Fail-Security in Access Control

ABSTRACT

Decentralized and distributed access control systems are subject to communication and component failures. These can affect access decisions in surprising and unintended ways, easily 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 prone to failures. 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 ought to handle failures. To illustrate our contributions, we analyze the consequences of failure handling in the XACML 3 standard and in examples from other domains. Our analysis reveals a number of security flaws.

General Terms

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. Firewalls must operate even when their log engines crash [27] or rule updates fail [29], web applications must service their clients even when authentication services are not responding [28], delegation systems must evaluate requests even when they cannot update their revocation lists, and perimeter security systems must control access even when 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 PDP's access decisions, it follows that formal analysis frameworks for access control systems 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 the 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 governs deciding access requests is therefore split into two parts. This separation makes the PDP's behavior particularly difficult to understand and analyze.

Existing formal analysis frameworks for access control policies are not adequate 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) 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 concepts using the BelLog analysis framework [32].

Contributions. This is the first paper that systematically analyzes the role of failure handling in access control systems. We investigate three classes of security flaws: misapplying composition operators to failed sub-policies, overly eager failure handling, and 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 PDP decisions. (2) We derive common failure-handling idioms from these seven systems; the idioms can be readily encoded in BelLog. (3) Through examples, we describe 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 classes 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, 14, 17, 19, 26], do not explicit 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, 15, 16, 18, 24] 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. Currently these frameworks all lack the three contributions of this paper, as listed above. We remark that dynamic analysis frameworks, such as [2, 15, 18], 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 of §2. In §5, we analyze the examples of §2 against their fail-security requirements. We point to our future work directions in §6.

2. MOTIVATION

In this section, we use the XACML 3 standard as an example to show that approximating failures with denials, although seemingly conservative, can lead to insecure systems. This is also evident in our second and third examples, which follow the common PDP implementation pattern of separating failure handlers from the normal policy engine. We use the latter two examples, taken from the web application and grid computing domains, to show that the separation makes understanding and analyzing such PDPs particularly difficult, and also to illustrate their fail-security requirements.

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 called *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-override, 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. Then, 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.

```
// PolicySet's evaluate method
evaluate(Request req)
  Set decisions
  for (pol in policies)
    try
      decision = pol.evaluate(req)
      if (pol.issuer == admin) or authorize(pol, req) then
        decisions.add(decision)
    catch (EvaluationException e)
      skip
  return compositionOperator.apply(decisions)
```

Figure 1. PDP model for evaluating XACML 3 policy sets. The methods pol.evaluate(req) and authorize(pol, req) throw an exception if the PDP fails to execute them.

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 the Revision 16, defined that the PDP should refrain from using policies that could not be evaluated or authorized due to communication and PIP failures. The decision follows the intuitive idea that all *suspicious* policies should be excluded from the PDP's decision. Figure 1 specifies this PDP, including the 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 modeled together with the deny-override composition operator, the following attack was discovered [37]. Consider a request r and a policy set P that contains one trusted policy P_1 that grants r and one authorized non-trusted policy P_2 that denies r. P's decisions are combined with deny-overrides. If the PDP successfully evaluates P_1 and fails to evaluate P_2, 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 access for r. In §5 we show how this attack can be found through automated analysis.

This example illustrates our argument: a PDP's failure handlers, regardless of their simplicity, can affect its access decisions in surprising ways. In this example, failure-oblivious composition of sub-policies is the root of the security flaw. To counter the attack, the XACML 3 standard currently uses and overloads a designated policy decision (the indeterminate *IN*) for every policy that cannot be evaluated due to failures. Consequently, failure handlers are now a concern of the policy writer.

```
isAuthorized(User u, Object o, ListAclId ids)
  try
    for (id in ids)
      if (readAcl(id).grants(u,o)) then return true
  catch (NotFoundException e)
    return def.grants(u,o) && logger.on()
  return false
```

Figure 2. A PDP object for the web app example.

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 a variety of declarative policy languages. A *PDP object* loads policy specifications and evaluates them within its *authorization* method. A reoccurring problem is that the PDP object fails to load a declarative specification due to syntactic errors or due to missing source files. To deal with this problem, administrators often maintain a *default specification* that is used as a fallback option. Using 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 object 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, consider the case where the PDP object is required to compose a finite list of access control lists (ACLs) using the permit-override operator, which permits if at least one of the ACLs permits and denies otherwise. To adhere to *FR1*, the PDP object must invoke the failure handler if and only if none of the ACLs grants and at least one of them is malformed. Otherwise, the PDP object should output a deny. The failure handler in this example would evaluate the default ACL `def`, if logging is enabled. Figure 2 gives a straight-forward authorization method for this scenario in pseudo-code. The method takes as input a user object `u`, the requested object `o`, and a list `ids` of ACL identifiers. The method `readAcl(id)` returns the ACL object corresponding to `id`, and throws a `NotFoundException` 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-override 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 transfer of control from the input ACLs to the default ACL `def` in the pseudo-code is overly eager and thus the method does not satisfy *FR1*. For instance, 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 grant.

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* [21], and it cannot be solved by, e.g., 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.

