Fail-Secure Access Control

Petar Tsankov
Institute of Information Security, ETH Zurich
ptsinkov@inf.ethz.ch

Srdjan Marinovic
Institute of Information Security, ETH Zurich
srdanm@inf.ethz.ch

Mohammad Torabi Dashti
Institute of Information Security, ETH Zurich
torabidm@inf.ethz.ch

David Basin
Institute of Information Security, ETH Zurich
basin@inf.ethz.ch

ABSTRACT

Decentralized and distributed access control systems are subject to communication and component failures. These can affect access decisions in surprising and unintended ways, resulting in insecure systems. Existing analysis frameworks however ignore the influence of failure handling in decision making. Thus, it is currently all but impossible to derive security guarantees for systems that may fail. To address this, we present (1) a model in which the attacker can explicitly induce failures, (2) failure-handling idioms, and (3) a method and an associated tool for verifying fail-security requirements, which describe how access control systems should handle failures. To illustrate these contributions, we analyze the consequences of failure handling in the XACML 3 standard and other domains, revealing security flaws.

Categories and Subject Descriptors
K.6.5 [Management of Computing and Information Systems]: Security and Protection; D.4.6 [Operating Systems]: Security and Protection—access controls

General Terms
Security, Verification

Keywords
Access Control; Failure Handling; Formal Analysis

1. INTRODUCTION

Modern access control systems are often decentralized and distributed, and therefore subject to communication and component failures. If failures affect the availability of information needed for security decisions, then access control systems must, either implicitly or explicitly, handle these failures. This concern permeates all access control domains. For example, firewalls must operate even when their log engines crash [26] or rule updates fail [28], web applications must service requests even if authentication services are unresponsive [27], delegation systems must evaluate requests even when they cannot update their revocation lists, and perimeter security systems must control access even when the wireless channels to their central database are jammed.

In such settings, the access decisions of a Policy Decision Point (PDP) cannot be understood without considering the PDP’s failure handlers as well.

The access control community has not thus far rigorously studied the effects of failure handlers on access decisions. One reason for this is that simply interpreting failures as denies appears sufficient to conservatively approximate the PDP’s desired behavior. This would suggest that the policy writer need not overly concern himself with analyzing the PDP’s failure handlers. However, failures can affect the PDP’s decisions in surprising and unintended ways. Such simplistic approximations are not only inflexible, they also do not necessarily result in secure systems. As an example, we describe later how the conservative approach of replacing failures with denies had been originally adopted in the XACML 3 standard, and was later dropped due to its insecurity.

Given that failure handling influences the PDP’s access decisions, it follows that formal analysis frameworks for access control should account for the PDP’s failure handlers. Only then can security guarantees be derived for the PDP’s access decisions, both in the presence and absence of failures. Analysis techniques for obtaining such security guarantees would be of immediate practical value because existing access control systems separate failure handling from the “normal” (typically declarative) policy interpreted by the PDP, i.e. the policy that defines the PDP’s decisions when no failures occur. The logic that decides access requests is therefore split into two parts. This separation makes the PDP’s behavior difficult to understand and analyze.

Existing formal analysis frameworks for access control policies are inadequate for the task at hand. This is neither an issue with the expressiveness of their formal languages nor the complexity of their decision problems. Rather, they lack (1) a system and attacker model tailored for failure scenarios, (2) idioms for specifying failure handlers, and (3) methods for verifying fail-security requirements, i.e. security requirements that describe how distributed access control systems ought to handle failures. Thus, currently it is all but impossible to derive security guarantees that extend beyond the PDP’s normal behaviors. In this paper, we show how to realize these three artifacts using the BeLog analysis framework [32].
Contributions. This is the first paper that systematically analyzes the role of failure handling in access control systems. We investigate three kinds of security flaws: failure-oblivious policy composition, overly eager failure handling, and the preemptive masking of failures. Examples of systems that exhibit these flaws are given in the following sections; a common thread in these systems is their seeming conformance to security common sense.

We also demonstrate how the PDP, including its failure handlers, can be modeled and analyzed using the BelLog policy analysis framework. In particular: (1) We investigate seven real-world access control systems and use these to extract a system and an attacker model tailored for analyzing the effect of failures on the PDP’s decisions. (2) We derive common failure-handling idioms from these systems, which can be readily encoded in BelLog. (3) Through examples, we show how to express fail-security requirements and we provide a tool to automatically verify them for a given PDP with respect to our attacker model. We argue that our verification method is effective by demonstrating how the three kinds of security flaws mentioned above can be discovered.

As a final remark, we choose BelLog for technical convenience. BelLog is a four-valued extension of Datalog (the core of most decentralized access control languages), where one of the truth values, borrowed from Belnap’s logic, can be used to denote failures; see §4 for details. Our contributions are however independent of the BelLog formalism. Any sufficiently expressive logic, for example first-order logic, can replace BelLog for our purpose.

Related Work. Although fail-security requirements have been discussed in the security literature [7, 33], there has been no rigorous, systematic treatment of fail-secure access control. The existing access control specification languages, such as [3, 9, 13, 16, 18, 25], do not explicitly deal with failure handlers in their analysis. Although failures are considered in [12], failure-handling mechanisms are not dealt with.

Static and dynamic policy analysis frameworks such as [2, 10, 14, 15, 17, 23] can potentially be tailored to reason about PDPs with failure handling, similarly to BelLog. In particular, PBel’s analysis framework [10] also supports policies with many-valued policy decisions and can, if delegations are excluded, express our failure-handling idioms. We remark that dynamic analysis frameworks, such as [2, 14, 17], consider history-based access decisions, which fall outside the scope of our paper.

Organization. In §2, we give examples of PDP failure handlers and fail-security requirements for access control systems. In §3, we define our system and attacker model. In §4, we summarize the BelLog specification language and use it to specify the examples from §2. In §5, we analyze these examples with respect to their fail-security requirements. We point to future research directions in §6.

2. MOTIVATION

As motivation, we use the XACML 3 standard to show that approximating failures with denials, although seemingly conservative, can lead to insecure systems. Through our second and third examples, taken from the web application and grid computing domains, we illustrate the common PDP implementation pattern that treats failure handlers as a separate add-on to the normal policy engine. We show how this separation makes understanding and analyzing PDPs particularly difficult, resulting in systems open to attacks.

XACML 3. XACML 3 is an OASIS standard for specifying access control policies [36]. XACML 3 policies are issued by principals and evaluated by a PDP. A policy issued by the PDP’s administrator is called trusted; otherwise, it is non-trusted. The administrator specifies whether a non-trusted policy is authorized to decide a given request. XACML 3 policies are grouped into policy sets and their decisions are combined with composition operators, such as permit-overrides, which grants access if at least one policy grants access. To decide a given request, the PDP first computes the decisions of all policies in the set. Afterwards, it checks which non-trusted policies are authorized by the administrator. Finally, the PDP combines the decisions of the trusted policies and the authorized non-trusted policies using the policy set’s composition operator.

