Local Policies for Resource Usage Analysis

MASSIMO BARTOLETTI
Università degli Studi di Cagliari, and Università di Pisa
PIERPAOLO DEGANO and GIAN-LUIGI FERRARI
Università di Pisa
and
ROBERTO ZUNINO
Università degli Studi di Trento

An extension of the λ-calculus is proposed, to study resource usage analysis and verification. It features usage policies with a possibly nested, local scope, and dynamic creation of resources. We define a type and effect system that, given a program, extracts a history expression, that is, a sound overapproximation to the set of histories obtainable at runtime. After a suitable transformation, history expressions are model-checked for validity. A program is resource-safe if its history expression is verified valid: If such, no runtime monitor is needed to safely drive its executions.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis; operational semantics*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages, Security, Theory, Verification

Additional Key Words and Phrases: Usage policies, type and effect systems, model-checking

ACM Reference Format:
Bartoletti, M., Degano, P., Ferrari, G.-L., and Zunino, R. 2009. Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31, 6, Article 23 (August 2009), 43 pages.
DOI = 10.1145/1552309.1552313 http://doi.acm.org/10.1145/1552309.1552313

This work has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers) and by the MIUR-PRIN project SOFT (Tecniche Formali Orientate alla Sicurezza).
Authors' addresses: M. Bartoletti, Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Via Ospedale, 72, I-09124 Cagliari, Italy; email: bart@unica.it; P. Degano, G.-L. Ferrari, Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo, 3, I-56127 Pisa, Italy; email: {degano, giangi}@di.unipi.it; R. Zunino, Dipartimento di Ingegneria e Scienza dell'Informazione, Università degli Studi di Trento, Via Sommarive, 14, I-38100 Povo, Italy; email: zunino@disi.unitn.it.

DOI 10.1145/1552309.1552313 http://doi.acm.org/10.1145/1552309.1552313

1. INTRODUCTION

An important aspect of programming language design and implementation is how to ensure that resources are used correctly. The problem is made even more crucial by the current programming trends, which provide us with mechanisms for reusing code, and exploiting services and components, offered by (possibly untrusted) third parties. It is common practice to pick from the Web some scripts, plugins, or packages, and assemble them into a bigger program, with little or no control about the security of the whole.

The typical runtime mechanisms for enforcing resource usage policies are *execution monitors*, which abort executions whenever they are about to violate the prescribed usage policy. The events observed by these monitors are accesses to sensible resources, such as opening socket connections, reading/writing files, allocating/deallocating memory. A main issue is finding a compromise between the expressivity of usage policies and the efficiency of the enforcement mechanism. Static analysis techniques may be applied to improve efficiency, but this often results in an unacceptable restriction of the expressive power of policies.

A common mechanism for enforcing usage policies consists in guarding with *local checks* the program points where critical resources can be accessed [Fournet and Gordon 2003; Skalka and Smith 2004]. Local checks have a main drawback: They must be explicitly inserted into code by the programmer. Since forgetting even a single check might compromise the safety of the whole application, programmers have to inspect very carefully their code. This may be cumbersome even for small programs, and it may easily lead to unnecessary checking.

A safer approach is that of *global policies*, where the execution monitor enforces a global invariant that must hold at any point of the execution. This may involve guarding each resource access, and ad hoc optimizations are then in order to recover efficiency, for example, compiling the global policy to local checks [Colcombet and Fradet 2000; Marriott et al. 2003]. Furthermore, a large monolithic policy may be hard to understand, and not very flexible either. Indeed, it is necessary to imagine all the possible resource usage scenarios in advance. If an unexpected situation occurs at runtime (e.g., downloading a piece of mobile code with specific resource usage requirements), the global policy must be dynamically updated, if possible at all.

A more flexible approach consists in attaching usage policies to resources, as in Igarashi and Kobayashi [2002], so to adapt them to the context where a resource is used. For example, it may be possible to restrain the capabilities before calling untrusted code. However, a limitation of this approach is that you can only control the usage of resources you have created. In a mobile code scenario, for example, a browser that runs untrusted applets, it is also important that you can impose constraints on how external programs manage the resources created in your local environment. For instance, an applet may create an unbounded number of resources on the browser site, and never release them. This clearly leads to denial-of-service attacks that may eventually crash the whole system.

We consider here an abstract language that aims at reconciling expressivity of resource usage policies with efficiency of the enforcement mechanism. This language, called $\lambda^{[\,]}$(lambda-box), has primitives for creating and accessing resources, and for defining *local* resource usage policies. Sequences of resource accesses in executions are called *histories*; a *policy* is (roughly) a regular property of histories, defined through our *usage automata*. A program fragment e protected by a policy φ is written $\varphi[e]$, called *policy framing*. Roughly, while evaluating e, the histories must respect the policy φ. Of course, policy framings can be nested.

Local policies generalize both local checks and global policies. They smoothly allow for safe composition of programs with their own private policies, also in mobile code scenarios. Indeed, there is no need to dynamically accommodate the local private policies into a single global one, possibly invalidating syntax-directed optimizations of policy enforcement. Local policies may offer protection also in the Web-services scenario [Bartoletti et al. 2006]: There, one has scarce control on the code to run, and thus inserting local checks is infeasible. For example, a browser must obey a usage policy specified by the user. Additionally, the browser can invoke a policy provider to obtain a stricter security policy, used for dynamically sandboxing applets. This rich interplay between policies seems difficult to express in the aforementioned approaches.

In $\lambda^{[\,]}$, efficiency of resource usage control is obtained through a suitable combination of static techniques. We introduce a type and effect system that overapproximates the runtime usage behavior of a program. It infers as effect a *history expression* that denotes all the possible histories resulting from executions. A history expression is *valid* when all the histories it denotes obey the relevant policies; a program with a valid history expression will never be aborted at runtime by the execution monitor. Constructing the history expression of a program is a basic step in our approach: Indeed, checking that a program obeys the usage policies requires knowing all its possible histories in their entirety; validity is *not* compositional. Validity of history expressions is thus verified subsequently through model-checking, for which we devised a sound and complete algorithm.

Indeed, having a complete verification technique like ours is a desirable feature, since, for example, it guarantees termination of the analysis (which is not always the case with incomplete techniques, like Igarashi and Kobayashi [2002] and Chaki et al. [2002]). Also, completeness provides us with a precise characterization of the policies that are statically enforceable. We design the language of effects expressive enough to accurately approximate program behavior, and at the same time simple enough to allow for complete verification. Our history expressions are basic process algebra processes [Bergstra and Klop 1985], extended with name restriction *à la* π-calculus [Milner et al. 1992]. Note in passing that polynomial-time algorithms are known for model-checking Basic Process Algebras, while for more expressive calculi (like, e.g., Petri nets, pushdown systems, etc.) model-checking is typically intractable or undecidable [Mayr 1998]. Further advantages of our two-phase approach (typing + model-checking) are the possibility of independently improving the accuracy

of the effects and the efficiency of the verification procedure, as well as the possibility of verifying a variety of policies over the same type.

We started our investigation on history-based access control in Bartoletti et al. [2005c], which we extend here with dynamic creation of resources (a preliminary version is in Bartoletti et al. [2007]). This apparently little extension demands for addressing the general problem of handling fresh names. In particular, it is necessary to correctly bind the creation of new resources to their usages. The solution to this problem deeply affects the techniques of Bartoletti et al. [2005c], with respect to the following points: (i) the usage policies and their enforcement mechanism, (ii) the type and effect system, and (iii) the verification technique. For the first point, we introduce our notion of "parametric" usage policies, that constrain the access to resources to obey a regular property. For instance, a file usage policy $\varphi(x)$ might require that "before reading or writing a file x, that file must have been opened, and not closed". Enforcing a usage policy $\varphi(x)$ is accomplished through the finite state automaton obtained by instantiating the formal parameter x to an actual resource r. For (ii), the problem is to correctly record the binding of fresh names in types and effects. Technically, we explore a novel approach to quantify types over freshly created resources, a sort of polymorphism à la ML on *both* types and effects. For (iii), the creation of new resources may give rise to an infinite number of formulae to be inspected while verifying validity. We solve this problem by suitably grouping resources with equivalent usage constraints. This allows us to extract from a history expression a Basic Process Algebra process and a regular formula, to be used in model-checking validity [Esparza 1994].

A key point of our proposal is that we offer a comprehensive framework for safely handling resources, within a linguistic setting. Our calculus offers an expressive and flexible way to protect code with usage policies. Resource usage control is made feasible by suitably extending and integrating techniques from type theory and model-checking.

The article is organized as follows. We first present some local policies at work on securing some small, yet illustrative, case studies. In Section 3 we formally introduce our usage policies, the language $\lambda^{[\,]}$, and its operational semantics. Section 4 defines history expressions, their operational semantics, the subeffecting relation, and validity. The type and effect system is presented in Section 5, along with the theorems stating the correctness of effects, and type safety. Our model-checking technique follows in Section 6, together with the main results, namely its correctness and completeness. We then discuss some related work, and we conclude by presenting some possible extensions to ours.

To relieve the reader from the burden of technicalities, we moved the proofs of our statements, and some additional examples, to the Appendixes. To help intuition, some insights on the proofs are, however, included in the main text.

2. SOME EXAMPLES

To illustrate our approach, we consider a simple Web browser scenario. A browser B downloads applets from the Web W, and runs them under suitable policies, obtained by a policy provider P on a per-applet basis. For the sake of

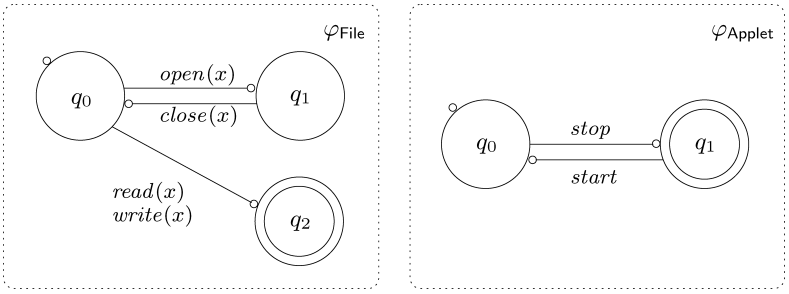

Fig. 1. File usage policy (left) and Applet policy (right).

simplicity, we shall consider some kinds of violations, only. Of course, a real scenario will require to take care of several other violations. We abstractly describe the browser as the following $\lambda^{[]}$expression.

$$B \;=\; \lambda u, x.\, \varphi_{\mathsf{File}}[\mathit{start}; (P\ u)\ (W\ u)\ x; \mathit{stop}]$$

The file usage policy φ_{File} is defined through the *usage automaton* $\varphi_{\mathsf{File}}(x)$ in Figure 1 (left), and it is always enforced. It guarantees that only open files can be read or written. For each file f, its usage must be accepted by the finite state automaton $\varphi_{\mathsf{File}}(f)$, obtained by instantiating the formal parameter x to the actual file f. To stress the difference between the parametric policy $\varphi_{\mathsf{File}}(x)$ and the concrete policy acting on a given file f, $\varphi_{\mathsf{File}}(f)$, we use the symbol —◦ for edges in the former, and $\rightarrow$ in the latter. The initial state q_0 represents the file being closed, while q_1 is for an open file. Reading and writing x in q_0 leads to the offending state q_2, modeling policy violation. Instead, in q_1 you can read and write x: Each state has (nondisplayed) self-loops for the events not explicitly mentioned among its outgoing edges.

The browser B takes as input the parameter x, a value to be passed to the applet. The parameter u is the URL from which the applet is obtained through the function W that models the Web, a rather partial and pessimistic view of the Web, actually: There are five URLs, three of which (u_{Bonk}, u_{Spam} and u_{DoS}) contain attackers.

$$\begin{aligned} W \;=\; \lambda u.\, \mathit{connect}(u);\ & \texttt{if}\ u = u_{Bank}\ \texttt{then}\ A_{Bank} \\ & \texttt{else if}\ u = u_{Bonk}\ \texttt{then}\ A_{Bonk} \\ & \texttt{else if}\ u = u_{Edit}\ \texttt{then}\ A_{Edit} \\ & \texttt{else if}\ u = u_{Spam}\ \texttt{then}\ A_{Spam} \\ & \texttt{else if}\ u = u_{DoS}\ \texttt{then}\ A_{DoS} \end{aligned}$$

The downloaded applet is run under the policy given by the provider P. Since policies are not first-class entities in our calculus, the provider returns back a closure $\lambda a, x.\, \varphi[a\, x]$, where a will be bound to the applet, x to its parameter, and

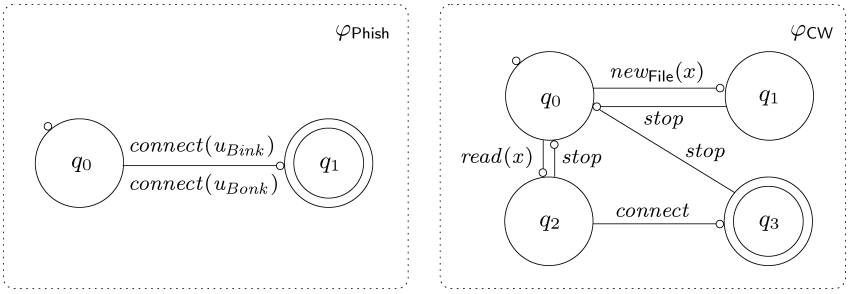

Fig. 2. Antiphishing policy (left) and Chinese Wall policy (right).

φ will be enforced while executing the applet.

$$
\begin{aligned}
P \;=\; \lambda u, a, x.\, \texttt{if}\ \mathit{Trusted}\, u\ \texttt{then}\ a\, x \\
\texttt{else}\ \varphi_{\mathsf{Applet}}[\texttt{if}\ \ u = u_{Edit}\ \texttt{then}\ \varphi_{\mathsf{CW}}[a\, x] \\
\texttt{else if}\ u = u_{Spam}\ \texttt{then}\ \varphi_{\mathsf{Spam}}[a\, x] \\
\texttt{else if}\ u = u_{DoS}\ \texttt{then}\ \varphi_{\mathsf{DoS}}[a\, x]\ ]
\end{aligned}
$$

Back to the browser B, each applet execution is delimited by the marker events *start* and *stop*. The *stop* marker will be exploited in the policies φ_{CW}, φ_{Spam}, φ_{DoS} (more details to follow). Since these policies rely on the *stop* event to represent the actual termination of the applet, untrusted applets must not be allowed to fire *stop* while they are still running. To this purpose, we use the policy $\varphi_{\mathsf{Applet}}$ (see Figure 1, right). A *stop* event leads to a policy violation. A *start* event instead resets the policy $\varphi_{\mathsf{Applet}}$: this is done before entering the $\varphi_{\mathsf{Applet}}$ policy framing to make the policy "forget" past *stop* events. Finally, note that the event *start*, unlike *stop*, is used in no other policies, and so there is no need to prevent applets from firing it.

We now show how local policies may help in driving secure execution of untrusted applets.

Phishing. Consider an attacker that attempts to fraudulently obtain sensitive information from the customers of an online bank. The bank applet A_{Bank} allows a customer to obtain a new PIN number for his ATM card. The applet takes as input a username, and returns a freshly generated PIN. To protect against phishing attacks, the bank applet constrains its executions to the policy φ_{Phish} in Figure 2 (left). This policy implements a blacklist of known phishers sites (e.g., u_{Bink} and u_{Bonk}). A phisher A_{Bonk} attempts a homograph attack by masquerading as the bank to acquire the PIN number. The A_{Bonk} applet takes as input a username, forwards it to the Bank so to obtain a PIN, stores it, and eventually returns the PIN to the customer.

$$
\begin{aligned}
A_{Bank} \;&=\; \lambda x.\, \varphi_{\mathsf{Phish}}[\mathit{login}(x); \texttt{new}\ \mathit{pin}\ \texttt{in}\ \cdots; \mathit{pin}] \\
A_{Bonk} \;&=\; \lambda x.\, \texttt{let}\ \mathit{pin} = (W\, u_{Bank})x\ \texttt{in}\ \mathit{store}(\mathit{pin}); \cdots; \mathit{pin}
\end{aligned}
$$

We now show an execution of the Bonk applet, assuming that the policy provider considers both Bank and Bonk as trusted. The program states are pairs. The first component is the execution history, that is, the sequence of events occurred so far (ε being the empty one). The second component is the program continuation, to be evaluated according to the call-by-value rule. Additionally, an expression enclosed in a policy framing can be reduced only if the history respects the policy. In case of nested framings, all the enclosing policies must be obeyed.

$$
\begin{aligned}
& \varepsilon,\ \underline{B\ u_{Bonk}\ \text{“Bob”}} \\
\rightarrow\ & \varepsilon,\ \varphi_{\mathsf{File}}[\,\underline{start};(P\ u_{Bonk})\,(W\ u_{Bonk})\,\text{“Bob”};stop] \\
\rightarrow^*\ & start,\ \varphi_{\mathsf{File}}[(\lambda a, x.\ a\ x)\,\underline{(W\ u_{Bonk})\,\text{“Bob”}};stop] \\
\rightarrow^*\ & start\ connect(u_{Bonk}),\ \varphi_{\mathsf{File}}[\underline{(\lambda a, x.\ a\ x)\,A_{Bonk}\ \text{“Bob”}};stop] \\
\rightarrow^*\ & start\ connect(u_{Bonk}),\ \varphi_{\mathsf{File}}[\underline{A_{Bonk}\ \text{“Bob”}};stop] \\
\rightarrow^*\ & start\ connect(u_{Bonk}),\ \varphi_{\mathsf{File}}[\texttt{let}\ pin = \underline{(W\ u_{Bank})\text{“Bob”}}\ \texttt{in}\ \cdots] \\
\rightarrow^*\ & start\ connect(u_{Bonk})\ connect(u_{Bank}),\ \varphi_{\mathsf{File}}[\texttt{let}\ pin = \varphi_{\mathsf{Phish}}[\underline{login(\text{“Bob”})};\cdots \\
\rightarrow^*\ & start\ connect(u_{Bonk})\ connect(u_{Bank})\ login(\text{“Bob”}),\ fail_{\varphi_{\mathsf{Phish}}}
\end{aligned}
$$

The execution aborts just before the customer logins, because the active policy φ_{Phish} is violated. Indeed, the history.

$$start\ connect(u_{Bonk})\ connect(u_{Bank})\ login(\text{“Bob”})$$

leads to the offending state q_1 in the automaton φ_{Phish} (recall that we do not display self-loops for the events not mentioned in outgoing edges).

