History Type Analysis

Christian Skalka
University of Vermont
skalka@cs.uvm.edu

Scott Smith
Johns Hopkins University
scott@cs.jhu.edu

ABSTRACT

Security abstractions in programming languages benefit from the ability to base access control decisions on the temporal context of program execution. In this paper we formalize the notion of a *history* as a sequence of program events produced during program execution, and which allows execution contexts to be precisely characterized. We define a language λ_{hist} to model the incorporation of histories in evaluation, and present a sound type analysis for statically verifying program safety in the presence of histories. An approximate type inference algorithm is defined, though inference in the general case is shown to be undecidable.

1. INTRODUCTION

Programming language-based security models have a distinct advantage over other security domains, such as networks or smart cards, in which access control decisions are made: in a programming language, there is an ability to base access control decisions on the *context* of execution. This is the case, since secure programs execute in trusted runtimes, where the order of events can be tracked without possibility of forgery. For example, stack inspection [10, 15] 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 theory of execution contexts. Following [1], we call these *histories*, and the naming is intended to evoke this previous work, which makes the case for histories as an efficient and natural foundation for programming language-based security. Our theory is distinguished from [1] in that it is more general, serving as a framework for any system of history-based access control, rather than their particular approach. Also, we focus on the development of relevant type systems, allowing history-based properties to be statically verified; this constitutes the bulk of our presentation.

1.1 History-Based Specification

Fundamental to our approach is the notion of *events* that occur during program execution, comprising a dynamic *history*, a stream of events having occurred so far. With an explicit notion of history as a component of execution, it is possible to specify *history predicates* that must hold at certain program points, given the current history state: for example, before reading a file, we can require that the execution history includes an "open" event for that file, or we can require that a "read" privilege for that file was added to the active security context in a stack inspection model. Given our formalization of the semantics of histories and history predicate verification, we then construct a type system which verifies abstract history properties statically, ensuring safety of typable programs in this model.

Our work is most closely related to static systems for specifying program state changes and resource usage patterns [2, 8, 14, 4]. However, these systems all construct a top-level specification for resource usage patterns; our approach, on the other hand, is modal, in that assertions are made relative to the current program execution context, not the global pattern of behavior. There are various benefits and drawbacks to this modal approach, but it is highly suited to reasoning about access control policies, since these policies are generally conceived as a gateway to pass through during program execution. In the presence of type inference, there is no need for programmers to specify global resource usage patterns; instead they are inferred. One need only insert assertions specifying requirements on run-time histories, a natural and easy-to-use mechanism. The modal approach also meshes well with partial/soft typing, in that some decisions can be postponed until runtime: if a predicate cannot be statically verified to hold, code can be inserted to enforce a dynamic check.

There is a long tradition of modal assertions in programs, beginning with Hoare Logic, and the modal aspect is made explicit in e.g. Dynamic Logic [3]. Our logic differs from these approaches in that past events, and the order in which they occurred, may be considered in assertions.

1.2 Applications of History-Based Specification

Our primary focus is on security applications. In Sect. 5 we show how stack inspection and history-based access control policies can be encoded in, and statically verified by, our theory. These examples suggest that a variety of access control policies can be specified and enforced in our system.

However, our approach is not necessarily biased toward access control; for example, resource usage patterns can be monitored, and audit trails can be precisely specified and verified. Our system can also realize a generalization of the `<assert.h>` C library, allowing assertions about sequences of past events and static verification of these assertions. Since the notion of history in computation is a basic one, it is easy to imagine a wide range of applications.

1.3 The Technical Development

In the rest of the paper, we formalize our ideas in the system λ_{hist}, which is developed via the definition of increasingly expressive variants. We first define an operational semantics of a basic language $\lambda^{c}_{\text{hist}}$ with atomic events, histories (event traces) in configurations, and a specification of how predicates on histories are evaluated. We then develop a sound logical type system that statically ensures validity of all history checks at run-time. After the core $\lambda^{c}_{\text{hist}}$ is presented, we define two different extensions, $\lambda^{\iota}_{\text{hist}}$ which allows multiple disinct history streams to be declared and localized, and $\lambda^{\zeta}_{\text{hist}}$ which supports dynamic constant generation in a static type system. One advantage of our theory is it is small and simple, but still quite expressive. Related theories in this general area [4, 8] are comparatively complex.

