Verification of Security Policy Enforcement in Enterprise Systems*

Puneet Gupta and Scott D. Stoller

Abstract Many security requirements for enterprise systems can be expressed in a natural way as high-level access control policies. A high-level policy may refer to abstract *information resources*, independent of where the information is stored; it controls both direct and indirect accesses to the information; it may refer to the context of a request, i.e., the request's path through the system; and its enforcement point and enforcement mechanism may be unspecified. Enforcement of a high-level policy may depend on the system architecture and the configurations of a variety of security mechanisms, such as firewalls, host login permissions, file permissions, DBMS access control, and application-specific security mechanisms. This paper presents a framework in which all of these can be conveniently and formally expressed, a method to verify that a high-level policy is enforced, and an algorithm to determine a trusted computing base for each resource.

1 Introduction

Many security requirements for enterprise systems can be expressed in a natural way as high-level access control policies. These policies may be high-level in multiple ways. First, a high-level policy may refer to abstract *information resources*, independent of where the information is stored. For example, consider the requirement that only employees in the registrar's office may access student transcripts. This should apply regardless of whether the transcripts are all stored in one DBMS,

Puneet Gupta
Computer Science Dept., Stony Brook University, e-mail: pgupta@cs.stonybrook.edu

Scott D. Stoller
Computer Science Dept., Stony Brook University, e-mail: stoller@cs.stonybrook.edu

* This work is supported in part by ONR under Grant N00014-07-1-0928 and NSF under Grants CNS-0831298, CNS-0627447, CCF-0613913, and CNS-0509230. Email: {pgupta,stoller}@cs.stonybrook.edu.

partitioned (e.g., by campus, college, or grad/undergrad) among multiple DBMSs, saved in backup files, etc. Second, a high-level policy controls both direct and *indirect* accesses to the information. For example, the above policy implies that other users cannot read transcripts by directly accessing them in a DBMS or by invoking operations of an application (possibly running with a different userid) that accesses the database and returns information from the transcripts. Third, a high-level policy may refer to the *context* of a request, i.e., the request's path through the system. For example, a policy might state that employees in the registrar's office are permitted to access student transcripts only via a web browser running on a host in the campus network and requesting the information from the Registrar Application Server. Note that this is analogous to the use of calling context (stack introspection) in the Java security model. Fourth, the policies may be *delocalized*, in the sense that the enforcement point and enforcement mechanism may be unspecified. For example, if transcripts are stored in a DBMS, the above requirement might be enforced in the DBMS or an application that connects to the DBMS. With the latter approach, the system should be designed so that unauthorized users cannot circumvent that application and access the DB directly. This policy might also be enforced in part by the operating system (based on login permissions and file permissions on the relevant servers) and the network (blocking connections to the server from hosts on which unauthorized users have login permissions).

Each high-level policy is enforced by one or more security mechanisms in a system (perhaps involving DBMSs, middleware, operating systems, file systems, firewalls, etc.). Enforcement also depends on the system architecture, which affects the possible paths that requests can take through the system. We sometimes refer to the configurations of security mechanisms as *low-level policies*. Ensuring that the low-level policies, together with a given system architecture, correctly enforce given high-level policies is a challenging problem.

Since enforcement of the high-level policies that control access to an information resource might involve multiple hardware and software components in the system, a natural question during security analysis is to identify a *trusted computing base* (TCB) for each information resource. Note that the answer may depend on the low-level policies as well as the system architecture.

Security policies with one or more of the above "high-level" characteristics are natural during system design. The main contributions of this paper are (1) explicit identification of these characteristics of high-level policies, (2) a framework that allows *convenient* and *formal* specification of such high-level policies, modeling of low-level policies, and modeling of relevant aspects of system architecture, (3) a method for verifying that the low-level policies in a system correctly enforce ("implement") the high-level policies, and (4) an algorithm for computing a trusted computing base (TCB) for a component or information resource.

Although there is a sizable literature on formal specification and analysis of security policies, we are not aware of any previous work that explicitly deals with high-level policies with these characteristics. The interplay between system architecture and the policies has a significant impact on our framework. Frameworks for security policy specification and analysis generally ignore system architecture and

request context (in the sense described above), except for specialized frameworks for network (e.g., firewall) policy analysis. Although our framework is broad and flexible enough to model relevant aspects of network security and operating system security, our focus is on application-level security policies.