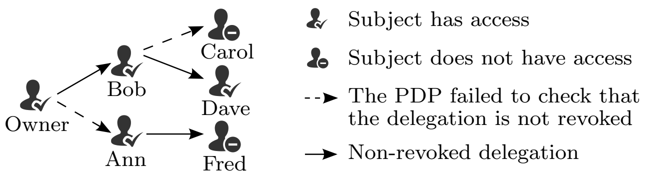

Carol
Bob
Dave
Owner
Ann
Fred
Subject has access
Subject does not have access
The PDP failed to check that the delegation is not revoked
Non-revoked delegation

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.

```
pol(X) :- owner(X)
pol(X) :- pol(Y), grant(Y,X)
```

(a) Access control policy del.policy (in Datalog).

```
isAuthorized(Subject s, List delegations)
  datalogEngine.load(del.policy)
  for ((delegator, delegatee) in delegations)
    try
      if (rev.query(delegator, delegatee) == false)
        datalogEngine.assert(grant(delegator, delegatee))
    catch (QueryException e)
      if isOwner(delegator)
        datalogEngine.assert(grant(delegator, delegatee))
  return datalogEngine.check(pol(s))
```

(b) PDP model, where engine represents a Datalog interpreter.

Figure 4. A PDP object for the grid example.

To conclude, because existing web access control frameworks typically separate failure handling from the normal policy of the PDP, it is challenging to gain confidence in their security. To rise to this challenge, policy analysis formalisms 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 (e.g. storage space) are located in different domains. Each domain has one 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. To this end, domain owners delegate authorization management to their *trusted* subjects, possibly from other domains. These subjects may then issue tokens to authorize, and to further delegate their rights to, other subjects. All tokens are stored as digital credentials. Subjects then submit their credentials, alongside their access requests, to a PDP. In addition, it is often necessary to be able to revoke subject's credentials, for example when dealing with 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 owner to S. We say that a delegation chain is non-revoked if none of the delegations in the chain are revoked. The PDP grants access if the subject has at least one *non-revoked* delegation chain or he is a domain owner. The revocation server may sometimes be unavailable, for example due to lost network connectivity. Denying all access

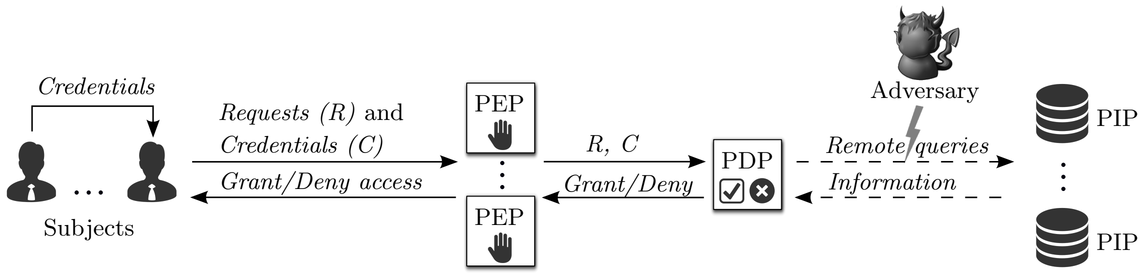


Figure 5. System model for decentralized and distributed access control systems. Subjects submit requests and credentials to a PEP. The PDP communicates with PIPs and decides all requests.

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 follows:

Fail-security requirement 2 (FR2): When the PDP cannot check due to failures whether a subject has at least one non-revoked delegation chain, the PDP grants access if the subject is an owner's direct delegate; 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 have 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 delegation logic from failure handling, is detailed in [8]. Based on these guidelines, Figure 4 depicts a possible PDP design for our grid access control scenario. In Figure 4(a), we specify the normal policy of the PDP in Datalog, the core of many delegation access control languages (e.g. see [3,14,17, 26]). 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 `rev`. 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 *FR 2*. Their interaction however leads to a subtle attack. The attack, described in § 5, is a result of 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.

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 an attribute (such as a role) to a subject or a resource. Credentials are issued by, and exchanged between, subjects; they can also be locally stored at the PDP. We assume that the PDP verifies their authenticity, e.g. 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 PIP components 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 the following responses: (1) the answer to q; or (2) *error*, indicating a communication failure. Note that in our model, PIP failures are indistinguishable from communication failures.

An adversary is a subject who can cause any remote query to fail. The adversary cannot however forge credentials or forge and replay past remote queries and obsolete responses. To this end, we consider that all communication channels are authentic and have freshness guarantees (through timestamps, nonces, etc.). Note that our adversary model subsumes all failures due to benign causes. The adversary can in particular cause complete channel failure by causing all remote queries through that channel to fail. We remark that query confidentiality and information flow concerns [4] are outside of this paper's scope.

In addition to the examples given in §2, this system model encompasses many other real-world access control settings, such as authorization systems for electronic health records [5].

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 derived 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. 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 fail or terminate successfully. Note that similar abstractions exist for exception handling in programming languages [20, 25].

Fallback. The fallback idiom abstracts the failure handlers that use *fallback* information sources when the communication channels to primary information sources fail. If a communication event fails then it is re-executed using the fallback source. The fallback source can be, e.g., a backup of the primary information source. For example, this idiom can be used in access control systems whose primary authentication services (such as LDAP) is unreliable; in this case, the system can fall back on the local user/password lists.

