

# SecPAL: Design and Semantics of a Decentralized Authorization Language

Moritz Y. Becker   Cédric Fournet   Andrew D. Gordon

Speaker: Brian Wongchaowart  
September 16, 2009

# SecPAL versus Cassandra

SecPAL can be described as a successor to Cassandra, but there are important differences between the two.

- Cassandra supports distributed query evaluation with automated credential retrieval, while a SecPAL query is evaluated against a local assertion context (authorization policy and imported credentials).
- The answer to a Cassandra query is a set of constraints, while a SecPAL query returns a finite set of substitutions of constants for variables.

Discussion: Why do you think that the designers of SecPAL chose not to support these Cassandra features?

# Facets of SecPAL

- SecPAL provides a readable, English-like language for policy assertions and authorization queries. (Note that assertions and authorization queries have different syntaxes with different expressive power.)
- SecPAL provides a set of safety conditions that guarantee that query evaluation will be sound, complete, and tractable.
- SecPAL specifies a deterministic evaluation algorithm for queries based on translation into Datalog with constraints.

# Syntax of SecPAL Policy Assertions

- A SecPAL policy is a set of assertions of the form  $A$  says :  $fact$  if  $fact_1, \dots, fact_n$  where  $c$ .
- The issuer  $A$  must be a constant.
- Each fact consists of a subject and a verb phrase.
- Verb phrases use application-specific predicates written in infix notation.
- The only requirement on constraints is that the validity of ground constraints must be decidable in polynomial time.
- An assertion is either locally defined by the policy or can be imported in a credential.

# Grammar for Facts

$e$	$::=$	$x$	(variables)
		$A$	(constants)
$pred$	$::=$	can read [-]	(predicates)
		has access from [-] till [-]	
		...	
$D$	$::=$	0	(no re-delegation)
		$\infty$	(with re-delegation)
$verbphrase$	$::=$	$pred e_1 \dots e_n$	for $n = Arity(pred)$
		can say <sub>D</sub> fact	(delegation)
		can act as $e$	(principal aliasing)
$fact$	$::=$	$e verbphrase$	

# Delegation

- The special verb phrases “can say<sub>∞</sub>,” “can say<sub>0</sub>,” and “can act as” have built-in semantics.
- They allow one principal to delegate authority to another principal and export this delegation as a credential.
- If  $A$  says  $B$  can say<sub>∞</sub>  $fact$  and  $B$  says  $fact$  are deducible, then  $A$  says  $fact$  is deducible.
- $B$  can redelegate with  $B$  says  $C$  can say<sub>∞</sub>  $fact$ . This means that  $A$  says  $fact$  if  $C$  says  $fact$ .
- If  $A$  says  $B$  can say<sub>0</sub>  $fact$ , then  $B$  is not allowed to redelegate.
- $A$  says  $B$  can act as  $C$  means that whenever  $A$  says  $C$   $verbphrase$ , then  $A$  says  $B$   $verbphrase$ .

# Semantics of SecPAL Policy Assertions

$$\text{(cond)} \frac{\begin{array}{l} (A \text{ says } fact \text{ if } fact_1, \dots, fact_k, c) \in \mathcal{AC} \\ \mathcal{AC}, D \models A \text{ says } fact_i \theta \text{ for all } i \in \{1..k\} \\ \models c\theta \quad \quad \quad \text{vars}(fact\theta) = \emptyset \end{array}}{\mathcal{AC}, D \models A \text{ says } fact\theta}$$

$$\text{(can say)} \frac{\begin{array}{l} \mathcal{AC}, \infty \models A \text{ says } B \text{ can say}_D fact \\ \mathcal{AC}, D \models B \text{ says } fact \end{array}}{\mathcal{AC}, \infty \models A \text{ says } fact}$$

$$\text{(can act as)} \frac{\begin{array}{l} \mathcal{AC}, D \models A \text{ says } B \text{ can act as } C \\ \mathcal{AC}, D \models A \text{ says } C \text{ verbphrase} \end{array}}{\mathcal{AC}, D \models A \text{ says } B \text{ verbphrase}}$$

