Electronic Notes in Theoretical Computer Science 80 (2009)
URL: http://www.elsevier.nl/locate/entcs/volume80.html 40 pages

A Case Study of a Formalized Security Architecture

Achim D. Brucker [1]

ETH Zürich

Burkhart Wolff [2]

Albert-Ludwigs-Universität Freiburg

Abstract

CVS is a widely known version management system, which can be used for the distributed development of software as well as its distribution from a central database.

In this paper, we provide an outline of a formal security analysis of a CVS-Server architecture performed in [1]. The analysis is based on an abstract architecture (enforcing a role-based access control on the repository), which is refined to an implementation architecture (based on the usual discretionary access control provided by the POSIX environment). Both architectures serve as framework to formulate access control and confidentiality properties.

Both the abstract as well as the concrete architecture are specified in the language Z. Based on a logical embedding of Z into Isabelle/HOL, we provide formal, machine-checked proofs for consistency properties of the specification, for the correctness of the refinement, and for some security properties.

Thus, we present a case study for the security analysis of realistic models over an off-the-shelf system by formal machine-checked proofs.

Key words: security, access control, POSIX, Unix, CVS, Z

1 Introduction

These days, the *Concurrent Versions System* (CVS) is a widely used tool for version management in many industrial software development projects, and plays a key role in open source projects usually performed by highly distributed teams [3,5,4]. CVS provides a central database (the *repository*) and

[1] Email: brucker@inf.ethz.ch
[2] Email: wolff@informatik.uni-freiburg.de

©2009 Published by Elsevier Science B. V.

© 2003 Elsevier Science B. V.. This is the author's version of the work. It is posted at http://www.brucker.ch/bibliography/abstract/brucker.ea-case-2003 by permission of Elsevier Science B. V. for your personal use. Not for redistribution. The definitive version was published in *Electronic Notes in Theoretical Computer Science 80 (2003) 24–40*.

means to synchronize local modifications of partial copies (the *working copies*) with the global repository. CVS can be accessed via a network; this requires a security architecture establishing authentication, access control and non-repudiation. A further complication of the CVS security architecture stems from the fact that the administration of authentication and access control is done via CVS itself; i.e. the information for authentication is accessed and modified via standard CVS operations.

The default CVS server has a number of security shortcomings (e.g. [12]). In this paper, we focus on two particular aims of an improved CVS server configuration presented in [1], which had been achieved by a formal analysis: The first aim of our work is to provide a particular configuration of a CVS server that enforces a role-based access control security policy [14]. Our second aim is to develop an "open CVS-Server architecture", where the repository is part of the usual shared filesystem of a local network and the server is a regular process on a machine in this network. While such an architecture has a number of advantages, the correctness and trustworthiness of the security mechanisms become a major concern.

Unfortunately, since we strive for modeling the sometimes hairy reality in operating systems, we cannot present the complete specification or any proofs here and refer the interested reader to [1]. Consequently, we will only *outline* this case study here in order to exemplify the contribution of this paper:

(i) we present a certain *modeling technique* — called architectural modeling — which has an abstraction level in-between the usual behavioral modeling used in protocol analysis and security requirements analysis on the one hand and code verification on the other,

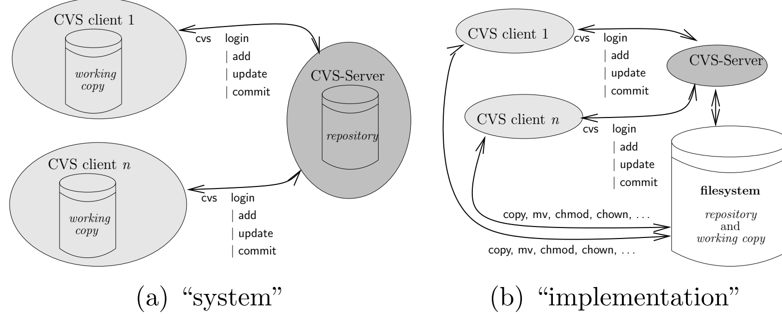
(ii) we use a technique to flatten-out concurrency in the architecture into a fairly coarse transition relation of the combined client-server system,

(iii) we present (partly reusable) models for widely used security technologies,

(iv) and we set up the mapping from security requirements to concrete security problems as a standard data refinement problem.

Thus, we present a *method* using formal methods for analyzing the access control problems of complex system technology and its configuration. Moreover, security analysis can be performed not only on the abstract, but also on the concrete level too.

As a means to identify conceptual entities of the problem domain as well as a means to structure the overall specification, we found it useful to describe the *architecture* of the system on several abstraction layers. Following Garlan and Shaw's approach [7,15], architectures are composed by *components* (such as *clients*, *servers* or *stores* like the filesystem) and *connectors* (like *channels*, *shared variables*, etc). In this terminology it is straight-forward to make the mentioned architectures more precise (as implementation architecture, we present the intended "open-Server Architecture", see Fig. 1): For each operation (such as `add`, `commit`,...) we assume a shared variable as connector that



(a) "system" (b) "implementation"

Fig. 1. The different CVS-Server architectures

keeps all necessary information that goes to and from the components; this paves the way for our approach to formalize this architecture by describing the transition relation of the combined system by the parallel composition of the local transition relations of the components synchronized over the corresponding shared variable. Since such transition relations can be represented in Z [9] by *operation schemas*, we can thus define, for example:

$$CVS_add = Client_add \land Server_add \setminus add_{\text{shared variable}}$$

where '$\land$' is the schema-conjunction and '$\setminus$' the hiding operator (i.e. an existential quantifier). We have a means to describe the transitions of our architectures by operation schemas in a standard specification formalism, and can therefore represent traces (i.e. sequences of all possible transitions).

