Policy Framings for Access Control

Massimo Bartoletti
Dipartimento di Informatica
Università di Pisa, Italy
bartolet@di.unipi.it

Pierpaolo Degano
Dipartimento di Informatica
Università di Pisa, Italy
degano@di.unipi.it

Gian Luigi Ferrari
Dipartimento di Informatica
Università di Pisa, Italy
giangi@di.unipi.it

ABSTRACT

A new model for access control is proposed, based on policy framings embedded into histories of execution. This allows for policies that have a possibly nested, local scope. In spite of the increased expressive power of our model, we present a way to use standard model checking for history verification.

1. INTRODUCTION

Models and techniques for language based security are receiving increasing attention [?, ?]. Among these, access control plays a relevant role [?]. Indeed, modern programming languages feature access control policies and mechanisms as design principles.

Access control *policies* specify the rules by which principals are authorized to access some protected objects or resources; while *mechanism* will implement the controls imposed by the given policy. For example, a policy may specify that a principal P can never read a certain file F. This policy can be enforced by a trusted component of the operating system, that intercepts any access to F and prevents P from reading.

Several models for access control have been proposed, among which *stack inspection*, adopted by Java and C♯. In this model, a policy grants static access rights to code, while actual run-time rights depend on the static rights of the code frames on the call stack. As access controls only rely on the current state of the calling sequence, stack inspection may be insecure, when trusted code depends on results supplied by untrusted code [?]. In fact, access controls are insensitive to the frame of an untrusted code, when popped from the call stack. Additionally, some standard code optimizations (e.g. method inlining) may break security in the presence of stack inspection.

The main weaknesses of stack inspection are caused by the fact that the call stack only records a fragment of the whole execution. *History-based access control* considers instead (a suitable abstraction of) the entire execution, and the actual rights of the running code depend on the static rights of *all* the pieces of code (possibly partially) executed so far. History-based access control has been recently receiving major attention, both at the foundational level [?, ?, ?] and at the implementation level [?, ?].

The typical run-time mechanisms for enforcing history-based policies are *reference monitors*, which observe program executions and abort them whenever about to violate the given policy. The observations are called *events*, and are an abstraction of security-relevant activities (e.g. opening a socket connection, reading and writing to a file). Sequences of events, possibly infinite, are called *histories*. Usually, the security policy of the monitor is a global property: it is an invariant that must hold at any point of the execution. Reference monitors have been proved to enforce exactly the class of *safety* properties [?].

Checking each single event in a history may be inefficient. A different approach is to instrument the code with *local checks* (see e.g. Java and C♯), each enforcing its own local policy. Under certain circumstances, the two ways are equivalent [?, ?]. Recently, Skalka and Smith [?] have addressed the problem of history-based access control with local checks, combining a static technique with model checking. In their approach, local checks enforce ω-regular properties of histories. These properties are written as μ-calculus logic formulae, verified by Büchi automata. From a given program, their type and effect system extracts a history expression, i.e. an over-approximate, finite representation of all the histories the program can generate. History expressions are then model checked to (statically) guarantee that each local check will always succeed at run-time. If so, all the local checks can be safely removed from the program.

We propose here a novel approach to history-based access control, along the line of [?]. We assume that policies are *ω-regular* properties of histories. We extend the notion of history to comprise policies with local scope. For example, suppose we have a history $\varphi[\alpha_0\alpha_1]$, made of two events in sequence, enclosed in a policy framing $\varphi[\cdots]$. This history is valid when both α_0 and $\alpha_0\alpha_1$ satisfy φ, to reflect that all partial computations (i.e. the prefixes of the history) must satify the enclosing policy. Similarly, suppose we have $\varphi'[\alpha_2]$. We can now concatenate the two histories and obtain $\varphi[\alpha_0\alpha_1]\varphi'[\alpha_2]$, expressing that α_0 and $\alpha_0\alpha_1$ both obey φ, while $\alpha_0\alpha_1\alpha_2$ only obeys φ'. This reflects our intuition that programs must be prevented from hiding events in the past: as a consequence, security policies inspect the *whole* history of execution. An additional feature of our model is that scopes can be nested, e.g. $\varphi''[\varphi[\alpha_0\alpha_1]\varphi'[\alpha_2]]$ in which α_0, $\alpha_0\alpha_1$ and $\alpha_0\alpha_1\alpha_2$ must also respect the policy φ''.

Even though policies are ω-regular properties, the nesting of policy framings may give rise to *non-regular* properties: indeed, every history η must obey to the conjunction of all the policies within the scope of which the last event of η occurs. A run-time mechanism enforcing this kind of properties needs to be at least as powerful as pushdown automata.

Note in passing that ω-regular local checks can be easily recovered in our model: the history $\varphi[\alpha]$ corresponds to a local check of the policy φ after the event α.

Furthermore, we enrich the history expressions of [?] with policy framings (in [?], we introduce a linguistic level and a static analysis to extract history expressions from programs). However, the model checking techniques of [?] cannot be straightforwardly imported in our model, because of the inherent non-regularity of the enforced policies.

The main technical contribution of this paper is showing that also histories with nested policy framings can be verified using standard model checking techniques. We define a transformation that, given an history expression H, obtains an expression H' such that (i) the histories represented by H' are regular, and (ii) they respect exactly the same policies (within their scopes) obeyed by the histories represented by H. From the history expression H' we then extract a Basic Process Algebra process p and a regular formula φ such that H' is valid if and only if p satisfies φ. This satisfiability problem is known to be decidable by model checking [?].