To instantiate this idiom, a fallback source must be configured for each information source that is prone to failures. Although the fallback source may be periodically synchronized with the information source, nevertheless it may provide stale information of inferior quality. Systems that employ this idiom include Kerberos [23], which uses fallback authentication mechanisms.

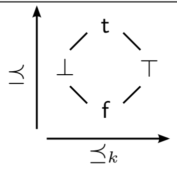
Catch. This idiom abstracts the failure handlers that catch failures and then enforce alternative access control policies when failures happen. The catch idiom is analogous to exception handling in programming languages, where a failure to execute a given procedure is handled by a designated procedure. 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 [22], 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 [29], 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 the requirements with such "delayed" failure-handling. Whenever an event fails, the PDP pushes a designated *error value* as the input to all subsequent events.

$$
\begin{array}{rlrl}
(\textit{program}) & P ::= \bar{r} & (\textit{rule}) & r ::= a \leftarrow \bar{l} \\
(\textit{literal}) & l ::= a \mid \neg a \mid {\sim}a & (\textit{atom}) & a ::= p(\bar{t}) \\
(\textit{term}) & t ::= c \mid v & & \\
(\textit{predicates}) & p \in \mathcal{P} \supset \mathcal{D} & (\textit{variables}) & v \in \mathcal{V} \\
(\textit{constants}) & c \in \mathcal{C} & &
\end{array}
$$

(a) Syntax.



	¬	∼
f	t	f
⊥	⊥	⊤
⊤	⊤	⊥
t	f	t

(b) BelLog's truth space $\mathcal{D} = \{\mathsf{f}, \bot, \top, \mathsf{t}\}$. $\wedge$ and $\vee$ denote the meet and join over $(\mathcal{D}, \preceq)$. $\preceq_k$ is the knowledge partial order.

$$
\begin{array}{rl}
(\textit{domain}) & \Sigma \subset \mathcal{C} \quad (\Sigma \text{ is finite}) \\
 & P^{\downarrow} = P\text{'s grounding over } \Sigma \\
(\textit{ground atoms}) & \mathcal{A} = \{p(\bar{t}) \mid \bar{t} \subseteq \Sigma\} \\
(\textit{interpretations}) & I, I' \in \mathcal{I} = \mathcal{A} \mapsto \mathcal{D} \\
 & I \sqsubseteq I' \text{ iff } \forall a \in \mathcal{A}.\ I(a) \preceq I'(a) \\
(\textit{consequence} & T_P = \mathcal{I} \mapsto \mathcal{I} \\
\textit{operator}) & T_P(I)a = \bigvee\{I(\bar{l}) \mid (a \leftarrow \bar{l}) \in P^{\downarrow}\} \\
(\textit{strata models}) & M_0 = \sqcap\mathcal{I},\ M_{i+1} = \mathsf{lfp}_{M_i} T_{P_{i+1}} \sqcup M_i \\
 & \mathsf{lfp}_I T_P = \sqcap\{\mathsf{fp} T_P \mid I \sqsubseteq \mathsf{fp} T_P\} \\
(\textit{model}) & [\![P]\!] = M_n
\end{array}
$$

(c) Semantics. $\sqcap$ and $\sqcup$ denote the meet and join over $(\mathcal{I}, \sqsubseteq)$. P's rules are partitioned into strata $P_1, \cdots, P_n$.

Figure 6. BelLog's syntax and semantics.

For example, to meet *FR1*, ideally an error value that indicates a failure to evaluate an ACL is propagated. If no ACL grants a given request and the PDP failed to evaluate an ACL, then the default ACL is evaluated; otherwise, the error value is ignored and the PDP grants the request. Note that the failure handler of Figure 2 is however an instance of the catch idiom. Systems that employ this 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 pick the BelLog language as the basis for specifying the PDP's behavior including its failure handlers. We favor BelLog because: (1) we can use its truth value $\bot$ to explicitly denote failures, (2) we can syntactically extend its core language to define common failure handlers, (3) it can encode the state-of-the-art decentralized policy languages [3, 14, 26]. 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 $\bar{r}$ to denote finitely many rule occurrences separated by a comma. 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, $\neg a$, or ${\sim}a$, where a is an atom, and $\neg$ and $\sim$ are the truth- and knowledgenegation 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 $(\mathcal{D}, \preceq, \wedge, \vee)$; see Figure 6b. The non-classical truth values $\bot$ and $\top$ denote *undefined* and *conflict*. In §4.3, we show how $\bot$ is used to denote missing information due to failures. BelLog's semantics is given in Figure 6c. Given a finite set of constants Σ, called the *domain*, $P^{\downarrow}$ denotes the program obtained by replacing all variables in P in all possible ways using Σ, and $\mathcal{A}$ is the set of *ground atoms* constructed from Σ without variables. P's model is an element from the lattice $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup)$, where $\mathcal{I}$ is the set of all *interpretations*. An interpretation $I \in \mathcal{I}$ maps ground atoms to truth values. The symbols $\neg$, $\sim$, and $\wedge$ are overloaded over interpretations in the standard way. A rule $a \leftarrow \bar{l}$ assigns the truth value of $I(\bar{l})$ to a. The *consequence operator* T_P applies P's rules and joins the results with $\bigvee$ when multiple rules have the same head. Program P's model is constructed by first partitioning P's rules into strata $P_1, \cdots, P_2$ and then computing, for each stratum, the least fixed point (lfp) that contains the model of the previous stratum. A partitioning $P_1, \cdots, P_n$ is a *stratification* if for each stratum P_i, any predicate symbol that appears in a negative literal $\neg p(\bar{t})$ does not appear in the head of $P_i \cup \cdots \cup P_n$.