An XACML 3 PDP obtains all information needed for policy evaluations, such as attributes and credentials, from Policy Information Points (PIPs). The XACML standard, up to Revision 16, stated that the PDP should refrain from using policies that could not be evaluated or authorized due to communication and PIP failures. This decision follows the intuitive idea that all suspicious policies should be excluded from the PDP’s decision. Figure 1 specifies such a PDP, including its failure handler, in pseudo-code. Although this failure handler is inflexible, the committee did not anticipate other consequences on the PDP’s decisions apart from always making them more conservative (less permissive). This however turned out to be wrong.

When the proposed failure-handling behavior was considered together with the deny-overrides composition operator, the following attack was discovered [37]. Consider a request $r$ and a policy set $P$ that contains one trusted policy $P_t$ that grants $r$ and one authorized non-trusted policy $P_a$ that denies $r$. $P$’s decisions are combined with deny-overrides. If the PDP successfully evaluates $P_t$ and fails to evaluate $P_a$, then the PDP will grant $r$, even though it does not have all the necessary information to make this decision. In this case, the attacker can simply launch denial-of-service attacks against PIPs and obtain a grant decision for $r$. In §5 we show how this attack can be found through automated analysis.

This example illustrates that a PDP’s failure handlers, regardless of their simplicity, can affect access decisions in surprising ways. In this example, the failure-oblivious composition of sub-policies is the root of the security flaw. To remedy this flaw, the XACML 3 standard currently uses
To conclude, because existing web access control frameworks typically separate failure handling from the normal policy of the PDP, it is difficult to gain confidence in their security. To rise to this challenge, policy specification languages and their analysis frameworks should also account for the interactions that result from the separation. In §4, we give a formal specification of the method of Figure 2, and we verify the specification against FR1 in §5, which reveals the discussed problem.