2. A MOTIVATING EXAMPLE

To illustrate our approach, consider a simple web browser that displays HTML pages and runs applets. If an applet is trusted (e.g. because downloaded from a trusted site), then it is executed with full privileges; otherwise, the applet is run in a policy framing φ, enforcing the following property: the applet cannot perform a write operation after it has read from the local disk. We define the browser as a function that processes the URL given as input (we assume that URLs represent both applets and HTML pages). This behaviour is implemented by the following program, written à la ML:

```
let Browser = fun url
    if is_html(url) then display(url)
    else if is_applet(url) then
        if is_trusted(url)
            then url;
            else φ[url]
```

Now consider a trusted applet that reads some data from the local disk, and then writes that data to a remote server via a socket connection. We represent this applet as a sequence of two events, that we call `read` and `write`.

```
let TrustedApplet = read; write
```

The behaviour of the browser applied to the trusted applet, i.e. the program `Browser TrustedApplet`, is illustrated by the following trace:

$$\begin{array}{ll}
 & (\varepsilon,\ \texttt{Browser TrustedApplet})\\
\to & (\varepsilon,\ \texttt{TrustedApplet})\\
\to & (\texttt{read},\ \texttt{write})\\
\to & (\texttt{read write},\ \texttt{skip})
\end{array}$$

where $\to$ is a transition of the operational semantics, and the program states are pairs, whose first component is a history (ε stand for the empty history), and the second one is the program continuation (see [?] for details).

Now consider an untrusted applet that attempts to exploit the privileges of the `TrustedApplet` as follows:

```
let UnknownApplet = Browser TrustedApplet
```

The behaviour of the program `Browser UnknownApplet` is represented by the following trace, where the history event $[_\varphi$ stands for entering in the scope of the policy φ:

$$\begin{array}{ll}
 & (\varepsilon,\ \texttt{Browser UnknownApplet})\\
\to & (\varepsilon,\ \varphi[\texttt{UnknownApplet}])\\
\to & ([_\varphi,\ \varphi[\texttt{Browser TrustedApplet}])\\
\to & ([_\varphi,\ \varphi[\texttt{TrustedApplet}])\\
\to & ([_\varphi\,\texttt{read},\ \varphi[\texttt{write}])
\end{array}$$

At this point, the computation aborts, because the history `read write` does not satisfy the property φ, i.e. no `write` can happen after a `read`.

To illustrate the expressive power of our model, consider an untrusted applet than can read, write, or call itself recursively, depending on the values of two guards `b` and `b'`:

```
let rec UnknownApplet2 =
    if b then read
    else if b' then write
         else Browser UnknownApplet2
```

The following trace shows a possible execution of the program `Browser UnknownApplet2; Browser write`. We assume that `b` and `b'` are false for the first n transitions.

$$\begin{array}{ll}
 & (\varepsilon,\ \texttt{Browser UnknownApplet2; Browser write})\\
\to & (\varepsilon,\ \varphi[\texttt{UnknownApplet2}];\texttt{Browser write})\\
\to & ([_\varphi,\ \varphi[\texttt{Browser UnknownApplet2}];\texttt{Browser write})\\
\to & ([_\varphi,\ \varphi[\varphi[\texttt{UnknownApplet2}]];\texttt{Browser write})\\
\to & ([_\varphi[_\varphi,\ \varphi[\varphi[\texttt{Browser UnknownApplet2}]];\texttt{Browser write})\\
\to^* & ([_\varphi^n,\ \varphi^n[\texttt{UnknownApplet2}];\texttt{Browser write})
\end{array}$$

where $\varphi^n[-]$ abbreviates $\varphi[\varphi[\cdots\varphi[-]\cdots]]$, i.e. n nestings of φ. At this point, if the guard `b` becomes true, then the computation proceeds as follows:

$$\begin{array}{ll}
\to & ([_\varphi^n,\ \varphi^n[\texttt{read}];\texttt{Browser write})\\
\to^* & ([_\varphi^n\texttt{read},\ \varphi^n[\,];\texttt{Browser write})\\
\to^* & ([_\varphi^n\texttt{read}]_\varphi^n,\ \texttt{Browser write})\\
\to & ([_\varphi^n\texttt{read}]_\varphi^n,\ \texttt{write})\\
\to & ([_\varphi^n\texttt{read}]_\varphi^n\,\texttt{write},\ \texttt{skip})
\end{array}$$

where the event $]_\varphi$ represents leaving the scope of φ. The `write` operation is performed with full privileges, because the number of $]_\varphi$ matches the number of $[_\varphi$. Note that the property represented by the history $[_\varphi^n\,\texttt{read}\,]_\varphi^n$ is non-regular, because the language $a^n\,b^n$ is context-free.

3. ACCESS CONTROL HISTORIES

We assume a finite set of *access events* Σ (ranged over by α, α'), and a finite set of *policies* Π (ranged over by φ, φ'), i.e. ω-regular properties on sequences of access events. Let $\Sigma_\Pi = \{\, [_\varphi,]_\varphi \mid \varphi \in \Pi \,\}$, with $\Sigma \cap \Sigma_\Pi = \emptyset$.

A *history* η is a (possibly infinite) sequence $(\beta_1, \beta_2, \ldots)$ where $\beta_i \in \Sigma \cup \Sigma_\Pi$. Intuitively, the events in Σ represent activities with possible security concerns, while the events in Σ_Π bind the scope of the access control policies in Π.

Let $\mathcal{H}$ range over sets of histories. Then, $\mathcal{H}\mathcal{H}'$ denotes the set $\{\, \eta\eta' \mid \eta \in \mathcal{H}, \eta' \in \mathcal{H}' \,\}$, and $\varphi[\mathcal{H}]$ denotes the set $\{\, [_\varphi\, \eta\,]_\varphi \mid \eta \in \mathcal{H} \,\}$. When unambiguous, we denote with η

both the history and the singleton set $\{\eta\}$. Note that, if η is infinite, then $\eta\eta' = \eta$, for each η' (in particular, $\varphi[\eta] = [_{\varphi}\eta]_{\varphi} = [_{\varphi}\eta)$.