A central result of this paper is that semantic equivalence of histories is undecidable, even for simple histories in the presence of recursion. This result characterizes the limitations of the general approach. Nevertheless, we argue that practical partial heuristics for unifying histories may be defined, and we define a type inference algorithm for $\lambda^{\zeta}_{\text{hist}}$, assuming the existence of these methods. Finally, we show how stack inspection and history-based access control can be modeled in $\lambda^{c}_{\text{hist}}$, demonstrating its usefulness.

The technical approach of this paper is most closely related to [4]; in particular, their notion of trace is similar to our history, and their usages U are related to our history types H. In other respects, the approaches differ significantly, e.g. the global vs. modal strategies employed, as discussed above. Also, we use a notion of dynamic singleton similar to the same in [8], rather than the dynamic resources of [4].

2. THE LANGUAGE λ_{hist}

In this section we formally define our first variation of the calculus λ_{hist}—its syntax and operational semantics, and a sound static type analysis that incorporates a notion of history. In subsequent sections we will extend the basic system, but presenting core elements in isolation elucidates the fundamental features of our approach.

2.1 Syntax and Semantics

The syntax and semantics of the basic λ_{hist} system, a variation denoted $\lambda^{c}_{\text{hist}}$, are given in Fig. 1 and Fig. 2. We assume the following 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 $[\![e]\!]_{\text{bool}}$ is 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.

Among the atomic first-class values of the system are constants $c \in \mathcal{C}$, which we do not explicitly interpret. The set $\mathcal{C}$ could be integers, for example, or IP addresses. Their significance in our system is as parameters of *events* and *history predicates*. 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 straghtforward). Ordered sequences of these events constitute histories η, which maintain the sequence of events experienced during program execution. History predicates P, also parameterized by a constant, function just as their name suggests, and may be used to assert desirable properties of histories: $P(c)(\eta)$ either holds or not.

Parameterization of events and predicates allows for a significantly more fine-grained analysis of execution context, as is illustrated in various examples below (especially in Sect. 3). In the presentation of the system, predicates P are left unspecified, but in later sections we discuss some concrete examples. Logics for analyzing security properties in execution contexts have recently been developed [5] that give a good starting point for defining a syntax for P. In some instances we may be interested in events and predicates that have no parameters; in this case we will write ev and P for $ev(c)$ and $P(c)$ respectively, where c is some dummy constant.

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 $P(c)$ is required to hold on the current program history. In case the predicate fails, execution is stuck.

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

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

In the operational semantics, we have:

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

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 (abusing nomenclature, we may also refer to these as simply "histories" if the distinction is clear from context). In essence, any H soundly 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 $ev(c)$, 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 also contain predicates $P(c)$, allowing us to verify predicate checks at the right points in history approximations. History types are inspired by previous systems that analyze ordering of events, including ordered linear logic [9], the control flow based security logic of [5], and especially the resource usage analysis of [4], which have a related typing construct called *usages* that track the order and kind of resource accesses.

The syntax of types for $\lambda^{c}_{\text{hist}}$ is given in Fig. 3. In addition to histories, we include function types $\tau_1 \xrightarrow{H} \tau_2$, where H

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

Figure 1: $\lambda^{c}_{\text{hist}}$ language syntax

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

Figure 2: $\lambda^{c}_{\text{hist}}$ language semantics

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 [6, 13]. 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 $\{c\}$ [12], 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 ϕ 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, the history type H in judgements represents the set of histories that may arise during execution of e; this intuition is formalized in Corollary 1. For example, with f defined as in Sect. 2.1, the following judgements are derivable:

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

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.

2.3 Interpretation

As mentioned 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 may include predicates as well as events, and a $\downarrow$ symbol to denote termination.