**Authorizations in Grids.** In grid computing platforms, resources (such as storage space) are located in different domains. Each domain has an owner, and only one PDP controls access to the domain’s resources. It is however infeasible for each PDP to manage authorizations for all subjects from all domains. Domain owners therefore delegate authorization management to trusted subjects, possibly from other domains. These subjects may then issue tokens to authorize other subjects and to further delegate their rights. All tokens are stored as digital credentials. Subjects then submit their credentials, alongside their access requests, to a PDP. In addition, it is sometimes necessary to revoke subject’s credentials, for example when dealing with

```java
isAuthorized(User u, Object o, List aclIDs)
try
   for (id in aclIDs)
      if (readAcl(id).grants(u, o)) return true
   catch (ReadAclException e)
      return def.grants(u, o) and logger.on()
return false

Figure 2. A PDP module for the web app example.
```

```java
def.grants(u, o)
return
for aclIDs
   if (readAcl(id).grants(u, o))
      return true
   catch (ReadAclException e)
      return def.grants(u, o) and logger.on()
return false

Figure 3. The figure shows which subjects in the depicted scenario have access according to FR2. Ann has access because her delegation is issued by the owner. Bob and Dave have access because they have non-revoked chains. Fred and Carol are denied access because they do not have non-revoked chains and they are not the owner’s direct delegates.

and overloads a designated policy decision (the indeterminate IN) for every policy that cannot be evaluated due to failures. Consequently, failure handling is now a concern of the policy writer.

**Authorizations in Web Apps.** Web applications use access control frameworks to specify and manage user permissions. Examples include the Java Authentication and Authorization Service JAAS, Apache Shiro, and Spring Security. Basic policies can be specified using declarative policy languages. A PDP module loads policy specifications and evaluates them within its authorization method. A reoccurring problem is that the PDP module fails to load a declarative specification due to syntactic errors or missing files. To deal with this problem, administrators often maintain a default specification that serves as a fallback option. Use of the default specification is typically conditioned on whether logging is enabled. This fallback approach imposes the following fail-security requirement:

*Fail-security requirement 1 (FR1): When the PDP module cannot compute an access decision due to malformed or missing policy specifications, then it uses the default specification if logging is enabled, and it denies access otherwise.*

To illustrate this, consider the case where the PDP module composes finitely many access control lists (ACLs) using the permit-overrides operator, which permits access if at least one of the ACLs permits access, and denies access otherwise. To adhere to FR1, the PDP module must invoke the failure handler if and only if none of the ACLs permits access and at least one of them is malformed. The failure handler in this example would evaluate the default ACL def and check whether logging is enabled. Figure 2 gives a straightforward authorization method for this scenario in pseudo-code. The method takes as input a user object u, the requested object o, and a list aclIDs of ACL identifiers. The method readAcl(id) returns the ACL object corresponding to id, and throws a ReadAclException exception when it cannot find or parse the associated ACL. The default ACL def is hard-coded in the method.

The pseudo-code describes a correct permit-overrides operator for ACLs under normal conditions, i.e. when there are no failures. The catch block is also correct as it intuitively follows the structure of FR1. However, the failure handling is overly eager in that if a ReadAclException is thrown while evaluating an input ACL then the PDP stops evaluating the remaining input ACLs and jumps to the catch block. This method therefore does not satisfy FR1: if a list of two ACL identifiers is passed to the method and the first ACL fails to load, then the method immediately consults def, which would be wrong if the second ACL would permit access.

This problem is rooted in the overly eager invocation of the failure handler. The problem here is not an instance of syntactic vulnerability patterns, such as overly-broad throws declaration and overly-broad catch block [20], and it cannot be solved for example by simply moving the try-catch construct inside the for loop. One solution would be to delay the invocation of the failure handler until all the ACLs have been evaluated.

To conclude, because existing web access control frameworks typically separate failure handling from the normal policy of the PDP, it is difficult to gain confidence in their security. To rise to this challenge, policy specification languages and their analysis frameworks should also account for the interactions that result from the separation. In §4, we give a formal specification of the method of Figure 2, and we verify the specification against FR1 in §5, which reveals the discussed problem.
ex-employees. A common solution is to store all revoked credentials on a central revocation server.

A (delegation) chain for a subject $S$ is a transitive delegation from the domain owner to $S$. We say that a delegation chain is non-revoked if none of the delegations in the chain has been revoked. A given domain’s PDP grants access if the subject has at least one non-revoked delegation chain or the subject is the domain’s owner. The revocation server may sometimes be unavailable, for example due to lost network connectivity. Denying all access in the case of failures may be too restrictive as the unavailability of some resources, to selected subjects, would be too costly [33]. One fail-security requirement that reflects this notion is:

Fail-security requirement 2 (FR2): When the PDP cannot check whether a subject has at least one non-revoked delegation chain due to failures, the PDP grants access if the subject is a direct delegate of the owner; otherwise it denies access.

The rationale is that the owner rarely revokes his direct delegates. This requirement also states that the owner chooses to ignore all delegations issued by subjects, including his direct delegates, whose delegation chains cannot be checked. Figure 3 illustrates one delegation scenario and shows which subjects are granted access according to FR2.

Existing delegation languages do not specify failure handling within policy specifications, but rely on having failure handlers within the PDP. This approach, which separates the delegation logic from failure handling, is described in [8]. Based on these guidelines, Figure 4 depicts a possible PDP design for our grid access control scenario. In Figure 4a, we specify the normal policy of the PDP in Datalog, the core of many delegation access control languages (see e.g. [3,13,16,25]). The policy grants access to a subject $X$ if $X$ is an owner or has a (transitive) delegation chain from an owner. Before evaluating the Datalog policy, the PDP checks whether each supplied delegation is still valid by querying the revocation server. If it is revoked then the PDP discards the delegation.

Considered separately, the Datalog normal policy and the failure handler of Figure 4 intuitively conform to FR2. Their interaction however leads to a subtle attack. The attack, described in §5, results from the preemptive masking of failures. We were unable to find the attack before specifying this PDP in BelLog; we believe this applies to most policy writers.

Finally, we remark that our goal in this paper is not to promote particular fail-security requirements; they can be determined for example from a risk analysis of each deployed system. Our goal is instead to raise and address the need for analyzing access control systems in the presence of (malicious) failures with respect to their security requirements. We stress that even systems that are intended to conform to simple conservative requirements, such as the fail-safe principle (deny all access if there is any failure) [29], are not exempt from failure-handling flaws, and thus should also be analyzed.

3. SYSTEM AND ADVERSARIAL MODEL

