History Effects and Verification

Christian Skalka and Scott Smith

The University of Vermont skalka@cs.uvm.edu
The Johns Hopkins University, scott@cs.jhu.edu

Abstract. This paper shows how type effect systems can be combined with model-checking techniques to produce powerful, automatically verifiable program logics for higher-order programs. The properties verified are based on the ordered sequence of events that occur during program execution—an *event history*. Our type and effect systems automatically infer conservative approximations of the event histories arising at run-time, and model-checking techniques are used to verify logical properties of these histories.
Our language model is based on the λ-calculus. Technical results include a powerful type inference algorithm for a polymorphic type effect system, and a method for applying known model-checking techniques to the *history effects* inferred by the type inference algorithm, allowing static enforcement of history- and stack-based security mechanisms.

1 Introduction

Safe and secure program execution is crucial for modern information systems, but is difficult to attain in practice due to both programmer errors and intentional attacks. Various programming language-based techniques increase program safety, by verifying at compile- and/or run-time that programs possess certain safety properties. This paper proposes a foundation for automated verification of program properties, by defining a process for automatically predicting *event histories* of program executions at compile-time, and for specifying and statically verifying properties of these histories. Our particular focus is on security applications, but the techniques apply broadly.

An *event* is a record of some program action, explicitly inserted into program code either manually (by the programmer) or automatically (by the compiler). Events are conceived of broadly, and can be a wide array of program actions—*e.g.* opening a file, an access control privilege activation, or entry to or exit from a critical region. *Histories* are ordered sequences of events; whenever an event is encountered during program execution, it is appended to the current history stream, and thus histories record the sequence of program events in temporal order. Program event histories are similar to audit trails, and provide many of the same benefits for monitoring system activity.

Verification consists of checking to make sure that histories are well-formed, *i.e.* the sequence of events prior to a check conforms to specifications. For example, if the program is sending and receiving data over an SSL socket, the relevant events are opening and closing of sockets, and reading and writing of data packets. An example event history produced by a program run could be:

```
ssl_open("snork.cs.jhu.edu",4434); ssl_hs_begin(4434);
ssl_hs_success(4434); ssl_put(4434); ssl_get(4434);
ssl_open("moo.cs.uvm.edu",4435); ssl_hs_begin(4434);
ssl_put(4435); ssl_close(4434); ssl_close(4435)
```

Here, `ssl_open` is a sample event with two arguments, a url and a port. Event histories can then be used to detect logical flaws or security violations. For SSL, sockets must first be opened, handshake begun, handshake success, and only then can data be get/put over the socket. For example, The above history is illegal because data is put on socket 4435 before notification has been received that handshake was successful on that socket.

The previous paragraph informally defines the well-formed event histories for SSL connections; codifying this assertion as a local check in a decidable logic would provide a rigorous definition of well-formedness, *and* would allow mechanical verification of it. Such a mechanical verification increases the reliability and security of programs.

1.1 Overview

Several systems have been developed along these general lines; perhaps the principal division between them is run-time [19, 1] *vs.* compile-time [3, 7, 4] verification. The focus of this paper is on the latter.

The systems all have in common the idea of extracting an abstract interpretation of some form from a program, and verifying properties of that abstraction. The MOPS system [7] compiles C programs to Push-down Automata (PDAs) reflecting the program control flow, where transitions are program transitions, and the automaton stack abstracts the program call stack. [14, 4] assume that some (undefined) algorithm has already converted a program to a control flow graph, expressed as a form of PDA.

These aforementioned abstractions work well for procedural programs, but are not powerful enough to fully address advanced language features such as higher-order functions or dynamic dispatch. Our approach is to develop a *type and effect* system to extract abstract interpretations from higher-order programs.

Most type systems predict the class of values to which a program will evaluate, but type and effect systems [24, 2] predict program side-effects as well. In our system, *history effects* capture the history events generated by programs, with the property that the effect of a program should conservatively approximate the history generated by that program during execution. History effects accomplish this by specifying a set of possible histories containing at least the realized execution history. History effects yield history streams via an LTS (Labelled Transition System) interpretation: the LTS transitions produce a label stream from a history effect.

The intepretation of history effects as LTSs allows the expression of program assertions as temporal logical formulae, and the automated verification of assertions via model-checking techniques [22]. Some of the aforecited systems also automatically verify assertions at compile-time via model-checking, including [3, 7, 4]. (None of these works however defines a rigorous process for extracting an LTS from higher-order programs; there are a few more closely related systems, discussed in the conclusion.) In these works, the specifications are temporal logics, regular languages, or finite automata, and the abstract control flow is extracted as an LTS in the form of a finite automaton, grammar, or PDA. These particular formats are chosen because these combinations of logics and abstract interpretations can be automatically model-checked.

Security automata [19] use finite automata for the specification and run-time enforcement of language safety properties. Systems have also been developed for statically verifying correctness of security automata using dependent types [25], and in a

$c \in \mathcal{C}$	*atomic constants*
$b ::= \mathsf{true} \mid \mathsf{false}$	*boolean values*
$v ::= x \mid \lambda_z x.e \mid c \mid b \mid \neg \mid \vee \mid \wedge \mid ()$	*values*
$e ::= v \mid e\, e \mid ev(e) \mid \phi(e) \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{let}\, x = v\, \mathsf{in}\, e$	*expressions*
$\eta ::= \epsilon \mid ev(c) \mid \eta; \eta$	*histories*
$E ::= [\,] \mid v\, E \mid E\, e \mid ev(E) \mid \phi(E) \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e$	*evaluation contexts*

Fig. 1. λ_{hist} language syntax

more general form as refinement types [17]. These systems do not extract any abstract interpretations, and so are in a somewhat different category than the aforementioned (and our) work. These systems also lack type inference, and do not use a logic that can be automatically verified.

Logical assertions can be *local*, concerning a particular program point, or *global*, defining the whole behavior required. However, access control systems [26, 1, 9], use local checks. Since we are interested in the static enforcement of access control mechanisms (see Sect. 6), the focus in this paper is on local, compile-time checkable assertions, though in principle the verification of global properties is also possible in our system.

1.2 The Technical Development

In the remaining text, we formalize our ideas in the system λ_{hist}: we define the type and effect system, and develop a type inference algorithm. We next show how the μ-calculus can be integrated as a static verification logic. A stack-history variant is also developed, by redefining the history as those events in active frames only. As an extended example of the system, we show how a rich model of Java stack inspection with first-class privilege parameters can be expressed via embedded μ-calculus formulae in this latter variant.

2 The Language λ_{hist}

In this Section we develop the syntax, semantics, and logical type theory for our language model, called λ_{hist}. History effect approximation and type safety results are stated in Theorem 1 and Theorem 2, respectively.

