to statically predict safe and computable approximations to the dynamic behaviour of programs. The approach is related to Data Flow Analysis and can be seen as an auxiliary analysis needed to establish the information about the intra- and inter-procedural flow of control assumed when specifying the familiar equations of data flow analysis. It can be expressed using different formulations such as the constraint-based formalism popular for the analysis of functional and object-oriented languages, or the Flow Logic style. Flow Logic is an approach to static analysis that separates the specification of when a solution proposed by analysis is acceptable form the actual computation of the analysis information. By predicting the behaviour of a software system, it leads to positive information even when the system under evaluation does not satisfy the property of interest, whereas the type-system approach is binary-prescriptive (a system is either accepted or discarded). Moreover, it is a semantics-based approach - meaning that the information obtained from the analysis can be proved correct with respect to the semantics of the programming language, that is, the result reflects an appropriate aspect of the program's dynamic behaviour. The formalization of Control Flow Analysis is due to Shivers in [29], where the analysis is developed in the context of functional languages. The CFA technique has been used extensively in the optimization of compilers, but over the last few years it has also been used for verification purposes. In the case of the Flow Logic approach, there is an extensive literature, showing how it has been specified for a variety of programming language paradigms. Moreover, this technique has been used to verify non-trivial security properties, such as stack inspection and a store authorization in a broadcast process algebra. To fix our ideas, let us consider the simplest case [16], where control flow analysis involves setting up a control flow graph, i.e., a *Directed Acyclic Graph* (DAG) which represents the program's control flow. Each node in the DAG corresponds to a program instruction and the edges from one node to another represent the possible flow of control. Control flow graphs for basic instructions are shown in Figure 4.2.

Control flow graphs for more complex statements can be constructed inductively from the control flow graphs of simple statements. Function calls are also represented as nodes in a control flow graph. When traversing a program's control flow graph if one comes across a node representing a function call then the control flow graph of the corresponding function (if it exists) is also traversed.

Data flow analysis determines the different properties a variable can have through taking different paths in the program, in order to identify for potential faults. The nodes in the CFG are used to store information about certain properties of data such as initialization of variables, references to variables, and so forth.

For example, let us consider the following code fragment:

```
int digitcount (char* s)
```

(a) S1, S2

(b) If E then S

(c) If E then S1 else S2
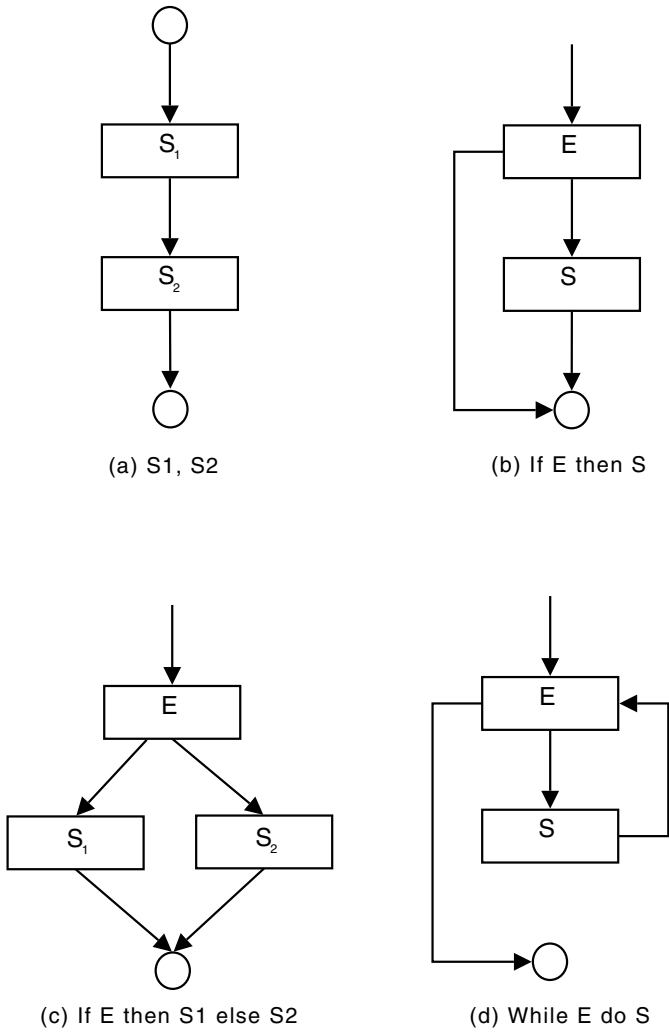
(d) While E do S

Fig. 4.2: Control flow graphs for basic instructions

```
{
 int digit;
 if (*s != '\0')
 {
  digit =0;
  for (int i = 1; *(s+i) !=' \0'; i++)
      if ('0' <= *(s+i) <= '9')
          digit++;
```

```
  }
  return digit;
}
```

Though the variable `digit` is initialized within the `if` statement, it is not always guaranteed that `digit` will actually be initialized before being accessed (when $*s ==' n0'$, `digit` is uninitialized). Static analysis will point out that variable `digit` is not initialized in one of the paths, a fact that may escape testing unless the void string is used as a test case. Analysis of the control flow graph can also be used for detecting memory-related faults like dereferencing uninitialized pointers, forgetting to free allocated memory, and so on.

To carry out this analysis, the verifier needs to use flow analysis and build the control flow graph, adding memory-related information to nodes. For each variable, such information will say whether at the execution of the instruction corresponding to the node the variable is initialized or not (remember that the variable will be considered as possibly not being initialized if there exists at least a path through the graph on which the initialization is skipped). As a formal notation, we can associate to each node S on the control graph a set (let's call it *NonInit*) consisting all variables for which a path exists which terminates in S and does not include initialization. Similarly, we can associate to S other sets such as *NullPointer*, including all pointers whose value may be null at S. This way, the instruction corresponding to S can be checked w.r.t. the sets associated to it. For instance if S dereferences a pointer which is in *NullPointer*, or accesses a variable belonging *NonInit*, the verifier receives a warning.

A major drawback of these analysis techniques is that they may generate *false positives*. In general, the verifier has no idea of the semantics of the program, and may consider faulty some code that the programmer has written intentionally.

4.2.3 Untrusted code

Significant steps forward have been made in the use of software from sources that are not fully trusted, or in the usage of the same software in different platforms or environments. The lack of OS-level support for safe execution of untrusted code has motivated a number of researchers to develop alternative approaches. The problem of untrusted binary code is solved when using certified code (both proof-carrying code (PCC) [22, 23] and model-carrying code (MCC) [27, 28]), which is a general mechanism for enforcing security properties (see Chapter 9).

In this paradigm, untrusted mobile code carries annotations that allow a host to verify its trustworthiness. Before running the guest software, the host checks its annotations and proves that they imply the host's security policy.

Despite the flexibility of this scheme, so far, compilers that generate certified
code have focused on simple type safety properties rather than more general
security policies. A major difficulty is that automated theorem provers are not
powerful enough to infer properties of arbitrary programs and constructing
proofs by hand is prohibitively expensive. Moreover the security policy needs
to be shared and known a priori by both code producer and consumer. Unable
to prove security properties statically, real-world security systems such as the
Java Virtual Machine (JVM) have fallen back on run-time checking. Dynamic
security checks are scattered throughout the Java libraries and are intended
to ensure that applets do not access protected resources inappropriately. This
situation is unsatisfactory for a number of reasons: *(i)* dynamic checks are
not exhaustive; *(ii)* tests rely on the implementation of monitors, which are
error-prone; and *(iii)* system execution is delayed during the execution of the
monitor.

4.2.4 Security by contract

The security-by-contract mechanism [13, 14] draws ideas from both the above
approaches: firstly, it distinguishes between the notion of *contract*, containing
a description of the relevant features and semantics of the code, and the
policy, describing the contractual requirements of the platform/environment
where the code is supposed to be run; secondly, it verifies at run-time if the
contract and the policy match. This approach derives from the "design by
contract" idea used to design computer software. It prescribes that software
designers should define precise verifiable interface specifications for software
components based upon the theory of abstract data types and the conceptual
metaphor of a business contract.

The key idea of security-by-contract derives from the growing diffusion of
mobile computing and nomadic devices. Since the demand for mobile ser-
vices, which are dynamically downloaded by users carrying mobile devices, is
growing, there is a need for mechanisms for certifying security properties of
the downloaded application and for certifying how the application interacts
with the host device. The *contract* accompanying the application specifies
the relevant security actions while the *policy* defines the host's requirements
i.e. the expected behaviour of applications when executed on the platform.
Contracts and policies are defined as a list of disjoint rules as follows.

```
<RULE> :=
    SCOPE [ OBJECT <class> |
       SESSION |
       MULTISESSION ]
       RULEID <identifier>
       <formal specification>
```

where:

- *SCOPE* defines at which level the specified contract will be applied [13]: *(i) object,* the obligation must be fulfilled by objects of a given type, *(ii) session,* the obligation must be fulfilled by each run of the application, *(iii) multisession,* the obligation must be fulfilled by all runs of the application;
- *RULEID* identifies the area at which the contract applies, as for instance, *files* or *connections*;
- *<formal specification>* provides a rigorous and not ambiguous definition of the rule semantics based on different techniques, such as, standard process algebra, security automata, Petri Nets and the like.

Based on contract and policy definition, the matching algorithm verifies if contracts and policies are compatible. A matching will succeed if for each behaviour happening during the code execution both contract and policy are satisfied. In Table 4.1, we present a simple example of a contract/policy matching taken from [13].

Contract/Policy Rule	Object can use one type of connection only	Object can use every type of connection only
Object can use HTTP connection only	X	X
Object can use HTTP and SMS connections		X

Table 4.1: Contract/Policy Matching.

The matching algorithm is defined in a generic way, that is, independently from the formal model used for specifying the rules. In [13] an example of how the matching algorithm can be used with rules specified as Finite State Automaton (FSA) is also provided. Differently from the model checking techniques introduced in Section 4.3.2, which use finite state automaton to define syntactic patterns the program should not contain, contracts/policies define the expected behaviour of the application.

4.3 Formal Methods for Error Detection in OS C-based Software

As we have seen in the previous sections, formal methods have been introduced in the past to answer a simple question: "What should the code do?'. In other words, formal methods were aimed at specifying some functional or non-functional properties the software should possess. Although the response to this question may seem trivial when looking at a toy function consisting of a few lines of code, it is much more difficult to answer in the case of huge software products (million of *Lines Of Code* (LOC)). In the previous sections, we

outlined how formal methods can be used to compare what the code should do, with what the code actually does.

The scenario described in this section is even more critical, because we need to consider open source software. The application of formal methods to open source software is somewhat difficult and it has sometimes been considered as a wrong choice. The open source paradigm is based on a cooperative community-based code development, where code changes rapidly over time and unambiguous specifications may simply not be available (see Chapter 5). The code itself, in fact, is considered the first-line specification of an open source system. This explains the problems faced by practitioners trying to apply formal methods to huge undocumented open source software, such as Linux. Usually open source software is not developed from a stable specification and is based on programming languages which do not readily support formal methods, like the C language used for coding Linux.

4.3.1 Static Analysis for C code verification

Much effort has been put into designing and implementing static analysis techniques for the verification of security-critical software. The need for solution to the problem of finding potential vulnerabilities is especially acute in the context of security-critical software written in C. As we have seen, the C language is inherently unsafe, since the responsibility for checking the safety of array and pointer references is entirely left to the programmer. Programmers are also responsible for checking buffer overflows. This scenario is even more poignant in the context of an open source development community, where different developers with heterogeneous skills and profiles contribute to the software.

Software faults spotlighted in the last few years, the ones related to buffer overflows have been the most frequently exploited. A notorious attack (the so-called Morris worm) in November 1988, which infected about 10% of all the computers connected to the Internet, exploited a buffer overflow in the *finger daemon* of Sun 3 systems and VAX computers running variants of Berkeley UNIX. This attack is often quoted as the first one that caused a widespread infection, captured the attention of the world, resisted expert analysis, and finally resulted in FBI investigations and legal actions. The Morris worm had at least one positive effect: it increased the awareness of the software industry about the dangers produced by Internet-based attacks. Also, as a result of the Morris worm, the Carnegie Mellon Computer Emergency Response Team (CERT) [8] was formed. Currently, the CERT institution represents the main reporting center for Internet security problems.

Wagner et al. [31] describe an approach based on static analysis to detect buffer overflow vulnerabilities. They start from the assumption that C is insecure and developers, including expert ones, are themselves sources of

vulnerabilities. Their solution applies static analysis to identify and fix security flaws before these can be exploited by a malicious adversary. In particular, the problem of detecting a buffer overflow is modeled as an integer range constraint problem, which is solved by means of an algorithm based on graph theoretic techniques. Also the authors focus on balancing precision and scalability. The trade-off between precision and scalability introduces some imprecision in the detection software, causing the identification of wrong vulnerabilities (false positives) and the non-identification of real ones (false negatives).

Wagner's solution is based on two major ideas: *(i)* since most buffer overflows happen in string buffers, C strings are modeled as an abstract data type; *(ii)* buffers are modeled as a pair (as,l) where as is the allocated size for the string buffer, and l is the length, that is, the number of bytes used.

In summary, the authors provide a conceptual framework modeling string operations as integer range constraints and then solving the constraint system. The implementation of the framework is achieved via three main steps, which are described below.

Constraint language definition. This step carries out the definition of a language of constraints to model string operations. To this end, the concepts of *range*, *range closure*, and arithmetic operations over ranges are introduced. An integer range expression is defined as:

$$e ::= v \| n \| n \times v \| e + e \| e - e \| max(e \cdots e) \| min(e \cdots e)$$

where n is an integer and v is a set of range variables. From the range expression a *range constraint* is then defined as $e \subseteq v$ and an assignment as $\alpha : v \longmapsto \alpha(v) \subseteq Z^\infty$. An assignment α satisfies system of constraints if all the assertions are verified when the variables are replaced with the corresponding values $\alpha(v)$ in the assignment.

Constraint generation. In this step, after parsing the source code and traversing the obtained parse tree, a system of integer range constraints is generated. Each integer variable is associated with a range variable, whereas a string variable is associated with two variables (the string's allocated size and the actual string length) plus a safety property $len(s) \leq alloc(s)$, where $len(s)$ includes the terminating '\0'. For each statement an integer range constraint is then generated. The safety property for each string will be checked later.

Constraint resolution. Finally, an algorithm is used to find a bounding box solution to the system of constraints defined by the previous steps. This is guided by the fact that a program with k variables generates a statespace Z^k where the i-th component is the value of the i-th variable. The program execution is modeled as a path and the goal becomes to find a minimal bounding box including all possible paths in the k-dim space. To this end a graph is built where each node represents a variable and an edge between two nodes in the graph represents a constraint involving the two variables.

The constraint solver works by propagating information along the paths
and finding a solution to the constraint system.

Although this technique provides a way to detect buffer overflow, it cannot
handle C pointers and aliasing. Other researchers working in static analysis
have focused on pointer analysis [1, 19] (see also Chapter 3). In order to
understand the notion of a pointer-related software fault, let us consider the
`memcpy` system call, whose prototype is:

`void memcpy(void dest, const void source, size_t n)`.

`memcpy` copies the content of the memory pointed to by `source` (an address
within the caller's address space) to the area, again in the caller's address
space, pointed to by `dest`.[6]

Intuition suggests that a program failure may occur if the `dest` address
passed to `memcpy` is invalid. This type of pointer-related faults are known
to be difficult to detect through testing alone; even static analysis tends to
perform rather poorly, because pointer-related faults concern specific pointer
values rather than its type. Of course, one could wrap `memcpy` to check the
destination address before calling it; but this might introduce a non-negligible
overhead. The problem becomes more intricate if *kernel pointers* are involved
[19], because user space and kernel space addresses cannot be mixed without
undesirable results being produced. Let us consider the function

```
int var;
void getint(int *buf)
{
 memcpy(buf, &var, sizeof(var))
}
```

which uses `memcpy` to copy the content of the variable `var` into the buffer
pointed to by `buf`. Let us assume that some malicious code initializes `buf`
with a value corresponding to some user space address before calling `getint`.
If the kernel blindly executes the copy using `buf` as the destination, a kernel
failure may occur. For instance, if the destination address were a typical
user space address like `0xbf824e60`, it would simply not be in the range of
kernel space addresses (which start at `0xc0000000` and trying to write to it
within the kernel would provoke a kernel `oops`. If the kernel were configured
to panic on `oops`, then the machine would crash. At this point, the reader
might object that `memcpy` is surely coded in such a way to prevent this attack.
Unfortunately, this is not the case. Here is the implementation of `memcpy` in
`linux − 2.6.24.2/lib/string.c`:[7]

---

[6] The behavior of `memcpy` is undefined when destination and source overlap

[7] Note that this implementation handles overlapping source and destination areas
correctly, provided the target address is below the source address.

```
/**
 * memcpy - Copy one area of memory to another
 * @dest: Where to copy to
 * @src: Where to copy from
 * @count: The size of the area.
 *
 * You should not use this function to access IO
 * space, use memcpy_toio() or memcpy_fromio()
 * instead.
 */
void *memcpy(void *dest, const void *src,
            size_t count)
{
 char *tmp = dest;
 const char *s = src;

 while (count--)
        *tmp++ = *s++;
 return dest;
}
EXPORT_SYMBOL(memcpy);
```

The memcpy implementation was kept as simple as possible, and for a good reason: one cannot risk memcpy going to sleep, as it could happen if memcpy was coded in a more sophisticated way.

Since memcpy includes no run-time checks for pointer-related faults, we can only hope that *a priori* pointer analysis can be used to pinpoint the fact that getint code can be unsafe. This can be done by annotating the pointer type declarations with additional information supporting program analysis. In [19], some *qualifier*s of pointer types are used to highlight pointers which contain kernel addresses. The code calling getint can be annotated using two qualifiers user and kernel as follows:

```
int memcpy(void * kernel to, void * kernel from,
          int len);
int var;
void getint(int * user buf)
{
 memcpy(buf, &var, sizeof(var))
}
```

When the above code is analyzed, the analyzer notices that getint receives a user pointer buf, which is then passed to memcpy as a first parameter. A type error is then raised and the potential fault identified. Of course, an annotation-based technique like this one will generate some false positives, i.e. cases in which the function was purposefully designed to handle both kernel and user space addresses. However, the interesting experimental results

reported by [19] show that the analysis of Linux kernel 2.4.20 and 2.4.23 has identified 17 previously unknown faults due to mixing user space and kernel space addresses.

Faults like writes via unchecked pointer dereferences are often exploited by malicious code. A classic attack (often called *stack smashing* [31]) uses unchecked string copy to cause a buffer overflow.

We have described this attack in Chapter 3; however, to recall this point, let us consider the following C procedure that uses a pointer to copy an input string into a buffer stored on the stack, incrementing the pointer after copying each character without checking whether the pointer is past the end of the buffer.

```
void bufcopy(char *src)
{
 char buf[256];
 char *dst = buf;
 do
   *dst = *src;
   dst++; src++;
 while (*src != "\0")
}
```

By providing a string longer than 256, an attacker can cause the above procedure to write after the end of the buffer, overwriting other locations on the stack, including the procedure's own return address. In Chapter 3 we have seen how crafting the input string, the attacker can replace the procedure's return address with the address of malicious code stored, say, in an environment variable, so that when the procedure returns, control is transferred to the attacker's code. Context-sensitive *pointer analysis* is used to detect faults due to lack of bounds checking like the one above [1]. Two types of pointer analysis have been defined: *(i) CONServative pointer analysis* (CONS), which is suitable for C programs following the C99 standard and *(ii) Practical C Pointer* (PCP), which imposes additional restrictions that make it suitable also for programs that do not follow the C99 standard [7].

PCP includes several assumptions that model typical C usage. First of all, PCP allows arithmetic applied to pointers of an array, such as buf + + in the example above, only if the result points to another element of the same array. When pointers to user-defined struct types are used, PCP applies the notion of *structural equivalence*: two user defined types are structurally equivalent if their physical layout is exactly the same. PCP allows assignments (such as ∗dst = ∗ src) and type casts (such as dst = (atype ∗)src) only between structurally equivalent types. For this reason, PCP has been shown to provide a better accuracy in detection of format string vulnerabilities; also, pointer analysis substantially reduces the overhead produced by dynamic string-buffer overflow tools (30%-100%) [1].

More advanced techniques mark all information coming from the outside world as *tainted*. A potentially vulnerable procedure should be written to raise an error if passed a tainted parameter. Chen and Wagner [10] introduced a static analysis technique that can find taint violations. The provided solution has been tested using the Debian 3.1 Linux distribution. The experiments considered the 66% of Debian packages and found 1533 format string taint warnings, 75% of which are real faults.

Static analysis can be complemented with run-time techniques, such as white lists of memory addresses pointers are allowed to contain. An instruction modifying the value of a pointer can only be executed within a procedure which checks whether the modified value will be in the white list. Ringenburg and Grossman [25] used white lists together with static analysis for preventing format string attacks, Their solution takes advantage of the dynamic nature of white-lists of %n-writable address changes, which are used to improve flexibility and encode specific security policies.

4.3.2 Model Checking for large-scale C-based Software verification

After introducing pointer analysis techniques, let us now survey some tools supporting automatic discovering of security flaws. We are particularly interested in model checking tools able to analyze huge software products such as an entire Linux distribution.

In [26], the MOPS static analyzer [9] is used to check security properties of a Linux distribution. MOPS relies on Finite State Automaton (FSA), whose state transitions correspond to syntactic patterns the program should not contain. For instance, syntactic patterns can express violations on pointer usage, structural type compatibility, and the like. The MOPS analysis process starts by letting users encode all the sequences of operations that do not respect the security properties they are interested in as paths leading to error states within FSAs. Program execution is then monitored against the FSAs; if an error state is reached, the program violates the security property. MOPS monitor takes a conservative view: potentially false positives are always reported, and then users have to manually check if an error trace is really a security vulnerability. MOPS-based experiments have been applied to the entire Red Hat Linux 9 distribution, which consists of 839 packages with about 60 millions of lines of code, and required the definition of new security properties to be model checked. To extend pattern expressiveness, [26] introduces *pattern variables*, which can describe different occurrences of the same expression. To improve scalability the concept of *compaction* is used, that is, simplifying the program to be analyzed by checking the relevant operations only, and MOPS is integrated with existing build processes and interposed with `gcc`. Error reporting in MOPS is then enhanced by dividing error traces

in groups and selecting a representative used by the users to determine if a bug has been discovered. With this extension, MOPS shows all programming errors and at the same time reduces the number of traces to be analyzed by the users by hand.

Below, we describe four main security properties defined in [26] and the results of the analysis of Red Hat Linux (see Table 4.2). The results show the feasibility of using MOPS for large scale security analysis.

Property	Warnings	Bugs
TOCTTOU	790	41
Standard File Descriptors	56	22
Temporary Files	108	34
strncpy	53	11

Table 4.2: Model-Checking Results

Time-To-Check-To-Time-Of-Use (TTCTTOU).   This technique checks whether access rights to an object have expired at the time it is used. A classic application is to find vulnerabilities of file systems due to *race conditions.* Let us consider the classic example of a process P that tries to access a file system object O and, once access has been granted, passes a reference to O as an argument of a system call, say, to display information in O. If a context switch takes place after P's access rights to O have been checked but before P executes of the system call, permissions may be changed while P is suspended. When P is scheduled again, it no longer has the right to access O; however, it goes on to pass the reference to O it already holds to the system call. Of this reference still allows displaying O, a vulnerability has been detected. In practice, three different types of vulnerabilities have been found: *(i) Access Checks,* discussed in the above example, *(ii) Ownership Stealing,* where an attacker creates a file where a program inadvertently writes, *(iii) Symlinks,* where the file a program is writing to gets changed by manipulating symbolic links. As shown in Table 4.2, 41 of 790 warnings (i.e., traces violating the security property) have been found to be real software faults.

Standard file descriptor.   This attack uses the three standard file descriptors of Unix (i.e., *stdin, stdout, stderr*) to exploit system vulnerabilities. Attackers can be able to append data to important files and then gain privileges, or read data from files that they are not supposed to access, by manipulating the standard file descriptors. As shown in Table 4.2, 22 of 56 warnings have been found to be real faults.

Secure Temporary Files.   An attacker exploits the practice of using temporary files to exchange data with other applications, writing logs, or storing temporary information. In Unix-like systems, these data are usually written to the /tmp directory, where each process can read/write. Also, the

functions to create temporary files are unsecure since they return a file name rather than a descriptor. An attacker guessing the file name is able to create a file with the same name and then access the information that will be stored in. As shown in Table 4.2, 34 of 108 warnings have been found to be real faults.

strncpy.   String copying is a classic source of potential attacks leading to buffer overflow attacks. strncpy is not safe since it leaves to the developer the responsibility of manually appending the null character ('n') that should terminate every C string. Both scenarios have been modeled with a FSA. However, this security property produced a set composed by 1378 unique warnings, which makes a complete manual analysis burdensome. An alternative to a complete analysis is selecting semi-randomly set of packages that contain at least one warning. In the experiments, 19 packages have been selected with 53 warnings, 11 of which have been identified as faults. If all warnings have the same probability of being faults, there are about 268 bugs among the 1378 unique warnings.

4.3.3 Symbolic approximation for large-scale OS software verification

A recent software solution aimed at the verification of large-scale software systems is based on an approach called *symbolic approximation* [2, 3, 4, 5, 6]. Symbolic approximation mitigates the state space explosion problem of model checking techniques (see Section 4.2.1), by defining an approximate logical semantics of C programs. The approach models program states as logical descriptions of what is true at each node in the program execution graph. These descriptions are compared with a program specification in order to identify those situations in which the program may do something bad. The seminal works in this field are the ones by Peter Breuer et al. [2, 4, 6]. These works were aimed at providing a formal solution for detecting deadlock, double-free and other errors in the several million lines of code in the Linux kernel. The application of their formal analysis to the Linux code detected faults and errors never identified by thousands of developers who had reviewed the Linux code. The analyzer, written in C, is based on a general compositional program logic called NRBG (the acronym comes from "Normal", "Return", "Break", "Goto", which represent different types of control flow).[8] A program fragment analyzed in NRBG terms is considered as operating in three phases: *i) initial*, a condition p holds at start of the execution of the fragment, *ii) during*, the fragment is executed, and *iii) final*, a condition q holds at the end of the execution of the fragment. Based on this logic, individual program

---

[8] NRBG logic is also defined for the treatment of loops, conditional statements and other functions such as lock, unlock, and sleep. Component G is used to represent the goto statement.

fragments are modeled in Hoare triplets. For instance, a normal exit from a
program fragment is modeled as follows:

$$\texttt{p N(a; b) q} \; = \; \texttt{p N(a) r} \; \wedge \; \texttt{r N(b) q}$$

To exit normally with q, the program flows normally through fragment a,
achieve an intermediate condition r, enter fragment b, and exit it normally.

A return exit (R) from a program fragment (i.e., the way code flows out
of the parts of a routine through a "return" path) is modeled as follows:

$$\texttt{p R(a; b) q} \; = \; \texttt{p R(a) q} \; \vee \; \texttt{r R(b) q}$$

Here, two paths are possible: *(i)* return from program fragment a, or *(ii)*
terminate a normally, enter fragment b, and return from b.

A static analyzer tool allows the detail of the logic above to be specified
by the user for the different program constructs and library function calls of
C, giving rise to different logics for different problem analysis. The tool in-
corporates a just-in-time compiler for the program logic used in each analysis
run. Logic specifications have the following form:

$$\texttt{ctx } \textit{precontext, precondition} \; :: \; \texttt{name(arguments)[subspecs]} = \textit{postconditions}$$

$$\textit{with ctx } \text{postcontext}$$

Here, *precondition* is the input condition for the code fragment, while
postconditions is a triple of conditions applying to the standard exit paths
(N, R, B) for the program *name*. The *precontext* and *postcontext* contain the
conditions pertaining to the additional exit paths provided by the `gotos` in
the program. As an example, let us consider the forever while loop logic.

$$\texttt{ctx } e, \; p :: \texttt{while(1)[body]} \; = \; \textit{(b,r,F)} \texttt{ with ctx } f$$

$$\texttt{where ctx } f, \; p :: \texttt{fix(body)} \; = \; \textit{(n,r,b)} \texttt{ with ctx } f$$

A normal exit occurs when the loop body hits a break statement with the
condition (b) holding. The normal loop body termination condition (n) and
the associated loop body return (r), and the break (b) conditions are defined
to be the fixpoint ('`fix(body)`') of the loop body, above the start condition
(p). That is:

$$n \; \geq \; p \; \wedge \; n \; :: \; \texttt{body} \; = \; \textit{(n,r,b)}$$

in the specification language terminology, or

$$p \; \Leftarrow \; n \; \wedge \; n \, \texttt{N(body)} \; n \; \wedge \; n \, \texttt{R(body)} \; r \; \wedge \; n \, \texttt{B(body)} \; b$$

in the abstract logic. A return exit happens when a return statement is ex-
ecuted from within the body of the while (r). The post-context f contains
the ways to exit the loop through a goto, as determined by the logic for the
loop body.

Each logic specification for the analyser covers the full C language. Below, we briefly discuss how the solution by Breuer et al. [2, 4, 6] can be used to locate instances of a well-known fault in programming for Symmetric Multi-Processing (SMP) systems called "sleep under spinlock". A spinlock is a well-known classical SMP resource-locking mechanism in which a thread waiting to obtain a lock continuously checks if the lock is available or not (a situation called *busy waiting*). The waiting thread occupies the CPU entirely until the spinlock is released by another thread on another CPU in the SMP system. Suppose now that, in a 2-CPU SMP system, the thread holding the lock (turns off interrupts and) calls a sleepy function (one which may be interrupted and scheduled out of the CPU for some time) and then is scheduled out of its CPU. If two new threads end up busy-waiting for the same spinlock before the spinlock holder can be rescheduled, the system is deadlocked: the two threads occupy both CPUs entirely and interrupts are off. This vulnerability is critical since an adversary can exploit it to bring a *denial of service* attack [11].

