History-Based Access Control with Local Policies

Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari

Dipartimento di Informatica, Università di Pisa, Italy
{bartolet, degano, giangi}@di.unipi.it

Abstract. An extension of the λ-calculus is proposed, to study history-based access control. It allows for security policies with a possibly nested, local scope. We define a type and effect system that, given a program, extracts a history expression, i.e. a correct approximation to the set of histories obtainable at run-time. Validity of history expressions is non-regular, because the scope of policies can be nested. Nevertheless, a transformation of history expressions is presented, that makes verification possible through standard model checking techniques. A program will never fail at run-time if its history expression, extracted at compile-time, is valid.

1 Introduction

Models and techniques for language-based security are receiving increasing attention [14, 16]. Among these, access control plays a relevant role [15]. Indeed, features for defining and enforcing access control policies are a main concern in the design of modern programming languages. Access control *policies* specify the rules by which principals are authorized to access protected objects and 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 call stack, stack inspection may be insecure when trusted code depends on results supplied by untrusted code [11]. In fact, access controls are insensitive to the frame of an untrusted piece of 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 (however, it is sometimes possible to exploit static techniques, e.g. those in [4], that allow for secure optimizations).

V. Sassone (Ed.): FOSSACS 2005, LNCS 3441, pp. 316–332, 2005.

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, at both levels of foundations [2, 10, 18] and of language design and implementation [1, 8].

The typical run-time mechanisms for enforcing history-based policies are *execution monitors*, which observe computations 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 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. Execution monitors have been proved to enforce exactly the class of *safety* properties [17].

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 [6, 7, 22]. Recently, Skalka and Smith [18] 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 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.

Building upon [18], we propose here $\lambda^{[]}$, an extension of the λ-calculus that allows for expressive and flexible history-based access control policies. The security properties imposed in our programs are *regular* properties of histories, and have a possibly nested, local scope. A program e protected by a policy φ is written $\varphi[e]$, called *policy framing*. During the evaluation of e, the whole execution history (i.e. the past history followed by the events generated so far by e) must respect the policy φ. This allows for safe composition of programs that are protected by different policies. For example, suppose to have an expression $\lambda x.\, \varphi[(x\, v)\, e]$ that takes as input a function for x, and applies it to the value v while enforcing the policy φ. Then, supplying the function $\lambda y.\, \varphi'[e']$, we have the following computation:

$$\begin{aligned}(\lambda x.\, \varphi[(x\, v)\, e])\, (\lambda y.\, \varphi'[e']) \;&\rightarrow\; \varphi[(\lambda y.\, \varphi'[e'])\, v\, e] \;\rightarrow\; \varphi[\varphi'[e'\{v/y\}]\, e] \\ &\rightarrow^{*} \varphi[\varphi'[v']\, e] \;\rightarrow\; \varphi[v'\, e]\end{aligned}$$

Evaluating the application of e' to v must respect, at each step, both policies φ and φ'. As soon as a function v' is obtained, the scope of φ' is left, and the application of v' to e' continues under the scope of φ. Note that the first step of

the computation above can be viewed as dynamically placing a program into a *sandbox* enforcing the policy φ. This programming paradigm seems difficult to express in a language with local checks or global policies, only.

Even though policies are regular properties, the nesting of policy framings may give rise to *non-regular* properties: indeed, every history η must obey the conjunction of all the policies within the scope of which the last event of η occurs. Run-time mechanisms enforcing this kind of properties need to be at least powerful as pushdown automata. Consequently, $\lambda^{[]}$ is strictly more expressive than the sub-language that only admits policy framings with single events, i.e. local checks (of course, the above holds under the assumption that the access control mechanism is not encoded in λ-expressions themselves).

We define a type and effect system for $\lambda^{[]}$. The types are standard, while the effects are *history expressions*, a finite approximation of the infinitary language of histories, together with explicit representation of the scope of policy framings. We say that a history expression is valid if all its histories are such, i.e. they represent safe executions. Considering finite histories only is sufficient, because the validity of histories is a safety property. Recall that computations not enjoying a safety property are rejected in a finite number of steps [17]. If the effect of a program is valid, then the program will never throw any security exceptions.

Even though validity of histories is a non-regular property, we show that history expressions can be model checked with standard 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 [9].

2 The Language $\lambda^{[]}$

To study access control in a pure framework, we consider $\lambda^{[]}$, a call-by-value λ-calculus enriched with access events and security policies. An *access event* $\alpha \in \Sigma$ abstracts from a security-relevant operation; sequences η of access events are called *histories*. Security policies $\varphi \in \Pi$ are regular properties of histories. A *policy framing* $\varphi[e]$ localizes the scope of the policy φ to the expression e; framings can be nested. To enhance readability, our calculus comprises conditional expressions and named abstractions (the variable z in $e' = \lambda_z x.e$ stands for e' itself within e). The syntax of $\lambda^{[]}$ follows. We omit the definition of policies φ and of guards b, as they are not relevant for the subsequent technical development.

Syntax of $\lambda^{[]}$ Expressions

$$e, e' \quad ::= \quad x \mid \alpha \mid \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e' \mid \lambda_z x.\, e \mid e\, e' \mid \varphi[e]$$

The values of $\lambda^{[]}$ are the variables and the abstractions. We write $*$ for a fixed, closed, event-free value, and $\lambda\_.\, e$ for $\lambda x.\, e$, for $x \notin fv(e)$. The following abbreviation is standard: $e; e' = (\lambda\_.\, e')e$.