2.1 Syntax

The syntax of the theory λ_{hist} is given in Fig. 1. The base values include booleans and the unit value $()$. Expressions $\mathsf{let}\, x = v\, \mathsf{in}\, e$ are included to implement let-polymorphism in the type system (Sect. 2.3). Functions, written $\lambda_z x.e$, possess a recursive binding

$$\begin{array}{rcllr}
\eta, (\lambda_z x.e)v & \rightsquigarrow & \eta, e[v/x][\lambda_z x.e/z] & & (\beta) \\
\eta, \neg\mathsf{true} & \rightsquigarrow & \eta, \mathsf{false} & & (\mathit{notT}) \\
\eta, \neg\mathsf{false} & \rightsquigarrow & \eta, \mathsf{true} & & (\mathit{notF}) \\
\eta, \wedge\,\mathsf{true} & \rightsquigarrow & \eta, \lambda x.x & & (\mathit{andT}) \\
\eta, \wedge\,\mathsf{false} & \rightsquigarrow & \eta, \lambda\_.\mathsf{false} & & (\mathit{andF}) \\
\eta, \vee\,\mathsf{true} & \rightsquigarrow & \eta, \lambda\_.\mathsf{true} & & (\mathit{orT}) \\
\eta, \vee\,\mathsf{false} & \rightsquigarrow & \eta, \lambda x.x & & (\mathit{orF}) \\
\eta, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \rightsquigarrow & \eta, e_1 & & (\mathit{ifT}) \\
\eta, \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \rightsquigarrow & \eta, e_2 & & (\mathit{ifF}) \\
\eta, \mathsf{let}\ x = v\ \mathsf{in}\ e & \rightsquigarrow & \eta, e[v/x] & & (\mathit{let}) \\
\eta, \mathit{ev}(c) & \rightsquigarrow & \eta; \mathit{ev}(c), () & & (\mathit{event}) \\
\eta, \phi(c) & \rightsquigarrow & \eta; \mathit{ev}_\phi(c), () & \text{if } \mathit{\Pi}(\phi(c), \hat{\eta}\ \mathit{ev}_\phi(c)) & (\mathit{check}) \\
\eta, E[e] & \rightarrow & \eta', E[e'] & \text{if } \eta, e \rightsquigarrow \eta', e' & (\mathit{context})
\end{array}$$

Fig. 2. λ_{hist} language semantics

mechanism where z is the self variable. We assume the following syntactic sugarings:

$$e_1 \wedge e_2 \triangleq \wedge e_1 e_2 \qquad e_1 \vee e_2 \triangleq \vee e_1 e_2 \qquad \lambda x.e \triangleq \lambda_z x.e \quad z \text{ not free in } e$$

$$\lambda\_.e \triangleq \lambda x.e \quad x \text{ not free in } e \qquad e_1; e_2 \triangleq (\lambda\_.e_2)(e_1)$$

Events ev are named entities parameterized by constants c (we treat only the unary case in this presentation, but the extension to n-ary events is straightforward). These constants $c \in \mathcal{C}$ are abstract; this set could for example be strings or IP addresses. Ordered sequences of these events constitute histories η, which maintain the sequence of events experienced during program execution. We let $\hat{\eta}$ denote the string obtained from this sequence by removing delimiters (;). History assertions ϕ, also parameterized by constants c, may be used to implement history checks. These assertions are in a to-be-specified logical syntax (Sect. 4). We presuppose existence of a meaning function $\mathit{\Pi}$ such that $\mathit{\Pi}(\phi(c), \hat{\eta})$ holds iff $\phi(c)$ is valid for $\hat{\eta}$; we also leave the meaning function $\mathit{\Pi}$ abstract until later (Sect. 4).

Parameterizing events and predicates with constants c allows for a more expressive event language; for example, in Sect. 6 we show how the parameterized privileges of Java stack inspection can be encoded with the aid of these parameters.

2.2 Semantics

The operational semantics of λ_{hist} is defined in Fig. 2 via the call-by-value small step reduction relations $\rightsquigarrow$ and $\rightarrow$ on configurations η, e, where η is the history of run-time program events. We write $\rightarrow^\star$ to denote the reflexive, transitive closure of $\rightarrow$. Note that in the *event* reduction rule, an event $ev(c)$ encountered during execution is added to the end of the history. The *check* rule specifies that when a configuration η, ϕ is

$\alpha \in \mathcal{V}_s, t \in \mathcal{V}_\tau, h \in \mathcal{V}_H, \beta \in \mathcal{V}_s \cup \mathcal{V}_\tau \cup \mathcal{V}_H$	*variables*
$s \ ::= \alpha \mid c$	*singletons*
$\tau \ ::= t \mid \{s\} \mid \tau \xrightarrow{H} \tau \mid \mathit{bool} \mid \mathit{unit}$	*types*
$\sigma \ ::= \forall \bar{\beta}.\tau$	*type schemes*
$H ::= \epsilon \mid h \mid ev(s) \mid H; H \mid H \vert H \mid \mu h.H$	*history effects*
$\Gamma \ ::= \varnothing \mid \Gamma; x : \sigma$	*type environments*

Fig. 3. λ_{hist} type syntax

encountered during execution, the "check event" $ev_\phi(c)$ is appended to the end of η, and $\phi(c)$ is required to be satisfied by the current history η, according to our meaning function Π. The reasons for treating checks as dynamic events is manifold; for one, some checks may ensure that other checks occur historically. Also, this scheme will simplify the definition of Π, as well as history typing and verification, as discussed in Sect. 4. In case a check fails at runtime, execution is "stuck"; formally:

Definition 1. *We say that a configuration η, e is* stuck *iff e is not a value and there does not exist η' and e' such that $\eta, e \to \eta', e'$. If $\epsilon, e \to^\star \eta, e'$ and η, e' is stuck, then e is said to* go wrong.

The following example demonstrates the basics of syntax and operational semantics.

Example 1. Let the function f be defined as:

$$f \triangleq \lambda_z x.\mathsf{if}\, x\, \mathsf{then}\, ev_1(c)\, \mathsf{else}\, (ev_2(c); z(\mathsf{true}))$$

Then, in the operational semantics, we have:

$$\epsilon, f(\mathsf{false}) \to^\star ev_2(c); ev_1(c), ()$$

since the initial call to f will cause ev_2 to be added to the history, followed by a recursive call to f that hits its basis, where event ev_1 is encountered.

2.3 Logical Type System

In the type analysis, we are challenged to statically identify the histories that result during execution, for which purpose we introduce *history effects H*. In essence, history effects H conservatively approximate histories η that may develop during execution, by representing a set of histories containing at least η. A history effect may therefore be an event $ev(c)$, or a sequencing of history effects $H_1; H_2$, a nondeterministic choice of history effects $H_1|H_2$, or a μ-bound history effect $\mu h.H$ which finitely represents the set of histories that may be generated by a recursive function. History types may contain predicate events $ev_\phi(c)$, allowing us to verify predicate checks at the right points in history effect approximations of "historical developments". Noting that the syntax of histories η is the same as linear, variable-free history effects: We abuse syntax and let η

VAR
$$\frac{\mathit{\Gamma}(x) = \forall\bar{\beta}.\tau}{\mathit{\Gamma}, \epsilon \vdash x : \tau[\bar{\tau}/\bar{\beta}]}$$

BOOL
$$\mathit{\Gamma}, \epsilon \vdash b : \mathit{bool}$$

UNIT
$$\mathit{\Gamma}, \epsilon \vdash () : \mathit{unit}$$

AND
$$\mathit{\Gamma}, \epsilon \vdash \wedge : \mathit{bool} \xrightarrow{\epsilon} \mathit{bool} \xrightarrow{\epsilon} \mathit{bool}$$

OR
$$\mathit{\Gamma}, \epsilon \vdash \vee : \mathit{bool} \xrightarrow{\epsilon} \mathit{bool} \xrightarrow{\epsilon} \mathit{bool}$$

NOT
$$\mathit{\Gamma}, \epsilon \vdash \neg : \mathit{bool} \xrightarrow{\epsilon} \mathit{bool}$$

CONST
$$\mathit{\Gamma}, \epsilon \vdash c : \{c\}$$

WEAKEN
$$\frac{\mathit{\Gamma}, H \vdash e : \tau}{\mathit{\Gamma}, H|H' \vdash e : \tau}$$

EVENT
$$\frac{\mathit{\Gamma}, H \vdash e : \{s\}}{\mathit{\Gamma}, H; \mathit{ev}(s) \vdash \mathit{ev}(e) : \mathit{unit}}$$