We are implementing a policy development environment based on our framework and plan to evaluate it on case studies based on a university and a financial institution. Important directions for future work are to consider policy administration and trust management.

2 Related Work

Coordination of Policies in Distributed Systems

Firmato [BMNW99] is a higher-level language for specifying firewall policies. Firmato policies get translated into rule-sets for different models of firewalls, insulating administrators from the details of each model's configuration language. In addition, given the network topology, each firewall's policy can be specialized to contain only the rules relevant to traffic that may pass through it. Work on Firmato does not consider verification of firewall policies against overall network security requirements or analysis of how firewall policies interact with security policies of other components.

García-Alfaro, Cuppens, and Cuppens-Boulahia [GACCB06] define and give algorithms to detect several specific kinds of anomalies (inconsistencies and potential errors) in network security configuration, specifically, configuration of firewalls and network intrusion detection systems (NIDS). In contrast, our work is aimed at verification of general application-level security requirements, taking network security configuration into account but in less detail. Thus, the kinds of properties verified, and the analysis algorithms used, are quite different.

Ioannidis *et al.* [IBI+07] propose the concept of *virtual private services* (VPSs) to describe a service implemented by a collection of components whose security policies must be configured in a coordinated way to enforce an access control policy associated with the service. They express all access control policies in the same language, namely KeyNote [BFIr99], without distinguishing "high-level" and "low-level" policies. A policy for a VPS can be delocalized—in particular, its enforcement might involve multiple components—but is otherwise basically a low-level policy, in our terminology. They describe a system architecture for deploying and enforcing policies. They do not consider formal analysis, verification, or refinement of policies.

Bandara, Lupu, Moffett, and Russo [BLMR04] propose a formal methodology for policy refinement, based on event calculus [BLR03]. Since most policies today are developed in *ad hoc* ways, not using a formal refinement methodology, we focus instead on verification of given low-level policies against given higher-level policies (requirements). Also, their framework is completely generic; in order to use it for

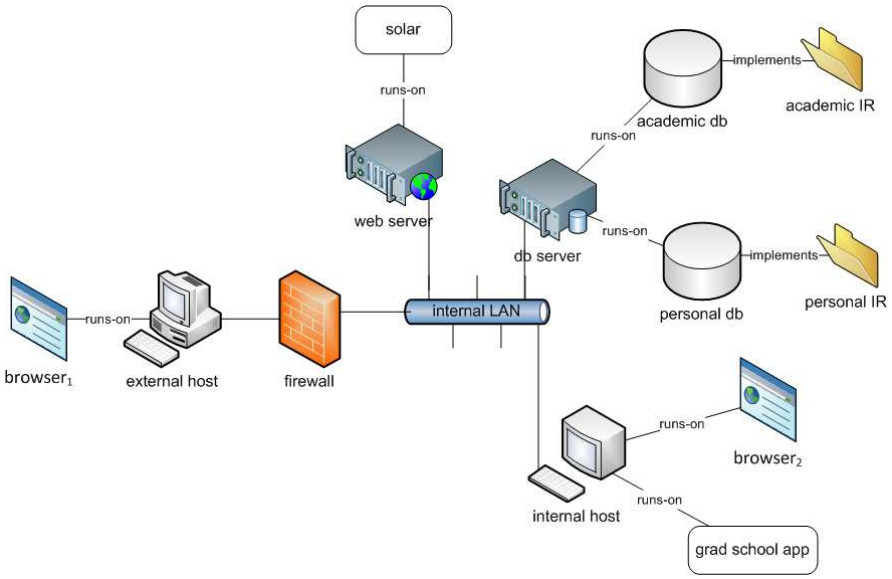

Fig. 1 Architecture of student information system. Edge labels specify the corresponding relation. The components connected on `internal LAN` are related to each other via `link` relation.

refinement of enterprise security policies, one would need to introduce relations and rules similar to those used in our framework to model system architecture and access control policies.

