History Types and Verification

Christian Skalka[1] and Scott Smith[2]

[1] The University of Vermont skalka@cs.uvm.edu
[2] The Johns Hopkins University, scott@cs.jhu.edu

Abstract. We develop a novel programming logic for verifying security properties, allowing *histories* of program execution to be taken into account: validity of access depends on the explicit sequence of past actions. The main result of the paper is a novel static analysis combining a type and effect inference system with a model-checking procedure, that guarantees validity of run-time history assertions. Among other examples, we illustrate the power of the framework by showing how the parameterized privileges of Java stack inspection may be statically verified, solving an open problem.

1 Introduction

Programming language-based security models for access control have a distinct advantage over other approaches: in a programming language, there is an ability to base access control decisions on fine-grained *contexts* of execution. Since secure programs execute in trusted runtimes, the order of events can be tracked without possibility of forgery. For example, stack-based access control [10, 16] associates authorization levels with regions of code, and ensures that when any privilege is checked, the sequence of callers on the call stack preceding the check are all authorized for that privilege. History-based access control is another example [1]: all execution events preceding the check must ensure authorization for the checked privilege. In this paper we develop a foundational programming logic of execution contexts, in two flavors: one for asserting properties of stack contexts, and one for asserting properties about the context of all past events.

1.1 History-Based Specification

Fundamental to our approach is the notion of atomic events that occur during program execution, comprising a dynamic *history*, a stream of events having occurred so far. In addition, at any point in a program, it is possible to assert a logical property of the sequence of events leading to that point. For example, right before reading a file, we could assert that the history must contain an "open" event for that file, or we could require that a "read" privilege for that file was added to the active security context. The system we define for this purpose is a programming logic, where logical assertions are embedded in the program, which are modal with respect to their history context. There is a rich tradition of modal assertions in programs, beginning with Hoare Logic, and the

modal aspect is made explicit in e.g. Dynamic Logic [5]. Our logic differs from these approaches in that it is not the present state of variables, but past events, and the order in which they occur, that defines the context of assertions.

These logical assertions are rooted in the program run-time. The main technical contribution of this work is a type effect system which verifies these assertions statically, ensuring safety of typable programs in this model. We also define a particular decidable assertion language based on the μ-calculus; combining this logic with our decidable type inference algorithm gives a completely automatic framework for verifying assertions in programs. Our theoretical approach is general, but our particular applications focus is security. In Sect. 8 we show how stack inspection with parameterized privileges can be encoded. We believe ours is the first system for statically enforcing stack inspection in the presence of parameterized privileges. Our work is related to static systems for specifying program state changes and resource usage patterns [4, 9, 15, 6], but these systems construct global specifications, not local, modal specifications. See Sect. 9 for a more thorough discussion of related work and our contributions.

1.2 The Technical Development

In the remaining text, we formalize our ideas in the system λ_{hist}. We first define an operational semantics of a basic language $\lambda^{c}_{\text{hist}}$, its operational semantics and type system. An extension $\lambda^{\zeta}_{\text{hist}}$ is then defined which supports static typing of dynamic constant generation. We next give a type inference algorithm for this latter system, and show how the μ-calculus can be integrated as a static verification logic, obtaining decidable assertion checking. Finally, a stack-history variant is defined that redefines the history as those events in active frames only; as an example stack-based assertion, we show how stack inspection can be expressed as a μ-calculus formula in this variant.

2 The Language λ_{hist}

In this section we formally define the core calculus of λ_{hist}, denoted $\lambda^{c}_{\text{hist}}$. In Sect. 3 a notion of dynamic constant is added; here we focus on the basic aspects of the system.

2.1 Syntax and Semantics

The syntax and semantics of the core theory $\lambda^{c}_{\text{hist}}$ are given in Fig. 1 and Fig. 2. We assume the syntactic sugar $\wedge e_1 e_2 \triangleq e_1 \wedge e_2$, and $\vee e_1 e_2 \triangleq e_1 \vee e_2$, and let $[\![e]\!]_{\text{bool}}$ denote the usual interpretation of boolean expressions. Functions, written $\lambda_z x.e$, possess a recursive binding mechanism where z is the self variable; we may write $\lambda x.e$ for $\lambda_z x.e$ if z is not free in e.

An event ev is a named entity parameterized by a constant 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

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

Fig. 1. $\lambda^{c}_{\text{hist}}$ language syntax

$$
\begin{array}{rll}
\eta, (\lambda_z x.e)v \rightarrow \eta, e[v/x][\lambda_z x.e/z] & & (\beta) \\
\eta, \neg b \rightarrow \eta, [\![\neg b]\!]_{\text{bool}} & & (not) \\
\eta, b_1 \wedge b_2 \rightarrow \eta, [\![b_1 \wedge b_2]\!]_{\text{bool}} & & (and) \\
\eta, b_1 \vee b_2 \rightarrow \eta, [\![b_1 \vee b_2]\!]_{\text{bool}} & & (or) \\
\eta, \mathsf{if\,true\,then}\, e_1\, \mathsf{else}\, e_2 \rightarrow \eta, e_1 & & (\textit{if1}) \\
\eta, \mathsf{if\,false\,then}\, e_1\, \mathsf{else}\, e_2 \rightarrow \eta, e_2 & & (\textit{if2}) \\
\eta, \mathsf{let}\, x = v\, \mathsf{in}\, e \rightarrow \eta, e[v/x] & & (let) \\
\eta, ev(c) \rightarrow \eta; ev(c), () & & (event) \\
\eta, \phi(c) \rightarrow \eta; ev_\phi(c), () & \text{if } (\Pi(\phi)(c))(\eta; ev_\phi(c)) & (check) \\
\eta, E[e] \rightarrow \eta', E[e'] & \text{if } \eta, e \rightarrow \eta', e' & (context)
\end{array}
$$