As specification formalism, we chose Z [16] for the following reasons: first, Z fits to our type of architectures since the connectors are primitive and can be factored out and the complex states of our components suggest to use a formalism with rich theories for data-structures. Second, syntax and semantics are specified in an ISO-standard [9]; for future standardization efforts of operating system libraries (such as our POSIX [17] model in Section 2.3.1), Z is therefore a likely candidate. Third, Z comes with a data-refinement notion [16, pp. 136], which gives us a formal correctness notion of the underlying "security technology mapping" between the two architectures and a means to compute the proof obligations. We assume a rough familiarity with Z (the interested reader is referred to excellent textbooks on Z such as [18]).

As modeling and theorem proving environment, we chose Isabelle/HOL-Z 2.0 [2], which is an integrated documentation/type-checking/theorem proving environment for Z specifications. Isabelle [10] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church's *higher-order logic* (HOL) [8], a classical logic with equality. Isabelle/HOL-Z is a conservative embedding of Z into HOL (which is semantically isomorphic to Z since Z is based on typed set-theory and HOL on typed λ-calculus). As a result, Isabelle/HOL-Z combines up-to-date theorem prover technology with

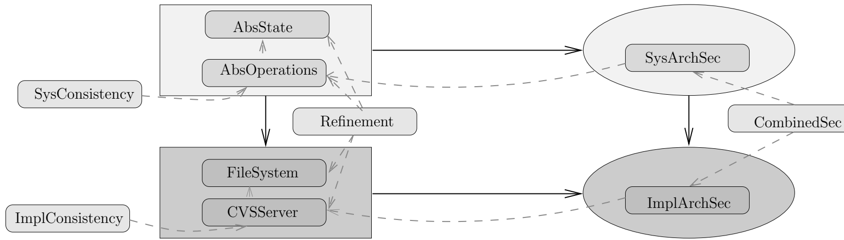


Fig. 2. The Specification Organization

a widespread specification formalism and powerful documentation facilities.

The paper is organized as follows: First, we present the abstract system architecture model, second, the model of the POSIX filesystem as an infrastructure for the implementation architecture embedded into it, and third the implementation architecture itself. Further, we describe the refinement relation between the system architecture and the implementation architecture, the security properties at the different layers and their analysis.

2 The CVS-Server Case Study

The specification of CVS-Server [1] consists of more than 100 pages, and the associated proof scripts are about 3000 lines of code. Its organization into Z-sections directly follows the overall scheme presented in Fig. 2. The Z-sections `AbsState` and `AbsOperations` describe the abstract system architecture of the client and the server components. The Z-section `SysConsistency` contains the consistency conditions (conservativity of axiomatic definitions, definedness of applications, non-blocking operation schemas) of the system architecture. This is mirrored at the implementation architecture level by the corresponding structures `FileSystem`, `CVSServer` and `ImplConsistency`. The section `Refinement`, which contains the usual abstraction predicate relating abstract and concrete states, also contains the proof obligations for the refinement. The security properties are defined and the corresponding proof obligations are postulated in the sections `SysArchSec` and `ImplArchSec`.

In the following only those parts of schemas and other Z definitions will be presented that are necessary to point out a certain feature of our model.

2.1 Entities of the Security Model

Following the standard role-based access model (RBAC) [14], we introduce abstract types for CVS users *Cvs_Uid*, permissions *Cvs_Perm* (which are isomorphic to "roles" in our setting), and passwords *Cvs_Passwd* used to authentify a CVS user for a permission. Permissions are organized in a hierarchy formalized by the reflexive, transitive relation *cvs_perm_order* (over CVS permissions *Cvs_Perm*) with *cvs_adm* as greatest element.

In the following, we discuss the security entities and mechanisms of CVS servers and clients in more detail. Working copies and repositories have both

maps that assign *abstract names* Abs_Name to *data* Abs_Data, types that are left abstract. The map is formalized as:

$$ABS_DATATAB == Abs_Name \pfun Abs_Data$$

A CVS server provides an authorization table, which is used to control access within the repository; this table is part of the servers repository itself and can be changed dynamically, i.e. the role cvs_adm may introduce CVS users and grant them permissions or withdraw them. The server stores to each file in the repository the permission that is necessary to access the file.

$$\begin{aligned} AUTH_TAB &== Cvs_Uid \times Cvs_Passwd \pfun Cvs_Perm \\ ABS_PERMTAB &== Abs_Name \pfun Cvs_Perm \end{aligned}$$

CVS clients possess in their working copy not only the data maps discussed above, but also a map that assigns to each abstract name a cvs user. Further, there is a map that associates cvs user's their password previously used during the cvs login procedure. Thus, for any data in the working copy and whenever an access to it may be processed, an individual role may be generated and validated by the server from the cvs user and password.

$$\begin{aligned} ABS_UIDTAB &== Abs_Name \pfun Cvs_Uid \\ PASSWD_TAB &== Cvs_Uid \pfun Cvs_Passwd \end{aligned}$$

2.2 The System Architecture

Modeling the server state via a Z schema is straight-forward. The state contains the data map rep and the map $rep_permtab$ containing the required permissions for each file. Our CVS-Server stores the authentication data inside rep, thus it can be accessed and modified with CVS operations. Therefore we require a abstract name $abs_cvsauth$ to be associated with data, that can be converted into an authentication table via a postulated function $authtab$. Accessing the authentication table will require to have the role cvs_adm.

