


# A Framework for Concrete Reputation-Systems

## with Applications to History-Based Access Control

Karl Krukow<sup>1</sup>   Mogens Nielsen<sup>1</sup>   Vladimiro Sassone<sup>2</sup>

<sup>1</sup>BRICS, Department of Computer Science  
University of Aarhus, Denmark  


<sup>2</sup>Department of Informatics  
University of Sussex, UK  


12th ACM Conference on Computer and Communications Security  
November 7-11, 2005  
Hilton Alexandria Mark Center, Alexandria, VA, USA

# Reputation Systems

what, why, how?

- What?

- ▶ Records, aggregates and distributes information about past behaviour of principals.
- ▶ Open concurrent systems with virtually anonymous interactions.

- Why?

- ▶ Use reputation information to guide trust decisions.
- ▶ Provides incentives for well-behaving (“the shadow of the future”).

- How?

- ▶ Common characteristics, but no universal mechanism.
- ▶ Examples: eBay, Amazon,  $\beta$  reputation-system, EigenTrust, ...

# Reputation Systems

a few observations

- Use of reputation information is often not part of design.
- Heavy abstraction of behavioural information  $\rightarrow$  information loss.
  - ▶ EigenTrust:
    - ★ Single transaction: satisfactory/unsatisfactory.
    - ★ History: normalized value  $c_j \in [0, 1]$ .
  - ▶ eBay:
    - ★ Single transaction: positive, neutral or negative.
    - ★ History:  $\#_{\text{pos}} - \#_{\text{neg}}$ .
- Known exceptions are Shmatikov & Talcott [1], and our proposed framework [2].

# Properties of reputation systems

and the notion of history-based access control

## Structure of reputation-system correctness-criteria

If interaction with entity  $p$  occurs in context  $c$  at time  $t$ , then the past behaviour of  $p$  up *until* time  $t$  satisfies requirement  $\psi$ .

- One specialization: History-based Access Control.
  - ▶ Program  $p$  can access resource  $r$  at time  $t$  only if  $p$ 's execution history up until time  $t$  satisfies  $\psi$ .
- Example: Edjlali *et al.* [3].
  - ▶ Suppose you've downloaded what claims to be a new cool browser from some web page.
  - ▶ "allow program to connect to a remote site if-and-only-if it has neither tried to **open a local file that it has not created**, nor tried to **modify a file it has created**, nor tried to **create a sub-process**."

# Properties of reputation systems

and the notion of history-based access control

## Structure of reputation-system correctness-criteria

If interaction with entity  $p$  occurs in context  $c$  at time  $t$ , then the past behaviour of  $p$  up *until* time  $t$  satisfies requirement  $\psi$ .

- One specialization: History-based Access Control.
  - ▶ Program  $p$  can access resource  $r$  at time  $t$  only if  $p$ 's execution history up until time  $t$  satisfies  $\psi$ .
- Example: Edjlali *et al.* [3].
  - ▶ Suppose you've downloaded what claims to be a new cool browser from some web page.
  - ▶ "allow program to connect to a remote site if-and-only-if it has neither tried to **open a local file that it has not created**, nor tried to **modify a file it has created**, nor tried to **create a sub-process**."

# Properties of reputation systems

and the notion of history-based access control

## Structure of reputation-system correctness-criteria

If interaction with entity  $p$  occurs in context  $c$  at time  $t$ , then the past behaviour of  $p$  up *until* time  $t$  satisfies requirement  $\psi$ .

- One specialization: History-based Access Control.
  - ▶ Program  $p$  can access resource  $r$  at time  $t$  only if  $p$ 's execution history up until time  $t$  satisfies  $\psi$ .
- Example: Edjlali *et al.* [3].
  - ▶ Suppose you've downloaded what claims to be a new cool browser from some web page.
  - ▶ "allow program to connect to a remote site if-and-only-if it has neither tried to **open a local file that it has not created**, nor tried to **modify a file it has created**, nor tried to **create a sub-process**."