We consider distributed access control systems where a policy decision point (PDP) communicates with multiple policy enforcement points (PEP), and multiple policy information points (PIP); see Figure 5. A subject submits requests and credentials to a PEP, which forwards them to the PDP. A credential maps a subject or a resource to an attribute (such as a role). Credentials are issued by, and exchanged between, subjects; they can also be stored locally at the PDP. We assume that the PDP verifies their authenticity, for example using digital signatures. The PDP sends remote queries to PIPs to obtain information relevant for making access decisions, for example information concerning revoked subjects and the current time. Remote queries can be implemented through inter-process communication mechanisms such as files, network sockets, and shared memory. The PDP software, either implicitly or explicitly, recognizes and handles communication failures.

The PDP interprets a normal access control policy, which maps access requests, credentials, and information to access decisions; the normal policy does not specify failure handling. The policy is defined by policy rules, which are issued by the subjects and given to the PDP for evaluation. The PDP has one designated subject, the administrator, who has the authority over all access requests and his policy rules are always evaluated. The PDP takes other rules into account only if the administrator has delegated to their issuers, either directly or transitively, authority over the given request. All access decisions made by the PDP are forwarded to, and enforced by, the PEPs.

In our model, we assume that the PDP and the PEPs do not fail, whereas PIPs can fail. We also assume that the communication channels between the PDP and the PIPs can fail, while all other channels (e.g. PEP-to-PDP) are reliable. We assume that communication delays are bounded and failures are determined either by timeouts or by receiving corrupted messages. After the PDP sends a remote query $q$ to a PIP, it therefore receives one of two responses: (1) the answer to $q$; or (2) error, indicating a communication failure.
Table 1. Analyzed access control systems and their failure-handling idioms.

<table>
<thead>
<tr>
<th>System</th>
<th>Failure-handling Idioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>KABA KES-2200 [21]</td>
<td>Catch</td>
</tr>
<tr>
<td>Kerberos [22]</td>
<td>Fallback</td>
</tr>
<tr>
<td>RedHat Firewall [28]</td>
<td>Catch</td>
</tr>
<tr>
<td>WebSphere [35]</td>
<td>Catch</td>
</tr>
<tr>
<td>XACML PDPs [38]</td>
<td>Propagate, Catch</td>
</tr>
</tbody>
</table>

4. SPECIFYING ACCESS CONTROL WITH FAILURE HANDLING

In this section, we first describe three failure-handling idioms, derived by analyzing seven existing access control systems and their failure handlers. These idioms are abstractions we use for modeling failure-handling mechanisms. We then give an overview of the specification language BelLog, and show how it can be used to specify the failure-handling idioms and the PDPs of §2, including their failure handlers.

4.1 Failure-handling Idioms

To understand how existing systems handle communication failures, we have inspected the documentation of seven access control systems; see Table 1. Our analysis revealed three failure-handling idioms, which are sufficient to describe how failures are handled in these systems. To describe the idioms, we abstract a PDP as evaluating a request through a finite sequence of computation and communication steps; hereafter referred to as events. We assume that computation events always terminate successfully, while communication events either terminate successfully or fail. Note that similar abstractions exist for exception handling in programming languages [19, 24].

**Fallback.** The fallback idiom abstracts the failure handlers that use fallback information sources when the communication channels to the primary information sources fail. If a communication event fails then it is re-executed using the fallback source. The fallback source can be, for example, a backup of a primary information source. This idiom is used in access control systems whose primary authentication services are unreliable. For example, Kerberos [22] can fall back on local user/password lists when its primary LDAP authentication service is unavailable.

To instantiate this idiom, a fallback source must be configured for each information source that may fail. Although the fallback source may be periodically synchronized with the information source, it may nevertheless provide stale information of inferior quality.

**Catch.** This idiom abstracts the failure handlers that catch failures and then enforce alternative access control policies. The catch idiom is analogous to exception handling in programming languages where the failure to execute a given procedure is handled by a designated procedure. In terms of the PDP’s execution, whenever an event fails, the execution branches to another (alternative) sequence of events.

We can use this idiom to implement a system that meets FR2. The system’s alternative access control policy would contain only the grants for the owners’ direct delegates. Systems that employ this idiom include: KABA KES-2200 [21], which is a token-based physical access control system that upon power failures is configured to either grant or deny all requests; IBM WebSphere [35], whose exception handlers evaluate designated error-override policies; and Cisco IOS [11] and RedHat Firewall [28], which in case of failures use alternative rule sets.

**Propagate.** Both the fallback and the catch idioms handle failed events immediately upon failure. In contrast, FR1 requires failures to be handled after all the ACLs have been evaluated. The propagate idiom abstracts the mechanisms for meeting requirements with such “delayed” failure handling. Whenever an event fails, the PDP pushes a designated error value as the input to all subsequent events.

For example, to meet FR1, ideally an error value that indicates a failure to evaluate an ACL is propagated. The default ACL is evaluated iff no ACL grants a given request and the PDP failed to evaluate at least one ACL. Note that the failure handler of Figure 2 implementing FR1 is, however, an instance of the catch idiom. Systems that employ the propagate idiom include XACML PDPs [38], which propagate indeterminate policy decisions, and Spring-based applications [30], which propagate data access exceptions.

4.2 BelLog Language

We use the BelLog language to specify the PDP’s behavior including its failure handlers. We favor BelLog because (1) we can use its truth value ⊥ to explicitly denote failures, (2) we can syntactically extend its core language to define common failure handlers, and (3) it can encode state-of-the-art decentralized policy languages [3, 13, 25]. We give a brief introduction to BelLog in the following; see [32] for further details.

BelLog is an extension of stratified Datalog [1], where the truth values come from Belnap’s four-valued logic [6]. BelLog’s syntax is given in Figure 6a. We use F, I, and ⊥ to denote finitely many rules, literals, and terms separated by commas. A BelLog program is a finite set of rules. Each rule has a head consisting of an atom and a body consisting of a list of literals. An atom is a predicate symbol together with a list of constants and variables. A literal is a, ¬a, or ¬a, where a is an atom, and ¬ and ¬ are the truth- and knowledge-negation operators; see Figure 6b. We write constants with sans font, and predicate symbols with italic font.
We use the capital letters \( P, R, \) and \( S \) to denote BelLog programs, and the remaining capital letters for variables.

BelLog’s truth space is the lattice \((D, \preceq, \land, \lor)\); see Figure 6b. The non-classical truth values \( \bot, \top \) and \( \land \) denote undefined and conflict. In §4.3, we show how \( \land \) is used to denote information missing due to failures. BelLog’s semantics is given in Figure 6c. Given a finite set of constants \( \Sigma \), called the domain, \( P^1 \) denotes the program obtained by replacing all variables in \( P \) in all possible ways using \( \Sigma \), and \( A \) is the set of ground atoms constructed from \( \Sigma \) without variables. \( P^1 \)’s model is an element of the lattice \((I, \subseteq, \cap, \cup)\), where \( I \) is the set of all interpretations. An interpretation \( I \in \Sigma \) maps ground atoms to truth values. The symbols \( \sim \) and \( \sim \) are overloaded over interpretations in the standard way, and \( I(1) \equiv I(1) \land \cdots \land I(1) \). A rule \( a \leftarrow I \) assigns the truth value of \( I(a) \) to \( a \). The consequence operator \( T_P \) applies \( P^1 \)’s rules and joins the results with \( \lor \) when multiple rules have the same head. Program \( P^1 \)’s model is constructed by first partitioning \( P^1 \)’s rules into strata \( P_1, \ldots, P_n \) and then computing, for each stratum \( P_i \), the join of (1) the previous stratum’s model \( M_{i-1} \) and (2) the meet of all fixed points of \( T_P \), \( f P_i \) that are greater than or equal to \( M_{i-1} \). A partitioning \( P_1, \ldots, P_n \) is a stratification if for each stratum \( P_i \), any predicate symbol that appears in a negative literal \( \neg P_i(a) \) does not appear in the head of \( P_i \cup \cdots \cup P_n \).