ClientState models the state of the client component containing the working copy wc, the wc_uidtab assigning a cvs user to each file and a password table (modeling the file .cvspass). Further, there is a set of abstract names $wfiles$ which is used as filter in update and commit operations; this filter corresponds to the concept of the *working directory* in the implementation, that restricts the effect of these operations to files below the working directory.

RepositoryState

$$\begin{aligned} & rep : ABS_DATATAB \\ & rep_permtab : ABS_PERMTAB \\ \hline & abs_cvsauth \in \operatorname{dom} rep \\ & \operatorname{dom} rep = \operatorname{dom} rep_permtab \\ & rep_permtab(abs_cvsauth) = cvs_adm \end{aligned}$$

ClientState

$$\begin{aligned} & wc : ABS_DATATAB \\ & wc_uidtab : ABS_UIDTAB \\ & abs_passwd : PASSWD_TAB \\ & wfiles : \mathbb{P}\, Abs_Name \end{aligned}$$

We define the abstract CVS operations that model combined state transi-

tions of the client and the repository. Here, we only present login and update.

The login operation, simply stores authentication data on the client-side. This is used to authenticate a CVS user for a permissions of the client. The Δ and Ξ notation is used in Z to import the schemas in two variants, one just as a copy, the other by replacing all variables with a stroke $'$, describing the post state of the operation. The Ξ also introduces equalities enforcing that the pre state components are equal to the post state components.

abs_login

$\Delta ClientState$
$\Xi RepositoryState$
$passwd? : Cvs_Passwd$
$uid? : Cvs_Uid$

$(uid?, passwd?) \in \mathsf{dom}(authtab\ rep)$
$abs_passwd' = abs_passwd \oplus \{uid? \mapsto passwd?\}$
$wc' = wc \land wc_uidtab' = wc_uidtab \land wfiles' = wfiles$

The update operation updates every file in the working copy if the client has sufficient permissions. The overriding of the working copy by the repository is controlled by the predicate is_valid_in which checks if a cvs user and a password represent a valid permission according to the state of the authentication table in *rep*. The cvs user table is extended for files with role-names that can be validated in the authentication table to the required permission (an underspecified function *choose_valid_rolename* suffices here). Please note that this operation does not block if the client does not have sufficient permissions, but silently ignores files for which this is the case.

abs_up

$\Delta ClientState$
$\Xi RepositoryState$
$files? : \mathbb{P}\ Abs_Name$

$wc' = wc \oplus \{n : wfiles \cap files? \mid n \in \mathsf{dom}\ rep \land n \in \mathsf{dom}\ wc_uidtab$
$\quad \land (wc_uidtab(n), abs_passwd(wc_uidtab\ n))\ \mathsf{is_valid_in}\ \ rep\} \lhd rep$
$wc_uidtab' = wc_uidtab \cup \{n : wfiles \cap files? \mid n \in \mathsf{dom}\ rep$
$\quad \land n \notin \mathsf{dom}\ wc_uidtab \bullet n \mapsto choose_valid_rolename(rep_permtab, n)\}$
$abs_passwd' = abs_passwd \land wfiles' = wfiles$

2.3 The Implementation Architecture

The implementation is based on a Unix-based CVS server [3]. Therefore, the implementation has to cope with the full range of POSIX methods for accessing files and changing their access attributes. A realistic model of POSIX is therefore a necessary prerequisite in order to study the implementation of our security model and to analyze attacks on the implementation level. We

derived the POSIX model by formalizing the specification documents [17] and detailed system descriptions [6] and validated it by carefully chosen tests and inspections of critical parts of the system sources. In our POSIX model, the *CVS Filesystem* is embedded — i.e. a repository is described as some area in the filesystem, where file attributes are set in a suitable way, etc.

2.3.1 Modeling the POSIX Filesystem Access Control

We declare basic abstract sorts for POSIX user IDs, group IDs, data (file contents left abstract in this model) and filenames. Thus, we assume a static table *groups* that assigns to each user a set of groups he belongs to. The axiomatic definition below also states the existence of a special user ID *root*, the system administrator (usually called *root*). In principle all security relations can only hold for all users except *root*, because root is allowed to do (almost) everything.

$$[Uid, Gid, Data, Name]$$
$$Path == \mathrm{seq}\, Name$$

$$\begin{array}{|l} groups : Uid \rightarrow \mathbb{P}\, Gid \\ root : Uid \end{array}$$

Within POSIX, every file belongs to a unique pair of owner (user) and group, and file access is divided into access by the *user* (owner), the *group* or *other* (world). The POSIX *discretionary access control* (DAC) distinguishes access for reading (r), writing (w), and executing (x). We also model the "set group id" (sg) on directories, which affects the default group of newly created files within that directory [6]:

$$Perm ::= ru \mid wu \mid xu \mid rg \mid wg \mid xg \mid ro \mid wo \mid xo \mid sg$$

The filesystem consists of files, which are represented by mapping to each path the file content (either *Data* for regular files or *Unit* for directories[3]) and of file attributes (assigning to each file or directory the permissions[4], the user ID of the owner and the group it belongs to).

$$\begin{array}{ll} FILESYS_TAB & == Path \rightarrow\!\!\!\!\!+ (Data + Unit) \\ FILEATTRIBUTES & == [perm : \mathbb{P}\, Perm;\ uid : Uid;\ gid : Gid] \\ FILEATTR_TAB & == Path \rightarrow\!\!\!\!\!+ FILEATTRIBUTES \end{array}$$

At this point we are ready to model the filesystem state, which mainly describes the map of (name) paths to their attributes. As mentioned, we require that all defined paths must be "prefix-closed", i.e. all prefix paths must be defined in the filesystem (thus constituting a tree) and point to directories.