# Outline

- 1 Modelling behavioural information
  - Event Structures as a general model
- 2 A Simple Policy Language
  - Examples
  - History Verification
- 3 Parameters and Quantification
  - Verifying Quantified Policies

# Outline

- 1 **Modelling behavioural information**
  - Event Structures as a general model
- 2 A Simple Policy Language
  - Examples
  - History Verification
- 3 Parameters and Quantification
  - Verifying Quantified Policies

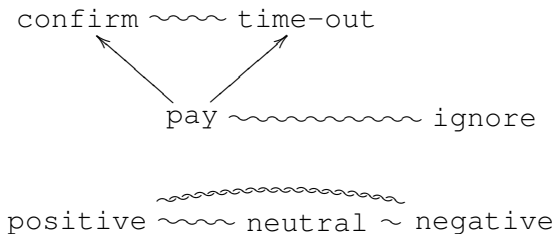
# An event-structure model

- Primary target environment: concurrent systems.
  - ▶ Entities in a concurrent system interact following protocols.
  - ▶ Behavioural information is information about a number of past protocol-runs (sessions).
- We use event structures as an abstract model of protocols.
  - ▶ A protocol is often specified as a concurrent process.
  - ▶ Event structures were invented to give formal semantics to concurrent processes.
  - ▶ More formally,  $ES = (E, \leq, \#)$ ,  $E$  a set of events,  $\leq$  and  $\#$  (causality and conflict) relations on  $E$ .

# Event structures

## configurations & example

Simple eBay example:

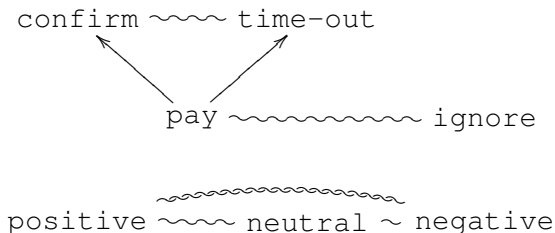


- Information about a session is a finite set of events  $x \subseteq E$ , called a **configuration** (which is always conflict free and causally closed).
- A history  $h$ , is a sequence of sessions,  $h = x_1 x_2 \cdots x_n \in \mathcal{C}_{ES}^*$ 
  - E.g.,  $h = \{\text{pay}, \text{confirm}, \text{pos}\} \{\text{pay}, \text{confirm}, \text{neu}\} \{\text{pay}\}$

# Event structures

## configurations & example

Simple eBay example:



- Information about a session is a finite set of events  $x \subseteq E$ , called a **configuration** (which is always conflict free and causally closed).
- A history  $h$ , is a sequence of sessions,  $h = x_1 x_2 \cdots x_n \in \mathcal{C}_{ES}^*$ 
  - ▶ E.g.,  $h = \{\text{pay}, \text{confirm}, \text{pos}\} \{\text{pay}, \text{confirm}, \text{neu}\} \{\text{pay}\}$

# Interface

- An entity may be involved in several concurrent protocol-runs.
- We use the following interface (where  $h = x_1 x_2 \cdots x_n$ ),
  - ▶ **update**( $e, i$ ), event  $e$  occurs in session  $i$ , where  $e \in E$  and  $i$  is a session-index.
    - ★  $h.\mathbf{update}(e, i) = x_1 x_2 \cdots (x_i \cup \{e\}) \cdots x_n$ .
  - ▶ **new**() , creates a new session.
    - ★  $h.\mathbf{new}() = h\emptyset$ .
- Hence, a stream of observations,

**new().upd( $e_1, 1$ ).new().upd( $e_2, 1$ ).upd( $e_1, 2$ ).upd( $e_3, 1$ ).upd( $e_4, 2$ )**

gets grouped into sessions, which becomes a history

$$h = \{e_1, e_2, e_3\}\{e_1, e_4\}$$

# Outline

- 1 Modelling behavioural information
  - Event Structures as a general model
- 2 A Simple Policy Language
  - Examples
  - History Verification
- 3 Parameters and Quantification
  - Verifying Quantified Policies

# Specification and Verification

## Reputation-system correctness-criteria

If interaction with entity  $p$  occurs in context  $c$  at time  $t$ , then the past behaviour of  $p$  up *until* time  $t$  satisfies requirement  $\psi$ .