Finally, the \textit{input} to a BelLog program \( P \) is a set \( I \) of rules of the form \( p(I) \leftarrow v \), where \( v \in D \) and \( p \) does not appear in the heads of \( P \)’s rules. We write \( P \mid I \) for \( [P \cup I] \). For a given input \( I \), a program \( P \) \textit{entails} the ground atom \( a \), denoted \( P \models I a \), iff \( [P \mid I](a) = \top \).

We now explain how a PDP, i.e. its normal policy and its failure handlers, can be specified in BelLog. We illustrate this by specifying the examples from §2.

### 4.3 Specifying PDPs in BelLog

As described in our system model given in §3, a PDP’s behavior is determined by three elements: (1) the PDP inputs, namely credentials forwarded by a PEP and information obtained from PIPs, (2) the (normal) access control policy evaluated by the PDP, and (3) the failure-handling procedures used when the communication channels between the PDP and PIPs fail. In the following, we describe how these elements can be specified in BelLog.

#### 4.3.1 Specification Preliminaries

As described in our system model given in §3, a PDP’s behavior is determined by three elements: (1) the PDP inputs, namely credentials forwarded by a PEP and information obtained from PIPs, (2) the (normal) access control policy evaluated by the PDP, and (3) the failure-handling procedures used when the communication channels between the PDP and PIPs fail. In the following, we describe how these elements can be specified in BelLog.

**Inputs.** We represent credentials as atoms whose first argument represents the issuing principal’s identifier. For example, \texttt{public(ann, file)} is interpreted as “\textit{ann} asserts that file is public”. Hereafter, we write \texttt{ann::public(file)} to emphasize a credential’s issuer. For brevity, we omit writing “\textit{admin}::” to denote \textit{admin}’s credentials.

We model the information obtained from PIPs as \textit{remote queries}, which check whether a specified credential is stored at a designated PIP. We write remote queries as \texttt{ann::public(file)@pip}, where \texttt{ann::public(file)} is a credential and \texttt{pip} is a PIP identifier. Formally, remote queries are represented as atoms where the PIP identifier is appended to the predicate symbol; for example, \texttt{ann::public(file)@pip} is represented with the atom \texttt{public_pip(ann, file)}.

The PDP’s input consists of credentials forwarded by a PEP and credentials obtained using remote queries to PIPs. We model a PDP’s input as BelLog input. Given a BelLog input \texttt{I} and a credential \texttt{cred}, the truth value \( I(\texttt{cred}) \) is: \( t \) if \texttt{cred} is a credential forwarded by the PEP, and \( f \) if \texttt{cred} is not forwarded by the PEP. For a remote query \texttt{cred@pip}, \( I(\texttt{cred@pip}) \) is: \( t \) if \texttt{cred} is stored at \texttt{pip}, \( f \) if \texttt{cred} is not stored at \texttt{pip}, and \( \bot \) if a failure prevents the PDP from obtaining \texttt{cred} from \texttt{pip}.

#### Access Control Policies

We specify the PDP’s access control policy using BelLog rules. Note that state-of-the-art decentralized access control languages such as SecPAL [8], RT [25], Binder [13], and DKAL [18], all have translations to Datalog. Any policy written in these languages can therefore be encoded in BelLog, since BelLog extends Datalog. Furthermore, algebraic policy languages, such as XACML [38] and PBel [10], can also be encoded in BelLog; see [32].

**Failure Handling.** We define failure-handling operators as syntactic sugar in BelLog. We use a syntactic extension of BelLog that allows for nesting and combining rule bodies with the operators \( \sim \) and \( \land \). For example, the rule \( r \leftarrow \neg(p \land \neg q) \), where \( p \) and \( q \) are rule bodies, assigns to \( r \) the

\[
\begin{align*}
p \lor q & := \neg(p \land \neg q) \\
p = f & := \neg(p \lor \neg p) \\
p = t & := p \land \neg p \\
p \land q & := ((c = t) \land p) \\
& \lor ((c \neq t) \land q) \\
\end{align*}
\]

\[
\begin{align*}
p \lor \bot & := \neg(p = v) \\
p = \bot & := (p \neq f) \land (p \neq t) \\
\land (p \lor \bot) & = t \\
p \lor q & := q \land (p = v) \lor p \\
\end{align*}
\]
The truth value computed by applying the operators ¬ and ∧ to the truth values computed for p and q. Additional BelLog operators, such as the if-then-else operator (\( _\land \rightarrow _\bot \)) and the v-override operator (\( _\bot \rightarrow _\bot \)), are defined in Figure 7.

We define the error-override operator as

\[ p \triangleright q := p \nrightarrow q, \]

where p and q are rule bodies. The construct p \( \triangleright \) q evaluates to q’s truth value if p’s truth value is \( \bot \); otherwise, the result of p is taken. Using this operator, we can model the failure-handling idioms given in §4.1. Consider the remote query cred@pip, which checks whether the credential cred is stored at pip. To instantiate the fallback idiom, where fallback is the fallback PIP’s identifier, we write cred@pip \( \triangleright \) cred@fallback.

To illustrate the catch idiom’s specification, consider a PDP with the following two policies.

\[
\begin{align*}
\text{pol}_1(X) &\leftarrow \text{cmp}(X)@\text{db} \quad \text{(Policy P1)} \\
\text{pol}_2(X) &\leftarrow \text{stud}(X) \quad \text{(Policy P2)}
\end{align*}
\]

Here the atom pol \(_i\)(X) denotes policy \(_i\)’s decision. The communication between the PDP and the PIP db can fail. Imagine that the PDP instantiates the catch idiom and uses \(_P2\) whenever it cannot evaluate \(_P1\) due to failures. We can specify this failure handler as

\[ \text{pol}(X) \leftarrow \text{pol}_1(X) \triangleright \text{pol}_2(X). \]

The propagate idiom is the default failure handler used in BelLog specifications. That is, we need not explicitly encode it using BelLog rules. This is because we represent failures with \( \bot \), and this truth value is always propagated unless it is explicitly handled with an operator such as error-override.

**4.3.2 Examples**

We now specify the PDPs discussed in §2.

**XACML 3.** We first observe that the failure handling in Figure 1 is independent of the policies in a policy set and of the composition operator used to compose their decisions. Therefore, to illustrate the specification of a complete PDP (i.e. one that contains both a normal policy and failure handling), we choose deny-overrides as the designated composition operator for the policies. In BelLog, the deny-overrides operator corresponds to the infinitary meet \( \bot \) over the truth ordering \( \preceq \); see Figure 6b. For a detailed, formal description of \( \bot \), see [32]. We note that other XACML 3 operators can also be encoded within the BelLog model.

The following BelLog program models the XACML 3 PDP’s failure handling with the deny-overrides operator:

**Specification 1 (S1):**

\[
\begin{align*}
\text{pol}_{\text{set}}(\text{Req}) &\leftarrow (X:\text{pol}(\text{Req}) \land \text{auth}(X, \text{Req}) \triangleright t) \\
\text{auth}(X, \text{Req}) &\leftarrow \text{admin}(X) \\
\text{auth}(X, \text{Req}) \leftarrow \text{auth}(X, \text{Req})@\text{check} \triangleright f
\end{align*}
\]

We use \( \text{Req} \) to denote access requests and \( X \) to denote principals. For brevity, we assume that each principal \( X \) has one policy for all requests, denoted by \( X:\text{pol}(\text{Req}) \). The outcome of evaluating the policy issued by the principal \( X \) is represented by \( \text{pol}(X, \text{Req})@\text{eval} \), where \( \text{eval} \) represents the PDP’s policy evaluation procedure. To represent whether \( X \) is authorized for a given \( \text{Req} \), we write \( \text{auth}(X, \text{Req}) \). Therefore \( \text{auth}(X, \text{Req})@\text{check} \) is a query to the procedure \( \text{check} \) to check whether a non-trusted policy issued by \( X \) is authorized to give decisions for the request \( \text{Req} \).

To encode that a policy is dropped if a PDP cannot evaluate it, we use the \( _\bot \rightarrow t \) pattern. This is because \( t \) is the identity element for the \( \bot \) operator. Thus, if there is an error while evaluating a policy, then \( t \) is returned, which does not influence the final outcome of the composition. It formalizes that the policy was ignored. If we were modeling another composition operator, then that operator’s identity element would be used.