In addition to the filesystem state, we introduce a state schema for client related information, namely the current user and group ID, the client's umask (which is used to set the initial file attributes on new files) and current working directory (*wdir*). The working directory is often used as an implicit parameter

[3] We do not consider *special files*, like devices, named pipes or process files.

[4] The terms *attributes* and *permissions* are used interchangeably.

to filesystem and CVS operations:

$$
\begin{array}{|l}
\hline
\textit{FileSystem} \\
\textit{files} : \textit{FILESYS_TAB} \\
\textit{attributes} : \textit{FILEATTR_TAB} \\
\hline
\forall\, p : \mathsf{dom}\,\textit{files} \bullet (p = \langle\rangle) \\
\qquad \vee\ (\textit{front}(p)\ \mathsf{is_dir_in}\ \textit{files}) \\
\mathsf{dom}\,\textit{files} = \mathsf{dom}\,\textit{attributes} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ProcessState} \\
\textit{uid} : \textit{Uid} \\
\textit{gid} : \textit{Gid} \\
\textit{umask} : \mathbb{P}(\textit{Perm} \setminus \{\textit{sg}\}) \\
\textit{wdir} : \textit{Path} \\
\hline
\end{array}
$$

As an example for our approach to specify POSIX-operations, we present the (shortened) file remove specification [17], which corresponds to unlink():

The unlink() function shall fail and shall not unlink the file if:

- *A component of path does not name an existing file . . .*
- *Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the directory entry to be removed.*

This text is formalized by a Z operation schema *rm* as follows: The first condition in the body is common for most filesystem operations and requires the path of the file must be a valid one in the filesystem table. The second condition requires that the client has write permissions on the file and the working directory ("*the directory containing the directory entry to be removed*"), which is checked via the *has_w_access* predicate:

$$
\begin{array}{|l}
\hline
\textit{rm} \\
\Delta \textit{FileSystem};\ \Xi \textit{ProcessState} \\
u? : \textit{Name} \\
\hline
(\textit{wdir} \frown \langle u? \rangle)\ \mathsf{is_file_in}\ \textit{files} \\
\textit{has_w_access}(\textit{uid}, \textit{wdir}, \textit{attributes}) \\
\qquad \wedge\ \textit{has_w_access}(\textit{uid}, \textit{wdir} \frown \langle u? \rangle, \textit{attributes}) \\
\textit{files}' = \{\textit{wdir} \frown \langle u? \rangle\} \lhd \textit{files} \wedge \textit{attributes}' = \textit{attributes} \\
\hline
\end{array}
$$

For space reasons, we omit the presentation of is_file_in and *has_w_access*. The other filesystem operations defined similarly, see [1] for details.

2.3.2 Mapping CVS Access Control onto POSIX DAC

Instead of mapping CVS roles and the access control policy on complex or non-standard operating system mechanisms (such as access control lists (ACL)), our setup requires only standard POSIX DAC: any CVS role will be mapped to a particular pair of system *owner* and a set of system *group*s. When creating new objects in the repository, an inheritance mechanism for generating default roles is provided. Users can "down-scale" the permissions in the repository, while "up-scaling" permissions is only possible for the CVS administrator.

For every CVS operation, the server determines the CVS permissions according to the client's CVS ID and password. These permissions are then

mapped to POSIX user and group IDs, and these are compared to the file attributes of the files and directories the operations operates on. This translation is done by the two functions *cvsperm2uid* and *cvsperm2gid*. It is important to notice that CVS IDs (*Cvs_Uid*) are independent of POSIX IDs (*Uid*) and that the POSIX IDs which are used by CVS are disjoint from "normal" POSIX user IDs, i.e. it is impossible to login with such a special POSIX ID.

From these distinctness constraints follows that the POSIX system administrator and the CVS administrator may be different. Moreover, we require that the group table (administrated by the system administrator and nobody else) is compatible with *cvs_perm_order*.

The CVS repository is a subtree in the normal filesystem, its root is denoted by the absolute path *cvs_rep*. All paths inside the repository are relative to *cvs_rep*. The administrative files of CVS are stored in the *CVSROOT* directory, which is a subdirectory of *cvs_rep*, and the file that contains all authentication information is called *cvsauth* and is located inside *CVSROOT*.