Definition 1. *Our interpretation of histories will be defined via strings (called* traces*), denoted θ, over the following alphabet:*

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

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

Definition 2. *The history transition relation is defined as follows:*

$$
\begin{array}{rcll}
ev(c) & \xrightarrow{ev(c)} & \epsilon & \\
P(c) & \xrightarrow{P(c)} & \epsilon & \\
H_1 | H_2 & \xrightarrow{\epsilon} & H_1 & \\
H_1 | H_2 & \xrightarrow{\epsilon} & H_2 & \\
\mu h.H & \xrightarrow{\epsilon} & H[\mu h.H/h] & \\
\epsilon; H & \xrightarrow{\epsilon} & H & \\
H_1; H_2 & \xrightarrow{a} & H_1'; H_2 & \textit{if } H_1 \xrightarrow{a} H_1'
\end{array}
$$

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

Definition 3. *The interpretation of histories is defined*

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

Figure 3: $\lambda^{c}_{\text{hist}}$ language type syntax

$$
\textsc{Var}\ \Gamma, \epsilon \vdash x : \Gamma(x) \qquad
\textsc{Bool}\ \Gamma, \epsilon \vdash b : \textit{bool} \qquad
\textsc{Unit}\ \Gamma, \epsilon \vdash () : \textit{unit}
$$

$$
\textsc{And}\ \Gamma, \epsilon \vdash \wedge : \textit{bool} \xrightarrow{\epsilon} \textit{bool} \xrightarrow{\epsilon} \textit{bool} \qquad
\textsc{Or}\ \Gamma, \epsilon \vdash \vee : \textit{bool} \xrightarrow{\epsilon} \textit{bool} \xrightarrow{\epsilon} \textit{bool}
$$

$$
\textsc{Not}\ \Gamma, \epsilon \vdash \neg : \textit{bool} \xrightarrow{\epsilon} \textit{bool} \qquad
\textsc{Const}\ \Gamma, \epsilon \vdash c : \{c\}
$$

$$
\textsc{Event}\ \frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; \textit{ev}(s) \vdash \textit{ev}(e) : \textit{unit}} \qquad
\textsc{Check}\ \frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; P(s) \vdash P(e) : \textit{unit}}
$$

$$
\textsc{If}\ \frac{\Gamma, H_1 \vdash e_1 : \textit{bool} \qquad \Gamma, H_2 \vdash e_2 : \tau \qquad \Gamma, H_3 \vdash e_3 : \tau}{\Gamma, H_1; H_2 | H_3 \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau}
$$

$$
\textsc{Abs}\ \frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2 \qquad h \text{ fresh}}{\Gamma, \varnothing \vdash \lambda_z x.e : \tau_1 \xrightarrow{\mu h.H} \tau_2}
$$

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

$$
\textsc{Let}\ \frac{\Gamma, H \vdash e[v/x] : \tau}{\Gamma, H \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}
$$

Figure 4: $\lambda^{c}_{\text{hist}}$ logical typing rules

as follows:

$$
[\![H]\!] = \begin{array}{c} \left\{ a_1 \cdots a_n \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} H' \right\} \\ \cup \\ \left\{ a_1 \cdots a_n \downarrow \mid H \xrightarrow{a_1} \cdots \xrightarrow{a_n} \epsilon \right\} \end{array}
$$

Any history interpretation is clearly prefix-closed, and any infinite trace is approximated by the set of its finite prefixes.

The validity of a history is then based on the validity of the predicates that occur in traces in its interpretation. While the definition of predicates is abstract in our system, it is always the case that predicates apply to the histories that precede them; so for any given predicate in a trace, we check its validity with respect to the events in its immediate prefix trace that precede it. The relevant definitions are as follows:

Definition 4. *We obtain a history from a $\downarrow$-free trace θ, written $\hat{\theta}$, inductively as follows:*

$$
\begin{array}{rcl}
\widehat{\textit{ev}(c)} & = & \textit{ev}(c) \\
\widehat{P(c)} & = & \epsilon \\
\widehat{\theta_1 \theta_2} & = & \hat{\theta_1}; \hat{\theta_2}
\end{array}
$$

Definition 5. *We say that a history H is* valid *iff for all $a_1 \cdots a_n P(c)$ in $[\![H]\!]$ it is the case that $P(c)(\widehat{a_1 \cdots a_n})$ holds. A type judgement $\Gamma, H \vdash e : \tau$ is* valid *iff it is derivable and H is valid.*

2.4 Properties

In this section, we 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 3) and progress (Theorem 2) 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 1).