- Specification language.
  - ▶ How to specify requirements  $\psi$  declaratively and expressively.
- Dynamic verification problem.
  - ▶ Given  $h$  and  $\psi$  does  $h$  satisfy  $\psi$ ?
  - ▶ Information is provided incrementally via operations: **update**( $e, i$ ) and **new**().

# Pure-Past Linear Temporal Logic

- Syntax.

$$\psi ::= e \mid \diamond e \mid \psi_0 \wedge \psi_1 \mid \psi_0 \vee \psi_1 \mid \neg \psi \mid \mathbf{X}^{-1} \psi \mid \psi_0 \mathbf{S} \psi_1$$

- Semantics: relation  $\models$  between histories  $h = x_1 x_2 \cdots x_n$  and formulas  $\psi$ .

$(h, i) \models e$	iff	$e \in x_i$
$(h, i) \models \diamond e$	iff	$e \notin x_i$
$(h, i) \models \psi_0 \wedge \psi_1$	iff	$(h, i) \models \psi_0$ and $(h, i) \models \psi_1$
$(h, i) \models \psi_0 \vee \psi_1$	iff	$(h, i) \models \psi_0$ or $(h, i) \models \psi_1$
$(h, i) \models \neg \psi$	iff	$(h, i) \not\models \psi$
$(h, i) \models \mathbf{X}^{-1} \psi$	iff	$i > 1$ and $(h, i - 1) \models \psi$
$(h, i) \models \psi_0 \mathbf{S} \psi_1$	iff	$\exists j \leq i. (h, j) \models \psi_1$ and $\forall j'. j < j' \leq i \Rightarrow (h, j') \models \psi_0$

- $h \models \psi \iff (h, n) \models \psi \quad (h \neq \epsilon)$

# Pure-Past Linear Temporal Logic

- Syntax.

$$\psi ::= e \mid \diamond e \mid \psi_0 \wedge \psi_1 \mid \psi_0 \vee \psi_1 \mid \neg \psi \mid X^{-1} \psi \mid \psi_0 \mathbf{S} \psi_1$$

- Semantics: relation  $\models$  between histories  $h = x_1 x_2 \cdots x_n$  and formulas  $\psi$ .

$(h, i) \models e$	iff	$e \in x_i$
$(h, i) \models \diamond e$	iff	$e \notin x_i$
$(h, i) \models \psi_0 \wedge \psi_1$	iff	$(h, i) \models \psi_0$ and $(h, i) \models \psi_1$
$(h, i) \models \psi_0 \vee \psi_1$	iff	$(h, i) \models \psi_0$ or $(h, i) \models \psi_1$
$(h, i) \models \neg \psi$	iff	$(h, i) \not\models \psi$
$(h, i) \models X^{-1} \psi$	iff	$i > 1$ and $(h, i - 1) \models \psi$
$(h, i) \models \psi_0 \mathbf{S} \psi_1$	iff	$\exists j \leq i. (h, j) \models \psi_1$ and $\forall j'. j < j' \leq i \Rightarrow (h, j') \models \psi_0$

- $h \models \psi \iff (h, n) \models \psi \quad (h \neq \epsilon)$

# Examples

- eBay Auction

- ▶ Policy: “only bid on auctions run by a seller that has never failed to send goods for won auctions in the past.”

$$\psi^{\text{bid}} \equiv \neg F^{-1}(\text{time-out})$$

- ▶ Furthermore, the buyer might require that “the seller has never provided negative feedback in auctions where payment was made.”