# Translation into Datalog

**Example 7.3.** For example, the assertion

$A \text{ says } B \text{ can say}_\infty y \text{ can say}_0 C \text{ can read } z \text{ if } y \text{ can read } \text{Foo}$

is translated into

$A \text{ says}_k B \text{ can say}_\infty y \text{ can say}_0 C \text{ can read } z \leftarrow A \text{ says}_k y \text{ can read } \text{Foo}$

$A \text{ says}_\infty y \text{ can say}_0 C \text{ can read } z \leftarrow$

$x \text{ says}_\infty y \text{ can say}_0 C \text{ can read } z ,$

$A \text{ says}_\infty x \text{ can say}_\infty y \text{ can say}_0 C \text{ can read } z$

$A \text{ says}_\infty C \text{ can read } z \leftarrow$

$x \text{ says}_0 C \text{ can read } z ,$

$A \text{ says}_\infty x \text{ can say}_0 C \text{ can read } z$

# SecPAL Assertion Safety

- Recall that the only requirement on the constraint domain is that the validity of a ground constraint must be decidable in polynomial time.
- The goal of SecPAL's assertion safety rules is to ensure that constraints are ground at runtime when they have to be evaluated.
- A ground constraint is simply equivalent to `true` or `false`.

# SecPAL Assertion Safety

A fact that includes “can say” is *nested*; otherwise, it is *flat*.

An assertion  $A$  says :  $fact$  if  $fact_1, \dots, fact_n$  where  $c$  is safe if:

- the conditional facts  $fact_1, \dots, fact_n$  are flat
- all variables in  $c$  also occur somewhere else in the assertion
- if  $fact$  is flat, all variables in  $fact$  also occur in a conditional fact

# Authorization Queries

- Upon receiving an access request, a service using SecPAL looks up an authorization query in an authorization query table and then executes this query against the local assertion context.
- The assertion context must include all credentials required to support the request (e.g., credentials submitted by the requester).
- The result of query evaluation is a set of substitutions that map variables in the query to constants.

# Syntax of SecPAL Authorization Queries

$q ::= e \text{ says } fact$	(atomic query)
$q_1, q_2$	(conjunction)
$q_1 \text{ or } q_2$	(disjunction)
$\text{not}(q)$	(negation)
$c$	(constraint)
$\exists x(q)$	(existential quantification)

- Conjunctions, disjunctions, negations, constraints, and existential quantification are permitted.
- Discussion: what about universal quantification?

# Authorization Query Evaluation

$$\text{AuthAns}_{\mathcal{AC}}(e \text{ says } fact) = \text{Answers}_p(e \text{ says}_\infty fact, \emptyset)$$

$$\text{AuthAns}_{\mathcal{AC}}(q_1, q_2) = \{\theta_1\theta_2 \mid \theta_1 \in \text{AuthAns}_{\mathcal{AC}}(q_1) \text{ and } \theta_2 \in \text{AuthAns}_{\mathcal{AC}}(q_2\theta_1)\}$$

$$\text{AuthAns}_{\mathcal{AC}}(q_1 \text{ or } q_2) = \text{AuthAns}_{\mathcal{AC}}(q_1) \cup \text{AuthAns}_{\mathcal{AC}}(q_2)$$

$$\text{AuthAns}_{\mathcal{AC}}(\text{not}(q)) = \begin{cases} \{\varepsilon\} & \text{if } \text{vars}(q) = \emptyset \text{ and } \text{AuthAns}_{\mathcal{AC}}(q) = \emptyset \\ \emptyset & \text{if } \text{vars}(q) = \emptyset \text{ and } \text{AuthAns}_{\mathcal{AC}}(q) \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{AuthAns}_{\mathcal{AC}}(c) = \begin{cases} \{\varepsilon\} & \text{if } \models c \\ \emptyset & \text{if } \text{vars}(c) = \emptyset \text{ and } \not\models c \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{AuthAns}_{\mathcal{AC}}(\exists x(q)) = \{\theta_{-x} \mid \theta \in \text{AuthAns}_{\mathcal{AC}}(q)\}$$