To specify that a policy is dropped if a PDP cannot check its authorization, we use the \( _\bot \rightarrow f \) pattern. This means that a policy is treated as unauthorized and thus its decision is ignored (i.e. mapped to \( t \) through the if-then operator).

Finally, the for-loop is implicitly modeled using \( \bot \) and the if-then operator. The \( \bot \) operator returns the decision evaluated over the set of policies of all principals. Those policies that are not authorized are treated as the identity element and thus do not influence the result.

**Authorizations in Web Applications.** To model the web application scenario given in §2, we suppose that there are \( n \) input ACLs and one default ACL. We specify the authorization method given in Figure 2 as follows.

**Specification 2 (S2):**

\[
\begin{align*}
\text{pol}(U, O) &\leftarrow (\text{isGranted}(U, O)@\text{acl}_1 \leftarrow \cdots \\
&\quad \leftarrow \text{isGranted}(U, O)@\text{acl}_n \leftarrow (\text{isGranted}(U, O)@\text{def} \land \text{logging})
\end{align*}
\]

We model the ACL \( i \)’s evaluation of the access request \((U, O)\) with the atom \( \text{isGranted}(U, O)@\text{acl}_i \), where \( U \) represents the user and \( O \) the requested object. We model the logger’s status with the credential \text{logging}, and instantiate the catch idiom using the error-override operator. To specify the list iterator of Figure 2, we unroll the loop’s \( n \) iterations. We use the f-override operator \( (\rightarrow) \) to capture that the PDP evaluates the ACL \( i \) if the ACL \( i - 1 \) does not permit the request. This models the exit from the loop when the decision is grant. Similarly, the exit from the loop when there is a failure is captured with the catch idiom using the error-override operator. This is because if \( \text{isGranted}(U, O)@\text{acl}_i \) evaluates to \( \bot \) then the entire expression on the left-hand side of \( \triangleright \) is evaluated to \( \bot \) as well.

We recall that this specification violates \( FR1 \) because the PDP does not evaluate all ACLs if it fails to evaluate, for example, the first ACL. The reason is that the catch block is invoked prematurely. In §5, we show how our analysis reveals this security flaw, and how the flaw can be fixed.

**Authorizations in Grids.** A BelLog specification of the PDP for the grid scenario (see Figure 4) is as follows.

**Specification 3 (S3):**

\[
\begin{align*}
\text{pol}(X) &\leftarrow \text{owner}(X) \\
\text{pol}(X) &\leftarrow \text{pol}(Y) \land X:\text{grant}(Y) \\
X:\text{grant}(Y) &\leftarrow X:\text{delegate}(Y)\land \\
&\quad (\neg X:\text{revoket}(Y)@\text{rev} \triangleright \text{owner}(X))
\end{align*}
\]

The PDP stores a credential \text{owner}(X) for each domain owner \( X \). We represent a delegation from the subject \( X \) to the object \( Y \) through the if-then operator. This is because if \( \text{grant}(Y) \) is a failure is captured with the catch idiom using the if-then-else pattern, where \( \bot \) is taken. Using this operator, we can model the failure-handling idioms given in §4.1. Consider the remote query cred@pip, which checks whether the credential cred is stored at pip. To instantiate the fallback idiom, where fallback is the fallback PIP’s identifier, we write cred@pip \( \triangleright \) cred@fallback.
to the subject $Y$ with the credential $X$: delegate($Y$). The credential $X$: revoke($Y$) represents that the subject $X$ has revoked $Y$, and the remote query $X$: revoke($Y$)@rev checks whether the revocation server stores such revocations.

The top two BelLog rules encode the Datalog policy in Figure 4. The last BelLog rule encodes the check for revoked credentials. Note that the for-loop is implicitly encoded, since this BelLog rule is evaluated for all principals and subjects. The rule establishes that $X$ grants $Y$ if $X$ delegates to $Y$ and has not revoked this delegation. The failure handler is invoked for each delegation separately whenever the revocation check cannot be made. This follows the inner-loop logic of Figure 4.

To summarize, these examples demonstrate the use of BelLog and its modeling capabilities. We believe that the failure-handling idioms considered in this paper, as well as other common authorization idioms, map naturally to BelLog constructs. This makes BelLog a suitable language for specifying PDPs. Of course, there are limitations to BelLog’s modeling power. Not all procedural constructs map naturally to BelLog’s declarative specifications, for example see the list iterator of the web app example. A further investigation of BelLog’s expressiveness is orthogonal to our results and outside the scope of this paper.

5. ANALYZING ACCESS CONTROL WITH FAILURE HANDLING

The goal of our analysis is to check a PDP’s access decisions in the presence of failures. In the following, we first show how one can simulate a PDP using entailment questions in BelLog. As an example, we simulate to discover the previously described security flaw in XACML 3. Second, we show how given a BelLog PDP specification $P$, and a fail-security requirement $r$, one can formulate the problem of checking whether $P$ meets $r$ as a containment problem in BelLog. We use this to determine whether $P$ conforms to $r$ for all possible PDP inputs in our attacker model. As examples, we check whether the PDPs given in §2 meet their requirements, and we use the analysis framework to reveal flaws that violate the fail-security requirements $FR1$ and $FR2$.

5.1 BelLog Analysis

Entailment. An entailment question asks whether a BelLog program $P$ produces an atom $a$ for a given input $I$, namely whether $P \vdash I a$. Entailment is in $\text{PTIME}$ [32].

Containment. The syntax and semantics of BelLog containment are given in Figure 8. Informally, given two BelLog programs $P_1$ and $P_2$ that specify PDPs, the containment question $c \Rightarrow P_1 \preceq P_2$ is answered positively if $P_1$ is not more permissive than $P_2$ for all PDP inputs that satisfy the condition $c$. Note that a PDP has infinitely many possible inputs.

In the following, we write $\models c \Rightarrow P_1 \preceq P_2$ for $\models c \Rightarrow P_1 \preceq P_2 \preceq P_2$. To ease writing containment conditions, we provide syntactic shorthands in Figure 9. We also omit writing the condition $t$ in containment questions.

Domain containment (see Figure 8c) is decidable because there are finitely many inputs for a given domain. In fact, it is $\text{CO-NP-COMPLETE}$ [32]. In contrast, containment is in general undecidable. Nevertheless, the containment prob-

(containment question) $q ::= c \Rightarrow P_1 \preceq P_2$

(condition) $c ::= t \mid a = v \mid \forall X. c \mid \neg c \mid c \wedge c$

(truth value) $v \in D$

(variable) $X \in V$

(a) Syntax. Here $a$ is an input atom, and $P_1$ and $P_2$ are BelLog PDP specifications.

(b) Satisfaction relation between an interpretation $I$ and a containment condition $c$ for a given domain $\Sigma$.

$$\models \Sigma c \Rightarrow P_1 \preceq P_2 \iff \forall I \in \Sigma, \forall X \in \Sigma, (I \models c) \rightarrow [P_1]_I(req) \preceq [P_2]_I(req)$$

(c) Semantics. The symbol $\text{req}$ is the atom, and $\overline{X}$ is the list of free variables in a condition $c$.

Figure 8. Syntax and semantics of BelLog containment.

lem is in $\text{CO-NEXP}$ for BelLog programs whose inputs consist of only unary predicate symbols [32]. Intuitively, the BelLog programs that fall into this fragment can model PDPs where (1) all credentials provided as input to the PDP are associated to a single user, a single group, and a single resource; and (2) there are finitely many subjects who issue credentials.

Implementation. We have implemented a BelLog interpreter for deciding entailment and an analysis tool for deciding domain containment; both tools can be downloaded at www.infsec.ethz.ch/research/software/bellog. The BelLog interpreter translates BelLog entailment problems into stratified Datalog entailment problems. The domain containment tool translates domain policy containment problems into propositional validity problems.

5.2 Simulating PDPs

Given a PDP input and a request, one can use the PDP’s specification to simulate the PDP and check whether it grants or denies the request also in the presence of failures. A PDP can be simulated by posing entailment questions to its BelLog specification $S$ as follows. First, the PDP input is encoded as a BelLog input $I$, and the request is encoded as a BelLog atom $r$, as described in §4.3. Second, to check whether the PDP grants or denies $r$, we pose the entailment question $S \vdash r$.