We say that a finite history η has *balanced framings* if $\eta = \varepsilon$, $\eta = \alpha$, or $\eta = \eta_0\eta_1$ and both η_0 and η_1 have balanced framings, or $\eta = \varphi[\eta']$, and η' has balanced framings. As an example, the history $\alpha[_{\varphi}\alpha'[_{\varphi'}\alpha'']_{\varphi'}]_{\varphi}$ has balanced framings, while $\alpha[_{\varphi}\alpha'$ has not.

Let $\eta = (\beta_1, \dots, \beta_n)$ be a history, let $\eta^{(k)} = (\beta_{i_1}, \dots, \beta_{i_k})$ be the history containing the first k access events of η, and $\varphi_{\eta}^{(k)}$ be the conjunction of the policies φ such that the number of $[_{\varphi}$ is greater than the number of $]_{\varphi}$ in $(\beta_1, \beta_2, \dots, \beta_{i_k})$. We say that η is *valid* if $\eta^{(k)} \models \varphi^{(k)}$, for all k.

For example, consider the history $\eta_0 = \alpha_r\varphi[\alpha_w]$, where φ is the property saying that no α_w can occur after α_r (read α_r and α_w as the events `read` and `write` in Section **??**). Then, η_0 is *not* valid, because $\eta_0^{(2)} = \alpha_r\alpha_w$ does not satisfy $\varphi_{\eta_0}^{(2)} = \varphi$. Instead, the history $\eta_1 = \varphi[\alpha_r]\alpha_w$ is valid, because $\eta_1^{(1)} = \alpha_r$ satisfies $\varphi_{\eta_1}^{(1)} = \varphi$, and $\eta_1^{(2)} = \alpha_r\alpha_w$ satisfies $\varphi_{\eta_1}^{(2)} = \mathit{true}$.

We extend the above definition by continuity, saying that an *infinite* history is valid when all its *finite* prefixes are valid. Assuming continuity is not a limitation, because our notion of validity is a *safety* property: nothing bad can happen in any execution step [**?**].

Note that our notion of validity ensures that the event sequence interested by access control is always the whole history. This is motivated by our basic assumption that no event can ever be hidden from the execution history. For example, a history $\alpha_1\varphi[\alpha_2]\alpha_3$ is valid when $\alpha_1\alpha_2 \models \varphi$ (even if α_1 is outside of the square brackets), while $\alpha_1\alpha_2\alpha_3$ is not required to satisfy φ any longer. Actually, the square brackets dictate the point in the execution where to perform the checks. It is in that sense that we call our policies *local*.

We say that $\mathcal{H}$ *has φ-framings* if and only if:

- $\mathcal{H} = \mathcal{H}_0\mathcal{H}_1$, and $\mathcal{H}_0$ or $\mathcal{H}_1$ have φ-framings.
- $\mathcal{H} = \mathcal{H}_0 \cup \mathcal{H}_1$, and $\mathcal{H}_0$ or $\mathcal{H}_1$ have φ-framings.
- $\mathcal{H} = \varphi'[\mathcal{H}']$, $\mathcal{H}' \neq \emptyset$, and $\varphi = \varphi'$, or $\mathcal{H}'$ has φ-framings.

We say that $\mathcal{H}$ *has framings in* Φ, if and only if Φ is such that $\varphi \in \Phi$ whenever $\mathcal{H}$ has φ-framings.

We say that $\mathcal{H}$ *has redundant framings* if and only if:

- $\mathcal{H} = \mathcal{H}_0\mathcal{H}_1$, and $\mathcal{H}_0$ or $\mathcal{H}_1$ have redundant framings.
- $\mathcal{H} = \mathcal{H}_0 \cup \mathcal{H}_1$, and $\mathcal{H}_0$ or $\mathcal{H}_1$ have redundant framings.
- $\mathcal{H} = \varphi[\mathcal{H}']$, $\mathcal{H}' \neq \emptyset$, and $\mathcal{H}'$ has φ-framings, or $\mathcal{H}'$ has redundant framings.

For example, the history $\eta = \varphi[\alpha]\varphi'[\alpha']$ has framings in $\{\varphi, \varphi'\}$ and no redundant framings. The history $\eta' = \varphi[\varphi''[\eta]]$ has framings in $\{\varphi, \varphi', \varphi''\}$ and redundant framings.

A crucial point about redundant framings is their relation with validity. Indeed, eliminating inner redundant framings preserves the validity of histories. For example, the history $\eta' = \varphi[\varphi''[\varphi[\alpha]\varphi'[\alpha']]]$ has an inner redundant φ-framing around the event α. Since α is already under the scope of the outermost φ, it happens that η' is valid if and only if $\varphi[\varphi''[\alpha\varphi'[\alpha']]]$ is valid.

By standard automata theory arguments, it turns out that identifying the redundant framings within a history requires again the expressive power of pushdown automata, because one has to match pairs of open and closed framings.

For example, consider the following history, where all events but $[_{\varphi}$ and $]_{\varphi}$ have been discarded:

$$\overbrace{[_{\varphi} \cdots [_{\varphi}}^{n}\ \overbrace{]_{\varphi} \cdots]_{\varphi}}^{m} [_{\varphi}$$

Then, the last $[_{\varphi}$ is redundant if $n > m$, and is not if $n = m$.

3.1 History Expressions

We finitely approximate below the infinitary language of histories. This is sufficient for our purposes, because the validity of histories is a safety property. Recall that computations rejected by a safety property are rejected after a finite number of steps [**?**]. *History expressions* have the following abstract syntax:

History Expressions

H, H'	$::=$	ε	empty
		h	variable
		α	access event
		$H \cdot H'$	sequence
		$H + H'$	choice
		$\varphi[H]$	policy framing
		$\mu h.H$	recursion

The free variables in a history expression H are defined as usual: $fv(\varepsilon) = fv(\alpha) = \emptyset$, $fv(H \cdot H') = fv(H + H') = fv(H) \cup fv(H')$, $fv(h) = \{h\}$, $fv(\mu h.\, H) = fv(H) \setminus \{h\}$. We say that a history expression is *closed* when it has no free variables. We fix the precedence of operators as follows: $\cdot$ has precedence over $+$, that in turn has precedence over μ.