We define the behaviour of $\lambda^{[]}$ expressions through the following SOS operational semantics. The configurations are pairs η, e. A transition $\eta, e \to \eta', e'$ means that, starting from a history η, the expression e may evolve to e', possibly extending the history η. We write $\eta \models \varphi$ when the history η obeys the policy φ. We assume as given a total function $\mathcal{B}$ that evaluates the guards in conditionals.

Operational Semantics of $\lambda^{[]}$

$$\frac{\eta, e_1 \to \eta', e_1'}{\eta, e_1 e_2 \to \eta', e_1' e_2} \qquad \frac{\eta, e_2 \to \eta', e_2'}{\eta, v e_2 \to \eta', v e_2'} \qquad \frac{}{\eta, (\lambda_z x.e)v \to \eta, e\{v/x, \lambda_z x.e/z\}}$$

$$\frac{}{\eta, \alpha \to \eta\alpha, *} \qquad \frac{\eta, e \to \eta', e' \quad \eta, \eta' \models \varphi}{\eta, \varphi[e] \to \eta', \varphi[e']} \qquad \frac{\eta \models \varphi}{\eta, \varphi[v] \to \eta, v}$$

$$\frac{\mathcal{B}(b) = \mathit{true}}{\eta, \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1 \to \eta, e_0} \qquad \frac{\mathcal{B}(b) = \mathit{false}}{\eta, \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1 \to \eta, e_1}$$

It is immediate to define a semantics of $\lambda^{[]}$, equivalent to that given above, that explicitly records entering and exiting a framing $\varphi[\cdots]$, by enriching histories with special events $[_\varphi$ and $]_\varphi$. Each transition requires to verify that the current history is *valid*, roughly it satisfies all the policies φ whose scope has been entered but not exited yet, i.e. the number of $[_\varphi$ is greater then that of $]_\varphi$. Counting is not regular: therefore, validity is not a regular property.

To illustrate our approach, consider a simple web browser that displays HTML pages and runs applets. Applets can be trusted (e.g. because signed, or downloaded from a trusted site), or untrusted. The browser has a site policy φ always applied to untrusted applets. The site policy says that an applet cannot connect to the web after it has read from the local disk. After executing an untrusted applet, the browser writes some logging information to the local disk. Additionally, all applets must obey a user policy that is supplied to the browser. We define the browser as a function that processes the URL u, be it an applet or an HTML page, and the user policy φ', rendered as a framing $p = \lambda x.\, \varphi'[x*]$.

$$\begin{aligned} \mathit{Browser} = \lambda u.\, \lambda p.\, &\mathtt{if}\ \mathit{html}(u)\ \mathtt{then}\ \mathit{display}(u)\ \mathtt{else} \\ &\mathtt{if}\ \mathit{trusted}(u)\ \mathtt{then}\ p\ u\ \mathtt{else}\ \varphi[p\ u; \mathit{Write}\ *] \end{aligned}$$

We consider three trusted applets: $\mathit{Read} = \lambda\_.\, \alpha_r$ to read files, $\mathit{Write} = \lambda\_.\, \alpha_w$ to write files, and $\mathit{Connect} = \lambda\_.\, \alpha_c$ to open web connections. Note that our applets are overly simplified, because we are only interested in the events they

can generate, namely $\alpha_r, \alpha_w, \alpha_c$. We also have an untrusted applet, that tries to spoof the browser by executing a supplied applet z with a void user policy.

$$\mathit{Untrusted} \;=\; \lambda z.\, \lambda\_.\, \mathit{Browser}\; z\; (\lambda y.\, y*)$$

The behaviour of an untrusted write, executed with a user policy φ' saying that applets cannot write the local disk, is illustrated by the following trace:

$$\begin{array}{rl} & \varepsilon,\ \mathit{Browser}\,(\mathit{Untrusted}\ \mathit{Write})\,(\lambda y.\varphi'[y*]) \\ \to^* & \varepsilon,\ \varphi[(\lambda y.\varphi'[y*])\,(\lambda\_.\ \mathit{Browser}\ \mathit{Write}\,(\lambda y.\ y*));\ \mathit{Write}*] \\ \to^* & \varepsilon,\ \varphi[\varphi'[\mathit{Browser}\ \mathit{Write}\,(\lambda y.\ y*)];\ \mathit{Write}*] \\ \to^* & \varepsilon,\ \varphi[\varphi'[(\lambda y.\ y*)\,\mathit{Write}];\ \mathit{Write}*] \quad \to^* \quad \varepsilon,\ \varphi[\varphi'[\alpha_w];\ \mathit{Write}*] \end{array}$$

At this point, a security exception is thrown, because the history α_w would not satisfy the innermost policy φ'. Consider now an untrusted applet that reads the local disk and then tries to connect.