Chinese Wall. Consider a simple text editor applet A_{Edit} that takes as input a file x, opens it, and then performs some operations. If the file x is not present on the local disk, then it is downloaded into a new, local file f from the same site where A_{Edit} comes. Eventually, f is uploaded to the same site. Assume that the policy provider associates u_{Edit} with a “Chinese Wall” policy φ_{CW} (Figure 2, right). The policy says that the applet can no longer connect to the network, after having read a file it has not created. The dots that follow stand for code with no events.

$$
\begin{aligned}
A_{Edit} &= \lambda x.\ \texttt{if}\ Local\,x \\
&\qquad \texttt{then}\ open(x);\cdots;read(x);\cdots;write(x);close(x) \\
&\qquad \texttt{elsenew}\ f : \mathsf{File}\ \texttt{in}\ (Download\,f\,x);read(f);\cdots;(Upload\,f\,x) \\
Download &= \lambda f, x.\ connect(u_{Edit});open(f);get(x);\cdots;write(f) \\
Upload &= \lambda f, x.\ connect(u_{Edit});open(f);read(f);\cdots;put(x)
\end{aligned}
$$

The following trace illustrates the behavior of the editor applet, first invoked on a local file f_L, and then on a remote file f_R

$$
\begin{aligned}
& \varepsilon,\ \underline{(B\ u_{Edit}\ f_L)};(B\ u_{Edit}\ f_R) \\
\rightarrow\ & \varepsilon,\ \varphi_{\mathsf{File}}[\underline{start};(P\ u_{Edit})](W\ u_{Edit})\ f_L;stop];(B\ u_{Edit}\ f_R) \\
\rightarrow^*\ & start,\ \varphi_{\mathsf{File}}[\underline{(\lambda a, x.\ \varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[a\ x]])(W\ u_{Edit})\ f_L};stop];(B\ u_{Edit}\ f_R) \\
\rightarrow^*\ & start\ connect(u_{Edit}),\ \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[\underline{A_{Edit}\ f_L}]];stop];(B\ u_{Edit}\ f_R) \\
\rightarrow^*\ & start\ connect(u_{Edit})\ open(f_L)\ read(f_L)\ write(f_L)\ close(f_L), \\
& \varphi_{\mathsf{File}}[\underline{\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[*]]};stop];(B\ u_{Edit}\ f_R) \\
\rightarrow^*\ & start\ connect(u_{Edit})\ open(f_L)\ read(f_L)\ write(f_L)\ close(f_L), \\
& \varphi_{\mathsf{File}}[\underline{stop}];(B\ u_{Edit}\ f_R) \\
\rightarrow^*\ & start\ connect(u_{Edit})\ open(f_L)\ read(f_L)\ write(f_L)\ close(f_L)\ stop, \\
& \underline{B\ u_{Edit}\ f_R}
\end{aligned}
$$

where $*$ stands for no-operation. This fragment of computation shows the successful execution of the editor on the local file. No security violations occur because $\eta_0 = start\ (u_{Edit})\ open(f_L)\ read(f_L)\ write(f_L)\ close(f_L)\ stop$ complies with φ_{File}, $\varphi_{\mathsf{Applet}}$, and φ_{CW}. In particular, φ_{CW} is obeyed because no *connect* occurs after the $read(f_L)$, within the framing (the automaton $\varphi_{\mathsf{CW}}(f_L)$ stays in state q_2). Note that upon completion of $(A_{Edit}\ f_L)$, the scope of the policies $\varphi_{\mathsf{Applet}}$ and φ_{CW} is left, while the scope of φ_{File} is left after the *stop* event. The execution proceeds as follows.

$$
\begin{aligned}
\rightarrow^*\ & \eta_0\ start,\ \varphi_{\mathsf{File}}[\underline{(\lambda a, x.\ \varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[a\ x]])(W\ u_{Edit})\ f_R};stop] \\
\rightarrow^*\ & \eta_0\ start\ connect(u_{Edit}),\ \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[\underline{A_{Edit}\ f_R}]];stop] \\
\rightarrow^*\ & \eta_0\ start\ connect(u_{Edit})\ new_{\mathsf{File}}(f), \\
& \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[\underline{(Download\ f\ f_R);read(f)};(Upload\ f\ f_R)]];stop]
\end{aligned}
$$

Let $\eta_1 = \eta_0\ start\ connect(u_{Edit})\ new_{\mathsf{File}}(f)\ open(f)\ get(f_R)\ write(f)$. The computation proceeds as follows.

$$
\begin{aligned}
\rightarrow^*\ & \eta_1\ read(f),\ \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[\underline{Upload\ f\ f_R}]];stop] \\
\rightarrow^*\ & \eta_1\ read(f)\ connect(u_{Edit}), \\
& \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{CW}}[open(f);read(f);\cdots;put(f_R)]];stop]
\end{aligned}
$$

Also this second fragment of computation is successful. Although a *connect* event happened after a $read(f)$, the file f was a freshly generated one, thus the execution conforms to φ_{CW} (the automaton $\varphi_{\mathsf{CW}}(f_R)$ stays in state q_1, while $\varphi_{\mathsf{CW}}(f_L)$ is reset). Note that the *stop* event in η_0 resets the policy φ_{CW}, making it discard the events generated by the previous applet execution (i.e., the events in η_0). This is crucial, otherwise the last *connect* in η_1 (which comes after the $read(f_L)$ event of η_0) would cause a violation.

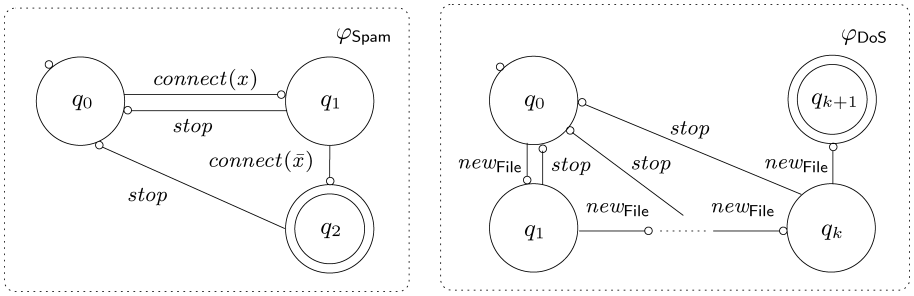

Fig. 3. Antispam policy (left) and Denial-of-Service policy (right).

Spamming. Consider a Trojan-horse applet A_{Spam} that maliciously attempts to send spam email through the mail server of the browser site. The applet first connects to the site it was downloaded from, so to implement some apparently useful and harmless activity. Then, the applet inquires the local host to check if relaying of emails is allowed: If so, it connects to the local SMTP server to send unsolicited bulk emails.

$$A_{Spam} \;=\; \lambda x.\, connect(u_{Spam}); \cdots ; \texttt{if}\ RelayAllowed\ \texttt{then}\ connect(u_{SMTP})$$

The policy provider associates with u_{Spam} the policy φ_{Spam} (Figure 3, left) that prevents the applet from connecting to two different URLs. The label $connect(\bar{x})$ in the edge from q_1 to q_2 means that a transition from q_1 to q_2 is possible for all the events *connect* performed on any socket $\bar{x} \neq x$. The following computation shows that the applet is blocked just before opening the connection with the SMTP server.

$$\begin{array}{ll}
 & \varepsilon,\ \underline{B\ u_{Spam}\ *} \\
\rightarrow & \varepsilon,\ \varphi_{\mathsf{File}}[start; \underline{(P\ u_{Spam})\,(W\ u_{Spam})\,*}; stop]; \cdots \\
\rightarrow^* & start,\ \varphi_{\mathsf{File}}[\underline{(\lambda a, x.\, \varphi_{\mathsf{Applet}}[\varphi_{\mathsf{Spam}}[a\ x]])\,(W\ u_{Spam})\,*}; stop] \\
\rightarrow^* & start\ connect(u_{Spam}),\ \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{Spam}}[\underline{A_{Spam}\ *}]]; stop] \\
\rightarrow^* & start\ connect(u_{Spam})\ connect(u_{Spam}), \\
 & \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{Spam}}[\underline{\texttt{if}\ RelayAllowed\ \texttt{then}\ connect(u_{SMTP})}]]; stop] \\
\rightarrow^* & start\ connect(u_{Spam})\ connect(u_{Spam}), \\
 & \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{Spam}}[\underline{connect(u_{SMTP})}]]; stop] \\
\rightarrow^* & start\ connect(u_{Spam})\ connect(u_{Spam}),\ fail_{\varphi_{\mathsf{Spam}}}
\end{array}$$

Denial of Service. Consider an applet A_{DoS} that attempts to perform a denial-of-service attack on the browser site. To do that, it recursively creates local files, and writes some data to exhaust the disk space.

$$A_{DoS} \;=\; \texttt{new}\ f : \mathsf{File}\ \texttt{in}\ open(f); write(f); A_{DoS}\ x$$

The policy provider associates to u_{DoS} the policy φ_{DoS} (Figure 3, right), that prevents the applet from creating more than k files. Any execution of the A_{DoS}

applet is aborted just before it attempts to create the $k+1$-th file, as shown by the following computation.

$$
\begin{aligned}
& \varepsilon,\ \underline{B\ u_{DoS}\ *} \\
\rightarrow\ & \varepsilon,\ \varphi_{\mathsf{File}}[\underline{start;(P\ u_{DoS})}\,(W\ u_{DoS})\,*;stop];\cdots \\
\rightarrow^*\ & start,\ \varphi_{\mathsf{File}}[\underline{(\lambda a,x.\,\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{DoS}}[a\ x]])\,(W\ u_{DoS})\,*};stop] \\
\rightarrow^*\ & start\ connect(u_{DoS}),\ \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{DoS}}[\underline{A_{DoS}\ *}]]);stop] \\
\rightarrow^*\ & start\ connect(u_{DoS}), \\
& \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{DoS}}[\underline{\texttt{new}\ f\ :\mathsf{File}\ \texttt{in}\ open(f);write(f);A_{DoS}\ *}]]);stop] \\
\rightarrow^*\ & start\ connect(u_{DoS})\ new_{\mathsf{File}}(f_1), \\
& \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{DoS}}[\underline{open(f_1);write(f_1);A_{DoS}\ *}]]);stop] \\
\rightarrow^*\ & start\ connect(u_{DoS})\ new_{\mathsf{File}}(f_1)\ open(f_1)\ write(f_1)\cdots new_{\mathsf{File}}(f_k), \\
& \varphi_{\mathsf{File}}[\varphi_{\mathsf{Applet}}[\varphi_{\mathsf{DoS}}[\underline{open(f_k);write(f_k);A_{DoS}\ *}]]);stop] \\
\rightarrow^*\ & start\ connect(u_{DoS})\ new_{\mathsf{File}}(f_1)\cdots new_{\mathsf{File}}(f_k)\ open(f_k)\ write(f_k),\ fail_{\varphi_{\mathsf{DoS}}}
\end{aligned}
$$

Note that the applet A_{DoS} cannot reset the policy φ_{Spam} through a malicious *stop* event, because otherwise it would violate the outer policy $\varphi_{\mathsf{Applet}}$.

3. PROGRAMMING MODEL

We consider a call-by-value λ-calculus enriched with primitives for creating and accessing resources, and with local usage policies. *Resources* are system objects that can either be already available in the environment (Res_s, a finite set), or be created dynamically (Res_d, an infinite denumerable set). Resources can be accessed through a given finite set of *actions*. This set is partitioned to reflect the kinds of resources (e.g., $\mathsf{Act} = \mathsf{File} \cup \mathsf{Socket} \cup \cdots$), where each element of the partition contains the actions admissible for the given kind (e.g., $\mathsf{File} = \{new_{\mathsf{File}}, open, close, read, write\}$). The action new_{Act_i} represents the creation of a new resource of kind Act_i. We assume a global *capability environment* Γ_s, namely a relation between static resources and the actions they admit. An *event* $\alpha(r)$ abstracts from accessing the resource r through the action α.

When working with a real-world language, we expect these events to be extracted through an abstraction phase respecting the side-effects of the language semantics. In case the target resource of an action α is immaterial, we stipulate that α acts on some special (static) resource, and we write just α for the event. When the resource accessed through α is unknown, we write $\alpha(?)$, where the special resource ? will be typically due to approximations made by the type and effect system. A *history* is a finite sequence of events. In Definition 3.1 we introduce the needed syntactic categories and some notation.

Definition 3.1. (*Syntactic Categories*).

$r, r', \ldots \in \mathsf{Res} = \mathsf{Res}_s \cup \mathsf{Res}_d$	resources (static/dynamic)
$\alpha, \alpha', new_{\mathsf{File}}, \ldots \in \mathsf{Act} = \bigcup_i \mathsf{Act}_i$	actions (a finite set)
$\alpha(r), \ldots \in \mathsf{Ev} = \mathsf{Act} \times (\mathsf{Res} \cup \{?\})$	events
$\varphi, \varphi', \ldots \in \mathsf{Pol}$	usage policies
$[_\varphi,]_\varphi, [_{\varphi'},]_{\varphi'}, \ldots \in \mathsf{Frm}$	framing events
$\eta, \eta', \ldots \in (\mathsf{Ev} \cup \mathsf{Frm})^*$	histories (ε is the empty history)
$x, x', \ldots \in \mathsf{Var}$	variables
$\Gamma, \Gamma', \ldots \subseteq \mathsf{Res} \times \mathsf{Act}$	capability environments
$\Gamma_s \subseteq \mathsf{Res}_s \times \mathsf{Act}$	static capability environment

3.1 Usage Policies

A *usage policy* φ is a set of histories, specified through a *usage automaton* $\varphi(x)$. Usage automata can be seen as an extension of Finite State Automata (FSA), where the labels on the edges may contain variables $x \in \mathsf{Var}$, which represents a universally quantified resource. Variables can also be "complemented", for instances, $\bar{x}$ represents any possible resource different from x. A usage automaton $\varphi(x)$ gives rise to an automaton $\varphi(r)$ with finite states, when the formal parameter x is instantiated to an actual resource r. This instantiation will be exploited to specify the set of histories obeying the usage policy φ.

Definition 3.2 (*Usage Automata*). A usage automaton $\varphi(x)$ is a 5-tuple $\langle S, Q, q_0, F, E\rangle$, where:

—$S \subset \mathsf{Act} \times (\mathsf{Res}_s \cup \{x, \bar{x}\})$ is the input alphabet,
—Q is a *finite* set of states,
—$q_0 \in Q \setminus F$ is the start state,
—$F \subset Q$ is the set of final "offending" states, and
—$E \subseteq Q \times S \times Q$ is a finite set of edges, written $q \overset{\vartheta}{\multimap} q'$.

The edges in a usage automaton can be of three kinds: $q \overset{\alpha(r)}{\multimap} q'$ where r is a static resource, $q \overset{\alpha(x)}{\multimap} q'$, and $q \overset{\alpha(\bar{x})}{\multimap} q'$ (where $\bar{x}$ means "different from x").

Given a resource $r \in \mathsf{Res}$ and a set of resources R, a usage automaton $\varphi(x)$ is instantiated into a automaton $A_{\varphi(r,R)}$ with finite states by binding x to r, and, accordingly, making $\bar{x}$ range over each resource in $R \setminus \{r\}$. Roughly, $q \overset{\alpha(x)}{\multimap} q'$ will give rise to a transition $q \overset{\alpha(r)}{\longrightarrow} q'$, $q \overset{\alpha(\bar{x})}{\multimap} q'$ will give rise to a set of transitions $q \overset{\alpha(r')}{\longrightarrow} q'$ for all $r' \in R \setminus \{r\}$, and, when r_s is a static resource, $q \overset{\alpha(r_s)}{\multimap} q'$ will yield $q \overset{\alpha(r_s)}{\longrightarrow} q'$. The preceding is made precise in Definition 3.3.

Definition 3.3 (*Policy Compliance*). Let $\varphi(x) = \langle S, Q, q_0, F, E\rangle$ be a usage policy, let $r \in \mathsf{Res}$, and let $R \subseteq \mathsf{Res}$. The *usage automaton* $A_{\varphi(r,R)} =$

$\langle \Sigma, Q, q_0, F, \delta \rangle$ is defined as

$$\Sigma = \{\alpha(r') \mid \alpha \in \mathsf{Act} \text{ and } r' \in R\}$$

$$\delta = \dot{\delta} \cup \{q \xrightarrow{\alpha(?)} q' \mid \exists \zeta \in R : q \xrightarrow{\alpha(\zeta)} q' \in \dot{\delta}\} \qquad \text{(unknown resources)}$$

$$\dot{\delta} = \ddot{\delta} \cup \{q \xrightarrow{\alpha(r')} q \mid r' \in R, \nexists q' : q \xrightarrow{\alpha(r')} q' \in \ddot{\delta}\} \qquad \text{(self-loops)}$$

where the relation $\ddot{\delta}$ is defined as follows.

$$\begin{aligned} \ddot{\delta} \;=\; & \{q \xrightarrow{\alpha(r)} q' \mid q \xrightarrow{\alpha(x)}\!\!\circ\; q' \in E\} && \text{(instantiation of } x\text{)} \\ \cup\; & \bigcup_{r' \in (R \cup \{?\}) \setminus \{r\}} \{q \xrightarrow{\alpha(r')} q' \mid q \xrightarrow{\alpha(\bar{x})}\!\!\circ\; q' \in E\} && \text{(instantiation of } \bar{x}\text{)} \\ \cup\; & \{q \xrightarrow{\alpha(r')} q' \mid q \xrightarrow{\alpha(r')}\!\!\circ\; q' \in E\} && \text{(static resources)} \end{aligned}$$

We write $\eta \lhd A_{\varphi(r,R)}$ when η is not in the language of the automaton $A_{\varphi(r,R)}$.