History expressions can be extracted from programs using static analysis, e.g. the type and effect system in [**?**]. To give some intuition, consider the example in Section **??**. Assume that the function `display` cannot generate events. Then, the history expression of `Browser UnknownApplet2` is:

$$\mu h.\, \varepsilon + \varphi[\mathtt{read} + \mathtt{write} + h] + (\mathtt{read} + \mathtt{write} + h)$$

This expression denotes the least language $\mathcal{H}$ such that $\mathcal{H}$ contains (i) the empty history ε, and the sets of histories (ii) $\varphi[\mathtt{read} \cup \mathtt{write} \cup \mathcal{H}]$, and (iii) $\mathtt{read} \cup \mathtt{write} \cup \mathcal{H}$.

The denotational semantics of history expressions is defined over the complete lattice $(2^{(\Sigma \cup \Sigma_\Pi)^*}, \subseteq)$. The environment ρ below maps variables to sets of (finite) histories. We stipulate that concatenation and union of sets of histories are defined only if both of their operands are defined. Hereafter, we feel free to omit curly braces, when unambiguous.

Denotational semantics of history expressions

$$\begin{array}{rcl}
[\![\varepsilon]\!]_\rho & = & \varepsilon \\
[\![\alpha]\!]_\rho & = & \alpha \\
[\![h]\!]_\rho & = & \rho(h) \\
[\![H \cdot H']\!]_\rho & = & [\![H]\!]_\rho\, [\![H']\!]_\rho \\
[\![H + H']\!]_\rho & = & [\![H]\!]_\rho \cup [\![H']\!]_\rho \\
[\![\varphi[H]]\!]_\rho & = & \varphi[[\![H]\!]_\rho] \\
[\![\mu h.H]\!]_\rho & = & \bigcup_{n \in \omega} f^n(\emptyset) \quad \text{where } f(X) = [\![H]\!]_{\rho\{X/h\}}
\end{array}$$

As an example, consider $H = \mu h.\, \alpha + h \cdot h + \varphi[h]$. The semantics of H consists of all the histories having an arbitrary number of occurrences of α, and arbitrarily deep, balanced framings of φ. For instance, $\alpha\varphi[\alpha], \varphi[\alpha]\varphi[\alpha\varphi[\alpha]] \in [\![H]\!]_\emptyset$. Note that the semantics of a closed history expression always contains histories with balanced framings.

We say that a history expression H is *valid* when all the histories in $[\![H]\!]$ are valid.

Let $h^* \in fv(H)$ be a selected occurrence of h in H, if any. We say that h^* is guarded by $guard(h^*, H)$, defined as the smallest set satisfying the following equations.

Guards

$$
\begin{aligned}
guard(h^*, h) &= \emptyset \\
guard(h^*, H_0 \cdot H_1) &= \begin{cases} guard(h^*, H_0) & \text{if } h^* \in H_0 \\ guard(h^*, H_1) & \text{otherwise} \end{cases} \\
guard(h^*, H_0 + H_1) &= \begin{cases} guard(h^*, H_0) & \text{if } h^* \in H_0 \\ guard(h^*, H_1) & \text{otherwise} \end{cases} \\
guard(h^*, \varphi[H]) &= \{\varphi\} \cup guard(h^*, H) \\
guard(h^*, \mu h'.\, H') &= guard(h^*, H') \qquad (h' \neq h)
\end{aligned}
$$

Notice that we don't need to treat the case $guard(h, \mu h.\, H)$, because h does not occur freely in $\mu h.\, H$.

For example, consider $\varphi[\alpha \cdot h \cdot \varphi'[h]] \cdot h$. Then, the first occurrence of h is guarded by $\{\varphi\}$, the second one is guarded by $\{\varphi, \varphi'\}$, and the third one is unguarded.

The next two lemmas exploit guards to tell when a history expression has φ-framings or redundant framings. Also, they help proving that redundant framings can be safely removed, as stated in the next section.

Lemma 1. *$[\![H]\!]_\rho$ has φ-framings, if:*

(??*a*) *for some $h \in fv(H)$, $\rho(h)$ has φ-framings, or*

(??*b*) *for some occurrence of $h \in fv(H)$ and set of policies Φ, h is guarded by $\{\varphi\} \cup \Phi$.*

Lemma 2. *$[\![H]\!]_\rho$ has redundant framings, if:*

(??*a*) *for some occurrence of $h \in fv(H)$, $\rho(h)$ has redundant framings, or*

(??*b*) *for some occurrence of $h \in fv(H)$ and some Φ, h is guarded by $\{\varphi\} \cup \Phi$, and $\rho(h)$ has φ-framings, or*

(??*c*) *$H = \mu h.H'$, and some occurrence of h is guarded in H'.*

3.2 Elimination of the redundant framings

The semantics of a history expression can have redundant framings. As a consequence, an automaton recognizing all and only the valid histories needs to have the expressive power of pushdown automata. This complexity is not acceptable. Thus, we introduce a transformation that, given a history expression H, produces an expression H' such that (i) H is valid if and only if H' is valid, and (ii) the histories generated by H' can be verified by a finite state automaton.

Let H be a (possibly non-closed) history expression. Without loss of generality, assume that all the variables in H have distinct names. We define below $H\!\downarrow_{\Phi,\Gamma}$, the expression produced by the *regularization* of H against a set of policies Φ and a mapping Γ from variables to history expressions.