$$\begin{array}{rl} & \varepsilon,\ \mathit{Browser}\,(\mathit{Untrusted}\,(\lambda\_.\ \mathit{Read}*;\ \mathit{Connect}*))\,(\lambda y.\varphi'[y*]) \\ \to^* & \varepsilon,\ \varphi[(\lambda y.\varphi'[y*])\,(\mathit{Untrusted}\,(\lambda\_.\ \mathit{Read}*;\ \mathit{Connect}*));\ \mathit{Write}*] \\ \to^* & \varepsilon,\ \varphi[\varphi'[\mathit{Browser}\,(\lambda\_.\ \mathit{Read}*;\ \mathit{Connect}*)\,(\lambda y.\ y*)];\ \mathit{Write}*] \\ \to^* & \varepsilon,\ \varphi[\varphi'[\mathit{Read}*;\ \mathit{Connect}*];\ \mathit{Write}*] \quad \to^* \quad \alpha_r,\ \varphi[\varphi'[\mathit{Connect}*];\ \mathit{Write}*] \end{array}$$

Again, we have a security exception, because the history $\alpha_r\alpha_c$ does not satisfy the outermost policy φ. As a further example, consider an untrusted read:

$$\begin{array}{rl} & \varepsilon,\ \mathit{Browser}\,(\mathit{Untrusted}\ \mathit{Read})\,(\lambda y.\varphi'[y*]) \\ \to^* & \varepsilon,\ \varphi[(\lambda y.\varphi'[y*])\,(\mathit{Untrusted}\ \mathit{Read});\ \mathit{Write}*] \\ \to^* & \varepsilon,\ \varphi[\varphi'[\mathit{Browser}\ \mathit{Read}\,(\lambda y.\ y*)];\ \mathit{Write}*] \ \to^* \varepsilon,\ \varphi[\varphi'[\mathit{Read}*];\ \mathit{Write}*] \\ \to^* & \alpha_r,\ \varphi[\varphi'[*];\ \mathit{Write}*] \quad \to^* \quad \alpha_r,\ \varphi[\mathit{Write}*] \quad \to^* \quad \alpha_r\alpha_w,\ \varphi[*] \end{array}$$

Unlike in the first computation, the write operation has been performed, because the scope of the policy φ' has been left upon termination of the untrusted applet.

3 A Type and Effect System for $\boldsymbol{\lambda}^{[]}$

To statically predict the histories generated by programs at run-time, we introduce *history expressions* with the following abstract syntax.

History Expressions

$$H, H' \;::=\; \varepsilon \;\mid\; h \;\mid\; \alpha \;\mid\; H \cdot H' \;\mid\; H + H' \;\mid\; \varphi[H] \;\mid\; \mu h.H$$

History expressions include the empty history ε, events α, sequencing $H \cdot H'$, non-deterministic choice $H + H'$, policy framing $\varphi[H]$, and recursion $\mu h.H$ (μ binds the occurrences of the variable h in H). Free variables and closed expressions are defined as expected. We assume that the operator $\cdot$ has precedence over $+$, that in turn has precedence over μ.

Hereafter, we extend histories with an explicit representation of policy framings. We use special symbols $[_\varphi$ and $]_\varphi$ to denote the opening and closing of the scope of the policy φ. Formally, an *enriched history* η, or simply *history* when unambiguous, is a (possibly infinite) sequence $(\beta_1, \beta_2, \ldots)$ where $\beta_i \in \Sigma \cup \Sigma_\Pi$, $\Sigma_\Pi = \{\, [_\varphi,]_\varphi \mid \varphi \in \Pi \,\}$, and $\Sigma \cap \Sigma_\Pi = \emptyset$.

Let $\mathcal{H}$ range over sets of histories. Then, $\mathcal{H}\mathcal{H}'$ denotes the set of histories $\{\, \eta\eta' \mid \eta \in \mathcal{H}, \eta' \in \mathcal{H}' \,\}$, and $\varphi[\mathcal{H}]$ is the set $\{\, [_\varphi\, \eta\,]_\varphi \mid \eta \in \mathcal{H} \,\}$. Note that, if η is infinite, then $\eta\eta' = \eta$, for each η' (in particular, $\varphi[\eta] = [_\varphi \eta\,]_\varphi = [_\varphi \eta)$.

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

Denotational Semantics of History Expressions

$$[\![\varepsilon]\!]_\rho = \varepsilon \qquad [\![\alpha]\!]_\rho = \alpha \qquad [\![h]\!]_\rho = \rho(h) \qquad [\![\varphi[H]]\!]_\rho = \varphi[[\![H]\!]_\rho]$$

$$[\![H \cdot H']\!]_\rho = [\![H]\!]_\rho\, [\![H']\!]_\rho \qquad [\![H + H']\!]_\rho = [\![H]\!]_\rho \cup [\![H']\!]_\rho$$

$$[\![\mu h.H]\!]_\rho = \bigcup_{n \in \omega} f^n(\emptyset) \quad \text{where } f(X) = [\![H]\!]_{\rho\{X/h\}}$$

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 nested framings of φ. For instance, $\alpha\varphi[\alpha], \varphi[\alpha]\varphi[\alpha\varphi[\alpha]] \in [\![H]\!]_\emptyset$.

We now introduce a type and effect system [19] for $\lambda^{[]}$, extending [18]. Types and type environments, ranged over by τ and Γ, are defined as follows.