To illustrate, we simulate the XACML 3 PDP and describe how one can find the attack described in §2. The PDP’s specification is $S_1$, given in §4.3, and we consider the following scenario. There are two policies, one issued by Ann and one by Bob. Ann is the PDP’s administrator. Let $\text{req}$ be a request such that Ann’s policy grants $\text{req}$, while Bob’s policy denies $\text{req}$. Imagine that Bob’s policy is authorized to give decisions for $\text{req}$. The PDP must therefore deny $\text{req}$ because Ann and Bob’s policies are composed us-
\[ a \neq v : = \neg(a = v) \]
\[ c_1 \lor c_2 : = \neg(\neg c_1 \land \neg c_2) \]
\[ a_1 = a_2 : = (a_1 = f \land a_2 = f) \lor (a_1 = \bot \land a_2 = \bot) \]
\[ \lor (a_1 = t \land a_2 = t) \lor (a_1 = t \land a_2 = t) \]

**Figure 9.** Shorthands for writing containment conditions.

The symbols \( a, a_1, \) and \( a_2 \) denote BelLog atoms; \( c_1 \) and \( c_2 \) denote containment conditions.

The only difference here is that the PDP fails to check whether Bob’s policy is authorized. The following BelLog input intends the deny-overrides operator. The following BelLog input models this scenario.

\[
I = \{ \text{admin}(\text{ann}) \leftarrow t, \ \text{pol}(\text{ann}, \text{req})@\text{eval} \leftarrow t, \\
\text{pol}(\text{bob}, \text{req})@\text{eval} \leftarrow f, \ \text{auth}(\text{bob}, \text{req})@\text{check} \leftarrow t \}.
\]

Here the input \( I \) describes a no-failure scenario where the PDP successfully evaluates both policies and successfully checks that Bob’s policy is authorized.

To simulate how the PDP behaves in the presence of failures, we may check the PDP’s decision for the input
\[
I_{\text{fail}} = \{ \text{admin}(\text{ann}) \leftarrow t, \ \text{pol}(\text{ann}, \text{req})@\text{eval} \leftarrow t, \\
\text{pol}(\text{bob}, \text{req})@\text{eval} \leftarrow f, \ \text{auth}(\text{bob}, \text{req})@\text{check} \leftarrow \bot \}.
\]

The only difference here is that the PDP fails to check whether Bob’s policy is authorized for \( \text{req} \). We observe that for this scenario we have \( S_1 \vdash_{I_{\text{fail}}} \text{pol}\_set(\text{req}) \), i.e., the PDP grants \( \text{req} \) because the PDP’s failure handler drops Bob’s policy decision. As the XACML committee discovered, this behavior is undesirable because an adversary may gain access by forcing the PDP to drop authorized policy decisions.

Preliminary experiments show that our simulation method scales reasonably well. Simulating the XACML 3 PDP scenario with 10, 100, and 1000 policies respectively takes 0.21, 0.41, and 1.81 seconds on a machine with a quad-core i7-4770 CPU and 32GB of RAM.

Note that our simulation method is similar to fault injection in software testing \[31,34\]: The system’s behavior is tested in various failure scenarios. The difference is that we do not directly execute the PDP’s code and instead work with its specification.

### 5.3 Verifying Fail-security Requirements

To verify that a PDP specification \( S \) meets a requirement \( r \), we formulate a number of containment problems. Each containment problem is defined using two BelLog specifications, where one of them is the PDP specification \( S \) and the other one constrains the PDP’s permissiveness, as prescribed by the requirement \( r \). In the following, we formulate and verify whether the web app and grid PDPs from §2 meet their fail-security requirements. We also give an example of a generic fail-security requirement and show that it can be verified similarly.

**Authorizations in Web Applications.** Consider the PDP specification \( S_2 \) and the fail-security requirement \( FR1 \), which states that when the PDP cannot compute an access decision due to malformed or missing specifications, then it uses the default specification if logging is enabled; otherwise, it denies access.

To determine whether \( S_2 \) meets \( FR1 \), we first write a condition that is satisfied by the inputs for which the PDP cannot compute an access decision due to failures. Since the ACLs are composed with the permit-overrides operator, the PDP grants a request if any of the ACLs grant the request, and it denies it if all the ACLs deny it; otherwise, the PDP cannot compute a decision and it must, as prescribed by \( FR1 \), evaluate the default ACL and check the logging status. We encode the containment condition as

\[
c_{\text{error}} = \neg \left( (\text{is Granted}(U, O)@\text{acl}_1 = t \lor \cdots (\text{is Granted}(U, O)@\text{acl}_n = f) \right) \\
\lor \left( (\text{is Granted}(U, O)@\text{acl}_1 = t \land \cdots \land (\text{is Granted}(U, O)@\text{acl}_n = f) \right) \right).
\]

We then construct the BelLog specification \( R_{\text{error}} \):

\[
R_{\text{error}} = \{ \text{pol}(U, O) \leftarrow (\text{is Granted}(U, O)@\text{def} \land \text{logging}) \}.
\]

The specification \( R_{\text{error}} \) evaluates to grant if the PDP’s default ACL evaluates to grant and logging is enabled; otherwise it evaluates to deny. Finally, to check whether the specification \( S_2 \) meets \( FR1 \), we formulate the containment problem

\[
c_{\text{error}} \Rightarrow S = R_{\text{error}}.
\]

Our analysis tool shows that the specification \( S_2 \) violates the requirement \( FR1 \) for the PDP input

\[
I = \{ \text{is Granted}(\text{ann}, \text{file})@\text{acl}_1 \leftarrow \bot, \\
\text{is Granted}(\text{ann}, \text{file})@\text{acl}_2 \leftarrow t, \\
\text{is Granted}(\text{ann}, \text{file})@\text{def} \leftarrow f \}.
\]

\( S_2 \) violates \( FR1 \) because it denies the request \( \text{pol}(\text{ann}, \text{file}) \) even though ACL 2 grants this request.

To meet \( FR1 \), the PDP must correctly implement the propagate failure-handling idiom and apply the failure handler only if it fails to evaluate an ACL and all remaining ACLs deny access. We correct the PDP’s specification as follows.

**Specification 4 (\( S_4 \)):**

\[
\text{pol}(U, O) \leftarrow (\text{is Granted}(U, O)@\text{acl}_1 \lor \cdots \lor (\text{is Granted}(U, O)@\text{acl}_n) \lor (\text{is Granted}(U, O)@\text{def} \land \text{logging})
\]

Our tool shows that \( S_4 \) meets \( FR1 \) for a PDP with 10 ACLs, for all PDP inputs in a fixed domain of 10 constants. The verification takes 0.03 seconds. Naturally, the verification time increases with the number of ACLs and the domain size. For example, the verification time for a PDP with 100 ACLs and inputs ranging over domains of size 10, 100, and 1000 is 0.13, 2.09, and 34.42 seconds, respectively.

We give the pseudo-code for the authorization method that implements \( S_4 \) in Figure 10. This method delays handling failures until all ACLs have been evaluated. The PDP correctly implements the propagate idiom, i.e. it consults the ACL def only if no input ACL grants the request and the PDP has failed to evaluate at least one of them (recorded in the error variable).

**Authorizations in Grids.** Consider the PDP specification \( S_3 \) and the fail-security requirement \( FR2 \), which states that when the PDP cannot check whether a subject has at
least one non-revoked delegation chain due to failures, the
PDP grants access if the subject is a direct delegate of the
owner; otherwise it denies access.

To verify that the specification meets the requirement,
we formulate two containment problems. The first problem
checks whether the PDP correctly evaluates the requests
made by direct delegates and the second one checks whether
the PDP correctly evaluates the requests made by non-direct
direct delegates. We formulate these containment problems as the
BelLog program

\[
R_{\text{chain}} = \{\,
\begin{align*}
\text{chain}(X) &\leftarrow \text{owner}(X) \\
\text{chain}(X) &\leftarrow \text{chain}(Y) \land Y:\text{delegate}(X) \land \neg Y:\text{revoke}(X)\text{@rev } \\
\end{align*}
\}
\]

Given a subject \(X\), \(\text{chain}(X)\) is: (1) \(\top\) if the PDP checks
that \(X\) has at least one non-revoked chain, (2) \(\bot\) if the PDP
fails to check whether \(X\) has at least one non-revoked chain,
and (3) \(\bot\) if \(X\) has no chains or the PDP checks that \(X\)
has only revoked chains. We use the containment condition

\[
e_{\text{direct}} = (\exists Y. \text{owner}(Y) = t \land Y:\text{delegate}(X) = t \land Y:\text{revoke}(X)\text{@rev } \neq t),
\]

which is satisfied by a PDP input iff the subject \(X\) who
makes the request is a direct delegate and the owner has
either not revoked the delegation or the PDP cannot check
if the delegation is revoked.

We formulate the first containment problem as

\[
e_{\text{direct}} \Rightarrow S = R_{\text{direct }} , \text{ where}
\]

\[