CHECK
$$\frac{\mathit{\Gamma}, H \vdash e : \{s\}}{\mathit{\Gamma}, H; \mathit{ev}_\phi(s) \vdash \phi(e) : \mathit{unit}}$$

IF
$$\frac{\mathit{\Gamma}, H_1 \vdash e_1 : \mathit{bool} \qquad \mathit{\Gamma}, H_2 \vdash e_2 : \tau \qquad \mathit{\Gamma}, H_2 \vdash e_3 : \tau}{\mathit{\Gamma}, H_1; H_2 \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau}$$

ABS
$$\frac{\mathit{\Gamma}; x : \tau_1; z : \tau_1 \xrightarrow{H} \tau_2, H \vdash e : \tau_2}{\mathit{\Gamma}, \epsilon \vdash \lambda_z x.e : \tau_1 \xrightarrow{H} \tau_2}$$

APP
$$\frac{\mathit{\Gamma}, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \qquad \mathit{\Gamma}, H_2 \vdash e_2 : \tau'}{\mathit{\Gamma}, H_1; H_2; H_3 \vdash e_1 e_2 : \tau}$$

LET
$$\frac{\mathit{\Gamma}, \epsilon \vdash v : \tau' \qquad \bar{\beta} \cap \mathsf{fv}(\mathit{\Gamma}) = \varnothing \qquad \mathit{\Gamma}; x : \forall\bar{\beta}.\tau', H \vdash e : \tau}{\mathit{\Gamma}, H \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}$$

Fig. 4. λ_{hist} logical typing rules

also range over linear, variable-free history effects, interpreting histories η as the same history effect.

The syntax of types for λ_{hist} is given in Fig. 3. In addition to histories, we include function types $\tau_1 \xrightarrow{H} \tau_2$, where H represents the histories that may result by use of the function. Events are side-effects, and so these function types are a form of effect type [24, 2]. Additionally, since events and predicates are parameterized in history types, we must be especially accurate with respect to our typing of constants. Thus, we adopt a very simple form of singleton type $\{c\}$ [23], where only atomic constants can have singleton type. Types contain three sorts of variables; regular type variables t, singleton type variables α, and history effect type variables h; β ranges over all sorts of variables. Universal type schemes $\forall\bar{\beta}.\tau$ bind any sort of type variable in τ. We write τ as syntactic sugar for $\forall\bar{\beta}.\tau$ with $\bar{\beta} \cap \mathsf{fv}(\tau) = \varnothing$.

Source code type derivation rules for judgements of the form $\mathit{\Gamma}, H \vdash e : \tau$ are given in Fig. 4, where $\mathit{\Gamma}$ is an environment of variable typing assumptions. Intuitively, the history effect H in judgements represents the set of histories that may arise during execution of e (this intuition is formalized in the upcoming Theorem 1). For example:

Example 2. Let f defined as in Example 1, and let:

$$H \triangleq (\mu h.\mathit{ev}_1(c) \mid \mathit{ev}_2(c); h); \mathit{ev}_3(c')$$

Then, the following judgements are derivable[1]:

$$\varnothing, \epsilon \vdash f : \mathit{bool} \xrightarrow{\mu h.ev_1(c)|ev_2(c);h} \mathit{unit} \qquad \varnothing, H \vdash f(\mathsf{false}); ev_3(c') : \mathit{unit}$$

Note that let-polymorphism over types, singletons, and history effects is included in our system. A typing $\Gamma, H \vdash e : \tau$ is *valid* iff it is derivable, and if H is valid in the interpretation defined in the next section. Additionally, typing judgements are identified modulo equivalence of history effects, as characterized in the next section. We observe that the addition of history effects is a conservative extension to the underlying type system: by using weakening before each if-then-else typing, any derivation in the underlying history-effect-free type system may be replayed here.

2.4 Interpretation of History Effects and Type Safety

As alluded to previously, the interpretation of a history effect is, roughly, a set of histories. More accurately, we define the Labelled Transition System (LTS) interpretation of history effects as sets of *traces*, which include a $\downarrow$ symbol to denote termination. Traces may be infinite, because programs may not terminate.

Definition 2. *Our interpretation of histories will be defined via strings (called* traces*) denoted θ, over the following alphabet:*

$$a ::= ev(c) \mid \epsilon \mid \downarrow$$

We let Θ range over prefix-closed sets of traces.

Sets of traces are obtained from history effects by viewing the latter as programs in a simple nondeterministic transition system:

Definition 3. *The history effect transition relation is defined as follows:*

$$ev(c) \xrightarrow{ev(c)} \epsilon \qquad H_1|H_2 \xrightarrow{\epsilon} H_1 \qquad H_1|H_2 \xrightarrow{\epsilon} H_2 \qquad \mu h.H \xrightarrow{\epsilon} H[\mu h.H/h]$$

$$\epsilon; H \xrightarrow{\epsilon} H \qquad\qquad H_1; H_2 \xrightarrow{a} H_1'; H_2 \text{ if } H_1 \xrightarrow{a} H_1'$$

We may formally determine the sets of traces Θ associated with a closed history effect in terms of the transition relation:

Definition 4. *The interpretation of history effects is defined as follows:*

$$[\![H]\!] = \{a_1 \cdots a_n \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} H'\} \cup \{a_1 \cdots a_n \downarrow \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} \epsilon\}$$

Any history effect interpretation is clearly prefix-closed. In this interpretation, an infinite trace is viewed as the set of its finite prefixes.

[1] These and following examples simplify history effects according to the equivalences specified in [21].

$\tau = \tau' \in \mathcal{C}$	*constraints*
$H \sqsubseteq H' \in \mathcal{HC}$	*effect constraints*
$C \in 2^{\mathcal{C}}$	*constraint sets*
$HC \in 2^{\mathcal{HC}}$	*effect constraint sets*
$k ::= \tau/C, HC$	*constrained types*
$\varsigma ::= \forall\bar{\beta}.k$	*constrained type schemes*
$\Gamma ::= \varnothing \mid \Gamma; x : \varsigma$	*constrained type environments*

Fig. 5. Constained types and environments

History effect equivalence is defined via this interpretation, i.e. $H_1 = H_2$ iff $[\![H_1]\!] = [\![H_2]\!]$. This relation is in fact undecidable: histories are equivalent to BPA's (basic process algebras) [21], and their trace equivalence is known to be undecidable [6].

We then base the validity of a history effect on validity of the assertion events that occur in traces in its interpretation. In particular, for any given predicate event in a trace, that predicate must hold for the immediate prefix trace that precedes it:

Definition 5. *A history effect H is* valid *iff for all $\theta\, ev_\phi(c) \in [\![H]\!]$ it is the case that:*

$$\Pi(\phi(c), \theta\, ev_\phi(c))$$

holds. A type judgement $\Gamma, H \vdash e : \tau$ is valid *iff it is derivable and H is valid.*

An important aspect of this definition is that $[\![H]\!]$ contains *prefix* traces. Essentially, if $\Gamma, H \vdash e : \tau$ is derivable, then $[\![H]\!]$ contains "snapshots" of e's potential run-time history at every step of reduction, so that validity of H implies validity of any check that occurs at run-time, "part-way through" the full program history as approximated by H. This is formally realized in our primary approximation result for history effects:

Theorem 1. *If $\Gamma, H \vdash e : \tau$ is derivable for closed e and $\epsilon, e \to^\star \eta, e'$ then $\hat{\eta} \in [\![H]\!]$.*

which in turn is the basis of a type safety result for λ_{hist}:

Theorem 2 (λ^c_{hist} Type Safety). *If $\Gamma, H \vdash e : \tau$ is valid for closed e then e does not go wrong.*

Proofs and details of these results are given in [21].