In addition to these positive results, we also report two important negative results. First, we demonstrate that history types do not have a principal types property in the usual sense; for arbitrary expressions there is no unique "best" type, which may be instantiated to other valid types (Lemma 1). As is demonstrated in the proof, this is because typings may be arbitrary with regard to orderings of events; this has obvious relevance for type inference, as discussed in Sect. 4. Even more significantly, for type inference and as a fundamental property of the system, we show that the equivalence relation on histories is undecidable (Theorem 1). We begin by demonstrating these latter results.

Lemma 1. *History type analysis does not possess a principal type property.*

Proof. Consider the following function, where e is some condition:

$$
\begin{array}{l}
\lambda f_1 f_2 g_1 g_2.\ \mathsf{if}\ e\ \mathsf{then}\ (\lambda x.f_1(); \textit{ev}_1(c); g_1()) \\
\qquad\qquad \mathsf{else}\ (\lambda x.f_2(); \textit{ev}_2(c); g_2())
\end{array}
$$

The type of the functions returned in either branch of the conditional in must be the same type $\tau \xrightarrow{H} \tau'$. Furthermore,

note that their definitions require that $ev_1(c)$ and $ev_2(c)$ appear in H, but the ordering of these events in H is not constrained.

From a type inference perspective, H is the unification of the following histories, where each h_x is the abstracted history annotation on the function type x:

$$h_{f_1}; ev_1(c); h_{g_1}$$
$$h_{f_2}; ev_2(c); h_{g_2}$$

Now, each of the following substitutions is clearly a unifier of these histories, each yielding a different ordering of events:

$$\phi_1 = [ev_2(c)/h_{f_1}, \epsilon/h_{g_1}, \epsilon/h_{f_2}, ev_1(c)/h_{g_2}]$$
$$\phi_2 = [\epsilon/h_{f_1}, ev_2(c)/h_{g_1}, ev_1(c)/h_{f_2}, \epsilon/h_{g_2}]$$

However, note that the LUB of ϕ_1 and ϕ_2 in an MGU ordering would be the identity substitution, which is not a unifier of the above histories. □

THEOREM 1. *The relation $[\![H]\!] = [\![H']\!]$ is undecidable.*

PROOF. This question is very similar to the question of whether two context-free grammars generate the same language. That question has been shown undecidable by reduction to Post's Correspondence Problem. The set of terminating traces of a history corresponds to the set of strings generated by a grammar. Any grammar may easily be mapped on to a history H with the same terminating traces—recursive grammar rules may be modeled by history μ's. The PCP undecidability mapping for grammars proceeds by constructing two grammars G_1 and G_2 that are equivalent iff an instance of PCP is solvable. These grammars may be mapped to histories, giving H_1 and H_2. The only difference in $L(G)$ and $[\![H]\!]$ is the presence of unbounded traces in the latter; but, by observing the construction of H_1 and H_2, they contain the identical set of unbounded traces and so for these particular grammars and histories, $L(G_1) = L(G_2)$ iff $[\![H_1]\!] = [\![H_2]\!]$ and a PCP instance has been mapped on to equivalence of histories. □

Now, we state our positive results for history type analysis; proofs are given in the Appendix.