To identify the calls to sleepy functions under spinlock, the logic specification in [6] provided a single unlock logic pattern for all the variants of spin-unlock calls in the Linux kernel. The logic decrements a spinlock total counter (see Figure 4.3). Similarly, the specification provided a single lock logic pattern for all the variants of spinlock calls of Linux.

$$\text{ctx } e, \ p :: \texttt{unlock(label l)} \ = \ (p[n{+}1/n],F,F) \text{ with ctx } e$$
$$\text{ctx } e, \ p :: \texttt{lock(label l)} \ = \ (p[n{-}1/n],F,F) \text{ with ctx } e$$

Fig. 4.3: Logic specification of `unlock` and `lock` function

A logical objective function was specified for this analysis which gauges the maximum upper limit of the spinlock counter at each node of the syntax tree. A set of trigger/action rules creates the sleepy call graph. When a new function is marked as sleepy, all callers of the function plus all callers of its aliases are marked as sleepy as well. The analysis creates a list with all the calls that may sleep under a spinlock.

files checked	1055
alarms raised	18 (5/1055 files)
false positives	16/18
real errors	2/18 (2/1055 files)
time taken	about 24h (Intel P3M, 733 MHz, 256M RAM)
LoC	about 700K

Table 4.3: Linux Kernel 2.6.3: testing for sleep under spinlock

The Linux kernel 2.6.3 was tested to find occurrences of sleep under spin-lock (see Table 4.3 for more details); *1055* files of about 700K LOC were considered and *18* alarms raised. The real faults found amounted to *2* of the *18* alarms. The fact that many of the alarms were false positives should not necessarily be seen as a problem when the effort of analyzing false positives is considered in relation to the effort involved in finding faults manually.

4.4 Conclusion

Formal method verification is an important aspect of software security aimed at reducing risks caused by software faults and vulnerabilities. Model-based certification gathers a variety of formal and semi-formal techniques, dealing with verification of software systems' reliability, safety and security properties. In particular, model-based techniques provide formal proofs that an abstract model (e.g., a set of logic formulas, or a formal computational model such as a finite state automaton), representing a software system, holds a given property. As discussed in this chapter, much work has been done in the context of static analysis, formal methods and model checking. These solutions have gained a considerable boost in the recent years and are now suitable for software verification in critical security contexts and for verification of large scale software systems such as a Linux distribution. However, some drawbacks still need to be addressed. Firstly, these techniques produce a non-negligible rate of false positives, which require a considerable effort for understanding which warnings correspond to real faults. Secondly, model-based techniques do not support configuration evolution: it is not guaranteed that a certification obtained for one configuration will still holds when the configuration changes.

References

1. D. Avots, M. Dalton, V. Livshits, and M. Lam. Improving software security with a c pointer analysis. In *Proc. of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, May 2005.
2. P.T. Breuer and S. Pickin. Checking for deadlock, double-free and other abuses in the linux kernel source code. In *Proc. of the Workshop on Computational Science in Software Engineering*, Reading, UK, May 2006.
3. P.T. Breuer and S. Pickin. One million (loc) and counting: static analysis for errors and vulnerabilites in the linux kernel source code. In *Proc. of the Reliable Software Technologies-Ada-Europe 2006*, Porto, Portugal, June 2006.
4. P.T. Breuer and S. Pickin. Symbolic approximation: an approach to verification in the large. *Innovations in Systems and Software Engineering*, 2(3–4):147–163, December 2006.
5. P.T. Breuer and S. Pickin. Verification in the light and large: Large-scale verification for fast-moving open source c projects. In *Proc. of the 31st Annual*

IEEE/NASA Software Engineering Workshop, Baltimore, MD, USA, March 2007.

6. P.T. Breuer, S. Pickin, and M. Larrondo Petrie. Detecting deadlock, double-free and other abuses in a million lines of linux kernel source. In *Proc. of the 30th Annual IEEE/NASA Software Engineering Workshop*, Columbia, MD, USA, April 2006.

7. *C - Approved standards.* http://www.open-std.org/jtc1/sc22/wg14/www/standards.

8. *Carnegie Mellon University's Computer Emergency Response Team.* http://www.cert.org/.

9. H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proc. of the 9th ACM Computer and Communications Security Conference (ACM CCS 2002)*, Washington, DC, USA, November 2002.

10. K. Chen and D. Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *Proc. of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, San Diego, California, USA, June 2007.

11. W. Cheswick and S. Bellovin. *Firewalls and Internet Security; Repelling the Wily Hacker.* Addison Wesley, 1994.

12. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*, December 1999. MIT Press.

13. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. of the Fourth European PKI Workshop: Theory and Practice (EUROPKI 2007)*, Mallorca, Balearic Islands, Spain, June 2007.

14. N. Dragoni, F. Massacci, C. Schaefer, T. Walter, , and E. Vetillard. A security-by-contracts architecture for pervasive services. In *Proc. of the 3rd International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing*, Istanbul, Turkey, July 2007.

15. D.A. Duffy. *Principles of Automated Theorem Proving.* John Wiley & Sons, 1991.

16. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January-February 2002.

17. U. Glasser, R. Gotzhein, and A. Prinz. Formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.

18. C. A. R. Hoare and C. B. Jones (eds.). *Essays in computing science.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

19. R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proc. of the 13th conference on USENIX Security Symposium*, San Diego, CA, USA, August 2004.

20. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

21. W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

22. G. Necula. Proof-carrying code. In *Proc. of the ACM Principles of Programming Languages (POPL 1997)*, Paris, France, January 1997.

23. G.C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI 1998)*, Montreal, Canada, May 1998.

24. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, 1999.

25. M.F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proc. of the 12th ACM conference on Computer and Communications Security (CCS 2005)*, Alexandria, VA, USA, November 2005.

26. B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proc. of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, Tucson, Arizona, USA, December 2005.

27. R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Model-carrying code (mcc): A new paradigm for mobile-code security. In *Proc. of the New Security Paradigms Workshop (NSPW 2001)*, New Mexico, USA, September 2001.

28. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, New York, USA, October 2003.

29. O. Shivers. Control-flow analysis in scheme. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, Atlanta, Georgia, USA, June 1988.

30. R.F. Staerk, J. Schmid, and E. Boerger. Java and the java virtual machine: Definition, verification, validation. *Springer-Verlag*, 2001.

31. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of the Network and Distributed Systems Security Symposium (NDSS 2000)*, pages 3–17, San Diego, California, USA, February 2000.

Chapter 5
OSS security certification

Abstract Open source software is being increasingly adopted for mission and even for safety-critical applications. Experience has shwon that many open source software products have achieved adequate functionality and scalability. Security, however, requires a specific analysis, since open source software development does not usually follow security best practices. Indeed, the lower number of security events involving open source software may be ascribed to its smaller market share rather than to its robustness. In this chapter we start by taking a closer look to the meaning of the open source label, discuss the connection between licenses and certificates. Then, we summarize the debate on open source security and discuss some issues pertaining to open-source assurance activity and to open source security certification.

5.1 Open source software (OSS)

The debate between open source and closed source supporters dates back to the origin of software and is still far away from a conclusion. The advent of the Internet has made this contraposition even more harsh. The closed software model was originally introduced in the 1970s, when software commercialization became a reality. Computer software was treated as a company asset, to be protected from competitors who might otherwise reproduce, study or modify the code, to resell, use, or learn from the product. The closed source paradigm allowed software houses to protect their products from piracy or misuse, reverse engineering and duplication. Also, the closed source paradigms allowed software suppliers to preserve competitive advantage and vendor lock-in.

By contrast, the concept of free software was born as a social movement (1983) aimed to protect user's rights to freely access and modify software. In 1985, Richard Stallman founded the *Free Software Foundation* [20] (FSF) to support such a movement. In 1998, a group of members of the FSF replaced

the term *free software* with *open source*, in response to Netscape's January 1998 announcement of a source code release for their Netscape Navigator browser.

As Stallman himself pointed out, OSS development departs radically from commercial software form the very beginning: an open source software solution is not planned as a product to be offered to the market to achieve a profit. Often, the initial idea and the design do not take place within the boundaries of a traditional business organization. Some OSS projects, such as business applications or middleware, are entirely autonomous. Others exist as components, as for instance, part of the Linux kernel. Large, mature projects such as the Apache Web server have they own autonomous evolution strategy, although benefiting from the contributions of hundreds of outside submitters. Whatever the status (component or application) of an open source project, it will remain useful only as long as it has a well-specified assurance process, that is, its maintainers properly update and test it. In this Chapter we shall discuss when and how software certification can become a part of the assurance effort of open source projects.

5.1.1 Open Source Licenses

The action of the FSF was instrumental for clarifying the semantics of the open source label: an open source software product is a software product made available under an open source license. Today, open source licenses are not just declarations which grant to the user unlimited rights of accessing and modifying the software product's source code. Rather, licenses specify in detail who is entitled to access the source code, and the allowed actions to be performed on it. The reader should not miss the conceptual link between a software license and a certificate: both of them assert some properties of a software product, and both may be distributed and checked in a digital format. A major difference between licenses and certificates is that the former seldom deals with software properties but, rather, the assertions made in a software license focus on the rights and obligations of both purchaser and supplier concerning the software product's usage or redistribution. The wording of the license may be such that the software supplier has no obligation whatsoever, not even that the software program will be useful for any specific particular purpose. In the license, however, the purchaser may get permission from the supplier to use one or more copies of software in ways where such a use would otherwise constitute infringement of the software supplier's rights under copyright law.[1]

---

[1] Such use (e.g., creating archival copies of the software) may be permitted by law in some countries, making it unnecessary to explicitly mention it in the license. In effect, this part of a proprietary software license amounts to little more than a promise from the software supplier not to sue the purchaser for engaging in activities that, under

Some licenses give very limited rights to the purchaser. Many proprietary licenses are *non-concurrent*, that is, do not allow the software product to be executed simultaneously on multiple CPUs, or set a limit to the number of CPUs that can be used. Also, the right of transferring the license to another purchaser, or to move the program from one computer to another may be limited in the license.[2] Finally, proprietary software licenses usually have an expiration date. The license validity may range from perpetual to a monthly lease. A time-limited license can be self enforcing: the supplier may insert into the software product a security mechanism that, once the license has expired, will disable the software product. An interesting by-product of time-limited licenses is the emerging need for certifying that security mechanism used to enforce license expiration do not impair the software performance or dependability while the license is still valid.

Software suppliers have traditionally been lax on license enforcement and then individual purchasers of proprietary software tend to pay little attention to compliance. However, most large companies and organizations, including universities, have established strong license compliance policies.

Open source is sometimes seen (or presented) as a way out from the complex problem of guaranteeing compliance to proprietary software licenses. However, OSS is itself distributed under a license and more than 30 licenses actually exist [30]. To help in establishing some degree of uniformity, the *Open Source Initiative* (OSI) [36], jointly founded by Eric Raymond and Bruce Perens in February 1998, has promoted since long a specification (called *Open Source Definition* (OSD)) of what must appear in a license in order for the software covered by it to be considered open source. Licensers are however free to use licenses that go beyond OSD minimum requirements, in the sense of providing more rights to the user. Thus, OSD-compatible licenses are not all the same.

OSI's open source definition mandates that the license of an open source software product must comply with ten criteria [26], described as follow.

1. *Free redistribution.* An open source license must permit anyone who obtains and uses the covered software to give it away to others without having to pay a royalty or other fee to the original copyright owner(s).
2. *Access to source code.* All types of open source licenses require everyone who distributes the software to provide access to the program source code. Often, distributors provide the source code along with the executable form of the program, but the license does not bind them to do so; for instance, they could make the code available via Internet download, or on other media, free or for a reasonable fee to cover the media cost.

---

the law of the country where the purchase is made, would be considered exclusive rights belonging to the supplier itself.

[2] For instance, OEM Windows licenses are not transferable. When the purchaser does not longer use the computer where the Windows software is pre-installed, the Windows license must be retired.

3. *Derivative works.* An open source license must allow users to modify the software and to create new works (called *derivatives*) based upon it. An open source license must permit the distribution of derivative works under the same terms as the original software. This provision, together with the requirement to provide source code, fosters the rapid evolution pace of open source software.

4. *Integrity* of the author's source code must be preserved.

5. *No discrimination on users.* An open source license does not discriminate against persons or groups. Everybody can use open source software, provided they comply with the terms of the open source license.

6. *No discrimination on purpose.* An open source license does not discriminate against application domains. In other words, the license may not restrict anyone from using the software based on the purpose of such usage. Specifically, it may not restrict the program from being used for commercial purposes. This permits business users to take advantage of open source products for commercial purposes.

7. *License Distribution.* The wording of an open source license must be made available to all interested parties, and not to the purchaser alone.

8. *Product Neutrality.* An open source license must not be specific to a single software product.

9. *No transfer of restrictions.* An open source license on a software product must not restrict the use of other software products, both open source and proprietary. In other words, an open source license must not mandate that all other programs distributed together with the one the license is attached to are themselves open source. This clause allows software suppliers to distribute open source and proprietary software in the same package. Some widespread licenses, including the GPL (*General Public License*) presented below, require that all software components "constituting a single work" to fall to under the GPL if anyone of them is distributed under GPL. This requirement may seem to have been spelled out clearly, but wrapping and dynamic invocation techniques have sometimes been used as a work-around to it.

10. *Technology Neutrality.* An open source license must not prescribe or supply a specific technology.

It is beyond the scope of this book to provide a detailed analysis of all open source licenses. Here, we shall limit ourselves to outlining the main features of some widespread ones. The interested reader is referred to our main reference, the classic book [30]. However, it is important to remark that organizations and individuals supporting the open source paradigm (including the authors of this book) firmly believe that the benefits given by a community of gifted and enthusiastic software developers working at the evolution of a software product are much more important than the (often illusory) advantages of protecting the intellectual property rights on it.

- GNU *General Public License* (GPL) is one of the first open source licenses and still by far the most widely used. It is considered a liberal license inasmuch the original programmer does not retain any right on modified versions of the software. Richard Stallman and Eben Moglen created the GPL and started the Free Software Foundation to promote its use. For instance, Linux is distributed under a GPL license.
- The *Mozilla Public License* (MPL) is another popular open source license. It came about to distribute the original Mozilla open source web browser. It is less liberal than GPL, inasmuch it requires the inclusion or publishing of the source code within one year (or six months, depending on the specific situation) for all publicly distributed modifications.
- The *Berkeley Software Distribution* (BSD) License was one of the earliest non-proprietary licenses, and follows a very different philosophy than GPL. BSD permits users to distribute BSD-licensed software for free or commercially, without providing the source code; they also may modify the software and distribute the changes without providing the source code. A major difference between BSD and GPL is that organizations or individuals who create modified versions of software originally licensed under BSD can distribute them as proprietary software, provided that they credit the developers of the original version. Two other widespread open source licenses, the Apache Software License and the MIT License, are very similar to the BSD License.

The differences between GPL and other open source licenses become very relevant when a user creates a derivative from existing open source code. In this case, with GPL the license is inherited. The new code must be distributed under the same license as the original version. This may not be true for other licenses.

5.1.2 Specificities of Open Source Development

The rapid pace of evolution and the multi-party development fostered by the open-source licensing policies described above make open source software products very different from proprietary ones. Generally speaking, open source software is developed and modified by programmers who devote their time, energy and skills without receiving any direct compensation for their work [24]. In this context, the whole relation between software purchasers and software suppliers changes dramatically. Open source gives much more power to customers who need customized products that fit their business activities. If a customer chooses to use open source software, say, for human resource management, a software supplier can offer to customize the software for that individual customer. In this case the customer will be charged a fee not for using the software itself, but for the service of customizing it.

Before discussing OSS security certification, it is therefore important to look at the process of developing OSS [43]. OSS has fostered a new software development style based on a heterogeneous mixture of existing methodologies and development processes. It does not provide any standard criteria to select activities for the different projects; instead, it is up to developers to agree on which methodology is more suitable for them [16]. Rather than by a specific set of activities, the OSS development process is characterized by its rapid release cycle, for teams that put together developers with diverse skills and competences, for fast rate of code change over time and for the use of readable code as a way to satisfy the need of a clear and unambiguous documentation.[3] Code is seen as the first specification of open source systems and, as a result, those systems are often otherwise undocumented.

The above description may convey the idea that OSS "emerges" from a Wikipedia-style "democratic" cooperation rather than from a disciplined development process. This is however not the case for some major OSS projects. In particular, here we are interested in the level of coordination which is crucial for testing and security assurance.

As an example, let us briefly consider the Linux development process. We will come to certifying Linux distributions in Chapter 6. Bill Weinberg[4] describes the Linux kernel development and maintenance process as a "benevolent dictatorship". It is probably more like feudalism: while contributions to the kernel come from developers worldwide, the authority of including and integrating them in the Linux kernel belongs to around 80 maintainers, each responsible of a subsystem of the Linux kernel. Each subsystem has its own versioning; sets of subsystems are integrated into *patch sets* that, in turn, are used to set up "experimental" kernel versions (in the past, these corresponded to odd-numbered kernel releases like $2.3.x$ and $2.5.x$). When the highest authority (Linus Torvalds and his team) considers an experimental kernel ready for deployment, a new production kernel is generated and handed off for testing to the production kernel maintainers, who are responsible for the entire testing process. Starting from production kernels, companies and consortia create Linux distributions aimed at the business or embedded systems domain.

Linux kernel maintainers can also select external OSS projects, such as device drivers, to be integrated into a Linux subsystem. Some examples are security-related; for instance, the *National Security Agency* (NSA) *Secure Linux* was adopted as a standard build option in the 2.6 Linux kernel.

---

[3] OSS is seldom developed from a stable specification and *a priori* software requisites are usually vague.

[4] The description below is partly based on his contribution "The Open Source Development Process", originally published on the *Embedded Computing Design* magazine, http://www.embedded-computing.com/departments/osdl/2005/1/.

5.1.2.1 The Open Source Code Assurance

Understanding the general quality assurance of OSS code is particularly important for the integration of the security certification process into OSS development. Let us start with a formal definition of *Software Assurance* (SwA): an activity aimed at increasing the level of confidence that a software product operates as intended and is free of faults. In a traditional, lifecycle-based software development process, assurance includes a number of tasks to be carried out by developers and testers along the software lifecycle.
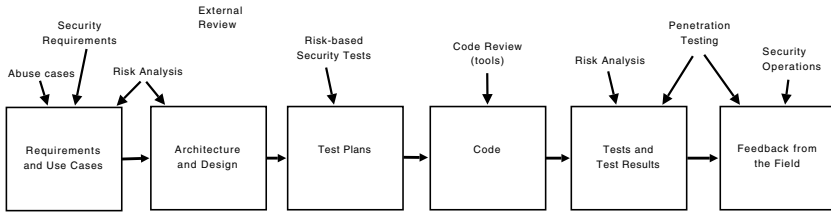

Fig. 5.1: Assurance tasks in a traditional, lifecycle-based development process

Security assurance activities for OSS code could be performed at several points in the code life cycle. Contributions to major OSS projects like Linux are strictly monitored and must meet quality standards; here, we are interested in the assurance process used to keep these standards. Of course, different OSS projects will use different assurance procedures, but some conditions are verified for all OSS projects. Upon submission, a contribution to an OSS project must be *well-formed*, that is, coded and packaged according to well-established OSS conventions. The first assurance activity is usually checking for novelty and interest of the contribution, that is, the properties (either functional or non-functional) it would add to the project. These checks are usually made by core members of the community (in the case of the Linux kernel, by the subsystem maintainers). Once a contribution makes is accepted into a OSS project, it will be tested, and reviewed by the project's community to become part of the project's mainstream code. In the specific case of Linux, the process is two-tiered: if the subsystem or project containing the new contribution is then picked up by a Linux distribution like SuSE, it will be subject to that distribution-specific assurance procedure. Typical assurance activities performed at distribution level include standards-compliance testing (for example, LSB and POSIX), stability and robustness testing. Today, Linux security certification is also carried out at the distribution level (Chapter 6).

It should be noted that the assurance process described above (see Figure 5.2) can be made three-tiered by adding a user-specific assurance activity at adoption time. In the case of Linux, this assurance procedure is coarser-grained, as adopters retain control over the inclusion of each specific sub-
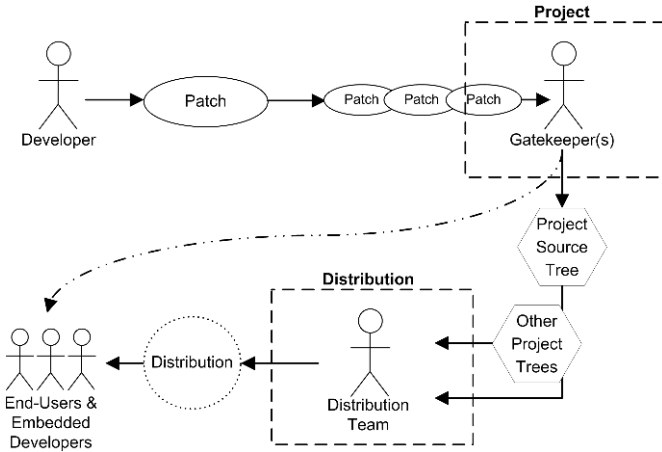
Fig. 5.2: The two- and three-tiered assurance process of OSS

system in the distribution, but of course not of individual contributions to subsystems. However, regression is always possible, that is, the adopter can always return to a previous version of the subsystem if an update does not meet the adopter's quality or dependability standards. Some adopters prefer to entirely delegate security and quality assurance to their preferred distribution; others try and influence the OSS product evolution at both design and assurance time.[5] We are now ready to summarize the set of properties that distinguish OSS development from a traditional development processes from the point of view of security assurance.

- *Large distributed community of developers.* It is one of strongest points of OSS development. No matter how a big a company can be, it will never have as many developers as some well known open source projects.
- *Fast feedback from users.* Fault reports, usability problems, security vulnerabilities reports are some examples of the feedback the open source community receives constantly from the users. This communication tends to be more frequent and intense, and above all more direct, than the one taking place for proprietary software products. This feedback allow open source developers to locate and fix reported faults very rapidly.
- *Users are an integral part of the development process.* Users have been always considered part of the OSS development cycle. Keeping the users involved helps them to express their point of views about functional re-

---

[5] A notable example is the telecommunication sector, where *Carrier Grade Linux* (CGL) defines a set of specifications as well as some assurance criteria which must be met in order for Linux to be considered "carrier grade" (i.e., ready for use within the telecommunications industry).

quirements, security, usability, write documentation, and other tasks that
developers may neglect.

- *Talented and highly motivated developers.* The idea of having their code
  reviewed by a member of the core group of an open source software, or
  knowing that their code would be part of the next release of a widespread
  software product attracts many motivated developers to contribute to
  open source projects [16].
- *Rapid release cycles.* Thanks to a large, active and high motivated com-
  munity of developers working around the clock, many open source soft-
  ware provide more rapid release cycle and updates compared to software
  developed inside companies. For example, in 1991 there was more than
  one release per day for the Linux kernel [16].
- *Terseness of analysis documentation.* OSS is not based on the definition
  of a set of formal and stable specifications and requisites to be used in
  the development process; rather, community-wide discussions are used as
  a kind of implicit specifications that drive the developers contribution to
  the software.

The above properties of OSS development correspond to many advan-
tages (as well as to some disadvantages) compared to other types of software
development processes. In Section 5.2, we shall discuss in detail how these
advantages and disadvantages affect the adoption of OSS in critical security
areas and the certification of OSS security properties. However, we can an-
ticipate some interesting facts. First of all, while the open source specificity
in terms of development process and licenses may still play a marginal role
in the decisions of individual users, it has already a huge impact on adoption
patterns of companies and organizations. Sometimes companies do not have
the time and resources for a complete pre-adoption analysis of OSS and just
"go and use it". This pattern of OSS adoption is often seen as a leverage
against vendors of proprietary software products. Other times, companies
rely on internal adoption guidelines, and look at the exploitation of OSS
community-based development as a way to reduce development costs and ac-
celerate the availability of new features. This opportunistic strategy does not
come for free, because it requires creating internal competence groups able to
influence the relevant development communities. Alternatively, brokers (often
individual consultants or small companies) familiar with the OSS develop-
ment process can be hired to manage partially or totally the interaction with
the development community.

5.2 OSS security

In the last few years, security has not been a major driver for open source
assurance [4]. Possible reasons include some reluctance by the OSS commu-
nity to set up a separate security assurance process, as opposed to a general

quality assurance one. On Tuesday July 15th 2008, posting to the `gmane` discussion group, Linus Torvalds wrote:

"…one reason I refuse to bother with the whole security circus is that I think it glorifies - and thus encourages - the wrong behavior.
It makes "heroes" out of security people, as if the people who don't just fix normal bugs aren't as important.
In fact, all the boring normal bugs are way more important, just because there's a lot more of them. I don't think some spectacular security hole should be glorified or cared about as being any more "special" than a random spectacular crash due to bad locking".

In turn, some security experts do not completely trust the open source communities and consider open source middleware a potential "backdoor" for attackers, potentially affecting overall system security. However, proprietary security solutions have their own drawbacks such as vendor lock-in, interoperability limitations, and lack of flexibility. Recent research suggests that the open source approach can overcome these limitations [3, 41].

A long debate has been going on in the security research community, whether open source software should be considered more or less secure than closed source software. A classical analysis is the one done by Wheeler in [52]. This debate has not led to any definitive conclusion so far [13], and it is unlikely to do so in the near future. It is however interesting to note that the discussion has been mainly confined to software implementing security solutions rather than extended to general system-level software products or to software applications, although it is widely recognized that most new threats to security are emerging at the application level. According to another essay published by David Wheeler, the author of "Secure Programming for Linux and Unix" [51] on his book's website (http://www.dwheeler.com/secure-programs): "There has been a lot of debate by security practitioners about the impact of open source approaches on security. One of the key issues is that open source exposes the source code to examination by everyone, including both attackers and defenders, and reasonable people disagree about the ultimate impact of this situation". Wheeler's interesting essay contains a number of contending opinions by leading security experts, but no conclusion is reached.

Probably, the only non-questionable fact is that open source software gives both attackers and defenders greater control over software security properties. Let us summarize Wheeler's arguments in favor and against the notion that a software being open source has a positive impact on its security-related properties.

- *Open source is less secure.* The availability of source code may increase the chances that an attacker will detect and exploit a software fault. Also, the informality and flexibility of the open source development process have been known to backfire. For instance, not all open source projects provide documentation on the secure deployment of the software they develop. This lack of information may cause faulty installations, which in

turn may result in security holes. Some open source development communities do not bother with hardening their products against well-known security vulnerabilities and do not provide any report on fault detection, even though some simple open source tools, such as FindBugs, are available (see Chapter 4 for more details). Delay in implementing well-understood security patches is a clear indication that security best practices are a low priority issue for some open source projects. This may be understandable in some cases, because security requirements may be quite different for different applications, but it could also indicate a lower level of security awareness on the part of the OSS community.

- *Open source is more secure.* The greater visibility of software faults [13] typical of open source products may also be exploited by defenders. Well trained defenders, taking advantage of knowledge normally not available with closed source software, have been known to ensure a short response time, fixing software vulnerabilities as soon as the corresponding threats are described. Defenders can also rely on the independent work of the open source community to identify new security threats, since open source software is usually subject to a community-wide review and validation process.

Whatever the merits of these positions, this controversy is bound to remain somehow academic. Open source projects largely differ from one another, and the same arguments can be brought in favor or against the thesis depending on the specific situation considered. In their paper "Trust and Vulnerability in Open Source Software" [25], Scott A. Hissam and Daniel Plakosh rightly observe that "just because a software is open to review, it should not automatically follow that such a review has actually been performed".[6]

Hopefully, however, the above discussion has clarified the need of some form of security certification based on a rigorous and in-depth analysis. What is still missing is a security certification framework allowing, on the one side, suppliers to certify the security properties of their software and, on the other side, users to evaluate the level of suitability of different open source security solutions. We shall discuss the requirements for such a framework in the next section.

5.3 OSS certification

In principle, the standard certification processes described in this book can be employed to certify OSS products, as they are for proprietary ones. The obstacle posed by the peculiar nature of OSS development process can in fact be overcame, since some certification standards accept applications from organizations adopting nearly every type of development process. Therefore

---

[6] Incidentally, we remark that the same comment can be made for testing.

forges, consortia or foundations promoting open source product development
do in principle qualify as applicants for obtaining the certification of OSS
products. For example, the norm ISO 9126 (see Chapter 1) explicitly men-
tions an application to OSS projects. Any OSS project compliant to these
standards can in theory acquire the same status as a conventional project.
This is also true for other certifications we described or mentioned in the
previous Chapters, like ISO 9000 for software suppliers, ITSEC (European),
or CC (international). The evaluation body will examine a software product's
specified functionality, the quality of its implementation, and the compliance
with security standards.

However, some other obstacles do exist in terms of laboratory techniques.
In fact, an important prerequisite to most certifications is the availability
of a testing framework to support all controls required by the certification
process. We shall deal with this problem in Chapter 8

5.3.1 State of the art

Comparative evaluation of OSS non-functional properties, including security-
related ones is a time-honored subject. Much work has been done on open
source security testing: for instance, the *Open Source Security Testing Method-
ology Manual* (OSSTMM) (`http://www.isecom.org/osstmm/`) is a peer-
reviewed methodology for performing security tests and metrics. The OS-
STMM test cases are divided into five sections which collectively test: infor-
mation and data controls, personnel security awareness levels, fraud and so-
cial engineering control levels, computer and telecommunications networks,
wireless devices, mobile devices, physical security access controls, security
processes, and physical locations such as buildings, perimeters, and military
bases. OSSTMM focuses on the technical details of exactly which items need
to be tested, what to do before, during, and after a security test, and how
to measure the results. New tests for international best practices, laws, regu-
lations, and ethical concerns are regularly added and updated. The method-
ology refers to risk-oriented metrics such as Risk Assessment Values (RAVs)
and defines and quantifies three areas (operations, controls, limitations) as
its relevance to the current and real state of security.

The *Qualify and Select Open Source Software* (QSOS) is a methodology
designed to qualify, select and compare free and open source software in
an objective, traceable and argued way (`http://www.qsos.org/`). QSOS
aims to compare solutions against formalized requirements and weighted cri-
teria, and to select the most suitable product set available. QSOS is based on
an iterative approach where the evaluation step is based on metrics defining,
on the one hand, the risks from the customer perspective and, on the other
hand, the extent to which these risks are addressed by OSS solutions. In
general, all selection techniques require information that many open source

projects fail to make available. The *Software Quality Observatory for Open Source Software* project (`http://www.sqo-oss.eu/`) includes techniques computing account quality indicators from data that is present in a project's repository. However, such metadata need to take into account the specific domain of the application. For instance, the dependency of loop execution times on hardware features is a relevant quality indicator for time-critical control loops, but has little interest for business application developers.