3 Polymorphic Type and Effect Inference

We implement our type and effect system using a constraint-based technique. This allows us to adopt existing methods [10] for realizing let-polymorphism, with appropriate modifications in the presence of history effects– in particular, the addition of effect constraints HC to capture weakening in inference. We define the new categories of type and effect constraints, constraint sets, constrained types, and constrained type schemes for the algorithm in Fig. 5. We write τ as syntactic sugar for $\forall\bar{\beta}.\tau/\varnothing, \varnothing$ when $\bar{\beta} \cap \text{fv}(\tau) = \varnothing$.

VAR

$$\frac{\Gamma(x) = \forall\bar{\beta}.k}{\Gamma, \epsilon \vdash_W x : k[\bar{\beta}'/\bar{\beta}]}$$

CONST

$$\Gamma, \epsilon \vdash_W c : \{c\}/\varnothing, \varnothing$$

EVENT

$$\frac{\Gamma, H \vdash_W e : \tau/C, HC}{\Gamma, H; ev(\alpha) \vdash_W ev(e) : \mathit{unit}/C \cup \{\tau = \{\alpha\}\}, HC}$$

CHECK

$$\frac{\Gamma, H \vdash_W e : \tau/C, HC}{\Gamma, H; ev_\phi(\alpha) \vdash_W \phi(e) : \mathit{unit}/C \cup \{\tau = \{\alpha\}\}, HC}$$

IF

$$\frac{\begin{array}{c}\Gamma, H_1 \vdash_W e_1 : \tau_1/C_1, HC_1 \\ \Gamma, H_2 \vdash_W e_2 : \tau_2/C_2, HC_2 \qquad \Gamma, H_3 \vdash_W e_3 : \tau_3/C_3, HC_3\end{array}}{\begin{array}{c}\Gamma, H_1; H_2|H_3 \vdash_W \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : t/C_1 \cup C_2 \cup C_3 \cup \{\tau_1 = \mathit{bool}, \tau_2 = t, \tau_3 = t\}, \\ HC_1 \cup HC_2 \cup HC_3\end{array}}$$

APP

$$\frac{\Gamma, H_1 \vdash_W e_1 : \tau_1/C_1, HC_1 \qquad \Gamma, H_2 \vdash_W e_2 : \tau_2/C_2, HC_2}{\Gamma, H_1; H_2; h \vdash_W e_1\ e_2 : t/C_1 \cup C_2 \cup \left\{\tau_1 = \tau_2 \xrightarrow{h} t\right\}, HC_1 \cup HC_2}$$

FIX

$$\frac{\Gamma; x : t; z : t \xrightarrow{h} t', H \vdash_W e : \tau/C, HC}{\Gamma, \epsilon \vdash_W \lambda_z x.e : t \xrightarrow{h} \tau/C \cup \left\{\tau = t'\right\}, HC \cup \{H \sqsubseteq h\}}$$

LET

$$\frac{\begin{array}{c}\Gamma, \epsilon \vdash_W v : \tau'/C', HC' \qquad \Gamma; x : \forall\bar{\beta}.\tau'/C', HC', H \vdash_W e : \tau/C, HC \\ \bar{\beta} = \mathsf{fv}(\tau', C', HC') - \mathsf{fv}(\Gamma) \qquad \psi = [\bar{\beta}'/\bar{\beta}]\end{array}}{\Gamma, H \vdash_W \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau/\psi(C') \cup C, \psi(HC') \cup HC}$$

Fig. 6. Type constraint inference rules for λ_{hist}

Substitutions ψ are fundamental to our inference method. Substitutions are mappings from type variables β to types, extended to types constraints and environments in the obvious manner. We define substitutions as *solutions* of constraints via interpretation in a partially ordered universe of monotypes:

Definition 6 (Interpretation). Monotypes *are types* τ *such that* $\mathrm{fv}(\tau) = \varnothing$. *We let* $\hat{\tau}$ *range over monotypes, and define* $\preccurlyeq$ *to be a partial ordering over monotypes such that* $\hat{\tau}_1 \xrightarrow{H} \hat{\tau}_2 \preccurlyeq \hat{\tau}_1' \xrightarrow{H'} \hat{\tau}_2'$ *implies* $\hat{\tau}_1' \preccurlyeq \hat{\tau}_1$ *and* $\hat{\tau}_2 \preccurlyeq \hat{\tau}_2'$ *and* $H \preccurlyeq H'$, *and* $H \preccurlyeq H'$ *implies* $[\![H]\!] \subseteq [\![H']\!]$. *An* interpretation ρ *is a total mapping from type variables to monotypes, extended to types and constraints in the usual manner (e.g. [18]).*

$$
\begin{aligned}
\mathit{unify}(\varnothing) &= \varnothing \\
\mathit{unify}(C \cup \{\tau = \tau\}) &= \mathit{unify}(C) \\
\mathit{unify}(C \cup \{\beta = \tau\}) &= \textbf{fail} \text{ if } \beta \in \mathrm{fv}(\tau), \text{ else} \\
&\ \mathit{unify}(C[\tau/\beta]) \circ [\tau/\beta] \\
\mathit{unify}(C \cup \{\tau = \beta\}) &= \textbf{fail} \text{ if } \beta \in \mathrm{fv}(\tau), \text{ else} \\
&\ \mathit{unify}(C[\tau/\beta]) \circ [\tau/\beta] \\
\mathit{unify}(C \cup \{\{s_1\} = \{s_2\}\}) &= \mathit{unify}(C \cup \{s_1 = s_2\}) \\
\mathit{unify}(C \cup \left\{\tau_1 \xrightarrow{h} \tau_2 = \tau_1' \xrightarrow{h'} \tau_2'\right\}) &= \mathit{unify}(C \cup \left\{h = h'\right\} \cup \left\{\tau_1 = \tau_1'\right\} \cup \left\{\tau_2 = \tau_2'\right\})
\end{aligned}
$$

Fig. 7. Constraint set unification algorithm

$$
\begin{aligned}
\mathit{MGS}(C, \mathit{HC}) &= \text{let } \psi_1 = \mathit{unify}(C) \text{ in } \mathit{MGS}_{\mathbf{H}}(\psi_1(\mathit{HC})) \circ \psi_1 \\
\mathit{bounds}(h, \mathit{HC}) &= H_1 | \cdots | H_n \quad \text{where } \{H_1, \ldots, H_n\} = \{H \mid H \sqsubseteq h \in \mathit{HC}\} \\
\mathit{MGS}_{\mathbf{H}}(\varnothing) &= \varnothing \\
\mathit{MGS}_{\mathbf{H}}(\mathit{HC}) &= \text{let } \psi = [(\mu h.\mathit{bounds}(h, \mathit{HC}))/h] \text{ in} \\
&\ \mathit{MGS}_{\mathbf{H}}(\psi(\mathit{HC} - \{H \sqsubseteq h \in \mathit{HC}\})) \circ \psi
\end{aligned}
$$

Fig. 8. Most general solution algorithm

Then, in addition to equality constraints on types, we posit a notion of $\sqsubseteq$ constraints on types; while type inference only treats such constraints on effects, a generalization of the relation to types is useful for characterizing type principality, as in Theorem 4:

Definition 7 (Constraint Solution). *We say that a substitution ψ* solves *or* satisfies *a constraint $\tau \sqsubseteq \tau'$, and we write $\psi \vdash \tau \sqsubseteq \tau'$, iff $\rho(\psi(\tau)) \preccurlyeq \rho(\psi(\tau'))$ for all ρ. We write $\vdash \tau \sqsubseteq \tau'$ iff $\varnothing \vdash \tau \sqsubseteq \tau'$. Finally, ψ solves an effect constraint set HC iff it solves every constraint in HC.*

The type inference rules are given in Fig. 6. These rules are nondeterministic only in the choice of type variables introduced in various rules; without loss of generality, we assume that all inference derivations are in *canonical* form, wherein fresh type variables are chosen wherever possible.