Finally, the *input* to a BelLog program P is a set P_I of rules of the form $p(\bar{t}) \leftarrow v$, where $v \in \mathcal{D}$ and p does not appear in the heads of P's rules. We write $[\![P]\!]_I$ for $[\![P \cup P_I]\!]$.

4.3 Specifying PDPs in BelLog

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

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, *public*(ann, file) is interpreted as "ann asserts that file is public". Hereafter, we write ann:*public*(file) to emphasize a credential's issuer. For brevity, we omit writing "admin:" to denote admin's credentials.

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

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

$$
\begin{aligned}
p \vee q &:= \neg(\neg p \wedge \neg q) \\
p = \mathsf{f} &:= \neg(p \vee {\sim} p) \\
p = \mathsf{t} &:= p \wedge {\sim} p \\
p \lhd c \rhd q &:= ((c = \mathsf{t}) \wedge p) \\
&\quad \vee ((c \neq \mathsf{t}) \wedge q)
\end{aligned}
\qquad
\begin{aligned}
p \neq v &:= \neg(p = v) \\
p = \bot &:= (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \\
&\quad \wedge ((p \vee \top) = \mathsf{t}) \\
p \overset{v}{\mapsto} q &:= q \lhd (q = v) \rhd p
\end{aligned}
$$

Figure 7. Derived BelLog operators. Here p, q, and c denote rule bodies, and $v \in \mathcal{D}$.

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 [3], RT [26], Binder [14], and DKAL [19], 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 $\neg$, $\sim$, and $\wedge$. For example, the rule $r \leftarrow \neg(\neg p \wedge \neg q)$, where p and q are rule bodies, assigns to r the truth value computed by applying the operators $\neg$ and $\wedge$ to the truth values computed for p and q. Additional BelLog operators, such as the *if-then-else* operator ($\_ \lhd \_ \rhd \_$) and the *v-override* operator ($\_ \overset{v}{\mapsto} \_$), are defined in Figure 7.

We define the *error-override* operator as

$$p \blacktriangleright q := p \overset{\bot}{\mapsto} q \ ,$$

where p and q are rule bodies. The construct $p \blacktriangleright q$ 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 $\blacktriangleright$ *cred*@fallback.

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

$$
\begin{aligned}
&pol_1(X) \leftarrow empl(X)@\mathsf{db} && \text{(Policy } P_1\text{)} \\
&pol_2(X) \leftarrow stud(X) && \text{(Policy } P_2\text{)}
\end{aligned}
$$

Here the atom $pol_i(X)$ denotes policy P_i's decision. The communication between the PDP and the PIP db can fail. Imagine that the PDP instantiates the catch idiom and uses P_2 when it cannot evaluate P_1 due to failures. We can specify this failure handler as

$$pol(X) \leftarrow pol_1(X) \blacktriangleright pol_2(X) \ .$$

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, and explain how the BelLog specifications reflect their corresponding PDPs.

XACML 3. We first observe that the failure handling in Figure 1 is independent of policies in a policy set and of the composition operator that composes 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-override as the designated composition operator for the policies. In BelLog, the deny-override operator corresponds to the infinitary meet $\bigwedge$ over the truth ordering $\preceq$; see Figure 6b. For a detailed, formal description of $\bigwedge$ see [32]. We note that other XACML 3 operators can also be encoded within the given PDP model.

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

Specification 1 (S1):

$$\begin{aligned} \mathit{pol\_set}(\mathit{Req}) &\leftarrow \bigwedge \big(X{:}\mathit{pol}(\mathit{Req}) \lhd \mathit{auth}(X, \mathit{Req}) \rhd \mathsf{t}\big) \\ \mathit{auth}(X, \mathit{Req}) &\leftarrow \mathit{admin}(X) \\ \mathit{auth}(X, \mathit{Req}) &\leftarrow \mathit{auth}(X, \mathit{Req})@\mathsf{check} \blacktriangleright \mathsf{f} \\ X{:}\mathit{pol}(\mathit{Req}) &\leftarrow \mathit{pol}(X, \mathit{Req})@\mathsf{eval} \blacktriangleright \mathsf{t} \end{aligned}$$

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

To encode that a policy is dropped if a PDP cannot evaluate it, we use the $(\_ \blacktriangleright \mathsf{t})$ pattern. This is because t is the identity element for the $\bigwedge$ operator. Thus, if there is an error while evaluating a policy, then *true* is returned, which does not have any influence on the final composition outcome. It formalizes that the policy was ignored. If we were modeling another composition operator, then the *drop* atom would be set to that operator's identity element.

To specify that a policy is dropped if a PDP cannot check its authorization, we use the $(\_ \blacktriangleright \mathsf{f})$ pattern. This means that a policy is treated as not authorized and thus its decision will be ignored (i.e. mapped to t through the if-then operator).