Several researchers [10] have proposed complex methodologies dealing with the evaluation of open source products from different perspectives, such as code quality, development flow and community composition and participation. General-purpose open source evaluation models, such as Bernard Golden's *Open Source Maturity Model* (OSMM) [21] do not address the specific requirements of security software selection. However, these models assess open source products based on their maturity, that is, their present production-readiness, while evaluating security solutions also involves trying to predict how fast (and how well) a security component will keep abreast of new attacks and threats. Several other OSS adoption methodologies have been proposed and developed into practical guidelines defining methodology- (or enterprise-) specific benchmarks in terms of functionality, community backing as well as maturity. Most of these methodologies, however, are biased toward business-related software systems and toward static integration. Therefore, they are of limited use for complex products like telecommunication devices, which bundle or dynamically integrate hardware and software components. For example, the *Business Readiness Rating* (BRR) method [40] supports quantitative evaluation of open source software identifying seven categories: functionality, software quality, service and support, documentation, development process, community, and licensing issues. However, additional work is required to deal with OSS non-functional properties (performance, security, safety) across the different integration mechanisms, and with white-box ones (terseness/sparseness, readability), which are crucial to OSS bundling within complex products [49]. Focusing on security area, a security-oriented software evaluation framework should provide potential adopters with a way to compare open source solutions identifying the one which *(i)* best suits their current non-functional requirements and *(ii)* is likely to evolve fast enough to counter emerging threats. Our own works in [2, 5] are aimed at providing a specific technique for evaluating open source security software, including access control and authentication systems. Namely, a *Framework for OS Critical Systems Evaluation* (FOCSE) [2] has been defined and is based on a set of *metrics* aimed at supporting open source evaluation, through a formal process of adoption. FOCSE evaluates an open source project in its entirety, assessing the community composition, responsiveness to change, software quality, user support, and so forth. FOCSE criteria and metrics are also aimed at highlighting the promptness of reacting against newly discovered vulnerabilities or incidents. Applications success, in fact, depends on the above principle because a low reaction rate to new vulnerabilities or incidents

Metrics	Putty	WinSCP	ClusterSSH
Age (GA)	2911 days	1470 days	1582 days
Last Update Age (GA)	636 days	238 days	159 days
Project Core Group (GA,DC)	Yes	Yes	Yes
Number of Core Developers (DC)	4	2	2
Number of Releases (SQ,IA)	15	32	15
Bug Fixing Rate (SQ,IA)	0.67	N/A	0.85
Update Average Time (SQ,IA)	194 days	46 days	105 days
Forum and Mailing List Support (GA,DIS)	N/A	Forum Only	Yes
RSS Support (GA,DIS)	No	Yes	Yes
Number of Users (UC)	N/A	344K	927
Documentation Level (DIS)	1.39 MB	10 MB	N/A
Community Vitality (DC,UC)	N/A	3.73	5.72

Table 5.1: Comparison of open source SSH implementations at 31 December 2006

implies higher risk for users that adopt the software, potentially causing loss of information and money. Finally, to generate a single estimation, FOCSE exploits an aggregator often used in multi-criteria decision techniques, the Ordered Weighted Average (OWA) operator [50, 54], to aggregate the metrics evaluation results. This way, two or more OSS projects, each one described by its set of attributes, can be ranked by looking at their FOCSE estimations. In [2], some examples of the application of FOCSE framework to existing critical applications are provided. Here, we provide a sketch of the FOCSE-based evaluation of security applications that implement the Secure Shell (SSH) approach. SSH is a communication protocol widely adopted in the Internet environment that provides important services like secure login connections, tunneling, file transfers and secure forwarding of X11 connections [55]. The FOCSE framework has been applied for evaluating the following SSH clients: Putty [39], WinSCP [53], ClusterSSH [48]. First, the evaluation of SSH client implementations based on the security metrics and information gathered by FLOSSmole database [17] is provided and summarized in Table 5.1.

	Putty	WinSCP	ClusterSSH
f_{OWA}	0.23	0.51	0.47

Table 5.2: OWA-based comparison of SSH clients

Then, an OWA operator is applied to provide a single estimation of each evaluated solution. Finally, the FOCSE estimations are generated (see Table 5.2), showing that the solution more likely to be adopted is WinSCP. In summary, FOCSE evaluation framework gives a support to the adoption of open source solutions in mission critical environments.

As far as model-based certification is concerned, some ad hoc projects toward applying model- and test-based certification techniques to individual OSS products have been taken. For instance, the U.S. Department of Homeland Security has funded a project called "Vulnerability Discovery and Remediation, Open Source Hardening", involving Stanford University, Coverity and Symantec. The project was aimed at hunting for security-related faults in open-source software, finding and correct the corresponding security vulnerabilities, and to improve Coverity's commercial tool for source code analysis. This effort resulted in a system that does daily scans of code contributed to popular open-source projects, and produces and maintains a database of faults accessible to developers.

Looking at test-based certification, some major players developed and published anecdotal experience in certifying open source platforms, including Linux itself [44], achieving the Common Criteria (CC) EAL-4 security certification. The work in [44] describes the IBM experience in certifying Open Source and illustrates how the authors obtained the Common Criteria security certification evaluating the security functions of Linux, the first open-source product receiving such certification. We shall discuss the Linux certification process in more detail in the following Chapters.

General frameworks are needed to provide a methodology for functional and non-functional testing of OSS. Referring to the terminology we introduced in Chapter 3, no widely accepted OSS-specific methodology is available supporting white-box testing in terms of code terseness/sparseness, readability and programming discipline. As far as black-box testing is concerned, description of tests carried out on OSS at unit or component level are sometimes made available by the development communities or as independent projects (see Chapter 6). However, complex systems whose components have been developed independently may require additional support for integration and system testing. Furthermore, to obtain a genuine certification, in terms of inward security and outward protection of a complex software system which includes open source, it is not sufficient that all elements of the system are certified: the composition of security properties across the integration technique must also be taken into account. When OSS is introduced into the context of complex modular architectures, certifying the overall security of the product is a particularly critical point [13, 37].

The issues to be addressed in the context of a security assurance and certification involving an OSS software product can be classified in the following four areas.

- *Functional test and certification.* Provide test and certification of functionalities, supporting the use of OSS within critical platforms and environments for operating and business support systems, gateway, signaling and management servers, and for the future generations of voice, data and wireless components; define a comprehensive approach to certification of products dynamically integrating OSS, creating an OSS specific

path to certify typical functionalities of complex networking systems like routing, switching, memory management and the like.

- *Integration support.* Complement existing approaches providing the specific design tools needed to bring OSS into the design and implementation path of advanced European products, providing the research effort needed for successful OSS integration within complex systems and for using OSS as a certified tool for complex systems development.
- *Advanced description.* Provide novel description techniques, suitable for asserting the relevant properties of OSS also integrated in telecommunication devices and other embedded systems. Properties expressed should include specific ones such as timing dependencies, usage of memory and other resources.
- *Governance and IPR issues.* Develop variety of across-project indicators on OSS dynamic integration and static bundling. Indicators will provide company management (as opposed to the leaders of individual development projects) with quantitative and value-related percentages of OSS adoption (e.g., within product lines), so that company wide governance policies regarding OSS adoption, as well as the integration techniques, can be monitored and enforced. Also, it will guarantee IPR peace of mind by providing support for assessing the IPR nature of products embedding OSS.

5.4 Security driven OSS development

Although the lack of a formal software development process is usually not seen as a drawback by OSS communities, it may become a problem in the security area, because security assurance techniques often assume a stable design and development process. Focusing on security aspects, discussions in OSS communities rarely state formal security requirements; rather they mention informal requirements (e.g., "the software must not crash due to buffer overflow"). However, these informal specifications are difficult to certify as such. In this context, the need arises for a mechanism for defining security requirements in a simply and unambiguous way.

The CC's *Security Target* (ST) can become a fundamental input to OS communities, improving the OSS software development process by providing clear indications of the contributions expected by the developers to the project [29, 31, 34]. The ST in fact describes the security problems that could compromise the system and identifies the objectives which explain how to counter the security problems. Also, the ST identifies the security requirements that need to be satisfied to achieve the objectives.

Intuitively, the ST assumes a twofold role as a community input: *(i)* it provides guidelines (ST's objectives and requirements), the developers need to follow when contributing to the community; *(ii)* it supports CC-based

evaluation and certification of the OSS. This is due to the fact that, thanks to point *(i)*, OSS systems can be designed and developed by already considering the ST to be used in a subsequent certification. Community developers will then be asked to provide, during the software development process, all documentations and tests required for certification during the software development phase. This solution results in a scenario where security targets become high-level specification documents to be jointly developed by the communities, driving OSS security assurance and security certification processes.

In the next section we illustrate through an example how an ad-hoc security target can be used to provide security specification and requirements. These requirements can then be used to manage the development process of OSS community contributing to the system under development.

5.5 Security driven OSS development: A case study on Single Sign-On

The use case we provide is on an *open source* Single Sign-On [8] solution, which allows users to enter a single username and password to access systems and resources, to be used in the framework of an open source e-service scenario. Single Sign-On (SSO) systems are aimed at providing functionalities for managing multiple credentials of each user and presenting these credentials to network applications for authentication (see Chapter 2).

Starting from a definition of the ST for a SSO system, we identify different trust models and the related set of requirements to be satisfied during the development phase. Then, we turn to the community for the development and informally evaluation of a fully functional SSO system. The ST-based solutions ensures that the software product will be developed with certification in mind.

5.5.1 Single Sign-On: Basic Concepts

Applications running on the Internet are increasingly designed by composing individual *e-services* such as e-Government services, remote banking, and airline reservation systems [15], providing various kind of functionalities, such as, paying fines, renting a car, releasing authorizations, and so on. This situation is causing a proliferation of user accounts: users typically have to log-on to multiple systems, each of which may require different usernames and authentication information. All these accounts may be managed independently by local administrators within each individual system [22]. In other words, users have multiple credentials and a solution is needed to give them the illusion

of having a single identity and a single set of credentials. In the multi-service scenario, each system acts as an independent domain. The user first interacts with a *primary domain* to establish a session with that domain. This transaction requires the user to provide a set of credentials applicable to the domain. The primary domain session is usually represented by an operating system shell executed on the user's workstation. From this primary domain session shell, the user can require services from other *secondary domains*. For each of such requests the user has to provide a set of credentials applicable to the secondary domain she is connecting to.

From the account management point of view, this approach requires independent management of accounts in each domain and use of different authentication mechanisms. In the course of time, several usability and security concerns have been raised leading to a rethinking of the log-on process aimed at co-ordinating and, where possible, integrating user log-on mechanisms of the different domains.

A service that provides such a co-ordination and integration of multiple log-on systems is called *Single Sign-On* [14] platform. In the SSO approach the primary domain is responsible for collecting and managing all user credentials and information used during the authentication process, both to the primary domain and to each of the secondary domains that the user may potentially require to interact with. This information is then used by services within the primary domain to support the transparent authentication to each of the secondary domains with which the user requests to interact. The advantages of the SSO approach include [12, 22]:

- *reduction* of *i)* the *time spent* by the users during log-on operations to individual domains, *ii) failed log-on* transactions, *iii)* the *time used to log-on* to secondary domains, *iv) costs and time* used for users profiles administration;
- *improvement to users security* since the number of username/password each user has to manage is reduced;[7]
- *secure and simplified administration* because with a centralized administration point, system administrators reduce the time spent to add and remove users or modify their rights;
- *improved system security* through the enhanced ability of system administrators to maintain the integrity of user account configuration including the ability to change an individual user's access to all system resources in a co-ordinated and consistent manner;
- *improvement to services usability* because the user has to interact with the same login interface.

---

[7] It is important to note that, while improving security since the user has less accounts to manage, SSO solutions imply also a greater exposure from attacks; an attacker getting hold of a single credential can in principle compromise the whole system.

Also, SSO provides a uniform interface to user accounts management, enabling a coordinated and synchronized management of authentication in all domains.

5.5.2 A ST-based definition of trust models and requirements for SSO solutions

Open source requirements for a SSO are unlikely to be much more detailed than the informal description made in the previous section. Such informal requirements can correspond to different SSO solutions, which could slightly differ in their purposes, depending on the business and trust scenario where they are deployed. In a traditional development process, formal specifications would be used in order to disambiguate this description and lead the development to the desired outcome. As an OSS-targeted alternative, let us show how this can be obtained by the definition of a community-wide Security Target. Namely, we will generate the *Single Sign-On Security Target* (SSO ST) starting by the informal requirements followed in the development of the Central Authentication Service (CAS) [6, 11] and by the Computer Associates eTrust Single Sign-On V7.0 [46] Security Target.

5.5.2.1 Central Authentication Service (CAS)

Central Authentication Service (CAS) [6, 11] is an open source framework developed by Yale University. It implements a SSO mechanism to provide a *Centralized Authentication* to a single server and *HTTP redirections.* The CAS authentication model is loosely based on classic Kerberos-style authentication [35]. When an unauthenticated user sends a service request, this request is redirected from the application to the authentication server (CAS Server), and then back to the application after the user has been authenticated. The CAS Server is therefore the only entity that manages passwords to authenticate users and transmits and certifies their identities. The information is forwarded to the application by the authentication server during redirections by using session cookies.

CAS is composed of pure-Java servlets running over any servlet engine and provides a very basic web-based authentication service. In particular, its major security features are:

1. passwords travel from browsers to the authentication server via an encrypted channel;
2. re-authentications are transparent to users if they accept a single cookie, called *Ticket Granting Cookie* (TGC). This cookie is opaque (i.e., TGC

contains no personal information), protected (it uses SSL) and private (it is only presented to the CAS server);

3. applications know the user's identity through an opaque one-time Service Ticket (ST) created and authenticated by the CAS Server, which contains the result of a hash function applied to a randomly generated value.

Also, CAS credentials are *proxiable*. At start-up, distributed applications get a *Proxy-Granting Ticket* (PGT) from CAS When the application needs access to a resource, it uses the PGT to get a proxy ticket (PT). Then, the application sends the PT to a back-end application. The back-end application confirms the PT with CAS, and also gains information about who proxied the authentication. This mechanism allows "proxy" authentication for Web portals, letting users to authenticate securely to untrusted sites (e.g., student-run sites and third-party vendors) without supplying a password. CAS works seamlessly with existing Kerberos authentication infrastructures and can be used by nearly any Web-application development environment (JSP, Servlets, ASP, Perl, mod_perl, PHP, Python, PL/SQL, and so forth) or as a server-wide Apache module.

5.5.2.2 Single Sign-On Security Target (SSO ST)

The SSO ST can be used to define security requirements for SSO solutions and to drive a certification-oriented SSO implementation (i.e., CAS++ [9]).[8]

As shown in Section 3.10, a security target is composed of 7 sections, each of which needs to be defined for a specific Target Of Evaluation (TOE). Since we have already explained the content of each ST section, here we shall focus on the specificities of the TOE and on the different parts of the SSO ST. The first section of the SSO ST we analyze is the *TOE overview*, which provides the reader with some initial insight about the context we are dealing with, and the type of product we are considering. The SSO ST defines the TOE overview in Figure 5.3.

Example of TOE overview.

- *TOE overview.* This Security Target (ST) defines the Information Technology (IT) security requirements for Single Sign-On secure e-Services. Single Sign-On for secure e-services implements a SSO mechanism to provide a Centralized Authentication to a single server and HTTP redirections. SSO system integrates an authentication mechanism with a Public Key Infrastructure (PKI).

Fig. 5.3: TOE overview

---

[8] The SSO ST presented here is defined starting from the Computer Associates eTrust Single Sign-On V7.0 ST [46] and represents a proof of concept only. It has not been subject to any formal certification process, neither created by any evaluation body.

To gain a better understanding of the TOE, we split the TOE description into two subsections where we describe the TOE type and architecture. Figure 5.4 illustrates the TOE description section and Figure 5.5 depicts CAS architecture which is taken from [27] and used as a template for describing a SSO architecture.

Example of TOE overview.

- *Product type.* While there is an increasing need for authenticating clients to applications before granting them access to services and resources, individual e-services are rarely designed in such a way to handle the authentication process. The reason e-services do not include functionality for checking the client's credentials is that they assume a unified directory system to be present, making suitable authentication interfaces available to client components of network applications. On some corporate networks, all users have a single identity across all services and all applications are directory enabled.
  As a result, users only log in once to the network, and all applications across the network are able to check their unified identities and credentials when granting access to their services. However, on most Intranet and on the open network users have multiple identities, and a solution is needed to give them the illusion of having a single identity and a single set of credentials. *Single Sign-On* (SSO) systems are aimed at providing this functionality, managing the multiple identities of each user and presenting their credentials to network applications for authentication.
- *TOE architecture.* The *Central Authentication Server* (CAS) is designed as a standalone web application. It is currently implemented as several Java servlets and runs through the HTTPS server on `secure.its.yale.edu`. It is accessed via the three URLs described in Figure 5.5 below: the login URL, the validation URL, and the optional logout URL.

Fig. 5.4: TOE description

In the *security problem definition* section of the ST document, we describe the expected operational environment of the TOE, defining the threats and the security assumptions.[9] Table 5.3 shows the threats that may be addressed either by the TOE or its environments, and Table 5.4 shows our assumptions concerning the TOE environment. For the sake of conciseness, we assume that no organizational security policies apply to our case, leaving it to the interested reader to come up with one as an exercise.

Based on the security problems defined in Tables 5.3 and 5.4, the security objectives section of the ST document contains a set of concise statements as a response to those problems. The security objectives we defined for a SSO application are listed in Table 5.5. We have also defined the security objectives for the TOE environment listed in Table 5.6.

---

[9] Since we consider the same security functionalities as in [46], many of the threats and assumptions defined here are taken from [46]
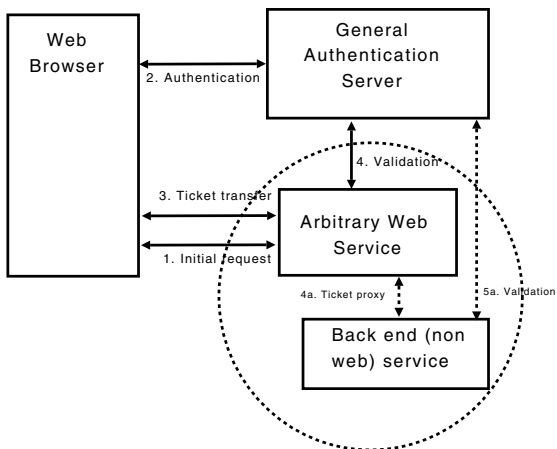
Fig. 5.5: CAS architecture

Threat ID	Threat Description
T.WeakCredentials	Users may select bad passwords, which make the system vulnerable for attackers to guess their passwords and gain access to the TOE.
T.Access	Attackers may attempt to copy or reuse authentication information to gain unauthorized access to resources protected by the TOE.
T.Impersonate	Attackers may impersonate other users to gain unauthorized access to resources protected by the TOE.
T.Mismanage	Administrators may make errors in the management and configuration of security functions of the TOE. Those errors may permit attackers to gain unauthorized access to resources protected by the TOE.
T.BlockSystem	Attacker may attempt to login as an authorized user and gain unauthorized access to resources protected by the TOE. The attacker may login multiple times, thus locking out the authorized user.
T.Reuse	An attacker may attempt to reuse authentication data, allowing the attacker to gain unauthorized access to resources protected by the TOE.
T.Undetected	Attempts by an attacker to violate the security policy and tamper with TSF data may go undetected.
T.NotLogout	A logged-in user may leave a workstation without logging out, which could enable an unauthorized user to gain access to the resources protected by the TOE.
T.CredentialsTransfer	An Attacker may listen to the communication traffic to find any authentication information.

Table 5.3: TOE threats

Assumption ID	Assumption Description
A.Admin	Administrator is trusted to correctly configure the TOE.
A.Trust	It is assumed that there will be no untrusted users and no untrusted software on the Policy Server host.
A.TrustedNetwork	It is assumed that the TOE components communicate over a physically protected Local Area Network.
A.Users	It is assumed that users will protect their authentication data.

Table 5.4: TOE security assmptions

Objective ID	Objective Description
O.Audit	The TOE must provide ways to track unexpected behaviors
O.Authentication	The TOE must identify and authenticate all users before providing them with application authentication information.
O.DenySession	The TOE must be able to deny session establishment based the maximum number of sessions a user can have open simultaneously and an idle time-out.
O.Reauthenticate	The TOE must be able to require the user to be re-authenticated under specified conditions.
O.StrongAuthentication	The TOE must integrate strong authentication mechanism based on two-factor authentication such as a smartcard and biometric properties of the user.
O.Authorization	The TOE shall determine the level of information/services the requester can see/use.
O.AUthoManagement	The TOE shall provide support for authorization management.
O.Provisioning	Before sending any request the TOE shall ensure that it satisfies all the required pre-conditions defined by the administrators.
O.Centralization	User profiles should be maintained within the TOE.
O.SafeTransport	The TOE architecture implies the exchange of user information between the TOE server and services to fulfill authentication and authorization processes. Secure data transfer shall be assured.
O.Control	The TOE shall provide a unique access control point for users who want to request a service.

Table 5.5: TOE security objectives

Objective ID	Objective Description
OE.Admin	Those responsible for the administration of the TOE are competent and trustworthy individuals, capable of managing the TOE and the security of the information it contains.
OE.Install	Those responsible for the TOE must establish and implement procedures to ensure that the hardware, software and firmware components that comprise the system are distributed, installed and configured in a secure manner.
OE.Operation	There must be procedures in place in order to ensure that the TOE will be managed and operated in a secure manner.
OE.Auth	The users must ensure that their authentication data is held securely and not disclosed to unauthorized persons.

Table 5.6: Security objectives for TOE environment

We can finally define the security requirements that need to be satisfied by the TOE in order to reach the defined objectives. This is a crucial step; in a OSS community, it is very important to put in charge of requirements definition someone having a deep understanding of the security objectives. The set of requirements is divided into *security functional requirements* (SFRs) and *security assurance requirements* (SARs), which have been taken respectively from CC Part2 [18], and Part3 [19].

The SFRs we defined for the TOE are listed in Table 5.7. For the sake of simplicity, we shall show only one example of SFR description (see Figure 5.6).

Component	Name
FAU_GEN.1	Audit data generation
FAU_GEN.2	User identity association
FIA_SOS.1	Verification of secrets
FIA_UAU.1	Timing of authentication
FIA_UAU.2	User authentication before any action [Primary Authentication]
FIA_UAU.5	Multiple authentication mechanisms
FIA_UAU.6	Re-authenticating [Primary Authentication]
FIA_UID.2	User identification before any action [Primary Authentication]
FTA_SSL.3	TSF-initiated termination
FTA_TSE.1	TOE session establishment
FTP_ITC.1	Inter-TSF trusted channel

Table 5.7: Security Functional Requirements for the TOE

FAU_GEN.1 Audit data generation
Hierarchical to: No other components.
FAU_GEN.1.1 The TSF shall be able to generate an audit record of the following auditable events:

a)   Start-up and shutdown of the audit functions;
b)   All auditable events for the [not specified] level of audit; and
c)   [**the following auditable events:**

- **User login/logout**
- **Failed attempts to login**
- **User session timeout**].

Fig. 5.6: Example of SFR

Looking at the original requirement defined by CC Part2 [18] for the example in Figure 5.6, it is clear that we need to change FAU_GEN.1.1 before being able to adopt it in our ST. The first operation is a selection operation. CC Part2 [18] defines the point *(b)* of FAU_GEN.1.1 as follows: "*All auditable events for the [selection, choose one of: minimum, basic, detailed, not specified] level of audit*". To adapt it to our case, we had to select one of the given alternatives. The second operation is an assignment operation in which more parameters can be specified. The original version of point *(c)* in CC part2 [18] is defined as *[assignment: other specifically defined auditable events]*. To use it in our ST, we need to specify the auditable events of our interest.

The security assurance requirements in SSO ST are the assurance components of Evaluation Assurance Level 2 (EAL2) taken from CC Part3 [19]. None of the assurance components has been refined. The EAL2 assurance requirements are listed in Table 5.8.

We are now ready to consider another important section of the SSO ST, namely the *TOE specification summary*, where more details about the security functions of the TOE are given. This section also provides also a mapping between the security functions of the TOE and the SFRs. The example in Figure 5.7 describes the auditing mechanism of a SSO solution taken from [27].

5.5.2.3 Trust Models

Trust models are the basis for designing interoperable systems. A *trust model* describes a software system by defining its underlying environment as well as its components, and the rules governing their interactions. Here, we focus on the definition of trust models for SSO environments, based on the functionalities that these environments support. We do not only consider the security functionalities and requirements in SSO ST (described in the previous sec-

Assurance Class	Assurance components
ADV: Development	ADV_ARC.1 Security architecture description
	ADV_FSP.2 Security-enforcing functional specification
	ADV_TDS.1 Basic design
AGD: Guidance documents	AGD_OPE.1 Operational user guidance
	AGD_PRE.1 Preparative procedures
ALC: Life-cycle support	ALC_CMC.2 Use of a CM system
	ALC_CMS.2 Parts of the TOE CM coverage
	ALC_DEL.1 Delivery procedures
ASE: Security Target evaluation	ASE_CCL.1 Conformance claims
	ASE_ECD.1 Extended components definition
	ASE_INT.1 ST introduction
	ASE_OBJ.2 Security objectives
	ASE_REQ.2 Derived security requirements
	ASE_SPD.1 Security problem definition
	ASE_TSS.1 TOE summary specification
ATE: Tests	ATE_COV.1 Evidence of coverage
	ATE_FUN.1 Functional testing
	ATE_IND.2 Independent testing - sample
AVA: Vulnerability assessment	AVA_VAN.2 Vulnerability analysis

Table 5.8: EAL2 Security Assurance Requirements

tion), but adopt a wider perspective, including also functional aspects of the SSO solutions. We identify three models.

Authentication and Authorization Model (AAM). This model is one of the traditional security/trust models describing all frameworks that provide authentication and authorization features [42]. It represents the basic mechanism in which a user requires an access to a service that checks the users' credentials to decide whether access should be granted or denied. The AAM model identifies two major entities: *users,* which request accesses to resources, and *services,* potentially composed by a set of intra-domain services, which share these resources. This model is based on the classic client-server architecture and provides a generic protocol for authentication and authorization processes.

Federated Model (FM). This model represents one of the emergent security/trust models in which several homogeneous entities interact to provide security services, such as identity privacy and authentication. The FM model identifies two major entities: *users,* which request accesses to resources, and *services,* which share these resources. The major difference with the previous model resides in the service definition and composition: in federated systems the services are distributed on different domains and they are built on the same level allowing mutual trust and providing functionalities as cross-authentication [32].

Full Identity Management Model (FIMM). This model represents one of the most challenging security and privacy/trust models. Besides dealing

---

AU.1: Auditing generation

CAS uses Log4J to write event logs to either flat files or to an Oracle table (source=
`http://ja-sig.org/wiki/pages/viewpagesrc.action?pageId=969`).
The logged events include:

1. sees login screen
2. successful authentication
3. requested warnings
4. unsuccessful authentication
5. authentication warning screen presented
6. inactivity timeout
7. wall clock timeout (TGT)
8. bad attempt lockout
9. logout

AU.2: Auditing information

Each log entry includes:

- date / time
- event type (e.g., TICKET_GRANT, TICKET_VALIDATE)
- username (if applicable)
- client IP address (if applicable)
- result (SUCCESS/FAILURE)
- service_url (if applicable)
- service ticket (if applicable)

These logs are used mainly for usage reports and for security reviews and incident response. The requirements of the security group are:

- ability to identify who was logged on based on IP address
- ability to identify who was logged on based on date and time
- online logs retained for at least two weeks
- archived logs retained for at least one quarter

This function contributes to satisfy the security requirements FAU_GEN.1 and FAU_GEN.2

Fig. 5.7: Example TOE specification summary describing the Logs mechanism of CAS

with all the security aspects covered by the previous two models, it provides mechanisms for identity and account management and privacy protection [1, 38]. The FIMM model identifies three major entities: *users,* which request accesses to resources, *services,* which share these resources, and *identity manager,* which gives functionalities to manage users identities.

Requirement	AAM Model	FM Model	FIMM Model
Authentication	X	X	X
Strong Authentication	X	X	X
Authorization	X		X
Provisioning	X		X
Federation		X	X
C.I.M. (Centralized Identity) Management	X		X
Client Status Info	X	X	X
Single Point of Control	X		
Standard Compliance	X	X	X
Cross-Language availability	X	X	X
Password Proliferation Prevention	X	X	X
Scalability	X	X	X

Table 5.9: Requirements categorization basing on the specific trust model.

5.5.3 Requirements

In order to compare our ST with a traditional analysis document, we need a requirement list for a Single Sign-On solution. The requirements that a SSO should satisfy are more or less well known within the security community, and several SSO projects published partial lists.[10] However, as is typical of the OSS development source, the requirements elicitation phase has been informal and no complete list of requirements has been published. A comparative analysis of the available lists brought us to formulating the following requirements (including the security ones). Focusing on security requirements, this informal list of requirements can be substituted by a ST-based requirements definition, which makes the development process stable and unambiguous. For each requirement we also report the trust model (AMM, FM, FIMM) to which it refers.[11]

Authentication (AAM,FM,FIMM).   A major requirement of a SSO system is to provide an authentication mechanism. Usually authentication is performed by means of a classic username/password log-in, whereby a user can be unambiguously identified. Authentication mechanisms should usually be coupled with a logging and auditing process to prevent and, eventually, discover malicious attacks and unexpected behaviors. From a purely

---

[10]   For an early attempt at a SSO requirements list, see middleware.internet2.edu/webiso/docs/draft-internet2-webiso -requirements-07.html.

[11] Note that different trust models fulfill a different set of requirements (see Table 5.9). SSO solution, therefore, should be evaluated by taking into consideration only the requirements supported by the corresponding trust model.

software engineering point of view, authentication is the only "necessary and sufficient" functional requisite for a SSO architecture.

Strong Authentication (AAM,FM,FIMM). For highly secure environments, the traditional username/password authentication mechanism is not enough. Malicious users can steal a password and impersonate the user. New approaches are therefore required to better protect services against unauthorized accesses. A good solution to this problem integrates username/password check with a strong authentication mechanism based on two-factor authentication such as a smartcard and biometric properties of the user (fingerprints, retina scans, and so on).

Authorization (AAM,FIMM). After the authentication process, the system can determine the level of information/services the requester can see/use. While applications based on domain specific authorizations can be defined and managed locally at each system, the SSO system can provide support for managing authorizations (e.g., role or profile acquisitions) that apply to multiple domains.