Fig. 2. $\lambda^{c}_{\text{hist}}$ language semantics

strings or IP addresses. Ordered sequences of these events constitute histories η, which maintain the sequence of events experienced during program execution. History assertions $\phi(c)$, also parameterized by a constant c, may be used to assert desirable properties of histories. These assertions are in a to-be-specified logical syntax (one example syntax is given in Sect. 5). We presuppose existence of a meaning function $\Pi(\phi)$ such that $(\Pi(\phi)(c))(\eta)$ holds iff $\phi(c)$ is valid for history η. For each formula ϕ there is a corresponding event ev_ϕ, issued at the point ϕ is checked.

The operational semantics is defined as a call-by-value small step reduction relation $\rightarrow$ 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, while in the *check* rule, the predicate $\phi(c)$ is required to hold on the current program history, according to our meaning function Π. In case the predicate fails, execution is stuck.

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

Fig. 3. λ^c_{hist} language type syntax

For example, assuming the addition of the usual syntactic sugar for sequencing of expressions, consider the following function f:

$$f \triangleq \lambda_z x.\mathsf{if}\, x\, \mathsf{then}\, \mathit{ev}_1(c)\, \mathsf{else}\, (\mathit{ev}_2(c); z(\mathsf{true}))$$

In the operational semantics, we have:

$$\epsilon, f(\mathsf{false}) \rightarrow^\star \mathit{ev}_2(c); \mathit{ev}_1(c), ()$$

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.

2.2 Logical Type System

In the type analysis, we are challenged to statically identify histories that result during execution, for which purpose we introduce *history types* H. In essence, any H conservatively approximates the history η generated by an expression during execution, by representing a set of possible histories containing at least η. A history type may therefore be an event $\mathit{ev}(c)$, or a sequencing of history types $H_1; H_2$, a nondeterministic choice of history types $H_1|H_2$, or a μ-bound history type $\mu h.H$ which finitely represents the set of histories that may be generated by a recursive function. History types may contain predicate events $\mathit{ev}_\phi(c)$, allowing us to verify predicate checks at the right points in history approximations.

The syntax of types for λ^c_{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 [8, 14, 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 $\{c\}$ [13], where only atomic constants can have singleton type. Types contain three sorts of variables; regular type variables t, singleton type variables α, and history type variables h. We let ψ range over variable substitutions, heterogeneous in these sorts, with the restriction that substitutions must be sort-consistent—*i.e.*, only history types may be substituted for history type variables, etc.

Type derivation rules for judgements of the form $\Gamma, H \vdash e : \tau$ are given in Fig. 4, where Γ is an environment of variable typing assumptions. Intuitively,

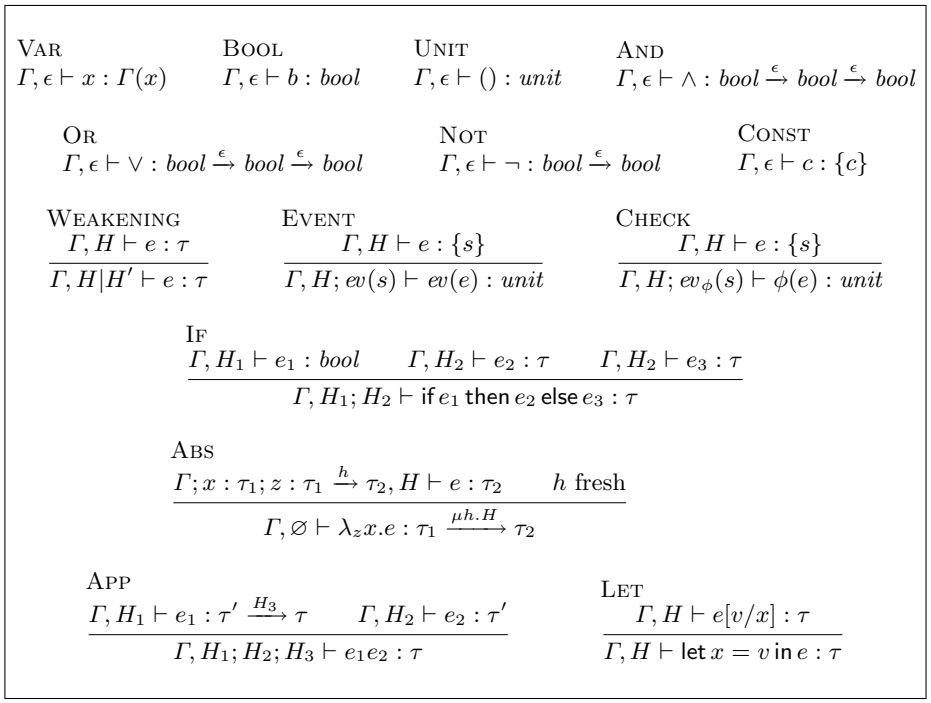

Fig. 4. $\lambda^{c}_{\text{hist}}$ logical typing rules

the history type H in judgements represents the set of histories that may arise during execution of e; this intuition is formalized in Corollary 2.7. For example, with f defined as in Sect. 2.1, the following judgements are derivable:

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

We include let-polymorphism in the form of a let-expansion typing rule. This approach, while less efficient in practice, significantly simplifies this presentation. 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. We note 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-free type system may be replayed here.