The use of effect constraints HC in the type inference rules allows necessary weakening of effects to be inferred where allowable, while equality constraints C enforce invariance elsewhere, in keeping with the logical system. To solve equality constraints, a standard unification technique is defined in Fig. 7; unification of history effects on function types is trivial, since only variables h annotate function types in any inferred constraint set C. Otherwise, as observed in Sect. 2, equality of history effects is undecidable. Solvability of effect constraint sets HC generated by inference is decidable as well, since these also adhere to a specific amenable form. In particular, the effect constraints generated by type inference will define a system of lower bounds on history effect variables:

Lemma 1. *If $\varnothing, H \vdash_W e : \tau/C, HC$ is derivable and $\psi = unify(C)$, then for all $H_1 \sqsubseteq H_2 \in \psi(HC)$, H_2 is a history effect variable h.*

The result follows since HC is a system of lower bounds as a consequence of the inference rules, and ψ preserves this property since unification generates at most a renaming of any upper bound h in HC.

Thus, a most general solution algorithm $MGS_{\mathbf{H}}$ for effect constraint sets is defined in Fig. 8, which joins the lower bounds inferred for any h, and μ-binds h on this join to account for the possibility of recursive effect constraints. Note that since each distinct upper bound h represents the effect of some function, μ-bound effects are function effects in inferred types, a fact that will become significant in Sect. 5. The $MGS_{\mathbf{H}}$ algorithm is composed with unification to obtain a complete solution for inferred type constraints. A soundness result is obtainable for this system, as follows:

Theorem 3 (Soundness). *If the judgement $\varnothing, H \vdash_W e : \tau/C, HC$ is derivable and $\psi = MGS(C, HC)$, then $\varnothing, \psi(H) \vdash e : \psi(\tau)$ is derivable.*

Completeness is also obtainable, comprising a principal types property, as follows:

Theorem 4 (Completeness). *If $\varnothing, H \vdash e : \tau$ is derivable, then so is $\varnothing, H' \vdash_W e : \tau'/C, HC$ with $MGS(C, HC) = \psi$, where $\vdash \psi(H') \sqsubseteq H$ and $\vdash \psi' \circ \psi(\tau') \sqsubseteq \tau$ for some ψ'.*

We have developed a prototype implementation of type inference in OCaml, that has confirmed correctness of the algorithm. The implementation has also demonstrated the usefulness of simplification techniques for enhancing readability of history effects, e.g. tranforming $\mu h.H$ to H in case h does not appear free in H, transforming $\epsilon; H$ to H, etc.

4 Verification of History Checks

We have described the dynamic model of λ_{hist}, which includes a new event history component in configurations. We have also described a type system that conservatively approximates run-time histories. However, we have been abstract so far with respect to the form of history *checks*, basing safety of computation and validity of typing on a yet-to-be-defined notion of history check validity. To fill in these details, we first define a logic for run-time checks, including a syntax for expressing checks as predicates in the logic, together with a notion of validity for these checks that can be automatically verified in the run-time system. Secondly, we define a means of verifying validity of history effects, as defined in Definition 5, where check events that are predicted by the history effect analysis are automatically shown to either succeed or fail in the relevant context. The latter point is necessary to address because, even though validity of history effects has been defined, the notion is logical but not algorithmic; in particular, $[\![H]\!]$ may be an infinite set. We accomplish automated verification using a temporal logic and model-checking techniques, allowing us to reuse existing algorithms and results for history effect verification.

$$\begin{aligned}
[\![\textbf{true}]\!]_V &= \Theta^\infty \\
[\![x]\!]_V &= V(x) \\
[\![\neg\phi]\!]_V &= \Theta^\infty - [\![\phi]\!]_V \\
[\![\phi_1 \wedge \phi_2]\!]_V &= [\![\phi_1]\!]_V \cap [\![\phi_2]\!]_V \\
[\![(a)\phi]\!]_V &= \{\theta^\infty \in \Theta^\infty \mid \theta^\infty = a;\theta_1^\infty \text{ and } \theta_1^\infty \in [\![\phi]\!]_V\} \\
[\![\boldsymbol{\nu} x.\phi]\!]_V &= \bigcup\{W \subseteq \Theta^\infty \mid W \subseteq [\![\phi]\!]_{V[x\mapsto W]}\}
\end{aligned}$$

Fig. 9. Semantics of the linear-time μ-calculus

4.1 Verified Checks in the Linear μ-calculus

While a plethora of model-checking logics are available, we use the μ-calculus [16] because it is powerful and is syntactically close to histories H. Further, efficient techniques for the automated verification of μ-calculus formulas on BPA processes have been developed [6, 11], and history effects are isomorphic to BPA's [21]. We use the *linear* variant of the μ-calculus [11] because at run-time only one linear trace of events is known, and so history effects H and runtime histories η can only be compared in a linear-time logic.

Definition 8. *The syntax of the linear μ-calculus is:*

$$\phi ::= x \mid \textbf{true} \mid \textbf{false} \mid (a)\phi \mid \boldsymbol{\mu} x.\phi \mid \boldsymbol{\nu} x.\phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

Here, a ranges over arbitrary transition labels; in particular, a may range over events $ev(c)$. The semantics of the linear μ-calculus is defined in Fig. 9. This semantics is defined over potentially infinite traces $\theta^\infty \in \Theta^\infty$ that, unlike sets of traces θ, may not be prefix-closed. V denotes a mapping from μ-calculus variables to sets of potentially infinite traces, $\varnothing$ denotes the empty mapping. $[\![\phi]\!]$ is shorthand for $[\![\phi]\!]_\varnothing$. Several formulae are not given in this figure because they can be defined in terms of the others: $\phi_1 \vee \phi_2$ is $\neg(\neg\phi_1 \wedge \neg\phi_2)$, **false** is $\neg$**true**, and $\boldsymbol{\mu} x.\phi$ is $\neg(\boldsymbol{\nu} x.\neg\phi)$.

Since our history effect semantics is prefix-closed, we will explicitly prefix-close $[\![\phi]\!]$ so the two sets are compatible.

Definition 9. *The prefix closure $\Theta^\infty\!\downarrow$ of a set of infinite traces Θ^∞ is:*

$$\{\theta \mid \theta \textit{ is a prefix of some } \theta^\infty \in \Theta^\infty\} \cup \{\theta\!\downarrow \mid \theta \textit{ is finite and } \theta \in \Theta^\infty\}$$

Definition 10. *The formula ϕ is valid for H, written $H \Vdash \phi$, iff $[\![H]\!] \subseteq [\![\phi]\!]\downarrow$.*

This relation is decidable by known model-checking results [11] and the above mentioned equivalence of BPA processes and histories [21].

4.2 Relating History Effect and History Runtime Properties

We now instantiate the logic of history checks in λ_{hist} with linear μ-calculus formulae ϕ. As discussed above, the relation $H \Vdash \phi$ is decidable, so this will make a natural foundation for history effect verification.

One important requirement of this logic is that formulae must have truth values for a given history effect H, *and* for a history runtime η. The meaning of a formula ϕ under a run-time history η is taken by verifying $\hat{\eta} \in [\![\phi]\!] \downarrow$. We will define the history check interpretation function Π of Fig. 2 in terms of this relation (Definition 11).