Provisioning (AAM,FIMM). Provisions are those conditions that need to be satisfied or actions that must be performed before a decision is taken [7]. A provision is similar to a pre-condition (see Chapter 4) it is responsibility of the user to ensure that a request is sent in an environment satisfying all the pre-conditions. The non-satisfaction of a provision implies a request to the user to perform some actions.

Federation (FM,FIMM). The concept of *federation* is strictly related to the concept of *trust*. A user should be able to select the services that she wants to federate and de-federate to protect her privacy and to select the services to which she will disclose her own authorization assertions.

C.I.M. (Centralized Identity Management) (AAM,FIMM). The centralization of authentication and authorization mechanisms and, more generally, the centralization of identity management implies a simplification of the user profile management task. User profiles should be maintained within the SSO server thus removing such a burden from local administrators. This allows a reduction of costs and effort of user-profile maintenance and improves the administrators' control on user profiles and authorization policies.

Client Status Info (AAM,FM,FIMM). The SSO system architecture implies the exchange of user information between SSO server and services to fulfill authentication and authorization processes. In particular, when the two entities communicate, they have to be synchronized on what concern the user identity; privacy and security issues need to be addressed. Different solutions of this problem could be adopted involving either the transport (e.g., communication can be encrypted) or the application layer.

Single Point of Control (AAM). The main objectives of a SSO implementation are to provide a unique access control point for users who want to request a service, and, for applications, to delegate some features from business components to an authentication server. This point of control

should be unique in order to clearly separate the authentication point from business implementations, avoiding the replication and the ad-hoc implementation of authentication mechanisms for each domain. Note that every service provider will eventually develop its own authentication mechanism.

Standard Compliance (AAM,FM,FIMM).   It is important for a wide range of applications to support well-known and reliable communication protocols. In a SSO scenario, there are protocols for exchanging messages between authentication servers and service providers, and between technologies, within the same system, that can be different. Hence, every entity can use standard technologies (e.g., X.509, SAML for expressing and exchanging authentication information and SOAP for data transmission) to interoperate with different environments.

Cross-Language availability (AAM,FM,FIMM).   The widespread adoption of the global Internet as an infrastructure for accessing services has consequently influenced the definition of different languages and technologies used to develop these applications. In this scenario, a requisite of paramount importance is integrating authentication to service implementations written in different languages, without substantial changes to service code. The first step in this direction is the adoption of standard communication protocols based on XML.

Password Proliferation Prevention (AAM,FM,FIMM).   A well-known motivation for the adoption of SSO systems is the prevention of password proliferation so to improve security and simplify user log-on actions and system profile management.

Scalability (AAM,FM,FIMM).   An important requirement for SSO systems is to support and correctly manage a continuous growth of users and subdomains that rely on them, without substantial changes to system architecture.

5.5.4 A case study: CAS++

Our SSO ST is meant to drive the development of open source SSO systems. As a case study, let us it as a guide to extend the *Central Authentication Service* (CAS) [6, 11] to an enhanced version we will call CAS++.[12]. Our extension integrates the CAS system with the authentication mechanism implemented by a *Public Key Infrastructure* (PKI) [33]. CAS++ implements a fully multi-domain stand-alone server that provides a simple, efficient, and reliable SSO mechanism through HTTP redirections, focused on user privacy (opaque cookies) and security protection. CAS++ permits a centralized man-

---

[12] Of course CAS++ is not the only implementation available on the Net. In particular, *SourceID* [47], an Open Source implementation of the SSO Liberty Alliance, *Java Open Single Sign-On* (JOSSO) [28], and *Shibboleth* [45] are other available SSO solutions.
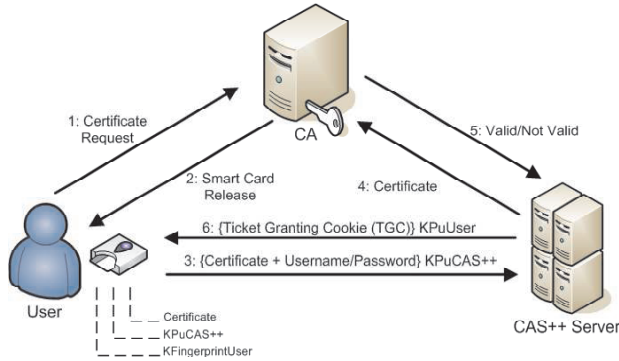
Fig. 5.8: CAS++ certificate-based authentication flow

agement of user profiles granting access to all services in the system with a unique pair username/password. The profiles repository is stored inside the SSO server application and is the only point where users credentials/profiles are accessed, thus reducing information scattering. In our implementation, services do not need an authentication layer because this feature is managed by CAS++ itself.

CAS++ relies on standard protocols such as SSL, for secure communications between the parties, and on X.509 digital certificates for credentials exchange. Besides being a "pure-Java" module like its predecessor, CAS++ is a fully J2EE compliant application integrable with services coded with every web-based implementation language. It enriches the traditional CAS authentication process through the integration of biometric identification (by fingerprints readers) and smart card technologies in addition to traditional username/password mechanism, enabling two authentication levels.

CAS++ strong authentication process flow is composed of the following steps (see Figure 5.8):[13]

1. the user requests an identity certificate to the CA (Certification Authority);
2. the user receives from the CA a smart card that contains a X.509 identity certificate, signed with the private key of the CA, that certifies the user identity. The corresponding user private key is encrypted with a symmetric algorithm (e.g., 3DES) and the key contained inside the smart card can be decrypted only with a key represented by user fingerprint (KFingerprintUser) [23];

---

[13] Note that, the first two actions are performed only once when the user requests the smart card along with an identity certificate.
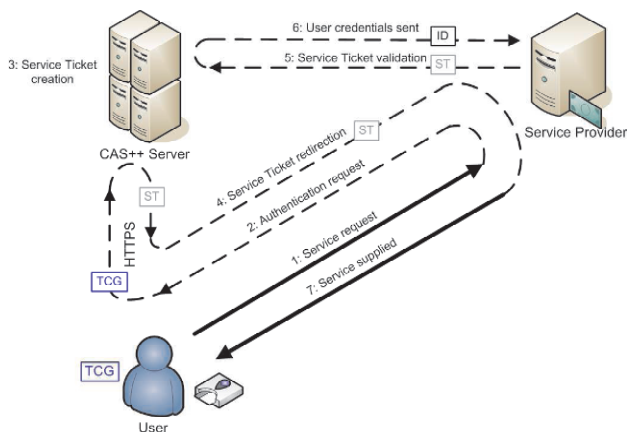
Fig. 5.9: CAS++ information flow for service request evaluation

3. to access a service the public key certificate, along with the pair user-
   name/password, is encrypted with the CAS++ public key (KPuCAS++)
   and sent to CAS++;
4. CAS++ decrypts the certificate with its private key, verifies the signature
   on the certificate with the CA public key, and verifies the validity of this
   certificate by interacting with the CA;
5. CAS++ retrieves from the CA information about the validity of the user
   certificate encrypted with KPuCAS++;
6. if the certificate is valid, CAS++ extracts the information related to the
   user, creates the ticket (TGC, Ticket Granting Cookie) and returns it to
   the user encrypted with the public key of the user (KPuUser). At this
   point, to decrypts the TGC, the user must retrieve the private key stored
   inside the smart card by mean of her fingerprint. As soon as the card
   is unlocked, the private key is extracted and the TGC decrypted. This
   ticket will be used for every further access, in the same session, to any
   application managed by the CAS++ Single Sign-On server.

At this point, for every further access in the session, the user can be
authenticated by the service providing only the received TGC without any
additional authentication action.[14]

The service access flow, that takes place over secure channels and is similar
to the one in CAS, is composed of the following steps (see Figure 5.9):

1. the user, via a web browser, requests access to the service provider;
2. the service provider requests authentication information through a HTTP
   redirection to the CAS++ Server;

---

[14] Note that the TGC lifetime should be relatively short to avoid conflicts with the
CA's certificate revocation process, which could cause unauthorized accesses.

3. the CAS++ Server retrieves the user TGC and the service requested URL. If the user has been previously authenticated by CAS++ and has the privilege to access the service a Service Ticket is created;
4. the CAS++ Server redirects the user browser to the requested service along with the ST;
5. service receives the ST and check its validity sending it to the CAS++ Server;
6. if the ST is valid the CAS++ Server sends to the Service an XML file with User's credentials;
7. the user gains access to the desired service.

5.5.4.1 Evaluating CAS++ Against the ST Document

CAS++ is based on the Authentication and Authorization Model. Also, CAS++ fulfills most of our ST requirements; specifically, it provides a central point of control to manage authentication, authorization, and user profiles.[15] Furthermore, CAS++ enriches the traditional CAS authentication process with the integration of biometric identification (via fingerprints readers) and smart card technologies and it is planned to include provisioning features in future releases. Note that, the lower level of CAS++ system is language independent and relies on traditional established standards, such as HTTP, SSL and X.509, without adopting emerging ones, such as SOAP and SAML. Focusing on client status info, all communications between user browser, services providers and authentication server in CAS++ scenario are managed by the exchange of opaque cookies and by the use of encrypted channels. Finally, since CAS++ development has been driven by SSO ST, the process of certifying CAS++ based on the CC standards, becomes straightforward.

5.6 Conclusions

Software Assurance (SwA) relates to the level of confidence that software functions as intended and is free of faults. In open source development, many stakeholders have a vested interest in the finalization of a standard assurance process for open source encompassing the areas of functionality, reliability, security, and interoperability. Most major OSS projects have some kind of assurance process in place which includes specific code reviews, and in some cases code analysis. Indeed, anecdotal evidence shows that code review is

---

[15] The centralization of users profiles affects system scalability. A solution that provides a balance between centralization and scalability needs is under study.

provenly faster and more effective in large communities.[16] As far as security certification is concerned, the process must be as public as possible, involving academia, the private sector, nonprofit organizations, and government agencies. Since no formal requirements elicitation is normally done in the OSS development process, the CC ST document can be used to collect and focus the stakeholders' view on the software product's desired security features.

References

1. C.A. Ardagna, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. Towards privacy-enhanced authorization policies and languages. In *Proc. of the 19th IFIP WG11.3 Working Conference on Data and Application Security*.
2. C.A. Ardagna, E. Damiani, and F. Frati. Focse: An owa-based evaluation framework for os adoption in critical environments. In *Proc. of the 3rd IFIP Working Group 2.13 Foundation on Open Source Software*, Limerick, Ireland, June 2007.
3. C.A. Ardagna, E. Damiani, F. Frati, and M. Madravio. *Open source solution to secure e-government services*. Encyclopedia of Digital Government, Idea Group Inc., 2006.
4. C.A. Ardagna, E. Damiani, F. Frati, and M. Montel. Using open source middleware for securing e-gov applications. In *Proc. of The First International Conference on Open Source Systems*, Genova, Italy, July 2005.
5. C.A. Ardagna, E. Damiani, F. Frati, and S. Reale. Adopting open source for mission-critical applications: A case study on single sign-on. In *Proc. of the 2nd IFIP Working Group 2.13 Foundation on Open Source Software*, Como, Italy, June 2006.
6. P. Aubry, V. Mathieu, and J. Marchal. Esup-portal: open source single sign-on with cas (central authentication service). In *Proc. of the EUNIS04 - IT Innovation in a Changing World*.
7. C. Bettini, S. Jajodia, X. Sean Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *Proc. of the 28th Conference on Very Large Data Bases (VLDB 2002)*, Honk Kong, China, August 2002.
8. D.A. Buell and R. Sandhu. Identity management. *IEEE Internet Computing*, 7(6).
9. S. De Capitani di Vimercati F. Frati P. Samarati C.A. Ardagna, E. Damiani. Cas++: an open source single sign-on solution for secure e-services. In *Proc. of the 21st IFIP TC-11 International Information Security Conference*, Karlstad, Sweden, May 2006.
10. A. Capiluppi, P. Lago, and M. Morisio. Characterizing the oss process: a horizontal study. In *Proc. of the 7th European Conference on Software Maintenance and Reengineering*, Benevento, Italy, March 2003.
11. *Central Authentication Service*.
12. J. De Clercq. Single sign-on architectures. In *Proc. of the International Conference on Infrastructure Security (InfraSec 2002)*.
13. C. Cowan. Software security for open-source systems. *IEEE Security & Privacy*, 1(1):38–45, January-February 2003.

---

[16] The `Interbase` database software contained a backdoor access (username `politically` and password `correct`) which was found soon after it was released as open source, thanks to large-scale code review by the interested community.

14. B. Galbraith et al. *Professional Web Services Security*. Wrox Press, 2002.
15. S. Feldman. The changing face of e-commerce. *IEEE Internet Computing*, 4(3).
16. J. Feller and B. Fitzgerald. *Understand Open Source Software Development*. Addison-Wesley, 2002.
17. FLOSSmole. Collaborative collection and analysis of free/libre/open source project data. `ossmole.sourceforge.net/`.
18. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components*, 2007. `http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R2.pdf`.
19. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components*, 2007. `http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R2.pdf`.
20. Free Software Foundation. `www.fsf.org/`.
21. B. Golden. *Succeeding with Open Source*. Addison-Wesley, 2004.
22. The Open Group. *Single Sign-On*.
23. F. Hao, R. Anderson, and J. Daugman. Combining cryptography with biometrics effectively. In *Technical report, Cambridge University - Computer Laboratory Technical Report UCAM-CL-TR-640*.
24. A. Hars and O. Shaosong. Working for free? motivations of participating in open source projects. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, USA, January 2001.
25. S.A. Hissam, D. Plakosh, and C. Weinstock. Trust and vulnerability in open source software. In *IEE Proceedings - Software*, volume 149, pages 47–51, February 2002.
26. Open Source Initiative. *The Open Source Definition*, July 2006. `opensource.org/docs/osd/`.
27. JA-SIG. Ja-sig central authentication service. `www.ja-sig.org/products/cas/`.
28. *Java Open Single Sign-On (JOSSO)*.
29. F. Keblawi and D. Sullivan. Applying the common criteria in systems engineering. *IEEE Security and Privacy*, 4(2):50–55, March 2007.
30. A.M. St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly Media, Inc., 2004.
31. J. Lee, S. Lee, and B. Choi. A cc-based security engineering process evaluation model. In *Proc. of the 27th Annual international Conference on Computer Software and Applications (COMPSAC 2003)*, Dallas, Texas, USA, November 2003.
32. *Liberty Alliance Project*.
33. U.M. Maurer. Modelling a public-key infrastructure. In *Proc. of the 4th European Symposium on Research in Computer Security (ESORICS 1996)*, Rome, Italy, September 1996.
34. D. Mellado, E. Fernandez-Medina, and M. Piattini. A common criteria based security requirements engineering process for the development of secure information systems. *Computer Standards & Interfaces*, 29(2):244–253, February 2007.
35. B.C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1995.
36. Open Source Initiative (OSI). `opensource.org/`.
37. C. Payne. On the security of open source software. *Info Systems Journal*, 12:61–78, 2002.
38. *PRIME (Privacy and Identity Management for Europe)*.
39. PuTTY. A free telnet/ssh client. `www.chiark.greenend.org.uk/$\sim$sgtatham/putty/`.

40. Business Readiness Rating. *Business Readiness Rating for Open Source*, 2005. `www.openbrr.org/wiki/images/d/da/BRR\_whitepaper\_2005RFC1.pdf`.

41. E.S. Raymond. *The cathedral and the bazaar*. Available at: www.openresources.com/documents/cathedral-bazaar/, August 1998.

42. P. Samarati and S. De Capitani di Vimercati. *Foundations of Security Analysis and Design*, chapter Access Control: Policies, Models, and Mechanisms, pages 137–196. Springer Berlin / Heidelberg, 2001.

43. W. Scacchi, J. Feller, B. Fitzgerald, S.A. Hissam, and K. Lakhani. Understanding free/open source software development processes. *Software Process: Improvement and Practice*, 11(2):95–105, 2006.

44. K.S. Shankar and H. Kurth. Certifying open source: The linux experience. *IEEE Security & Privacy*, 2(6):28–33, November-December 2004.

45. *Shibboleth Project*.

46. Sygnacom solutions. *Computer Associates eTrust Single Sign-On V7.0 Security Target V2.0*, October 2005. `www.commoncriteriaportal.org/files/epfiles/ST\_VID3007-ST.pdf`.

47. *SourceID Open Source Federated Identity Management*.

48. Cluster SSH. Cluster admin via ssh. `sourceforge.net/projects/clusterssh`.

49. I. Stamelos, L. Angelis, A. Oikonomou, and G.L. Bleris. Code quality analysis in open source software development. *Info Systems Journal*, 12:43–60, 2002.

50. V. Torra. The weighted owa operator. *International Journal of Intelligent Systems*, 12(2).

51. D.A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. Available : http://www.dwheeler.com/secure-programs/, 2003.

52. D.A. Wheeler. *Free-Libre/Open Source Software (FLOSS) and Software Assurance/Software Security*, December 2006. `www.dwheeler.com/essays/oss\_software\_assurance.pdf`.

53. WinSCP. Free sftp and scp client for windows. `winscp.net/eng/index.php`.

54. R.R. Yager. On ordered weighted averaging aggregation operators in multi-criteria decision making. *IEEE Transaction Systems, Man, Cybernetics*, 18(1).

55. T. Ylönen. Ssh - secure login connections over the internet. In *Proc. of the Sixth USENIX Security Symposium*, July.

Chapter 6
Case Study 1: Linux certification

Abstract In this chapter we examine the SuSE Linux Enterprise Server (SLES) CC certification process in detail. SLES was one of the first open source operating systems to be certified with EAL 2, 3 and 4. Throughout the chapter, we use SuSE Linux Enterprise Server version 8 with Service Pack 3 for our practical experiments. The certification of SLES8 was a joint effort between IBM and SuSE [16]. As a by-product of this case study, we shall discuss the matching techniques for associating the ST security functions to tests, discussing the mapping of SLES8 security functions to Linux Test Project (LTP) test suites. We rely on an extended set of test suites that have been used for SLES8 EAL3 certification [8].

6.1 The Controlled Access Protection Profile and the SLES8 Security Target

As we have seen in the previous chapters, the Common Criteria certification is based on the notion that security is *context dependent.* In other terms, CC (rather sensibly) assumes that any meaningful statement about the security level of a software product can be made only by considering the context in which the software product will be operating. Based on this idea, CC certification cannot be sought for a software product line, and even less for a family of heterogeneous products; rather, it must be obtained for a specific software distribution.

As a consequence, in the past, there has been no attempt to get a certification for Linux as such. By contrast, conformance to a *Controlled Access Protection Profile* (CAPP) has been claimed for specific distributions, such as, the SuSE Linux Enterprise Server v8 (SLES8). In other words, the authors of a SLES8-specific Security Target (ST) claimed that SLES8 fully satisfies the security requirements of CAPP.

The CAPP profile itself was released by the *Information Systems Security Organization* (ISSO) as part of its program to promote security standards for information systems [14]. According to ISSO, "CAPP-conformant products support access controls that are capable of enforcing access limitations on individual users and data objects. CAPP-conformant products also provide an audit capability which records the security-relevant events which occur within the system" [14]. Below, we shall analyze both the CAPP protection profile and SLES8 security target, to highlight the strict correspondence between the different sections of these two documents. Establishing a clear mapping between the requirements of the Protection Profile and the features of the Security Target is indeed a key point in the CC certification procedure. Before focusing on the mapping, however, let us take a closer look to SLES8 features.

6.1.1 SLES8 Overview

SuSE SLES8 is based on UnitedLinux v1.0. UnitedLinux is an effort of a consortium including SuSE, SCO, Turbolinux and Conectiva, whose main goals were to stop the proliferation of distributions within the Linux community and give application developers a larger target of common Linux features and functions. In the Spring of 2002, the UnitedLinux board put together a base technical specification, which resulted in the release of UnitedLinux v1.0 on November 19th, 2002. The technical requirements for UnitedLinux v 1.0 included, among others, a Linux kernel 2.4.18 or higher, the GNU C Library 2.2.5, the C compiler GCC 3.1, window environment and GUI based on XFree86 4.2 and KDE 3.0.[1]

The SLES family of servers share many features with the other Linux based operating systems, such as, providing services for several users at the same time, file access mechanisms, users data protection, privileged mode to administrative users and so forth. SuSE Linux has a reputation for technical excellence and it stands out in the area of systems management. In particular, SuSE YaST2 (*Yet Another Setup Tool*) administrative tool is a GUI installer that has gained wide acceptance thanks to its good support of cluster-wide *Logical Volume Management* (LVM). SuSE also includes an advanced event logging system.

In summary, the SLES8 includes the following features:

- A complete set of software and tools to build clusters and server farms.
- Some enhancements to the scheduler to improve process scheduling on multiprocessor systems.

---

[1] The SLES8 we use to reproduce the certification testing includes a Linux kernel v2.4.

- Asynchronous Input/Output, to minimize waiting on I/O on large, heavy-duty systems.

and a number of enhancements aimed at improving Linux's reliability and avoiding downtime. SLES8 reliability-related features include:

- A standard POSIX-compliant event logging and notification capability.
- Dynamic probes for profiling and debugging, supporting dynamic insertion of breakpoints in code.
- Non-disruptive and tailored dumping of system data.
- Advanced toolkit to record and trace system events.
- Hotplug PCI support, enabling the addition or removal of attached devices without system restart.

Among the SLES8 features, we are mainly interested in SLES8 security features regarding access control. SLES, in fact, has added considerable security extensions to accommodate the security needs of its users [7]. Much effort has been devoted to making SLES8 suitable for sectors where security is critical. SLES8 security features include: *(i) Kerberos* (TM), a strong network authentication protocol originally developed at MIT, and *(ii) Bastille*, a system hardening application and firewall support to separate secure areas of the system from less restricted areas. More importantly to our purposes, SLES8 includes robust and scalable file systems, with full support for file system Access Control Lists (ACLs), where each data object stored on a SLES8 system is assigned a description of the access rights related to it. As a consequence, ACLs enable SLES8 to support advanced access control models, such as, Discretionary Access Control (DAC) (see Chapter 2).

6.1.2 Target of Evaluation (TOE)

SLES8 has several advanced security features, which allow it to be competitive with proprietary server solutions, including some enabling advanced access control models. However, listing such features is not enough to gain user acceptance, especially if a server must host mission- or safety-critical applications. To demonstrate the quality of the implementation of its access control features and their suitability to achieve the security goals related to access control, SuSE needed to certify that SLES8 satisfies the security requirements of a Protection Profile specific to access control, the *Controlled Access Protection Profile* (CAPP).[2]

---

[2] SLES8 was not able to satisfy the CAPP requirements "out of the box"; however only some minor modifications were required.

6.1.3 Security environment

The security environment sets by the PP and ST documents specifies all the
security concerns that a software product has to deal with. This includes
threats, organizational security policy and security assumptions. The exam-
ples used here are taken from CAPP [14] and SLES8 ST [7]. Based on the
intended use of SLES8 and the security environment defined in CAPP, SLES8
ST has identified its specific security environment.

6.1.3.1 Threats

Threats generally identify any danger that could compromise a system's as-
sets. The CAPP, however, does not specify any explicit threats, because
threats can be defined only by knowing how the TOE will be used. Based on
the intended usage of SLES8 ST, the assets to be protected include the in-
formation stored, processed or transmitted by the TOE (SLES8) [7]. Threats
can be classified in two categories:

- Threats to be countered by the TOE, that is, threats which exploit weak-
  nesses in the TOE itself.
  Example: *T.UAACCESS* An authorized user of the TOE may access
  information resources without having permission from the person who
  owns, or is responsible for, the information resource for the type of ac-
  cess.
- Threats to be countered by the TOE environment, that is, threats which
  exploit the weaknesses of the TOE environment.
  Example: *TE.HW_SEP* An attacker (possibly, but not necessarily, an
  unauthorized user of the TOE) with legitimate physical access to the
  hardware the TOE is running on; alternatively, some environmental con-
  ditions may cause the underlying hardware functions of the hardware the
  TOE is running on to not provide sufficient capabilities to support the
  self protection of the TSF from unauthorized programs. Note that, this
  also covers people with legitimate access to the TOE hardware causing
  such a problem accidentally without malicious intent.

The first example provided deals with a threat which affects the TOE di-
rectly, since it concerns the access mechanisms of the TOE itself. The second
threat, instead, affects the TOE indirectly, because it concerns the organiza-
tional policy of who has access to the TOE itself.

6.1.3.2 Organizational Security Policies

The *Organizational Security Policies* (OSPs) are all the procedures defined by
the organization deploying the certified software product to protect sensitive

data [14]. In the following example, we show how the CAPP defines an OSP and how the ST responds to it.

CAPP	P.NEED_TO_KNOW: The system must limit access to, modification, and deletion of the information contained in protected resources to those authorized users which have a real need to know for that information.
SLES8	P.NEED_TO_KNOW: The organization must define a discretionary access control [12] policy on a need-to-know basis which can be modeled based on: • the owner of the object; and • the identity of the subject attempting the access; and • the implicit and explicit access rights to the object granted to the subject by the object owner or an administrative user.

Table 6.1: Example of an Organizational Security Policy defined in the CAPP and SLES8 ST

Table 6.1 shows the difference between the PP and the ST definition of the OSPs. Specifically, the CAPP mentions a generic access control mechanism; any system that implements an access control mechanism with the needed features will qualify. By contrast, the ST responds providing more detailed information that is largely SLES8 dependent.

6.1.3.3 Assumptions

The CAPP considers three types of assumptions, related to different aspects *physical, personnel and connectivity*. SLES8 ST reports the assumptions as they are in the CAPP, because they are not part of the TOE itself. An example of assumption related to available expertise is reported below:

A.MANAGE It is assumed that there are one or more competent individuals in charge of managing the TOE and the security of the information it contains.

6.1.4 Security objectives

Based on the statements defined in the security environment, the security objectives provide statements to address and counter the identified threats and maintain the OSP. CAPP distinguishes between two types of security objectives: *(i)* security objectives for the TOE that are linked directly to it, and *(ii)* security objectives related to the TOE's environment. Security objectives listed in the CAPP and the ST are generally identical. To appreciate the difference, let us now spell out an example of a security objective for the TOE:

O.AUTHORIZATION The TOE must ensure that only authorized users gain access to the TOE and its resources

and an example of a security objective for the TOE environment:

OE.ADMIN Those responsible for the administration of the TOE are competent and trustworthy individuals, fully capable of managing the TOE and the security of the information it contains.

6.1.5 Security requirements

In the security environment and security objectives sections both CAPP and SLES8 ST were discussed and compared to demonstrate the close similarity between their contents. However, from now on we will refer to the SLES8 ST only. In case the CAPP is needed, it will be stated explicitly.

The security requirements section is perhaps the one that occupies the largest portion of SLES8 ST. The rationale behind this choice is that security requirements need to be described clearly, including all the needed details to avoid any ambiguous interpretations. As we have seen throughout the book, security requirements can be classified in two categories, namely SFRs and SARs. SLES8 ST defines some SFRs and SARs that are not mentioned in the CAPP, because SLES8 needed some specific extensions [7]. Namely, the SLES8 ST identifies the set of SFRs from six SFR classes.

- Security Audit (FAU).
- Cryptographic Support (FCS).
- User Data Protection (FDP).
- Identification and authentication (FIA).
- Security Management (FMT).
- Protection of the TOE Security Functions (FPT).

Whereas the SFRs defined in Part 2 of CC provide a generic set of SFRs, the ST needs to describe the SFRs of a product-specific code base. To help doing so, the CC standard includes a list of operations which are used to refine SFRs and SARs. The following example shows how a SFR defined in Part 2 of CC has been refined first by CAPP and then by SLES8 ST to fit the required functionality.

FDP_ACF.1.1 is the first element of the **Discretionary Access Control [12] Functions** component. It has been defined in the Part 2 of CC as follows:

The TSF shall enforce the [assignment: access control SFP (Security Functional Policy)] to objects based on the following: [assignment: list of sub-

jects and objects controlled under the indicated SFP, and for each, the SFP-relevant security attributes, or named groups of SFP-relevant security attributes].

The assignment operation allows the authors of the PP and ST to explicitly assign parameters that are relevant to the context of the TOE. The CAPP authors have assigned the *FDP_ACF.1.1* parameters as follows:

The TSF shall enforce the Discretionary Access Control [12] Policy to objects based on the following:

1. *The user identity and group membership(s) associated with a subject; and*
2. *The following access control attributes associated with an object: [assignment: List access control attributes. The attributes must provide permission attributes with:*
   2.1. *the ability to associate allowed or denied operations with one or more user identities;*
   2.2. *the ability to associate allowed or denied operations with one or more group identities; and*
   2.3. *defaults for allowed or denied operations.]*

Notice that in this improved definition of CAPP, some new parameters have been added to *FDP_ACF.1.1*. Namely, the **access control SFP** was substituted with the type of access control required by CAPP, which is the **Discretionary Access Control** [12]. The second parameter is the list of subjects and objects controlled by the chosen SFP. In this case, the CAPP specifies several alternatives such as user identity and group memberships, as well as defining a new assignment that needs to be solved by the ST. Correspondingly, the SLES8 ST has specified new parameters to handle the assignment as follows.

The TSF shall enforce the Discretionary Access Control Policy to system objects based on the following information:

1. *The effective user identity and group membership(s) associated with a subject,* and
2. *The following access control attributes associated with an object:*

File system objects	• POSIX ACLs and permission bits.
	• (ACLs can be used to grant or deny access to the granularity of a single user or group using Access Control Entries. Those ACL entries include the standard Unix permission bits. Posix ACLs can be used for file system objects within the ext3 file system).
Access rights for file system objects are	• read
	• write
	• execute (ordinary files)
	• search (directories)
IPC objects	• permission bits
Access rights for IPC objects are	• read
	• write

When *FDP_ACF.1.1* was used in SLES8 ST, the authors have added some new parameters that are specific to SLES8. Those parameters concerned the types of objects supported by SLES8 and the access control attributes associated with each one of them.

To conclude, the goal of SLES8 is to receive an EAL3 augmented by ALC_FLR. EAL3 requires the TOE to be "methodologically tested and checked" [3]. In practice, it provides a moderate level of security assurance by investigating the ST content from different prospectives. EAL3 analyzes the TOE using functional and interfaces specifications, design documentation, as well as all security related documentation. To increase the level of assurance EAL3 looks also to aspects related to the development environment, configuration management and delivery procedures [4].