Types and Type Environments

$$\tau \;::=\; \mathit{unit} \mid \tau \xrightarrow{H} \tau \qquad\qquad \Gamma \;::=\; \emptyset \mid \Gamma; x : \tau \quad (x \notin \mathit{dom}(\Gamma))$$

A typing judgment $\Gamma, H \vdash e : \tau$ means that the expression e evaluates to a value of type τ, and produces a history belonging to the effect H. The history expression H in the functional type $\tau \xrightarrow{H} \tau'$ describes the latent effect associated with an abstraction, i.e. one of the histories in $[\![H]\!]$ is generated when a value is

applied to an abstraction with that type. The relation $\Gamma, H \vdash e : \tau$ is defined as the least relation closed under the following rules.

Type and Effect System for $\lambda^{[]}$

$$\frac{}{\Gamma, \varepsilon \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma, \alpha \vdash \alpha : \mathit{unit}} \qquad \frac{}{\Gamma, \varepsilon \vdash * : \mathit{unit}}$$

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \varepsilon \vdash \lambda_z x.e : \tau \xrightarrow{H} \tau'} \qquad \frac{\Gamma, H \vdash e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash ee' : \tau'}$$

$$\frac{\Gamma, H \vdash e : \tau \quad \Gamma, H \vdash e' : \tau}{\Gamma, H \vdash \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau} \qquad \frac{\Gamma, H \vdash e : \tau}{\Gamma, \varphi[H] \vdash \varphi[e] : \tau} \qquad \frac{\Gamma, H \vdash e : \tau}{\Gamma, H + H' \vdash e : \tau}$$

Typing judgments are standard. The last rule allows for *weakening* of effects. The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics. The rule for abstraction constraints the premise to equate the effect and the latent effect of functional type. Let η be a history; let $\eta^\flat$ be the subsequence of η containing only events in Σ; and let η^π be the set of all the prefixes of η. For example, if $\eta = \alpha\varphi[\alpha'\varphi'[\alpha'']]$, then $(\eta^\flat)^\pi = \{\alpha, \alpha\alpha', \alpha\alpha'\alpha''\}$. The next theorem ensures that our type and effect system does approximate the actual run-time histories (its proof, as well as others, and further technical details can be found in [3]).

Theorem 1. *If $\Gamma, H \vdash e : \tau$ and $\varepsilon, e \to^* \eta, e'$, then $\eta \in (\llbracket H \rrbracket^\flat)^\pi$.*

We now define when an access control history is valid. Intuitively, valid histories represent viable computations, while invalid ones represent computations that would have been stopped by the access control mechanism of $\lambda^{[]}$. Let $\eta = \beta_1 \cdots \beta_n$ be a history. Let $\Phi(\eta)$ be the set of the policies φ such that the number of $[_\varphi$ is greater than the number of $]_\varphi$ in η. We say that η is *valid* when $(\beta_1 \cdots \beta_k)^\flat \models \bigwedge \Phi(\beta_1 \cdots \beta_k)$, for all $k \in 1..n$. A history expression H is *valid* when all the histories in $\llbracket H \rrbracket$ are such.

For example, consider the history $\eta_0 = \alpha_r\varphi[\alpha_c]$, where φ is the property saying that no α_c occurs after α_r. Then, η_0 is *not* valid, because $(\alpha_r[_\varphi\alpha_c)^\flat = \alpha_r\alpha_c$ does not satisfy $\bigwedge \Phi(\alpha_r[_\varphi\alpha_c) = \varphi$. The history $\eta_1 = \varphi[\alpha_r]\alpha_c$ is valid, because $([_\varphi\alpha_r)^\flat = \alpha_r$ satisfies $\bigwedge \Phi([_\varphi\alpha_r) = \varphi$, and $\bigwedge \Phi(\eta_1) = \bigwedge \emptyset = \mathit{true}$.

We now state the type safety property. We say that e *goes wrong* when $\varepsilon, e \to^* \eta', e'$, and e' is not a value, and there is no η'', e'' such that $\eta', e' \to \eta'', e''$. For example, a computation goes wrong when attempting to execute an event forbidden by a currently active policy framing.

Theorem 2 (Type Safety). *Let $\Gamma, H \vdash e : \tau$, for e closed. If H is valid, then e does not go wrong.*

Proof (Sketch). The proof is greatly simplified by defining a new transition system with transition relation $\twoheadrightarrow$, where the special framing events $[_\varphi$ and $]_\varphi$ replace the policy framing $\varphi[\cdots]$. The original and the new transition systems do agree step by step, up to obvious transformations on expressions (to convert policy framings into framing events) and on histories (to insert framing events in histories). Similarly, we introduce a type and effect system with relation $\Gamma, H \vdash_\sharp$, much in the style of $\vdash$ above. Indeed, if $\Gamma, H \vdash e : \tau$ then $\Gamma, H \vdash_\sharp e : tau$. Then, we prove first a Subject Reduction lemma, ensuring that, if $\Gamma, H_0 \vdash_\sharp e_0 : \tau$ and $\eta_0, e_0 \twoheadrightarrow \eta_1, e_1$, for e_0 closed and well-formed, then there exists H_1 such that $\Gamma, H_1 \vdash_\sharp e_1 : \tau$ and $\eta_1[\![H_1]\!] \subseteq \eta_0[\![H_0]\!]$. Secondly, we prove a Progress lemma, stating that if $\Gamma, H \vdash_\sharp e : \tau$, for e closed, and let ηH is valid for some η, then, either e is a value, or there exists a transition $\eta, e \twoheadrightarrow \eta', e'$.