$$\psi^{\text{bid}} \equiv \neg F^{-1}(\text{time-out}) \wedge G^{-1}(\text{negative} \rightarrow \text{ignore})$$

# Examples

- eBay Auction

- ▶ Policy: “only bid on auctions run by a seller that has never failed to send goods for won auctions in the past.”

$$\psi^{\text{bid}} \equiv \neg F^{-1}(\text{time-out})$$

- ▶ Furthermore, the buyer might require that “the seller has never provided negative feedback in auctions where payment was made.”

$$\psi^{\text{bid}} \equiv \neg F^{-1}(\text{time-out}) \wedge G^{-1}(\text{negative} \rightarrow \text{ignore})$$

# An automata-based algorithm for verification

- We provide a data-structure (or dynamic algorithm)  $DS$ ,
  - ▶ initialized with an event structure  $ES$  and a policy  $\psi$ ,
  - ▶ supporting three operations:
    - ★  $DS.new()$
    - ★  $DS.update(e, i)$
    - ★  $DS.check()$   
( $h \models \psi?$ )
- $DS$  maintains
  - ▶ a finite automaton  $A_\psi = (S, C_{ES}, s_{init}, A, \delta)$  with  
 $\mathcal{L}(A_\psi) = \{h \in C_{ES}^* \mid h \models \psi\}$
  - ▶ the history  $h = x_1 x_2 \cdots x_m$  corresponding to the **update** and **new** operations performed.
  - ▶ a vector of automaton states  $v = s_1 s_2 \cdots s_m$  (same length as  $h$ ).

# An automata-based algorithm for verification

operations

$x_1$	$x_2$	$\dots$	$x_{i-1}$	$x_i$	$\dots$	$x_m$
$s_1$	$s_2$	$\dots$	$s_{i-1}$	$s_i$	$\dots$	$s_m$

$$\text{Inv.: } s_k = \hat{\delta}(s_{\text{init}}, x_1 x_2 \dots x_k)$$

- Once  $A_{\psi}$  has been constructed, the data-structure operations are simple.
  - check()** is checking if  $s_m$  is an accept state:  
 $O(1)$ .
  - new()** is computing  $s = \delta(s_m, \emptyset)$ , and extending  $v = s_1 s_2 \dots s_m s$ ,  
 $h = x_1 \dots x_m \emptyset$ :  
 $O(1)$ .
  - update**( $e, i$ ) is starting automata in state  $s_{i-1}$ , and computing  
 $s_i := \delta(s_{i-1}, x_i \cup \{e\})$ , and so on up to  $s_m = \delta(s_{m-1}, x_m)$ :  
 $O(m - i + 1)$ .

(assuming transitions are constant time)

# An automata-based algorithm for verification

operations

$x_1$	$x_2$	$\dots$	$x_{i-1}$	$x_i$	$\dots$	$x_m$
$s_1$	$s_2$	$\dots$	$s_{i-1}$	$s_i$	$\dots$	$s_m$

$$\text{Inv.: } s_k = \hat{\delta}(s_{\text{init}}, x_1 x_2 \dots x_k)$$

- Once  $A_{\psi}$  has been constructed, the data-structure operations are simple.
  - ▶ **check()** is checking if  $s_m$  is an accept state:  
 $O(1)$ .
  - ▶ **new()** is computing  $s = \delta(s_m, \emptyset)$ , and extending  $v = s_1 s_2 \dots s_m s$ ,  
 $h = x_1 \dots x_m \emptyset$ :  
 $O(1)$ .
  - ▶ **update**( $e, i$ ) is starting automata in state  $s_{i-1}$ , and computing  
 $s_i := \delta(s_{i-1}, x_i \cup \{e\})$ , and so on up to  $s_m = \delta(s_{m-1}, x_m)$ :  
 $O(m - i + 1)$ .