6.2 Evaluation process

An important question that we might ask ourselves after looking at SLES8 ST is: how did the certifiers manage to obtain all the required documents to construct the SLES8 ST? If we were looking at some commercial, closed-source software this question would probably not arise, because most software suppliers adopt a structured development process and generate the corresponding documentation concerning the whole life cycle of their products, including requirement, analysis, design, tests and so forth. However, the less structured nature of the open source development process (see Chapter 5) makes this task much more difficult. According to ATSEC's EAL3 security

feature *Criteria Developer Evidence* [1], the EAL3 requires the sponsor and the developer to jointly provide the following evidence:

- Security Target.
- Configuration Management (ACM).
- Life-cycle support Activity (ALC).
- Delivery and Operation (ADO).
- Customer Guidance (ADO, AGD).
- Design Documentation (ADV).
- High-level Design.
- Testing (ATE).
- Vulnerability assessment (AVA).

This evidence is then used by the evaluation body to understand and analyze the security functional specification of the TOE.

6.2.1 Producing the Evidence

A major challenge that was faced at this stage of the Linux SLES8 certification process was producing accurate and updated versions of the required documents for the certification. Being an open source system developed by a large distributed community, SLES8 simply lacked some of the documentation required by CC. Among the missing documents there was an high level design document that describes the security functions implementation. Indeed, most open software products lack this type of high level documentation, and Linux is no exception. Despite the fact there are many documents and books discussing the implementation details of Linux kernel in general, few of them discuss security related aspects in detail. As a result, SLES8 did not have a document describing the high level security implementation. To comply with CC specification, IBM had to develop a new high level design document describing the security functions of SLES8, including the ones defined in the ST [16].

As a part of customer documentation, IBM has developed another important piece of evidence: a *security guide*, which provides all the details to install and configure a version of SLES8 for evaluation. The security guide specifies all installation procedures, packages to remove, network configuration and all the modifications of the configuration files necessary to bring SLES8 to the state for which the CC certification holds.[3] IBM also developed a vul-

---

[3] Of course, IBM did not have to create all evidence mentioned here from scratch for CC EAL3. Since SLES8 had previously been certified with EAL2, most of the documents were taken from it and then adapted to fit the EAL3 requirements. The documents that were reused from EAL2 included the `man` pages, part of the test cases, and the ST previously used for SLES8 EAL2.

nerability analysis for SLES8 to discover any residual exploits that were not discovered by SuSE developers' tests.

When SLES8 was certified at level EAL3, many documents were substantially improved upon the initial ones. This improvement included fixing some minor omissions or inaccuracies that were not identified earlier. The later evaluation reports also contained a "functional specification mapping" spreadsheet which explicitly listed the associations between the TSF/TSFI and the corresponding test cases.

As we shall see, IBM did a great effort to enrich the test cases suite that was used to test the security functions of SLES8. This effort includes the preparation of a spreadsheet containing the mapping between the interfaces to be tested and the actual TSF. The construction of this spreadsheet was performed manually, since some test cases (mostly belonging to the *Linux Test Project* (LTP), see below) had been written long before the evaluation and did not provide enough information for easy automated mapping.

However, although originally the test cases and test scripts of open source products were created by development communities independently of the certification process, the LTP evolution is leading to defining the CC certification test plans entirely in terms LTP test cases [13]. This reliance on an externally defined test base is an important trend to be taken into account for the certification of Linux-based software systems.

In the next sections, after briefly discussing the evolution and structure of LTP, we will discuss how to compute the mapping between the security functions and the LTP test cases.

6.3 The Linux Test Project

For many years, testing the Linux kernel was done informally; Linux users and developers would simply run the latest version of the Linux kernel on their desktop and server computers, and the kernel was tested executing real applications. Performance or correctness regression were observed and reported on by the user community. Also, some Linux developers autonomously performed unit testing, but there was little community-wide system and integration testing (see Chapter 3).

The Linux Test Project [11] was originally started by Silicon Graphics to bring organized testing to Linux. LTP aims to provide a complete test suite allowing to validate the stability and robustness of the Linux kernel. It has represented a real "revolution" in Linux testing and assurance, since no formal testing environment was previously available to Linux developers. The major advantage provided by the development of LTP was not the definition of batteries of test cases in itself, but the provisioning of a framework for systematic integration of testing activities. Thanks to recent, massive maintenance and cleanup [13], LTP has been developed to improve the Linux

kernel by providing automated testing of kernel functionalities [6]. Furthermore, LTP has included an environment for defining new tests, integrating existing benchmarks and analyzing test results. Nowadays, the latest version of LTP test suite contains more than three thousands tests.[4] In terms of the test types introduced in Chapter 3, LTP currently deals with functional, system and stress testing. In addition to the test suites, LTP also provides test results, a test tools matrix, technical papers and HowTos on Linux testing, and a code coverage analysis tool [11].

LTP is now an integral part of the daily activity of several developers involved with Linux kernel testing. It contains system tests for various subsystems of the kernel, such as, the memory management code, the scheduler, system calls, file systems, real time features, networking, resource management, containers, IPC, security, timer subsystem and much more. Several users of LTP use it to validate their entire Linux system.

The increasingly strict link between certification and the LTP test base is documented by the addition to LTP of a sample SGI Common Criteria EAL4 certification test suite for the distribution RHEL5.1 o systems SGI Altix 4700 (ia64) and Altix XE (x86_64) systems. Also, LTP output logs are now provided in HTML format. Figure 6.1 depicts the HTML output format for LTP with clear distinction between `FAILED`, `PASSED`, `WARNED` and `BROKEN` test cases.

LTP support for test automation was improved by removing bugs that prevented concurrent execution of tests and by adding *discrete sequential run capability* [13], which allows test to execute as many loops as specified by the tester, irrespective of the time consumed.

Much work has also been done to increase LTP code coverage (see Chapter 3). Results are encouraging [13], although the average value of statement coverage for LTP subsystems tests is still around 50%.

6.3.1 Writing a LTP test case

Different ST will require different tests, and there is no guarantee that the desired test will be available as part of the current LTP packages. For this reason, following [9, 13, 15], we shall now examine the construction of a test case to be added to LTP.

Since its conception, a major goal of LTP was being easy to use in order to encourage Linux developers to integrate it into their development process. To this end, LTP provides developers with basic templates for developing new test cases to keep a uniform coding standards throughout the LTP. As the predominant programming language used to develop LTP test cases is ANSI-

---

[4] When version 2.3 of the Linux kernel was released, LTP included only 100 tests [6].

LTP Output/Log (Report Generated on Mon Aug 18 14:14:56 CEST 2008)

PASSED FAILED WARNING BROKEN RETIRED CONFIG-ERROR

Click Here for Detailed Report
Click Here for Summary Report

Detailed Report

No	Test-Name	Command-Line	Test-Output	Termination-id
1	abort01	"ulimit -c 1024;abort01"	abort01 1 PASS : Test passed	0
2	accept01	"accept01 "	accept01 1 PASS : bad file descriptor successful accept01 2 PASS : bad file descriptor successful accept01 3 PASS : invalid socket buffer successful accept01 4 PASS : invalid salen successful accept01 5 PASS : invalid salen successful accept01 6 PASS : no queued connections successful accept01 7 PASS : UDP accept successful	0
30	fallocate01	"fallocate01"	fallocate01 0 WARN : System does not support execution of the test	1
161	syslog01	"syslog01"	syslog 1 0 (INFO) : Send messages to socket at a given level syslog 1 1 (INFO) : and test for given for these messages syslog 1 0 (INFO) : syslog testing whether messages are logged into log file Running tests syslog 1 1 FAIL : failed to generate messages syslog 1 0 INFO : restart ./syslog1 testing	1
787	dio04	"diotest-1"	diotest4 1 PASS : Negative Offset diotest4 2 PASS : removed diotest4 3 PASS : Odd count of read and write diotest4 4 PASS : Read beyond the file size diotest4 5 PASS : Invalid file descriptor diotest4 6 PASS : Out of range file descriptor diotest4 7 PASS : Closed file descriptor diotest4 8 PASS : removed diotest4 9 CONF : Direct I/O on /dev/null is not supported diotest4 10 PASS : read, write to a mmaped file diotest4 11 PASS : read, write to an unmapped file diotest4 12 PASS : read from file not open for reading diotest4 13 PASS : write to file not open for writing diotest4 14 PASS : read, write with non-aligned buffer diotest4 15 PASS : read, write buffer in read-only space diotest4 16 PASS : read, write in non-existant space diotest4 17 PASS : read, write for file with O_SYNC diotest4 0 INFO : 15 testblocks completed	0

Summary Report

Test Summary	Can report as score test of LTP
LTP Version	LTP-20080229
Start Time	Mon Aug 18 14:14:56 CEST 2008
End Time	Mon Aug 18 14:55:46 CEST 2008
Log Result	/home/wedro/ltp-full-20080229/results
Output/Failed Result	/home/wedro/ltp-full-20080229/output
Total Tests	860
Total Failures	8
Kernel Version	2.6.22.1-41.fc7
Machine Architecture	i386
Hostname	rfc-1918

Fig. 6.1: The HTML format of LTP Output

C [9], in this example we shall use the ANSI-C basic template to illustrate how test cases are created.

To successfully write and execute a new testcase under LTP, a test case needs to satisfy the criteria listed below [9].

1. A testcase shall be self-contained, so that it can be executed independently.

2. The outcome of a testcase (pass/fail) shall be detected within the testcase itself.

3. The return value of a successful testcase shall be 0 and anything otherwise.

A typical ANSI-C testcase using LTP APIs should look like the following code, with the few modifications needed to fit the intended purpose of the example.[5]

```c
// Standard Include Files
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

// Harness Specific Include Files
#include "test.h"
#include "usctest.h"

#define TRUE  1
#define FALSE 0


// Extern Global Variables
extern int  Tst_count; //counter for tst_xxx routines.
extern char *TESTDIR;  //temporary dir created by tst_tmpdir()

// Global Variables
char *TCID     = "dummy_syscall_test01"; //program identifier
int  TST_TOTAL = 1;    // total number of tests in this file

//****************** cleanup ******************************
extern void cleanup()
{
  // Close all open file descriptors

  In case an unexpected failure occurs report it and exit

  // Remove all temporary directories used by this test

  // kill child processes if any

  // Exit with appropriate return code

  tst_exit();
}

//****************** setup ******************************
void setup()
{
  // Capture signals
```

---

[5] For the sake of conciseness, some comments have been omitted. The complete template can be found on the LTP website (http://ltp.sourceforge.net/).

```
  // Create temporary directories

  // Create temporary files

   In case an unexpected failure occurs
   report it and exit
}

//***************** main ******************************
int main(int argc, char **argv)
{
  // parse options

  LTP test harness provides a function called
  parse_opts() that may be used to parse standard
  options. For a list of standard option that are
  available refer on-line man page at the LTP web-site

  // perform global test setup, call setup() function
  setup();

  // Test Case Body

  In case of unexpected failure, or failure not
  directly related to the entity being tested report
  using tst_brk() or tst_brkm() functions.

  if (something bad happened)
  {
    // cleanup and exist
  }

  // print results and exit test-case
  if(results are favorable)
    tst_resm(TPASS, "test worked as designed\n");
  else
    tst_resm(TFAIL, "test failed to work as designed\n");

  tst_exit()
}
```

The three main functions in this code are the *main*, *setup* and *cleanup* functions. The main function is the entry point of the test case, where the test of some specific functionality is performed and the return value of the test (either success or failure) is returned. The setup function is used to create and initialize all needed resources, such as signals to be captured, as well as temporary files and directories. As the name suggests, the cleanup function releases all the resources that were used by the test, removing all temporary files and directories. Note that the cleanup function can be excluded from some tests, depending on whether there is actually something to delete after

the test is done or not. Some additional global variables are also defined inside
the testcase; they are listed with their respective meanings in Table 6.2.

Variable	Description
extern int Tst_count	It is used to report the number of test cases being executed.
extern char *TESTDIR	In case a directory is created during the test, the name of that directory will be saved in TESTDIR.
char *TCID	It contains the test name. The convention used in other test cases is to have the test name followed by two digits number.
int TST_TOTAL	Specifies the number of test cases included in the program.

Table 6.2: Some of the global variables used in LTP

The following example presents a real test case taken from LTP. The test
case checks the result of the *nextafter(double x, double y)* function, which is
a math function that returns the next representable number of x in direction
y. The test checks the result of *nextafter* against different expected values.
Several comments have been omitted from the example, but the complete test
case can be found at *LTP-folder/testcases/misc/math/nextafter/nextafter.c*.

```c
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <stdlib.h>
#include "test.h"
#include "usctest.h"

#define FAILED 0
#define PASSED 1

char *TCID = "nextafter01";

int local_flag = PASSED;
int block_number;
int errno;
FILE *temp;
int TST_TOTAL = 1;
extern int Tst_count;

void setup();
int blenter();
int blexit();

/*--------------------------------------------------------*/
int main()
{
  double answer;
  double check;  /* tmp variable */
```

```
  setup(); /* temp file is now open */
/*----------------------------------------------------------*/
 blenter();

  answer = nextafter(1.0, 1.1);
  check = (answer + 1.0) / 2;
  if ((check != answer) && ((float)check != 1.0))
  {
    fprintf(temp, "nextafter returned \%e, expected answer or
      1.0\n", answer);
    local_flag = FAILED;
  }

  blexit();
/*----------------------------------------------------------*/
  blenter();

  answer = nextafter(1.0, 0.9);
  if ((check != answer) && (check != 1.0))
  {
    fprintf(temp, "nextafter returned \%e, expected answer or
      1.0\n", answer);
    local_flag = FAILED;
  }

  blexit();
/*----------------------------------------------------------*/
  blenter();

  answer = nextafter(1.0, 1.0);
  if (answer != 1.0)
  {
    fprintf(temp, "nextafter 3 returned \%e, expected 1.0\n",
     answer);
    local_flag = FAILED;
  }

  blexit();
/*----------------------------------------------------------*/

  tst_exit();      /* THIS CALL DOES NOT RETURN - EXITS!!  */
  return(0);
}
/*----------------------------------------------------------*/

/***** ***** LTP Port *****/

/* FUNCTIONS */

void setup()
{
  temp = stderr;
}
```

```
int blenter()
{
  local_flag = PASSED;
  return(0);
}

int blexit()
{
  (local_flag == PASSED ) ? tst_resm(TPASS, "Test passed") :
    tst_resm(TFAIL, "Test failed");

  return(0);
}
```

In this example the *nextafter* function is tested with different test values. By comparing this example with the template we discussed above, we can see that there is no cleanup function. The reason is that there is nothing to delete (e.g., no files or directories have been created). After every call of the function the test checks the value of the answer, which shows if the test was passed successfully or not.

6.4 Evaluation Tests

One of the main responsibilities of a CC accredited evaluation body is to ensure that the security functions claimed by the ST are implemented and are correctly enforced. To this end, evaluation bodies must run series of tests to verify each security function of the TOE. The tests used during the evaluation process include the functional test cases provided by the developers, as well as additional tests developed by the evaluation body as needed.

As we have seen, the SLES8 certification relies on existing LTP tests to test most of the system's security function. LTP includes three scripts for executing three different suites of automatic tests [9]: *i)* `runalltests.sh`, *ii)* `network.sh`, and *iii)* `diskio.sh`. To give a clear understanding of a LTP test suite evaluation, in Table 6.3 we provide the summary report of the evaluation of Linux kernel 2.6 on a 64 bit Intel Itanium architecture (formerly called IA-64).

6.4.1 Running the LTP test suite

Software virtualization provides a robust framework for running different types of tests in a single or few virtual machines (see Chapter 8)). During our tests we have then used a virtual machine with the following configuration:

Test Summary	Pan reported some tests FAIL
LTP Version	LTP-20071130
Start Time	Tue Dec 4 02:11:29 PST 2007
End Time	Tue Dec 4 03:11:52 PST 2007
Log Result	/root/subrata/ltp/ltp-full-20071130/results
Output/Failed Result	/root/subrata/ltp/ltp-full-20071130/output
Total Tests	849
Total Failures	0
Kernel Version	2.6.16.21-0.8-default
Machine Architecture	ia64
Hostname	elm3b159

Table 6.3: LTP summary report of Linux kernel evaluation

- Host OS: Windows XP SP2.
- VMware Server 1.4 as the Virtual machine console.
- 256 RAM memory.
- 4 GB of Hard drive space.
- CD drive since the test virtual machine is not connected to the Internet.
- One CPU.

During the installation we followed the Security Guide instructions of *Sles_test_suite*, with the following variations.

- The Sles8 Installation CDs have been downloaded from Novell web site.
- The Virtual machine does not have any network device installed.

Since the installation straight from the SLES8 CDs does not meet the required level of security, some packages must be upgraded as described in the security guide. Novell provides an RPM package that contains all the necessary packages, as well as all the scripts to bring up the base installation to the level needed for running in an evaluated configuration.[6]

6.4.2 Test suite mapping

One of the main responsibilities of an evaluation body is to ensure that the security functions claimed by the software supplier have been correctly implemented. To this end, security evaluation bodies perform a series of tests to check each security function of the TOE. To be able to carry out these tests successfully, security evaluation bodies need to plan the tests to use for each security function or, in other words, to establish a mapping between the SFRs and the test cases. In the case of software products developed

---

[6] However, the RPM package is not available for free. An alternative solution would be to do all the job manually, which is what we did in this case study.

with the CC certification in mind, there are usually no problems, since the developers can create a mapping between the SFRs and some test cases "as they go", in the course of the product development life cycle [10]. However, if the software product (an the corresponding test cases) already exist, the certification authority will have to find out which tests can be used for which SFRs. For SLES8 or Linux in general, finding the right mapping even harder because it is very difficult to track the activities of the developers of a large open code base like Linux (see Chapter 5. In many cases, the developers of the system functionalities are not the same people who provide the test cases for the same functionalities. Therefore, finding the right mappings becomes a very time consuming and error-prone task.

As far as security features are concerned, a major problem to be faced in security testing is that no framework for automatic linking between tests and security features is available. The developers are then forced to associate tests to security functions based on their experience only.

In the case of SLES8, this task has been done manually by creating two types of mappings. The first mapping connects each SFR with the security function or set of security functions that implement that SFR. For instance the security function *User Identity Changing (IA.4)* contributes to satisfy the SFR *User-Subject Binding (FIA_USB.1)* [7]. The second mapping instead is established between the security functions and the actual testcases. Figure 6.2 depicts the two mappings. In the case of SLES8 the first mapping is part of the TOE security specification section of the ST, since it is the developer who knows what are the security functions that satisfy each SFR. On the other hand, the second mapping is not part of the ST, because it is the responsibility of the evaluation body to create the mapping [3].
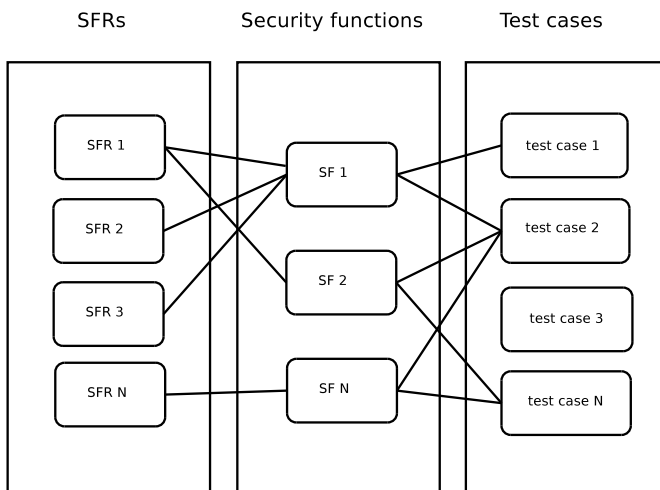
Fig. 6.2: The mapping between SFRs, security functions and testcases

Certification Test Suites   **Security Functions (SFs)**

Matching Engine

SF1   SF2   ...   SFn

Test Suite 1   Test Suite 2   ··· Test Suite n
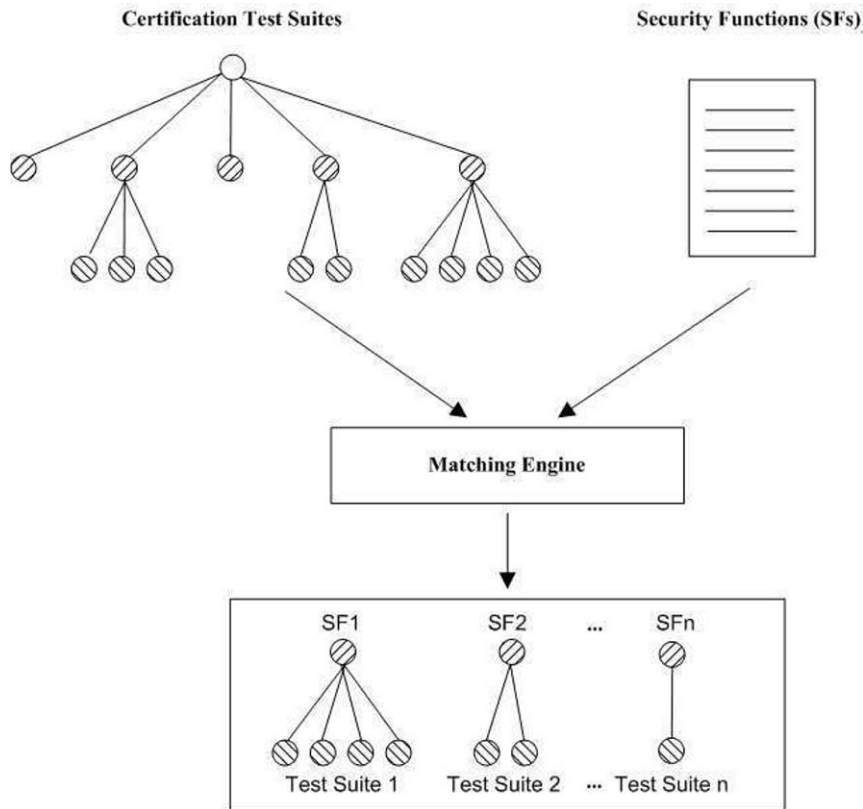
Fig. 6.3: Matching engine

We will now describe a solution consisting of a generic framework for an automatic selection of test suites for the evaluation of the security functions. Our framework, depicted in Figure 6.3, is composed of three main components:

- *Certification Test Suites,* which represents a hierarchical structure containing the set of test suites to be used in the certification process. Each node of the structure is assumed to be labeled using a self-explaining name and enriched of metadata describing its semantics. Each leaf-node represents a single test;
- *Security Functions,* which include the functions description of the TOE to be evaluated;
- *Matching Engine,* which is the component responsible for associating tests to security functions.

The first step toward the development of a matching engine for automatic test selection is the introduction of a security feature vocabulary. To our pur-

poses, a vocabulary can be defined as a set of keywords $\{k_1, \ldots, k_n\}$ to be used in the definition of test suites metadata and name, and for the description of the security functions to be evaluated.

Vocabulary V allows to define both certification test suites and security functions based on a security feature grammar, which simplify the matching engine process. Our matching engine can be defined as a function f that takes in input the vocabulary V, the security functions SF and the test suites TS, and produces in output the association between the security functions SF and the test suites TS. The mapping process is composed of three phases: *Keyword extraction*, *Semantic expansion* and *Search/Match*.

In the first phase, given a security function $sf \in SF$, the matching engine searches inside sf all keywords $\{k_1, \ldots, k_n\} \in V$ extracting a set $K \subseteq V$ of all the keywords used in the security function description. Then, each keyword $k_i \in K$ is expanded by means of a suitable thesaurus. After that, each k_i in the expanded set is searched in the certification test suites paths and in each node metadata. All nodes that contain at least one matching, that is, where at least one keyword k_i is found, are selected and used by the matching engine as the root of a subtree to be considered in the final step. Finally, the matching engine returns all the selected subtrees, which are used to test security function sf.

Although general enough to be adopted in a generic CC certification process, our solution was tested in the context of the certification of the SLES8 Linux distribution. In particular, our framework relies on an extended version of LTP, which has been used for SLES8 EAL3 certification [8], and on the security functions specification used for the definition of the security target of SLES8. Since no formal metadata describing the test suites for SLES8 EAL3 certification are available and no agreement between security functions and test suites vocabulary is in place for this application, the vocabulary used by our matching engine is composed by the set of node names used in the hierarchical structure of the certification test suites. For instance, the certification test suites used in SLES8 EAL3 certification contains the node path network/nsfv4/acl that includes a huge set of tests among which: *create_users*, *test_acl*, and many others. All the node names in the path and the test names are used as keywords for searching inside the security functions description in order to find the association security functions/test suites.

In the following, we present a real example based on SLES8 EAL3 test suite and Security Target. The selected set of security functions belong to *Identification and Authentication (IA)* security functional area. We used our engine to select the set of tests to be used to validate the requirements of each security function.

6.4.3 Automatic Test Selection Example Based on SLES8 Security Functions

The SLES8-based example is summarized in Figure 6.4 and the results are provided in Table 6.4. During the keyword extraction phase, all the security-related vocabulary keywords are extracted by security functions description (see underlined words in second column of Table 6.4). In the second phase, keywords are semantically expanded using an *ad-hoc* security related thesaurus, generating the column tree of Table 6.4. Finally, in the search/match phase, the expanded set of keywords is searched in the LTP testcases directory hierarchy.[7] If a node matches at least one keyword, it will be automatically mapped to the security function to which that keyword belongs (see column four in Table 6.4).
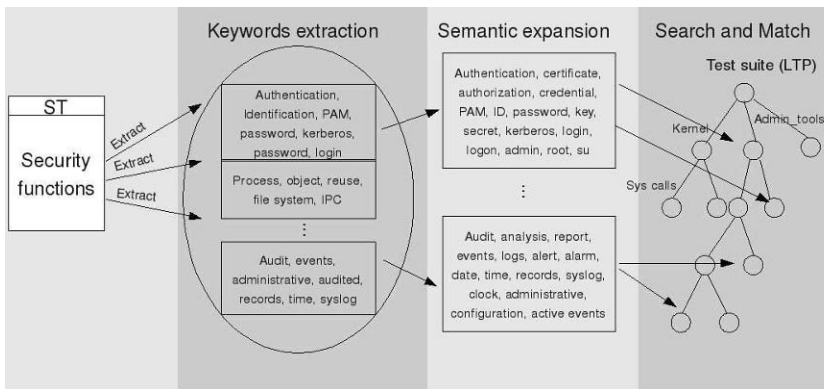
Fig. 6.4: The automated mapping approach between security functions and testcases

In our approach, the search/match phase relies on the Linux command `grep`, which is applied to the extended LTP tree structure. The `grep` command, developed within the GNU project, searches one or more input strings and returns any line containing a match to a specified pattern. Based on grep command, scripts are generated automatically starting from the keywords retrieved in the semantic expansion phase. These scripts search the keywords retrieved during the keyword extraction and semantic expansion phases, inside the directory structure of extended LTP tree and inside the comments in the testcases. Figure 6.5 presents a sample script, which is executed to retrieve the set of tests for the SC.1 security function described in Table 6.4.[8]

---

[7] Search process considers also the comments inside the testcases files.

[8] The complete output produced by the execution of the grep command is summarized in Appendix A.

```
grep -l -i -e 'tunneling' -e 'port 22'
            -e 'secure channel' -e 'secure socket layer'
            -e 'ssl' -e 'ssh'  -e 'secure shell'
            -r linux_security_test_suite_EAL3
```

Fig. 6.5: Example of search/match script applied to the SC.1 security function. The -l and -i options allow to print the list of matching files ignoring case distinctions.

Note that our matching engine has perfect recall: it always returns all the relevant test cases selected manually by developers during SLES8 certification. Of course, our matching is not entirely precise, and some redundant tests are also selected. However, here the point is achieving total recall rather than high precision, even at the price of carrying out some unnecessary tests.

Security Function	Description	Semantic Expansion	Tests
User Identification and Authentication Data management (IA.1)	... user may have different usernames, different user IDs ... users are allowed to change their passwords using the passwd command ... the read the content of /etc/shadow	credential, certificate, authorization, identity, encryption, encoding, identification	group01, pam01, test_login.c, passwd01, shadow01, etc.
security feature authentication mechanism (IA.2)	SLES includes a security feature authentication mechanism...including all the interactive login activities, bash jobs, and authentication for the SU command...password authentication...check password expiration	credential, PAM, username, root, access	pam01, test_login.c, test_sshd.c, test_vsftp.c, passwd01, passwd02, etc.
Interactive login and related mechanisms (IA.3)	The ssh and ftp as the su command used to change the real, filesystem and effective user ID all use the same authentication mechanism in the evaluated configuration... to protect the user's entry of a password correctly...as long as the remote system is also an evaluated version of the TOE	PAM, login, passwd, remote	group01, pam01, test_login.c, passwd01, shadow01, etc.

User identity changing (IA.4)	Users can change their identity...using the <u>su</u> command ...the <u>su</u> command within the SLES is to allow appropriately <u>authorized</u> individuals the ability to assume the <u>root</u> identity...to switch to <u>root</u> has been restricted to users belonging to the trusted group	impersonate, super user, login, authorization	setfsgid01.c, setsgfid02.c, setfsuid01.c, setgid01.c, etc.
Login processing (IA.5)	At the <u>login</u> process the <u>login</u>, real, filesystem, and effective <u>user ID</u> are set to the ID of the user that has logged in.	authentication, credential, identify, pam, password	pam01, test_login.c, test_sshd.c, test_vsftp.c, passwd01, passwd02, etc.

Table 6.4: Automatic test selection results. For sake of conciseness, the table only reports a part of each security function and a partial list of tests. A complete description of the functions can be found in [7].

6.5 Evaluation Results

The results of the SLES8 TOE evaluation are documented in an *evaluation technical report* (ETR) which presents the overall findings and their justifications [3]. It is produced by the evaluation body and then submitted to a validation body to validate it. Most of the time, two versions of the ETR are produced: *i)* a complete version that includes proprietary and confidential information and *ii)* a lightweight version that excludes proprietary and confidential information. The minimum content of an ETR is shown in Figure 6.6. For SLES8, as reported in SLES8 certification report, the ETR is a confidential document, so we cannot reproduce it here. However, all publicly accessible contents are described in the certification report itself.