In Theorem 1, history effects were shown to approximate dynamic histories—in particular, if $\epsilon, e \rightarrow^\star \eta, e'$ and $H, \Gamma \vdash e : \tau$ is derivable, then $\hat{\eta} \in [\![H]\!]$. The key result linking the static and dynamic histories is the following:

Lemma 2. *If $\hat{\eta} \in [\![H]\!]$ and $H \Vdash \phi$, then $\hat{\eta} \in [\![\phi]\!] \downarrow$.*

Proof. Immediate from Definition 10 and Definition 4.

Theorem 1 ensures that by the above Lemma, verifying a history effect entails verifying all history checks that may occur at run-time, meaning a combination of type inference and automated verification yields a sound static analysis for λ_{hist}.

4.3 Soundness of Verification

We are close to a complete definition of our logical framework for history checks and history effect verification. However, a few small points remain: firstly, in λ_{hist}, we are able to parameterize checks with constants with the syntax $\phi(c)$. To implement this, we specify a distinguished variable χ that may occur free in history checks, which is assumed to be instantiated with a history check parameter during verification.

Secondly, checks ϕ should express expected history patterns that may occur up to the point of the check. This is the same as requiring that if an event $ev_\phi(c)$ occurs historically (resp. is predicted statically), the history η that precedes it (resp. any history that may precede it as predicted by typing) must exhibit the pattern specified by ϕ. However, there is an infinite regress lurking here: a property ϕ that mentions an event $ev_\phi(c)$ suggests circularity in the syntax of ϕ. Thus, we introduce a distinguished label Now, that in any formula ϕ represents the relevant checkpoint of ϕ. This label is interpreted appropriately during verification.

We now stand ready to instantiate our logic and verification framework. In the dynamic system, this is accomplished by defining the language of history checks, and by defining the implementation of Π, our previously abstract representation of history check verification:

Definition 11 (Definition of Π). *The framework of Section 2 is officially instantiated to let ϕ range over linear μ-calculus formulae, where labels a in ϕ are defined as follows:*

$$a ::= ev(s) \mid \mathrm{Now}$$

and, letting $\chi \in \mathcal{V}_s$ be a distinguished variable for parameterizing constants c in ϕ, we define Π as follows:

$$\Pi(\phi(c), \theta) \iff \theta \in [\![\phi[c/\chi][ev_\phi(c)/\mathrm{Now}]]\!] \downarrow$$

Now, we specify what it means for a history effect to be verified; intuitively, it means that if a history effect H predicts the occurrence of a check event $ev_\phi(c)$, then H semantically entails ϕ instantiated with c. Formally:

Definition 12 (History Effect Verification). *A history effect H is* verified *iff for all subterms $ev_\phi(c)$ of H it is the case that:*

$$H \Vdash \phi[c/\chi][ev_\phi(c)/\mathrm{Now}]$$

The preceding construction completes the definition of the framework. Lemma 2 and definition of Π together yield the desired formal property for history effect verification as a Corollary:

Corollary 1 (Verification Soundness). *If H is verified, then H is valid.*

Proof. Immediate by Definition 5, Definition 11, Definition 12, and Lemma 2.

5 The Stack-Based Variation

In this section we define a stack-based variation on the framework of the previous sections, allowing properties of the runtime stack at a program point to be verified at compile-time. Instead of keeping track of *all* events, only events for functions on the current call stack are maintained, in keeping with a general stack-based security model (as in e.g. [14]). Assertions ϕ in this framework are run-time assertions about the *active* event sequence, not all events. While the stack-based model is somewhat distinct from the history-based model, we show that this variation requires only a minor "post-processing" of inferred history effects for a sound analysis. There are results showing how it is possible to directly model-check stack properties of a Push-Down Automata (PDA) computation [12]; our approach based on post-processing history effects represents an alternative method, which may also prove useful for modeling features such as exceptions: the raising of an exception implies any subsequent effect is discarded.

We note that our system captures a more fine-grained stack-based model than has been previously proposed; in particular, the use of stacks of histories allows the ordering of events within individual stack frames to be taken into account, along with the ordering of frames themselves.

5.1 Syntax and Semantics

The values, expressions, and evaluation contexts of $\lambda_{\mathrm{hist}}^{\mathtt{S}}$ are exactly those of λ_{hist}, extended with an expression form $\cdot e \cdot$ and evaluation context form $\cdot E \cdot$ for delimiting the scope of function activations. We impose the requirement that in any function $\lambda_z x.e$, there exist no subexpressions $\cdot e' \cdot$ of e. The operational semantics of $\lambda_{\mathrm{hist}}^{\mathtt{S}}$, is a relation on *configurations* $\mathtt{S}, e$, where $\mathtt{S}$ ranges over stacks of histories, defined in Fig. 10. The active security context for run-time checks is obtained from the history stack in configurations, by appending histories in the order they appear in the stack; to formalize this, we define the notation $\mathtt{S};\eta$ as follows:

$$\mathrm{nil};\eta = \eta \qquad\qquad \mathtt{S} :: \eta; \eta' = \mathtt{S}; \eta; \eta'$$

Selected rules for the reduction relations $\leadsto$ and $\to$ on configurations are then specified in Fig. 10 (those not specified are obtained by replacing metavariables η with $\mathtt{S}$ in reductions for other expression forms in Fig. 2). The history interpretation function Π is defined as for λ_{hist}.

$$\mathtt{S} ::= \text{nil} \mid \mathtt{S} :: \eta \qquad \textit{history stacks}$$

$$\begin{array}{rcll}
\mathtt{S}, (\lambda_z x.e)v & \rightsquigarrow & \mathtt{S} :: \epsilon, \cdot e[v/x][\lambda_z x.e/z]\cdot & (\beta) \\
\mathtt{S} :: \eta, ev(c) & \rightsquigarrow & \mathtt{S} :: \eta; ev(c), () & (\textit{event}) \\
\mathtt{S} :: \eta, \phi(c) & \rightsquigarrow & \mathtt{S} :: \eta; ev_\phi(c), () & (\textit{check}) \\
 & & \text{if } \Pi(\phi(c), (\mathtt{S}\hat{;}\eta) ev_\phi(c)) & \\
\mathtt{S} :: \eta, \cdot v \cdot & \rightsquigarrow & \mathtt{S}, v & (\textit{pop})
\end{array}$$

Fig. 10. Semantics of $\lambda^{\mathtt{S}}_{\text{hist}}$ (selected rules)

$$\begin{aligned}
\textit{stackify}(\epsilon) &= \epsilon \\
\textit{stackify}(\epsilon; H) &= \textit{stackify}(H) \\
\textit{stackify}(ev(c); H) &= ev(c); \textit{stackify}(H) \\
\textit{stackify}(h; H) &= h|\textit{stackify}(H) \\
\textit{stackify}((\mu h.H_1); H_2) &= (\mu h.\textit{stackify}(H_1)) \mid \textit{stackify}(H_2) \\
\textit{stackify}((H_1|H_2); H) &= \textit{stackify}(H_1; H) \mid \textit{stackify}(H_2; H) \\
\textit{stackify}((H_1; H_2); H_3) &= \textit{stackify}(H_1; (H_2; H_3)) \\
\textit{stackify}(H) &= \textit{stackify}(H; \epsilon)
\end{aligned}$$

Fig. 11. The $\textit{stackify}$ algorithm

5.2 Stackified History Effects

Although $\lambda^{\mathtt{S}}_{\text{hist}}$ uses stack rather than history contexts at run-time, we are able to use the type and verification framework developed previously, assigning types to $\lambda^{\mathtt{S}}_{\text{hist}}$ expressions in the same manner as λ_{hist} expressions. The only additional requirement will be to process history effects– to *stackify* them, yielding an approximation of the stack contexts that will evolve at run-time. The trick is to use μ-delimited scope in history types, since this corresponds to function scope in inferred types as discussed in Sect. 3, and function activations and deactivations induce pushes and pops at run-time. The $\textit{stackify}$ algorithm is defined inductively in Figure 11. This algorithm works over histories that are sequences; for histories that are not sequences, the last clause puts it into sequence form.