Sheyner, Haines, Jha, Lippmann, and Wing [SHJ+02] present a method to efficiently construct *attack graphs*, which represent attacks involving sequences of exploits of vulnerabilities in components of a system. Our work is largely complementary to attack graph analysis. Attack graphs are based primarily on vulnerabilities in components; access control policies and calling behavior are not considered, except when they affect a vulnerability. Also, attack graphs are generally used to find violations of system-level security requirements (e.g., who may login to a host), not application-level security policies.

3 Framework

Running Example. We use a student information system as a running example to illustrate our framework. Student information is classified as academic (transcript, etc.) or personal (SSN, citizenship, etc.). The system architecture is shown in Figure 1. Academic information and personal information are stored in separate databases. `solar` is a web-based university information system; for brevity, we model `solar` and the associated web server as a single component.

Information Resources. An *information resource*, abbreviated IR, represents a kind of information handled by the system. The relation `implements(`*C*`, `*I*`)` means that component *C* (partially or completely) implements IR *I*, i.e., *C* stores that kind of information. For example, the student information system contains two IRs, `academicIR` and `personalIR`, each implemented by a corresponding database (e.g., `implements (academicDB, academicIR)`). The distinction between an IR and the components that implement it is useful if the information in the IR is partitioned, replicated, archived, etc.

The information in an IR is assumed to be structured as a set of records, whose attributes (fields) and their types are specified in the definition of the IR. We refer to these as attributes of the IR, although they are actually attributes of the records in it. An attribute type can be a primitive data type (e.g., String) or an IR, denoting a reference to a record in another IR (recursive types are prohibited). For example, the attributes of `academicIR` and `studentIR` include an attribute `id` with type String, which identifies the student that the record is about. IRs have a straightforward API with operations for manipulating records. For example, the API includes an operation `readField` with arguments `record` (the record being accessed) and `field` (the field being accessed).

Components. A system is built from *components*, which may represent software (e.g., `solar`) or hardware (e.g., a host or firewall). Each component has attributes, accessed using the dot operator. For example, for a software component *C*, *C*`.host` is the host on which *C* runs. Attributes can also provide information about identity management, e.g., which authentication services and directory services are used by the component.

Each component has an API. For example, the API for the databases `academicDB` and `personalDB` is modeled (ignoring details of SQL) as containing functions like `readField`, `writeField`, `readRecord`, and `addRecord`. The API for `solar` contains `getTranscript`, `getSSN`, and `getCitizenship`. We model the browser as offering its user a single function, `request`, which nondeterministically sends some request to a web server (in this case, solar). For brevity, we consider only the above functions; other functions can be modeled and analyzed similarly.

Each component has a *low-level permit policy* that controls invocations of functions in the component's API and is enforced locally by the component. The language for low-level policies is described later in this section.

High-Level Policies. High-level policies are expressed in a simple rule-based language, which is an extension of Datalog with simple data structures that can be read, but not constructed or updated, by policy rules. A policy rule has the form *Q* `<-` P_1`, ...,`P_n and means: *Q* holds if P_1 through P_n hold. Variables start with an uppercase letter, constants start with a lowercase letter, and string constants appear in single quotes. The rules define the relation `hPermit` ("high-level permit"). `hPermit(`*U*`, `*R*`, `*Op*`, `*C*`)` holds if the system should permit (allow) requests from user *U* to perform operation *Op* on resource *R* in context *C*. A *resource* is a component or IR. The rules may also define auxiliary relations. For convenience, the name and arguments of the operation are modeled as attributes of *Op* (this is just a

```
% A Student can read any field in the records for himself or
% herself.
(P1) hPermit(User, Resource, Op, Context) <-
       Resource in {academicIR, personalIR},
       Op.function = readField, Op.record.id = User.id

% A Graduate School Clerk can read every student's transcript,
% if accessed through solar from (a browser running on) an
% internal host. Note: Context.head() is the first element of
% the context. internalHost(H) is an auxiliary predicate
% (definition elided) that holds if host H is part of the campus
% network.
(P2) hPermit(User, academicIR, Op, Context) <-
      Op.function = readField, Op.field = 'transcript',
      User.role = 'GradSchlClerk', Context.contains(solar),
      runs-on(Context.head(), H), internalHost(H)

% A registrar can read a student's personal information, if
% accessed from an internal host
(P3) hPermit(User, personalIR, Op, Context) <-
      Op.function = readRecord, User.role = 'Registrar',
      runs-on(Context.head(), H), internalHost(H)

% An administrative user can add new records to academicIR
(P4) hPermit(User, academicIR, Op, Context) <-
      Op.function = addRecord, User.role = 'admin'

% An administrative user can add new records to personalIR
(P5) hPermit(User, personalIR, Op, Context) <-
      Op.function = addRecord, User.role = 'admin'
```