Regularization of history expressions

$$
\begin{aligned}
\varepsilon\!\downarrow_{\Phi,\Gamma} &= \varepsilon \\
h\!\downarrow_{\Phi,\Gamma} &= h \\
\alpha\!\downarrow_{\Phi,\Gamma} &= \alpha \\
(H \cdot H')\!\downarrow_{\Phi,\Gamma} &= H\!\downarrow_{\Phi,\Gamma} \cdot H'\!\downarrow_{\Phi,\Gamma} \\
(H + H')\!\downarrow_{\Phi,\Gamma} &= H\!\downarrow_{\Phi,\Gamma} + H'\!\downarrow_{\Phi,\Gamma} \\
\varphi[H]\!\downarrow_{\Phi,\Gamma} &= \begin{cases} H\!\downarrow_{\Phi,\Gamma} & \text{if } \varphi \in \Phi \\ \varphi[H\!\downarrow_{\Phi\cup\{\varphi\},\Gamma}] & \text{otherwise} \end{cases} \\
(\mu h.\, H)\!\downarrow_{\Phi,\Gamma} &= \mu h.\, (H'\sigma'\!\downarrow_{\Phi,\Gamma\{(\mu h.H)\Gamma/h\}} \sigma) \\
&\text{where } H = H'\{h/h_i\}_i,\ h_i \text{ fresh, } h \notin fv(H'), \text{ and} \\
\sigma(h_i) &= (\mu h.H)\Gamma\!\downarrow_{\Phi\cup guard(h_i,H'),\Gamma} \\
\sigma'(h_i) &= \begin{cases} h & \text{if } guard(h_i, H') \subseteq \Phi \\ h_i & \text{otherwise} \end{cases}
\end{aligned}
$$

Intuitively, $H\!\downarrow_{\Phi,\Gamma}$ results from H by eliminating all the redundant framings, and all the framings in Φ. The environment Γ is needed to deal with free variables in the case of nested μ-expressions, as shown by Example **??** below. We sometimes omit to write the component Γ when unneeded, and, when H is closed, we abbreviate $H\!\downarrow_{\emptyset,\emptyset}$ with $H\!\downarrow$.

The last two regularization rules would benefit from some explanation. Consider first a history expression of the form $\varphi[H]$ to be regularized against a set of policies Φ. To eliminate the redundant framings, we must ensure that H has neither φ-framings, nor redundant framings itself. This is accomplished by regularizing H against $\Phi \cup \{\varphi\}$.

Consider a history expression of the form $\mu h.H$. Its regularization against Φ and Γ proceeds as follows. Each free occurrence of h in H guarded by some $\Phi' \not\subseteq \Phi$ is unfolded and regularized against $\Phi \cup \Phi'$. The substitution Γ is used to bind the free variables to closed history expressions. Technically, the i-th free occurrence of h in H is picked up by the substitution $\{h/h_i\}$, for h_i fresh. Note also that $\sigma(h_i)$ is computed only if $\sigma'(h_i) = h_i$.

As a matter of fact, regularization is a total function, and its definition induces a terminating rewriting system.

Example 1. *Let $H_0 = \mu h.\, H$, where $H = \alpha + h \cdot h + \varphi[h]$. Then, H can be written as $H'\{h/h_i\}_{i\in 0..2}$, where $H' = \alpha + h_0 \cdot h_1 + \varphi[h_2]$. Since $guard(h_2, H') = \{\varphi\} \not\subseteq \emptyset$:*

$$
\begin{aligned}
H_0\!\downarrow_\emptyset &= \mu h.\, H'\{h/h_0, h/h_1\}\!\downarrow_\emptyset \{H_0\!\downarrow_\varphi /h_2\} \\
&= \mu h.\, \alpha + h \cdot h + \varphi[H_0\!\downarrow_\varphi]
\end{aligned}
$$

To compute $H_0\!\downarrow_\varphi$, note that no occurrence of h is guarded by $\Phi \not\subseteq \{\varphi\}$. Then:

$$H_0\!\downarrow_\varphi \;=\; \mu h.\, (\alpha + h \cdot h + \varphi[h])\!\downarrow_\varphi \;=\; \mu h.\, \alpha + h \cdot h + h$$

Since $[\![H_0\!\downarrow_\varphi]\!] = \{\alpha\}^\omega$ has no φ-framings, we have that $[\![H_0\!\downarrow]\!] = \big(\{\alpha\}^\omega \varphi[\{\alpha\}^\omega]\big)^\omega$ has no redundant framings.

Example 2. *Let $H_0 = \mu h.\, H_1$, where $H_1 = \mu h'.\, H_2$, and $H_2 = \alpha + h \cdot \varphi[h']$. Since $guard(h, H_1) = \emptyset$, we have that:*

$$H_0\!\downarrow_{\emptyset,\emptyset} \;=\; \mu h.\, (H_1\!\downarrow_{\emptyset,\{H_0/h\}})$$

Note that H_2 can be written as $H'_2\{h/h_0\}$, where $H'_2 = \alpha + h \cdot \varphi[h_0]$. Since $guard(h_0, H'_2) = \{\varphi\} \not\subseteq \emptyset$, it follows that:

$$\begin{aligned} H_1 {\downarrow_{\emptyset,\{H_0/h\}}} &= \mu h'.\, H'_2 {\downarrow_{\emptyset,\{H_0/h, H_1\{H_0/h\}/h'\}}} \\ &\quad \{H_1\{H_0/h\} {\downarrow_{\varphi,\{H_0/h\}}} /h_0\} \\ &= \mu h'.\, \alpha + h \cdot \varphi[h_0] \\ &\quad \{(\mu h'.\, \alpha + H_0 \cdot \varphi[h']) {\downarrow_{\varphi,\{H_0/h\}}} /h_0\} \\ &= \mu h'.\, \alpha + h \cdot \varphi[H_3 {\downarrow_{\varphi,\{H/h\}}}] \\ &= \alpha + h \cdot \varphi[H_3 {\downarrow_{\varphi,\{H/h\}}}] \end{aligned}$$

where $H_3 = \mu h'.\, \alpha + H_0 \cdot \varphi[h']$, and the last simplification is possible because the outermost $\mu h'$ binds no variable. Since $guard(h', \alpha + H_0 \cdot \varphi[h']) = \{\varphi\} \subseteq \{\varphi\}$, it follows that:

$$H_3 {\downarrow_\varphi} \;=\; \mu h'.\, (\alpha + H_0 \cdot \varphi[h']) {\downarrow_\varphi} \;=\; \mu h'.\, \alpha + H_0 {\downarrow_\varphi} \cdot h'$$