(assuming transitions are constant time)

# An automata-based algorithm for verification

operations

$x_1$	$x_2$	$\dots$	$x_{i-1}$	$x_i$	$\dots$	$x_m$
$s_1$	$s_2$	$\dots$	$s_{i-1}$	$s_i$	$\dots$	$s_m$

$$\text{Inv.: } s_k = \hat{\delta}(s_{\text{init}}, x_1 x_2 \dots x_k)$$

- Once  $A_{\psi}$  has been constructed, the data-structure operations are simple.
  - ▶ **check()** is checking if  $s_m$  is an accept state:  
 $O(1)$ .
  - ▶ **new()** is computing  $s = \delta(s_m, \emptyset)$ , and extending  $v = s_1 s_2 \dots s_m s$ ,  
 $h = x_1 \dots x_m \emptyset$ :  
 $O(1)$ .
  - ▶ **update**( $e, i$ ) is starting automata in state  $s_{i-1}$ , and computing  
 $s_i := \delta(s_{i-1}, x_i \cup \{e\})$ , and so on up to  $s_m = \delta(s_{m-1}, x_m)$ :  
 $O(m - i + 1)$ .

(assuming transitions are constant time)

# An automata-based algorithm for verification

operations

$x_1$	$x_2$	$\dots$	$x_{i-1}$	$x_i$	$\dots$	$x_m$
$s_1$	$s_2$	$\dots$	$s_{i-1}$	$s_i$	$\dots$	$s_m$

$$\text{Inv.: } s_k = \hat{\delta}(s_{\text{init}}, x_1 x_2 \dots x_k)$$

- Once  $A_{\psi}$  has been constructed, the data-structure operations are simple.
  - ▶ **check()** is checking if  $s_m$  is an accept state:  
 $O(1)$ .
  - ▶ **new()** is computing  $s = \delta(s_m, \emptyset)$ , and extending  $v = s_1 s_2 \dots s_m s$ ,  
 $h = x_1 \dots x_m \emptyset$ :  
 $O(1)$ .
  - ▶ **update**( $e, i$ ) is starting automata in state  $s_{i-1}$ , and computing  
 $s_i := \delta(s_{i-1}, x_i \cup \{e\})$ , and so on up to  $s_m = \delta(s_{m-1}, x_m)$ :  
 $O(m - i + 1)$ .

(assuming transitions are constant time)

# An automata-based algorithm for verification

automata  $A_\psi$

- States of  $A_\psi$  are sets of sub-formulas of  $\psi$ .
  - ▶  $O(2^{|\psi|})$
- Transition  $\delta(s, x)$  can be computed in time linear in  $\psi$ .
  - ▶ Stored and then accessed in constant time.
- Automaton  $A_\psi$  can be constructed in  $O(2^{|\psi|} \cdot |\mathcal{C}_{ES}| \cdot |\psi|)$ .
- Constructing  $A_\psi$  is a one-time computation, performed upon initialization.

# Optimizations

- When creating  $A_\psi$  we construct only reachable states.
  - ▶ A lot of states are unreachable, e.g., for each conjunction  $\psi_0 \wedge \psi_1$  states containing  $\psi_0 \wedge \psi_1$  but not both of  $\psi_0$  and  $\psi_1$ , are unused.
- Since  $A_\psi$  is a deterministic finite automaton, there is a unique minimal automaton accepting same language.
  - ▶ This captures the minimal space requirement for checking property in question.
- We don't actually need to store the entire history  $h = x_1 x_2 \cdots x_m$ , (or  $v = s_1 \cdots s_m$ ).
  - ▶ Once a prefix  $x_1 \cdots x_k$  consists of only maximal configurations, it can be summarized as a single automata state  $s_{\text{summary}}$ .