Fig. 2 Illustrative high-level policy rules for the student information system

modeling convention, not an assumption about the implementation); the operation name is stored in Op`.function`. The context C is a sequence of tuples (c, f)—where c is a component or IR, and f is a function in c's API—representing the call chain (or "path") by which the request propagated through the system. Figure 2 shows some high-level policies for the running example.

Call Map. A function in a component's API may call functions provided by other components. Such calls must be considered to determine whether the restrictions on indirect calls expressed by high-level policies are enforced. We introduce a function `callMap` that captures the possible calls made by each component function. For simplicity and efficiency, `callMap` provides, and our analysis tracks, only equalities involving function arguments. Such equalities are often needed to verify enforcement of high-level policies; for example, to verify enforcement of (P1) in Figure 2, the analysis must track equalities involving the `id` argument, which identifies the user whose record is being accessed. `callMap` represents all interactions between components, regardless of the actual communication mechanism.

Given a component C and a function F in its API, `callMap`(C,F) returns a set of tuples of the form $(\mathit{calledBy}, R, F', \mathit{args})$, each describing a possible call made during execution of that function. The above tuple represents a call to function F' (the "target function") of the "target" resource (component or IR) R. *calledBy* is analogous to a setuid flag. If *calledBy*=`self`, the target resource sees the user executing the calling component C as the caller; if *calledBy*=`caller`, it sees the user that called F on C as the caller. *args* characterizes the possible arguments of the call to the target function. *args* is represented as a set of equalities of the form *attrib* = *val*, where *attrib* is an attribute name (recall that we model function arguments as attributes of an operation object), and *val* can be a constant, the name of an attribute (meaning that attribute *attrib* of the target call equals attribute *val* of the enclosing call to F), or `newVar` (meaning that a fresh variable will be used in the analysis to represent this value).

For example, `callMap(solar, getTranscript)` contains the tuple `(self, academicDB, readField, {id=id, field='transcript'})`. The values of `callMap` for `solar`'s `getSSN` and `getCitizenship` functions are similar. `callMap(browser`$_1$`, request)` contains a tuple for every function of every other component, with `newVar` arguments, reflecting that `browser`$_1$ is untrusted and may make arbitrary calls.

When analyzing the security of a design, the `callMap` for each component is based on the component's behavior as described in the design. For an implemented system, `callMap` could be determined from the code. Determining it accurately might be difficult, but an over-approximation can safely be used when verifying enforcement of high-level policies. Over-approximations in `callMap` may cause false alarms, but in many cases, the low-level permit policy of the target component or an intervening component will block the spurious calls or nested calls they make, preventing false alarms. If the analysis does raise false alarms, the corresponding call chains indicate exactly what assumptions about possible calls and their arguments are needed for enforcement of the high-level policies, and the `callMap`, permit policies, or system architecture can be refined accordingly.

Hosts and Firewalls. Each component has an attribute `type`. This attribute can have any value, but the values `host` and `firewall` have special significance. Hosts and firewalls are hardware components with network connections. Network connectivity is modeled by the relation `link(`C_1`, `C_2`)`, which means that the network may contain a path between C_1 and C_2 that does not pass through a host or firewall. This reflects the fact that we explicitly model hosts and firewalls but not routers. By taking all paths in the network topology into account in the `link` relation, we are making no assumptions about routing (or its security), although such assumptions could be used to restrict the `link` relation.

Hosts, like all components, have attributes, e.g., the set of users with accounts on the host. Since each software component must run on a host, we introduce a relation `runs-on(`C`, `H`)`, which means that component C may run on host H. Hosts provide various services, notably communication services, to components running on them. Host-based security mechanisms may limit the communication performed by a component, e.g., blocking connections with components on untrusted hosts.