Finally, the for-loop construct is implicitly modeled through $\bigwedge$ and the if-then operator. The $\bigwedge$ 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):

$$\begin{aligned} \mathit{pol}(U, O) \leftarrow &(\mathit{isGranted}(U, O)@\mathsf{acl}_1 \overset{\mathsf{f}}{\mapsto} \cdots \\ &\cdots \overset{\mathsf{f}}{\mapsto} \mathit{isGranted}(U, O)@\mathsf{acl}_\mathsf{n}) \\ &\blacktriangleright (\mathit{isGranted}(U, O)@\mathsf{acl\_def} \wedge \mathit{logging}) \end{aligned}$$

We model the ACL i's evaluation of the access request (U, O) with the atom $\mathit{isGranted}(U, O)@\mathsf{acl\_i}$, where U represents the user and O the requested object. We model the logger's status with the credential *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 ($\overset{\mathsf{f}}{\mapsto}$) 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 $\blacktriangleright$ operator. This is because if $\mathit{isGranted}(U, O)@\mathsf{acl\_i}$ evaluates to e then the whole expression on the left-hand side of $\blacktriangleright$ is evaluated to e as well.

We recall that this specification does not meet FR1, because the PDP does not evaluate all ACLs if it fails to evaluate, for example, the first ACL. The reason for this modeling error is that the catch idiom is overly eager, i.e. it handles the failure prematurely. In §5, we show how our analysis is used to reveal this problem, and how it can be fixed.

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

Specification 3 (S3):

$$\begin{aligned} \mathit{pol}(X) &\leftarrow \mathit{owner}(X) \\ \mathit{pol}(X) &\leftarrow \mathit{pol}(Y), X{:}\mathit{grant}(Y) \\ X{:}\mathit{grant}(Y) &\leftarrow X{:}\mathit{delegate}(Y), \\ &\qquad ((\neg X{:}\mathit{revoke}(Y)@\mathsf{rev}) \blacktriangleright \mathit{owner}(X)) \end{aligned}$$

The PDP stores a credential $\mathit{owner}(X)$ for each domain owner X. We represent a delegation from the subject X to the subject Y with the credential $X{:}\mathit{delegate}(Y)$. The credential $X{:}\mathit{revoke}(Y)$ represents that the subject X has revoked Y, and the remote query $X{:}\mathit{revoke}(Y)@\mathsf{rev}$ 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 conclude, through these examples we have demonstrated 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. There are naturally limitations to BelLog's modeling power. Not all procedural constructs map naturally to BelLog's declarative specifications, e.g. see the list iterator of the web app example. Further investigation into the expressiveness of BelLog 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

$$\begin{array}{rl}(\textit{containment question}) & q ::= c \Rightarrow P_1 \preceq P_2 \\ (\textit{condition}) & c ::= \mathsf{t} \mid a = v \mid \forall X.\ c \mid \neg c \mid c \wedge c \\ (\textit{truth value}) & v \in \mathcal{D} \\ (\textit{variable}) & X \in \mathcal{V}\end{array}$$

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

$$\begin{array}{ll}I \Vdash_\Sigma \mathsf{t} & \\ I \Vdash_\Sigma a = v & \textit{if } I(a) = v \\ I \Vdash_\Sigma \forall X.\ c(X) & \textit{if } \forall X \in \Sigma.\ I \Vdash_\Sigma c(X) \\ I \Vdash_\Sigma \neg c & \textit{if } I \nVdash_\Sigma c \\ I \Vdash_\Sigma c_1 \wedge c_2 & \textit{if } I \Vdash_\Sigma c_1 \textit{ and } I \Vdash_\Sigma c_2\end{array}$$

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

$(\textit{domain containment})$

$$\Vdash_\Sigma c \Rightarrow P_1 \preceq P_2 \textit{ iff } \forall I \in \mathcal{I}.\ \forall \overline{X} \in \Sigma.\ (I \Vdash_\Sigma c) \rightarrow [\![P_1]\!]_I(\textit{req}) \preceq [\![P_2]\!]_I(\textit{req})$$

$(\textit{containment})$

$$\Vdash c \Rightarrow P_1 \preceq P_2 \quad \textit{iff} \quad \forall \Sigma \subset \mathcal{C}.\ \Vdash_\Sigma c \Rightarrow P_1 \preceq P_2$$

(c) Semantics. The symbol *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.

show how one can *simulate* a PDP using *entailment* questions in BelLog. As an example, we use simulation to discover the discussed security flaw in XACML 3; see §2. 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 verify 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 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 derives an atom a for a given input I, namely whether $[\![P]\!]_I(a) = \mathsf{t}$. Entailment is in 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 $\Vdash c \Rightarrow P_1 = P_2$ for $\Vdash c \Rightarrow P_1 \preceq P_2$ and $\Vdash c \Rightarrow P_2 \preceq P_1$. 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 CO-NP-COMPLETE [32]. In contrast, containment is in general undecidable. Nevertheless, the containment problem is in CO-NEXP for BelLog programs whose inputs consists of only unary predicate symbols [32]. Intuitively, the BelLog programs that fall into this fragment can model PDPs where

$$\begin{array}{rcl}c_1 \vee c_2 & := & \neg(\neg c_1 \wedge \neg c_2) \\ a_1 = a_2 & := & (a_1 = \mathsf{f} \wedge a_2 = \mathsf{f}) \vee (a_1 = \bot \wedge a_2 = \bot) \\ & & \vee\ (a_1 = \top \wedge a_2 = \top) \vee (a_1 = \mathsf{t} \wedge a_2 = \mathsf{t})\end{array}$$

Figure 9. Shorthands for writing containment conditions. The symbols a_1 and a_2 denote BelLog atoms, c_1 and c_2 denote containment conditions.