We say that η *respects* φ, written $\eta \models \varphi$, when $\eta \lhd A_{\varphi(r,\mathsf{Res})}$, for all $r \in \mathsf{Res}$. Otherwise, we say that η *violates* φ, written $\eta \not\models \varphi$.

The auxiliary relation $\ddot{\delta}$: (i) instantiates x to the given resource r, (ii) instantiates $\bar{x}$ with all $r' \in R$ such that $r' \neq r$, and (iii) maintains the transitions $\alpha(r')$ for r' static. The auxiliary relation $\dot{\delta}$ adds self-loops for all the events not explicitly mentioned in the usage automaton. Finally, the relation δ adds to $\dot{\delta}$ the transitions to deal with the unknown resource ?. Intuitively, any transition labelled $\alpha(r)$ in $\dot{\delta}$ can also be played by $\alpha(?)$.

Note that usage automata can be nondeterministic, in the sense that for all states q and input symbols $\alpha(\rho)$, the set of states

$$\{q' \mid q \xrightarrow{\alpha(\rho)}\!\!\circ\; q'\} \qquad \rho \in \mathsf{Res}_s \cup \{x, \bar{x}\}$$

is not required to be a singleton. Given a trace η, we want that *all* the paths of the instances $A_{\varphi(x,R)}$ comply with φ, that is, they lead to a nonfinal state. Recall that we used final states to represent violations to the policy. An alternative approach would be the "default-deny" one, that allows for specifying the permitted usage patterns, instead of the denied ones. We could deal with this approach be regarding the final states as accepting. Both the default-deny and the default-allow approaches have known advantages and drawbacks. We think this form of demonic nondeterminism is particularly convenient when designing usage automata, since it makes possible to focus on the sequences of events that lead to violations, while neglecting those that do not affect the compliance to the policy. All the needed self-loops will be added by the relation $\dot{\delta}$ in Definition 3.3.

We now show that the compliance of a history with a policy is a decidable property. According to Definition 3.3, to check $\eta \models \varphi$ we should instantiate the formal parameter x of a usage automaton $\varphi(x)$ with infinitely many resources (one for each $r \in \mathsf{Res}$). The resulting automata have a finite number of states, yet they may have infinitely many transitions, for instance, instantiating $q \xrightarrow{\alpha(\bar{x})}\!\!\circ\; q'$ originates $q \xrightarrow{\alpha(r)}\!\!\circ\; q'$ for all but one $r \in \mathsf{Res}$. A similar situation

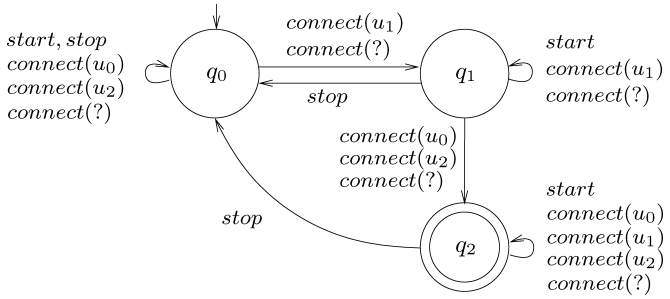

Fig. 4. Automaton $A_{\varphi_{\mathsf{Spam}}(u_1,\{u_0,u_1,u_2\})}$.

happens for self-loops. The next lemma shows that checking $\eta \models \varphi$ can be decided through a *finite* set of *finite* state automata.

LEMMA 3.4. *Let η be a history, let φ be a usage policy, and let $\mathsf{R}(\eta)$ be the set of resources occurring in η. Then, $\eta \models \varphi$ if and only if $\eta \lhd A_{\varphi(r,\mathsf{R}(\eta))}$ for each $r \in \mathsf{R}(\eta) \cup \{\bar{r}\}$, where $\bar{r}$ is an arbitrary resource in $\mathsf{Res} \setminus \mathsf{R}(\eta)$.*

The decidability of $\models$ follows from the fact that resources not occurring in the given history are indistinguishable. Indeed, the actual identity of such resources is immaterial: Whenever $r', r'' \notin \mathsf{R}(\eta)$, we have that $\eta \lhd A_{\varphi(r',R)}$ if and only if $\eta \lhd A_{\varphi(r'',R)}$, and $\eta \lhd A_{\varphi(r,R\cup\{r'\})}$ if and only if $\eta \lhd A_{\varphi(r,R\cup\{r''\}}$ for all r and R. Therefore, we can safely limit our attention to the resources in $\mathsf{R}(\eta)$, and "collapse" all the resources in $\mathsf{Res} \setminus \mathsf{R}(\eta)$ into a single resource $\bar{r} \notin \mathsf{R}(\eta)$ (see also Example 3.2). Note in passing that this additional resource resembles the additional, fresh name used to check bisimilarity between π-calculus processes [Dam 1997].

Example 3.1. Consider from Section 2 the policy φ_{Spam}. Let $R = \{u_0, u_1, u_2\}$. The automaton $A_{\varphi_{\mathsf{Spam}}(r_1,R)}$ is in Figure 4 (the self-loops for events other than *start*, *stop*, and *connect* are omitted). Note that this automaton is nondeterministic, because of the transitions labeled *connect*(?). Consider now the history

$$\eta \;=\; \mathit{start}\;\mathit{connect}(u_0)\;\mathit{stop}\;\mathit{start}\;\mathit{connect}(u_1)\;\mathit{connect}(u_2).$$

Since η drives an offending run in the automaton $A_{\varphi_{\mathsf{Spam}}(u_1,\{u_0,u_1,u_2\})}$, then we have $\eta \not\models \varphi_{\mathsf{Spam}}$. This correctly models the fact that the policy φ_{Spam} prevents applets from connecting to two different sites. Observe that removing the last event $\mathit{connect}(u_2)$ from η would make the history obey φ_{Spam}, since the *stop* event resets the policy φ_{Spam} to the starting state. Actually, $\eta' = \mathit{start}\;\mathit{connect}(u_0)\;\mathit{stop}\;\mathit{start}\;\mathit{connect}(u_1)$ is *not* in the language of any instantiations $A_{\varphi_{\mathsf{Spam}}(r,R)}$ (i.e., $\eta' \lhd A_{\varphi_{\mathsf{Spam}}(r,R)}$), for all $r \in \mathsf{Res}$ and $R \subseteq \mathsf{Res}$.

Example 3.2. Consider the policy $\varphi_{\neg\alpha}$ specified by the usage automaton in Figure 5. Actually, $\varphi_{\neg\alpha}$ forbids any α actions: When a history η contains some event $\alpha(r)$, the instantiation $A_{\varphi_{\neg\alpha}(r',\mathsf{Res})}$ recognizes η as offending, for all $r' \neq r$ (indeed, the same behavior would have been obtained with the usage automaton $q_0 \overset{\alpha(x)}{\longrightarrow\!\!\circ}\; q_1$). Therefore, $\eta \not\models \varphi_{\neg\alpha}$. Consider, for example, $\eta = \alpha(r_0)\,\beta(r_0)$. Although

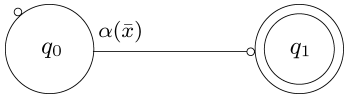

Fig. 5. Usage policy $\varphi_{\neg\alpha}$.

$\eta \lhd A_{\varphi(r,\mathsf{R}(\eta))}$ for all $r \in \mathsf{R}(\eta) = \{r_0\}$, as noted before η violates $\varphi_{\neg\alpha}$. This motivates Lemma 3.4 checking $A_{\varphi(r',\mathsf{R}(\eta))}$ also for an extra resource $r' \in \mathsf{Res} \setminus \mathsf{R}(\eta)$.

For simplicity, in this article we only consider monadic usage automata, namely automata with a *single* formal parameter, such as $\varphi(x)$. The extension to the polyadic case, for example, $\varphi(x_1, \ldots, x_n)$, is considered in Bartoletti [2009]. While polyadic usage automata augment the set of expressible usage policies, they only required smooth extensions of the formal machinery presented here. The results of this article scale to the polyadic case, yet requiring further technicalities in their proofs.

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

The syntax of $\lambda^{[]}$ is in Definition 3.3. Variables, applications, and conditionals are as expected. The variable z in $\lambda_z x.e$ is bound to the whole abstraction, so to allow for an explicit form of recursion. The definition of guards b in conditionals is irrelevant here, and so it is omitted. An event $\alpha(r)$ is an explicit representation of a side-effect that occurs in evaluation when accessing the resource r through the action α. The expression $\texttt{new}\ \gamma$ evaluates to a fresh resource r with capabilities γ, a finite set of actions that are permitted on this resource. When this happens a $new(r)$ event is fired; we also assume that *new* events cannot be explicitly written in programs. The primitive $\varphi[e]$ defines the scope of the policy φ, that must be enforced while evaluating e.

Definition 3.3 (*Syntax of* $\lambda^{[]}$ *Expressions*).

e, e' ::=	x	variable	($x \in \mathsf{Var}$)
	r	resource	($r \in \mathsf{Res}$)
	$\alpha(\xi)$	event	($\alpha \in \mathsf{Act}$, $\xi \in \mathsf{Var} \cup \mathsf{Res}$)
	$\texttt{new}\ \gamma$	resource creation	($\gamma \subset \mathsf{Act}$)
	$\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e'$	conditional	
	$\lambda_z x.e$	abstraction	
	$e\,e'$	application	
	$\varphi[e]$	policy framing	($\varphi \in \mathsf{Pol}$)

We specify the behavior of $\lambda^{[]}$expressions in Definition 3.6, through a small-step operational semantics. Besides the standard error configurations, for example, a stuck one where a resource is applied to an expression, we have two different kinds of policy violations. To make them explicit, in configurations we also allow the following special expressions. A failure $fail_\kappa$ occurs when about to access a resource with a nonpermitted action. A failure $fail_{\varphi[e]}$ is raised when e is about to violate the policy φ. A transition $\eta, \Gamma, e \to \eta', \Gamma', e'$ means that, starting from a history η and a capability environment Γ, the expression e may evolve

to e' (possibly a failure), the history η to η', and the capability environment Γ to Γ'.

Remark 3.1. Let *fail* denote both kinds of failures. For all expressions e, policies φ, and actions α, we assume that $fail = efail = faile = \varphi[fail]$. We write $*$ for a fixed, closed, access-free, nonfailure value, and use the following abbreviations.

$$\lambda_z.e = \lambda_z x.e \quad \text{if } x \notin fv(e) \qquad \lambda x.e = \lambda_z x.e \quad \text{if } z \notin fv(e)$$

$$e;e' = (\lambda.e')e \qquad \texttt{new}\ x : \gamma\ \texttt{in}\ e = (\lambda x.e)(\texttt{new}\ \gamma)$$

$$(\texttt{let}\ x = e\ \texttt{in}\ e') = (\lambda x.e')e \qquad \alpha(e) = (\texttt{let}\ z = e\ \texttt{in}\ \alpha(z))$$

Definition 3.6 (*Operational Semantics of* $\lambda^{[\,]}$). The semantics of $\lambda^{[\,]}$is given by a transition system. Values $v, v', \ldots \in \mathsf{Val}$ are variables, resources, abstractions, and failures. Configurations are triples η, Γ, e, where e is closed. Initial configurations have the form ε, Γ_s, e. The transition relation is the minimal relation defined by the inference rules that follow.

$$\textsc{E-App1}\ \frac{\eta, \Gamma, e_1 \to \eta', \Gamma', e_1'}{\eta, \Gamma, e_1 e_2 \to \eta', \Gamma', e_1' e_2} \qquad \textsc{E-App2}\ \frac{\eta, \Gamma, e_2 \to \eta', \Gamma', e_2'}{\eta, \Gamma, v\, e_2 \to \eta', \Gamma', v\, e_2'}$$

$$\textsc{E-Beta}\quad \eta, \Gamma, (\lambda_z x.e)v \to \eta, \Gamma, e\{v/x, \lambda_z x.e/z\}$$

$$\textsc{E-If}\quad \eta, \Gamma, \texttt{if}\ b\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} \to \eta, \Gamma, e_{\mathcal{B}(b)}$$

$$\textsc{E-Ev1}\ \frac{(r, \alpha) \in \Gamma}{\eta, \Gamma, \alpha(r) \to \eta\alpha(r), \Gamma, *} \qquad \textsc{E-Ev2}\ \frac{(r, \alpha) \notin \Gamma}{\eta, \Gamma, \alpha(r) \to \eta, \Gamma, fail_\kappa}$$

$$\textsc{E-Fr1}\ \frac{\eta, \Gamma, e \to \eta', \Gamma', e' \quad \eta' \models \varphi}{\eta, \Gamma, \varphi[e] \to \eta', \Gamma', \varphi[e']} \qquad \textsc{E-Fr2}\ \frac{\eta \models \varphi}{\eta, \Gamma, \varphi[v] \to \eta, \Gamma, v}$$

$$\textsc{E-Fr3}\ \frac{\eta, \Gamma, e \to \eta', \Gamma', e' \quad \eta' \not\models \varphi}{\eta, \Gamma, \varphi[e] \to \eta', \Gamma', fail_{\varphi[e']}}$$

$$\textsc{E-New}\ \frac{\kappa(\gamma) \neq \emptyset \qquad r \in \mathsf{Res}_d \ \text{ fresh in } \eta}{\eta, \Gamma, \texttt{new}\ \gamma \to \eta\, new_{\kappa(\gamma)}(r),\ \Gamma \cup (r, \gamma),\ r}$$

$$\text{where } (r, \gamma) = \{(r, \alpha) \mid \alpha \in \gamma\},\ \text{and } \kappa(\gamma) = \begin{cases} \mathsf{Act}_i & \text{if } \exists i.\, \emptyset \subset \gamma \subseteq \mathsf{Act}_i \\ \emptyset & \text{otherwise} \end{cases}.$$

The rules in Definition 3.6 are mostly standard, except for those governing resource access and creation, and evaluation within policy framings. An access $\alpha(r)$ can be executed if the capabilities associated with r include the action α, otherwise it generates a failure $fail_\kappa$. A new resource r is created through the primitive $\texttt{new}\ \gamma$, which evaluates to a *fresh* resource r and extends the capability

environment Γ. Note that the creation of a resource is explicitly recorded in the execution history through the event $new(r)$. For conditionals, we assume as given a total function $\mathcal{B}$ that evaluates the boolean guards. An expression $\varphi[e]$ can evolve to $\varphi[e']$, provided that the resulting history η' complies with φ. Otherwise, a failure $fail_{\varphi[e']}$ occurs.

The rules for evaluating an expression within the scope of a policy framing (E-Fr1, E-Fr2, E-Fr3) always inspect the *whole* past history. If needed, we can easily limit the scope from the side of the past, as done in Section 2 for the policies φ_{CW}, φ_{Spam}, and φ_{DoS}. Roughly, it is necessary to mark in the history, through a special event β, the point in time from which checking a policy φ has to start. The usage automaton for φ must be designed to ignore all the events before β.

4. HISTORY EXPRESSIONS

We statically predict the histories that will be generated by programs at runtime through a type and effect system, building upon Bartoletti et al. [2007, 2005c] and Skalka and Smith [2004]. The types extend those of the simply-typed λ-calculus, and the effects are *history expressions*, which overapproximate the aspects of the program behavior that are relevant to resource usage. Before formalising in Section 4.1 the syntax and semantics of history expressions, we give some intuition and auxiliary notions.

The intended meaning of a history expression is a set of histories, extended to keep track of the policy framings $\varphi[\cdots]$ through the *framing events* $[_\varphi$ and $]_\varphi$. These special events have no parameter, and they stand, respectively, for opening and closing the scope of φ. For example, a history $\alpha[_\varphi\alpha']_\varphi$ represents a computation that: (i) generates an event α, (ii) enters the scope of a framing $\varphi[\cdots]$, (iii) generates α' within the scope of φ, and (iv) leaves the scope of φ. Note that framing events were not needed in the operational semantics of $\lambda^{[]}$, because the scope of policies is dictated by policy-framed expressions $\varphi[e]$. Unlike the histories produced by the operational semantics, here we also consider histories comprising events with unknown resources.

We define next some operators on histories. The set of histories $\eta^{-?}$ is obtained by instantiating the unknown resource ? with all the possible resources $r \in \mathsf{Res}$. The history $\eta^{-[]}$ results from erasing all the framing events in η.

Definition 4.1 (*Elimination of Framing Events and Instantiation of ?*). Elimination of framing events $[_\varphi,\ ]_\varphi, \ldots \in \mathsf{Frm}$.

$$\varepsilon^{-[]} = \varepsilon \qquad ([_\varphi\eta)^{-[]} = (\,]_\varphi\eta)^{-[]} = \eta^{-[]}$$

Instantiation of ?.

$$\varepsilon^{-?} = \{\varepsilon\} \qquad (\alpha(r)\,\eta)^{-?} = \alpha(r)\,\eta^{-?} \qquad (\alpha(?)\,\eta)^{-?} = \{\alpha(r)\,\eta^{-?} \mid r \in \mathsf{Res}\}$$

4.1 History Expressions: Syntax and Semantics

History expressions are the abstract counterpart of, and overapproximate, the behavior of $\lambda^{[]}$expressions. Their syntax is in Definition 4.2, where Nam is a denumerable set of *names* $n, n', \ldots$, taken apart from the variables in Var.