2.3 Interpretation of History Types

As alluded to previously, the interpretation of a history type is, roughly, a set of histories. More accurately, we define the interpretation of history types as sets of *traces*, which include a $\downarrow$ symbol to denote termination. Traces may be infinite– appropriately, since we analyze programs which may not terminate.

Definition 2.1. *Our interpretation of histories will be defined via strings (called* traces*), denoted θ, over the following alphabet:*

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

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

Definition 2.2. *The history 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 associated with a closed history type in terms of the transition relation:

Definition 2.3. *The interpretation of histories is defined as follows:*

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

The question $[\![H_1]\!] = [\![H_2]\!]$ is in fact undecidable: histories are a generalization of BPA's (basic process algebras), and their trace equivalence is known to be undecidable [3].

The validity of a history is then based on the 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. The relevant definitions are as follows:

Definition 2.4. *We say that a history H is* valid *iff for all $a_1 \cdots a_n ev_\phi(c)$ in $[\![H]\!]$ it is the case that $\Pi(\phi(c)(a_1 \cdots a_n))$ holds. A type judgement $\Gamma, H \vdash e : \tau$ is* valid *iff it is derivable and H is valid.*

We now observe several significant properties of the type system defined above. Since failure of predicate checks at run-time results in stuck expressions, soundness of our analysis can be stated via type safety (Theorem 2.6) and progress (Theorem 2.5) results. The formalization of our basic intuition about history types, that they approximate run-time histories, falls out as a corollary of the Lemmas preceding progress and type safety, so we also state that corollary here as a fundamental property of the system (Corollary 2.7). Proofs are omitted for lack of space.

Theorem 2.5 (Progress). *If $\Gamma, H \vdash e : \tau$ is derivable and η, e is irreducible with $\eta; H$ valid, then e is a value.*

Theorem 2.6 (History Type Safety). *Well-typed expressions don't go wrong in λ^c_{hist}.*

Corollary 2.7. *If $\Gamma, H \vdash e : \tau$ and $\epsilon, e \rightarrow^\star \eta, v$ then $\eta \in [\![H]\!]$.*

3 Dynamic Constants

So far, we have considered events and predicates parameterized by statically declared constants. While this is useful, it falls short of capturing *dynamically* generated constants, which are common in programming languages, and are often important elements of access control models in particular. For example, a common property of previous language systems is the guarantee that files are open before they are read. We can enforce this as follows; first, given a function *open* (resp. *close*) that opens (resp. closes) files, define a function *open′* (resp. *close′*) that appends an ev_{open} (resp. ev_{close}) event to the history:

$$open' \triangleq \lambda x.ev_{\mathsf{open}}(x); open(x) \qquad close' \triangleq \lambda x.ev_{\mathsf{close}}(x); close(x)$$

Then, define a predicate ϕ_{open} that checks for the appropriate property: $\phi_{\text{open}}(c)(\eta)$ iff there is an $ev_{\mathsf{open}}(c)$ on η with no $ev_{\mathsf{close}}(c)$ following it. We may then define a *read′* function that only *read*s open files:

$$read' \triangleq \lambda x.\phi_{\text{open}}(x); read(x)$$

However, our basic calculus is inadequate for a realistic model, in that new files may be dynamically allocated in practice, while constants c_f in the basic system must be statically declared in the basic calculus. There are numerous other examples from practice, where dynamically generated program entities are relevant to the program security analysis. Thus, we introduce extensions to the basic language semantics and type system for dynamic generation of constants, yielding the λ_{hist} variant $\lambda^{\zeta}_{\text{hist}}$. Generative constants are also developed in [9], which inspired us to include them; our rules are in a novel format which we believe is simpler and more direct than the ones in the aforementioned paper.

3.1 Language Extensions

To obtain dynamic constants in $\lambda^{\zeta}_{\text{hist}}$, we make a simple extension to the syntax and operational semantics of $\lambda^{c}_{\text{hist}}$. Specifically, we extend the language of expressions e with a construct for dynamically generating fresh constants, $\mathsf{new}\ x\ \mathsf{in}\ e$. This expression form is equipped with the following semantics:

$$\eta, \mathsf{new}\ x\ \mathsf{in}\ e \rightarrow \eta, e[c/x] c \text{ fresh} \qquad (gen)$$

3.2 Logical Typing Extensions

Now we extend the $\lambda^{c}_{\text{hist}}$ type system to analyze dynamic constants in $\lambda^{\zeta}_{\text{hist}}$. We differentiate between types of declared constants and dynamically generated constants, via the use of a new sort of singleton variables $\zeta \in \mathcal{V}_{\zeta}$, where $\mathcal{V}_{\zeta}$ is denumerable and disjoint from the variable sets defined previously: the singletons s are either variables α, fresh constants ζ, or fixed constants c. In the type analysis, any dynamically generated constant will be assigned a $\{\zeta\}$ type. Since functions may generate fresh constants within their scope with the $\mathsf{new}\ x\ \mathsf{in}\ e$

$$\textsc{New}\ \frac{\Gamma; x : \{\zeta\}, H \vdash e : \tau \quad \zeta \text{ fresh}}{\Gamma, H \vdash \mathsf{new}\ x\ \mathsf{in}\ e : \tau} \qquad \textsc{Abs2}\ \frac{\Gamma; x : \tau_1, H \vdash e : \tau_2 \quad \bar{\zeta} \cap \mathrm{fv}_\zeta(\Gamma) = \varnothing}{\Gamma, \epsilon \vdash \lambda x.e : \nabla\bar{\zeta}.\tau_1 \xrightarrow{H} \tau_2}$$