2.3.3 Modeling the CVS Filesystem

A major design decision for our specification is to enrich the *FileSystem* state by new state components relevant to CVS, or more precisely, the combined client/server component of CVS. In CVS, working copies contain specific attributes assigned to the files; we restrict ourselves to security relevant attributes, i.e. the CVS user ID and password, and the path *rep* where the file is located in the repository. This information is kept in an own table implicitly associated to the working copies. For simplicity, we require that the client has write permissions for his working copy.

$$
\begin{array}{ll}
CVS_ATTRIBUTES & == [rep : Path;\ f_uid : Cvs_Uid] \\
CVS_ATTR_TAB & == Path \nrightarrow CVS_ATTRIBUTES
\end{array}
$$

We introduce two state variables *cvs_uid* and *passwd* in the *Cvs_FileSystem* (in the implementation they are set by *cvs_login*). Due to the limited space, we only show some requirements of the combined POSIX and CVS filesystem:

- Working copies and the repository are distinct areas of the filesystem.
- The repository contains a special directory that contains the administrative data of CVS. Certain restrictive access permissions must be ensured to this directory and its contents to preserve the system integrity.
- General requirements on file attributes within the repository:
  - The owners of files must be POSIX user IDs that are disjoint from "regular" POSIX user IDs, and the group IDs must be legal w.r.t. the CVS role hierarchy. Therefore a regular user has only the rights for *others* and as such a regular user cannot use any POSIX operation within the repository.
  - Read, write and execute permissions are the same for users and groups. Together with our group setup this ensures that the initial CVS role and all roles with higher precedence have the same access rights on that file.

These two invariants are formally described in the axiomatic definitions:

$$rep_attributes_ : \mathbb{P}\, FileSystem$$

$$\begin{array}{l}\forall fs : FileSystem \bullet rep_attributes(fs) \Leftrightarrow \\ \quad (\forall p : \mathsf{dom}\, fs.files \mid (cvs_rep\ \mathsf{prefix}\ p) \bullet \\ \quad (((fs.attributes\, p).uid) \in \mathsf{ran}\, cvsperm2uid \wedge \\ \quad ((fs.attributes\, p).gid) \in groups((fs.attributes\, p).uid) \wedge \\ \quad (ru \in ((fs.attributes\, p).perm) \Leftrightarrow rg \in (fs.attributes\, p).perm) \wedge \\ \quad (wu \in ((fs.attributes\, p).perm) \Leftrightarrow wg \in (fs.attributes\, p).perm) \wedge \\ \quad (xu \in ((fs.attributes\, p).perm) \Leftrightarrow xu \in (fs.attributes\, p).perm)))\end{array}$$

We turn now to a formal description of the repository *within* the filesystem; this system invariant is captured in the state schema *Cvs_FileSystem*. Additionally to *rep_attribues*, we impose similar requirements for the administrative area of the repository by the predicate *admin_attributes*, and we define requirements for the data in the repository, i.e. the files that are subject to version control, in the predicate *data_attributes* (both predicate formalizations are omitted here).

$$\begin{array}{l}\textbf{Cvs_FileSystem} \\ FileSystem \\ wcs_attributes : CVS_ATTR_TAB \\ cvs_passwd : PASSWD_TAB \\ \hline \mathsf{dom}\, wcs_attributes \subseteq \mathsf{dom}\, files \\ (cvs_rep \frown \langle CVSROOT, cvsauth\rangle\ \mathsf{is_file_in}\ files) \\ attributes(cvs_rep) = \langle\!|\ perm == \{ru, wu, xu, xg, sg\}, \\ \qquad\qquad uid == cvsperm2uid(cvs_adm), \\ \qquad\qquad gid == cvsperm2gid(cvs_public)\ |\!\rangle \\ rep_attributes(\theta FileSystem) \\ ((attributes(cvs_rep \frown \langle CVSROOT\rangle)).gid) = cvsperm2gid(cvs_adm) \\ admin_attributes(\theta FileSystem) \wedge data_attributes(\theta FileSystem)\end{array}$$

Now we have established a basis for the operations on the combined POSIX and CVS environment. As in Sec. 2.2, we present the login and update in order to compare the the two different architecture levels.

The login operation mainly updates the variable *cvs_passwd*, provided that for the combination of user ID and password the authentication will succeed.

$$\begin{array}{l}
\textbf{cvs_login} \\
\Delta Cvs_FileSystem;\ \Xi ProcessState \\
cvs_uid? : Cvs_Uid;\ cvs_pwd? : Cvs_Passwd \\
\hline
(cvs_uid?, cvs_pwd?) \in \mathsf{dom}(get_auth_tab\ files) \\
cvs_passwd' = cvs_passwd \oplus \{cvs_uid? \mapsto cvs_pwd?\} \\
wcs_attributes' = wcs_attributes \\
\theta FileSystem = \theta(FileSystem)'
\end{array}$$

In the update operation, the current working directory *wdir* can be restricted by the parameter $p?$ to just one file or directory. All files below $p?$ for which the client has access will be updated.

$$\begin{array}{l}
\textbf{cvs_update} \\
\Delta Cvs_FileSystem;\ \Xi ProcessState \\
p? : Path \\
\hline
cvs_rep \frown (wcs_attributes\ wdir).rep \frown p? \in \mathsf{dom}\ files \\
has_w_access(uid, wdir \frown p?, attributes) \\
files' = files \oplus \\
\quad \{q : rep_access(\theta Cvs_FileSystem)((wcs_attributes\ wdir).rep \frown p?) \bullet \\
\qquad wdir \frown q \mapsto files(cvs_rep \frown q)\} \\
attributes' = attributes \oplus \\
\quad \{q : rep_access(\theta Cvs_FileSystem)((wcs_attributes\ wdir).rep \frown p?) \bullet \\
\qquad wdir \frown q \mapsto \langle\!|\ perm == \varnothing, uid == uid, gid == gid\ |\!\rangle\} \\
wcs_attributes' = wcs_attributes \cup \\
\quad \{q : Q(\theta Cvs_FileSystem)((wcs_attributes\ wdir).rep \frown p?) \mid \\
\qquad wdir \frown q \notin \mathsf{dom}\ wcs_attributes \bullet \\
\qquad wdir \frown q \mapsto \langle\!|\ rep == q, f_uid == choose(\theta Cvs_FileSystem, q)\ |\!\rangle\} \\
cvs_passwd' = cvs_passwd
\end{array}$$