Definition 4.2 (*Syntax of History Expressions*).

H, H'	::=	ε	empty	
		h	variable	
		$\alpha(\rho)$	event	$(\rho \in \mathsf{Res} \cup \mathsf{Nam} \cup \{?\})$
		$\nu n.H$	resource creation	
		$H \cdot H'$	sequence	
		$H + H'$	choice	
		$\varphi[H]$	policy framing	$(\varphi[H] = [_\varphi \cdot H \cdot]_\varphi)$
		$\mu h.H$	recursion	

History expressions include ε, representing the empty history, events $\alpha(\rho)$, resource creation $\nu n.H$, sequencing $H \cdot H'$, nondeterministic choice $H + H'$, policy framing $\varphi[H]$, and recursion $\mu h.H$. In $\nu n.\, H$, the free occurrences of the name n in H are bound by ν; similarly μh binds the free occurrences of the variable h in H. Sequencing $\cdot$ binds more strongly than choice $+$, that in turn has precedence over resource creation νn and recursion μh.

Definition 4.3 (*Free Variables and Free Names*). The free variables $fv(H)$ are those h in H not inside the scope of a μh.

The free names $fn(H)$ are those n in H not inside the scope of a νn.

A history expression H is *closed* when $fv(H) = \emptyset = fn(H)$.

A history expression H is *initial* when it is closed and it has no dynamic resources, that is, $r \notin \mathsf{Res}_d$ for each $\alpha(r)$ occurring in H.

In Definition 4.4 we give semantics to (closed) history expressions, in terms of a labelled transition system. Intuitively, the semantics of a history expression H is the set of all the prefixes (including the empty history ε) of the traces of H. As usual, $\varepsilon\eta = \eta = \eta\varepsilon$.

Definition 4.4 (*Operational Semantics of History Expressions*). The semantics $[\![H]\!]$ of a closed history expression H is the set of histories

$$\{a_1 \cdots a_i \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_i} H_i\} \qquad (i \geq 0).$$

The auxiliary relation $\xrightarrow{a}$, where $a \in \mathsf{Ev} \cup \mathsf{Frm} \cup \{\varepsilon\}$, is inductively defined as

$$\varepsilon \cdot H \xrightarrow{\varepsilon} H \qquad \alpha(\rho) \xrightarrow{\alpha(\rho)} \varepsilon \qquad \nu n.\, H \xrightarrow{\varepsilon} H\{r/n\} \quad (r \in \mathsf{Res}_d \text{ fresh})$$

$$\frac{H \xrightarrow{a} H''}{H \cdot H' \xrightarrow{a} H'' \cdot H'} \qquad H + H' \xrightarrow{\varepsilon} H \qquad H + H' \xrightarrow{\varepsilon} H' \qquad \mu h.\, H \xrightarrow{\varepsilon} H\{\mu h.\, H/h\}.$$

Example 4.1. Consider the following history expressions.

$$H_0 = \mu h.\, \alpha \cdot h \qquad H_1 = \mu h.\, h \cdot \alpha \qquad H_2 = \mu h.\, \nu n.\, \alpha(n) \cdot h$$

Then, $[\![H_0]\!] = \alpha^*$, that is, H_0 generates histories with an arbitrary number of α. Instead, $[\![H_1]\!] = \{\varepsilon\}$, that is, H_1 loops forever, without generating any events. The semantics of H_2 consists of all the histories of the form $\alpha(r_1) \cdots \alpha(r_k)$ for all $k \geq 0$ and pairwise distinct r_i.

4.2 Validity

We now define when histories and history expressions are valid. Intuitively, valid histories represent viable computations. Instead, invalid ones happen to violate some resource usage constraints, so they are going to be detected and rejected by our static analysis. For example, consider the history

$$\eta \;=\; \mathit{open}(r)\,\mathit{read}(r)\,[_{\varphi_{\mathsf{CW}}}\,\mathit{connect}\,]_{\varphi_{\mathsf{CW}}},$$

where φ_{CW} is the Chinese Wall policy of Section 2. Then, η is *not* valid according to our intended meaning, because the *connect* event occurs within a framing enforcing φ_{CW}, and $\mathit{open}(r)\,\mathit{read}(r)\,\mathit{connect}$ does not obey φ_{CW}. The formal definition of validity is in Definition 4.5.

Definition 4.5 (*Active Policies and Validity*). The multiset $ap(\eta)$ of the active policies of a history η is defined as follows.

$$\begin{array}{ll} ap(\varepsilon) = \{\!|\,|\!\} & ap(\eta\,[_\varphi) = ap(\eta) \cup \{\!|\varphi|\!\} \\ ap(\eta\,\alpha(\rho)) = ap(\eta) & ap(\eta\,]_\varphi) = ap(\eta) \setminus \{\!|\varphi|\!\} \end{array}$$

The validity of a history η ($\models \eta$ in symbols) is inductively defined as follows.

$$\models \varepsilon \qquad \models \eta'\beta \quad \text{if} \quad \models \eta' \text{ and } (\eta'\beta)^{-[\,]} \models \varphi \;\text{ for all } \varphi \in ap(\eta'\beta)$$

A history expression H is *valid* when $\models \eta$ for all $\eta \in [\![H]\!]$.

The following lemma, straightforward after Definition 4.5, states that validity is a prefix-closed property.

LEMMA 4.6. *For all histories η and η', if $\eta\eta'$ is valid, then η is valid.*

Example 4.2. Consider the history $\eta = [_\varphi \alpha_1 [_{\varphi'} \alpha_2]_{\varphi'} \alpha_3$. The active policies of η are $ap(\eta) = \{\!|\varphi|\!\}$. The history η is valid if and only if

$$\begin{array}{llll} \varepsilon \models \varphi & \alpha_1 \models \varphi & \alpha_1\alpha_2 \models \varphi & \alpha_1\alpha_2\alpha_3 \models \varphi \\ & \alpha_1 \models \varphi' & \alpha_1\alpha_2 \models \varphi' & \end{array}$$

Note that validity is *not* a compositional property: In general, $\models \eta$ and $\models \eta'$ do not imply $\models \eta\eta'$ (see the next example). This is a consequence of our basic assumption that no event can be hidden. Indeed, in the operational semantics of $\lambda^{[\,]}$, the policies enforced by policy framings can always inspect the whole history generated so far, at each execution step.

Example 4.3. Consider the history $\eta = \alpha[_\varphi\,\alpha\,]_\varphi\alpha$, where the policy φ requires that α is not executed three times. Since $\alpha \models \varphi$ and $\alpha\alpha \models \varphi$, then η is valid. Note that the first α is checked, even though it is outside of the framing: Since it happens in the past, our policies can inspect it. Instead, the third α occurs after the framing has been closed, therefore it is not checked.

Note that verifying the validity of the history $\alpha\beta[_\varphi\,\gamma\,]_\varphi\delta$ requires to check $\alpha\beta \models \varphi$ and $\alpha\beta\gamma \models \varphi$, only.

Now, consider the history $\eta' = \alpha\eta$. In spite of both α and η being valid, their composition η' is not. To see why, consider the history $\bar\eta = \alpha\alpha[_\varphi\,\alpha$, which is a prefix of η'. Then, $ap(\bar\eta) = \{\!|\varphi|\!\}$, but $\bar\eta^{-[\,]} = \alpha\alpha\alpha \not\models \varphi$. This shows that validity is not compositional.

4.3 Subeffecting

In this section we define a preorder $H \sqsubseteq H'$ betweeen history expressions. Roughly, when $H \sqsubseteq H'$ holds, the traces of H are included in those of H'. We first define an equational theory of history expressions. The operation $+$ is associative, commutative, and idempotent; $\cdot$ is associative, has identity ε, and distributes over $+$. The binders μ and ν allow for α-conversion of bound names and variables, and can be rearranged. A μh can be introduced/eliminated when h does not occur free. A νn can be extruded when it does not bind a free occurrence of n. A $\mu h.\, H$ can be folded/unfolded as usual.

Definition 4.7 (*An Equational Theory of History Expressions*). The relation $=$ over history expressions is the least congruence including α-conversion such that

$$H + H = H \qquad (H + H') + H'' = H + (H' + H'') \qquad H + H' = H' + H$$

$$(H \cdot H') \cdot H'' = H \cdot (H' \cdot H'') \qquad \varepsilon \cdot H = H = H \cdot \varepsilon$$

$$(H + H') \cdot H'' = H \cdot H'' + H' \cdot H'' \qquad H \cdot (H' + H'') = H \cdot H' + H \cdot H''$$

$$\mu h.H = H\{\mu h.\, H/h\} \qquad \mu h.\mu h'.H = \mu h'.\mu h.H \qquad \nu n.\nu n'.H = \nu n'.\nu n.H$$

$$\nu n.\varepsilon = \varepsilon \qquad \nu n.(H + H') = (\nu n.H) + H' \ \text{ if } n \notin fn(H')$$

$$\nu n.(H \cdot H') = \begin{cases} H \cdot (\nu n.H') & \text{if } n \notin fn(H) \\ (\nu n.H) \cdot H' & \text{if } n \notin fn(H'). \end{cases}$$

The preorder $\sqsubseteq$ includes equivalence, and it is closed under contexts. A history expression H can be arbitrarily "weakened" to $H + H'$. An event $\alpha(\rho)$ can be weakened to $\alpha(?)$, as ? stands for an unknown resource.

Definition 4.8 (*Subeffects*). The relation $\sqsubseteq$ over history expressions is the least precongruence such that

$$H \sqsubseteq H' \quad \text{if } H = H' \qquad H \sqsubseteq H + H' \qquad \alpha(\rho) \sqsubseteq \alpha(?).$$

Note that our notion of subeffecting is *structural*, that is, it only depends on the syntax of history expressions. The relation between subeffecting and inclusion of the semantics of the involved history expressions is made precise in Lemma 4.9 that follows. Alternatively, we could have provided a *semantic* notion of subeffecting as language inclusion $[\![H]\!] \subseteq [\![H']\!]$, similarly to Skalka et al. [2008]. We preferred to adopt structural rather than semantic, subeffecting because it allowed us to have some further insights on how type and effect judgements could be deduced in a constructive definition of typing. Note also that semantic subeffecting could be harder here than in Skalka et al. [2008] because of the presence of unknown resources. Actually, an event $\alpha(?)$ with an unknown resource can be concretized, in a subeffecting relation, as $\alpha(r)$ for all resources r.

LEMMA 4.9. *For all closed H, H', if $H \sqsubseteq H'$ then $[\![H]\!]^{-?} \subseteq [\![H']\!]^{-?}$.*

The proof of this lemma is in Appendix B, and it requires some intermediate results. First, we introduce there a denotational semantics of history expressions, which allows for a compositional reasoning about the needed properties. Then, we prove that the denotational and the operational semantics are fully abstract. Note that having an operational semantics of history expressions is, however, convenient for two reasons. First, it is much simpler than the denotational one; second, it is directly comparable to the semantics of BPAs, the structures used in our model-checking technique.

5. TYPE AND EFFECT SYSTEM

In this section we define a type and effect system for $\lambda^{[]}$. The types ζ, formally introduced in Definition 5.1, have the form $\nu N.\, \tau \triangleright H$, meaning that an expression with *pure type* τ will produce a history represented by the effect H, that is, a history expression. The heading νN is used to bind the names $n \in N$ in τ and H. Pure types comprise the following.

—The unit type $\mathbf{1}$, inhabited by the value $*$.

—Pairs (S, γ), where S is either a nonempty set containing resources and possibly one name, or $S = \{?\}$, and γ is a set of capabilities. For instance, a resource r with capabilities α_1 and α_2 may have the type $(\{r\}, \{\alpha_1, \alpha_2\})$. Hereafter, we shall omit the capabilities when irrelevant.
The restriction on pure types to contain at most a single name causes sometimes a less precise typing. This constraint is only needed to keep simple the proof arguments. We conjecture, however, that it can be safely relaxed, preserving all our results.

—Functional types $\tau \to \zeta$. The type ζ may comprise the *latent* effect associated with an abstraction. For instance, applying a function with type $\mathbf{1} \to (\nu n.\mathbf{1} \triangleright \alpha(n))$ to a value with type $\mathbf{1}$, will create a new resource upon which the action α is fired.

Example 5.1. We intuitively illustrate how expressions are typed through three simple examples. Their actual typing derivations are in Example 5.2, after the definition of our type and effect system.

Since in what follows the capability environments play no role, we shall omit them.

—The expression $e = \texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ r'$ can be typed as $\{r, r'\} \triangleright \varepsilon$. The resource set $\{r, r'\}$ means that e will evaluate to either r or r', while producing an empty history (denoted by ε).

—The expression $e' = \lambda y\,.\,\texttt{new}\ x\ \texttt{in}\ x$ can be typed as $\mathbf{1} \to (\nu n.\, \{n\} \triangleright \mathit{new}(n)) \triangleright \varepsilon$. Again, the actual effect is empty. When applied to a value of type $\mathbf{1}$, e' will: (i) create a fresh resource, (ii) fire the *new* event, and (iii) return the resource created. This behavior is predicted by the type $\nu n.\, \{n\} \triangleright \mathit{new}(n)$.

—The expression $e'' = \lambda_z\, y\,.\,\texttt{new}\ x\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y$, instead, can be typed as $\mathbf{1} \to (\{?\} \triangleright \mu h.\, \nu n.\, \mathit{new}(n) \cdot (h + \varepsilon)) \triangleright \varepsilon$. The latent effect $\mu h.\, \nu n.\, \mathit{new}(n) \cdot (h + \varepsilon)$ records that e'' is a recursive function that creates a fresh resource upon each recursion step. The type $\{?\}$ says that e'' will return a resource, the identity of

which is unknown, since it cannot be predicted when the guard b will become true.

Type environments are finite mappings from variables and resources to types. A typing judgment $\Delta \vdash e : \nu N.\, \tau \rhd H$ means that, in a type environment Δ, the expression e evaluates to a value of type $\nu N.\, \tau$, and it produces a history represented by $\nu N.\, H$.

Definition 5.1 (*Types, Type Environments, and Typing Judgements*).

$S ::= R \mid R \cup \{n\} \mid \{?\}$	$R \subseteq \mathsf{Res}, n \in \mathsf{Nam}, S \neq \emptyset$	resource sets
$\tau ::= \mathbf{1} \mid (S, \gamma) \mid \tau \to \zeta$	$\gamma \subseteq \mathsf{Act}$	pure types
$\zeta ::= \nu n.\, \zeta \mid \tau \rhd H$		types
$\Delta ::= \emptyset$		type environments
$\mid \Delta; r : (\{r\}, \Gamma_s(r))$		
$\mid \Delta; x : \tau$	$x \notin dom(\Delta)$	
$\Delta \vdash e : \zeta$		typing judgements

We also introduce the following shorthands.

$N \between M$	if $N \cap M = \emptyset$
$\nu N.\, \zeta = \nu n_1 \cdots \nu n_k.\, \zeta$	if $N = \{n_1, \ldots n_k\}$
$H \cdot \zeta = \nu N.\, \tau \rhd H \cdot H'$	if $\zeta = \nu N.\, \tau \rhd H'$ and $N \between fn(H)$

We say $\nu N.\, \tau \rhd H$ is in *ν-normal form* (abbreviated νNF) when $N \subseteq fn(\tau)$.

We now define the subtyping relation $\sqsubseteq$. It builds over the subeffecting relation between history expressions (Definition 4.8). The first equation is a variant of the usual extrusion. The first two rules for $\sqsubseteq$ allow for weakening a pure type (S, γ) to a wider one, possibly with fewer capabilities, or to the pure type $(\{?\}, \gamma)$. The last rule extends to type the relations $\sqsubseteq$ over pure types and effects. The side condition prevents from introducing name captures. For instance, consider $\zeta = \nu n.\, \{r\} \rhd \alpha(n)$ and $\zeta' = \nu n.\, \{r, n\} \rhd \alpha(n)$. Since $n \in fn(\{r, n\}) \setminus fn(\{r\})$, then $\zeta \not\sqsubseteq \zeta'$. Indeed, by the equational theory we have that

$$\zeta \;=\; \{r\} \rhd \nu n.\, \alpha(n) \;=\; \{r\} \rhd \nu n'.\, \alpha(n')$$

which permits the subtyping $\zeta \sqsubseteq \zeta'' = \{r, n\} \rhd \nu n'.\, \alpha(n')$. Indeed, in ζ'' the name n' upon which α acts has nothing to do with name n in the pure type, while in ζ' both α and the pure type refer to the same name.

Definition 5.2 (*Subtypes*). The equational theory of types includes that of history expressions (if $H = H'$ then $\tau \rhd H = \tau \rhd H'$), α-conversion of names, and the following equation.

$$\nu n.\, (\tau \rhd H) = \tau \rhd (\nu n.\, H) \quad \text{if } n \notin fn(\tau)$$

The relation $\sqsubseteq$ over pure types is the least preorder including $=$ such that

$$(S, \gamma) \sqsubseteq (S', \gamma') \text{ if } \{?\} \neq S \subseteq S' \text{ and } \gamma \supseteq \gamma' \qquad (S, \gamma) \sqsubseteq (\{?\}, \gamma)$$

$$\nu N.\, \tau \rhd H \sqsubseteq \nu N.\, \tau' \rhd H' \quad \text{if } \tau \sqsubseteq \tau' \text{ and } H \sqsubseteq H' \text{ and } (fn(\tau') \setminus fn(\tau)) \between N.$$

Remark 5.1. Note that it is always possible to rewrite any type $\nu N.\,\tau \triangleright H$ in νNF. To do that, let $\hat{N} = N \cap fn(\tau)$, and let $\check{N} = N \setminus fn(\tau)$. Then, the equational theory of types gives: $\nu N.\,\tau \triangleright H = \nu\hat{N}.\,\tau \triangleright (\nu\check{N}.H)$.