$$\textsc{Abs1}\ \frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2 \quad \bar{\zeta} \cap \mathrm{fv}_\zeta(\Gamma) = \varnothing \quad \mathrm{fv}_\zeta(\tau_1, \tau_2) - \mathrm{fv}_\zeta(\Gamma) = \varnothing}{\Gamma, \varnothing \vdash \lambda_z x.e : \tau_1 \xrightarrow{\mu h.\nabla\bar{\zeta}.H} \tau_2 \quad h \text{ fresh}}$$

$$\textsc{App}\ \frac{\Gamma, H_1 \vdash e_1 : \nabla\bar{\zeta}.\tau' \xrightarrow{H_3} \tau \quad \Gamma, H_2 \vdash e_2 : \tau'}{\Gamma, H_1; H_2; H_3[\bar{\zeta}'/\bar{\zeta}] \vdash e_1 e_2 : \tau[\bar{\zeta}'/\bar{\zeta}] \quad \bar{\zeta}' \text{ fresh}}$$

Fig. 5. Logical type rules for generative constants in $\lambda^{\zeta}_{\mathrm{hist}}$

construct, we redefine functions types as $\nabla\bar{\zeta}.\tau_1 \xrightarrow{H} \tau_2$. Here, $\bar{\zeta}$ may occur free in τ_1, H, or τ_2; we write $\mathrm{fv}_\zeta(\tau)$ to denote the set of free variables of this sort in τ. The vector $\bar{\zeta}$ specifies the types of constants that will be freshly generated by application of the function, and will themselves be freshly instantiated in typings. For brevity, we write $\tau \xrightarrow{H} \tau'$ for $\nabla\varnothing.\tau \xrightarrow{H} \tau'$.

We also extend the language of history types with a similar binding construct, $\nabla\bar{\zeta}.H$. We use this form of binding in history annotations to yield greater flexibility in the type system, an issue that is discussed in more detail below. The history transition relation is extended to accommodate this new form:

$$\nabla\bar{\zeta}.H \xrightarrow{\epsilon} H[\bar{c}/\bar{\zeta}] \qquad \bar{c} \text{ fresh}$$

Interpretation of histories, $[\![H]\!]$, is defined as before. The necessary updates to the logical type derivation rules are given in Fig. 5. We must restrict ∇-binding for recursive function types in Abs1 to gain a sound rule: with this weaker rule, a recursive function cannot itself return a singleton type $\{\zeta\}$ for a constant ζ generated by the function. The ζ can occur in H, however, and so a bound $\mu h.\nabla\bar{\zeta}.H$ is placed on H. For nonrecursive functions, the more general Abs1 rule may be used, which allows the return type to be a fresh $\{\zeta\}$. Note that the App rule yields fresh $\bar{\zeta}$, ensuring the type system tracks "newness" when new-constant-generating functions are used.

Returning to our previous example, and assuming the same definition of *open′*, we can now imagine a function *openfresh* that dynamically generates a new file, and opens it; the type of this function will incorporate ζ variables to represent the constant freshly generated in its scope:

$$\mathit{openfresh} \triangleq \lambda\_.\mathsf{new}\ x\ \mathsf{in}\ \mathit{open}'(x) : \nabla\zeta.\mathit{unit} \xrightarrow{ev_{\mathsf{open}}(\zeta)} \{\zeta\}$$

The application rule then ensures that fresh ζ variables are generated at every application point of *openfresh*. For example, a function that uses *openfresh* twice

within its scope will be assigned a history type that reflects the generation of two fresh constants:

$$f \triangleq \lambda\_.\mathit{openfresh}(); \mathit{openfresh}() : \nabla\zeta_1\zeta_2.\mathit{unit} \xrightarrow{ev_{\mathsf{open}}(\zeta_1);ev_{\mathsf{open}}(\zeta_2)} \{\zeta_2\}$$

Note also that the return type of f correctly distinguishes the returned fresh constant.

4 Type Inference

We define a type inference algorithm for the more expressive generative constant theory $\lambda^{\zeta}_{\text{hist}}$. Selected type inference rules are given in Fig. 6. The type inference system is a descendent of Milner's algorithm W, presented in the form of a ruleset directly invoking the unification algorithm U_τ defined below.

The rules are generally the obvious variations on the logical typing rules of Figure 5; one exception is App/AppLet: there are two application rules, the former is always applicable but unification will fail for the case the function returns a fresh ζ, i.e. if the return type is $\{\zeta\}$. If a function has been concretely defined (say via let), the actual type will be present at application and the more accurate AppLet may be used which allows for the case of functions returning freshly generated constants. The weaker App still allows functions to generate fresh constants, since h there can unify to e.g. $h = \nabla\zeta.ev(\zeta)$, but this ζ is bound in the history alone and thus cannot occur in the return type. We don't consider this restriction that significant in practice, but it is the (only) source of incompleteness; we show below that the ∇-free theory is complete.

4.1 Type Unification

The type unification algorithm is given in Figure 7; functions U_τ, U_Γ and M_H are defined to unify types, type environments, and histories, and are implicitly symmetric. It assumes each variable ζ in a ∇-bound in the original derivation is unique and distinct from any free ζ. And, we assume there a predicate $\mathit{bound}(\zeta)$ that holds iff ζ is bound and not free. The substitution ψ produced, when applied, assumes ∇ does not bind $\bar{\zeta}$.

For histories, we take advantage of the $|$ operator and simply merge them to make a disjunction of two histories. This merge is semantically sound and complete; unification of histories is inappropriate because histories may be structurally different but of equivalent meaning.