Now type safety follows by contradiction. Assume that $\varepsilon, e \to^* \eta, e_0$, and η, e_0 is a stuck configuration, i.e. e_0 is not a value, and there is no transition from η, e_0. By the Subject Reduction lemma, $\Gamma, H' \vdash_\sharp e_0 : \tau$, for some H' such that $\eta[\![H']\!] \subseteq [\![H]\!]$. Since H is valid by hypothesis, then $\eta[\![H']\!]$ is valid, as well as η, because validity is a prefix-closed property. We assumed that η, e_0 is stuck, then by the Progress lemma, e_0 must be a value: contradiction. □

Example 1. Consider the following expression, where b and b' are boolean guards:

$$e \;=\; \lambda_z x.\, \mathtt{if}\ b\ \mathtt{then}\ \alpha\ \mathtt{else}\ (\mathtt{if}\ b'\ \mathtt{then}\ zzx\ \mathtt{else}\ \varphi[zx])$$

Let $\Gamma = \{z : \tau \xrightarrow{H} \tau, x : \tau\}$. Then, the following typing derivation is possible:

$$\dfrac{\dfrac{}{\Gamma, \alpha \vdash \alpha : unit} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \varepsilon \vdash z : \tau \xrightarrow{H} \tau} \quad \dfrac{}{\Gamma, \varepsilon \vdash x : \tau}}{\Gamma, H \vdash z\, x : \tau}}{\Gamma, H \cdot H \vdash z\, z\, x : \tau}}{\Gamma, H \cdot H + \varphi[H] \vdash z\, z\, x : \tau} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \varepsilon \vdash z : \tau \xrightarrow{H} \tau} \quad \dfrac{}{\Gamma, \varepsilon \vdash x : \tau}}{\Gamma, H \vdash z\, x : \tau}}{\Gamma, \varphi[H] \vdash \varphi[z\, x] : \tau}}{\Gamma, H \cdot H + \varphi[H] \vdash \varphi[z\, x] : \tau}}{\Gamma, H \cdot H + \varphi[H] \vdash \mathtt{if}\ b'\ \mathtt{then}\ z\, z\, x\ \mathtt{else}\ \varphi[z\, x] : \tau}$$

$$\Gamma, \alpha + H \cdot H + \varphi[H] \vdash \mathtt{if}\ b\ \mathtt{then}\ \alpha\ \mathtt{else}\ (\mathtt{if}\ b'\ \mathtt{then}\ z\, z\, x\ \mathtt{else}\ \varphi[z\, x]) : \tau$$

To apply the rule for abstraction, the constraint $H = \alpha + H \cdot H + \varphi[H]$ must be solved. A solution is $\mu h.\, \alpha + h \cdot h + \varphi[h]$, thus $\emptyset, \varepsilon \vdash e : unit \xrightarrow{\mu h.\, \alpha + h \cdot h + \varphi[h]} unit$. Note in passing that a simple extension of the type inference algorithm of [18] suffices for solving constraints as the one above.

4 Verification of History Expressions

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. The standard decision procedure for verifying that a BPA process p satisfies a ω-regular property φ amounts to constructing the pushdown automaton for p and the Büchi automaton for the negation of φ. Then, the property holds if the (context-free) language accepted by the conjunction of the above, which is still a pushdown automaton, is empty. This problem is known to be decidable, and several algorithms and tools show this approach feasible [9].

Recall that our notion of validity is non-regular, and that context-free languages are not closed under intersection, thus making the emptiness problem undecidable. We then need to manipulate history expressions in order to make validity a regular property. Indeed, the intersection of a context-free language and a regular language is context-free, so emptiness is decidable.

4.1 Regularization of History Expressions

History expressions can generate histories with *redundant framings*, i.e. nestings of the same policy framing. For example, the history $\eta = \varphi[\alpha\varphi'[\varphi[\alpha']]]$ has an inner redundant φ-framing around the event α'. Since α' is already under the scope of the outermost φ-framing, it happens that η is valid if and only if $\varphi[\alpha\varphi'[\alpha']]$ is valid. While removing inner redundant framings from a history preserves its validity, one needs the expressive power of a pushdown automaton, because open and closed framings are to be matched in pairs. Below, we introduce a transformation that, given a history expression H, yields an 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 $h^* \in fv(H)$ be a selected occurrence of h in H. We say that h^* is guarded by $\mathit{guard}(h^*, H)$, defined as the smallest set satisfying the following equations.