To compute $H_0 {\downarrow_\varphi}$, note that $\{\varphi\}$ contains both $guard(h, H_1) = \emptyset$, and $guard(h', H_2) = \{\varphi\}$. Then:

$$\begin{aligned} H_0 {\downarrow_\varphi} &= \mu h.\, (\mu h'.\, \alpha + h \cdot \varphi[h']) {\downarrow_\varphi} \\ &= \mu h.\, \mu h'.\, (\alpha + h \cdot \varphi[h']) {\downarrow_\varphi} \\ &= \mu h.\, \mu h'.\, \alpha + h \cdot h' \end{aligned}$$

Putting together the computations above, we have that:

$$\begin{aligned} H_0 {\downarrow_\emptyset} &= \mu h.\, \alpha + h \cdot \varphi[H_3 {\downarrow_\varphi}] \\ H_3 {\downarrow_\varphi} &= \mu h'.\, \alpha + \big(\mu h.\, \mu h'.\, \alpha + h \cdot h'\big) \cdot h' \end{aligned}$$

We conclude this subsection by establishing the following basic property of regularization.

Theorem 1. *$H{\downarrow}$ has no redundant framings.*

3.3 Normalization of histories

So far, we have proved that regularization makes history expressions to generate histories with no redundant framings. We now show that regularization also preserves the validity of histories.

It is convenient to introduce a *normal form* for histories. It permits to compare the histories produced by an expression H with those of the regularization of H.

Normalization of histories

$$\begin{aligned} \varepsilon {\Downarrow_\Phi} &= \varepsilon \\ \alpha {\Downarrow_\Phi} &= (\textstyle\bigwedge \Phi)\, [\alpha] \\ (\mathcal{H}\mathcal{H}') {\Downarrow_\Phi} &= \mathcal{H} {\Downarrow_\Phi}\, \mathcal{H}' {\Downarrow_\Phi} \\ (\mathcal{H} \cup \mathcal{H}') {\Downarrow_\Phi} &= \mathcal{H} {\Downarrow_\Phi} \cup \mathcal{H}' {\Downarrow_\Phi} \\ \varphi[\mathcal{H}] {\Downarrow_\Phi} &= \mathcal{H} {\Downarrow_{\Phi \cup \{\varphi\}}} \end{aligned}$$

Intuitively, normalization transforms histories with policy framings into histories with local checks. Indeed, $\eta {\Downarrow_\Phi}$ is intended to record that each event in η must obey to *all* the policies in Φ. This is apparent in the second and in the last equation above.

Note that constructing the normal form of a history requires counting the framing openings and closings (see the last equation): a pushdown automaton is therefore needed.

We abbreviate $\mathcal{H} {\Downarrow_\emptyset}$ with $\mathcal{H}{\Downarrow}$. Note that $\mathcal{H} {\Downarrow_\emptyset}$ is defined if and only if $\mathcal{H}$ has balanced framings.

Example 3. *Consider the history $\eta = \alpha\varphi[\alpha'\varphi'[\alpha'']]$. Its normal form is computed as follows:*

$$\begin{aligned} \eta{\Downarrow} &= \alpha{\Downarrow}\, (\varphi[\alpha'\varphi'[\alpha'']]){\Downarrow} \\ &= \alpha\; (\alpha'\varphi'[\alpha'']) {\Downarrow_\varphi} \\ &= \alpha\; (\alpha' {\Downarrow_\varphi})\; (\varphi'[\alpha'']) {\Downarrow_\varphi} \\ &= \alpha\; \varphi[\alpha']\; (\alpha'' {\Downarrow_{\varphi,\varphi'}}) \\ &= \alpha\; \varphi[\alpha']\; (\varphi \wedge \varphi')[\alpha''] \end{aligned}$$

The following theorem establishes that a history expression H and its regularization $H{\downarrow}$ have the same normal form.

Theorem 2. $[\![H{\downarrow}]\!]{\Downarrow} = [\![H]\!]{\Downarrow}$.

The next theorem states that normalization also preserves the validity of histories.

Theorem 3. *A history η is valid iff $\eta{\Downarrow}$ is valid.*

Summing up, we conclude that a history expression H is valid if and only if its regularization $H{\downarrow}$ is valid.

4. VERIFICATION

We now introduce a procedure to verify the validity of history expressions. Our technique is based on model checking Basic Process Algebras (BPAs) with Büchi automata, which is known to be decidable [?]. Furthermore, several algorithms and tools show this approach feasible.

BPAs provide a natural characterization of (possibly infinite) histories. A *BPA process* is given by the following abstract syntax:

$$p \;::=\; \varepsilon \;\mid\; \alpha \;\mid\; p \cdot p' \;\mid\; p + p' \;\mid\; X$$

where ε denotes the terminated process, $\alpha \in \Sigma$, X is a variable, $\cdot$ denotes sequential composition, $+$ represents (nondeterministic) choice.

A BPA process p is *guarded* if each variable occurrence in p occurs in a subexpression $\alpha \cdot q$ of p. We assume a finite set $\Delta = \{X \overset{def}{=} p\}$ of guarded definitions, such that each variable X has a unique defining equation, i.e. there exists a single, guarded p such that $\{X \overset{def}{=} p\} \in \Delta$. As usual, we consider the process $\varepsilon \cdot p$ to be equivalent to p.