The last three clauses use history effect equalities characterized in [21] to "massage" history effects into appropriate form. Observe that the range of $\textit{stackify}$ consists of history effects that are all tail-recursive; stacks are therefore finite-state transition systems and more efficient model-checking algorithms are possible for stacks than for general histories [12].

Example 3. With a, b, c, d representing arbitrary events:

$$\textit{stackify}(a; (\mu h.b; c); (\mu h.c; (\epsilon|(d; h; a)))) = a; ((\mu h.b; c)|(\mu h.c; (\epsilon|(d; h)|(d; a))))$$

Validity of a type judgement $\Gamma, H \vdash e : \tau$, and type safety in the $\lambda_{\mathrm{hist}}^{\mathsf{S}}$, will hinge upon validity of $\mathit{stackify}(H)$. We obtain the desired result indirectly, via an equivalent, but more immediately applicable and technically convenient type logic. Details, omitted here for brevity, are given in [21]. The main result is Type Safety for stackified history typings, as follows; the definition of e "going wrong" here is analogous to Definition 1. Note that we make an expected restriction on expressions to ensure that the empty stack is never popped:

Theorem 5 **($\lambda_{\mathrm{hist}}^{\mathsf{S}}$ Type Safety).** *If the judgement $\Gamma, H \vdash e : \tau$ is derivable for closed e with no subexpressions of the form $\cdot e' \cdot$, and $\mathit{stackify}(H)$ is valid, then e does not go wrong.*

As a corollary of this result and Theorem 2, type inference and verification as developed for λ_{hist} may be re-used in the stack model.

6 Applications

In this section we show that our program logic is sufficiently expressive to be useful in practice for security applications. First, we show how access control decisions based on past events in a history-based model can be expressed and typechecked. Then, we solve an open problem by showing how Java's parameterized privileges can be statically modeled.

In our examples we will be interested in unparameterized events and predicates; in such cases we will write ev and ϕ for $ev(c_{\mathsf{dummy}})$ and $\phi(c_{\mathsf{dummy}})$ respectively, where c_{dummy} is a distinguished dummy constant. Also, we will abbreviate events ev_i by their subscripts i, and the notation:

$$(\vee \{ev_1(c_1), \ldots, ev_n(c_n)\})\phi \triangleq ev_1(c_1)\phi \vee \cdots \vee ev_n(c_n)\phi$$

will be convenient, as will:

$$(.*)\phi \triangleq ev_1(c_1)\phi \vee \cdots \vee ev_n(c_n)\phi$$

where $\{ev_1(c_1), \ldots, ev_n(c_n)\}$ for the latter definition is the set of events in a specified context.

6.1 History-based access control

History-based access control is a generalization of Java's notion of stack inspection that takes into account all past events, not just those on the stack [1]. Our language is perfectly suited for the static typechecking of such security policies. In the basic history model of [1], some initial *current rights* are given, and with every new activation the static rights of that activation are automatically intersected with the current rights to generate the new current rights. Unlike stack inspection, removal of the activation does not return the current rights to its state prior to the activation.

Before showing how this form of assertion can be expressed in our language, we define the underlying security model. We assume all code is annotated with a "principal" identifier p, and assume an ACL policy $\mathcal{A}$ mapping principals p to resources $r(c)$ for which they are authorized. An event ev_p is issued whenever a codebase annotated with p is entered. A demand of a resource r with parameter c, $\phi_{\text{demand},r}(c)$, requires that all invoked functions possess the right for that resource. This general check may be expressed in our language as $\phi_{\text{demand},r}$, defined in Fig. 12, where we assume given for verification history effect H containing events $ev_1(c_1), \ldots, ev_n(c_n)$.

Assertion $\phi_{\text{demand},r}(c)$ forces all code principals invoked thus far to have the rights for $r(c)$. For example, validity of the following requires $r(c) \in \mathcal{A}(p_1) \cap \mathcal{A}(p_2)$:

$$\Gamma, p_1; p_2; \phi_{\text{demand},r}(c) \vdash p_1; (\lambda x.p_2; \phi_{\text{demand},r}(x))\, c : \textit{unit}$$

The model in [1] also allows for a combination of stack- and history-based properties, by allowing the *amplification* of a right on the stack: it stays active even after function return. Such assertions can be expressed in our framework using a combination of stack- and history-based assertions.

6.2 Stack inspection with parameterized privileges

Java stack inspection [26, 20, 18] uses an underlying security model of principals and resources as defined in the previous section. One additional feature of Java is that any principal may also explicitly enable a resource for which they are authorized. When a function is activated, its associated principal identifier is pushed on the stack, along with any resource enablings that occur in its body. Stack inspection for a particular resource $r(c)$ then checks the stack for an enabling of $r(c)$, searching frames from most to least recent, and failing if a principal unauthorized for $r(c)$ is encountered before an enabling of $r(c)$, or if no such enabling is encountered.

Stack inspection can be modeled in the stack-based variant of our programming logic defined in Section 5. Rather than defining the general encoding, we develop one particular example which illustrates all the issues. Consider the following function checkit:

$$\text{checkit} \triangleq \lambda x.p\text{:system}; \phi_{\text{inspect},r:\text{filew}}(x)$$

Every function upon execution first issues an owner (principal) event, in this case p:system indicating "system" is the principal p that owns checkit. The function takes a parameter x (a file name) and inspects the stack for the "filew" resource with parameter x, via embedded μ-calculus assertion $\phi_{\text{inspect},r:\text{filew}}(x)$. This assertion is defined below; it enforces the fact that all functions on the call stack back to the nearest enable must be owned by principals p that according to ACL $\mathcal{A}$ are authorized for the r:filew(x) resource.

Now, to model resource enabling we use a special parameterized event $\text{enable}_r(x)$, indicating resource $r(x)$ is temporarily enabled. We illustrate use of explicit enabling via an example "wrapper" function enableit, owned by say the "accountant" principal

$$
\begin{aligned}
(p_{\neg r(c)})\phi &\triangleq (\vee\{p \mid r(c) \notin \mathcal{A}(p)\})\phi \\
\phi_{\text{demand},r}(c) &\triangleq \neg((.*)(p_{\neg r(c)})(.*)(\mathrm{Now})\textbf{true}) \\
\\
(\bar{p})\phi &\triangleq (\vee(\{ev_1(c_1),\ldots,ev_n(c_n)\} \setminus \mathrm{dom}(\mathcal{A})))\phi \\
(\neg ev(c))\phi &\triangleq (\vee(\{ev_1(c_1),\ldots,ev_n(c_n)\} \setminus \{ev(c)\}))\phi \\
(a*)\phi &\triangleq \boldsymbol{\mu}x.(a)x \vee \phi \qquad \text{for } a \in \{\bar{p}, \neg ev(c)\} \\
\phi_{\text{enable-ok,r}}(c) &\triangleq \neg((.*)(p_{\neg r(c)})(\bar{p}*)(\mathrm{enable}_r(c))\textbf{true}) \\
\phi_{\text{inspect-ok,r}}(c) &\triangleq \neg((\neg\mathrm{enable}_r(c)*)(\mathrm{Now})\textbf{true}) \wedge \neg((.*)(p_{\neg r(c)})(\neg\mathrm{enable}_r(c)*)(\mathrm{Now})\textbf{true}) \\
\phi_{\text{inspect},r}(c) &\triangleq \phi_{\text{enable-ok,r}}(c) \wedge \phi_{\text{inspect-ok,r}}(c)
\end{aligned}
$$