In contrast to the system architecture specification we also must adjust the POSIX file attributes of the updated files. The particularity of the update operation is the use of *rep_access* which computes the paths into the repository to which the client has read access according to his CVS role.

$$\begin{array}{l}
rep_access : Cvs_FileSystem \to Path \to \mathbb{P}\ Path \\
\hline
\forall\ cfs : Cvs_FileSystem;\ p : Path \bullet rep_access(cfs)(p) \\
\quad = \{q : Path \mid p\ \mathsf{prefix}\ q \land cvs_rep \frown q \in \mathsf{dom}\ cfs.files \\
\quad \land (\exists\ idpwd : cfs.cvs_passwd \bullet idpwd \in \mathsf{dom}(get_auth_tab(cfs.files)) \\
\quad \land has_r_access(cvsperm2uid(get_auth_tab(cfs.files)(idpwd)), \\
\qquad cvs_rep \frown q, cfs.attributes))\}
\end{array}$$

3 Formal Analysis

3.1 Checking the Consistency

Two types of "sanity checks" are useful and have been routinely carried out with HOL-Z throughout this case study:

- Definedness checks for all applications of partial functions in their context, as undefined applications usually indicate that some part of the precondition of a schema context is missing, and
- checking the state invariant of all operation schemas; in particular, we require that in a schema, all syntactic preconditions (i.e. the conjuncts in the predicate part that contain occurrences of variables without stroke "'" and "!" suffix) suffice to show that the successor state exists.

Violating these consistency conditions does not results in inconsistencies but in unprovable statements or operation definitions with undesired semantical effects.

3.2 Establishing the Refinement

In order to prove that the concrete implementation architecture correctly implements the abstract system architecture, we have to define an abstraction schema R which relates the components of the abstract state schemas to the components of the concrete state schemas. In particular, we must map abstract names and data to paths and files in the sense of the POSIX filesystem.

To give an idea of these mappings, we illustrate some constraints of the abstraction schema. As a prerequisite, let us define a function $Rname2path$, which maps abstract names, i.e. files, to paths in the implementation model.

One constraint on the abstraction is that the authentication tables in both models are related, and that the authentication information is equal:

$$Rname2path(abs_cvsauth) = cvs_rep \frown \langle CVSROOT, cvsauth \rangle$$
$$authtab(rep) = get_auth_tab(files)$$

Further we consider the implicit arguments ($wfiles$ in the system architecture, $wdir$ in the implementation architecture) to CVS operations like commit and update. Since the path $wdir$ represents all possible paths that have $wdir$ as a prefix, we require that all names in $wfiles$ are related:

$$\forall\, n : wfiles \bullet wdir \ \mathsf{prefix}\ Rname2path(n)$$

The last example predicate we present here enforces the abstract working copy to have a counterpart in the implementation working copy:

$$Rname2path(\!|\mathsf{dom}\ wc|\!) = \mathsf{dom}\ wcs_attributes$$

To verify the refinement relation R, following [16], we must prove two refinement conditions for each operation on the abstract state and its cor-

responding operation on the concrete state: Condition (a) ensures that a concrete operation terminates whenever its corresponding abstract operation is guaranteed to terminate, condition (b) ensures that the state after the concrete operation represents one of those abstract states in which the abstract operation could terminate.

As an example of the refinement, we show the instantiation of conditions (a) and (b) for the CVS login operation. The refinement conditions, though, as defined in [16], assume that both operations have the same input parameters, but since we define them differently in our two models, we have to introduce an additional schema *Asm*, which is used to insert further assumptions into the refinement proofs. In the case of the login operation, these are simple (they differ only in their names) although one could imagine arbitrarily complex relations between the arguments of other abstract and concrete operations.

$$\begin{array}{l} \textit{Asm} \\ \hline \textit{passwd?}, \textit{cvs_pwd?} : \textit{Cvs_Passwd} \\ \textit{uid?}, \textit{cvs_uid?} : \textit{Cvs_Uid} \\ \hline \textit{passwd?} = \textit{cvs_pwd?} \land \textit{uid?} = \textit{cvs_uid?} \end{array}$$