The operational semantics of BPAs is given by the following labelled transition system, in the SOS style:

Operational Semantics of BPA processes

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

$$\frac{p \xrightarrow{\alpha} p'}{p \cdot q \xrightarrow{\alpha} p' \cdot q} \qquad \frac{p \xrightarrow{\alpha} p' \quad X \overset{def}{=} p \in \Delta}{X \xrightarrow{\alpha} p'}$$

We denote with $[\![p_0, \Delta]\!]$ the set $\{\, (a_i)_i \mid p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} p_i \,\}$ $\cup \{\, (a_i)_i \mid p_0 \cdots \xrightarrow{a_i} \cdots \}$. We write $[\![p, \Delta]\!]^{\mathrm{fin}}$ for the first set, containing the strings that label finite computations. We omit the component Δ, when empty.

We now introduce an encoding of history expressions into BPAs, in the line of [?]. The encoding takes as input a history expression H and a mapping Ψ from history variables h to BPA variables X. Without loss of generality, we assume that all the variables in H have distinct names. The encoding outputs a BPA process p and a finite set of definitions Δ.

To avoid the problem of unguarded BPA processes, we assume a standard preprocessing step, that inserts a dummy event before each unguarded occurrence of a variable in a history expression. Dummy events are eventually discarded before the verification phase.

The rules that transform history expressions into BPAs are rather standard. History events, variables, concatenation and choice are mapped into the corresponding BPA counterparts. A history expression $\mu h.H$ is encoded into a fresh BPA variable X, bound to the translation of H in the set of definitions Δ (the mapping Ψ is extended by binding h to X). An expression $\varphi[H]$ is encoded in the BPA for H, surrounded by the opening and closing of the φ-framing.

Encoding of history expressions into BPAs

$$\begin{array}{rcl}
BPA(\varepsilon, \Psi) & = & \langle \varepsilon, \emptyset \rangle \\
BPA(\alpha, \Psi) & = & \langle \alpha, \emptyset \rangle \\
BPA(h, \Psi) & = & \langle \Psi(h), \emptyset \rangle \\
BPA(H_0 \cdot H_1, \Psi) & = & \langle p_0 \cdot p_1, \Delta_0 \cup \Delta_1 \rangle, \\
 & & \text{where } BPA(H_i, \Psi) = \langle p_i, \Delta_i \rangle \\
BPA(H_0 + H_1, \Psi) & = & \langle p_0 + p_1, \Delta_0 \cup \Delta_1 \rangle, \\
 & & \text{where } BPA(H_i, \Psi) = \langle p_i, \Delta_i \rangle \\
BPA(\mu h.H, \Psi) & = & \langle X, \Delta \cup \{ X \overset{def}{=} p \} \rangle, \\
 & & \text{where } BPA(H, \Psi\{X/h\}) = \langle p, \Delta \rangle \\
 & & \text{and } X \notin dom(\Delta) \\
BPA(\varphi[H], \Psi) & = & \langle [_\varphi \cdot p \cdot]_\varphi, \Delta \rangle, \\
 & & \text{where } BPA(H, \Psi) = \langle p, \Delta \rangle
\end{array}$$

We now state the correspondence between the semantics of history expressions and of BPAs. The histories generated by a history expression H are all and only the finite prefixes of the strings that label the computations of $BPA(H)$. Recall that this is enough, because validity is a safety property.

Lemma 3. $[\![H]\!] = [\![BPA(H)]\!]^{fin}$.

A standard approach to define properties of computations is modelling them as an infinitary language accepted by a Büchi automaton. Büchi automata are finite state automata whose acceptance condition roughly says that a computation is accepted if some final state is visited infinitely often; see [?] for details.

Since we also need to establish the validity of finite histories, we use the standard trick of assuming that a finite string is padded at the end with a special symbol \$. Then, each final state has a self-loop labelled with \$. For brevity, we will omit these transitions hereafter.

Given a policy φ, we are interested in defining a formula $\varphi[\,]$ to be used in verifying that a history η, with no redundant framings of φ, respects φ within its scope.

Let the formula φ be defined by the Büchi automaton $A_\varphi = \langle \Sigma, Q, Q_0, \rho, F \rangle$, with $Q = \{q_1, \ldots, q_k\}$. We assume that A_φ has a non-final sink state from which you cannot leave.

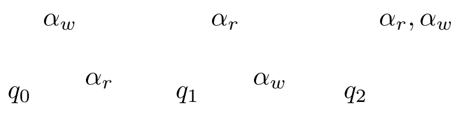

Figure 1: Büchi automaton for φ.

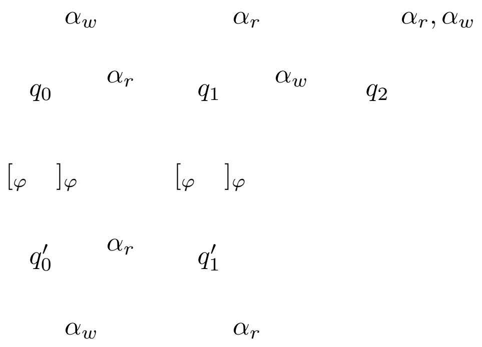

Figure 2: Büchi automaton for $\varphi[\,]$.

We define the formula $\varphi[\,]$ through the Büchi automaton $A_{\varphi[\,]} = \langle \Sigma', Q', Q_0, \rho', F' \rangle$, where: $\Sigma' = \Sigma \cup \{ [_\varphi,]_\varphi \mid \varphi \in \Pi \}$, $Q' = F' = Q \cup \{ q'_i \mid q_i \in F \}$, and

$$\begin{array}{rcl}
\rho' & = & \rho \cup \{ \langle q_i, [_\varphi, q'_i \rangle \mid q_i \in F \} \;\cup\; \{ \langle q'_i,]_\varphi, q_i \rangle \} \\
 & & \cup \; \{ \langle q'_i, \alpha, q'_j \rangle \mid \langle q_i, \alpha, q_j \rangle \in \rho \text{ and } q_j \in F \} \\
 & & \cup \; \{ \langle q, [_{\varphi'}, q \rangle \cup \langle q,]_{\varphi'}, q \rangle \mid q \in Q' \text{ and } \varphi' \neq \varphi \}
\end{array}$$

Intuitively, the automaton $A_{\varphi[\,]}$ is partitioned into two layers. The first layer is a copy of A_φ, where all the states are final. This models the fact that we are outside the scope of φ, i.e. the history leading to any state in this layer has balanced framings of φ (or none).