Firewalls provide a similar security mechanism, typically forwarding some messages and dropping others, based on the firewall's local policy. An obvious way to capture this is to model network security mechanisms as they are implemented (e.g., at the packet level). However, this level of detail would unnecessarily complicate the model and slow the analysis. We adopt a higher-level view, in which hosts and firewalls are modeled as forwarding (or dropping) inter-component function calls, rather than packets. We include relevant network-layer information, such as the source and destination network addresses, as attributes of the operation object *Op* representing the call. With this approach, the API of a host or firewall includes the operations (of other components) that it forwards; its low-level permit policy allows calls that it forwards and denies calls that it drops; and its `callMap` normally indicates that the call gets forwarded with unchanged arguments.

Low-Level Policies. Low-level policies for all components are represented in a common rule-based language. The actual configuration languages of the access control mechanisms get translated to this common language; this can be automated. Low-level policy rules have the same form as high-level policy rules. They define auxiliary relations (if desired) and the relation `permit(`*U*`, `*R*`, `*Op*`, `*M*`)`, where the user *U*, resource *R*, and operation *Op* are the same as for `hPermit`, and *M* mode *M* describes the communication mechanism through which the operation is invoked. The mode *M* enables us to model the fact that different functions may be offered through different interfaces or with different policies. To avoid irrelevant details and distinctions about communication mechanisms, we define modes that reflect how the communication mechanism relates to the system architecture. A mode *M* has an attribute `type` whose possible values are: `direct`, indicating that the function is called by a user directly executing/running the component; `local`, indicating that the function is called via some inter-process communication mechanism by another component on the same host; or `remote`, indicating that the function is called over the network via some communication mechanism. The mode *M* may have additional attributes, depending on its type. If *M*`.type=local`, *M*`.requester` identifies the calling component. If *M*`.type=remote`, the attributes *M*`.srcIP`, *M*`.srcPort`, *M*`.destIP`, and *M*`.destPort` represent the source IP address, source port, destination IP address, and destination port, respectively.

We could express low-level policies in an existing language for attribute-based access control, such as OrBAC [ABB+03], which offers useful abstractions for structuring policies. Our language is simple but flexible and expressive: those abstractions can easily be represented in our language using auxiliary relations, and making them built-in would complicate our analysis algorithm without providing any additional leverage.

Figure 3 contains low-level policies for the student information system. `campusIPaddr(`*IPaddr*`)` is an auxiliary predicate that holds if the given IP address is part of the campus network.

```
firewall:
  permit(User, Resource, Op, Mode) <-
    Resource in {webServer, solar}, Mode.type = remote,
    Mode.destPort = 443

solar:
  permit(User, solar, Op, Mode) <-
    Op.function in {getTranscript, getSSN, getCitizenship},
    Op.recordId = User.id, Mode.type = remote
  permit(User, solar, Op, Mode) <-
    User.role = 'GradSchlClerk', Op.function = getTranscript,
    Mode.type = remote, campusIPaddr(Mode.srcIP)

webServer:
  permit(_, solar, _, _)

dbServer:
  permit(User, Resource, Op, Mode) <-
    Resource in {academicDB, personalDB}, Mode.type = remote,
    Mode.destPort = 8000

personalDB:
  permit(User, personalDB, Op, Mode) <-
    User.role = 'Registrar', Op.function = readRecord,
    Mode.type = remote, campusIPaddr(Mode.srcIP)
  permit(User, personalDB, Op, Mode) <-
    User.role = 'solar', Op.function = readField,
    Mode.type = remote
  permit(User, personalDB, Op, Mode) <-
    User.role = 'admin', Op.function = addRecord,
    Mode.type = direct

academicDB:
  permit(User, academicDB, Op, Mode) <-
    User.role = 'solar', Op.function = readField,
    Mode.type = remote
  permit(User, academicDB, Op, Mode) <-
    User.role = 'admin', Op.function = addRecord,
    Mode.type = direct
```

Fig. 3 Low-level policies for student information system

4 Verification of Enforcement

This section sketches an algorithm for verifying that the low-level policies and system architecture together enforce the high-level policies. For simplicity, the algorithm assumes that the policies does not contain recursion. This restriction is satisfied by most policies and can easily be relaxed if necessary.