Lemma 4.1 (Soundness of type inference). *Suppose $\Gamma, H \vdash_W e : \tau$ then, $\Gamma, H \vdash e : \tau$ is derivable.*

The inference algorithm is not complete, but in only one respect: functions returning fresh constants in their return types, $\{\zeta\}$, cannot be used in a fully higher-order manner. For ζ-free $\lambda^{c}_{\text{hist}}$, we now show a principal type property and thus completeness. The histories produced in type derivations conform to those

$$\textsc{Abs1}\quad \frac{\Gamma; x:\tau_0'; z:\tau_0'' \xrightarrow{h} \tau_0, H \vdash_W e:\tau \qquad \bar{\zeta} \cap \mathrm{fv}_\zeta(\psi(\Gamma)) = \varnothing \qquad (\tau', \psi_1) = U_\tau(\tau_0', \tau_0'') \qquad (\tau', \psi) = U_\tau(\psi_1(\tau_0), \psi(\tau)) \qquad \mathrm{fv}_\zeta(\psi(\tau)) - \mathrm{fv}_\zeta(\psi(\Gamma)) = \varnothing}{\Gamma, \epsilon \vdash_W \lambda_z x.e : \tau_0' \xrightarrow{\mu h.\nabla\bar{\zeta}.H} \tau'}$$

$$\textsc{Abs2}\quad \frac{\Gamma; x:\tau, H \vdash_W e:\tau' \qquad \bar{\zeta} \cap \mathrm{fv}_\zeta(\Gamma) = \varnothing}{\Gamma, \epsilon \vdash_W \lambda x.e : \nabla\bar{\zeta}.\tau \xrightarrow{H} \tau'}$$

$$\textsc{App}\quad \frac{t, h \text{ fresh} \qquad \Gamma_1, H_1 \vdash_W e_1 : \tau_1 \qquad \Gamma_2, H_2 \vdash_W e_2 : \tau_2 \qquad (\tau, \psi_1) = U_\tau(\tau_1, \tau_2 \xrightarrow{h} t) \qquad (\Gamma, \psi) = U_\Gamma(\psi_1(\Gamma_1), \psi_1(\Gamma_2))}{\Gamma, \psi(H_1; H_2; h) \vdash_W e_1\ e_2 : \psi(t)}$$

$$\textsc{AppLet}\quad \frac{\bar{\zeta}' \text{ fresh} \qquad \Gamma_1, H_1 \vdash_W e_1 : \nabla\bar{\zeta}.\tau' \xrightarrow{H_3} \tau \qquad \Gamma_2, H_2 \vdash_W e_2 : \tau'' \qquad (\tau''', \psi) = U_\tau(\tau', \tau'') \qquad (\Gamma, \psi) = U_\Gamma(\psi_1(\Gamma_1), \psi_1(\Gamma_2))}{\Gamma, \psi(H_1; H_2; H_3)[\bar{\zeta}'/\bar{\zeta}] \vdash_W e_1\ e_2 : \psi(\tau)[\bar{\zeta}'/\bar{\zeta}]}$$

Fig. 6. Selected type inference rules for $\lambda^{\zeta}_{\text{hist}}$

inferred, with the exception that the weakening rule could have been used in a derivation and not used in inference (in inference, weakening is used only at conditionals). We define a relation $H \leq_{\text{weak}} H'$ indicating H' can be obtained from H through a series of subterm replacements of H_0 with $(H_0|H_0')$ for arbitrary H_0'.

Lemma 4.2 (Completeness of $\lambda^{c}_{\text{hist}}$ type inference). *If $\Gamma, H \vdash e : \tau$ in $\lambda^{c}_{\text{hist}}$, then $\Gamma', H' \vdash_W e : \tau'$ and for some ψ, $\psi(\Gamma') = \Gamma$, $\psi(H') \leq_{\text{weak}} H$, and $\psi(\tau') = \tau$.*

Here is an example of history merging:

$$hoif = \lambda g_1 \lambda g_2.\, \mathsf{if}\ \ldots\ \mathsf{then}\ (\lambda x.ev_1; g_1())\ \mathsf{else}\ (\lambda x.ev_2; g_2())$$

The type inferred for this program is

$$\forall h_1 h_2 t_1 t_2 t.(t_1 \xrightarrow{h_1} t) \xrightarrow{\epsilon} (t_2 \xrightarrow{h_2} t) \xrightarrow{(ev_1;h_1)|(ev_2;h_2)} t$$

—there is no need to force the then- and else-histories to match, since their disjunction can be formed.