COROLLARY 1. *If $\Gamma, H \vdash e : \tau$ and $\epsilon, e \rightarrow^\star \eta, v$ then $\eta \in [\![\hat{H}]\!]$.*

THEOREM 2 (PROGRESS). *If $\Gamma, H \vdash e : \tau$ is derivable and η, e is irreducible with $\eta; H$ valid, then e is a value.*

THEOREM 3 (HISTORY TYPE SAFETY). *Well-typed expressions don't go wrong in λ^c_{hist}.*

2.5 Localization of Multiple Histories

So far, we have considered a system with a single, global history. However, it may be desirable to allow multiple, distinct histories in programs; as will be discussed in Sect. 5, events and predicates can be defined that implement distinct security paradigms, and multiple histories can allow distinct paradigms to operate in the same program, in a clean and useable manner. Furthermore, localization of histories to certain program regions promotes privacy and security. In this section, we discuss constructs for the implementation and analysis of multiple localized histories.

2.5.1 Ad-Hoc

In a relevant sense, the parameterization of events and histories allows a delineation of history information; given a predicate $P(c)$, it is easy to imagine that the predicate P can "pick out" those events $ev(c)$ in the history parameterized by c. Certain conventions can sharpen this delineation; given a trivial extension of the system with n-ary predicates and events, we can obtain an ad-hoc definition of multiple localized histories with additional parameters using existing language constructs. That is, we may adopt a convention whereby the first parameter of any event and predicate is a history identifier constant c_ι. Then, given c_ι, any predicate $P(c_\iota, \bar{c})$ is defined to apply only to events of the form $ev(c_\iota, \bar{c})$ in the preceding history. Localization of histories is accomplished with normal scoping constructs, e.g.:

$$\mathsf{let}\ hid = c_\iota\ \mathsf{in}\ e$$

will localize the history identifier c_ι to e, where it is referred to as *hid*. Of course, in this scheme it is possible for c_ι to escape from e, and so violate localization, but this flexibility may be desirable in practice.

In the system λ^c_{hist} studied so far, this ad-hoc approach is limited by the fact that constants are declarative– no history can be dynamically generated, a capability that may be desirable in practice. However, in the next Section we extend the system with generative constants, allowing us this level of expression using the same ad-hoc approach.

2.5.2 By Design

In case multiple histories are used heavily, it may be more efficient to provide them as part of the basic design of the system, to include a set of histories in configurations, indexed by unique identifiers, rather than a single history. To model this, we define a new variation on λ_{hist}, called $\lambda^\iota_{\text{hist}}$. With histories defined as before, we posit a set of history identifiers ι as primitives, along with sets of histories indexed by identifiers:

$$\eta \in \mathcal{H} \qquad \textit{histories}$$
$$\iota \in \mathcal{I} \qquad \textit{history identifiers}$$
$$\varsigma \in \mathcal{I} \rightarrow \mathcal{H} \qquad \textit{history indexes}$$

Furthermore, we update the language syntax so that history identifiers are values, and expressions include constructs for declaring histories, and referencing them in events and predicates:

$$e ::= v \mid e\,e \mid ev_e(e) \mid P_e(e) \mid \mathsf{history}\ x\ \mathsf{in}\ e \qquad \textit{expressions}$$

Then, taking a declarative view of history naming, we formalize pre-processing of programs to resolve history names:

DEFINITION 6. *The* pre-processing *of an expression e, denoted $(\!|\, e \,|\!)$, is a function paramaterized by a substitution ϕ, that replaces each declared history variable with a distinct index; so, we have inductively:*

$$(\!|\, ev_x(e) \,|\!)\,\phi = ev_{\phi x}((\!|\, e \,|\!)\,\phi)$$
$$(\!|\, P_x(e) \,|\!)\,\phi = P_{\phi x}((\!|\, e \,|\!)\,\phi)$$
$$(\!|\, \mathsf{history}\ x\ in\ e \,|\!)\,\phi = (\!|\, e \,|\!)\,\phi[x : \iota] \qquad \iota\ fresh$$

along with a homomorphic extension to the other expression forms.