# Optimizations

- When creating  $A_\psi$  we construct only reachable states.
  - ▶ A lot of states are unreachable, e.g., for each conjunction  $\psi_0 \wedge \psi_1$  states containing  $\psi_0 \wedge \psi_1$  but not both of  $\psi_0$  and  $\psi_1$ , are unused.
- Since  $A_\psi$  is a deterministic finite automaton, there is a unique minimal automaton accepting same language.
  - ▶ This captures the minimal space requirement for checking property in question.
- We don't actually need to store the entire history  $h = x_1 x_2 \cdots x_m$ , (or  $v = s_1 \cdots s_m$ ).
  - ▶ Once a prefix  $x_1 \cdots x_k$  consists of only maximal configurations, it can be summarized as a single automata state  $s_{\text{summary}}$ .

# Optimizations

- When creating  $A_\psi$  we construct only reachable states.
  - ▶ A lot of states are unreachable, e.g., for each conjunction  $\psi_0 \wedge \psi_1$  states containing  $\psi_0 \wedge \psi_1$  but not both of  $\psi_0$  and  $\psi_1$ , are unused.
- Since  $A_\psi$  is a deterministic finite automaton, there is a unique minimal automaton accepting same language.
  - ▶ This captures the minimal space requirement for checking property in question.
- We don't actually need to store the entire history  $h = x_1 x_2 \cdots x_m$ , (or  $v = s_1 \cdots s_m$ ).
  - ▶ Once a prefix  $x_1 \cdots x_k$  consists of only maximal configurations, it can be summarized as a single automata state  $s_{\text{summary}}$ .

# Outline

- 1 Modelling behavioural information
  - Event Structures as a general model
- 2 A Simple Policy Language
  - Examples
  - History Verification
- 3 Parameters and Quantification
  - Verifying Quantified Policies

# Parameters and Quantification

- Recall example property: "... [never] open a local file that it has not created ..."
  - ▶ We want *for any* file  $f$  "if  $\text{open}(f)$  then  $F^{-1}\text{create}(f)$ ."
  - ▶ But infinitely many possible filenames
- Need a notion of *parameterized* event structure.
  - ▶ Events occur with parameters from (infinite) parameter sets.
  - ▶ Otherwise as usual event structures.
- Specify property as

$$G^{-1} \left( \forall x. \left[ \text{open}(x) \rightarrow F^{-1}(\text{create}(x)) \right] \right)$$

# Verifying quantified policies

## constraints

- We can generalize the basic algorithm.
  - ▶ Set of sub-formulas are replaced by functions from sub-formulas to so-called constraints.
  - ▶ A constraint is a finite representation of the set of substitutions that make a specific sub-formula true.
  - ▶ Constraints generalize booleans.
- Unfortunately, the verification problem is *PSPACE* complete even in single-element models.

# Verifying quantified policies

the good news. . .

- We have a result which bounds the worst-case running-time of our algorithm.
  - ▶ **update** is “only” exponential in the number of free-variables of formula  $\psi$ , with the base of the exponential being “*O*-of” the number of occurred distinct parameters.
  - ▶ In practice, if policies do not have too many variables, we are OK.
- Constraints can be efficiently represented and manipulated using Binary Decision Diagrams (BDDs).
  - ▶ Need to evaluate efficiency in practice.

# Summary

- A framework for “concrete” reputation systems and a notion of “security” (or correctness) of these systems.
- Basic policies can be declaratively specified and efficiently verified.
- Quantified policies are more expressive, and quantified model checking is decidable (though hard with many quantifiers).
- Applications in history-based access control (Java Security-Manager prototype under development).

# Thank you for your time!

-  Vitaly Shmatikov and Carolyn Talcott.  
Reputation-based trust management.  
*Journal of Computer Security*, 13(1):167–190, 2005.
-  Karl Krukow, Mogens Nielsen, and Vladimiro Sassone.  
A framework for concrete reputation-systems.  
Technical Report RS-05-23, BRICS, University of Aarhus, July 2005.
-  Guy Edjlali, Anurag Acharya, and Vipin Chaudhary.  
History-based access control for mobile code.  
In *Proceedings from the 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 38–48. ACM Press, 1998.