$$
\begin{array}{l}
U_\tau(\tau,\tau) = (\tau,\emptyset) \\
U_\tau(t,\tau) = (\tau,\{t \mapsto \tau\}) \\
U_\tau(\nabla\bar{\zeta}.\tau_1 \xrightarrow{H} \tau_2, \nabla\bar{\zeta}'.\tau_1' \xrightarrow{H'} \tau_2') = \\
\quad \text{let } (\tau_1'',\psi_1) = U_\tau(\tau_1,\tau_1') \text{ in} \\
\qquad \text{let } (\tau_2'',\psi_2) = U_\tau(\psi_1(\tau_2),\psi_1(\tau_2')) \\
\qquad \text{in } (\tau_1'' \xrightarrow{M_H(H,H'[\bar{\zeta}/\bar{\zeta}'])} \tau_2'',\psi_2) \\
U_\tau(\{s\},\{s'\}) = \\
\quad \text{if } s = s' \text{ then } (\{s\},\emptyset) \\
\quad \text{else if } s = \alpha \text{ then } (\{s'\},\{\alpha \mapsto \{s'\}\}) \\
\quad \text{else if } s' = \alpha \text{ then } (\{s\},\{\alpha \mapsto \{s\}\}) \\
\quad \text{else fail}
\end{array}
\qquad
\begin{array}{l}
U_\Gamma(\emptyset,\Gamma) = (\Gamma,\emptyset) \\
U_\Gamma(x:\tau_1;\Gamma_1,\ x:\tau_2;\Gamma_2) = \\
\quad \text{let } (\tau,\psi) = U_\tau(\tau_1,\tau_2) \text{ in} \\
\quad \text{let } (\Gamma,\psi') = U_\Gamma(\psi(\Gamma_1),\psi(\Gamma_2)) \text{ in} \\
\qquad (x:\tau;\Gamma,\ \psi') \\
U_\Gamma(x:\tau_1;\Gamma_1,\ \Gamma_2) \text{ for } x \notin \mathrm{dom}(\Gamma_2) = \\
\quad \text{let } (\Gamma,\psi') = U_\Gamma(\psi(\Gamma_1),\psi(\Gamma_2)) \text{ in} \\
\qquad (x:\tau_1;\Gamma,\ \psi') \\
\\
M_H(h,H) = (H,\{h \mapsto H\}) \\
M_H(H_1,H_2) = (H_1 \mid H_2,\emptyset)
\end{array}
$$

Fig. 7. Type unification algorithm

5 A Verification Framework

In Definition 2.4, validity of any history type H has been specified in logical form, with respect to the set of trace prefixes in $[\![H]\!]$. However, this definition is clearly insufficient as a decision procedure, since $[\![H]\!]$ may be infinite. Thus, to algorithmically verify history types, we must supply some procedure which is sound with respect to this definition of validity. To this end, we will take a model-checking approach, via branching-time logic, in keeping with our interpretation of history types as branching-time labeled transition systems (LTS).

While a plethora of model-checking logics are available, we use the modal μ-calculus [3] because it is syntactically close to histories H. Further, efficient techniques for the automated verification of μ-calculus formulas on BPA processes (also known as context-free processes) have been developed [3, 12]; history types in fact correspond to BPA processes, and so these decision procedures may be directly applied in our framework. We lack the space here to prove the correspondence of BPA processes α and history types H, but their syntax and semantics are virtually identical; both denote process algebras lacking parallelism and synchronization. The $\nabla\zeta.H$ histories are the one form that have no BPA analogue, and so the results here apply to non-recursive $\nabla\zeta.H$ only, which can without loss of generality be replaced by $H[c/\zeta]$ for fresh c. The appropriate generalization of our model-checking theory to fully handle $\nabla\zeta.H$ is future work.

The syntax of the μ-calculus is:

$$\phi ::= x \mid \mathbf{true} \mid \mathbf{false} \mid \langle a\rangle\phi \mid [a]\phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \boldsymbol{\mu} x.\phi \mid \boldsymbol{\nu} x.\phi$$

We use the so-called standard interpretation of μ-calculus formulae [12]: a relation $\alpha \Vdash \phi$ meaning ϕ holds in a BPA state α. For brevity we omit the details of this interpretation . In our interpretation, BPA processes are replaced by histories H and so the relation is $H \Vdash \phi$. From [12], we have the fact that this relation is decidable.

We restrict our attention to formulae which have consistent meanings both at "compile time" and at "run time". This is a subtle issue, because history types are branching time, and run-time histories are linear-time, but we want our assertions to be the same statically and dynamically. The desirable *linearizable* ϕ are those for which branching-time truth implies linear-time truth: if a fact holds during typechecking, it will hold at runtime.

Definition 5.1. *We say that ϕ is* linearizable *iff it contains no positive occurrences of subformulae $\langle a\rangle\phi'$ or $\nu x.\phi'$ in ϕ.*

Both of the non-linearizable forms have "exists a path" semantics, and thus branching-time truths may not be linear-time truths. Any all-paths assertion can be expressed in a linearizable ϕ, and so the logic is very expressive, as will be shown from examples.

Lemma 5.2 (Linearization). *If ϕ is linearizable and $\eta \in [\![H]\!]$, then $H \Vdash \phi$ implies $\eta \Vdash \phi$.*

Note we are implicitly coercing run-time history η to its corresponding linear compile-time history here. This Lemma directly follows by induction on the standard semantics of $H \Vdash \phi$.

The framework of Section 2 is officially instantiated to use the μ-calculus as its logic by defining the abstract formulae $\phi(c)$ there to range over linearizable μ-calculus formulae ϕ parameterized by c. These ϕ can in addition contain new modal formulae $[\text{Now}]\phi'$ and $\langle\text{Now}\rangle\phi'$ that mark "now", the point in execution at which the formula is being checked for validity. These new formulae are just shorthand: in fixed formula ϕ, [Now] (resp. $\langle\text{Now}\rangle$) hereafter is assumed to be replaced by $[ev_\phi]$ (resp. $\langle ev_\phi\rangle$). We then may define Π as $(\Pi(\phi)(c))(\eta) \Leftrightarrow \eta \Vdash \phi(c)$.

For any assertion $\phi(c)$ and history type H containing only the events $ev_1(c_1), \ldots, ev_n(c_n)$, define:

$$\langle . \rangle \phi \triangleq \langle ev_1(c_1)\rangle\phi \vee \cdots \vee \langle ev_n(c_n)\rangle\phi$$
$$\text{occurs}(\phi(c)) \triangleq \boldsymbol{\mu} x.\langle . \rangle x \vee \langle ev_\phi(c)\rangle \mathbf{true}$$

Putting together Definition 2.4 and Lemma 5.2, we may directly obtain:

Corollary 5.3 (Logical Soundness). *If $H \Vdash \phi(c) \vee \neg occurs(ev_\phi(c))$ for all $\phi(c)$ occurring in e, then H is valid.*

Thus, combining this with Theorem 2.6 and Corollary 2.7, we know that any program for which a type is inferred and embedded assertions model-checked will have all assertions succeed at run-time.