R_{\text{direct}} = R_{\text{chain}} \cup \{\text{pol}(X) \leftarrow \text{chain}(X)\text{@t}\}
\]

The condition \(e_{\text{direct}}\) restricts PDP inputs to direct delegates
and \(R_{\text{direct}}\) specifies which direct delegates \(S\) must grant and
deny access to. Since the PDP must grant access to a
direct delegate \(X\) iff the PDP either checks, or fails to check,
that \(X\) has at least one non-revoked chain, \(R_{\text{direct}}\) conflates
\(\bot\) and \(\top\) into the grant decision using the (\(\top\) \(\Rightarrow\) \(\top\)) pattern.

We formulate the second problem as

\[
(\neg e_{\text{direct}}) \Rightarrow S = R_{\text{non-direct}} , \text{ where}
\]

\[
R_{\text{non-direct}} = R_{\text{chain}} \cup \{\text{pol}(X) \leftarrow \text{chain}(X)\text{@f}\}
\]

The condition \(\neg e_{\text{direct}}\) restricts PDP inputs to non-direct del-
egates and revoked direct delegates, and \(R_{\text{non-direct}}\) specifies
which ones \(S\) must grant and deny access to. Since the PDP
must deny access to a non-direct delegate \(X\) iff the PDP
fails to check that \(X\) has at least one non-revoked chain or
\(X\) has only revoked chains, \(R_{\text{non-direct}}\) conflates \(\bot\) and \(\bot\)
to the deny decision using the (\(\bot\) \(\Rightarrow\) \(\bot\)) pattern.

Our analysis tool shows that the PDP specification S3
does not meet \(FR_2\) because the problem

\[
(\neg e_{\text{direct}}) \Rightarrow S = R_{\text{non-direct }}
\]

is answered negatively. The tool outputs the following PDP
input:

\begin{align*}
I &= \{\text{owner(piet) } \leftarrow \text{t}; \text{ piet}\text{.delegate(ann) } \leftarrow \text{t}; \text{ piet}\text{.revoke(ann)@rev } \leftarrow \bot; \text{ ann}\text{.delegate(fred) } \leftarrow \text{t}; \text{ ann}\text{.revoke(fred)@rev } \leftarrow \text{f}\}\}
\end{align*}

In this scenario, Piet is the owner, and he delegates access
to Ann, who further delegates access to Fred. Furthermore,
the PDP fails to check whether Piet’s delegation to Ann
is revoked, and it succeeds in checking that Ann has not
revoked Fred; see Figure 3. The PDP must deny access
to Fred because he does not have a non-revoked delegation
chain and he is not a direct delegate. The PDP, however,
grants access to Fred, thus violating \(FR_2\). This flaw stems
from the preemptive masking of failures. The adversary Fred
can exploit this flaw and force an unintended grant decision
by preventing the PDP from checking whether the owner’s
delegation to Ann is revoked. To confirm the attack, we
simulated the attack scenario using our BelLog interpreter;
for details see Appendix A.

To meet \(FR_2\), we modify the specification as follows.

**Specification 5 (S5):**

\begin{align*}
\text{pol}(X) &\leftarrow \text{grant}(X) \Rightarrow (\text{owner}(Y) \land Y:\text{delegate}(X) \land \\
\neg Y:\text{revoke}(X)\text{@rev }) \\
\text{grant}(X) &\leftarrow \text{owner}(X) \land Y:\text{delegate}(X) \land (\neg Y:\text{revoke}(X)\text{@rev }) \\
\end{align*}

In the original specification S3, errors are not propagated
down delegation chains. In contrast, the specification S5
propagates errors through delegation chains and thus denies
access to subjects who are not direct delegates and do not
have a non-revoked chain. The pseudo-code that reflects S5
would have to, in effect, distinguish between permissions
solely due to direct delegation versus permissions due to
non-revoked chains.

Our analysis tool shows that the specification S5 meets \(FR_2\)
for all PDP inputs in a fixed policy domain with eight
constants; the verification takes 149.38 seconds. Our tool
did not terminate in a reasonable time for larger domains.

We remark that domain containment gives weaker secur-
ity guarantees than (general) policy containment because
the guarantees are only for the given policy domain. Hence,
domain policy containment does not account for possible
attacks in other domains. For example, domain policy con-
ainment misses the attack described in our grid example
if the policy domain has only two constants (e.g., two
subjects). This is because the adversary must assume the role
of a subject who is delegated access by a direct delegate, and
such a subject does not exist in a domain with fewer than
three constants.

**Generic Requirements.** In addition to the aforementioned
requirements, one can verify whether a PDP meets
certain generic security requirements. For example, one may
want to ensure that a PDP handles all failures, i.e. it always
evaluates requests to either grant or deny decisions. We refer
to this requirement as *error-freeness*, and show how it can be checked by formulating suitable containment problems.

Let $S$ be the PDP specification and $pol(X)$ be the atom used to denote the PDP’s access decisions. We construct a specification $R$ as follows. Let $R = \emptyset$. We rename the predicate symbol $pol$ to $tmp$ in $S$’s rules and add the changed rules to $R$. Finally, we add the rule

\[ \text{pol}(X) \leftarrow \text{tmp}(X) \lor f, \]

to $R$. We formulate the containment problem as $S \equiv R$. By construction, $R$ denies all requests that $S$ evaluates to $\bot$. Therefore, if $S$ evaluates a request to $\bot$, then $R$ is not equal to $S$; otherwise, $S$ is error-free. Note that one can similarly verify that any atom other than $\text{pol}()$ in the PDP’s specification is error-free.

To conclude, these examples show that our simulation and verification methods can reveal security flaws in PDPs that handle failures incorrectly. Our preliminary experiments show that our simulation tool scales well to realistic problems. The runtimes for our analysis tool, however, are mixed. In our grid example the analysis tool does not terminate in a reasonable amount of time for a domain with nine constants, whereas in the web app example the tool terminates in less than a minute for domains with thousands of constants.

6. SUMMARY AND FUTURE WORK

We have initiated the study of how failure handlers affect access decisions made by a PDP, and we have provided methods and tools to analyze their effects. We have given examples from standards and existing systems that back our arguments.

We are currently working on employing our analysis framework in physical access control systems used in industry. Addressing BelLog’s usability is a major challenge in this context. As future work, we also plan to improve the scalability of our analysis tool, and extend our system model to multiple communicating PDPs, where PDPs themselves can fail.

7. ACKNOWLEDGMENTS

This work was supported in part by the Zurich Information Security and Privacy Center. We thank Andreas Häberli and Paul Studerus from KABA AG for their feedback on our system model, and Sasa Radomirovic for his comments on the paper.

8. REFERENCES

APPENDIX

A. ATTACK ON THE GRID PDP

We simulate here the attack on the grid PDP module, as discussed in §5. To do this, we first encode the PDP module in our interpreter using BelLog’s ASCII format:

\[
\text{pol}(X) :- \text{owner}(X) \\
\text{pol}(X) :- (\text{pol}(Y) \land \text{grant}(Y,X)) \\
\text{grant}(X,Y) :- (\text{delegate}(X,Y) \land (\neg \text{revoke}(X,Y) \land \text{rev})) \\
\text{rev} :- \bot \rightarrow \text{owner}(X)
\]

The ASCII encoding of a BelLog rule \( A \leftarrow B \) is \( A :- B \). We write the operators \( \neg \), \( \sim \), and \( \land \) with the ASCII symbols \(!\), \(\sim\), and \(^\land\), respectively. To ease the writing of BelLog specifications, we have implemented several of the syntactic shorthands presented in §4. For example, the \( v \)-override operator \( v \mapsto \rightarrow \) is written as \(-v->\), where \( v \) is true (t), false (f), bot (⊥), or top (⊤). Our interpreter also supports BelLog’s extension for nesting and composing rule bodies with the operators \(!\), \(\sim\), and \(^\land\); see for example the last rule in the specification given above. For the complete ASCII syntax see www.infsec.ethz.ch/research/software/bellog.

Second, we specify the PDP input produced by our analysis tool in §5.3:

\[
\text{owner}(piet) :- \text{true} \\
\text{delegate}(piet,ann) :- \text{true} \\
\text{delegate}(ann,fred) :- \text{true} \\
\text{revoke}(piet,ann)@rev :- \bot \\
\]

Note that we need not explicitly write the rule

\[
\text{revoke}(ann,fred) :- \text{false}
\]

because the default truth value assigned to any atom is false.

Finally, we run the interpreter and verify that the attacker gains access. We observe that the PDP grants access to Fred because his request \( \text{pol}(fred) \) evaluates to true. This behavior does not conform to FR2: the PDP must deny access to Fred because he does not have a non-revoked delegation chain and he is not among the owner’s direct delegates.

References