In Definition 5.3 we introduce the type and effect system for $\lambda^{[\,]}$. Here we briefly comment on the most peculiar typing rules.

—T-Ev. An access $\alpha(\xi)$ has type $\mathbf{1}$, provided that the type of ξ is a pair (S, γ), and α is a capability belonging to the set γ. The effect of $\alpha(\xi)$ can be any of the accesses $\alpha(\rho)$, for ρ member of the set S.

—T-Abs. The actual effect of an abstraction is the empty history expression. Note that the latent effect (included in the type ζ) is equal to the actual effect of the function body.

—T-Wk. This rule allows for *weakening* of types and effects, according to the subtyping relation of Definition 5.2.

—T-App. The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics (function, argument, latent effect). The side conditions ensure that there is no clash of names. In particular, the last side condition makes sure that the names created by the function are never used by the argument. On the contrary, N' can intersect $fn(H)$, so allowing for "name extrusion," when a new resource is generateed by e' and then passed to e.

—T-New. The type of a $\texttt{new}\ \gamma$ is a set $\{(n, \gamma)\}$, and the actual effect is the event $new(n)$. Both occurrences of the name n are bound through a νn.

Definition 5.3 (*Type and Effects System for* $\lambda^{[\,]}$). Typing judgements $\Delta \vdash e : \zeta$ are inductively defined through the following inference rules, where $\xi \in \mathsf{Var} \cup \mathsf{Res}$.

$$\text{T-Unit}\ \ \Delta \vdash * : \mathbf{1} \triangleright \varepsilon \qquad \text{T-Var}\ \ \Delta; \xi : \tau \vdash \xi : \tau \triangleright \varepsilon$$

$$\text{T-Ev}\ \frac{\alpha \in \gamma}{\Delta; \xi : (S, \gamma) \vdash \alpha(\xi) : \mathbf{1} \triangleright \sum_{\rho \in S} \alpha(\rho)} \qquad \text{T-AddVar}\ \frac{\Delta \vdash e : \zeta}{\Delta; \xi : \tau \vdash e : \zeta}$$

$$\text{T-Abs}\ \frac{\Delta; x : \tau; z : \tau \to \zeta \vdash e : \zeta}{\Delta \vdash \lambda_z x.e : (\tau \to \zeta) \triangleright \varepsilon} \qquad \text{T-Wk}\ \frac{\Delta \vdash e : \zeta}{\Delta \vdash e : \zeta'}\ \zeta \sqsubseteq \zeta'$$

$$\text{T-App}\ \frac{\Delta \vdash e : \nu N.(\tau \to \zeta) \triangleright H \quad \Delta \vdash e' : \nu N'.(\tau \triangleright H')}{\Delta \vdash e\,e' : \nu(N \cup N').(H \cdot H' \cdot \zeta)}\ \begin{array}{l} N \not\sharp N' \\ N \not\sharp fn(\Delta) \not\sharp N' \\ N \not\sharp fn(H') \end{array}$$

$$\text{T-If}\ \frac{\Delta \vdash e : \zeta \quad \Delta \vdash e' : \zeta}{\Delta \vdash \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e' : \zeta} \qquad \text{T-Fr}\ \frac{\Delta \vdash e : \nu N.\,\tau \triangleright H}{\Delta \vdash \varphi[e] : \nu N.\,\tau \triangleright \varphi[H]}$$

$$\text{T-New}\ \frac{\kappa(\gamma) \neq \emptyset}{\Delta \vdash \texttt{new}\ \gamma : \nu n.(\{n\}, \gamma) \triangleright new_{\kappa(\gamma)}(n)}$$

Example 5.2. Next we show the typing derivations for the expressions in Example 5.1. Also here we omit capabilities, because they are irrelevant; the Appendix A has three examples with explicit capabilities.
The expression $\texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ r'$ can be typed as follows.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash r : \{r\} \rhd \varepsilon}\,\text{T-Var}}{\emptyset \vdash r : \{r, r'\} \rhd \varepsilon}\,\text{T-Wk} \qquad \dfrac{\dfrac{}{\emptyset \vdash r' : \{r'\} \rhd \varepsilon}\,\text{T-Var}}{\emptyset \vdash r' : \{r, r'\} \rhd \varepsilon}\,\text{T-Wk}}{\emptyset \vdash \texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ r' : \{r, r'\} \rhd \varepsilon}\,\text{T-If}$$

Recall that the expression $\texttt{new}\ x\ \texttt{in}\ x$ abbreviates $(\lambda x.\, x)(\texttt{new})$. Then, we have the following typing derivation for the expression $e' = \lambda y.\, \texttt{new}\ x\ \texttt{in}\ x$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \{n\} \vdash x : \{n\} \rhd \varepsilon}\,\text{T-Var}}{\emptyset \vdash \lambda x.\, x : \{n\} \to \{n\} \rhd \varepsilon}\,\text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \texttt{new} : (\nu n.\, \{n\} \rhd new(n))}\,\text{T-New}}{\emptyset \vdash (\lambda x.\, x)(\texttt{new}) : (\nu n.\, \{n\} \rhd new(n))}\,\text{T-App}}{\emptyset \vdash \lambda y.\, \texttt{new}\ x\ \texttt{in}\ x : \mathbf{1} \to (\nu n.\, \{n\} \rhd new(n)) \rhd \varepsilon}\,\text{T-Abs}$$

Consider now the expression $\lambda_z\, y.\, \texttt{new}\ x\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y$, and let in the typing derivations that follow $H = \mu h.\, \nu n.\, new(n) \cdot (h + \varepsilon)$ and $\Delta = \{y : \mathbf{1}; z : \mathbf{1} \to (\{?\} \rhd H)\}$, $\Delta' = \Delta \cup \{x : \{n\}\}$. Then, we have the following typing judgement.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta' \vdash x : \{n\} \rhd \varepsilon}\,\text{T-Var}}{\Delta' \vdash x : \{?\} \rhd H + \varepsilon}\,\text{T-Wk} \qquad \dfrac{(\text{T-Var on } z,\ y)}{\Delta' \vdash z\, y : \{?\} \rhd H + \varepsilon}\,\text{T-App}}{\Delta' \vdash \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\, y\ : \{?\} \rhd H + \varepsilon}\,\text{T-If}}{\Delta \vdash \lambda x.\, \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\, y\ : \{n\} \to (\{?\} \rhd H + \varepsilon) \rhd \varepsilon}\,\text{T-Abs}$$

A

Now we have that $H = \nu n.\, new(n) \cdot (H + \varepsilon)$, according to Definition 4.7. Also, recall that $\texttt{new}\ x\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y$ abbreviates $(\lambda x.\, \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y)(\texttt{new})$. We can then build the following typing judgement for the expression in hand.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\text{A}} \qquad \dfrac{}{\Delta \vdash \texttt{new} : \nu n.\, \{n\} \rhd new(n)}\,\text{T-New}}{\Delta \vdash \texttt{new}\ x\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y : \{?\} \rhd \nu n.\, new(n) \cdot (H + \varepsilon)}\,\text{T-App}}{\Delta \vdash \texttt{new}\ x\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y : \{?\} \rhd H}\,\text{T-Wk}}{\emptyset \vdash \lambda_z\, y.\, \texttt{new}\ x\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ z\ y : (\mathbf{1} \to \{?\} \rhd H) \rhd \varepsilon}\,\text{T-Abs}$$

Our next example has further illustrative cases of expressions and of their types and effects; the actual typing derivations are in Appendix A.

Example 5.3. Consider the following $\lambda^{[]}$expressions and their typing judgements, in the (omitted) empty typing environment. Also, in $e_{A.3}$ we must have

$\alpha \in \gamma \cap \Gamma(r)$, while in $e_{A.4} - e_{A.7}$, we assume $\alpha, \alpha' \in \gamma$.

$\vdash e_{A.1} \triangleq \texttt{if } b \texttt{ then } \lambda_z x.\alpha \texttt{ else } \lambda_z x.\alpha' : \mathbf{1} \to (\mathbf{1} \rhd \alpha + \alpha') \rhd \varepsilon$

$\vdash e_{A.2} \triangleq \lambda_g x.\texttt{if } b' \texttt{ then } * \texttt{ else } \varphi[g(e_{A.1}\, x)] : \mathbf{1} \to (\mathbf{1} \rhd \mu h.\, \varepsilon + \varphi[(\alpha + \alpha') \cdot h]) \rhd \varepsilon$

$\vdash e_{A.3} \triangleq \alpha(\texttt{new } x : \gamma \texttt{ in if } b \texttt{ then } x \texttt{ else } r) : \mathbf{1} \rhd \nu n.\, new_{\kappa(\gamma)}(n) \cdot (\alpha(n) + \alpha(r))$

$\vdash e_{A.4} \triangleq \texttt{let } f = (\lambda x.\texttt{new } y : \gamma \texttt{ in } \alpha(y); y) \texttt{ in } \alpha'(f*; f*) :$
$\mathbf{1} \rhd (\nu n.\, new_{\kappa(\gamma)}(n) \cdot \alpha(n)) \cdot (\nu n'.\, new_{\kappa(\gamma)}(n') \cdot \alpha(n') \cdot \alpha'(n'))$

$\vdash e_{A.5} \triangleq \texttt{let } g = (\texttt{new } y : \gamma \texttt{ in } \lambda x.\alpha(y); y) \texttt{ in } \alpha'(g*; g*) :$
$\mathbf{1} \rhd \nu n.\, new_{\kappa(\gamma)}(n) \cdot \alpha(n) \cdot \alpha(n) \cdot \alpha'(n)$

$\vdash e_{A.6} \triangleq (\lambda_z x.\texttt{new } y : \gamma \texttt{ in if } b \texttt{ then } \alpha(y) \texttt{ else } \alpha'(y); zx) * :$
$\mathbf{1} \rhd \mu h.\, \nu n.\, new_{\kappa(\gamma)}(n) \cdot (\alpha(n) + \alpha'(n) \cdot h)$

$\vdash e_{A.7} \triangleq \alpha((\lambda_z x.\texttt{new } y : \gamma \texttt{ in if } b \texttt{ then } y \texttt{ else } \alpha'(y); zx) *) :$
$\mathbf{1} \rhd (\mu h.\, \nu n.\, new_{\kappa(\gamma)}(n) \cdot (\varepsilon + \alpha'(n) \cdot h)) \cdot \alpha(?)$

The effects of $e_{A.4}$ and $e_{A.5}$ represent, as expected, the fact that two distinct resources are generated by $e_{A.4}$, while the evaluation of $e_{A.5}$ creates a single fresh resource. The effect of $e_{A.6}$ is a recursion, at each step of which a fresh resource is generated. The effect of $e_{A.7}$ is more peculiar: It behaves similarly to $e_{A.6}$ until the recursion is left, when the last generated resource is exported. Since its identity is lost, the event α is fired on the unknown resource "?".

Theorem 5.4 that follows guarantees that our type and effect system correctly approximates the dynamic semantics, that is, the effect of an expression e represents all the actual runtime histories of e. As usual, precision is lost with conditionals and with recursive functions. Also, you may lose the identity of names exported by recursive functions (see the type of $e_{A.7}$ in Example 5.3).

THEOREM 5.4 (CORRECTNESS OF EFFECTS). *Let* $\Delta \vdash e : \nu N.\, \tau \rhd H$, *and let* $\varepsilon, \Gamma_s, e \to^* \eta, \Gamma, e'$. *Then,* $\eta \in [\![\nu N.\, H]\!]^{-[\,]?}$.

Our type and effect system guarantees the following type safety property. If an expression is *typable*, then it will not try to access a resource without the needed capabilities. If, additionally, the effect is *valid*, then the evaluation will respect all the policies within the scope defined by their framings.

Note that, in our model, typability is not enough to guarantee policy compliance: A model-checking phase is then performed to verify validity. This design choice (type-checking capabilities and model-checking policies) is mainly motivated by technical reasons, besides obvious separation of concerns. Checking policies directly within a type system would add a significant complexity. Indeed, having a compositional type system for policies conflicts with noncompositionality of validity of history expressions (see Example 4.3).

THEOREM 5.5 (TYPE SAFETY). *Let* $\Delta \vdash e : \nu N.\, \tau \rhd H$. *Then:*

(*a*) $\varepsilon, \Gamma_s, e \not\to^* \eta, \Gamma, \mathit{fail}_\kappa$ *(e respects capabilities).*

(*b*) *if* $\nu N.\, H$ *is valid, then* $\varepsilon, \Gamma_s, e \not\to^* \eta, \Gamma, \mathit{fail}_{\varphi[e']}$ *(e respects policies).*

The proofs of the previous two theorems are in Appendix C, and they require some auxiliary definitions and technical results. Firstly, it turned out convenient to treat policy framings in $\lambda^{[\,]}$expressions as syntactic sugar, and explicitly mark their openings and closing through framing events $[_\varphi$, $]_\varphi$. This is done by mapping an expression e with policy framings into an expression $e^\sharp$ without them. Very roughly, the expression $(\varphi[e])^\sharp$ is $[_\varphi; e^\sharp;]_\varphi$. An auxiliary result is that $e^\sharp$ mimicks the behavior of e, and that the two expressions have the same type and effect.

Then, we introduce a big-step operational semantics for $\lambda^{[\,]}$, and we prove some properties it enjoys, among which that the small-step and the big-step semantics are equivalent. This is a major point, because it permits stating the subject reduction result of our type and effect system in the traditional form, where each step of a computation preserves the type. As a matter of fact, this is *not* the case with the small-step operational semantics in Definition 3.6, according to which, whenever an expression e reduces to e' while generating an event, this no longer occurs in the effect of e'. Then, the proof of the subject reduction lemma, yet long, greatly benefits from the big-step semantics.

By exploiting this result, we eventually prove that the runtime histories of an expression are included in the effects deduced by the type system (Theorem 5.4), and that an expression with a valid effect will violate no security policy at runtime (Theorem 5.5).

6. STATIC VERIFICATION

We verify the validity of history expressions by transforming them into Basic Process Algebras (BPAs), then model-checked against a framing-aware version of automata. The standard decision procedure for verifying that a BPA process p satisfies a regular property φ amounts to constructing the pushdown automaton for p and the automaton for $\neg\varphi$. Then, the property is proved by checking the emptiness of the (context-free) language accepted by the conjunction of the aforesaid automata, that is still a pushdown automaton. This problem is known to be decidable, and several algorithms and tools show this approach feasible [Esparza 1994].

Note that the arbitrary nesting of framings and the infinite alphabet of resources make validity *nonregular*, for example, the history expression

$$H \;=\; \varphi[\mu h.(\varepsilon + \nu n.\, new(n) \cdot \alpha(n) \cdot h + h \cdot h + \varphi[h])]$$

has traces with unbounded pairs of balanced $[_\varphi$ and $]_\varphi$ and unbounded number of symbols, so it is *not* a regular language. This prevents us from directly applying the standard decision technique sketched earlier.

To cope with the first source of nonregularity, due to the arbitrary nesting of framings, we define a static transformation of history expressions, called *regularization*, that removes the redundant framings.

For the second source of nonregularity, due to the possibly infinite alphabet of resources, the major challenge for verification is that history expressions may create fresh resources, while BPAs cannot. A naïve solution could lead to the generation of an unbounded set of automata $A_{\varphi(r,R)}$ that must be checked to verify validity. For example, the histories denoted by the history expression H

given before must satisfy all the policies $\varphi(r_0), \varphi(r_1), \ldots$ for each freshly created resource. Thus, we would have to intersect an infinite number of finite state automata to verify H valid, which is unfeasible.

To cope with that, we shall define a mapping from history expressions to BPAs that preserves validity. Our mapping groups fresh resources in two sets. The intuition is that a policy $\varphi(x)$ can only distinguish between x and all the other resources, represented by $\bar{x}$. There is no way for $\varphi(x)$ to further discriminate among the dynamic resources: This is because our policy framings $\varphi[e]$ are not parametrized with dynamically generated resources; instead they cause the policy φ to be uniformly applied to *every* resource. Hence, it is sound to only consider two representatives of dynamic resources: the "witness" resource # that represents x, and the "don't care" resource $\_$ for $\bar{x}$.

The rest of this section is organized as follows. The validity-preserving regularization of history expressions is introduced in Section 6.1. In Section 6.2 we define a transformation of policy automata that makes them able to recognize valid histories. Section 6.3 contains the semantic-preserving mapping from history expressions to Basic Process Algebras. The main result of this section is Theorem 6.13, that states the correctness of our procedure. Together with the type safety result, this implies that, if a $\lambda^{[\,]}$expression is well-typed and its effect is checked valid, then it never violates security.

6.1 Regularization of History Expressions

History expressions can generate histories with *redundant framings*, that is, 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. Removing inner redundant framings from a history preserves its validity. However, the expressive power of a pushdown automaton is necessary because open and closed framings are to be matched in pairs. For example, consider the following history.

$$\eta \;=\; \alpha\, \overbrace{[_\varphi \cdots [_\varphi}^{n}\, \overbrace{]_\varphi \cdots]_\varphi}^{m} [_\varphi$$

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

Definition 6.1. We say a history η has *redundant framings* when $\eta = \eta_0[_\varphi \eta_1[_\varphi \eta_2$ for some $\varphi, \eta_0, \eta_1, \eta_2$ with $]_\varphi \notin \eta_1$. We say a closed history expression H has redundant framings when some $\eta \in [\![H]\!]$ has redundant framings.

To define regularization, it is convenient to first specify in Definition 6.2 the guards of a history expression.