6 A Stack-based Variation

In this section we sketch a variation on the framework of the previous sections that defines a stack-based framework instead of a history-based framework. In

this variation, instead of keeping track of *all* events, only events for functions on the current call stack are kept. Assertions ϕ are then assertions at run-time about the *active* event sequence, not all events. The changes to the previous sections are mostly minor, and for brevity we only summarize the differences.

We will use e.g. η^s to refer to the stack-based variant of a previous definition, in this case of a history η.

- The operational semantics rules are identical except the (now stack) η^s is checkpointed at function call, and restored to the checkpointed value at function return.
- The type rules are identical in the core theory; in the generative constant theory, Abs2 is modified so the resulting function has history effect $\mu h_d.H$ for a dummy variable h_d,and not just H. With this change, μ will serve to demarcate the scope of all functions.
- The history types H still represent histories, and not stacks. But, there is a simple translation possible to give a stack view of H, removing all inactive functions from H. This function is called $\mathit{stackify}(H)$ and is defined below.
- Then, $[\![H]\!]^s$ is defined to be $[\![\mathit{stackify}(H)]\!]$.
- Using these definitions, the key results (Theorems 2.5 and 2.6, Corollary 2.7, Lemma 5.2, and Corollary 5.3) can be replayed in the stack-based variant.

The key technical modification is the definition of *stackify*:

Definition 6.1. *The stackify algorithm is defined inductively as follows.*

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

(This stackify algorithm is not quite optimal; since the optimal definition is less intuitive we give a simpler conservative one here.)

Observe that the range of *stackify* consists of history types that are all tail-recursive; thus stacks are finite-state transition systems and more efficient model-checking algorithms are possible for stacks than for general histories.

An example of stackification, for a, b, c, d representing arbitrary events, is as follows:

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

7 Application: History-based access control

History-based access control is a generalization of Java's notion of stack inspection to take 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 the 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 annoted with p is entered. A demand of a resource r with parameter c, $\phi_{\text{demand},r}(c)$, requires that all invoked functions posess the right for that resource. This general check may be expressed in our language as follows (we will abbreviate events ev_i by their subscripts):

$$\phi_{\text{demand},r}(c) \triangleq \neg(\boldsymbol{\mu} x_1.\langle . \rangle x_1(\boldsymbol{\mu} x_2.\langle p_{\neg r(c)} \rangle x_2 \vee \langle \text{Now} \rangle \mathbf{true}))$$

where, given parameterized resource $r(c)$ and access control list $\mathcal{A}$ specifying that principals $p_1, ..., p_j$ are *not* authorized for $r(c)$, we define

$$\langle p_{\neg r(c)} \rangle \phi \triangleq \langle p_1 \rangle \phi \vee \cdots \vee \langle p_j \rangle \phi$$

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 $r(c) \in \mathcal{A}(p_1) \cap \mathcal{A}(p_2)$:

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

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.

8 Application: Stack inspection

Java stack inspection [16, 11, 10] 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 the enabling of $r(c)$, searching frames from most to least recent, and failing if a principal unauthorized for $r(c)$ is encountered before the enabling.

Stack inspection can be modeled in the stack-based variant of our programming logic defined in Section 6. Rather than defining the general encoding, we develop one particular example which illustrates all the issues. Consider the following function checkit:

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

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 $\phi_{\text{inspect},\mathbf{r:filew}}(x)$. 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 $\mathbf{r:filew}(x)$ resource.

Now, to model resource enabling we use a special parameterized event $\mathbf{enable}_r(x)$, 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 **p:acct**, that takes a function f and a constant x, and enables $\mathbf{r:filew}(x)$ for the application of f to x:

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

The definition of $\phi_{\text{inspect},r}(c)$, for fixed $r(c)$, is generalized over parameterized resources $r(c)$. For history type H containing only the events $ev_1(c_1), \ldots, ev_n(c_n)$, and parameterized resource $r(c)$, define the following:

$$\{ev'_1(c'_1), \ldots, ev'_m(c'_m)\} = \{ev_1(c_1), \ldots, ev_n(c_n)\} \setminus \text{dom}(\mathcal{A})$$

We also define the following μ-calculus formula abbreviations:

$$\langle \neg ev_i(c_i) \rangle \phi \triangleq \langle ev_1(c_1) \rangle \phi \vee \cdots \vee \langle ev_{i-1}(c_{i-1}) \rangle \phi \vee \langle ev_{i+1}(c_{i+1}) \rangle \phi \vee \cdots \vee \langle ev_n(c_n) \rangle \phi$$
$$\langle \bar{p} \rangle \phi \triangleq \langle ev'_1(c'_1) \rangle \phi \vee \cdots \vee \langle ev'_m(c'_m) \rangle \phi$$

Then, ϕ_{inspect} has two parts: first, any $r(c)$ enabling is valid:

$$\phi_{\text{enable-ok,r}}(c) \triangleq \neg(\boldsymbol{\mu} x_1.\langle . \rangle x_1 \vee \langle p_{\neg r(c)} \rangle (\boldsymbol{\mu} x_2.\langle \bar{p} \rangle x_2 \vee \langle \mathbf{enable}_r(c) \rangle \mathbf{true}))$$

And, we must check that stack inspections for $r(c)$ are valid with the following μ-calculus formula:

$$\phi_{\text{inspect-ok,r}}(c) \triangleq \neg(\boldsymbol{\mu} x_1.(\langle . \rangle x_1 \vee \langle p_{\neg r(c)} \rangle \boldsymbol{\mu} x_2.(\langle \neg \mathbf{enable}_r(c) \rangle x_2 \vee \langle \text{Now} \rangle \mathbf{true})))$$

$\phi_{\text{inspect},r}(c)$ is then defined as $\phi_{\text{enable-ok,r}}(c) \wedge \phi_{\text{inspect-ok,r}}(c)$. Observe this formula is linearizable.