Instantiating condition (a) and (b) for the login operation and adding the assumption schema *Asm* leads to the following two proof obligations:

$$\begin{array}{l} \textit{login}_a == \forall\, \textit{ClientState};\ \textit{RepositoryState};\ \textit{ProcessState};\ \textit{Cvs_FileSystem}; \\ \quad \textit{passwd?}, \textit{cvs_pwd?} : \textit{Cvs_Passwd};\ \textit{uid?}, \textit{cvs_uid?} : \textit{Cvs_Uid} \bullet \\ \qquad \textit{Asm} \land \mathsf{pre}\ \textit{abs_login} \land R \Rightarrow \mathsf{pre}\ \textit{cvs_login} \\ \textit{login}_b == \forall\, \textit{ClientState};\ \textit{RepositoryState};\ \textit{ProcessState};\ \textit{Cvs_FileSystem}; \\ \quad \textit{ProcessState}';\ \textit{Cvs_FileSystem}';\ \textit{passwd?}, \textit{cvs_pwd?} : \textit{Cvs_Passwd}; \\ \qquad \textit{uid?}, \textit{cvs_uid?} : \textit{Cvs_Uid} \bullet \textit{Asm} \land \mathsf{pre}\ \textit{abs_login} \land R \land \textit{cvs_login} \\ \qquad\quad \Rightarrow (\exists\, \textit{ClientState}';\ \textit{RepositoryState}' \bullet R' \land \textit{abs_login}) \end{array}$$

The obligations for the other operations are defined analogously. So far, we proved these obligations formally for the refinement of login, add and update. These proofs helped a lot to debug our specifications and find subtle side-conditions that had to be dealt in *Cvs_FileSystem* and thus to get our real repository configuration “right”.

3.3 Security Properties of Both Architecture Layers

Analyzing security properties means considering the set of possible *sequences* of operations (*traces*) and postulating requirements on the possible states the system may reach. Hence, the specification of the security properties motivates a Z section each on the system architecture and the implementation architecture containing a classical *behavioral* specifications. In section `SysArchSec` we investigate security properties of the system architecture, and in section `ImplArchSec` we investigate the same properties and additional ones that are specific to the implementation architecture.

Methodically, we need an interface between the operation schemas of the two architecture layers and the behavioral part allowing to specify safety properties. This is done by converting the suitably restricted operation schemas of both system layers into explicit relations over the underlying state:

$$\begin{array}{lcl} op_1R & = & op_1 \wedge R_1 \\ & \cdots & \\ op_nR & = & op_n \wedge R_n \end{array} \qquad \begin{array}{ll} \text{step} & = op_1R \vee \ldots \vee op_nR \\ \text{trans} & = \{step \mid (\theta state, \theta state')\}* \\ \text{SecProp} & = \forall\, \text{trans}(\!|\ \text{init}\ |\!) \bullet P \end{array}$$

Here, op_iR represent the restricted operation schemas, their schema disjunction *step* the overall step relation of the system, that is converted into a transitively closed relation *trans*. The security property *SecProp* can be stated over the set of states reachable via *trans* from an initial state.

We instantiate this scheme: A client "knows" a set of pairs of roles and passwords, and "invents" only files from a given set of pairs from names to data in the add operation. We assume login being restricted to roles and passwords the client "knows" and add being restricted to data the client "invents".

$$\begin{array}{lll} abs_loginR & == & abs_login \wedge [cvs_uid? : Cvs_Uid;\ passwd? : Cvs_Passwd \mid \\ & & \qquad (cvs_uid?, passwd?) \in Aknows] \\ step & == & abs_loginR \vee abs_addR \vee abs_ci \vee abs_up \vee abs_cd \\ AbsState & == & ClientState \wedge RepositoryState \\ trans & == & \{step \bullet (\theta AbsState, \theta AbsState')\}^* \end{array}$$

The security property $SecProp_1$ is formulated for both architectures. Informally, the only difference between the two levels is that on the implementation level, we must define the step over CVS *and* filesystem operations, whereas on the system level only CVS operations are of concern.

The meaning of the security property is described as follows: Any sequence of CVS operations starting from an empty working copy does not lead to a working copy with data from the repository the client has no permission to.

We define the initial abstract state (empty working copy) and the set *Areachable*1, which contains the states that are reachable from it.

$$\begin{array}{rl} InitAbsState1 == & AbsState \wedge [wc : ABS_DATATAB \mid wc = \varnothing] \\ Areachable1 == & Atrans(\!|InitAbsState1|\!) \end{array}$$

*AProp*1 captures the actual security property: files in the working copy are either invented or the client knows a password to obtain sufficient permissions.

*AProp*1

$$\begin{array}{l} wc : ABS_DATATAB \\ rep : ABS_DATATAB \\ rep_permtab : ABS_PERMTAB \\ \hline \forall\, n : \mathsf{dom}\, wc \bullet (n, wc(n)) \in Ainvents \vee ((wc(n) = rep(n)) \wedge \\ (\exists\, m : Aknows \bullet (rep_permtab(n), authtab(rep)(m)) \in cvs_perm_order)) \end{array}$$

The complete security property is then defined by the schema expression:

$$AbsSecProp_1 == \forall\, Areachable1 \bullet AbsState \land AProp1$$

The proofs of these properties are essentially inductions over the transitive closure *trans*. Our approach is very similar to Paulson's inductive method on protocol verification [11]; the main difference is that the steps were defined in Z operation schema and not in inductive rules. Via the restriction schema R_i, side-conditions modeling the restricted use of operations in a particular variant of a security model can be introduced easily; for instance, sequences of operations may be constructed where all operations not use *cvs_adm* permissions followed by subsequences where operations do use it (modeling interference by the CVS administrator in withdraw-permission properties).