Another important document produced by the SLES8 evaluation body is the *observation report* (OR) which contains all problems that have been identified during the TOE evaluation [3]. Based on this documentation, the evaluator verifies all the documents produced by the evaluation body and issues the certification report (CR). The CR is a publicly available document; it targets mainly the potential customers of the TOE. It summarizes the TOE security functions, the evaluator testing effort, the evaluated configuration as well as the results of the evaluation, by showing all the assurance components that have passed the certification requirements.

SLES8 has been certified with EAL3+. The evaluation has proven that

- The TOE conforms to the CAPP protection profile.

Evaluation Technical Report

1. Introduction
2. Evaluation
3. Result of the evaluation
4. Conclusions and recommendations
5. List of evaluation evidences
6. List of acronyms/glossary of terms
7. Observation reports

Fig. 6.6: The content of the Evaluation Technical Report

- That the SFRs specified in SLES8 ST conform to CC part 2 extended.
- That the SARs specified in SLES8 ST conform to CC part 3 augmented.
- The security functions satisfy the required level of security.

6.6 Horizontal and Vertical reuse of SLES8 evaluation

6.6.1 Across distribution extension

Let us now pose ourselves a question that the reader may have anticipated: how can we reuse what has been done for SLES8 to certify other distributions? To answer this question, we had to analyze and isolate the portion of SLES8 that has actually been certified, then compare it with other distributions. According to SLES8 certification report [5] SLES8 provides the security functions depicted in Table 6.5.

By taking a closer look at Table 6.5, we found that the security functions that were considered in SLES8 fall in one of three categories: *Kernel syscalls*, *programs/processes with root privileges* and *configurations files for security functions*. Let us now check the level of compatibility between SLES8 and other distributions. As a starting point for testing this we consider the following experiment.

We install another Linux distribution (i.e., *Fedora 7*) and we ran part of EAL3 test suite on top of it. The part of EAL3 considered was LTP testcases, since the other testcases were more specific to SLES. The EAL3 automated tests are divided into six separate testcases packages presented in Table 6.6. For our experiments, we used *Test suite mapping* and *ltp_OpenSSL* testcases, because they are LTP compliant. An interesting point to mention here is that ltp_EAL2 was reused from the previous certification of SLES8 with EAL2, which reduces time and effort of the certification process.

During our testing activities, total of 860 test cases were executed on Fedora 7 running on top of the Xen virtual environment (see Chapter 8 for

Name	Function
Identification and Authentication (IA)	
IA.1	User Identification and Authentication Data Management
IA.2	User security feature Authentication Mechanism
IA.3	Interactive Login and Related Mechanisms
IA.4	User Identity Changing
IA.5	Login Processing
Audit (AU)	
UA.1	Audit Configuration
UA.2	Audit Processing
UA.3	Audit Record Format
UA.4	Audit Post-Processing
Discretionary Access Control (DA)	
DA.1	General DAC Policy
DA.2	Permission Bits
DA.3	Access Control Lists supported by SLES
DA.4	Discretionary Access Control: IPC Objects
Object Reuse (OR)	
OR.1	Object Reuse: File System Objects
OR.2	Object Reuse: IPC Objects
OR.3	Object Reuse: Memory Objects
Security Management (SM)	
SM.1	Roles
SM.2	Access Control Configuration and Management
SM.3	Management of User, Group and Authentication Data
SM.4	Management of Audit Configuration
SM.5	Reliable Time Stamps
Secure Communication (SC)	
SC.1	Secure Protocols
TSF Protection (TP)	
TP.1	TSF Invocation Guarantees
TP.2	Kernel
TP.3	Kernel Modules
TP.4	Trusted Processes
TP.5	TSF Databases
TP.4	Internal TOE Protection Mechanisms
TP.5	Testing the TOE Protection Mechanisms

Table 6.5: SLES8 security functions

Testcases packages
at_test_EAL
ext3_ACLs_tests
laus_test
ltp_EAL2
ltp_OpenSSL

Table 6.6: SLES8 EAL3 Testcases packages

the details on the virtual testing environment we used).[9] The result was that only 14 tests failed, and none of them was security related. This result shows a high degree of compatibility between the different Linux distributions, which can be used as a starting point to extend the range of certified open source products.

6.6.2 SLES8 certification within a composite product

A modular TOE is composed of two or more components which need to be individually compliant to security criteria. There are several scenarios in which splitting a TOE into small TOE components proves to be more efficient than consider a monolithic one. For example in the case of a firmware which needs to be evaluated on different hardware platforms, using a composite TOE will distinguish a firmware component and a hardware component. At every evaluation the hardware component can be changed. The CC *Composition Assurance Class* (CAC) defines activities to determine a secure integration between the different components of the composite TOE. During the composite TOE testing, the evaluators need to test the TOE SFRs at composed level, as well as for the base components [3].

Again, we need to take into account the fact that the Linux kernel and the applications running on top of it are developed by different parties. Many companies and communities contribute different components (see Chapter 5 which are loosely bound to the operating system. Such components use Linux as an execution environment but they do not depend heavily on specific implementations details. Many Linux applications can be installed and executed independently on the kernel version or the used distribution. Figure 6.7 depicts some Linux based environments in which Linux kernel is considered to be an integral part of the product. When certifying such products, the existing SLES certification can help ST authors to reduce the time and effort required to analyze the operating system part. In this scenario, however, the focus will be on the application running on top of the operating system as well as the communication interfaces between the OS and the application. The objective is to integrate an already certified product in the process of a new product certification as depicted in Figure 6.8.

---

[9] For this experiment, we have used the latest LTP package available at the time (i.e, the one released on January 31st 2008) and we have checked manually that the latest LTP includes all the security tests that were used in EAL3. Some of the files such as `fileperm01`, `procperm01`, `object_reuse01` are missing from LTP since they were developed specifically for Suse during the certification process.
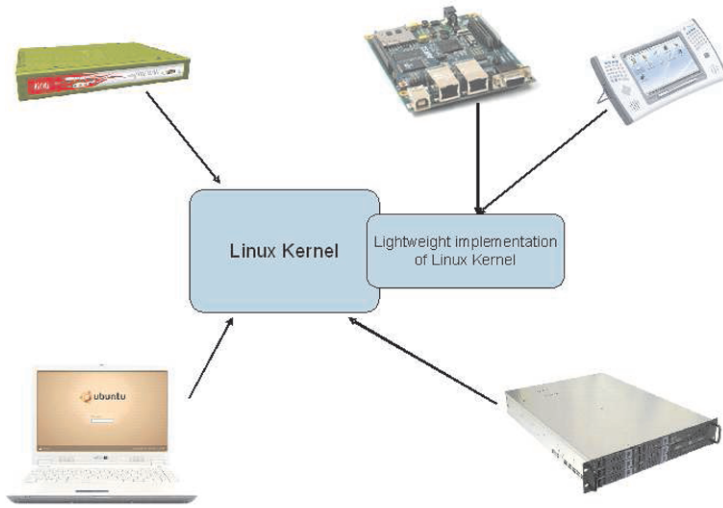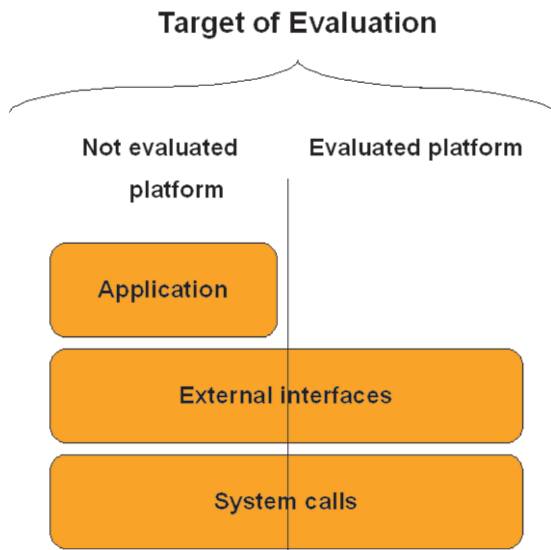
Fig. 6.7: Linux-based environments

Fig. 6.8: Integrate an evaluated platform in the certification of a new TOE

6.7 Conclusions

The aim of the CC standard is to develop and provide a standard security evaluation process to be used by software suppliers to validate the level of confidence for their products. However, many software vendors and security experts do not share this definition. Whereas vendors see CC as an expensive process both in terms of time and effort, security experts consider CC to be nothing more than validating the paperwork related to a software.

Many software suppliers do not consider CC as a methodology for software security assessment; rather they treat it as an additional requirement to be able to sell their products to governmental organizations. An argument made by Jonathan S. Shapiro [17] when Windows 2000 received the EAL4 certificate goes as follows: "The certification applies to an existing product. It was the same product on 29 October that it was on 28 October: no more secure and no less" [17]. Shapiro argued that even if Windows 2000 had been certified with EAL4, this did not change what it was known about it, such as the fact of it being the target of many threats and exploits.

Another issue regards the TOE. It is stated in the CC part 1 that "In general, IT products can be configured in many ways: installed in different ways, with different options enabled or disabled. As, during a CC evaluation, it will be determined whether a TOE meets certain requirements, this flexibility in configuration may lead to problems, as all possible configurations of the TOE must meet the requirements. For these reasons, it is often the case that the guidance part of the TOE strongly constrains the possible configurations of the TOE, that is, "the guidance of the TOE may be different from the general guidance of the IT product" [2]. This statement makes the TOE bound to the configurations that have been used during the certification. This strict connection between security and configuration changes the way we look to the TOE from being a certified product to a certified configuration. Once again, this reminds us that we never certify a product line or even a specific product, but only specific configurations of it. Also, current certifications become obsolete when the context in which they were obtained changes. This poses a research problem which we shall describe in more detail in Chapter 9.

References

1. atsec information security corporation. *EAL3 Common Criteria Developer Evidence.*
2. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model*, 2006.
3. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security*

Evaluation, Evaluation methodology, 2007.

4. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components*, 2007.

5. BSI Bundesamt fr Sicherheit in der Informationstechnik. *Certification report for SUSE Linux Enterprise Server V 8 with Service Pack 3*, 2006.

6. N. Hinds. Kernel korner: The linux test project. *Linux Journal*, December 2004.

7. IBM and SuSE. *SuSE Linux Enterprise Server V8 with Service Pack 3 Security Target for CAPP Compliance*, November 2003.

8. SuSE IBM and atsec. *SuSE Linux Enterprise Server 8 w/SP3 CAPP/EAL3+ Certification Test Suite*.

9. P. Larson. Testing linux with the linux test project. In *Proc. of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, June 2002.

10. C. Lennerholt, B. Lings, and B. Lundell. Architectural issues in opening up the advantages of open source in product development companies. In *Proc. of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008)*, Turku, Finland, July-August 2008.

11. *Linux Test Project.* ltp.sourceforge.net.

12. P.A. Loscocco, S.D. Smalley, P.A. Muckelbauer, R.C. Taylor, S.J. Turner, and J.F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proc. of the 21st National Information Systems Security Conference*, Arlington, Virginia, USA, October 1998.

13. S. Modak and B. Singh. A building a robust linux kernel piggybacking the linux test project. In *Proc. of the 2008 Linux Symposium*, Ottawa, Canada, July 2008.

14. Information Systems Security Organization. *Controlled Access Protection Profile version 1.d*, 1999.

15. Hubertus Franke Rajan Ravindran Paul Larson, Nigel Hinds. Improving the linux test project with kernel code coverage analysis. In *Proc. of the Ottawa Linux Symposium*, Ottawa, Canada, July 2003.

16. K.S. Shankar and H. Kurth. Certifying open source: The linux experience. *IEEE Security & Privacy*, 2(6):28–33, November-December 2004.

17. J.S. Shapiro. Understanding the windows eal4 evaluation. *Computer*, 36(2):103–105, February 2003.

Chapter 7
Case Study 2: ICSA and CCHIT Certifications

Abstract In this chapter we briefly examine some "lightweight" software certification processes, starting with the ICSA security certification. Unlike Common Criteria (CC), the ICSA scheme is not aimed at relating protection requirement to the presence of specific security-related features; rather, it intended to provide a simple yet rigorous security assurance procedure for network and Internet-related software products. The ICSA certification goal is to alleviate certification costs, certifying software products across multiple versions and configurations. We start by describing how ICSA can be applied to an open source software firewall. Then, we outline the notion of domain specific certifications aimed at *Iliability reduction*, such as the CCHIT certification for healthcare-related software products.

7.1 Introduction

A major motivation for an open source software product to be certified is to reduce both real risk (see Chapter 3) and perceived risk. Thanks to certification, software purchasers gain reassurance that the software product they are using meets industry-accepted standards and that the software supplier has taken due care, having addressed all known security issues. Some lightweight test-based security certification can achieve the goal of reassuring the users without the complexities of CC. As an example, we shall examine a "lightweight" certification process, totally different from the CC one: the ICSA security certification.[1] The ICSA certification goal is to support a simple yet rigorous assurance process for security-related software products.

ICSA applies the same set of certification plans to each product type, based on a set of certification criteria yielding a *pass-fail* result. These certification

---

[1] ICSA Labs (`www.icsalabs.com`) is an independent division of Verizon Business. It has certified a large portion of the well known security related software products, including anti-virus, anti-spyware and firewalls.

criteria are based on testing the immunity degree of the systems under evaluation against a set of threats and risks, rather than on the system's design, architecture, or any attempt to assess the underlying technology. Generally speaking, this is what we refer to as black-box test-based approach (Chapter 3) [5].

An important aspect of the ICSA certification is *consistency*. ICSA certification criteria are based on how software products react to a list of threats, rather than on a mapping between protection requirements and specific features. Therefore, ICSA criteria support comparative analysis and product ranking. To keep the list of threats used during the evalution up-to-date, ICSA Labs relies on the data and views collected from different sources including users, software suppliers, security experts and academic researchers. As a result, a draft of the new set of criteria, called *notice of proposed certification criteria*, is created and then circulated for further reviews before making the criteria final and publicly posted [5].

Another important feature of ICSA certificates is *genericity*: "the ICSA certification criteria and process are designed so that once a software product is certified, all future versions of the product (as applicable) are inherently certified" [5]. This is achieved by a three-steps assurance process [5], aimed at making the ICSA certification relatively independent of product updates and version changes. Here is a summary of the ICSA assurance process:

1. ICSA Labs gain a contractual commitment from the software vendor, agreeing that the software product will be consistently maintained at the current ICSA Labs Certification standard. In other words, the vendor's own quality assurance programs must incorporate current ICSA certification criteria into their product development processes. Thus, evaluation bodies play a lesser role in ICSA than in CC: a significant part of the ICSA certification process consists of self-checking by the organization whose product is certified.
2. ICSA Labs reserve the right to perform random assessments of the software products against current criteria for that certification category. If a product fails an assessment, the responsible party is requested to rectify the problem(s). If the product still does not meet certification criteria by the end of a short grace period, the certification is revoked.
3. ICSA certification expires after one year, so that the certification process is repeated yearly.

Since manual testing requires additional time and effort, especially in case of re-certification, the ICSA Labs approach to testing is based on a battery of automated tests aimed at reducing the evaluation process time and increasing the tests quality. When automated testing is not possible, ICSA Labs advises a checklist oriented approach where skilled security analysts - either from ICSA Labs or a third-party lab specialists trained and authorized by ICSA Labs - perform the tests [5].

As the reader may already have noticed, there is absolutely nothing in the ICSA assurance process that prevents it from being applied to open source software. As an example, in the remainder of the chapter we shall discuss the ICSA certification of an open source firewall product, the Endian firewall. However, ICSA is not the only lightweight software certification process available today; several domain specific certifications have been proposed. As an example of a domain-specific certification, in this chapter we shall briefly describe how certifications are awarded by the *Certification Commission for Healthcare Information Technology* (CCHIT). The rationale of CCHIT software certification is the reduction of the software supplier's *liability* (Chapter 1). Healthcare organization welcome all means of reducing their liability, and security certifications are considered by some as an important step in this direction.[2] Today, open source stakeholders seldom see the need for going through the process of achieving domain-specific certifications, this situation may change soon and healthcare related IT products bundling OSS become widespread.

7.2 ICSA Dynamic Certification Framework

Let us now briefly describe the ICSA certification process. The ICSA certification is based on what is called the *ICSA Labs Dynamic Certification Framework*, defining the activities composing the process. ICSA defines a dynamic certification process that starts by analyzing risks, identifying potential threats to the software product under evaluation. The first activity is a complete analysis of the potential failures of the software product for which certification is contemplated, assessing their probability and impact. In other words, a complete risk analysis and prioritization is performed as illustrated in Chapter 3).

This risk analysis activity brings out failures with the least impact and lowest probability, allowing to filter out small risks; for all other risks, this activity identifies a set of *safeguards* aimed at mitigating them. Next, the safeguards are converted into practical ICSA certification criteria, which are submitted to vendor groups, end-user groups, and the software market at large. The ICSA certification will actually assert the presence of such safeguards [5].

When a sufficient number of certifications have been performed, ICSA Labs develop and apply metrics to confirm the actual risk-reduction accomplished through the certification process. These analysis lead to updating the criteria

---

[2] Of course, different legal systems have different notions of liability, and the extent of any liability reduction due to software certification would of course depend on where the organization using certified software is based. Dealing with the legal implications of liability issues is beyond the scope of this book.

and the corresponding set of safeguards, triggering a new iteration of the
certification framework cycle [5].

7.3 A closer look to ICSA certification

We are now ready to take a closer look to ICSA certification process. In the
following, to fix our ideas, we shall focus on the certification of firewalls.

As their name suggest, firewalls generally refer to a set of technologies and
devices designed to stop unauthorized network activities. A firewall typically
behaves as a bottleneck between the internal and the external network, and
it uses a set of defined rules to allow or reject certain types of traffic to pass
through it [3]. Generally, firewall systems operate on the principle packet
filtering, in which the header of each data packet traversing the firewall is
analyzed and compared with a defined set of rules, to decide whether it should
be allowed or rejected.

The ICSA firewall evaluation adopts the same certification criteria for all
firewalls under evaluation. All the firewalls are evaluated against the same
set of functional and assurance criteria. Version 4.1 of the modular firewall
certification criteria contains the following documents.

- *The Baseline module*: it defines the requirements that all firewall products
  must satisfy in order to be certified.
- *4.1a Logging criteria*: an update of the 4.1 baseline Logging criteria
- *Residential, Small/Medium Business, Corporate*: the firewall vendor may
  choose one of these modules depending on the target customer
- *Glossary*: definitions of terms used in 4.1 criteria documents

ICSA has grouped the security criteria that a candidate firewall must satisfy
into six wide categories, depicted in Table 7.1.

To obtain ICSA Labs certification, a software firewall needs to provide
evidence of implementing all the safeguards corresponding to known threats
in the six areas listed above.

7.3.1 Certification process

As stated in Section 7.1, ICSA Labs certification process differs from the
CC one in many respects. Specifically, ICSA certification criteria are based
on functional black-box tests (Chapter 3), verifying if all the safeguards to
threats identified during the risk analysis are in place and work correctly. The
evaluation body uses the test cases outcome to assess the extent to which the
software product can resist to such threats [5].

ICSA Labs certification testing methodology relies on a combination of:

Category	Description
Logging	The candidate Firewall Product must provide the capability to log various types of events
Administration	The candidate Firewall must provide all the necessary configuration tools and administrative functions
Persistence	The candidate Firewall must be persistent after a lost of power
Functional testing	Assures that during the tests only the services in the security policy pass through the candidate Firewall and no other services pass
Security testing	Administrative access testing, Vulnerability testing, and other types of security tests
Documentation	The candidate Firewall needs to have Installation Documentation, Administrative Documentation, Additional documented coverage, Accurate Documentation and Log event dispositions defined

Table 7.1: ICSA security criteria categories

- Automated testing.
- Checklist oriented where no automated.
- Objective testing.
- Ability of tests reproduction.

After the risk identification phase, a set of safeguards to overcome such risks is created. Then, the controls are converted to attainable criteria that are examined by vendor and user groups for correctness, consistency and completeness. Finally, metrics for each criteria are developed [5]. During this process all the activities may need to be adjusted or re-examined, which improves the quality of the criteria.

7.4 A case study: the ICSA certification of the Endian firewall

ICSA Labs have certified different IT security products including anti-Spams, anti-spywares, anti-viruses, firewalls, network intrusion prevention, PC firewalls, Secure Socket Layer - Transport Layer Security (SSL-TLS), web application firewalls and wireless products. Here we will present the actual certification process[3] for a specific open source software product, the Endian firewall. Endian is a software development company based in Bozen/ Bolzano, in South Tyrol, Italy. It is specialized in developing firewalls over the Linux

---

[3] The certification process presented here represents a proof of concept only, which has not been subject to any formal certification process, neither created by an accredited evaluation body.
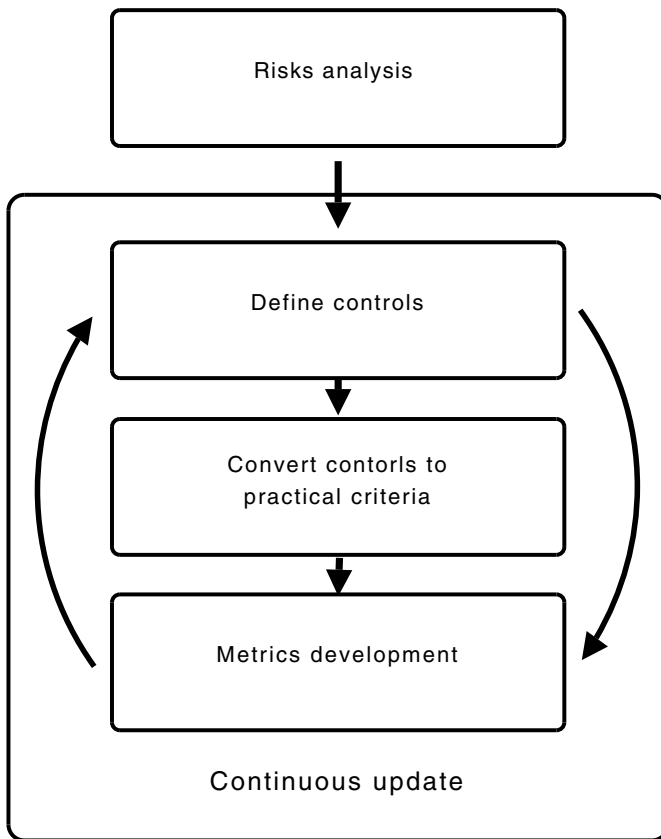
Fig. 7.1: ICSA Labs Dynamic Certification Framework

platform. Endian has released all its software as open source, and has encouraged independent developers to join its community in order to benefit from the community's efforts to test and optimize the firewall's capabilities.

Endian Firewall comes in two versions: *(i)* the open-source community version freely distributed, and *(ii)* a firewall appliance that comes as a Unified Threat Management (UTM) [4] to protect and improve the network's connectivity. Here we shall focus on the former. Endian firewall supports a variety of features showed in Table 7.3.

We will now discuss the detailed test plan for testing the functionality of the Endian firewall safeguards corresponding to known threats in each area of ICSA firewall certification procedure.

Features supported by Endian firewall
Stateful packet inspection
Application-level proxies for various protocols (HTTP, POP3, SMTP, SIP)
Antivirus support
Virus and spamfiltering for email traffic (POP and SMTP)
Content filtering of Web traffic
VPN solution (based on OpenVPN)
Event log management
HTTPS Web Interface
Detailed network traffic graphs

Table 7.2: Some of the features supported by Endian firewall

7.5 Endian Test Plan

7.5.1 Hardware configuration

To be able to test all the aspects of the Endian firewall, the hardware configuration needs to be adapted for each type of tests. For the sake of agility, the tests have been performed in our virtual testing lab (see Chapter 8) using Xen [7] as the virtual machine monitor. Xen allows multiple instances of Linux to run on top of another instance acting as a host operating system.

7.5.2 Software configuration

- Firewall Machine: Only the Endian Community Edition is running.
- Attacker Machine(s): We use Debian distribution as operating system, and some security testing tools described below.
- Victim Machine(s): They use Debian distribution as operating system, and depending on the type of test, other components will be added, for instance web servers, ftp servers, and the like.

7.5.3 Features to test

The following are the features to tests required by ICSA Baseline module.

7.5.3.1 Logging

The specific functionalities we are interested in testing are the ones defined by the ICSA list of safeguards about logging. According to the ICSA base-

line module, the candidate firewall must have "extensive capabilities" of logging events. Also, it must provide detailed information for each logged event. Therefore, tests will be carried out in two steps: the first step to check which events the firewall is actually able to log, and the second to analyze the log's data.

7.5.3.2 Administration

The Administration area includes the actions that an authenticated administrator is allowed to perform. The first aspect to identify here is the Administrative interface provided by the firewall, since the authentication mechanism could be different from the one used to connect for each interface. Only users with the appropriate credentials can authenticate to the Administrative functions. Here, we will test if the Administrative functions provided by the Endian firewall comply with the ICSA Baseline Module.

7.5.3.3 Persistence

We deal with what happens when a power failure occurs, e.g., if the electrical plug of the computer running the firewall is suddenly removed. We would like to make sure that, once the power returns, the firewall security policy will be exactly the same that was being applied when the failure occurred. This property is called *policy persistence*, and corresponds to a software safeguard that guarantees persistence whenever the firewall is rebooted. To test this feature we need to compare: *(i)* security policy before and after the power removal, *(ii)* logs data before and after the power removal, and *(iii)* what happens while the firewall is booting up.

7.5.3.4 Functional and Security testing

This area corresponds to the firewall basic operations, that is, packet filtering according to a security policy. Once a security policy has been defined, the firewall must allow the authorized services to pass and block all the others. Also, the firewall must be able to detect any anomaly in the services operation and block them to prevent and repel attacks.

To test the firewall security the following tests as reported in ICSA baseline module are performed.

- *Administrative Access Testing.* The Candidate Firewall Product must demonstrate through testing that no unauthorized control of its Administrative Functions can be obtained.
- *Vulnerability Testing.* When enforcing a security policy, the Candidate Firewall Product must demonstrate through testing that it is not vulner-

able to the evolving set of vulnerabilities known in the Internet community, which are capable of being remotely tested.

- *No Vulnerabilities Introduced.* When enforcing a security policy, the Candidate Firewall Product must demonstrate through testing that it does not introduce new vulnerabilities to private and service network servers.
- *No Other Traffic.* The Candidate Firewall Product must demonstrate through testing that nothing other than that specified in the security policy traverses the Candidate Firewall Product.
- *Denial of Service.* The Candidate Firewall Product must demonstrate through testing that:

  – its operation is not disrupted or disabled by any trivial denial of service type attacks; and
  – it shuts down gracefully if its operation is disrupted or disabled by any denial of service type attack for which there is no known defense.

- *Fragmented Packets.* The Candidate Firewall Product must demonstrate through testing that fragmented packets can be denied from traversing the Candidate Firewall Product.

7.5.4 Testing tools

We are now ready to describe, with the help of some examples, how the ICSA testing actually takes place. Let us start by listing the test generation tool and the test drivers used in the certification process.

- *Linux Test Project* (LTP) (see Chapter 6).
- *Nessus* (`http://www.nessus.org/`). One of the widest adopted open source security scanners [1]. Its extensible plug-in model allows the developers to work independently and concentrate their effort on different types of vulnerabilities. During our tests we enabled different types of plugins which are suitable for testing product, such as, Denial of service plugins, Firewalls plugins and port scanners plugins.
- *Nmap* (`http://nmap.org/`). An open source utility for network exploration or security auditing. The power of Nmap lies in its flexibility in creating and manupulating IP packets. Using Nmap, we performed many types of scans including packet fragmentation, IP address spoofing, TCP scans with customized flags.
- *Hping3* (`http://www.hping.org/`). A command-line oriented TCP/IP packet assembler/analyzer. It allows the creation of TCP, UDP and ICMP payloads and the manipulation of all their attributes. We have used Hping3 as a port scanner and a DoS tool (see Figure 7.6).

7.6 Testing

7.6.1 Configuration

In our test, we use Endian firewall version EFW − 2.1.2, installed in our virtual environment based on Xen [7] (see Chapter 8 for a discussion on the testing environment). One of the main issues of ICSA certification is deciding which configuration of the firewall should be tested, to carry over the certification to most practical cases. The Endian firewall (like many firewall products) includes many configuration options to accommodate different needs. Describing the testing of all possible configurations is of course not a viable option, because the number of possible configurations is very large. We will therefore discuss testing the default configuration only, which is typical for many scenarios.

Figure 7.2 depicts the network configuration we used to install the Endian firewall:
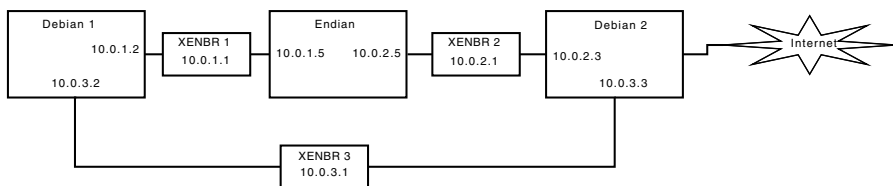
Fig. 7.2: Network configuration for Endian firewall testing

The table below contains the services running on Endian firewall

Services
CRON server
DNS proxy server
E-Mail scanner (POP3)
Kernel logging server
Logging server
NTP Server
Secure shell server
Web server

Table 7.3: Running services in a default installation

7.6.2 Logging

ICSA certification requires the Candidate Firewall to have extensive capabilities of events logging. The Endian firewall provides several different types of logs concerning different functionalities. Table 7.6.2 lists some of the those logs.

Log	Description
Proxy logs	Logs all the files cached by the web proxy server of Endian firewall
Firewall logs	Logs all the traffic that passes through the firewall. Table 7.6.2 shows an example of firewall log data
IDS logs	Logs all the anomalies and incidents discovered by the IDS
Content Filter logs	Logs the pages that have been blocked by the HTTP content filter
System logs	Logs other system activities

Table 7.4: Examples of Endian firewall logging capabilities

All logs share the same administration interface, through which the firewall administrator specifies the number of lines to display, remote logging, the details to be logged, and so forth. Below, we provide two examples of the the required logs.

Example: Each startup of the system itself or of the security policy enforcement component(s)

endian10_0_1_5.localdomain - 22:31:27 up 74 days, 5:07, 0 users, load average: 0.10, 0.09, 0.02

Example: All dropped or denied access requests from private, service and public network clients to traverse the CFP that violate the security policy. The test was carried by starting a port scanner from another host (10.0.2.3) to check if there are any open ports on the firewall host (10.0.2.5). Table 7.6.2 shows an example of the logged data.