(1) all credentials provided as input to the PDP are associated to a single user, group, resource, and (2) there are finitely many subjects who issue credentials.

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 in the presence of failures as well. 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 $[\![S]\!]_I(r) = \mathsf{t}$.

To illustrate, we simulate the XACML 3 PDP and describe how one can find the attack described in §2. The PDP's specification is $S1$, 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 req be a request such that Ann's policy grants req, while Bob's policy denies req. Imagine that Bob's policy is authorized to give decisions for req. The PDP must therefore deny req because Ann and Bob's policies are composed using the deny-overrides operator. The following BelLog input models this scenario.

$$\begin{array}{rl}I = \{ & \textit{admin}(\mathsf{ann}) \leftarrow \mathsf{t},\ \textit{pol}(\mathsf{ann}, \mathsf{req})@\mathsf{eval} \leftarrow \mathsf{t} \\ & \textit{pol}(\mathsf{bob}, \mathsf{req})@\mathsf{eval} \leftarrow \mathsf{f},\ \textit{auth}(\mathsf{bob}, \mathsf{req})@\mathsf{check} \leftarrow \mathsf{t} \}\end{array}$$

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

$$\begin{array}{rl}I_{\mathsf{fail}} = \{ & \textit{admin}(\mathsf{ann}) \leftarrow \mathsf{t},\ \textit{eval\_pol}(\mathsf{ann}, \mathsf{req}) \leftarrow \mathsf{t} \\ & \textit{eval\_pol}(\mathsf{bob}, \mathsf{req}) \leftarrow \mathsf{f},\ \textit{check\_auth}(\mathsf{bob}, \mathsf{req}) \leftarrow \bot \}\end{array}$$

The only difference here is that the PDP fails to check whether Bob's policy is authorized for req. We observe that for this scenario the PDP grants req, i.e. $[\![S1]\!]_{I_{\mathsf{fail}}}(\mathsf{req}) = \mathsf{t}$, because the PDP's failure-handler drops Bob's policy decision. As the XACML committee has discovered, this behavior is deemed undesirable because an adversary may gain access by forcing the PDP to drop authorized policy decisions.

The 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; rather, we work with PDP's 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 $\phi_{S,r}$. Intuitively, one of the BelLog specifications in the containment problem $\phi_{P,r}$ is the PDP specification S and the other BelLog specification 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 of §2 meet their fail-security requirements.

For the purposes of this paper, we have implemented a prototype analysis tool for deciding BelLog's domain containment; the tool is available at `http://goo.gl/JzKKxk`. The tool works by translating a given domain policy containment problem into a propositional validity problem that is output in the SMT 2.0 format. Afterward, it uses the Z3 SMT solver [13] to decide propositional validity.

Authorization in Web Applications. Consider the PDP specification $S2$ 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 verify whether $S2$ meets *FR1*, we first write a condition that is satisfied by the inputs for which the PDP cannot compute an access decision due failures. Since the ACLs are composed with the permit-override 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

$$\begin{aligned} c_{\mathsf{error}} = \neg\Big(&\big(\mathit{isGranted}(U,O)@\mathsf{acl_1} = \mathsf{t} \ \vee \cdots \\ &\qquad\vee \mathit{isGranted}(U,O)@\mathsf{acl_n} = \mathsf{t}\big)\vee \\ &\big(\mathit{isGranted}(U,O)@\mathsf{acl_1} = \mathsf{f} \wedge \cdots \\ &\qquad\wedge \mathit{isGranted}(U,O)@\mathsf{acl_n} = \mathsf{f}\big)\Big) \ . \end{aligned}$$

We then construct the BelLog specification R_{error}:

$$R_{\mathsf{error}} = \{\mathit{pol}(U,O) \ \leftarrow \ (\mathit{isGranted}(U,O)@\mathsf{def} \wedge \mathit{logging}\} \ .$$

The specification R_{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 $S2$ meets *FR1*, we formulate the containment problem

$$c_{\mathsf{error}} \ \Rightarrow \ S = R_{\mathsf{error}} \ .$$

Our analysis tool shows that the specification $S2$ violates the requirement *FR1* for the PDP input

$$\begin{aligned} I = \{ \ &\mathit{isGranted}(\mathsf{ann},\mathsf{file})@\mathsf{acl_1} \leftarrow \bot, \\ &\mathit{isGranted}(\mathsf{ann},\mathsf{file})@\mathsf{acl_2} \leftarrow \mathsf{t}, \\ &\mathit{isGranted}(\mathsf{ann},\mathsf{file})@\mathsf{def} \leftarrow \mathsf{f}, \} \ . \end{aligned}$$

$S2$ violates *FR1* because it denies the request $\mathit{pol}(\mathsf{ann},\mathsf{file})$ 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 (S4):

$$\begin{aligned} \mathit{pol}(U,O) \leftarrow &\big(\mathit{isGranted}(U,O)@\mathsf{acl_1} \vee \cdots \\ &\qquad \vee \mathit{isGranted}(U,O)@\mathsf{acl_n}\big) \\ &\qquad\quad \blacktriangleright \ \big(\mathit{isGranted}(U,O)@\mathsf{def} \wedge \mathit{logging}\big) \end{aligned}$$

```
isAuthorized(User u, Object o, ListAclId ids)
  error = false
  for (id in ids)
    try
      if (readAcl(id).grants(u,o)) then return true
    catch (NotFoundException e)
      error = true
  if error
    return def.grants(u,o) && logger.on()
  return false
```

Figure 10. A PDP object that meets *FR1*.