# Authorization Query Safety

$$\frac{\text{fact is flat}}{I \Vdash e \text{ says fact} : \text{vars}(e \text{ says fact}) - I} \qquad \frac{\text{vars}(c) \subseteq I}{I \Vdash c : \emptyset}$$
$$\frac{I \Vdash q_1 : O_1 \quad I \Vdash q_2 : O_2}{I \Vdash q_1 \text{ or } q_2 : O_1 \cap O_2} \qquad \frac{I \Vdash q : O \quad \text{vars}(q) \subseteq I}{I \Vdash \text{not}(q) : \emptyset}$$
$$\frac{I \Vdash q_1 : O_1 \quad I \cup O_1 \Vdash q_2 : O_2}{I \Vdash q_1, q_2 : O_1 \cup O_2} \qquad \frac{I \Vdash q : O \quad x \notin I}{I \Vdash \exists x(q) : O - \{x\}}$$

- An authorization query  $q$  is safe if and only if there exists a set of variables  $O$  such that  $\emptyset \Vdash q : O$ .
- Note that only flat facts can occur in an authorization query, ensuring that “can say” goals are always ground at runtime.

# Query Evaluation in Cassandra

$E_{loc} \diamond \text{RESOLVE-CLAUSE}(E_{req}, \text{root}(P_0; c_0))$

1 **foreach**  $R \equiv P_0 \leftarrow \vec{P}, c \in \mathcal{P}$  such that  $c_0 \wedge c$  is satisfiable **do**

2     **if**  $R$  is an aggregation rule **then**

3          $E_{loc} \diamond \text{AGGREGATE}(E_{req}, (P_0, c_0), R)$

4     **else if**  $E_{req} = E_{loc}$  **then**

5          $E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \vec{P}; c_0 \wedge c))$

6     **else**

7          $E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((P_0, c_0); [\text{canReqCred}(E_{req}, P_0), \vec{P}]; c_0 \wedge c))$

# Query Evaluation in Cassandra

```
 $E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \vec{P}, c_1))$   
1 if  $\vec{P} = []$  then  
2   foreach satisfiable  $c \in \exists_{-P_0}(c_1)$  do  
3      $E_{req} \diamond \text{PROCESS-ANSWER}(\text{ans}((P_0, c_0); c))$   
4 else  
5   foreach satisfiable  $c \in \exists_{-P_1}(c_1)$  do  
6      $E_{loc} \diamond \text{PROPAGATE-ANSWER}(E_{req}, \text{goal}((P_0, c_0); (P_1, c); \vec{P}; c_1))$ 
```

# Query Evaluation in Cassandra

$E_{loc} \diamond \text{PROCESS-ANSWER}(\text{ans}(P_0, c_0), c)$

- 1 **if**  $c$  is not subsumed by a constraint in  $E_{loc} \diamond \text{Ans}(P_0, c_0)$  **then**
- 2      $E_{loc} \diamond \text{Ans}(P_0, c_0) := E_{loc} \diamond \text{Ans}(P_0, c_0) \cup \{c\}$ ;
- 3     **foreach**  $(E_{req}, \text{goal}((Q_0, d_0); (P_0, d); \vec{Q}; d_1)) \in E_{loc} \diamond \text{Wait}(P_0, c_0)$
- 4         such that  $c \wedge d_1$  is satisfiable **do**
- 5          $E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((Q_0, d_0); \vec{Q}; c \wedge d_1))$

# Query Evaluation in Cassandra

```
 $E_{loc} \diamond \text{PROPAGATE-ANSWER}(E_{req}, \text{goal}((P_0, c_0); (P_1, d_0); \vec{P}; c_1))$   
1 if there exists  $(P_1, d_1) \in \text{Dom}(E_{loc} \diamond \text{Ans})$  such that  $d_0 \Rightarrow d_1$  then  
2    $E_{loc} \diamond \text{Wait}(P_1, d_1) :=$   
3      $E_{loc} \diamond \text{Wait}(P_1, d_1) \cup (E_{req}, \text{goal}((P_0, c_0); (P_1, d_0); \vec{P}; c_1));$   
4   foreach  $a \in \text{Ans}(P_1, d_1)$  such that  $a \wedge c_1$  is satisfiable do  
5      $E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \vec{P}; a \wedge c_1))$   
6 else  
7    $E_{loc} \diamond \text{Ans}(P_1, d_0) := \emptyset;$   
8    $E_{loc} \diamond \text{Wait}(P_1, d_0) := \{(E_{req}, \text{goal}((P_0, c_0); (P_1, d_0); \vec{P}; c_1))\};$   
9    $\text{Loc}(P_1, d_0) \diamond \text{RESOLVE-CLAUSE}(E_{loc}, \text{root}(P_1; d_0))$ 
```