Time	Chain	Proto	Source	Src Port	Destination	Dst Port
Mar 11 22:16:04	PORTSCAN	TCP	10.0.2.3	51548	10.0.2.5	80
Mar 11 22:16:04	PORTSCAN	TCP	10.0.2.3	51548	10.0.2.5	443
Mar 11 22:16:05	PORTSCAN	TCP	10.0.2.3	51549	10.0.2.5	443

Table 7.5: Part of Endian Firewall's logs for denied access requests

7.6.3 Administration

Endian firewall can be administered using the web browser based interface provided by Endian and accessible through HTTPS on port 4443. The firewall can also be accessed using SSH. The sample tests we used have failed to bypass the authentication mechanism.
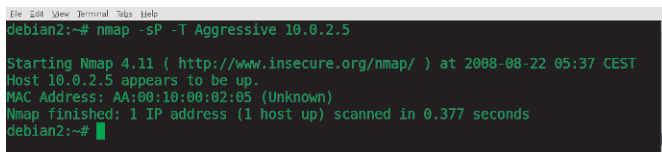
As an example of the administrative requirements, the *Administrative Interface Authentication* to access the Administrative Functions states that, the candidate firewall product must have the capability to require authentication through an administrative interface using an authentication mechanism. The Endian firewall meets this requirement by providing a simple username/password based authentication mechanism.

7.6.4 Security testing

The first step in security testing is to check whether the Endian firewall is able to enforce a default security policy. We started by testing whether the Endian firewall can stand well known security exploits such as the ping of death, flooding attacks and other *Denial Of Service* (DOS) attacks [2]. Examples of the tests we conducted are shown in the next subsections.

7.6.4.1 Using Nmap and Hping3

The test has been carried out as follow. First, the `nmap` tool has been used to ping the firewall machine and scan for open ports. Figures 7.3 and 7.4 depict the two operations. Once the open ports had been detected, we used `Nmap` to check the services running on those ports (see Figure 7.5).

Fig. 7.3: Nmap ping

As shown on Figure 7.5, the port 4333, used by Apache Web server, was found open by this test. As a consequence, as the next step of the test, we launch a *SYN flooding* DOS attack on that port. Using `Hping3`, one can create and send TCP messages to the Apache Web server using the following

Fig. 7.4: nmap scan the open ports

Fig. 7.5: Nmap checking the running services

command:

$$hping3 - S - i\ u10 - p\ 4443 - a\ 10.0.1.2\ 10.0.2.5$$

The first argument *-S* sets the SYN flag; the second argument *-i* specifies the time interval between the messages sent to the open port (in micro seconds); *-p* specifies the destination port; and *-a* is the source IP address which could be also spoofed. After launching the attack, no output was returned for several minutes (see Figure 7.6). The test result suggested that the Endian firewall had detected and stopped the attack. Supporting this intuition, the Endian firewall's logs show that the requests generated during the attack were all blocked.

7.6.4.2 Using Nessus

These tests were conducted using the Nessus tool version 3.0 [6]. The Nessus configuration was as follows:

Fig. 7.6: Hping3 SYN flooding attack

- Port scanners used

  Nessus SNMP scanner
  Nessus TCP scanner
  Netstat scanner
  Ping to remote host
  SYN scan

- Some of Nessus plugins used

  backdoors
  CGI abuses
  CGI abuses: XSS
  Debian local security checks
  Default Unix accounts
  firewalls
  Gain a shell remotely
  gain root remotely
  general
  Misc
  Web servers
  Settings
  Service detection
  Remote file access

Some of the tests results are reported below in Table 7.6. The overall test results show that the Endian firewall meets the criteria of ICSA certification.

7.7 The CCHIT certification

Lightweight certifications have also been gaining acceptance as far as *safety critical* software is concerned. In particular, healthcare organizations welcome all means of reducing their liability, and security certifications are considered by some as a step in this direction. Open source stakeholders seldom see the need for going through the process of achieving domain-specific certifications.

Port ssh (22/tcp)	
Synopsis	An SSH server is listening on this port.
Description	It is possible to obtain information about the remote SSH server by sending an empty authentication request.
Risk factor	None
Port general/tcp	
Synopsis	The physical network is set up in a potentially insecure way.
Description	The remote host is on a different logical network. However, it is on the same physical subnet. An attacker connecting from the same network as your Nessus scanner is on could reconfigure his system to force it to belong to the subnet of the remote host. This makes any filtering between the two subnets useless.
Risk factor	Low
Solution	Use VLANs to separate different logical networks.
Usable remote name server	
Synopsis	The remote name server allows recursive queries to be performed by the host running *nessusd*.
Description	It is possible to query the remote name server for third party names. If this is your internal nameserver, then forget this warning. If you are probing a remote nameserver, then it allows anyone to use it to resolve third parties names (e.g., `www.nessus.org`). This allows hackers to do cache poisoning attacks against this nameserver. If the host allows these recursive queries via UDP, then the host can be used to 'bounce' Denial of Service attacks against another network or system.
Risk factor	Medium
Solution	Restrict recursive queries to the hosts that should use this nameserver (e.g., those of the LAN connected to it). If you are using bind 8, you can do this by using the instruction 'allow-recursion' in the 'options' section of your named.conf. If you are using bind 9, you can define a grouping of internal addresses using the 'acl' command.

Table 7.6: Part of the output generated by Nessus during the Endian firewall testing

This situation may change soon as healthcare related IT products bundling OSS become widespread, and interested stakeholders may emerge in reducing their liability as suppliers or users of healthcare related products bundling OSS. Also, domain-specific certification processes are lightweight and require only a fraction of the effort of complex certifications like CC.

As an example of a domain-specific certification, we shall briefly describe how certifications are awarded by the *Certification Commission for Healthcare Information Technology* (CCHIT). CCHIT has been officially recognized by the US federal government as a certification body for *Health Information Technology* (HIT) products. The Commission certification criteria represent basic requirements that software products must satisfy to deliver acceptable levels of functionality, interoperability and security. CCHIT certificates are released in three separate subdomains: *Ambulatory*, *Inpatient*, and *Emergency Department*. CCHIT certification goals include the following ones.

- *Risk reduction.* Reduce risk of physician and provider investment in healthcare related IT.
- *Interoperability.* Facilitate interoperability between EHRs, health information exchanges, and other entities.
- *Incentives.* Enhance availability of incentives to the adoption of certified products.
- *Privacy.* Protect the privacy of personal health information.

Here, we will briefly discuss some CCHIT certification requirements, following the Handbook published by the Commission.[4]

7.7.1 The CCHIT certification process

The CCHIT certification process includes three phases: a *demonstration* phase, a *document inspection* phase and a *technical testing* phase. In the latter, test scripts are used to simulate realistic clinical use scenarios, with each step in the script mapped against one or more of the certification criteria. The CCHIT inspection process employs a combination of several methodologies, including *Documentation review, Jury-observed demonstrations* and *Technical testing.* All CCHIT evaluation processes are accomplished virtually using a combination of online forms, electronic mail, telephone and Web-conferencing tools to allow jury observation of demonstrations, and online and downloadable technical testing tools.

CCHIT certification requires 100% compliance with all applicable criteria. If an applicant is able to demonstrate that the product meets 100% of the criteria, the evaluation body will take the final step of verifying that the software product is in use at least one production site. When this final step of the evaluation process is complete, the software product becomes officially CCHIT Certified for the specific certification domain and version (e.g., CCHIT Certified Ambulatory EHR 08). Like ICSA, CCHIT is trying to ensure portability of the certification across software versions. Once software products are certified, software suppliers are required to notify CCHIT when new versions are released.

7.8 Conclusions

In this chapter we have discussed some "lightweight" approaches to security certification, such as ICSA. Also, we outlined the notion of domain specific

---

[4] The interested reader can find the complete handbook, together with the Certification Criteria, Test Scripts, and CCHIT Interoperability Test Guide documents at `http://www.cchit.org/files/certification/08/Forms/` `CCHITCertified08Handbook.pdf`.

certifications aimed at liability reduction, such as the CCHIT certification for healthcare-related software products. Unlike CC, the ICSA Labs certifications are based on a single set of criteria (corresponding to known threats) that all products of the same type should satisfy. During the certification process, the software product under evaluation is tested against each of these criteria, in order to verify whether it includes a suitable safeguard. While it is much narrower in scope than CC, being specifically aimed at security-related software products, the ICSA Labs certification has the major advantage of being clear and unambiguous, supporting ranking and comparison of security products.

References

1. Jay Beale, Renaud Deraison, Haroon Meer, Roelof Temmingh, and Charl Van Der Walt. *Nessus Network Auditing*. Syngress Publishing, 2004.
2. W. Cheswick and S. Bellovin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison Wesley, 1994.
3. B. Dempster and J. Eaton-Lee. *IPCOP Firewalls*. PACKT Publishing, 2006.
4. C.J. Kolodgy. The rise of the unified threat management security appliance. In *Worldwide Threat Management Security Appliances 2004-2008 Forecast and 2003 Vendor Shares*.
5. ICSA Labs. *ICSA Labs Product Certification Goals and Objectives*.
6. *Nessus 3.0 Advanced User Guide*. www.nessus.org/documentation/.
7. *Xen users' manual v3.0*. bits.xensource.com/Xen/docs/user.pdf.

Chapter 8
The role of virtual testing labs

Abstract Security certification involves expensive testing challenges that require innovative solutions. In this chapter we discuss how to address this problem by using a virtual testing environment early on and throughout the testing process. Recent technological advances in open source virtual environments in fact satisfy the demands of test-based software certification, since virtual testing environments can run the actual binary that ships in the final product. Also, a virtual test laboratory can simulate not only the system being tested but also the other systems it interacts with.

8.1 Introduction

The term *virtualization* can be used for any software technology that hides the physical characteristics of computing resources from the software executed on them, be it an application or an operating system. A *virtual execution environment* can run software programs written for different physical environments, giving to each program the illusion of being executed on the platform it was originally written for.

The potentiality of such an environment for carrying out testing and test-based certification come immediately to mind. Thanks to virtualization, it looks possible to set up multiple test environments on the same physical machine, saving both time and money. However, how easy is it to set up a virtual testing environment to support the security certification process? More importantly, can we trust test results obtained on the virtual platform to be equivalent to those obtained on a native one?

To answer these questions, we need to take a look at the basic principles on which the idea of virtualization is based. Let us consider the typical dual-state organization of a *computer processing unit* (CPU). A CPU can operate

either in privileged *kernel* or non-privileged (*user*) mode.[1] In kernel mode, all instructions belonging to the CPU instruction set are available to software, whereas in user mode, I/O and other privileged instructions are not available (i.e., they would generate an exception if attempted). User programs can execute the user mode hardware instructions or make *system calls* to the OS kernel in order to request privileged functions (e.g., I/O) performed on their behalf by kernel code.

It is clear that in this dual-state any software that requires direct access to I/O instructions cannot be run alongside the kernel. So how can we execute one operating system kernel on top of another? The answer is by executing on the first (host) kernel a simulated computer environment, called *virtual machine*, where the second (guest) kernel can be run just as if it was installed on a stand-alone hardware platform. To allow access to specific peripheral devices, the simulation must support the guest's interfaces to those devices (see Figure 8.1).

Although the beginning of virtualization dates back to the 1960s,[2] when it was employed to allow application environments to share the same underlying mainframe hardware, it was only in recent years that it reached out to the public market as a way to decouple physical computing facilities from the execution environment expected by applications [5]. Today, many virtual machines are simulated on a single physical machine and their number is limited only by the hosts hardware resources. Also, there is no requirement for a guest OS to be the same as the host one. Furthermore, since the operating systems are running on top of virtual, rather than physical, hardware, we can easily change the physical hardware without affecting the operating systems' drivers or function [17].

There are several approaches to platform virtualization: Hardware Emulation/Simulation, Native/Full Virtulization, Paravirtualization, and Operating System (OS) Level Virtualization (container/jail system).

- *Hardware Emulation/Simulation.* In this method, one or more VMs are created on a host system. Each virtual machine emulates some real or fictional hardware, which in turn requires real resources from the host machine. In principle, the emulator can run one or more arbitrary *guest operating system* without modifications, since the underlying hardware is completely simulated; however, kernel-mode CPU instructions executed by the guest OS will need to be trapped by a *virtual machine monitor* (VMM) to avoid interference with other guests. Specifically, the virtualization safe instructions are executed directly in the processor, while the unsafe ones (typically privileged instructions) get intercepted and

---

[1] Actually, some CPUs have as many as four or even six states. Most operating systems, however, would only use two, so we will not deal with multiple levels of privileges here.

[2] In the mid 1960s, the IBM Watson Research Center started the M44/44X Project, whose architecture was based on virtual machines. The main machine was an IBM 7044 (M44) and each virtual machine was an image of the main machine (44X).

trapped by the VMM.[3] The VMM can be run directly on the real hardware, without requiring a host operating system, or it can be hosted, that is, run as an application on top of a host operating system. Full emulation has a substantial computational overhead and can be very slow. Emulation's main advantage is the ability to simulate hardware which is not yet available.

- *Full Virtualization.* This approach creates a virtual execution environment for running unmodified operating system images, fully replicating the original guest operating system behavior and facilities on the host system. The paravirtualization approach is used by the most currently well-established virtualization platforms, such as *VMWare* [16].

- *Paravirtualization* . The full virtualization approach outlined above uses the virtual machine to mediate between the guest operating systems and the native hardware. Since (guest) VMs run in unprivileged mode, mode-sensitive instructions that require a privileged mode do not work properly, while other kernel-mode instructions need to be trapped by the VM, slowing down execution. The Paravirtualization approach tackles this problem using a simplified VMM called *hypervisor*. Paravirtualization relies on dynamic modification of the guest OS code to avoid unnecessary use of kernel-mode instructions. It enables running different OSs in a single host environment, but requires them to be patched to know they are running under the hypervisor rather than on real hardware. In other words, the host environment presents a software interface with dedicated APIs that can be used by a modified OS. As a consequence, the virtualization-unsafe privileged instructions can be identified and trapped by the hypervisor, and translated into virtualization-safe directly from the guest modified OS. Paravirtualization offers performance close to the one of an unvirtualized system, and, like full virtualization, can support multiple different OSs concurrently. The paravirtualization approach is used by some open source virtualization platforms, such as, Xen [18].

- *OS Level Virtualization* The notion of operating system-level virtualization was originally introduced with the Mach operating system [14]. OS Level Virtualization supports a single OS. Different copies of the same operating systems are executed as user-mode servers isolated from one another. Applications running in a given guest environment view it as a stand-alone system. When a guest program is executed on a server tries to make a system call, the guest OS in the server maps it out to the host system. Both the servers and the host must therefore run the same OS kernel, but different Linux distributions on the different servers are allowed.

All the above virtualization approaches have become widespread thanks to the variety of applications areas in which virtual environments can be

---

[3] In a variation of this approach, unsafe privileged instructions are executed directly via hardware, achieving a better performance level.

deployed. The scenarios in which virtual machines can be used are many, and testing toward multiple platform is clearly an important one [8], since virtualization can be used to combine on the same server different operating systems.
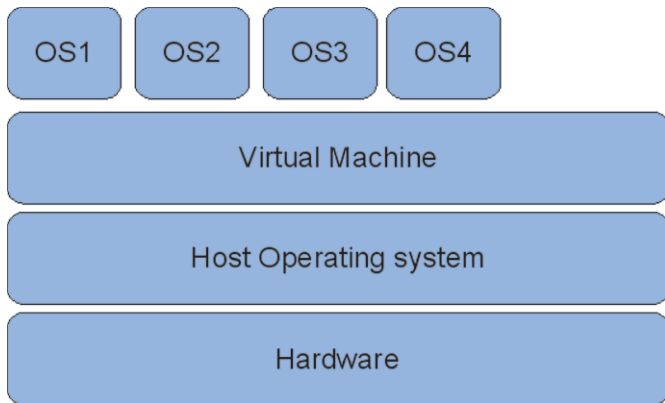

Fig. 8.1: General model of virtualization infrastructure.

In this chapter we provide a brief, informal overview of virtualization internals, aimed at understanding the role of virtual environments in software testing and certification.

8.2 An Overview of Virtualization Internals

To understand how a virtual platform can be used for security testing and certification, let us start by providing a very simplified yet hopefully accurate description of how virtualization actually works. For a more detailed discussion, the interested reader can consult [11].

As mentioned above, most CPU architectures have two levels of privilege: kernel and user-mode instructions. One might envision a VMM as a user mode program which receives in input the binaries of the guest software it is supposed to run. This way, the guest software's user mode instructions can be executed directly on the physical CPU (without involving the VMM), while the kernel mode instructions will cause a trap intercepted and simulated by the VMM in software. In principle, the VMM could completely reproduce the behavior of a real CPU (in this case, the guest software would feature a complete operating system, including a whole set of device drivers). In prac-

tice, the VMM usually maps some of the I/O functions to the host operating system [4].

Problems arise from the fact that many CPU instruction sets include *mode sensitive* instructions that belong to both user and kernel-mode instruction sets. The behavior of mode sensitive instructions depends on the processor current execution mode, and they cannot be trapped like kernel mode ones.[5] For this reason, the original Intel x86 hardware is not straightforwardly virtualizable. Many techniques have been proposed to address the virtualization-unfriendliness due to mode-sensitive instructions. A basic technique to handle double mode instructions is scanning code dynamically and inserting before each mode-sensitive instruction an illegal instruction, which causes a shift to kernel mode and triggers a trap. A related issue is the one of *system calls*. When a process belonging to a guest system running on a VMM enters kernel mode in order to invoke a system call, the shift to kernel mode is trapped by the VMM, which in turn should invoke the guest (and not the host) operating system. A solution is the VMM to use a host system call like `ptrace`() to identify system call invocation on the part of the guest software. When trapping the corresponding shift to kernel mode, the VMM will not execute the call on the host system; rathe,r it will notify the guest system kernel (e.g., by sending a signal), in order to trigger the appropriate action.

8.2.1 Virtualization Environments

Recent CPUs are capable of running all instructions in an unprivileged hardware subsystem, and virtualization software can take advantage of this feature (often called *hardware assisted* virtualization) to eliminate the need for execution-time code scanning. Figuring out efficient virtualization mechanisms, particularly when the underlying hardware is not virtualization-friendly, is still an active area of research; however, the above description should have clarified the basic trapping mechanism through which a virtual machine can operate as a user program under a host kernel, and still look like a "real" machine to its guest software. Software virtualization platforms can set up multiple virtual machines, each of which can be identical to the underlying computer. This section provides a list of some existing virtualization platforms relevant to our purposes.

- *User-Mode Linux.* User-Mode Linux, or simply UML, is a port of the Linux kernel to become a user mode program. In other words, UML is the Linux kernel ported to run on itself. UML runs as a set of Linux

---

[4] Alternatively, some I/O devices can be stubbed by means of NULL drivers, i.e. device drivers that do nothing.

[5] In the Intel 32 bit architecture, mode sensitive instructions like `STR` can be executed both in user and kernel-mode level, but retrieve different values.

user processes, which run normally until they trap to the kernel. UML originally ran in what is now referred to as the `tt` (trace thread) mode, where a special trace thread uses the `ptrace` call to check when UML threads try and execute a system call. Then the trace thread converts the original call to an effectless one (e.g., `getpid()`), and notifies the UML user-mode kernel to execute the original system call.

- *VMware.* VMware Workstation [16] was introduced in 1999, while the GSX Server and ESX Server products were announced in 2001. VMware Workstation (as well as the GSX Server) runs on top of a host operating system (such as Windows or Linux). It acts as both a VMM (talking directly to the hardware), and as an application that runs on top of the host operating system. VMWare Workstation's architecture includes three main components: a user-level application (VMApp), a device driver (VMDriver) for the host system, and a virtual machine monitor (VMM). As a program runs, its execution context can switch from native (that is, the host's) to virtual (that is, belonging to a virtual machine). The VM-Driver is responsible for this switching; for instance, an I/O instruction attempted by a guest system is trapped by the VMDriver and forwarded to the VMApp, which executes in the host's context and performs the I/O using the "regular" system calls of the host operating system [13]. VMware includes numerous optimizations that reduce virtualization overhead. One of the key features for using VMWare for software testing is VMWares *non-persistent mode*. In non-persistent mode, any disk actions are forgotten when the machine is halted and the guest OS image returns to its original state. This is a relevant feature in an environment for software testing, because tests need to start in a known state. VMware ESX Server enables a physical computer to look like a pool of secure virtual servers, each with its own operating systems. Unlike VMware workstation, ESX Server does not need a host operating system, as it runs directly on host hardware. This introduces the problem of mode-sensitive instructions we mentioned earlier. When a guest program tries to execute a mode sensitive instruction it is difficult to call in the VMM, because these instructions have a user mode version and do not cause an exception when run in user mode. VMware ESX Server catches mode-sensitive instructions by rewriting portions of the guest kernel's code to insert traps at appropriate places.
- *z/VM.* z/VM [19], a multiple-access operating system that implements IBM virtualization technology, is the successor to IBM's historical VM/ESA operating system. z/VM can support multiple guest operating systems (there may be version, architecture, or other constraints), such as Linux, OS/390, TPF, VSE/ESA, z/OS, and z/VM itself. z/VM includes comprehensive system management API's for managing virtual images. The real machine's resources are managed by the z/VM Control Program (CP), which also provides the multiple virtual machines. A virtual machine can be defined by its architecture (ESA, XA, and XC, that refer to specific

IBM architectures), and its storage configuration (V=R, V=F, and V=V, refers to how the virtual machine's storage is related to the real storage on the host).

- *Xen.* Xen is a virtual environment developed by the University of Cambridge [6, 2, 18] and released under the GNU GPL license. Xen's VMM, called *hypervisor*, embraces the paravirtualization approach, in that it supports x86/32 and x86/64 hardware platforms, but requires the guest operating system kernel to be ported to the x86-xenon architecture [6]. However, when hardware support for virtualization is available, Xen can run unmodified guest kernels, coming closer to the full virtualization approach. We shall discuss Xen in more detail in section 8.3.2.

8.2.2 Comparing technologies

Let us briefly discuss how the different approaches to virtualization suit the needs of software testing and certification. Full emulation can provide an exact replica of hardware for testing, development, or running code written for a different CPU. This technique is of paramount importance for running proprietary operating systems which can not be modified. It is however computationally very expensive. OS Level Virtualization is a technique aimed at supporting production sites (e.g., Web server farms), rather than software testing or development, since the individual kernels that run as servers are not completely independent from each other. Paravirtualization addresses the performance problem of full virtualization, while preserving good isolation between virtual machines. However, the mediation by the hypervisor requires a level of patching or dynamic modification of the guest OS which is best suited to open source platforms like Linux. Paravirtualization is most useful for testing and distributing software, and for stress tests (Chapter 3) trying to crash the software under test without affecting the host computer. Individual users can be satisfied with emulation or full virtualization products, such as, VMware Player, and VMWare Server for Linux and Windows, and with free products for Linux, such as, Qemu [10] and Bochs [3].

Of course, emulation (full virtualization) has practically no alternative when deploying, on a hardware platform, software originally written for a different hardware architecture. Paravirtualization is an interesting alternative to full virtualization for automating software testing activities. Using paravirtualization, a single machine can be used to test applications in multiple configurations and on different OSs. Often, development is done on one platform or distribution, but has to be verified in other environments. Also, it is possible to quickly create all combinations, and assign them to testers. Another benefit is performing tests on multiple platforms in parallel: a failure in one VM does not stop testing in others.

8.3 Virtual Testing Labs

Since they were introduced, virtualization platforms have caputred the interest of software suppliers as a way to reduce testing costs. Testing is one of the most expensive activities that software suppliers must incorporate in their development process. Many of the testing methodologies we described in Chapter 3 are the result of years of collaborative work by academics and researchers. In a typical software development project, around 30 % of the project's effort is used for testing. Instead, in a mission critical project, software testing is known to take between 50 to 80 % of projects effort [4]. Being able to provide the right testing infrastructure in a short time and at a reasonable cost is a major issue in reducing the impact of testing on software projects. Historically, however, the only option for comprehensively testing a software system, required replicating the system execution environment in a test lab.

Today, many suppliers have adopted alternative approaches including full-system simulation early on and throughout the software development process. Paravirtualization provides an efficient and flexible environment for software testing. More tests can be executed in parallel without affecting the outcome of each other, since they are running in different environments. For example, if two operating systems are installed in two virtual environments, running a test on the first one would not affect the second.

In terms of test-based certification, virtualization offers a new prospective for security testing. Before releasing any software product, typically it has to pass all the functional and security tests designed for it. However, in certification related security testing (Chapter 3), testers need to run the tests on different configurations with different input parameters to check all possible sources of vulnerabilities. In such cases, virtualization offers an effective environment to run certification-related security tests. Testers can simulate different hacking scenarios on different virtual machines, or create an entire virtual network to simulate networks attacks such as flooding attacks. Let us now describe how a testing environment can be set up based on Xen open source virtualization platform. Then, we will briefly discuss how CC tests can be conducted in a Xen-based virtual execution environment.

8.3.1 The Open Virtual Testing Lab

We now present our Xen-based *open virtual lab* (OVL) and its usage for carrying out the testing required by CC certification. An OVL is composed by different OVL virtual machines, each one consisting of an image of the Linux operating system and application-level software. OVL administrators can interact with the OVL virtual machines via a user-friendly administration interface (OVL-AI) [1] presented in Section 8.3.5.
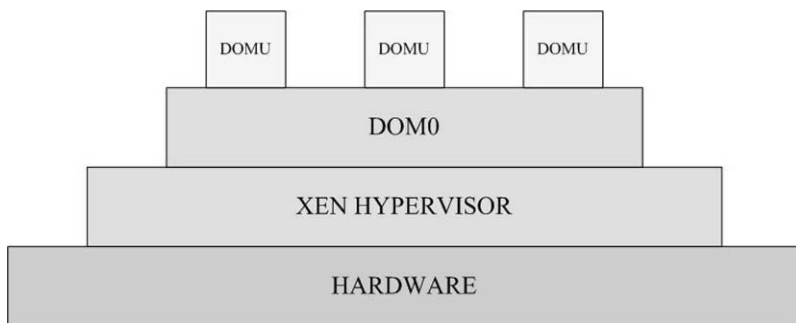
Fig. 8.2: Xen system layers.

8.3.2 Xen Overview

A Xen virtualization system is composed of multiple software layers (see Figure 8.2). Individual virtual execution environments are called *domains*. Xen's hypervisor [6] manages the scheduling operation related to the execution of each domain, while each guest operating system manages the VM application scheduling. During system initialization, a domain with special privileges, called *Domain 0*, is automatically created. Domain 0 can initialize other domains (*DomUs*) and manage their virtual devices. Most management and administration tasks are performed through this special domain.

Xen's current usage scenarios include kernel development, operating system and network configuration testing, server consolidation, and server resources allocation. Several hosting companies have recently adopted Xen to create *public virtual computing facilities*, that is, Web farms capable of flexibly increasing or decreasing their computing capacity. On a public virtual computing facility, customers can commission one, hundreds, or even thousands of server instances simultaneously, enabling Web applications to automatically scale up or down depending on computational needs.

8.3.3 OVL key aspects

The Open Virtual Lab provides each user with a complete Linux-based system image. Also, OVL allows for setting up *virtual internet networks*, by connecting multiple virtual machines, to perform network tests. This feature allows testers to set up their own client-server applications in a virtual network environment. OVL's full support for network programming and middleware is a distinctive feature with respect to commercial virtual laboratories, which focus more on network equipment configuration than on distributed application development.

OVL supports two adoption models: *OVL as a product*, that is, OVL distributed and adopted as a Xen-based open source environment; and *OVL as a service*, showing how OVL can be shared with testers from partner institutions. In both models, costs are mostly related to hosting the environment or purchasing the hardware for running it, since OVL is entirely open source software licensed under the GPL license.

In OVL, each virtual machine is represented by an image of its operating system and application-level software. When configuration changes on a set of virtual machines are needed, OVL administrators can operate via the OVL Administration Interface (OVL-AI). In particular, OVL's design is focused on supporting *scale-up* operations [1]. In a scale-up approach, the system is expanded by adding more devices to an existing node. This action consists in modifying the configuration of every single virtual machine adding, for example, more processors, storage and memory space, or network interfaces, depending on testers needs in a particular situation. Instead, in a scale-out approach, the system is expanded by adding more nodes. In this case, the number of available virtual machines can again be increased (or reduced) easily by OVL-AI. This operation will be beneficial, for example, when new testers join or leave the virtual testing environment.

8.3.4 Hardware and Software Requirements

Intuitively, OVL hardware requirements are essentially two: a storage unit large enough to give a complete software environment to all testers, and enough RAM memory to manage hundreds of virtual machines at the same time. Fortunately, both these requirements can be met remaining within the limits of a tight budget.

The implementation of OVL's virtual machines required some additional considerations.

- *Protection.* Under OVL, each virtual machine has to be an efficient, isolated duplicate of a real machine [9]. In other words, every virtual machine must work in a sealed environment, insulating its disks and memory address space and protecting its system integrity from VM failures.
- *Uniformity.* All virtual machines support a complete and up-to-date operating system to provide the testers with all the instruments needed to carry out administration tasks and test programs. While paravirtualized VMM can, in principle, support a diverse set of guest operating systems, some hardware constraints, in particular the 64-bit server architecture, restrict the range of acceptable guest kernels.

By default, OVLs virtual machines are implemented on the Gentoo *Gentoo* Linux distribution. Gentoo [15] has some distinctive characteristics that fit the requirements of a virtual testing environment. First, a major feature
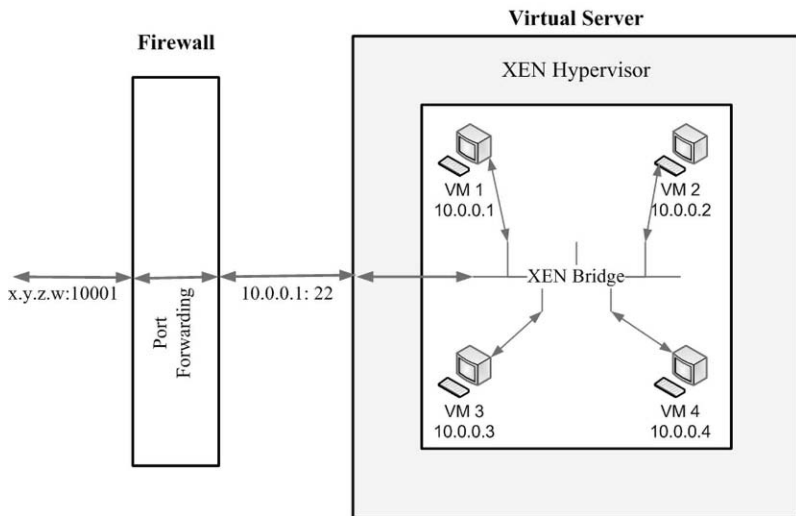
Fig. 8.3: Communications between virtual machines and the external net.

of Gentoo distribution is its high adaptability, because of a technology called *Portage*. Portage performs several key functions: *software distribution*, that permits developers to install and compile only the needed packages that can be added at any time without reinstalling the entire system; *package building and installation*, that allows building a custom version of the package optimized for the underlying hardware; and *automatic updating* of the entire system. Second, Gentoo supports 64 bit hardware architectures and implements the Xen environment in full. Finally, Gentoo is an open source system, distributed under GNU General Public License.