Definition 6.2 (*Guards*). Let $\tilde{H}$ be a history expression with a single hole $\bullet$, such that $H = \tilde{H}\{h/\bullet\}$. We say that h (more precisely, the occurrence of h replacing $\bullet$) is *guarded by* Φ *in* H when $\Phi \subseteq guard(\tilde{H})$.

The set $guard(\tilde{H})$ is inductively defined by the following equations.

$$\begin{aligned} guard(H_0 \cdot H_1) &= guard(H_i) \quad \text{if } \bullet \in H_i \\ guard(H_0 + H_1) &= guard(H_i) \quad \text{if } \bullet \in H_i \\ guard(\varphi[H]) &= \{\varphi\} \cup guard(H) \\ guard(\nu n.\, H) &= guard(H) \\ guard(\mu h.\, H) &= guard(H) \\ guard(\bullet) &= \emptyset \end{aligned}$$

We say that h is *guarded* in H when h is guarded by some nonempty Φ in H; otherwise h is *unguarded*.

Example 6.1. Let $H = \varphi[\alpha \cdot h_0 \cdot \varphi'[\alpha' + h_1]] \cdot h_2$, and let $\tilde{H} = \varphi[\alpha \cdot h_0 \cdot \varphi'[\alpha' + \bullet]] \cdot h_2$. Then, $H = \tilde{H}\{h_1/\bullet\}$, and so h_1 is guarded by $guard(\tilde{H}) = \{\varphi, \varphi'\}$. Similarly, h_0 is guarded by $\{\varphi\}$, and h_2 is unguarded (i.e., guarded by $\emptyset$).

Definition 6.3 (*Regularization of History Expressions*). Let H be a history expression where all the variables have distinct names. The regularization $H\downarrow$ of H is defined as $H\downarrow_{\emptyset,\emptyset}$, where $H\downarrow_{\Phi,\Upsilon}$, inductively defined by the following rules, is the regularization of H against a set of policies Φ and a substitution Υ of variables into history expressions.

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

$$\alpha(\rho)\downarrow_{\Phi,\Upsilon} = \alpha(\rho) \qquad (H \cdot H')\downarrow_{\Phi,\Upsilon} = H\downarrow_{\Phi,\Upsilon} \cdot H'\downarrow_{\Phi,\Upsilon}$$

$$(\nu n.\, H)\downarrow_{\Phi,\Upsilon} = \nu n.\, H\downarrow_{\Phi,\Upsilon} \qquad (H + H')\downarrow_{\Phi,\Upsilon} = H\downarrow_{\Phi,\Upsilon} + H'\downarrow_{\Phi,\Upsilon}$$

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

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

$$\sigma(h_i) = (\mu h.H)\Upsilon\downarrow_{\Phi\cup guard(H'\{\bullet/h_i\}),\Upsilon}$$

$$\sigma'(h_i) = \begin{cases} h & \text{if } guard(H'\{\bullet/h_i\}) \subseteq \Phi \\ h_i & \text{otherwise} \end{cases}$$

Intuitively, $H\downarrow_{\Phi,\Upsilon}$ results from H by eliminating all the redundant framings, and all the framings in Φ. The environment Υ is needed to deal with free variables in the case of nested μ-expressions, as shown by Example 6.3 later. We sometimes omit to write the component Υ when unneeded.

The rules for framings and recursion will 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 now a history expression of the form $\mu h.H$. Its regularization against Φ and Υ 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 Υ is used to bind the free variables to closed history expressions. Technically, the ith 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 $\sigma'(h_i) = h_i$.

Two examples of regularization follow. In Example 6.2, we regularize a history expression with simple recursion, while in Example 6.3 we deal with the more complex case of mutual recursion.

Example 6.2. Let $H_0 = \mu h.\, H$, be a history expression 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 $guard(H'\{\bullet/h_2\}) = \{\varphi\} \not\subseteq \emptyset$ and $guard(H'\{\bullet/h_0\}) = guard(H'\{\bullet/h_1\}) = \emptyset$ we have that

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

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

Example 6.3. Let $H_0 = \mu h.\, H_1$, where $H_1 = \mu h'.\, H_2$, and $H_2 = \alpha + h \cdot \varphi[h']$. Since $guard(H_1'\{\bullet/h\}) = \emptyset$, 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 $guard(H_2'\{\bullet/h_0\}) = \{\varphi\} \not\subseteq \emptyset$, it follows that

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

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

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

Since $\{\varphi\}$ contains both $guard(H_1\{\bullet/h\}) = \emptyset$, and $guard(H_2\{\bullet/h_2\}) = \{\varphi\}$, 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 preceding computations, we have that

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

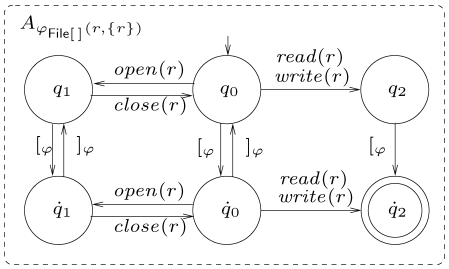

Fig. 6. The framed automaton $A_{\varphi_{\mathsf{File}[\,]}(r,\{r\})}$.

As a matter of fact, regularization is a total function, and its definition can be easily turned into a terminating rewriting system. We now establish the following basic properties of regularization, stating its correctness.

THEOREM 6.4. *For each history expression H:*

—*$H\downarrow$ is defined.*
—*$H\downarrow$ has no redundant framings.*
—*$H\downarrow$ is valid if and only if H is valid.*

6.2 Framed Automata

We specify in Definition 6.5 a transformation of automata that makes them able to recognize valid traces. This is done by instrumenting an automaton $A_{\varphi(r,R)}$ with framing events, so obtaining a *framed* automaton $A_{\varphi_{[\,]}(r,R)}$ that will recognize those traces that are valid with respect to the policy φ.

Definition 6.5 (*Instrumenting Automata with Framing Events*). Let $A_{\varphi(r,R)} = \langle \Sigma, Q, q_0, F, \delta\rangle$ be an automaton. Then, we define $A_{\varphi_{[\,]}(r,R)} = \langle \Sigma', Q', q_0, F', \delta'\rangle$ as follows.

$$\Sigma' = \Sigma \cup \{[_\varphi,]_\varphi, [_{\varphi'},]_{\varphi'}, \ldots\} \qquad Q' = Q \cup \{\dot{q} \mid q \in Q\} \qquad F' = \{\dot{q} \mid q \in F\}$$
$$\delta' = \delta \;\cup\; \{q \xrightarrow{[_\varphi} \dot{q} \mid q \in Q\} \;\cup\; \{\dot{q} \xrightarrow{]_\varphi} q \mid q \in Q \setminus F\} \;\cup\; \{\dot{q} \xrightarrow{\vartheta} \dot{q} \mid \dot{q} \in F'\}$$
$$\cup\, \{\dot{q} \xrightarrow{\vartheta} \dot{q}' \mid q \xrightarrow{\vartheta} q' \in \delta \text{ and } q \in Q \setminus F\} \;\cup\; \{q \xrightarrow{[_\psi} q, q \xrightarrow{]_\psi} q \mid \psi \neq \varphi\}$$

Intuitively, the automaton $A_{\varphi_{[\,]}(r,R)}$ is partitioned into two layers. Both are copies of $A_{\varphi(r,R)}$, but in the first layer of $A_{\varphi_{[\,]}(r,R)}$ all the states are made nonfinal. This represents being compliant with φ. The second layer is reachable from the first one when opening a framing for φ, while closing gets back, unless we are in a final (i.e., offending) state. The transitions in the second layer are a copy of those in $A_{\varphi(r,R)}$, the only difference being that the final states are sinks. The final states in the second layer are exactly those final in $A_{\varphi(r,R)}$.

Example 6.4. Consider the file usage policy $\varphi_{\mathsf{File}}(x)$ of Section 2, requiring that only open files can be read or written. The instrumentation $A_{\varphi_{\mathsf{File}[\,]}(r,\{r\})}$ of $A_{\varphi_{\mathsf{File}}(r,\{r\})}$ is in Figure 6; the self-loops are omitted. The top (respectively, bottom) layer models being outside (respectively, inside) the scope of φ_{File}.

We now relate framed automata with validity. Let η be a history with no redundant framings. Then η is valid if and only if it complies with the framed automata $A_{\varphi_{[]}(r,\mathsf{R}(\eta))}$ for all the policies φ spanning over η. Recall that if η is obtained from a regularized history expression, then it is guaranteed free from redundant framings.

LEMMA 6.6. *Let η be a history with no redundant framings, and let r' be an arbitrary resource in $\mathsf{Res} \setminus \mathsf{R}(\eta)$. Then, η is valid if and only if $\eta \lhd A_{\varphi_{[]}(r,\mathsf{R}(\eta))}$, for all φ occurring in η, and for all $r \in \mathsf{R}(\eta) \cup \{r'\}$.*

6.3 Model-Checking Validity

Basic Process Algebras [Bergstra and Klop 1985] provide a natural characterization of history expressions. BPA processes contain the terminated process 0, events β, that we use for access and framing events, the operators $\cdot$ and $+$ that denote sequential composition and (nondeterministic) choice, and variables $X, Y, \ldots$. To allow for recursion, a BPA is then defined as a process p and a set of definitions Δ for the variables X that occur therein.

Definition 6.7 (*Syntax of Basic Process Algebras*). A *BPA process* is given by the following abstract syntax, where $\beta \in \mathsf{Ev} \cup \mathsf{Frm}$.

$$p, p' \quad ::= \quad 0 \mid \beta \mid p \cdot p' \mid p + p' \mid X$$

A *BPA definition* has the form $X \triangleq p$. A set Δ of BPA definitions is *coherent* when $X \triangleq p \in \Delta$ and $X \triangleq p' \in \Delta$ imply $p = p'$.

A *BPA* is a pair $\langle p, \Delta \rangle$ such that: (i) Δ is finite and coherent, and (ii) for all X occurring in p or Δ, there exists some q such that $X \triangleq q \in \Delta$.

We write $\Delta + \Delta'$ for $\Delta \cup \Delta'$, when coherent (otherwise, $\Delta + \Delta'$ is undefined).

Let $P = \langle p, \Delta \rangle$ and $P' = \langle p', \Delta' \rangle$. We write $P \cdot P'$ for $\langle p \cdot p', \Delta + \Delta' \rangle$, and $P + P'$ for $\langle p + p', \Delta + \Delta' \rangle$.

The semantics of BPAs is given by the labelled transition system in Definition 6.8. Note that there is no need to consider infinite computations, because validity is a safety property [Schneider 2000], that is, nothing bad can happen in any (finite) trace of execution steps.

Definition 6.8 (*Operational Semantics of Basic Process Algebras*). The semantics $[\![P]\!]$ of a BPA $P = \langle p_0, \Delta \rangle$ is the set of the histories labeling finite computations:

$$\{\eta = a_1 \cdots a_i \mid p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} p_i\}$$

where $a \in \mathsf{Ev} \cup \{\varepsilon\}$, and the relation $\xrightarrow{a}$ is inductively defined by the rules

$$0 \cdot p \xrightarrow{\varepsilon} p \qquad \beta \xrightarrow{\beta} 0 \qquad p + q \xrightarrow{\varepsilon} p \qquad p + q \xrightarrow{\varepsilon} q$$

$$\frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad X \xrightarrow{\varepsilon} p \quad \text{if } X \triangleq p \in \Delta.$$

Our transformation from history expressions to BPAs is formalized in Definition 6.9 through the function $B(H)$. The auxiliary function $\Phi(H)$ returns a *finite* set of automata, to be used in model-checking validity. Without loss of generality, we assume that the recursion variables in H are all distinct.

Events, variables, concatenation, and choice are mapped by $B(H)$ into the corresponding BPA counterparts. Note that an expression $\varphi[H]$ is handled as $[_\varphi \cdot H \cdot]_\varphi$, so $B(\varphi[H])_\Theta = \langle [_\varphi \cdot p \cdot]_\varphi, \Delta\rangle$, where $B(H)_\Theta = \langle p, \Delta\rangle$. 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. The tricky case is that of new name generation. For each binder $\nu n.H$, we generate a choice of two BPA processes: In the first, the name n is replaced by the "don't care" resource $\_$, while in the second, n is replaced by the "witness" resource #. For simplicity, we here consider a single witness. To deal with the general case where resources are partitioned in kinds Act_i, it suffices using a $\#_i$-witness for each kind (e.g., $\#_{\mathsf{File}}$, $\#_{\mathsf{Socket}}$, etc.).

The set $\Phi(H)$ contains, for each policy φ and static resource r_0, the framed automaton $A_{\varphi[\,](r_0,\mathsf{R}(H))}$. Also, $\Phi(H)$ includes the instantiations $A_{\varphi[\,](\#,\mathsf{R}(H))}$, to be ready on controlling φ on dynamic resources, represented by the witness #.

The transformation presented here handles monadic policies $\varphi(x)$, only. However, it can be extended to the case of polyadic policies $\varphi(x_1, \ldots, x_k)$ by using k witnesses $\#_1, \ldots, \#_k$ (see Bartoletti et al. [2008] and Bartoletti [2009] for additional details).

Definition 6.9 (*Mapping History Expressions to BPAs*). Given a closed history expression H, its associated BPA is

$$B(H) \;=\; B(H)_\emptyset,$$

where the transformation $B(H)_\Theta$ takes as input a history expression and a function Θ from variables h to BPA variables X. The symbols $\_$ and # are distinguished resources in Res_d.

$$\begin{aligned}
B(\varepsilon)_\Theta &= \langle 0, \emptyset\rangle \\
B(h)_\Theta &= \langle \Theta(h), \emptyset\rangle \\
B(\alpha(\rho))_\Theta &= \langle \alpha(\rho), \emptyset\rangle \\
B(H_0 \cdot H_1)_\Theta &= B(H_0)_\Theta \cdot B(H_1)_\Theta \\
B(H_0 + H_1)_\Theta &= B(H_0)_\Theta + B(H_1)_\Theta \\
B(\nu n.H)_\Theta &= B(H\{_/n\})_\Theta + B(H\{\#/n\})_\Theta \\
B(\mu h.H)_\Theta &= \langle X, \Delta \cup \{X \triangleq p\}\rangle, \text{where } B(H)_{\Theta\{X/h\}} = \langle p, \Delta\rangle, X \text{ fresh}
\end{aligned}$$

Furthermore, the set $\Phi(H)$ is defined as

$$\Phi(H) \;=\; \{A_{\varphi[\,](r_0,\mathsf{R}(H))} \mid r_0, \varphi \in H\} \cup \{A_{\varphi[\,](\#,\mathsf{R}(H))} \mid \varphi \in H\}$$

where: $\mathsf{R}(H) \;=\; \{r_0 \in \mathsf{Res}_s \mid r_0 \in H\} \cup \{\#, \_\}$.

We now state the correspondence between history expressions and BPAs. The histories generated by a history expression H are all and only the strings

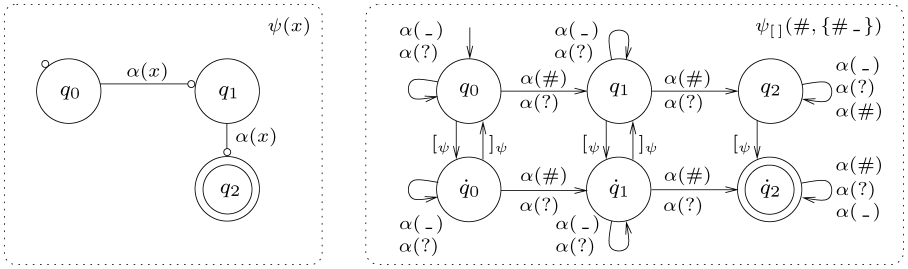

Fig. 7. The usage automaton $\psi(x)$ and the instantiation $A_{\psi_{[]}(\#,\{\_,\#\})}$. The self-loops for the *new* events are omitted.

that label the computations of the extracted BPA, where dynamic resources are suitably renamed to permit model-checking. An auxiliary definition follows.

Definition 6.10. Let σ be a substitution from $\mathsf{Res} \cup \{?\}$ to $\mathsf{Res} \cup \{?\}$. We say that σ is *name-collapsing* when, for some set of dynamic resources $R \subset \mathsf{Res}_d$:

$$\begin{array}{llllll} \sigma(r) = \# & \text{if } r \in R & & \sigma(\#) = \# & \sigma(?) = ? & \\ \sigma(r) = \_ & \text{if } r \in \mathsf{Res}_d \setminus R & \sigma(\_) = \_ & & \sigma(r) = r & \text{otherwise.} \end{array}$$

THEOREM 6.11. *For each history expression H, history η, and name-collapsing σ:*

$$\eta \in [\![H]\!] \implies \exists P.\, B(H) \xrightarrow{\eta\sigma} P.$$

A couple of examples follow.

Example 6.5. Let $H = \mu h.(\nu n.\, \varepsilon + \alpha(n) \cdot h)$. Then, the BPA extracted from H is $\langle X, \Delta \rangle$, where $\Delta = \{X \triangleq 0 + \alpha(\_) \cdot X + 0 + \alpha(\#) \cdot X\}$. Consider the history $\eta = \alpha(r_0)\alpha(r_1)\alpha(r_2) \in [\![H]\!]$, and let $\sigma = \{\_/r_0, \#/r_1, \_/r_2\}$ be a name-collapsing substitution. Then, $\eta\sigma = \alpha(\_)\alpha(\#)\alpha(\_)$ is a trace in $[\![\langle X, \Delta \rangle]\!]$.