# Atomic Query Evaluation in SecPAL

```
RESOLVE-CLAUSE( $\langle P \rangle$ )  
   $Ans(P) := \emptyset$ ;  
  foreach  $(Q \leftarrow \vec{Q}, c) \in \mathcal{P}$  do  
    if  $nd = resolve(\langle P; Q :: \vec{Q}; c; Q; []; Cl \rangle, P)$   
      exists then  
        PROCESS-NODE( $nd$ )  
  
PROCESS-ANSWER( $nd$ )  
  match  $nd$  with  $\langle P; []; c; -; -; - \rangle$  in  
    if  $nd \notin Ans(P)$  then  
       $Ans(P) := Ans(P) \cup \{nd\}$ ;  
    foreach  $nd' \in Wait(P)$  do  
      if  $nd'' = resolve(nd', nd)$  exists then  
        PROCESS-NODE( $nd''$ )
```

# Atomic Query Evaluation in SecPAL

```
PROCESS-NODE( $nd$ )  
  match  $nd$  with  $\langle P; \vec{Q}; c; ; ; - \rangle$  in  
    if  $\vec{Q} = []$  then  
      PROCESS-ANSWER( $nd$ )  
    else match  $\vec{Q}$  with  $Q_0 :: -$  in  
      if there exists  $Q'_0 \in \text{dom}(Ans)$   
        such that  $Q_0 \preceq Q'_0$  then  
           $Wait(Q_0) := Wait(Q'_0) \cup \{nd\};$   
          foreach  $nd' \in Ans(Q'_0)$  do  
            if  $nd'' = \text{resolve}(nd, nd')$  exists then  
              PROCESS-NODE( $nd''$ )  
      else  
         $Wait(Q_0) := \{nd\};$   
        RESOLVE-CLAUSE( $\langle Q_0 \rangle$ )
```

# Understanding Atomic Query Evaluation

$$(\{\langle P \rangle\} \uplus \text{Nodes}, \text{Ans}, \text{Wait}) \xrightarrow{\text{ResolveClause}} (\text{Nodes} \cup \text{Nodes}', \text{Ans}[P \mapsto \emptyset], \text{Wait})$$

$$\text{if } \text{Nodes}' = \{nd : Cl \equiv Q \leftarrow \vec{Q}, c \in \mathcal{P},$$

$$nd = \text{resolve}(\langle P; Q :: \vec{Q}; c; Q; []; Cl \rangle, P) \text{ exists } \}$$

$$(\{nd\} \uplus \text{Nodes}, \text{Ans}, \text{Wait}) \xrightarrow{\text{PropagateAnswer}} (\text{Nodes} \cup \text{Nodes}', \text{Ans}[P \mapsto \text{Ans}(P) \cup \{nd\}], \text{Wait})$$

$$\text{if } nd \equiv \langle P; []; \text{True}; \text{--}; \text{--} \rangle$$

$$nd \notin \text{Ans}(P)$$

$$\text{Nodes}' = \{nd'' : nd' \in \text{Wait}(P), nd'' = \text{resolve}(nd', nd) \text{ exists}\}$$

$$(\{nd\} \uplus \text{Nodes}, \text{Ans}, \text{Wait}) \xrightarrow{\text{RecycleAnswers}} (\text{Nodes} \cup \text{Nodes}', \text{Ans}, \text{Wait}[Q' \mapsto \text{Wait}(Q') \cup \{nd\}])$$

$$\text{if } nd \equiv \langle \text{--}; Q :: \text{--}; \text{--}; \text{--}; \text{--} \rangle$$

$$\exists Q' \in \text{dom}(\text{Ans}) : Q \preceq Q'$$

$$\text{Nodes}' = \{nd'' : nd' \in \text{Ans}(Q'), nd'' = \text{resolve}(nd, nd') \text{ exists}\}$$

$$(\{nd\} \uplus \text{Nodes}, \text{Ans}, \text{Wait}) \xrightarrow{\text{SpawnRoot}} (\text{Nodes} \cup \{\langle Q \rangle\}, \text{Ans}[Q \mapsto \emptyset], \text{Wait}[Q \mapsto \{nd\}])$$

$$\text{if } nd \equiv \langle \text{--}; Q :: \text{--}; \text{--}; \text{--}; \text{--} \rangle$$

$$\forall Q' \in \text{dom}(\text{Ans}) : Q \not\preceq Q'$$

# Understanding Atomic Query Evaluation

**Lemma A.11. (answer groundness)** If  $(Nodes, Ans, Wait)$  is reachable from some initial state and  $\langle P; []; c; S; \vec{nd}; Cl \rangle \in Nodes$  then  $S$  and  $c$  are ground and  $c$  is valid.