Guards

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

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

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

$$\varepsilon\downarrow_{\Phi,\Gamma} = \varepsilon \qquad h\downarrow_{\Phi,\Gamma} = h \qquad \alpha\downarrow_{\Phi,\Gamma} = \alpha$$

$$(H\cdot H')\downarrow_{\Phi,\Gamma} = H\downarrow_{\Phi,\Gamma}\cdot H'\downarrow_{\Phi,\Gamma} \qquad (H+H')\downarrow_{\Phi,\Gamma} = H\downarrow_{\Phi,\Gamma} + H'\downarrow_{\Phi,\Gamma}$$

$$\varphi[H]\downarrow_{\Phi,\Gamma} = \begin{cases} H\downarrow_{\Phi,\Gamma} & \text{if } \varphi\in\Phi \\ \varphi[H\downarrow_{\Phi\cup\{\varphi\},\Gamma}] & \text{otherwise} \end{cases}$$

$$(\mu h.\, H)\downarrow_{\Phi,\Gamma} = \mu h.\, (H'\sigma'\downarrow_{\Phi,\Gamma\{(\mu h.H)\Gamma/h\}}\sigma)$$

where $H = H'\{h/h_i\}_i$, h_i fresh, $h\notin fv(H')$, and

$$\sigma(h_i) = (\mu h.H)\Gamma\downarrow_{\Phi\cup guard(h_i,H'),\Gamma} \qquad \sigma'(h_i) = \begin{cases} h & \text{if } guard(h_i,H')\subseteq\Phi \\ h_i & \text{otherwise} \end{cases}$$

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 3 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 can be easily turned into a terminating rewriting system.

Example 2. 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 $guard(h_2,H')=\{\varphi\}\not\subseteq\emptyset$:

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

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 $[\![H_0\downarrow]\!] = \left(\{\alpha\}^\omega\varphi\,[\{\alpha\}^\omega\,]\right)^\omega$ has no redundant framings.

Example 3. Let $H_0 = \mu h.\, H_1$, where $H_1 = \mu h'.\, H_2$, and $H_2 = \alpha + h \cdot \varphi[h']$. Since $\mathit{guard}(h, H_1) = \emptyset$, 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 $\mathit{guard}(h_0, H_2') = \{\varphi\} \not\subseteq \emptyset$, it follows that:

$$\begin{aligned} H_1{\downarrow_{\emptyset,\{H_0/h\}}} &= \mu h'.\, H_2'{\downarrow_{\emptyset,\{H_0/h, H_1\{H_0/h\}/h'\}}}\ \{H_1\{H_0/h\}{\downarrow_{\varphi,\{H_0/h\}}}/h_0\} \\ &= \mu h'.\, \alpha + h \cdot \varphi[h_0]\ \{(\mu h'.\, \alpha + H_0 \cdot \varphi[h']){\downarrow_{\varphi,\{H_0/h\}}}/h_0\} \\ &= \mu h'.\, \alpha + h \cdot \varphi[H_3{\downarrow_{\varphi,\{H/h\}}}] \;=\; \alpha + h \cdot \varphi[H_3{\downarrow_{\varphi,\{H/h\}}}] \end{aligned}$$

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

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

Since $\{\varphi\}$ contains both $\mathit{guard}(h, H_1) = \emptyset$, and $\mathit{guard}(h', H_2) = \{\varphi\}$, then:

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

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 now establish the following basic property of regularization.

Theorem 3. *$H\downarrow$ has no redundant framings.*

Regularization preserves validity. To prove that, 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. Note that normalization (as well as regularization) are non-regular transformations: constructing the normal form of a history requires counting the framing openings and closings (see the last equation below): a pushdown automaton is therefore needed.

Normalization of Histories

$$\varepsilon{\Downarrow_{\Phi}} = \varepsilon \qquad (\mathcal{H}\mathcal{H}'){\Downarrow_{\Phi}} = \mathcal{H}{\Downarrow_{\Phi}}\,\mathcal{H}'{\Downarrow_{\Phi}} \qquad (\mathcal{H} \cup \mathcal{H}'){\Downarrow_{\Phi}} = \mathcal{H}{\Downarrow_{\Phi}} \cup \mathcal{H}'{\Downarrow_{\Phi}}$$

$$\alpha{\Downarrow_{\Phi}} = \big(\bigwedge \Phi\big)\,[\alpha] \qquad \varphi[\mathcal{H}]{\Downarrow_{\Phi}} = \mathcal{H}{\Downarrow_{\Phi \cup \{\varphi\}}}$$

Intuitively, normalization transforms histories with policy framings in histories with local checks. Indeed, $\eta{\Downarrow_{\Phi}}$ is intended to record that each event in η must obey *all* the policies in Φ. This is apparent in the second and in the last equation above. 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 4. Consider the history $\eta = \alpha\varphi[\alpha'\varphi'[\alpha'']]$. Its normal form is:

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

A history expression H and its regularization $H\downarrow$ have the same normal form.

Theorem 4. $[\![H\downarrow]\!]\Downarrow = [\![H]\!]\Downarrow$.

The next theorem states that normalization preserves the validity of histories. Summing up, a history expression H is valid iff its regularization $H\downarrow$ is valid.