Example 6.6. Consider the history expression

$$H = \psi[(\nu n.\, new(n) \cdot \alpha(n)) \cdot (\nu n'.\, new(n') \cdot \alpha(n')) \cdot \alpha(?)],$$

where the policy ψ asks that the action α cannot be performed twice on the same resource (lefthand side of Figure 7). Then

$$\begin{aligned} B(H) = [_\psi \cdot (new(\_) \cdot \alpha(\_) \;+\; & new(\#) \cdot \alpha(\#)) \cdot (new(\_) \cdot \alpha(\_) \\ +\; & new(\#) \cdot \alpha(\#)) \cdot \alpha(?) \cdot]_\psi. \end{aligned}$$

The BPA $B(H)$ violates $A_{\psi_{[]}(\#,\{\_,\#\})} \in \Phi(H)$, displayed in righthand side of Figure 7. The violation is consistent with the fact that the wildcard ? represents any resource, for example, $\alpha(r')\alpha(r)\alpha(r) \in [\![H]\!]$ violates ψ. In general, we can prove that whenever $B(H)$ is valid, then also H is valid. However, this verification technique is not complete. Consider for instance a policy ψ', requiring that α is not executed *three* times on the same resource. Let:

$$H' = \psi'[(\nu n.\, new(n) \cdot \alpha(n)) \cdot (\nu n'.\, new(n') \cdot \alpha(n')) \cdot \alpha(?)].$$

Note that $B(H')$ violates $A_{\psi'_{[]}(\#,\{\_,\#\})}$, while all the histories denoted by H' respect ψ'. Theorem 6.13 shows a sound and complete verification technique. □

As shown in Example 6.6, the validity of H does not imply the validity of $B(H)$, so leading to a sound but incomplete decision procedure. The problem is that $B(H)$ uses the same "witness" resource # for all the resource creations in H. This obviously makes $B(H)$ identify (as #) resources that are distinct in H. This leads to violations of policies, such as those that prevent some action from being performed twice (or more) on the same resource.

To recover a (sound and) complete decision procedure for validity, it suffices to check any history of $B(H)$ only *until* the second "witness" resource is generated (i.e., before the second occurrence of $new(\#)$). Recall that $B(\nu n.\bar{H})$ is a sum of two BPAs: the first one instantiating n with #, and the second one using $\_$ instead. By truncating histories of $B(H)$ before the second $new(\#)$ occurs, we ensure that # corresponds to a single resource $r \in \mathsf{Res}_d$ occurring in a history η of H. Note that, while this prevents from abstracting other resources $r' \neq r$ in η with the witness #, these can still be abstracted by $\_$. This abstraction preserves completeness because, as discussed at the beginning of this section, a policy $\varphi(x)$ can only distinguish between the resource to which x is instantiated and the other resources. Truncating the histories of $B(H)$ can be accomplished by composing $B(H)$ with the "unique witness" automaton through a "weak until" operation (Definition 6.12). The weak until W is standard; the unique witness $A_{\#}$ is a finite state automaton that reaches an offending state on those traces containing more than one $new(\#)$ event.

Example 6.7. Consider again the history expression H' in Example 6.6. The maximal histories generated by $B(H')$ are shown next.

$$
\begin{array}{l}
[_{\psi'}\ new(\_)\alpha(\_)\ new(\#)\alpha(\#)\ \alpha(?)\ ]_{\psi'} \\
[_{\psi'}\ new(\#)\alpha(\#)\ new(\_)\alpha(\_)\ \alpha(?)\ ]_{\psi'} \\
[_{\psi'}\ new(\_)\alpha(\_)\ new(\_)\alpha(\_)\ \alpha(?)\ ]_{\psi'} \\
[_{\psi'}\ new(\#)\alpha(\#)\ new(\#)\alpha(\#)\ \alpha(?)\ ]_{\psi'}
\end{array}
$$

Note that the first three preceding histories are valid with respect to the policies in $\Phi(H') = \{\psi'_{[\,]}(\#)\}$. Instead, the last history violates $\psi'_{[\,]}(\#)$. As said before, we discard this history before model-checking, because considering it would be a source of incompleteness. Indeed, in $[_{\psi'}\ new(\#)\alpha(\#)\ new(\#)\alpha(\#)\ \alpha(?)\ ]_{\psi'}$ the two fresh resources are identical, so constrasting with the semantics of history expressions. Note also that the automaton $A_{\psi'_{[\,]}(\_,\{\_,\#\})}$ is not in $\Phi(H')$. This is sound because if a violation has to occur, it always occurs on $A_{\psi'_{[\,]}(\#,\{\_,\#\})}$.

Definition 6.12 (*Weak Until and Unique Witness Automata*). Let $A_0 = \langle \Sigma, Q_0, q_0, F_0, \rho_0 \rangle$ and $A_1 = \langle \Sigma, Q_1, q_1, F_1, \rho_1 \rangle$ be automata. Assume that A_0 and A_1 are *complete*, that is, for each state q and $\beta \in \Sigma$, there exists a transition from q labelled β. The *weak until* automaton $A_0 \,\mathsf{W}\, A_1 = \langle \Sigma, Q, q, F, \rho \rangle$ is

defined as follows.

$$
\begin{aligned}
Q &= Q_0 \times Q_1 \\
F &= F_0 \times (Q_1 \setminus F_1) \\
q &= \langle q_0, q_1 \rangle \\
\rho &= \{\langle q_i, q_j \rangle \xrightarrow{\vartheta} \langle q_i', q_j' \rangle \mid q_i \xrightarrow{\vartheta} q_i' \in \rho_0,\ q_j \xrightarrow{\vartheta} q_j' \in \rho_1,\ q_j \in Q_1 \setminus F_1\} \\
&\quad \cup \{\langle q_i, q_j \rangle \xrightarrow{\vartheta} \langle q_i, q_j \rangle \mid q_j \in F_1, \vartheta \in \Sigma\}
\end{aligned}
$$

The *unique witness* automaton $A_\# = \langle \Sigma, Q, q_0, F, \rho \rangle$ is defined as follows:

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
F &= \{q_2\} \\
\rho &= \{q_0 \xrightarrow{new(\#)} q_1,\ q_1 \xrightarrow{new(\#)} q_2\} \\
&\quad \cup \{q_0 \xrightarrow{\vartheta} q_0,\ q_1 \xrightarrow{\vartheta} q_1 \mid \vartheta \in \Sigma \setminus \{new(\#)\}\} \cup \{q_2 \xrightarrow{\vartheta} q_2 \mid \vartheta \in \Sigma\}
\end{aligned}
$$

The following theorem enables us to verify the validity of a history expression H by: (i) regularizing H, (ii) extracting a BPA $P = B(H)$ from H, and (iii) model-checking P against the finite set of framed policy automata $\Phi(H)$. Together with type safety (Theorem 5.5), a $\lambda^{[\,]}$expression *will never be aborted* at runtime by the execution monitor if its effect is checked valid.

THEOREM 6.13. *An initial history expression H is valid if and only if, for all $A_{\varphi_{[\,]}(r,R)} \in \Phi(H)$:*

$$[\![B(H\downarrow)]\!] \lhd A_{\varphi_{[\,]}(r,R)} \mathsf{W}\, A_\#.$$

The proof of this theorem and of all the intermediate results are in Appendix D.

Proving the correctness of regularization is done by using the denotational semantics of history expressions, proved equivalent to the operational one in Appendix B.

We then prove that policy compliance can be established by checking membership in a regular language (Lemma 3.4). The validity of a history amounts then to being (not) accepted by the relevant framed automaton (Lemma 6.6).

Comparing the operational semantics of a history expression H to the corresponding BPA $B(H)$ relies on proving their bisimilarity. This, however, requires to consider execution traces where the dynamic resources are abstracted into the witness # or the "don't care" _.

The preceding results are then used to establish the correctness of our model-checking procedure, as stated by Theorem 6.13.

As it is, our model-checking algorithm has exponential complexity in the number of the ν-binders contained in the history expression. The source of the exponential comes from the rule $B(\nu n.H)_\Theta = B(H\{\_/n\})_\Theta + B(H\{\#/n\})_\Theta$ in Definition 6.9. Our recent article [Bartoletti et al. 2008] proposes a polynomial algorithm that is based on a more careful grouping of witnesses. It has been implemented in the prototypical model-checker LocUsT [Bartoletti and Zunino 2008].

We conclude this section with two examples.

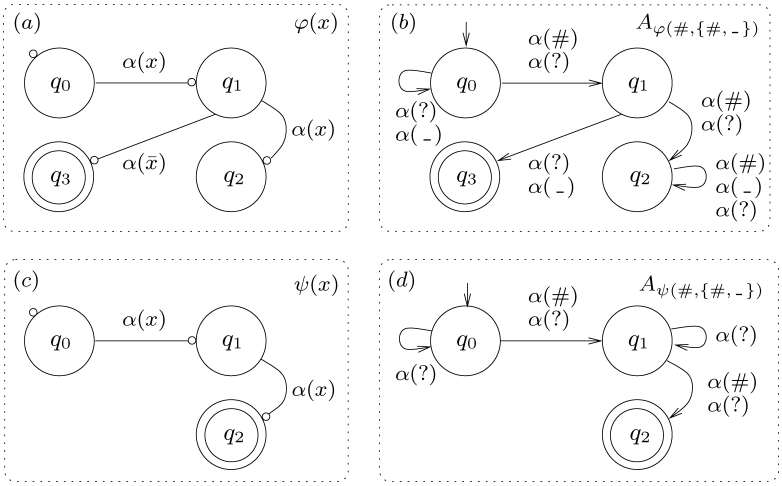

Fig. 8. The usage policies $\varphi(x)$ and $\psi(x)$ and the automata $A_{\varphi(\#,\{\#,\_\})}$ and $A_{\psi(\#,\{\#,\_\})}$. The self-loops for the *new* events are omitted.

Example 6.8. Let $H = \mu h.(\varepsilon + \nu n.\, new(n) \cdot \alpha(n) \cdot h)$. Consider then $H_\varphi = \varphi[H]$, where $\varphi(x)$ asks that, for each resource x, the first occurrence of the event $\alpha(x)$ is necessarily followed by another $\alpha(x)$ (Figure 8(a)). Then, H_φ is *not* valid, because, for example, $\eta = [_\varphi new(r)\alpha(r)new(r')\alpha(r')$ is a history of H_φ, and $\eta \not\models A_{\varphi_{[]}(r,\{r,r'\})}$ (the nonframed automaton is in Figure 8(b)). This agrees with our model-checking technique. We have that $\Phi(H_\varphi) = \{A_{\varphi_{[]}(\#,\{\#,\_\})}\}$, and:

$$B(H_\varphi) = \langle [_\varphi \cdot X \cdot]_\varphi, X \triangleq \varepsilon + (new(\#) \cdot \alpha(\#) \cdot X) + (new(\_) \cdot \alpha(\_) \cdot X) \rangle$$

and the history $[_\varphi new(\#)\alpha(\#)new(\_)\alpha(\_) \in [\![B(H_\varphi)]\!]$ drives the framed automaton $A_{\varphi_{[]}(\#,\{\#,\_\})}$ to an offending state. Consider now $H_\psi = \psi[H]$, where the policy $\psi(x)$ says that the action α cannot be performed twice on the same resource x (Figure 8(c)). We have that $\Phi(H_\psi) = \{A_{\psi_{[]}(\#,\{\#,\_\})}\}$, and:

$$B(H_\psi) = \langle [_\psi \cdot X \cdot]_\psi, X \triangleq \varepsilon + (new(\#) \cdot \alpha(\#) \cdot X) + (new(\_) \cdot \alpha(\_) \cdot X) \rangle.$$

Although H_ψ obeys ψ, the BPA does not, since $[_\psi new(\#)\alpha(\#)new(\#)\alpha(\#)]_\psi$ violates $A_{\psi_{[]}(\#,\{\#,\_\})}$ (the nonframed instantiation $A_{\psi(\#,\{\#,\_\})}$ is in Figure 8(d)). Completeness is recovered through the weak until and unique witness automata, that filter the histories, like the one preceding, where $new(\#)$ is fired twice.

Example 6.9. Recall from Section 2 the policy φ_{File} (only open files can be read or written), and the policy φ_{DoS} (never create more than k files). Let

$$H = \varphi_{\mathsf{File}}[\varphi_{\mathsf{DoS}}[\mu h.\, \varepsilon + \nu n.\, new_{\mathsf{File}}(n) \cdot open(n) \cdot read(n) \cdot close(n) \cdot h]].$$

Then, $\Phi(H) = \{A_{\varphi_{\mathsf{File}}(\#)}, A_{\varphi_{\mathsf{DoS}}(\#)}\}$, and $B(H) = \langle [_\varphi \cdot [_{\varphi_{\mathsf{DoS}}} \cdot X \cdot]_{\varphi_{\mathsf{DoS}}} \cdot]_\varphi, \Delta \rangle$, where Δ comprises the following definition.

$$\begin{aligned} X \triangleq\ & \varepsilon + (new_{\mathsf{File}}(\_) \cdot open(\_) \cdot read(\_) \cdot close(\_) \cdot X) \\ & + (new_{\mathsf{File}}(\#) \cdot open(\#) \cdot read(\#) \cdot close(\#) \cdot X) \end{aligned}$$

Note that each computation of $B(H)$ obeys $A_{\varphi_{\mathsf{File}[\,]}(\#,\{\#,\,\_\})}$, while there exist computations that violate $A_{\varphi_{\mathsf{DoS}[\,]}(\#,\{\#,\,\_\})}$.

7. RELATED WORK

A wide variety of languages for expressing policies have been proposed over the years. A typical design compromise is that between expressivity and performance of the enforcement mechanism. The latter can be obtained either through an efficient implementation of the runtime monitor, or by exploiting suitable verification techniques to optimize the monitor (e.g., by removing some or all the dynamic checks).

A characterization of the class of policies enforceable through runtime monitoring systems is given in Schneider [2000]. There, the policy language is that of *security automata*, a class of Büchi automata that recognize safety properties. To handle the case of parameters ranging over infinite domains (e.g., the formal parameters of a method), security automata resort to a countable set of states and input symbols. While this makes security automata suitable to provide a common semantic framework for policy languages (e.g., the semantics of our usage automata can be given in terms of a mapping into security automata) it also makes security automata hardly usable as an effective formalism for expressing policies. In Hamlen et al. [2006] a characterization is given of the policies that can be enforced through program rewriting techniques. In Bauer et al. [2002] a kind of automata that can delete and insert events in the execution trace is considered. Polymer [Bauer et al. 2005] is a language for specifying, composing, and enforcing (global) security policies. In the lines of *edit automata* [Bauer et al. 2002], a policy can intervene in the program trace, to insert, or suppress some events. Policy composition can then be unsound, because the events inserted by a policy may interfere with those monitored by another policy. To cope with that, the programmer must explicitly fix the order in which policies are applied. Being Turing-equivalent, Polymer policies are more expressive than ours, but this gain in expressivity has some disadvantages. First, a policy may potentially be unable to decide whether an event is accepted or not (i.e., checking it may not terminate). Second, no static guarantee is given about the compliance of a program with the imposed policy. Runtime monitoring is then necessary to enforce the policy, while our model-checking technique may avoid this overhead.

Igarashi and Kobayashi [2002] extended the λ-calculus with primitives for creating and accessing resources, and for defining their permitted usage patterns. The primitive $new^{\Phi}()$ creates a new resource with a *trace set* Φ, that is, a set of permitted access sequences, while the primitive $acc^{\ell}(x)$ for accessing the resource x (the label ℓ represents the program point where the access is performed). Also abstractions $e = \lambda^{\Phi} x.e'$ can be protected by a trace set Φ, to be satisfied when evaluating a labelled application $e@^{\ell}e'$. To access a resource x with trace set Φ from an expression labelled ℓ, there must exist a valid trace in Φ, namely a trace beginning with ℓ. An execution is resource-safe when the possible patterns are within the permitted ones. A type system guarantees well-typed expressions to be resource-safe. Types abstract the usages permitted at runtime, while typing rules check that resource accesses respect the deduced

permitted usages. Since collecting the usages and checking resource-safety is done within a single phase, the language of usages needs to be quite expressive, which causes type inference to be undecidable in general. Indeed, the authors are focussed on achieving generality (at the cost of a rather complex type system) rather than offering a concrete verification algorithm like ours. Actually, here we provided $\lambda^{[]}$with a sound and complete verification technique. Separating the analysis of effects from their verification offers a further advantage. It makes it possible to independently improve the accuracy of the analysis (e.g., through more sophisticated type systems or through abstract interpretation techniques) and the efficiency of the verification procedure. Clearly, also Igarashi and Kobayashi [2002] is amenable to static verification, provided that one either restricts the language of permitted usages to a decidable subset, or one uses a sound but incomplete algorithm. Along the same lines, Kobayashi [2003] designs even a more general type system, while Iwama et al. [2006] extends Igarashi and Kobayashi [2002] to languages with exceptions.

We took much inspiration from Skalka and Smith's [2004] and Skalka et al. [2008], who proposed λ_{hist}, a λ-calculus with local checks that enforce linear μ-calculus properties [Bradfield 1996; Kozen 1983] on the past history. A type and effect system approximates the possible runtime histories, whose validity can be statically verified by model-checking μ-calculus formulae over Basic Process Algebras [Bergstra and Klop 1985; Esparza 1994]. Compared to Skalka et al. [2008], our calculus features parametric policies, dynamic resource creation, and local policies instead of local checks. As a matter of fact, λ_{hist} local checks can be encoded in $\lambda^{[]}$as local policies that include no events: Indeed, 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 [Wallach et al. 2000]) can do the job, because λ_{hist} is Turing complete. As far as the type systems are concerned, that of Skalka et al. [2008] also allows for let-polymorphism, subtyping of functional types, and type inference, while ours handles resource creation through the ν-binders in types. In Section 8 we further discuss these issues. On the same lines of Skalka and Smith [2004], Skalka defines in Skalka [2005] a type and effect system for an extension of Featherweight Java, featuring histories and security checks.