**Lemma A.12. (node invariant)** We write  $\bigcup Ans$  as short hand for  $\bigcup_{P \in dom(Ans)} Ans(P)$ . If  $(Nodes, Ans, Wait)$  is reachable from some initial state and  $\langle P; \vec{Q}; c; S; \vec{nd}; Cl \rangle \in Nodes$  with  $Cl \equiv R \leftarrow \vec{R}, d$ , then:

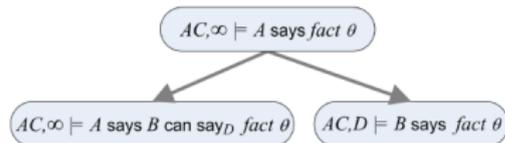
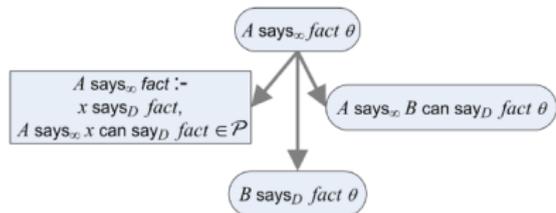
1.  $S \preceq P$ ;
2.  $Cl \in \mathcal{P}$ ;
3.  $\vec{nd} \subseteq \bigcup Ans$ ;
4. there is some  $\theta$  such that  $R\theta = S$ , and  $\vec{R}\theta = \vec{Q}' @ \vec{Q}$  (where  $\vec{Q}'$  are the answers in  $\vec{nd}$ ), and  $d\theta$  is equivalent to  $c$ .

# Strengths of SecPAL

- The SecPAL language was designed from the beginning to be easy to read and understand for users unfamiliar with formal logic.
- Usability is a critical part of security: a trust management system can be considered a security weakness if policy authors are not able to correctly express their intentions in the policy language.

# Strengths of SecPAL

- SecPAL's evaluation algorithm builds a proof tree for each answer to a query, helping users and administrators understand why an answer was returned.
- The Datalog proof graph is easily converted into a SecPAL proof graph whose semantics may be more accessible.



# Strengths of SecPAL

- SecPAL's simplicity was made possible by the insight that authorization queries can have a more expressive syntax than policy assertions without affecting the evaluation of atomic queries.
- Since authorization queries can include negation and existential quantification, policy idioms like separation of duties can be written naturally when the underlying evaluation model is just Datalog.

# Limitations of SecPAL

- SecPAL has no support for automated credential retrieval, and there is no way for a user to learn what set of credentials must be submitted along with a request without knowing the details of the service's policy.
- See Moritz Y. Becker, Jason F. Mackay, and Blair Dillaway, "Abductive Authorization Credential Gathering," IEEE International Symposium on Policies for Distributed Systems and Networks, July 2009.

# Limitations of SecPAL

- SecPAL has no explicit support for the role activations and deactivations that are central to Cassandra policies.  
DynPAL: Moritz Y. Becker, “Specification and Analysis of Dynamic Authorisation Policies,” 22nd IEEE Computer Security Foundations Symposium, July 2009.
- SecPAL’s query evaluation algorithm may not work well in a distributed setting. In particular, the left-to-right tabling resolution may exhibit poor performance if answers from remote locations have to be waited for.

# Summary

- The SecPAL language combines a readable, English-like syntax and intuitive semantic rules with a translation into Datalog with constraints for evaluation.
- Safety conditions on policy assertions and authorization queries guarantee that query evaluation remains decidable without restricting the choice of the constraint domain.
- Authorization queries are syntactically distinct from policy assertions. Conjunctions, disjunctions, negations, constraints, and existential quantification are supported without compromising the tractability of the language.

# Questions/Comments