The default starting points for requests are all functions s_f of all components s_r that can be directly invoked . At each starting point, the arguments to the (top-level)

function call and the identity of the user making the call are represented by variables. The algorithm computes all possible chains of functions call that can propagate from each starting point through the system, based on the system architecture and `callMap`. Note that these call chains, ignoring the arguments to each function, correspond to the "context" argument of `hPermit` in the high-level policy. If the call map contains cycles, the number of call chains may be infinite. If a possible call C would extend a call chain with a call that is the same, modulo renaming of variables introduced by `newVar`, as a call already in the call chain, then that call is not explored. To ensure this condition is sound, we include in the policy language only selected functions for accessing the context; currently, we include `head()` and `contains(`*expr*`)` (not, e.g., `length()`).

While constructing call chains, the algorithm accumulates constraints on the values of variables (the starting variables and variables introduced by `newVar`) that represent function arguments; the constraints express that the calls in the chain are permitted by the low-level policies of the components involved (including hosts and firewalls). Values of function arguments obtained from `callMap` are reflected in the formula as equality conjuncts; for example, if `callMap` indicates that a function call represented by `Op1` has `CS` as the value of the `dept` argument, `Op1.dept = CS` is conjoined to the formula. The constraint for a call is determined by matching the conclusions of the `permit` rules in the low-level policy of the component with the call, and, for each rule that matches, instantiating the variables in the rule based on the match and then backchaining to construct a first-order logic formula representing conditions under which the instantiated conclusion can be derived. Since we assume the policy rules are not recursive, the backchaining always terminates. If the accumulated constraint becomes unsatisfiable, the algorithm does not explore extensions of that call chain.

For each call chain S (including prefixes of longer call chains), the algorithm checks whether the call chain is consistent with the high-level policy. Specifically, let Ψ_L be the constraint computed for S, and let C be the context defined by S, i.e., $S[i]$ is a call to function $\mathit{first}(C[i])$ of component $\mathit{second}(C[i])$, where *first* and *second* return the indicated components of a tuple. Call chain S is consistent with the high-level policy if, for every instantiation of the variables that satisfies Ψ_L (in other words, S is feasible), the instantiated call $\mathit{last}(S)$ with context C is permitted by the high-level policy. To check this efficiently, we use backchaining to compute a first-order logic formula Ψ_H representing the conditions (including conditions on the context) under which the call $S[n]$ is permitted by the high-level policy, using a variable V to represent the call's context, and then we check whether the formula $(V = C) \wedge \Psi_L \wedge \neg\Psi_H$ is satisfiable. The satisfiability of this formula implies an inconsistency in the system. Our current prototype uses Yices (`http://yices.csl.sri.com/`) for this purpose. If the satisfiability check succeeds, the logic tool can provide an instantiation of the variables for which the formula is true; this instantiation of S is a counterexample that illustrates how the high-level policy can be violated.

The following example illustrates how our analysis works and how it can identify vulnerabilities. For this example, we modify the low-level policies in Figure 3 as

follows: the rule for `GradSchlClerk` in `solar`'s low-level policy is removed and replaced with the following rule in the low-level policy for `academicDB`:

```
permit(User, academicDB, Op, Mode) <-
   User.role = 'GradSchlClerk', Op.function = readField,
   Op.field = 'transcript', Mode.type = remote,
   campusIPaddr(Mode.srcIP)
```

Consider a call chain that propagates along the following path (i.e., context) `C0`: `[(browser`$_2$`, request), (internalHost, request), (dbServer, readField), (academicDB, readField)]`. The constraint associated with S is (note: when it is necessary to rename a variable in a rule during backchaining, in order to avoid name collisions, the algorithm appends the name of the component that the rule is for and/or a sequence number; variables characterizing the top-level call, such as `User` and `Op` in the formula below, never get renamed):

