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 $fact_1, \ldots, fact_n$ 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

\[ e ::= \begin{array}{c}
    x \quad \text{(variables)} \\
    A \quad \text{(constants)} \\
\end{array} \]

\[ pred ::= \begin{array}{c}
    \text{can read [-]} \quad \text{(predicates)} \\
    \text{has access from [-] till [-]} \\
    \ldots
\end{array} \]

\[ D ::= \begin{array}{c}
    0 \quad \text{(no re-delegation)} \\
    \infty \quad \text{(with re-delegation)}
\end{array} \]

\[ verbphrase ::= \begin{array}{c}
    pred \, e_1 \ldots \, e_n \quad \text{for } n = \text{Arity}(pred) \\
    \text{can say}_D \text{fact} \quad \text{(delegation)} \\
    \text{can act as } e \quad \text{(principal aliasing)}
\end{array} \]

\[ fact ::= e \, verbphrase \]
Delegation

- The special verb phrases “can say\(_\infty\),” “can say\(_0\),” 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\(_\infty\) fact and \(B\) says fact are deducible, then \(A\) says fact is deducible.
- \(B\) can redelegate with \(B\) says \(C\) can say\(_\infty\) fact. This means that \(A\) says fact if \(C\) says fact.
- If \(A\) says \(B\) can say\(_0\) 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

(A says fact if fact₁, ..., factₖ, c) ∈ AC
AC, D ⊢ A says factᵢθ for all i ∈ {1..k}

(\text{cond})
\begin{align*}
\models c\theta & \\
\text{vars}(fact\theta) = \emptyset & \\
\hline \\
\hline
AC, D & \models A \text{ says } fact\theta \\
\end{align*}

AC, ∞ ⊢ A says B can say₂ fact
AC, D ⊢ B says fact

(can say)
\begin{align*}
AC, D & \models B \text{ says fact} \\
\hline \\
\hline
AC, ∞ & \models A \text{ says fact} \\
\end{align*}

AC, D ⊢ A says B can act as C
AC, D ⊢ A says C \text{ verbphrase}

(can act as)
\begin{align*}
\hline \\
\hline
AC, D & \models A \text{ says B verbphrase} \\
\end{align*}
Example 7.3. For example, the assertion

\[ A \text{ says } B \text{ can say}_\infty y \text{ can say}_0 C \text{ can read } z \text{ if } y \text{ can read } F_{oo} \]

is translated into

\[ A \text{ says}_k B \text{ can say}_\infty y \text{ can say}_0 C \text{ can read } z \leftarrow A \text{ says}_k y \text{ can read } F_{oo} \]

\[ A \text{ says}_\infty y \text{ can say}_0 C \text{ can read } z \leftarrow \]
\[ x \text{ says}_\infty y \text{ can say}_0 C \text{ can read } z , \]
\[ A \text{ says}_\infty x \text{ can say}_\infty y \text{ can say}_0 C \text{ can read } z \]

\[ A \text{ says}_\infty C \text{ can read } z \leftarrow \]
\[ x \text{ says}_0 C \text{ can read } z , \]
\[ A \text{ says}_\infty x \text{ can say}_0 C \text{ can read } z . \]
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.
A fact that includes “can say” is nested; otherwise, it is flat. An assertion $A$ says: $fact$ if $fact_1, \ldots, fact_n$ where $c$ is safe if:

- the conditional facts $fact_1, \ldots, fact_n$ 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