Finally, configurations are updated to include history indexes instead of histories, and reduction is updated appro-

priately:

$$\begin{array}{rcll} \varsigma, ev_\iota(c) & \rightarrow & \varsigma[\iota : (\varsigma(\iota); ev(c))], () & (event) \\ \varsigma, P_\iota(c) & \rightarrow & \varsigma, () \quad \text{if } P(c)(\varsigma(\iota)) & (check) \\ & \vdots & & \end{array}$$

The appropriate extension to the type system is straightforward: History identifier annotations on events and predicates are included in history types, and the derivation rules can be easily be extended to enforce scoping of names. Note that in this system, it is impossible for valuations of history names to escape their scope, unlike in the ad-hoc scheme.

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:

$$\begin{array}{rcl} open' & \triangleq & \lambda x.ev_{\mathsf{open}}(x); open(x) \\ close' & \triangleq & \lambda x.ev_{\mathsf{close}}(x); close(x) \end{array}$$

Then, using a predicate P_{open} that checks for the appropriate property, as follows:

$$\begin{array}{rcl} P_{\text{open}}(c)(\epsilon) & = & \mathsf{false} \\ P_{\text{open}}(c)(\eta; ev_{\mathsf{open}}(c)) & = & \mathsf{true} \\ P_{\text{open}}(c)(\eta; ev_{\mathsf{close}}(c)) & = & \mathsf{false} \\ P_{\text{open}}(c)(\eta; ev(c')) & = & P_{\text{open}}(c)(\eta) \end{array}$$

we may define a *read′* function that only *read*s open files:

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

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 $\lambda^{\zeta}_{\text{hist}}$.

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 with a construct for dynamically generating fresh constants:

$$e ::= \cdots \mid \mathsf{new}\ x\ \mathsf{in}\ e \qquad \textit{expressions}$$

This expression form is equipped with the following semantics:

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

Since these constants are dynamically generated, we must be careful in the type rules to be faithful to this fact.

3.2 Logical Typing Extensions

Now we extend the $\lambda^{c}_{\text{hist}}$ type system to analyze dynamic constants in $\lambda^{\zeta}_{\text{hist}}$, which is nontrivial. 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:

$$s \quad ::= \quad \alpha \mid \zeta \mid c \qquad \textit{singletons}$$

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 $\mathsf{new}\ x\ \mathsf{in}\ e$ construct, we redefine functions types to the following form:

$$\nabla\bar{\zeta}.\tau_1 \xrightarrow{H} \tau_2$$

Here, $\bar{\zeta}$ may occur free in τ_1, H, or τ_2; we write $\text{fv}_\zeta(\tau)$ 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 $\nabla\varnothing.\tau \xrightarrow{H} \tau'$. An analogy to our use of ∇-binding of ζ variables in $\lambda^{\zeta}_{\text{hist}}$ for typing freshly generated constants in the presence of histories, is the use of the $\exists$ binder on function results in [8] in conjunction with a linear logic.

We also extend the language of history types with a similar binding construct:

$$H ::= \cdots \mid \nabla\bar{\zeta}.H \qquad \textit{history types}$$

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:

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

Interpretation of histories is defined as before. With the addition of ∇-binding, we must be explicit about the interpretation of types, and type equality:

DEFINITION 7. *Overloading $[\![\cdot]\!]$, we inductively define the interpretation of singletons and types, parameterized by substitutions ϕ, as follows:*

$$\begin{array}{rcl} [\![\alpha]\!]\phi & = & \phi(\alpha) \\ [\![\zeta]\!]\phi & = & \phi(\zeta) \\ [\![t]\!]\phi & = & \phi(t) \\ [\![c]\!]\phi & = & c \\ [\![unit]\!]\phi & = & unit \\ [\![bool]\!]\phi & = & bool \\ [\![\{s\}]\!]\phi & = & \{[\![s]\!]\phi\} \\ [\![\nabla\bar{\zeta}.\tau_1 \xrightarrow{H} \tau_2]\!]\phi & = & [\![\tau_1]\!]\phi' \xrightarrow{[\![\phi'(H)]\!]} [\![\tau_2]\!]\phi' \\ & & \phi' = \phi[\bar{c}/\bar{\zeta}], \bar{c} \textit{ fresh} \end{array}$$