Our tool shows that $S4$ 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 on a machine with a quad-core i7-4770 CPU and 32GB of RAM. 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 $S4$ 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 of them (recorded in the `error` variable).

Authorizations in Grids. Consider the PDP specification $S3$ and the fail-security requirement *FR2*, which states that *when, due to failures, the PDP cannot check whether a subject has at least one non-revoked delegation chain, the PDP grants access if the subject is an owner's direct delegate, and denies access otherwise.*

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

$$\begin{aligned} R_{\mathsf{chain}} = \{ \ &\mathit{chain}(X) \leftarrow \mathit{owner}(X) \\ &\mathit{chain}(X) \leftarrow \mathit{chain}(Y) \wedge Y{:}\mathit{delegate}(X) \\ &\qquad\qquad \wedge \neg Y{:}\mathit{revoke}(X)@\mathsf{rev} \qquad \} \ . \end{aligned}$$

Given a subject X, $\mathit{chain}(X)$ is: (1) t 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) f if X has no chains or the PDP checks that X has only revoked chains. We use the containment condition

$$\begin{aligned} c_{\mathsf{direct}} = (\exists Y.\ \mathit{owner}(Y) = \mathsf{t} \wedge Y{:}\mathit{delegate}(X) = \mathsf{t} \ \wedge \\ Y{:}\mathit{revoke}(X)@\mathsf{rev} \neq \mathsf{t}) \ , \end{aligned}$$

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

$$\begin{aligned} &c_{\mathsf{direct}} \ \Rightarrow \ S = R_{\mathsf{direct}} \ , \text{ where} \\ &R_{\mathsf{direct}} = R_{\mathsf{chain}} \cup \{\mathit{pol}(X) \leftarrow \mathit{chain}(X) \blacktriangleright \mathsf{t}\} \ . \end{aligned}$$

The condition c_{direct} restricts PDP inputs to direct delegates and R_{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_{direct} conflates $\perp$ and t into the grant decision using $\blacktriangleright \mathsf{t}$.

We formulate the second problem as

$$(\neg c_{\mathsf{direct}}) \Rightarrow S = R_{\mathsf{non-direct}}\ , \text{ where}$$
$$R_{\mathsf{non-direct}} = R_{\mathsf{chain}} \cup \{pol(X) \leftarrow chain(X) \blacktriangleright \mathsf{f}\}\ .$$

The condition $\neg c_{\mathsf{direct}}$ restricts PDP inputs to non-direct delegates and revoked direct delegates, and $R_{\mathsf{non-direct}}$ 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, $R_{\mathsf{non-direct}}$ conflates $\perp$ and f into the deny decision using $\blacktriangleright \mathsf{f}$.

Our analysis tool shows that the PDP specification $S3$ does not meet *FR2* because the problem $(\neg c_{\mathsf{direct}}) \Rightarrow S = R_{\mathsf{non-direct}}$ is answered negatively. The tool outputs the following PDP input:

$$I = \{\ owner(\mathsf{piet}) \leftarrow \mathsf{t}; \quad \mathsf{piet}{:}delegate(\mathsf{ann}) \leftarrow \mathsf{t};$$
$$\mathsf{piet}{:}revoke(\mathsf{ann})@\mathsf{rev} \leftarrow \perp; \quad \mathsf{ann}{:}delegate(\mathsf{fred}) \leftarrow \mathsf{t};$$
$$\mathsf{ann}{:}revoke(\mathsf{fred})@\mathsf{rev} \leftarrow \mathsf{f}\ \}\ .$$

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 *FR2*. The adversary Fred can exploit this unintended grant decision by preventing the PDP from checking whether the owner's delegation to Ann is revoked.

To meet *FR2*, we modify the specification as follows.

Specification 5 (S5):

$$pol(X) \leftarrow grant(X) \blacktriangleright \big(owner(Y) \wedge Y{:}delegate(X) \wedge$$
$$(\neg(Y{:}revoke(X)@\mathsf{rev}))\big)$$
$$grant(X) \leftarrow owner(X)$$
$$grant(X) \leftarrow grant(Y) \wedge Y{:}delegate(X) \wedge (\neg Y{:}revoke(X)@\mathsf{rev})$$

In the specification $S3$, errors are not propagated through the delegation chains. In contrast, the specification $S5$ does propagate errors through the delegation chain and thus denies access to subjects who are not direct delegates and do not have a non-revoked chain. A 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 Policy $S5$ meets *FR2* for all PDP inputs in a fixed policy domain with seven constants; the verification takes 266.29 seconds. Our tool did not terminate in a reasonable time for larger domains.