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

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

\[
\begin{align*}
\text{AuthAns}_{AC}(e \text{ says } fact) &= \text{Answers}_\varphi(e \text{ says } fact, \emptyset) \\
\text{AuthAns}_{AC}(q_1, q_2) &= \{\theta_1\theta_2 \mid \theta_1 \in \text{AuthAns}_{AC}(q_1) \text{ and } \theta_2 \in \text{AuthAns}_{AC}(q_2\theta_1)\} \\
\text{AuthAns}_{AC}(q_1 \text{ or } q_2) &= \text{AuthAns}_{AC}(q_1) \cup \text{AuthAns}_{AC}(q_2) \\
\text{AuthAns}_{AC}(\text{not}(q)) &= \begin{cases} 
\{\varepsilon\} & \text{if } \text{vars}(q) = \emptyset \text{ and } \text{AuthAns}_{AC}(q) = \emptyset \\
\emptyset & \text{if } \text{vars}(q) = \emptyset \text{ and } \text{AuthAns}_{AC}(q) \neq \emptyset \\
\text{undefined} & \text{otherwise}
\end{cases} \\
\text{AuthAns}_{AC}(c) &= \begin{cases} 
\{\varepsilon\} & \text{if } \models c \\
\emptyset & \text{if } \text{vars}(c) = \emptyset \text{ and } \not\models c \\
\text{undefined} & \text{otherwise}
\end{cases} \\
\text{AuthAns}_{AC}(\exists x(q)) &= \{\theta_{-x} \mid \theta \in \text{AuthAns}_{AC}(q)\}
\end{align*}
\]
An authorization query $q$ is safe if and only if there exists a set of variables $O$ such that $\emptyset \models q : O$.

Note that only flat facts can occur in an authorization query, ensuring that “can say” goals are always ground at runtime.
$E_{loc} \triangle RIGHT\text{-}\text{RESOLVE}-\text{CLAUSE}(E_{req}, \text{root}(P_0; c_0))$

1. foreach $R \equiv P_0 \leftarrow \bar{P}$, $c \in \mathcal{P}$ such that $c_0 \land c$ is satisfiable do
2.   if $R$ is an aggregation rule then
3.       $E_{loc} \triangle RIGHT\text{-}\text{AGGREGATE}(E_{req}, (P_0, c_0), R)$
4.   else if $E_{req} = E_{loc}$ then
5.       $E_{loc} \triangle RIGHT\text{-}\text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \bar{P}; c_0 \land c))$
6.   else
7.       $E_{loc} \triangle RIGHT\text{-}\text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \text{canReqCred}(E_{req}, P_0); \bar{P}; c_0 \land c))$
Query Evaluation in Cassandra

\[ E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \bar{P}, c_1)) \]

1. \textbf{if} $\bar{P} = \emptyset$ \textbf{then}
2. \hspace{1em} \textbf{foreach} satisfiable $c \in \exists_{-P_0}(c_1)$ \textbf{do}
3. \hspace{2em} $E_{req} \diamond \text{PROCESS-ANSWER}(\text{ans}((P_0, c_0); c))$
4. \hspace{1em} \textbf{else}
5. \hspace{2em} \textbf{foreach} satisfiable $c \in \exists_{-P_1}(c_1)$ \textbf{do}
6. \hspace{3em} $E_{loc} \diamond \text{PROPAGATE-ANSWER}(E_{req}, \text{goal}((P_0, c_0); (P_1, c); \bar{P}; c_1))$
\[ \text{\emph{Process-Anwer}}(\text{\emph{ans}}(P_0, c_0), c) \]
1. \textbf{if} \(c\) is not subsumed by a constraint in \(E_{loc} \triangle \text{\emph{Ans}}(P_0, c_0)\) \textbf{then}
2. \(E_{loc} \triangle \text{\emph{Ans}}(P_0, c_0) := E_{loc} \triangle \text{\emph{Ans}}(P_0, c_0) \cup \{c\};\)
3. \textbf{foreach} \((E_{\text{req}}, \text{\emph{goal}}((Q_0, d_0); (P_0, d); Q; d_1)) \in E_{loc} \triangle \text{\emph{Wait}}(P_0, c_0)\)
4. \textbf{such that} \(c \land d_1\) is satisfiable \textbf{do}
5. \(E_{loc} \triangle \text{\emph{Project}}(E_{\text{req}}, \text{\emph{body}}((Q_0, d_0); Q; c \land d_1)\)
Query Evaluation in Cassandra

\[ E_{loc} \diamond \text{PROPAGATE-ANSWER}(E_{req}, \text{goal}((P_0, c_0); (P_1, d_0); \overline{P}; c_1)) \]

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. \[ E_{loc} \diamond \text{Wait}(P_1, d_1) \cup (E_{req}, \text{goal}((P_0, c_0); (P_1, d_0); \overline{P}; c_1)); \]