Theorem 5. *A history η is valid if and only if $\eta\Downarrow$ is valid.*

4.2 Basic Process Algebras

Basic Process Algebras [5] 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 $\Delta = \{X \stackrel{def}{=} p\}$ of guarded definitions, such that, for each variable X, there exists a single, guarded p such that $\{X \stackrel{def}{=} p\} \in \Delta$. 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

$$\frac{}{\alpha \stackrel{\alpha}{\longrightarrow} \varepsilon} \qquad \frac{p \stackrel{\alpha}{\longrightarrow} p'}{p+q \stackrel{\alpha}{\longrightarrow} p'} \qquad \frac{q \stackrel{\alpha}{\longrightarrow} q'}{p+q \stackrel{\alpha}{\longrightarrow} q'} \qquad \frac{p \stackrel{\alpha}{\longrightarrow} p'}{p\cdot q \stackrel{\alpha}{\longrightarrow} p'\cdot q} \qquad \frac{p \stackrel{\alpha}{\longrightarrow} p' \quad X \stackrel{def}{=} p \in \Delta}{X \stackrel{\alpha}{\longrightarrow} p'}$$

The set $\{\,(a_i)_i \mid p_0 \stackrel{a_1}{\longrightarrow} \cdots \stackrel{a_i}{\longrightarrow} p_i\,\} \cup \{\,(a_i)_i \mid p_0 \cdots \stackrel{a_i}{\longrightarrow} \cdots\,\}$ is denoted by $[\![p_0, \Delta]\!]$, where $[\![p, \Delta]\!]^{fin}$ is the first set, containing the strings that label finite computations. We omit the component Δ, when empty.

We now introduce a mapping from history expressions to BPAs, in the line of [18]. Without loss of generality, we assume that all the variables in H have distinct names. The mapping takes as input a history expression H and an injective function Γ from history variables h to BPA variables X, and it 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 mapped to a fresh BPA variable X, bound to the translation of H in the set of definitions Δ. An expression $\varphi[H]$ is mapped to the BPA for H, surrounded by the opening and closing of the φ-framing.

Mapping History Expressions to BPAs

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

We now state the correspondence between history expressions and BPAs. The prefixes of the histories generated by a history expression H (i.e. $[\![H]\!]^\pi$) 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 1. $[\![H]\!]^\pi = [\![BPA(H)]\!]^{fin}$.

4.3 Büchi Automata

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 [21] for details. Since we also need to establish the validity of finite histories, we use the standard trick of padding a finite string 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 $A_\varphi = \langle \Sigma, Q, Q_0, \rho, F \rangle$, which we assume to have a non-final sink state. We define the formula $\varphi_{[\,]}$ through the following Büchi automaton $A_{\varphi_{[\,]}}$.

Büchi Automaton for $\varphi_{[]}$

$$
\begin{aligned}
A_{\varphi_{[]}} &= \langle \Sigma', Q', Q_0, \rho', F' \rangle \\
\Sigma' &= \Sigma \cup \{\, [_\varphi,]_\varphi \mid \varphi \in \Pi \,\} \\
Q' &= F' = Q \cup \{\, q' \mid q \in F \,\} \\
\rho' &= \rho \;\cup\; \{\, \langle q, [_\varphi, q' \rangle \mid q \in F \,\} \;\cup\; \{\langle q',]_\varphi, q \rangle\} \\
&\qquad \cup\; \{\, \langle q'_0, \alpha, q'_1 \rangle \mid \langle q_0, \alpha, q_1 \rangle \in \rho \text{ and } q_1 \in F \,\} \\
&\qquad \cup\; \{\, \langle q, [_{\varphi'}, q \rangle \cup \langle q,]_{\varphi'}, q \rangle \mid q \in Q' \text{ and } \varphi' \neq \varphi \,\}
\end{aligned}
$$

Intuitively, $A_{\varphi_{[]}}$ has two layers. The first 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 5. Let φ be the policy saying that no event α_c can occur after an α_r. The Büchi automata for φ and for $\varphi_{[]}$ are in Figure 1. For example, the history $[_\varphi \alpha_r]_\varphi \alpha_c$ is accepted by $A_{\varphi_{[]}}$, while $\alpha_r [_\varphi \alpha_c]_\varphi$ is not (recall that we do not draw the self-loops labelled by \$).

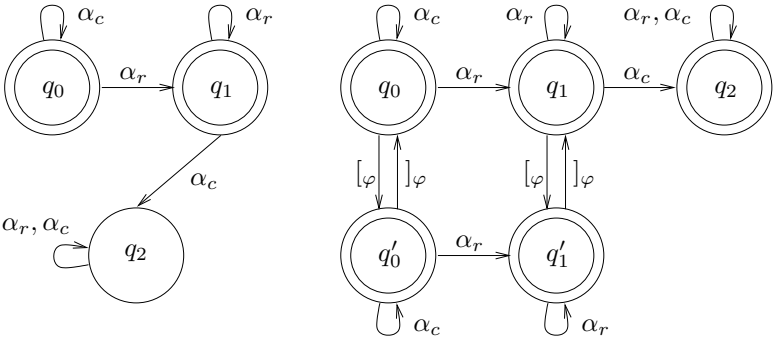

Fig. 1. Büchi automata for φ (left) and for $\varphi[\,]$ (right)

We now relate validity of histories with the formulae $\varphi_{[]}$. Since BPAs can generate infinite histories, we extend by continuity our notion of validity, saying that an *infinite* history is valid when all its *finite* prefixes are valid. Assuming continuity is not a limitation, because validity is a *safety* property: nothing bad

can happen in any execution step [17]. The following lemma states that a history η is valid if and only if it satisfies $\varphi_{[]}$ for all the policies φ spanning over η.