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

$$\text{checkit} : \forall \alpha.\{\alpha\} \xrightarrow{\mathbf{p:system};\mathbf{inspect}_{\mathbf{r:filew}}(\alpha)} \mathit{unit}$$
$$\text{enableit} : \forall \alpha h t.(\{\alpha\} \xrightarrow{h} t) \xrightarrow{\mathbf{p:acct}} \{\alpha\} \xrightarrow{\mathbf{p:acct};\mathbf{enable}_{\mathbf{r:filew}}(\alpha);h} t$$

The stackification of the application (enableit checkit (/accts/ledger.txt)) will then generate the following history:

$$\mathbf{p:acct}; \mathbf{enable}_{\mathbf{r:filew}}(\text{/accts/ledger.txt}); \mathbf{p:system}; \mathbf{inspect}_{\mathbf{r:filew}}(\text{/accts/ledger.txt})$$

Assuming that both **p:system** and **p:acct** are authorized for **r: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:

$$\mathbf{p:system}; \mathbf{inspect}_{\mathbf{r:filew}}(/\text{accts/ledger.txt})$$

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

9 Contributions and Related Work

We have presented a new type effect system, an inference algorithm for the system that is (nearly) complete, and an algorithm for automatically verifying assertions made about the effects. With this system, users merely need to decorate code with 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 security policies. We believe we have succeeded in this regard. But, the system we have produced is not particularly biased toward security properties, and thus may be useful in other domains (*e.g.*, debugging).

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. We give a new system for typing generative constants ζ that is simple but powerful. The inference algorithm merges histories H in a lossless manner instead of attempting to (partially) unify them.

Our system supports general model checking properties of inferred effect types. Related systems either lack automatic assertion checking [6], or allow checking of only a limited range of properties [7]. One technical hurdle we had to overcome was how the branching-time nature of the effects H can be reconciled with the linear nature of events η at run-time; the *linearizable* formulae bridge this gap.

In order to elegantly express stack inspection we needed to use a notion of contextual assertion; we believe this form of assertion will be useful in other contexts, side-by-side with global assertions. The systems that support global assertions alone lack the ability to focus an assertion on a particular program point, and thus assertions must be more complex. Our framework can in fact express global assertions: if $H \Vdash \phi$ for ϕ not using the contextual $\langle\text{Now}\rangle/[\text{Now}]$, a global property ϕ holds.

9.1 Related Work

While there are several related type systems, none of them directly solve the problems we are interested in, such as the static typing of parameterized privileges in Java stack inspection. None of the systems incorporate our contextual

form of assertion, and none are integrated with a decision prodecure for automatic verification of assertions. The two systems which are technically the closest are [2, 6].

Our type effect system is related to the formulation in [2]; their system lacks our singletons, generative constants, and contextual assertions, and includes a theory of atomic subtyping which we do not. Their effects have a grammar remarkably close to our history types H. Since they lack assertions, they also do not use their system for any automatic verification. Their approach has one significant drawback: the inference algorithm given is not "sound enough" – it is formally proved sound, but the soundness property doesn't preclude inconsistent constraints such as $ev = ev'$ from being present. The *hoif* example at the end of Section 4 is one program for which their system will infer an inconsistent constraint set; our algorithm thus has a stronger soundness property.

The system of [6] is based on linear types and not effect types. Their system allows multiple, dynamically generated, histories. So for instance a unique history can be created for each file opened. This feature seems useful, but it introduces several complexities; for example, an independent escape analysis procedure must be invoked by the type rules. We can achieve similar expressiveness in our theory by using one history stream, but creating a unique singleton to label each separate stream of events; the *openfresh* example at the end of Section 3 shows how a fresh constant can be associated with each opened file in our theory. Their type inference algorithm is not complete, more properly it is unfinished—they provide no particular mechanism for expressing or deciding assertions. Our history types can easily be seen to form a BPA; their analogous usages are more complex (for example including parallelism) and since they are not *prima facie* BPA's it is not known whether model-checking will be possible or feasible. Their system lacks a form of contextual assertion and so it would also not be directly applicable to stack inspection.

In the domain of stack inspection, [11, 10] present constraint-based type systems that statically verify stack soundness. [7] presents a static analysis that model checks a control flow analysis using the CTL logic to express assertions, and they can statically verify the correctness of stacks. These aforementioned systems cannot statically check parameterized privileges, and the analyses are specialized to the stack inspection question and do not take a more general approach as this paper does.

Acknowledgements

Thanks to Rao Kosaraju for discussions on grammar decision problems, and Fei Lu for ideas about model-checking histories.

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] 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.

[4] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI01)*, pages 59–69, 2001.

[5] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic Volume II — Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984.

[6] 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.

[7] 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.

[8] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 303–310. ACM Press, 1991.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Chris Stone. Singleton types and singleton kinds. Technical Report CMU-CS-00-153, Carnegie Mellon University, 2000.

[14] 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.

[15] 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.

[16] Dan S. Wallach and Edward Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, May 1998.