Unfortunately, *implementing* one security architecture by another opens the door to *new* types of attacks on the implementation architecture: on the implementation level, we have more operations available (the schema disjunction *step* additionally comprises the POSIX commands `cp` or `setumask`). Our technique is also applicable here: The essential difference in the analysis is that we define the step relation more liberally:

$$step_{impl} == cvs_cd \lor \ldots \lor cvs_chmod \lor cvs_login \lor \ldots \lor cvs_update$$

and introduce the side-conditions on the concrete level.

Although the proofs on the implementation architecture have the same structure as on the system architecture, they are far more complex since concepts such as paths, the distinction between files and directories, and their permissions are involved. Moreover, they require new side-conditions (for example, the refinement can only be established for the case that the user is not `root`; i.e. all security properties are not met, if some attacking client obtains `root` permissions on the filesystem level) which were systematically introduced by the abstraction predicate R.

On the other hand, the higher degree of detail on the implementation architecture makes a formalization of new types of security properties possible: For example, since the crucial concept "directory" is present on the implementation level and since the existence of files can only be established by having access to all father directories of a file, one can express confidentiality properties such as "the user can not find out that a file with name X exists in the repository" on this level.

4 Conclusion

4.1 Discussion

We presented a case-study of an access control security problem for an "real" system made amenable to formal, machine-based analysis. This demonstrates a *method* for analyzing the security in off-the-shelve system components: First,

specify the security architecture (as a framework for formal security properties), second, specify the implementation architecture (validated by inspecting informal specifications or testing code), third, set up the security technology mapping as a refinement, and forth, prove refinements and security properties by mechanized proofs. This method is applicable for a wider range of problems such as mission critical e-commerce applications or e-government applications.

It has been widely recognized that security properties can not be easily refined — actually, finding refinement notions that preserve security properties are a hot research topic. However, standard refinement proof technology has still its value here since it checks that abstract security requirements (which can be seen as *security against unintentional misuse*) are indeed achieved by a mapping to concrete security technology, and that implicit assumptions on this implementation have been made explicit. With respect to security against *intentional exploits of security leaks*, we believe that specialized refinement notions will be limited to restricted aspects of a system. For this problem, in most cases the answer will be to analyze the security on the implementation level, possibly by reusing results from the abstract level.

4.2 Related Work

Wenzel developed a specification of the basic Unix functionality. This specification was done in Isabelle/HOL and is part of the actual Isabelle [10] distribution. On the file system part, only a simple access model, not supporting groups and the concepts of set-id bits, is formalized.

Sandhu described in [13] a method for embedding role-based access control with the Discretionary Access Control provided by standard Unix systems. Our implementation used this construction for providing the static role.

4.3 Future Work

In our opinion, amazingly few work has been addressed to the specification of the POSIX interface; due to its often not intuitive features, its importance for security implementations and its high degree of reuse, this is a particularly rewarding target. We believe that our formalization is a good starting point for a comprehensive, more complete model of the filesystem related commands.

The formal proofs we did so far represent in our opinion a "proof of technology" for this type of reasoning, but we do not claim that they represent a *complete* analysis of the (real) CVS-server. So far, most consistency properties, but only selected refinement and security properties have been proven.

References

[1] Brucker, A. D., F. Rittinger and B. Wolff, *A CVS-Server security architecture — concepts and formal analysis*, Technical Report 182, Albert-Ludwigs-Universität Freiburg (2002).

[2] Brucker, A. D., F. Rittinger and B. Wolff, *HOL-Z 2.0: A proof environment for Z-specifications*, Journal of Universal Computer Science **9** (2003), pp. 152–172.

[3] Cederqvist, P. et al., "Version Management with CVS," (2000).

[4] http://www.cvshome.org.

[5] Fogel, K., "Open source development with CVS," The Coriolis Group, 1999.

[6] Frisch, Æ., "Essential System Administration," O'Reilly, 1995.

[7] Garlan, D. and M. Shaw, *An introduction to software architecture*, in: V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 1993 pp. 1–39.

[8] Gordon, M. J. C. and T. F. Melham, "Introduction to HOL," Cambridge University Press, 1993, 472 pp.

[9] *Z formal specification notation — syntax, type system and semantics* (2002), ISO/IEC 13568:2002, International Standard.

[10] Paulson, L. C., "Isabelle: a generic theorem prover," LNCS 828, Springer, 1994.

[11] Paulson, L. C., *The inductive approach to verifying cryptographic protocols*, Journal of Computer Security **6** (1998), pp. 85–128.

[12] http://mail-index.netbsd.org/tech-security/1997/06/27/0009.html.

[13] Sandhu, R. and G.-J. Ahn, *Decentralized group hierarchies in UNIX: An experiment and lessons learned*, in: *Proc. 21st NIST-NCSC National Information Systems Security Conference*, 1998, pp. 486–502.

[14] Sandhu, R. S., E. J. Coyne, H. L. Feinstein and C. E. Youman, *Role-based access control models*, Computer **29** (1996), pp. 38–47.

[15] Shaw, M. and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline," Prentice-Hall, 1996.

[16] Spivey, J. M., "The Z Notation: A Reference Manual," Prentice Hall International Series in Computer Science, Prentice-Hall, 1992.

[17] "The Single UNIX Specification Version 3," The Open Group and IEEE, 2002, this standard superseeds the "Single UNIX Specification Version 2" (Unix 98) and the "IEEE Standard 1003.1-2001" (POSIX.1).

[18] Woodcock, J. and J. Davies, "Using Z: Specification, Refinement, and Proof," Prentice Hall International Series in Computer Science, Prentice-Hall, 1996.