Lemma 2. *Let η be a history with no redundant framings. Then, η is valid if and only if $\eta \models \varphi_{[]}$, for all φ such that $[_\varphi \in \eta$.*

Büchi automata are closed under intersection [21]: 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. Together with Theorem 2, a $\lambda^{[]}$ expression never violates security if its effect is checked valid.

Theorem 6. *$[\![H]\!]$ is valid if and only if $[\![BPA(H\downarrow)]\!] \models \bigwedge_{\varphi \in H} \varphi_{[]}$.*

Proof. By lemma 5, $[\![H]\!]$ is valid if and only if $[\![H]\!]\Downarrow$ is valid. By theorem 4, $[\![H]\!]\Downarrow = [\![H\downarrow]\!]\Downarrow$. By lemma 5, $[\![H\downarrow]\!]\Downarrow$ is valid if and only if $[\![H\downarrow]\!]$ is valid. By theorem 3, $[\![H\downarrow]\!]$ has no redundant framings. By definition, $[\![H\downarrow]\!]$ is valid if and only if $[\![H\downarrow]\!]^\pi$ is valid. By lemma 1, $[\![H\downarrow]\!]^\pi = [\![BPA(H\downarrow)]\!]^{fin}$. By continuity, $[\![BPA(H\downarrow)]\!]^{fin}$ is valid if and only if $[\![BPA(H\downarrow)]\!]$ is valid. Then, by lemma 2, $[\![BPA(H\downarrow)]\!]$ is valid if and only if $[\![BPA(H\downarrow)]\!] \models \bigwedge_{\varphi \in H} \varphi[]$.

5 Conclusions

We proposed a novel approach to history-based access control. To this aim, we have introduced $\lambda^{[]}$, an extension of the λ-calculus that allows for security policies with a local scope. Along the lines of Skalka and Smith [18], we have used a type and effect system to extract from a given program a history expression that approximates its run-time behaviour. Verifying the validity of a history expression ensures that there will be no security violations at run-time. Our security policies are regular properties of histories; however, the augmented flexibility due to nesting of scopes makes validity a non-regular property, unlike [18]. So, $\lambda^{[]}$ is expressive enough to describe and enforce security policies that cannot be naturally dealt with local checks or global policies. 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.

Our model is less general than the resource access control framework of Igarashi and Kobayashi [12], but we provide a static verification technique, while [12] does not. We have no explicit notion of resource, as they have, but we plan to introduce it in the future.

Compared to Skalka and Smith's λ_{hist}, our $\lambda^{[]}$ features a different programming construct for access control. The programming model and the type system of [18] also allow for access events parametrized by constants, and for let-polymorphism. Although omitted for simplicity, these features can be easily recovered by using the same techniques of [18]. As a matter of fact, λ_{hist} turns out

to be the sub-calculus of $\lambda^{[]}$ where the scope of policies can only include single events. Intuitively, a framing $\varphi[*]$ corresponds to a local check of the regular policy φ on the current history. It is not always possible to transform a program in $\lambda^{[]}$ into a program in λ_{hist} that obeys the same security properties, provided that the transformation is only allowed to substitute suitable local checks for policy framings. Clearly, unrestricted transformations, (e.g. security-passing style ones that record the set of active framings) can do the job, because λ_{hist} is Turing complete.

Our policy framings roughly resemble the scoped methods of [20]. This construct extends the Java source language by allowing programmers to limit the sequence of methods that may be applied to an object. A scoped method is annotated with a regular expression which describe the permitted sequences of access events. Methods must explicitly declare the sequence of events they may produce, while we infer them by a type and effect system.

Colcombet and Fradet [7] and Marriot, Stuckey and Sulzmann [13] mix static and dynamic techniques to transform programs in order to make them obey a given safety property. Compared to [7, 13], our programming model allows for local policies, while the other only considers global ones. In future work, we aim at investigating if a similar mixed approach is feasible in our programming model. This might be non-trivial, because local policies seem to make the techniques used in [7, 13] not directly exploitable.

Acknowledgments

We wish to thank the anonymous referees for their insightful comments and suggestions. This work has been partially supported by EU project DEGAS (IST-2001-32072) and FET project PROFUNDIS (IST-2001-33100).

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, Submitted.
4. M. Bartoletti, P. Degano, and G. L. Ferrari. Method inlining in the presence of stack inspection. In *Workshop on Issues in the Theory of Security*, 2004.
5. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.
6. 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.
7. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.
8. G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Secure Internet Programming*, volume 1603 of *LNCS*, 1999.

9. J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.
10. P. W. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, 2004.
11. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.
12. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.
13. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. First Asian Programming Languages Symposium*, 2003.
14. A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on selected areas in communication*, 21(1), 2003.
15. P. Samarati and S. de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD Tutorial Lectures*, volume 2171 of *LNCS*. Springer, 2001.
16. F. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*. Springer, 2001.
17. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.
18. C. Skalka and S. Smith. History effects and verification. In *Asian Programming Languages Symposium*, 2004.
19. J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. 7th IEEE Symposium on Logic in Computer Science*, 1992.
20. G. Tan, X. Ou, and D. Walker. Resource usage analysis via scoped methods. In *Foundations of Object-Oriented Languages*, 2003.
21. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. Banff Higher order workshop conference on Logics for concurrency*, 1996.
22. D. Walker. A type system for expressive security policies. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.