We remark that domain containment gives weaker security 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 containment 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 less than three constants.

In addition to the aforementioned requirements, 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 via 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

$$pol(X) \leftarrow tmp(X) \blacktriangleright \mathsf{f}\ ,$$

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

6. SUMMARY AND FUTURE WORK

We have initiated the study of how failure handlers affect PDP's access decisions, and 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 an industrial physical access control system. 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. REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. In *CSF*, pages 203–217. IEEE Computer Society, 2009.

[3] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language. *Journal of Computer Security*, pages 619–665, 2010.

[4] Moritz Y. Becker, Alessandra Russo, and Nik Sultana. Foundations of Logic-Based Trust Management. In *Proceedings of the Symposium on Security and Privacy*, pages 161–175, 2012.

[5] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible Trust Management, Applied to Electronic Health Records. In *Proceedings of the 17th Workshop on Computer Security Foundations*, pages 139–154, 2004.

[6] N. D. Belnap. A Useful Four-Valued Logic. In *Modern Uses of Multiple-Valued Logic*. D. Reidel, 1977.

[7] Bob Blakley and Craig Heath. Security Design Patterns. Technical report, The Open Group, 2004.

[8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704 (Informational), 1999.

[9] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173, 1996.

[10] Glenn Bruns and Michael Huth. Access Control via Belnap Logic: Intuitive, Expressive, and Analyzable Policy Composition. *Transactions on Information and System Security*, pages 1–27, 2011.

[11] Network Admission Control Configuration Guide Cisco IOS Release 15MT. `http://www.cisco.com/en/US/docs/ios-xml/ios/sec_usr_nac/configuration/15-mt/sec-usr-nac-15-mt-book.pdf`.

[12] Jason Crampton and Michael Huth. An Authorization Framework Resilient to Policy Evaluation Failures. In *Proceedings of the 15th European Conference on Research in Computer Security*, pages 472–487, 2010.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] John DeTreville. Binder, a Logic-Based Security Language. In *Proceedings of the Symposium on Security and Privacy*, pages 105–113, 2002.

[15] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[16] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and Change-impact Analysis of Access-control Policies. In *Proceedings of the 27th International Conference on Software Engineering*, pages 196–205. ACM, 2005.

[17] Simone Frau and Mohammad Torabi Dashti. Integrated Specification and Verification of Security Protocols and Policies. In *Proceedings of the Computer Security Foundations Symposium*, pages 18 –32, 2011.

[18] Deepak Garg and Frank Pfenning. Stateful authorization logic - proof theory and a case study. *Journal of Computer Security*, 20(4):353–391, 2012.

[19] Yuri Gurevich and Itay Neeman. DKAL: Distributed-Knowledge Authorization Language. In *Proceedings of the 21st Computer Security Foundations Symposium*, pages 149–162, 2008.

[20] Arno Haase. Java Idioms: Exception Handling. In *Proceedings of the 7th European Conference on Pattern Languages of Programs*, pages 41–70, 2002.

[21] Michael Howard, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* McGraw Hill, 2009.

[22] KABA KES-2200. `http://www.kaba.co.nz/Products-Solutions/Access-Control/Electric-Locking/34392-32446/electric-strikes.html`.

[23] Kerberos 5, Release 1.2.8. http://web.mit.edu/kerberos/www/krb5-1.2/krb5-1.2.8/.

[24] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing Web Access Control Policies. In *Proceedings of the 16th International Conference on World Wide Web*, pages 677–686. ACM, 2007.

[25] B.S. Lerner, S. Christov, L.J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise. Exception Handling Patterns for Process Modeling. *IEEE Transactions on Software Engineering*, pages 162–183, 2010.

[26] Ninghui Li, J.C. Mitchell, and W.H. Winsborough. Design of a Role-based Trust-management Framework. In *Proceedings of the Symposium on Security and Privacy*, pages 114–130, 2002.

[27] Disabling Firewall Service Lockdown due to Logging Failures. http://technet.microsoft.com/en-us/library/cc302466.aspx.

[28] OpenSSO Enterprise 8.0, Authentication Service Failover. `http://docs.oracle.com/cd/E19681-01/820-3885/gbarl/index.html`.

[29] Red Hat 6.5, 2.8.2.1 Firewall Configuration Tool. `http://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/`.

[30] Spring Security. `http://projects.spring.io/spring-security/`.

[31] Herbert H. Thompson, James A. Whittaker, and Florence E. Mottay. Software Security Vulnerability Testing in Hostile Environments. In *Proceedings of the Symposium on Applied Computing*, pages 260–264, 2002.

[32] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin. Decentralized Composite Access Control. In *Principles of Security and Trust*, pages 245–264, 2014.

[33] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, 2002.

[34] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, 1997.

[35] IBM WebSphere. `http://www-01.ibm.com/software/websphere/`.

[36] eXtensible Access Control Markup Language (XACML) Version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cd-03-en.html`.

[37] XACML Failure Handling Flaw. `https://lists.oasis-open.org/archives/xacml/200703/doc00000.doc`.

[38] eXtensible Access Control Markup Language (XACML) Version 2.0. `http://www.oasis-open.org/committees/tc_home.php`.