The second layer is reachable from the first one when opening a framing for φ, while closing a framing gets back. The transitions in the second layer are a copy of those connecting final states in A_φ. Consequently, the states in the second layer are exactly the final states in A_φ.

Since A_φ is only concerned with the verification of φ, the transitions for opening and closing framings $\varphi' \neq \varphi$ are rendered as self-loops.

Example 4. *Let φ be the policy saying that there cannot be an event α_w after an α_r (actually, this is the policy of Section* **??**, *where* `read` *and* `write` *become α_r and α_w). The Büchi automaton for φ is in Fig.* **??**, *while the one for $\varphi[\,]$ is in Fig.* **??**. *For example, the history $[_\varphi \alpha_r]_\varphi \alpha_w$ is accepted by $A_{\varphi[\,]}$, while $\alpha_r [_\varphi \alpha_w]_\varphi$ is not (recall that we do not draw the self-loops labelled by \$).*

The following lemma relates validity of histories with the formulae $\varphi[\,]$. A history η is valid if and only if it satisfies $\varphi[\,]$ for all the policies φ spanning over η.

Lemma 4. *A history η with no redundant framings is valid if and only if $\eta \models \varphi[\,]$, for all φ such that $[_\varphi \in \eta$.*

Recall that Büchi automata are closed under intersection [?]. Therefore, a valid history η is accepted by the intersection of the automata $A_{\varphi[\,]}$, for all φ occurring in η.

The main result of our paper follows. Validity of a history expression H can be decided by showing that the BPA generated by the regularization of H satisfies a ω-regular formula.

THEOREM 4. *$[\![H]\!]$ is valid if and only if:*

$$[\![BPA(H\!\downarrow)]\!] \models \bigwedge_{\varphi \in H} \varphi[\,]$$

PROOF. By theorem **??**, $[\![H]\!]$ is valid if and only if $[\![H]\!]\Downarrow$ is valid. By theorem **??**, $[\![H]\!]\Downarrow = [\![H\!\downarrow]\!]\Downarrow$. By theorem **??**, $[\![H\!\downarrow]\!]\Downarrow$ is valid if and only if $[\![H\!\downarrow]\!]$ is valid. By theorem **??**, $[\![H\!\downarrow]\!]$ has no redundant framings. By lemma **??**, $[\![H\!\downarrow]\!] = [\![BPA(H\!\downarrow)]\!]^{\text{fin}}$. By continuity, $[\![BPA(H\!\downarrow)]\!]^{\text{fin}}$ is valid if and only if $[\![BPA(H\!\downarrow)]\!]$ is valid. Then, by lemma **??**, $[\![BPA(H\!\downarrow)]\!]$ is valid iff $[\![BPA(H\!\downarrow)]\!] \models \bigwedge_{\varphi \in H} \varphi[\,]$. $\Box$

5. CONCLUSIONS

We have tackled the problem of controlling accesses to protected objects or critical resources, along the lines of Skalka and Smith [?]. A novel approach to history-based access control has been proposed, by endowing security policies with a local scope. Security policies are regular properties of histories; histories are abstract representations of the activities performed while running programs, enriched with special events that mark the scope of policies. Following [?] we have also introduced history expressions, that represent sets of histories.

A history is valid whenever it satisfies all the policies occurring in it, within their scopes. Policy framings explicitly represent the scope of policies within our histories, and can be arbitrarily nested. Even though polices are regular properties, nesting policy framings in history expressions makes validity of histories a non-regular property. Non-regularity seemed to prevent us from verifying validity by standard model checking techniques, but we have been able to transform history expressions so that model checking is feasible. Finally, we have extended known techniques for that, using Basic Process Algebras and Büchi automata.

All the above has been carried out on finite histories, but our results extend to infinite histories by continuity, as validity turns out to be a safety property (nothing bad will happen). The extension to infinite histories and the use of Büchi automata is not an unnecessary complication, as we are currently considering liveness properties (something good will happen), which are not prefix closed.

In [?], we consider an abstract language for history-based access control, based on the λ-calculus. We define for it a type and effect system that, given a program, extracts a history expression H. The set of histories represented by H includes those obtainable at run-time. One can then exploit our present proposal to check at compile-time if a program will respect a given policy at run-time, so giving firm grounds to program optimization.

6. ACKNOWLEDGMENTS

We wish to thank Roberto Zunino, Emilio Tuosto and Alberto Lluch Lafuente for their keen remarks on preliminary versions of this paper.

This work has been partially supported by EU project DEGAS (IST-2001-32072) and FET project PROFUNDIS (IST-2001-33100).

7. REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.

[2] A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)*, 2004.

[3] M. Bartoletti. PhD thesis, Università di Pisa, Forthcoming.

[4] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[5] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[6] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2000.

[7] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*. Springer, 1999.

[8] J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th International Colloquium on Trees in Algebra and Programming*, 1994.

[9] P. W. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, 2004.

[10] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[11] A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on selected areas in communication*, 21(1), 2003.

[12] P. Samarati and S. de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design: Tutorial Lectures*, volume 2171 of *LNCS*. Springer-Verlag, 2001.

[13] F. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*. Springer-Verlag, 2001.

[14] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[15] C. Skalka and S. Smith. History effects and verification. In *Asian Programming Languages Symposium*, 2004.

[16] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. Banff Higher order workshop conference on Logics for concurrency*, 1996.