In the current OVL environment, each tester accesses his or her own virtual machine using a secure `ssh` client connected directly to the OVL firewall on a specific port number (computed as `tester_id + 10000`) (see Figure 8.3). Based on the source port, the OVL firewall forwards the connection to the corresponding virtual machine. Figure 8.3 shows how the tester whose `tester_id` is equal to 1 gains access to the firewall. Based on the tester's port number (`10001`), firewall rules forward the incoming connection to the local IP that identifies the tester's own virtual machine. Looking at the example in Figure 8.3, the incoming communication on port `10001` is forwarded to the local IP address `10.0.0.1` on port `22`, and then to the virtual machine `1`.
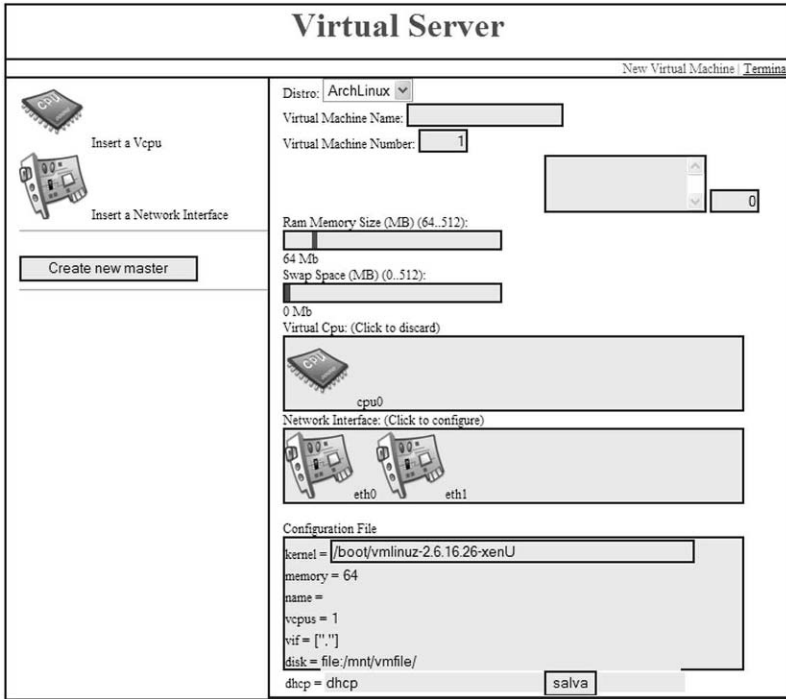
Fig. 8.4: The OVL Administration Interface (OVL-AI).

8.3.5 OVL Administration Interface

The Administration Interface (OVL-AI) module lies at the core of the OVL environment. OVL-AI enables simple management of the entire system via a straightforward Web interface (see Figure 8.4). OVL-AI provides a simplified procedure for the creation, configuration, and disposal of single virtual machines, or pools of virtual machines. Configuration is performed by choosing visually the simulated hardware cards to be inserted in each virtual machine. OVL-AI has been implemented following a multi-tiered approach. Namely, OVL-AI relies on *AJAX* on the client-side, on *PHP* on the server-side, and on *Bash*, for the interaction with the OVL server's operating system.

8.4 Using OVL to perform LTP tests

We now describe how the set of LPT tests used in CC certification process can be executed over a Xen-based virtualization platform like OVL.

Our experiments used a Xen virtual environment based on a Fedora Core 7 distribution, a general purpose Linux distribution developed by Fedora community and sponsored by Red Hat, which supports the Xen hypervisor in a native way. In addition, the Fedora distribution fully embraces the Open Source philosophy and wants "to be on the leading edge of free and open source technology, by adopting and helping develop new features and version upgrades" [7].

The main purposes of our experiments was to investigate and prove the reliability of a virtual environment as a base for the CC certification process. Today, in fact, the CC certification of a system or a product is strictly bounded to the TOE, that is, the precise HW configuration and the OS running over it (see Chapter 3). OS virtualization represents a key factor in limiting this drawbacks, if it provides the same security strength of an OS running on a physical machine. This would also make the testing and certification of an OS less costly, in terms of required hardware.

We then tested the OVL-based environment to prove that the LTP results under a virtualized system are the same of traditional testing, or at least have negligible variations.[6]

Machine Type	Kernel Type	Total failures over 860 tests
Fedora 7	2.6.20-2925.9.fc7**xen**	14
	2.6.21.7.fc7**xen**	14
	2.6.21-1.3194.fc7	9
	2.6.22.1-41.fc7	9
Fedora 8	2.6.21-2950.fc8**xen**	6
	2.6.21.7-2.fc8**xen**	6
	2.6.23.1-42.fc8	1

Table 8.1: LTP Test Results

Table 8.1 presents our results based on more than 860 tests. The discrepancies between physical and virtual results are 0.6% for Fedora 7, and 0.5% for Fedora 8, and are mostly caused by test growfiles having more than 13 repetitions. It is then probable that such discrepancies are not caused by security issues, but rather by tests generating a lot of I/O processes that cause failure due to the expiration timeout. Also, our experimental results show a minimal discrepancy between the LTP failures in the virtual and in the real machine (less then 1%) due to the kernel version, but no false negatives. In conclusion, our experiments prove that although a perfect matching between real and virtual environments is not possible, the Xen paravirtualization technique provides a reliable environment suitable for CC certification process.

---

[6] As discussed in Section 8.2.1, it is not possible to have exactly the same kernel version running on virtual and real systems, since the modified kernel version must be prepared to be integrated with Xen.

8.5 Conclusions

The availability of a testing infrastructure is a major factor in keeping software testing costs under control, especially as a part of test-based certification processes [12]. Linux SLES 8 CC certification tests showed practically no discrepancies when re-executed under a virtual Xen-based environment.

References

1. M. Anisetti, V. Bellandi, A. Colombo, M. Cremonini, E. Damiani, F. Frati, J.T. Hounsou, and D. Rebeccani. Learning computer networking on open paravirtual laboratories. *IEEE Transactions on Education*, 50(4):302–311, November 2007.
2. M. Anisetti, V. Bellandi, E. Damiani, F. Frati, U. Raimondi, and D. Rebeccani. The open source virtual lab: a case study. In *Proc. of the Workshop on Free and Open Source Learning Environments and Tools (FOSLET 2006)*, Como, Italy, June 2006.
3. *Bochs.* http://bochs.sourceforge.net/.
4. J. Collofello and K. Vehathiri. An environment for training computer science students on software testing. In *Proc. of the 35th Annual Conference Frontiers in Education (FIE 2005)*, Indianapolis, Indiana, USA, October.
5. D. Dobrilovic and Z. Stojanov. Using virtualization software in operating systems course. In *Proc. of the 4th IEEE International Conference on Information Technology: Research and Education (ITRE 2006)*, Tel Aviv, Israel, October 2006.
6. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Bolton Landing, NY, USA, October 2003.
7. Red Hat. Inc. Fedora objectives. http://fedoraproject.org/wiki/Objectives.
8. P.S. Magnusson. The virtual test lab. *Computer*, 5(95–97):38, May 2005.
9. G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.
10. *Qemu open source processor emulator.* http://bellard.org/qemu/.
11. M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technologies and future trends. *Computer*, 5(39–47):38, May 2005.
12. S. Seetharama and K. Murthy. Test optimization using software virtualization. *IEEE Software*, 5(66–69):23, September-October 2006.
13. J. Sugerman, G. Venkitachalam, and L. Beng-Hong. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proc. of the USENIX Annual Technical Conference 2002*, Monterey, CA, USA, June 2002.
14. The mach project home page. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html/.
15. G.K. Thiruvathukal. Gentoo linux: The next generation of linux. *Computing in Science & Engineering*, 6(5):66–74, September-October 2004.
16. *Vmware.* http://www.vmware.com/.
17. C. Wolf and E.M. Halter. *Virtualization from the Desktop to the Enterprise.* Apress, 2005.
18. *Xen.* http://www.xen.org/.
19. *z/VM.* http://www.vm.ibm.com/.

Chapter 9
Long-term OSS security certifications: An Outlook

9.1 Introduction

Throughout the book, we have seen that certifying security features of software products is a core requirement for commercial software suppliers. A major challenge for software vendors is to produce software that certifiably holds some security-related properties, such as, supporting discretionary access control or a lightweight security policy administration. In the case of open source communities, building more secure open source software involves several factors including: a repeatable assurance process, a higher security awareness of individuals and organizations contributing to open source communities, and above all the identification of an actor who, by requesting an open source certification, will bear some limited degree of accountability for the open software declared security features. Actors who may become involved in this process belong to the following five categories [1].

- *Chartered OSS Communities*. Today, most open source communities do not have formal charters enabling them to be held accountable, morally if not legally, for any specific claimed feature of their software products. There are however some notable exceptions, which correspond to the Linux distributions and to a relatively small number of major open source products.
- *OSS Forges*. Other potentially crucial actors are OSS *forges*, themselves emerging from the status of informally delivered services to the one of chartered consortia. Besides *SourceForge* (`http://sourceforge.net/`), which rightly claims to be "the global technology community's hub for information exchange, open source software distribution and services, and goods for geeks", other chartered subjects are emerging, such as the OW2 consortium (`http://www.ow2.org`), a non-profit organization whose role is to provide the governance and service framework through which suppliers and users of OSS platforms can work together.

- *OSS Consultants.* Companies whose business model includes administration, training and other services on OSS products.
- *OSS Corporate adopters.* Organizations bundling OSS into their products or adopting OSS for mission-critical applications. Such organizations may adopt different strategies when adopting OSS. A basic one ("just use it") is *opportunistic* OSS adoption, used as a leverage against commercial software vendors. A second strategy, more interesting for our purposes is *managed* adoption, which requires stable relationship with other actors (forges and/or communities), the creation and management of internal OSS skills and a necessity to keep some degree of involvement in the evolution of the OSS products. Experience has shown that when an OSS product becomes (under whatever license) an integral part of a company's products portfolio, the company is held accountable by its customers of the features of the entire portfolio, including the OSS part. Integrating security testing and security certification as part of open source software development process is therefore only a major priority for corporate adopters strategic effort towards build high quality products.
- *Software market.* The fact that OSS is distributed under an open source license does not mean it has no market. Users are forming their own communities, and customer groups and organizations increasingly show interest in extending the scope of certifications to include the application layer.

All these stakeholders have a vested interest in certifying OSS security features, and their cooperation (or lack thereof) will undoubtedly shape the future of OSS security certifications. Adoption of Common Criteria (CC) for the Linux operating system platform and more lightweight certification schemes like ICSA for security-related products are important steps toward building a collective awareness on this issue. However, as we have seen in Chapter 3, the CC security certification has the Achilles heel of a high context dependence. When a user receives a certificate on a software product, it may well be that the certificate was obtained (and therefore is valid) under lab conditions very different from the ones the user will encounter in practice. Another issue of CC (already discussed in Chapter 6) and stated in CC part 1 [6] regards the very notion of TOE: "In general, IT products can be configured in many ways: installed in different ways, with different options enabled or disabled. During a CC evaluation, it will be determined whether a TOE meets certain requirements, therefore, this flexibility in configuration may lead to problems, as all possible configurations of the TOE must meet the requirements. It is often the case that the guidance part of the TOE strongly constrains the possible configurations of the TOE. That is: the guidance of the TOE may be different from the general guidance of the IT product". This statement makes the TOE strictly bound to the configurations that have been used during the certification. This strong connection between security and configuration changes the way users look at the TOE, from the notion of a "certified product" to the one of a "certified configuration". So, evaluation

bodies do not really certify a software product but only specific configurations of it. In this chapter we shall examine the advantages and the issues of moving from today's context dependent, test-based certifications to hybrid, long-term certification of OSS. In the chapter's conclusions, we shall briefly discuss the role of the actors listed above in this evolution.

9.2 Long-term Certifications

While software platforms are heterogeneous in size and nature, many of them share an important feature: they are costly to develop and, therefore, must stay in service for a long time. Operating systems are increasingly expected to accommodate changes in underlying hardware; middleware and applications are expected to do the same w.r.t. modifications in the operating system configuration. In general, software is expected to preserve across configuration changes its original non-functional properties, including security, safety and dependability. When this expectation is not supported by a sound verification procedure, users get into a dangerous situation: while their reliance on software systems grows with the systems' service time, so does their vulnerability.

9.2.1 Long-lived systems

A software product's security features, and its corresponding resilience against threats, can be certified for the original deployment configuration only. In other words, CC-style, test-based certificates become invalid when the software configuration changes, since changes in the execution environment and in the overall context may compromise the certified system's security and reliability, creating problems that range from a decrease in the software users' productivity to dangerous vulnerabilities to external attacks.

On the other hand, software platforms such as banking, health-care, transportation and e-government systems (which we shall call collectively *Long-Lived Systems* (LLS)) tend to remain in service for a long time. Therefore, they need specific development-time and run-time techniques to certify their security, safety and dependability properties, possibly checking them against formal requirements contained in a *contract* (see Chapter 4) between the system and its users.

As far as open source development is concerned, the research problem underlying LLS is twofold.

- Even more than commercial software, open source platforms need to be verifiably secure (as opposed to informally claimed security), measurably secure (as opposed to vague best-effort security), and withstand not only

threats but also context changes and ageing. To really trust open source components in long-lived mission-critical applications, users must be able to determine, reliably and in advance, how they will behave. Such behavior can be certified using a combination of model and test based techniques. However, current OSS adoption and deployment practice come nowhere near this.

• At run-time, users should be provided with a mechanism and a solution that allows re-checking a system's properties when the system context changes. This way, security and dependability properties can be re-computed on demand, for instance, after a change in the context or on the execution environment.

In a nutshell, techniques for long-term OSS certification should provide a hybrid way of expressing security properties, which supports their dynamic re-checking with respect to security and dependability contracts, to tolerate changes in their deployment environment and configuration. Dynamic re-check is especially important when we consider changes for which function is preserved, but performance objectives or hardware technology are required to be different. Such changes are typical of two types of systems: *(i)* LLS that must be in service for very long time, which must evolve (in many cases, at least partly automatically) to remain in use when their context also evolves; and *(ii)* systems for emergent computing scenarios such as ubiquitous computing or ambient intelligence where it is not possible to foresee all possible situations that may arise at runtime. Both types share the difficulty that the knowledge needed for taking security and dependability-related design decision is either not available at development time, or is subject to change and evolution during the software system lifetime.

9.2.2 Long-term certificates

The functional, safety and dependability characteristics of every LLS module should be certified on the basis of integrated formal verification and testing. A fundamental aspect of this notion stating security and dependability properties in a uniform declarative format, and at a fine granularity level. *Long-term certificates* rely on three categories of properties, respectively derived from:

1. an abstract model-based specification of each module based on logics or a computational model such as an abstract infinite-state automaton (see Chapter 4);
2. a model reverse-engineered from the module code;
3. a set of tests associated to it (having a status of possibility: "the tests T do not disprove that property P holds").

The three categories of properties may not be all present for a given certificate; also they need not be disjoint; for instance, test cases can be automati-

cally generated from the abstract model and executed on a black-box implementation of the module. Properties of individual modules will be composed using automatic techniques based on approximated state-space exploration (see Chapter 4), to derive the properties of the overall system. The properties of the overall system will then be compared with the desired security and dependability profile.

The reader interested in the computational aspects of certifications should note that the techniques, which are sound for such a computation, may not be complete for a specific class of properties. Lack of completeness is a reasonable price to pay for efficient on-demand computation. However, the impossibility of proving a certain set of system-level properties is much less a concern when re-checking a set of properties already proved in the past; also, no erroneous validation of security and dependability profile is possible.

Long-term certificates should allow fast re-evaluation of security properties on demand, whenever the system configuration evolves. In particular, the definition of fine-grained security and dependability properties facilitates runtime composition of properties belonging to components in different systems configurations and coexisting in the same environment. Also, it will support discovery of relationships (e.g., subsumption, see Chapter 4) between the properties of a component in presence of evolving systems configurations.

The main idea and objective behind long-term certificates can be formulated as "permanently predictable systems". A major research problem is reducing current context dependency of certificates, making them hold even when software systems evolve or are moved to new computing platforms.

Let us now briefly review the goals to be attained by long-term security certification.

- *Enhancement of confidence.* Today, users of long-lived systems may get into a dangerous situation: while their reliance on software systems grows with the systems' service time, so does their vulnerability, due to accumulation of changes in the overall execution context and to the emergence of new attacks. A sound solution should provide a mechanism for continuous assessment of security and dependability properties, supporting long-term confidence in long-lived systems.
- *Long-term protection from threats.* Fast checking of security and dependability properties involving single components and entire systems should be supported, allowing for detecting breaches potentially exploitable on the part of attackers. Continuous evaluation of security and dependability contracts will allow protection against attacks or faults which were unknown at system design time.
- *Support for diverse development and deployment practices.* Capability of flexibly integrating code, model and test-based properties should ensure the methodology applicability to a diverse set of development processes. These processes include highly decentralized and incremental development and deployment practices, such as, those prevalent for large-scale and open source platforms.

A key aspect which must be dealt with in order to achieve the above goals is full integration of test-based and model-based certification. Open research issues related to this integration can be classified in the following two categories.

- *Predictable Systems Engineering.* Integration of test-based and formal methods for complex systems engineering, with special emphasis on security and dependability engineering, into a development process which seamlessly integrates test- and formal model-based properties. The major issue here is on formal methods application and tool-supported engineering activities, that is, requirements capturing, model analysis (validation and verification by model checking and theorem proving), test case generation, test case execution, refinement, abstraction, model transformation, safety assessment, metrics and certification.
- *Support for Dynamic System Evolution.* Support for the dynamic evolution of systems (time mobility), especially with regard to the provision of security and dependability, by means of mechanisms that are built-in in the system at development time, but work throughout the system's lifecycle. The work on this line will introduce important innovations with regards to the semantic specification of security and dependability requirements, dynamic re-check of such requirements after reconfiguration, self-* capabilities, automated adaptation, and runtime monitoring.

9.3 On-demand certificate checking

An important distinction that needs to be made at this point in the one between our notion of long-term certificates and the one of *proofs*. The latter term, as discussed in Chapter 4, is used to designate the concept introduced by G.C. Necula et al. [7, 8] of run-time demonstrations of program code properties. Obtaining such proofs at runtime is especially important in mobile applications, where some untrusted programs downloaded from the network might damage the host system or use too many local resources (CPU, memory, bandwidth). From a security point of view, executing untrusted code poses many threats, as it may reveal confidential data to an attacker, or disrupt the host system functionality.

Computing proofs carried by code can be seen as the runtime counterpart of computing model-based certificates, much in the same way as a trial execution in controlled execution environments (often called *sandboxes*) can be seen as a runtime counterpart of computing test-based certificates. The runtime proof-checking procedure usually goes as follows:

1. a software supplier, e.g., a remote Web site, provides a candidate program P for execution on an host environment H,

2. H produces a logic formula ψ that, if true, guarantees that P has the properties required for its execution,
3. a trusted external certifier C, receives P and ψ, computes a proof object π which establishes the validity of ψ, that is, proves $\psi(P)$,
4. H receives π and, using a trusted verifier V, checks to its satisfaction that π is a valid proof of $\psi(P)$. If this is the case, P can be executed on H.

The proof $\pi(P)$ can be seen as a certificate that the non-functional properties corresponding to ψ actually hold; the external certifier plays a role analogous to the one of the evaluation body producing a model-based certificate on a piece of code. Of course in principle the two vital modules C and V could be compromised by an attacker; therefore proof-checking system must define a notion of *minimum trusted computing base,* minimum trusted computing base composed by the modules that are required to be trusted for the entire construction to work [3]. Within this trusted computing base, proof systems can rely on encryption-based protocols for the establishment of a trust relationship. The logical model used for expressing the conditions (and computing their proof) is another crucial factor, as of course $\psi(P)$ must be decidable and the proof must be efficiently computed. In their seminal work [7], Necula and Lee used LF_i, a logical framework which allows to define logic systems with their proof rules and provide an efficient generic proof checker. By contrast, the corresponding certificates tend to be very big (sometimes even 1000 times bigger than the program they are supposed to certify). Finally, sometimes computing a proof is not enough, since H may adopt a *skeptical* attitude and inquire on the decision procedure used by C and a detailed proof must be generated showing the decision procedures. Classically, proof-checking was expected to ensure *soundness* and *completeness.* The active research trend has however focused on soundness; as we anticipated above, soundness is also a crucial point for long-term certificates.

The *model-carrying code* (MCC) paradigm [9, 10]) offers a general mechanism for enforcing security properties. In this paradigm, untrusted mobile code carries annotations that allow a host execution environment to verify its trustworthiness. Before running the code, the host environment checks the annotations and proves that they imply the host's security policy. Although the flexibility of the provided scheme, in the past compilers focused on simple type safety properties rather than more general security policies. Here, the main problem is that automated theorem provers cannot generate properties of arbitrary programs. Also, constructing proofs by hand is prohibitively expensive and the security policy needs to be shared and known a priori by both code producer and consumer.

The scenario we envision for long-term certificates is related but distinct to the proof and model-carrying code one. In particular, long-term certificates have the following characteristics:

- *Hybrid nature.* Long-term certificates need integrate test-based and model-based aspects [2]. They will include properties which can be proved on the basis of the program source code and others which require testing, and can be proved to an extent only.
- *Delayed verification.* In the case of certificates, proofs of assertions are computed by trusted external entities at an independently set verification (rather than at execution) time. Also, while certificates are verified on-demand, there is no need to link the verification to a specific program execution.

Furthermore, long-term certificates must be able to express and enforce more complex security policies while conciliating many other features, including small certificates, efficient verifiers, and effective tools to produce and distribute the certificates. The framework should be expressive enough to describe all different aspects of LLS, such as possible failure modes, fault-tolerance provided, security and dependability properties. Currently, formal techniques are not capable to deal with all these aspects: some are more easily expressed by properties derived from code analysis, while others are the traditional bailiwick of model checking.

9.4 The certificate composition problem

Traditionally, formal methods have focused on software verification, formalizing requirements into a specification, and using those specifications to get early and regular feedback, as to whether the implementation actually meets the requirements. Some efforts have been done to support design-time verification, as for instance, in design environments dedicated to safety-critical embedded software applications. As far as security and dependability requirements are concerned, however, design time verification is not enough, as new threats can emerge during the system lifetime. When this happens, usually it is too late to even try reproducing the formal verification procedure. This situation is closely related to the one of testing. In principle, a testing process can be repeated to support system (or context) evolution; however, the testing environment is (or becomes) forcibly different from the deployment one, making the lifespan of current test-based (e.g., CC) and model- based certifications much shorter than the one of the software system they are supposed to certify. Both formal verification and test-based techniques for assessing security and dependability properties are known to have additional weaknesses. Security models based on formal specification are a lot harder for developers to deal with than working prototypes, while unit tests offer, at best, rough case-wise specifications and leave plenty of scope for unforeseen corner cases, and unexpected behaviour. At the component development level, the usual "verification via unit testing" should be augmented with lightweight fine-grained formal specification coming from design models, as well as from code

analysis. Later, and even at run-time, composition can be applied to achieve the far more powerful security and dependability verification that becomes possible.

Long-term certificates must support agile and loose development processes which, like the OSS one, include rapid and regular customer feedback (see Chapter 5), while getting all the strength of formal methods for verification. The first source of security and dependability properties will be design models or specification. Instead of just adding new unit tests, the developer will add or refine a specification, much in the same way as she currently does when using standard languages and tools, like JML for Java, Spec# with C#, or Eiffel. Such component-level, fine-grained specifications are hopefully not harder to write than a unit test, but potentially very effective for preventing security attacks (e.g. the ones based on parameter passing). The second major feature of component-level operation is extracting models from automatic analysis of program code, and using these models to derive additional security and dependability properties. Properties coming from specifications and code analysis should then be integrated with other properties derived from automated testing. A hybrid strategy is also possible, exploiting formal specification as a test oracle and generate random test data. Test data can be generated by many different schemes, from purely random, to genetic algorithms, to schemes designed to provide maximal search space coverage. At the system level, properties of individual components need to be composed in a context-aware fashion. This is really the core of the problem, as security and dependability properties can be derived on-demand and checked against suitable, system-level security contracts. The notion of "security-by-contract" (see Chapter 4) is aimed at supporting long-lived systems, context evolution and the emergence of threats unexpected at the time of design should finally be integrated [4, 5].

9.5 Conclusions

As we have seen in Chapter 5, OSS development is based on a paradigm by which coding and debugging efforts are shared among the greatest possible number of developers, thus keeping individual effort at an acceptable level and improving the quality of the code. Open source is also a business development strategy. In some segments of the software market, such as operating systems, databases and middleware platforms, open source licensing was instrumental to breaking through existing entry barriers posed by commercial software vendors. The need of *Corporate adopters* for a physical counterpart to be held accountable for OSS products has been gradually fulfilled by *Consultants*, companies whose business model includes administration, training and other services on OSS products.

However, *Corporate adopters* and the *Software market* will look to other actors for certified security and dependability properties. Both *Chartered Communities* and *OSS Forges* need to actively support the development of an open source security assurance process. In particular, *Corporate adopters* have both technical and business expectations from OSS ass. On the technical side they need to support communities in achieving and keeping acceptable security and dependability levels, e.g., via technical expertise sharing. On the business side, they expect to be able to guarantee to their own customers and to the *Software market* that the security and dependability of their products and services will stay the same or improve due to OSS adoption. In principle, certificates and OSS go together well, as long-term certificates can represent and carry the information of software security and dependability needed throughout the open source ecosystem.

A potentially crucial role is to be played by *OSS Forges*. Forges provide trusted locations for other actors to gather information, download OSS programs and applications, and share knowledge and experience on open source software and other open technologies. Specifically, forges role is twofold: *(i) Information Aggregator*; and *(ii) Open Source Resource Repository*.

In the latter case, forges serve as a vendor-neutral, non-exclusive liaison node between *Corporate adopters* and numerous open source communities. Here we are interested in forges operating in the former capacity. Some of them are already active in the security area, by providing security and vulnerability alerts. In the fullness of time forges may play a normative role in a collaborative security assurance process for OSS, based on the notions of collaborative production, interchange and checking of long-term certificates.

References

1. S.A. Ajilaa and D. Wub. Empirical study of the effects of open source adoption on software development economics. *Journal of Systems and Software*, 80(9):1517–1529, September 2007.
2. E. Albert, G. Puebla, and M.V. Hermenegildo. Abstraction-carrying code. In *Proc. of the 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2004)*, Montevideo, Uruguay, March 2004.
3. A.W. Appel. Foundational proof-carrying code. In *Proc. of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*, Boston, MA, USA, June 2001.
4. N. Dragoni and F. Massacci. Security-by-contract for web services. In *Proc. of the ACM Workshop on Secure Web Services (SWS 2007)*, Fairfax, VA, USA, November 2007.
5. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. of the Fourth European PKI Workshop: Theory and Practice (EuroPKI 2007)*, Mallorca, Balearic Islands, Spain, June 2007.
6. The International Organization for Standardization and the International Electrotechnical Commission. *Common Criteria for Information Technology Secu-*

rity Evaluation, Part 1: Introduction and general model, 2006. `http://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R1.pdf`.

7. G.C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (LICS 1998)*, Indianapolis, IN, USA, June 1998.

8. G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *Proc. of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2001)*, London, UK, January 2001.

9. R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Model-carrying code (mcc): A new paradigm for mobile-code security. In *Proc. of the New Security Paradigms Workshop (NSPW 2001)*, New Mexico, USA, September 2001.

10. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, New York, USA, October 2003.

Appendix A
An example of a grep-based search/match phase

We provide the complete output produced by our matching engine when mapping between security function SC.1 and testcases is searched. Based on the following grep-based script

```
grep -l -i -e 'tunneling' -e 'port 22' -e 'secure channel'
        -e 'secure socket layer' -e 'ssl' -e 'ssh'
        -e 'secure shell' -r linux_security_test_suite_EAL3
```

the set of testcases to be used in the evaluation of security function SC.1 is retrieved.

```
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh04
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh01_s1
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh03
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh03_s1
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh02_s1
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh02
linux_security_test_suite_EAL3/ltp_EAL2/testcases/network/tcp_cmds/ssh/ssh01
linux_security_test_suite_EAL3/ltp_EAL2/testcases/user_databases/lastlog01
linux_security_test_suite_EAL3/ltp_EAL2/testcases/user_databases/faillog01
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/send_new_rsa_key.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/restore_date.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/rsa_login.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/rsa_test.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/dsatest.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/aestest.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/delete_accounts.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/shaltest.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/setup_date.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/rsa_auth.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/server.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/setup_accounts.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/rc4test.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/dsa_auth.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/stunnel_dsa.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/dhtest.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/password_auth.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/send_new_dsa_key.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/stunnel_rsa.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/dsa_login.sh
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/client.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/destest.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/test/randtest.c
linux_security_test_suite_EAL3/ltp_OpenSSL/testcases/openssl/openssl01
linux_security_test_suite_EAL3/laus_test/audit_tools/ssh01_s1
linux_security_test_suite_EAL3/laus_test/audit_tools/au_login
linux_security_test_suite_EAL3/laus_test/audit_tools/ssh01
linux_security_test_suite_EAL3/laus_test/libpam/tests/test_sshd.c
linux_security_test_suite_EAL3/laus_test/libpam/libpam.c
linux_security_test_suite_EAL3/laus_test/pam_laus/pam_laus.c
linux_security_test_suite_EAL3/laus_test/pam_laus/tests/test_sshd.c
```

Index