Fig. 12. Definitions of $\phi_{\text{demand},r}$ and $\phi_{\text{inspect},r}$

$p{:}\mathrm{acct}$, that takes a function f and a constant x, and enables $r{:}\mathrm{filew}(x)$ for the application of f to x:

$$\text{enableit} \triangleq \lambda f.p{:}\mathrm{acct}; (\lambda x.p{:}\mathrm{acct}; \mathrm{enable}_{r:\mathrm{filew}}(x); \text{let } y = f(x) \text{ in } y)$$

The definition of $\phi_{\text{inspect},r}(c)$, for fixed $r(c)$, is generalized over parameterized resources $r(c)$. For history effect H containing only the events $ev_1(c_1),\ldots,ev_n(c_n)$, and parameterized resource $r(c)$, Fig. 12 gives the definition of $\phi_{\text{inspect},r}(c)$. ϕ_{inspect} has two parts: first, any $r(c)$ enabling must be valid via $\phi_{\text{enable-ok,r}}(c)$; second, we must check that stack inspections for $r(c)$ are valid via $\phi_{\text{inspect-ok,r}}(c)$.

Returning to our previous example expressions checkit and enableit, the following most general types are inferred in our system:

$$
\begin{aligned}
\text{checkit} : \forall\alpha.\{\alpha\} &\xrightarrow{p:\mathrm{system};\phi_{\text{inspect},r:\mathrm{filew}}(\alpha)} \textit{unit} \\
\text{enableit} : \forall\alpha h t.(\{\alpha\} \xrightarrow{h} t) &\xrightarrow{p:\mathrm{acct}} \{\alpha\} \xrightarrow{p:\mathrm{acct};\mathrm{enable}_{r:\mathrm{filew}}(\alpha);h} t
\end{aligned}
$$

The stackification of the application (enableit checkit (/accts/ledger.txt)) will then generate the history effect $p{:}\mathrm{acct}|H$, where:

$$H = p{:}\mathrm{acct}; \mathrm{enable}_{r:\mathrm{filew}}(\text{/accts/ledger.txt}); p{:}\mathrm{system}; \phi_{\text{inspect},r:\mathrm{filew}}(\text{/accts/ledger.txt})$$

This reflects that the call and return of the application (enableit checkit) is assigned the effect $p{:}\mathrm{acct}$, while the subsequent application to /accts/ledger.txt is assigned effect H. Assuming that both $p{:}\mathrm{system}$ and $p{:}\mathrm{acct}$ are authorized for $r{:}\mathrm{filew}$(/accts/ledger.txt) in $\mathcal{A}$, verification will clearly succeed on this expression. On the other hand, stackification of the application checkit(/accts/ledger.txt) will generate the following history effect:

$$p{:}\mathrm{system}; \phi_{\text{inspect},r:\mathrm{filew}}(\text{/accts/ledger.txt})$$

for which verification will fail: there is no required $\mathrm{enable}_{r:\mathrm{filew}}$(/accts/ledger.txt) on the stack.

7 Conclusions

We have presented a new type effect system, an inference algorithm for inferring effects, and an algorithm for automatically verifying assertions made about the effects. With this system, users merely need to decorate code with logical assertions about past events, and the system will automatically verify those assertions without user intervention.

The original goal of our project was to build the first system to statically model the parameterized privileges of Java stack inspection, and also to allow the static checking of general runtime history assertions for enforcing access control policies. We believe we have succeeded in this regard. But, the system we have produced is not particularly biased toward access control properties, and thus will be useful in other domains: it can statically enforce a wide class of event-based runtime invariants.

The type system itself makes several contributions in its combination of expressiveness and completeness. The effect system conservatively extends effect-free typing, so the effects will not "get in the way" of typing. The inference algorithm merges history effect H in a sound and complete manner instead of attempting to unify them.

7.1 Related Work

Our type effect system shares much of the basic structure of the effect system in [2]; their system lacks our singletons and contextual assertions. Their approach has one significant drawback as compared to ours, in that their inference algorithm is not "sound enough"; it is formally proven sound, but the soundness property doesn't preclude inconsistent constraints such as $ev_{\mathrm{up}} = ev_{\mathrm{down}}$.

Perhaps the most closely related work is [15], which proposes a similar type and effect system and type inference algorithm, but their "resource usage" abstraction is of a markedly different character, based on grammars rather than LTSs. Their system lacks parametric polymorphism, which restricts expressiveness in practice, and verifies global, rather than local, assertions. Furthermore, their system analyzes only history-based properties, not stack-based properties as in our $stackify$ transformation.

The system of [13] is based on linear types, not effect types. Their usages U are similar to our history effects H, but the usages have a much more complex grammar, and appear to have no real gain in expressiveness. Their specification logic is left abstract, thus they provide no automated mechanism for expressing or deciding assertions. Our history effects can easily be seen to form an LTS for which model-checking is decidable; their usages are significantly more complex, so it is unclear if model-checking will be possible.

The systems in [8, 5, 14, 4] use LTSs extracted from control-flow graph abstractions to model-check program security properties expressed in temporal logic. Their approach is close in several respects, but we are primarily focused on the programming language as opposed to the model-checking side of the problem. Their analyses assume the pre-existence of a control-flow graph abstraction, which is in the format for a first-order program analysis only. Our type-based approach is defined directly at the language level, and type inference provides an explicit, scalable mechanism for extracting an abstract program interpretation, which is applicable to higher-order functions and other features. Furthermore, polymorphic effects are inferrable, and events may be parameterized by constants so partial dataflow information can be included.We believe our results are critical to bringing this general approach to practical fruition for production programming languages such as ML and Java.

Acknowledgments

Thanks to David Van Horn for his useful comments, implementations, and testing.

References

1. Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, feb 2003.
2. T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems*. Imperial College Press, 1999.
3. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
4. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. Computer Security*, 9:217–250, 2001.
5. Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In *Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 76–87. ACM Press, 2002.
6. O. Burkart, D. Caucal, F. Moller, , and B. Steffen. Verification on infinite structures. In S. Smolka J. Bergstra, A. Pons, editor, *Handbook on Process Algebra*. North-Holland, 2001.
7. Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, November 18–22, 2002.
8. Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2000.
9. Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, pages 38–48, 1998.
10. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. volume 1, 1995. `http://www.elsevier.nl/locate/entcs/volume1.html`.
11. J. Esparza. On the decidability of model checking for several mu-calculi and Petri nets. In *Proceeding of CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, 1994.
12. J. Esparza, A. Kucera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *TACS: 4th International Conference on Theoretical Aspects of Computer Software*, 2001.
13. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342, Portland, Oregon, January 2002.
14. T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.
15. P. J. Stuckey K. Marriott and M. Sulzmann. Resource usage verification. In *Proc. of First Asian Programming Languages Symposium, APLAS 2003*, 2003.

16. Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.
17. Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Uppsala, Sweden, August 2003.
18. François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001.
19. Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
20. Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, Montréal, Canada, September 2000.
21. Christian Skalka and Scott Smith. History types and verification. Extended manuscript, `http://www.cs.uvm.edu/˜skalka/skalka-smith-tr04.ps`, 2004.
22. B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR'92, Stony Brook (NY)*, volume 630 of *Lecture Notes in Computer Science (LNCS)*, pages 123–137, Heidelberg, Germany, 1992. Springer-Verlag.
23. Chris Stone. Singleton types and singleton kinds. Technical Report CMU-CS-00-153, Carnegie Mellon University, 2000.
24. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.
25. David Walker. A type system for expressive security policies. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, Boston, Massachusetts, January 2000.
26. Dan S. Wallach and Edward Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998.