4. foreach \(a \in \text{Ans}(P_1, d_1)\) such that \(a \land c_1\) is satisfiable do

5. \[ E_{loc} \diamond \text{PROJECT}(E_{req}, \text{body}((P_0, c_0); \overline{P}; a \land c_1)) \]

6. else

7. \[ E_{loc} \diamond \text{Ans}(P_1, d_0) := \emptyset; \]

8. \[ E_{loc} \diamond \text{Wait}(P_1, d_0) := \{(E_{req}, \text{goal}((P_0, c_0); (P_1, d_0); \overline{P}; c_1))\}; \]

9. \[ \text{Loc}(P_1, d_0) \diamond \text{RESOLVE-CLAUSE}(E_{loc}, \text{root}(P_1; d_0)) \]
Atomic Query Evaluation in SecPAL

\textbf{Resolve-Clause}(\langle P \rangle)
\begin{align*}
\text{Ans}(P) & := \emptyset; \\
\text{foreach} \ (Q \leftarrow \tilde{Q}, c) \in \mathcal{P} \ do & \\
& \text{if } nd = \text{resolve}(\langle P; Q :: \tilde{Q}; c; Q; []; Cl \rangle, P) \\
& \quad \text{exists then} \\
& \quad \text{PROCESS-NODE}(nd)
\end{align*}

\textbf{Process-Answer}(nd)
\begin{align*}
\text{match } nd \text{ with } \langle P; []; c; \_; \_; \_ \rangle \text{ in} & \\
& \text{if } nd \notin \text{Ans}(P) \text{ then} \\
& \quad \text{Ans}(P) := \text{Ans}(P) \cup \{nd\}; \\
& \text{foreach } nd' \in \text{Wait}(P) \ do \\
& \quad \text{if } nd'' = \text{resolve}(nd', nd) \text{ exists then} \\
& \quad \quad \text{PROCESS-NODE}(nd'')
\end{align*}
Atomic Query Evaluation in SecPAL

```plaintext
PROCESS-NODE(nd)
    match nd with ⟨P; Q; c; ...⟩ in
    if Q = [] then
        PROCESS-ANSWER(nd)
    else match Q with Q₀ :: ... in
        if there exists Q’₀ ∈ dom(Ans)
            such that Q₀ ≤ Q’₀ then
                Wait(Q’₀) := Wait(Q’₀) ∪ {nd};
                foreach nd’ ∈ Ans(Q’₀) do
                    if nd” = resolve(nd, nd’)
                    exists then
                        PROCESS-NODE(nd”)
        else
            Wait(Q₀) := {nd};
            RESOLVE-CLAUSE((Q₀))
```
Understanding Atomic Query Evaluation

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

if \( \text{Nodes}' = \{\text{nd} : \text{Cl} \equiv Q \leftarrow \tilde{Q}, c \in P, \text{nd} = \text{resolve}(\langle P; Q :: \tilde{Q}; c; Q; [\ ]; \text{Cl}, P) \text{ exists} \} \)

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

if \( \text{nd} \equiv \langle P; [\ ]; \text{True}; \_; \_; \_ \rangle \)

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

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

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

if \( \text{nd} \equiv \langle \_; Q :: \_; \_; \_; \_; \_ \rangle \)

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

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

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

if \( \text{nd} \equiv \langle \_; Q :: \_; \_; \_; \_; \_ \rangle \)

\( \forall Q' \in \text{dom}(\text{Ans}) : Q \not\preceq Q' \)
Lemma A.11. (answer groundness) If \((Nodes, Ans, \text{Wait})\) is reachable from some initial state and \(\langle P; []; c; S; \tilde{n}d; Cl \rangle \in Nodes\) then \(S\) and \(c\) are ground and \(c\) is valid.

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

1. \(S \subseteq P\);

2. \(Cl \in \mathcal{P}\);

3. \(\tilde{n}d \subseteq \bigcup Ans\);

4. there is some \(\Theta\) such that \(R\Theta = S\), and \(\tilde{R}\Theta = \tilde{Q}' \odot \tilde{Q}\) (where \(\tilde{Q}'\) are the answers in \(\tilde{n}d\)), 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.

Limitations of SecPAL

- 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