Regions have been used in type and effect systems [Talpin and Jouvelot 1992; Nielson and Nielson 1994] to approximate new names in impure call-by-value λ-calculi. Similarly to ours, the static semantics of Nielson and Nielson [1994] aims at overapproximating the set of runtime traces, while that of Talpin and Jouvelot [1992] only considers flat sets of events. A main difference from our approach is that, while our ν-types deal with the *freshness* of names, both Talpin and Jouvelot [1992] and Nielson and Nielson [1994] use universal polymorphism for typing resource creations. Since a region n stands for a *set* of resources, in an effect $\alpha(n) \cdot \beta(n)$ their static approximation does not ensure that α and β act on the same resource. This property can instead be guaranteed in our system through the effect $\nu n.(\alpha(n) \cdot \beta(n))$. This improvement in the precision

of approximations is crucial, since it allows us to model-check in Bartoletti et al. [2008] regular properties of traces (e.g., permit *read*(*file*) only after an *open*(*file*)) that would otherwise fail with the approximations of Talpin and Jouvelot [1992] and Nielson and Nielson [1994].

Many authors [Colcombet and Fradet 2000; Erlingsson and Schneider 1999; Marriott et al. 2003; Thiemann 2003] mixed static and dynamic techniques to transform programs and make them obey a given global policy. Colcombet and Fradet abstracted a program into a control flow graph, instrumented with annotations that keep track of the state of the global policy [Colcombet and Fradet 2000]. This is expressed by a finite state automaton, which is then minimized to remove the unnecessary tracking. Finally, the optimized control flow graph is converted back to a program, that is guaranteed to abort just before violating the global property. Marriott et al. overapproximated the runtime behavior of a program through a context-free grammar [Marriott et al. 2003]. A finite state automaton models the permitted resource usages. If the language generated by the grammar is not included in the language accepted by automaton, the program is instrumented with the local checks and the tracking operations needed to make it obey the policy. Both Erlingsson and Schneider [1999] and Thiemann [2003] specialize security automata to programs by inserting security checks into the object code generated by a compiler. The first article reports on work on the field, and describes prototypes for the machine architectures Intelx86 and Java JVML. The second article faces the problem in the more abstract framework of the λ-calculus and exploits type specialization to also remove runtime operations on the security state. Unlike Colcombet and Fradet [2000], Erlingsson and Schneider [1999], Marriott et al. [2003], and Thiemann [2003], our $\lambda^{[]}$also allows for local, parametric policies, and for dynamically created resources. In Colcombet and Fradet [2000] and Marriott et al. [2003] when a program cannot be statically proved to obey the global policy, it is instrumented with runtime checks that will abort its execution before the policy is violated. Although not presented here, we also proposed a similar approach in Bartoletti et al. [2005a].

In Walker [2000], static and dynamic techniques are combined with proof-carrying code [Necula 1997]. Security properties are specified by *security automata* [Bauer et al. 2002; Schneider 2000]. Types encode assertions about program security. When a security-unaware program is compiled, a centralized security policy tells where to insert local checks, in order to obtain provably-secure compiled code. An optimization phase follows: Whenever a check is removed, it is replaced by a proof that the optimized code is still safe. Before executing a piece of code, a certified verifier ensures that it respects the centralized security policy. Thus, compilers are no longer required to belong to the trusted computing base.

In Besson et al. [2005, 2001], local checks are related with global policies, in the context of stack-based execution monitors. They developed a static technique to prove that local checks enforce a given global policy, and to characterize when a program can safely call a stack-inspecting method.

Fong investigates *shallow history automata*, that can only record a finite approximation of the actual execution history [Fong 2004]. These automata can

keep track of the *set* of past access events, rather than the *sequence* of events. Although shallow history automata can express some interesting security properties, they are clearly less expressive than our usage policies.

Wang et al. propose a model for history-based access control [Wang et al. 2006]. They use control flow graphs enriched with permissions and a primitive to check them, similarly to Bartoletti et al. [2004]. The runtime permissions are the intersection of the static permissions of all the nodes visited in the past. The model-checking technique can decide in EXPTIME (in the number of permissions) if all the permitted traces of the graph respect a given regular property on its nodes. Unlike our local policies, that can enforce any regular policy on traces, the technique of Wang et al. [2006] is less general because there is no way to enforce a policy unless it is encoded as a suitable assignment of permissions to nodes.

Our policy framings roughly resemble the scoped methods of Tan et al. [2003]. Their 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 that describes the permitted sequences of access events. Methods must explicitly declare the sequence of events they may produce, while in our model these sequences are inferred by a type and effect system.

8. CONCLUSIONS AND FUTURE WORK

We proposed a novel approach to the resource usage problem, within an extension of the λ-calculus that features access/creation of resources, and regular usage policies with a local scope. To efficiently enforce policies, we have exploited a two-step static analysis, resulting in a mixed approach to resource usage control. We defined a type and effect system that overapproximates the runtime behavior of a program as a history expression. In spite of the augmented flexibility given by the nesting of policies scopes and by resource creation, we transformed history expressions so that model-checking their validity is decidable. When a history expression is valid, we can safely dispose the execution monitor. Otherwise, the soft-typing approach in Bartoletti et al. [2005a] allows for substituting local checks for local policies, thus making the dynamic control of accesses efficient. The approach proposed in this article has been pushed further in Bartoletti et al. [2009] to design and implement an extension of the security mechanism of Java, based on history-based local policies. While most of the runtime checking and inference of types and effects had to be redesigned to cope with an object-oriented language like Java, the security model and the verification algorithm are exactly those presented here. Our preliminary experiments show our approach scalable to actual programming languages and real-world applications.

We outline next some directions for future work. To improve the accuracy of types, we plan to relax the constraint that a single name can appear in pure types S. For instance, this will allow for the following typing judgements:

$$\vdash e = \texttt{new}\ x\ \texttt{in}\ \texttt{new}\ y\ \texttt{in}\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y : \nu n.\, \nu n'.\, \{n, n'\} \rhd new(n) \cdot new(n')$$

$$\vdash \alpha(e) : \mathbf{1} \rhd \nu n.\, \nu n'.\, new(n) \cdot new(n') \cdot (\alpha(n) + \alpha(n'))$$

whereas in the current treatment, we would have the less precise judgements.

$$\vdash e : \{?\} \rhd \nu n.\, \nu n'.\, new(n) \cdot new(n')$$
$$\vdash \alpha(e) : \mathbf{1} \rhd \nu n.\, \nu n'.\, new(n) \cdot new(n') \cdot \alpha(?)$$

We conjecture that all the existing results are preserved by this extension, at the price of a more complex proof obligation (see Appendix C).

A further improvement of the precision of our system would come from allowing subtyping of functional types. We plan to extend Definition 5.2 with the rule $\tau \to \zeta \sqsubseteq \tau' \to \zeta'$ if $\tau' \sqsubseteq \tau$ and $\zeta \sqsubseteq \zeta'$ (i.e., controvariant in the argument and covariant in the result). Consider for instance the following $\lambda^{[]}$function.

$$f \;=\; \lambda x.\, (\mathtt{if}\; b\; \mathtt{then}\; \lambda.\, \alpha\; \mathtt{else}\; x); x$$

With the current type and effect system, we have, for all pure types τ:

$$\vdash f\,(\lambda.\,\beta) : (\tau \to (\mathbf{1} \rhd \alpha + \beta)) \rhd \varepsilon.$$

Note, however, that the function $\lambda.\,\alpha$ is discarded, and so we would like to have the following, more accurate, typing.

$$\vdash f\,(\lambda.\,\beta) : (\tau \to (\mathbf{1} \rhd \beta)) \rhd \varepsilon$$

Subtyping of functional types would allow for such a typing, using the weakening: $\tau \to (\mathbf{1} \rhd \beta) \sqsubseteq \tau \to (\mathbf{1} \rhd \alpha + \beta)$ within the typing judgement of f. This issue has been considered in Skalka et al. [2008], and we feel their results can be also carried over our framework.

To put our proposal at work, it is mandatory to design and implement a type and effect inference algorithm. We have some preliminary result, obtained by extending the inference algorithm in Skalka et al. [2008], which, however, cannot handle creation of fresh resources.

The language of history expressions has two simple extensions. The first consists in attaching policies to resources upon creation, similarly to Igarashi and Kobayashi [2002]. The construct $\nu(n : \varphi).\ H$ is meant to enforce the policy φ on the freshly created resource. An encoding into $\lambda^{[]}$is possible. First, the whole history expression has to be sandboxed with the policy $\varphi_\nu(x)$, obtained from $\varphi(x)$ by adding a new initial state q_ν and an edge labelled *check(x)* from q_ν to the old initial state. The encoding then transforms $\nu(n : \varphi).\ H$ into $\nu n.\, check(n) \cdot H$. The second extension consists in allowing parallel history expressions $H|H'$, which come at hand when modelling multithreaded programs. Model-checking is still possible by transforming history expressions into Basic Parallel Processes [Christensen 1993] instead of BPAs. However, the time complexity becomes exponential in the number of parallel branches [Mayr 1998].

A further research direction consists in extending $\lambda^{[]}$to a distributed setting, to study secure discovery and orchestration of services. Some preliminary work on this line has been done in [Bartoletti et al. 2006, 2005b], where services are modeled as computational units that may invoke each others through a *call-by-contract* primitive $\mathtt{req}\ \tau \xrightarrow{\varphi} \tau'$. To match such a contract, the invoked service must have type $\tau \to \tau'$, and it must also guarantee that all the histories it generates obey the policy φ. Our previous work needs to be extended by considering resource creation and usage policies with parameters.

ELECTRONIC APPENDIX

The electronic appendix for the article can be accessed in the ACM Digital Liberary by visiting the following URL: `http://www.acm.org/pubs/citations/journals/toplas/20--/p1-URLend`.

Next, we summarise the contents of the four appendixes. Appendix A contains some typing examples. Appendix B has some auxiliary definitions and lemmata, along with their proofs, that are needed for proving the main theorems of the article. Appendix C establishes that an expression with a valid effect will violate no security policy at runtime. Besides some technical results about types and effects, it translates expressions with security framings into expressions with framing events, preserving the semantics. We then define a big-step operational semantics for $\lambda^{[]}$, and prove it equivalent to the small-step semantics. Using the big-step semantics we prove subject reduction, the correctenss of effects (Theorem 5.4), and type safety (Theorem 5.5). Appendix D proves the correctness of our verification method (Theorem 6.13). After a few technical results (correctness of regularization of history expressions in Definition 6.3 and of redundant framings elimination in Theorem 6.4), we prove that validity of histories amounts to acceptance by the relevant framed policy automaton (through Lemmata 3.4 and 6.6). Finally, we show that history expressions are bisimilar to their related BPAs.

ACKNOWLEDGMENTS

The authors warmly thank the anonymous referees for their comments and suggestions.

REFERENCES

Bartoletti, M. 2009. Usage automata. In *Proceedings of the Workshop on Issues in the Theory of Security*. Lecture Notes in Computer Science, vol. 5511. Springer, Berlin, 52–69.

Bartoletti, M., Costa, G., Degano, P., Martinelli, F., and Zunino, R. 2009. Securing Java with local policies. *J. Object Technol. 8*, 4, 5–32.

Bartoletti, M., Degano, P., and Ferrari, G.-L. 2004. Stack inspection and secure program transformations. *Int. J. Inform. Secur. 2*, 3-4, 187–217.

Bartoletti, M., Degano, P., and Ferrari, G.-L. 2005a. Checking risky events is enough for local policies. In *Proceedings of the 9th Italian Conference on Theoretical Computer Science (ICTCS)*. Lecture Notes in Computer Science, vol. 3701. Springer, Berlin, 97–112.

Bartoletti, M., Degano, P., and Ferrari, G.-L. 2005b. Enforcing secure service composition. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW-18)*. IEEE Computer Society, Los Alamitos, 211–223. (Full version to appear in *J. Comput. Secur*.

Bartoletti, M., Degano, P., and Ferrari, G.-L. 2005c. History-based access control with local policies. In *Proceedings of the 8th Foundations of Software Science and Computational Structures (FOSSACS)*. Lecture Notes in Computer Science, vol. 3441. Springer, Berlin, 316–332.

Bartoletti, M., Degano, P., and Ferrari, G.-L. 2006. Types and effects for secure service orchestration. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop. (CSFW-19)*. IEEE Computer Society, Los Alamitos, 57–69.

Bartoletti, M., Degano, P., Ferrari, G.-L., and Zunino, R. 2007. Types and effects for resource usage analysis. In *Proceedings of the 10th Foundations of Software Science and Computational Structures (FOSSACS'07)*. Lecture Notes in Computer Science, vol. 4423. Springer, Berlin, 32–47.

BARTOLETTI, M., DEGANO, P., FERRARI, G. L., AND ZUNINO, R. 2008. Model checking usage policies. In *Proceedings of the 4th Trustworthy Global Computing*. Lecture Notes in Computer Science, vol. 5474. Springer, Berlin, 19–35.

BARTOLETTI, M. AND ZUNINO, R. 2008. LocUsT: A tool for checking usage policies. Tech. rep. TR-08-07, Dipartimento Informatica, Universita Pisa.
`http://compass2.di.unipi.it/TR/Files/TR-08-07.pdf.gz`.

BAUER, L., LIGATTI, J., AND WALKER, D. 2002. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS)*.

BAUER, L., LIGATTI, J., AND WALKER, D. 2005. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 305–314.

BERGSTRA, J. A. AND KLOP, J. W. 1985. Algebra of communicating processes with abstraction. *Theor. Comput. Sci. 37*, 77–121.

BESSON, F., DE GRENIER DE LATOUR, T., AND JENSEN, T. P. 2005. Interfaces for stack inspection. *J. Functional Program. 15*, 2, 179–217.

BESSON, F., JENSEN, T. P., MÉTAYER, D. L., AND THORN, T. 2001. Model-checking security properties of control flow graphs. *J. Comput. Secur. 9*, 3, 217–250.

BRADFIELD, J. C. 1996. On the expressivity of the modal mu-calculus. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS'96)*. Lecture Notes in Computer Science, vol. 1046. Springer, Berlin, 479–490.

CHAKI, S., RAJAMANI, S. K., AND REHOF, J. 2002. Types as models: Model-checking message-passing programs. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 45–57.

CHRISTENSEN, S. 1993. Decidability and decomposition in process algebras. Ph.D. thesis, Edinburgh University.

COLCOMBET, T. AND FRADET, P. 2000. Enforcing trace properties by program transformation. In *Proceedings of the 27th Annual Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 54–66.

DAM, M. 1997. On the decidability of process equivalences for the pi-calculus. *Theor. Comput. Sci. 183*, 2, 215–228.

ERLINGSSON, U. AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the Workshop on New Security Paradigms*. ACM, New York, 87–95.

ESPARZA, J. 1994. On the decidability of model checking for several μ-calculi and Petri nets. In *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (CAAP'94)*. Lecture Notes in Computer Science, vol. 787. Springer, Berlin, 115–129.

FONG, P. W. 2004. Access control by tracking shallow execution history. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'04)*. IEEE Computer Society, Los Alamitos, 43–55.

FOURNET, C. AND GORDON, A. D. 2003. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst. 25*, 3, 360–399.

HAMLEN, K. W., MORRISETT, J. G., AND SCHNEIDER, F. B. 2006. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst. 28*, 1, 175–205.

IGARASHI, A. AND KOBAYASHI, N. 2002. Resource usage analysis. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 331–342.

IWAMA, F., IGARASHI, A., AND KOBAYASHI, N. 2006. Resource usage analysis for a functional language with exceptions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, 38–47.

KOBAYASHI, N. 2003. Time regions and effects for resource usage analysis. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. ACM, New York, 50–61.

KOZEN, D. 1983. Results on the propositional μ-calculus. *Theor. Comput. Sci. 27*, 333–354.

MARRIOTT, K., STUCKEY, P. J., AND SULZMANN, M. 2003. Resource usage verification. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS'03)*. Lecture Notes in Computer Science, vol. 2895. Springer, Berlin, 212–229.

MAYR, R. 1998. Decidability and complexity of model-checking problems for infinite-state systems. Ph.D. thesis, Technischen Universität München.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes, I and II. *Inform. Comput. 100*, 1, 1–40, 41–77.

NECULA, G. C. 1997. Proof-carrying code. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 106–119.

NIELSON, H. R. AND NIELSON, F. 1994. Higher-order concurrent programs with finite communication topology. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York.

SCHNEIDER, F. B. 2000. Enforceable security policies. *ACM Trans. Inform. Syst. Secur. 3*, 1, 30–50.

SKALKA, C. 2005. Trace effects and object orientation. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, New York, 139–150.

SKALKA, C. AND SMITH, S. 2004. History effects and verification. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*. Lecture Notes in Computer Science, vol. 3302. Springer, Berlin, 107–128.

SKALKA, C., SMITH, S., AND HORN, D. V. 2008. Types and trace effects of higher-order programs. *J. Functional Program. 18*, 2, 179–249.

TALPIN, J.-P. AND JOUVELOT, P. 1992. Polymorphic type, region and effect inference. *J. Functional Program. 2*, 3, 245–271.

TAN, G., OU, X., AND WALKER, D. 2003. Resource usage analysis via scoped methods. In *Proceedings of the Foundations of Object-Oriented Languages*.

THIEMANN, P. 2003. Program specialization for execution monitoring. *J. Functional Program. 13*, 3, 573–600.

WALKER, D. 2000. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 254–267.

WALLACH, D. S., APPEL, A. W., AND FELTEN, E. W. 2000. SAFKASI: A security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol. 9*, 4, 341–378.

WANG, J., TAKATA, Y., AND SEKI, H. 2006. HBAC: A model for history-based access control and its model-checking. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06)*. Lecture Notes in Computer Science, vol. 4189. Springer, Berlin, 263–278.

Received September 2008; accepted February 2009