Ψ_L : `Mode_academicDB.type = remote` $\wedge$ `Mode_academicDB.destPort = 8000` $\wedge$ `Op.function = readField` $\wedge$ `Op.field = 'transcript'` $\wedge$ `User.role = 'GradSchlClerk'` $\wedge$ `campusIPaddr(Mode_academicDB.srcIP)`

The last call in this chain is to function `readField` of component `academicDB`, which implements `academicIR`. The following constraint is computed for this function call from the high-level policy:

Ψ_H : `Op.function = readField` $\wedge$ `Op.field = 'transcript'` $\wedge$ `User.role = 'GradSchlClerk'` $\wedge$ `Context.contains(solar)` $\wedge$ `runs-on(Context.head(), H)` $\wedge$ `internalHost(H)`

The formula $(\texttt{Context} = \texttt{C0} \wedge \Psi_L) \wedge \neg\Psi_H$ is satisfiable; note that the conjunct `Context.contains(solar)` in Ψ_H is not satisfied when $\texttt{Context} = \texttt{C0}$. This shows that the modified low-level policy does not enforce the high-level policy. The significance of this violation depends on why the high-level policy requires that `solar` be in the context for these accesses. For example, `solar` might be responsible for logging accesses to student transcripts by grad school clerks, for compliance with student privacy regulations. Such an error might not be noticed during system execution, while our analysis exposes it during the design stage.

5 Trusted Computing Base

In general, a *trusted computing base* (TCB) consists of the hardware and software responsible for enforcing a security policy. We define a set T of components to be a TCB for resource (component or IR) r in system S (a system is defined by sets of components and IRs, with their attributes; `links`, `runs-on`, and `implements` relations; and low-level policies for each component) with high-level policy H if "correct" behavior by the components in T (i.e., behavior consistent with their low-level policy and `callMap`) is sufficient to ensure that all call chains that end at r are

consistent with H. Recall that consistency of a call chain with a high-level policy is defined at the end of Section 4.

More formally, to check whether T is a TCB for enforcement of the high-level policy for r in system S with high-level policy H, we construct a variant $\texttt{relax}(S, \overline{T})$ of the system, where $\overline{T}$ (the complement of T) is the set of components of S not in T, and then use the method described in Section 4 to check whether call chains in that system that end at r are consistent with H. The variant $\texttt{relax}(S, \overline{T})$ is the same as system S except that, for every component C in $\overline{T}$, the low-level permit policy of C is replaced with the single rule `permit(User, Resource, Op, Mode) <- true`, and for every function F in C's API, $\texttt{callMap}(C, F)$ returns the set containing all tuples of the form $(\mathit{calledBy}, R', F', \mathit{args})$ such that $\mathit{calledBy} \in \{\texttt{self}, \texttt{caller}\}$, R' is a component or IR of S other than C, F' is a function in the API of R', and *args* maps all parameters of F' to `newVar`.

Designers might want to specify conditions on the acceptable TCB for a resource—for example, that the TCB for a resource contains only components with specified administrators. Our TCB analysis provides a basis for checking such properties.

References

[ABB+03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[BFIr99] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Ke romytis. The KeyNote trust management system, version 2, IETF RFC 2704, September 1999.

[BLMR04] Arosha K. Bandara, Emil Lupu, Jonathan D. Moffett, and Alessandra Russo. A goal-based approach to policy refinement. In *5th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 229–239, 2004.

[BLR03] Arosha K. Bandara, Emil C. Lupu, and Alessandra Russo. Using event calculus to formalise policy specification and analysis. In *Proc. 4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, 2003.

[BMNW99] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[GACCB06] Joaquín García-Alfaro, Frédéric Cuppens, and Nora Cuppens-Boulahia. Analysis of policy anomalies on distributed network security setups. In *Proc. 11th European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *Lecture Notes in Computer Science*, pages 496–511. Springer, September 2006.

[IBI+07] Sotiris Ioannidis, Steven M. Bellovin, John Ioannidis, Angelos D. Keromytis, Kostas G. Anagnostakis, and Jonathan M. Smith. Virtual private services: Coordinated policy enforcement for distributed applications. *International Journal of Network Security*, 4(1):69–80, January 2007.

[SHJ+02] Oleg Sheyner, Joshua W. Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284, 2002.