Note that interpretation $[\![\cdot]\!]$ is non-deterministic, in the sense that we do not specify how to pick $\bar{c}$ fresh, so *any* will do. This entails a careful phrasing of type equality:

DEFINITION 8. *We write $\phi \Vdash \tau = \tau'$ iff $[\![\tau]\!]\phi = [\![\tau']\!]\phi$ is provable, and $\tau = \tau'$ iff $\forall\phi.\phi \Vdash \tau = \tau'$.*

The necessary updates to the logical type derivation rules are given in Fig. 5. The basic ABS and APP are replaced with

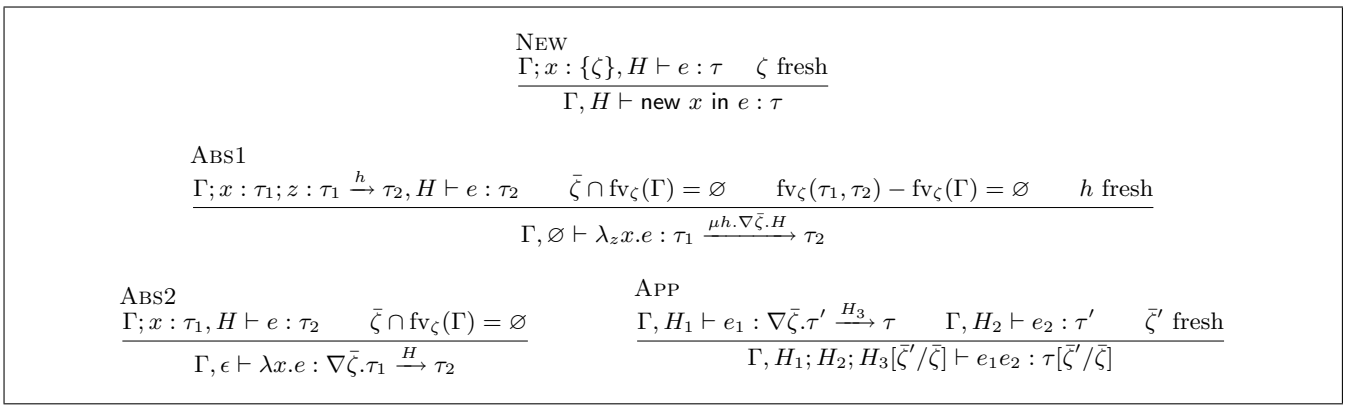

Figure 5: Logical type rules for generative constants in $\lambda^{\zeta}_{\text{hist}}$

the new APP, ABS1, and ABS2, and the NEW rule is introduced. 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:

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

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:

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

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 generative constant theory $\lambda^{\zeta}_{\text{hist}}$. The type inference rules are given in Fig. 6. It is a constraint system, generating a conjunction of equations E. Every term has an obvious typing derivation with these rules, but the equations must then be solved. 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 one source of incompleteness. The ABS rules need to look into the equation set to make sure variable occurrence restrictions are not violated; the easiest way to express that here is to invoke the unification algorithm.

To solve the equations, we first define a type unification algorithm unify_τ in Section 4.1, that either fails or returns a pair ϕ, E_H, where ϕ is a unifier for the "type part" of E and E_H is the set of history equations $H = H'$ imposed by E.

We postulate existence of an algorithm unify_H that returns a (possibly empty) set of unifiers Φ of history equations E_H. History unification is undecidable as a consequence of the undecidability of history equivalence, Theorem 1, so any unify_H algorithm must be an approximation. Also, there may be no "best" unifier, so a set Φ of unifiers which can collectively be viewed as defining the "best" unifier is returned by unify_H. Even though the general question is undecidable, the form of unification equations arising in practice are likely to be quite specialized, and so there is a likelihood that unification will work in practice. Unification of word equations with variables ranging over regular expressions was shown decidable by Shulz [11]. His algorithm is a generalization of Makanin's algorithm [7] for solving word equations. A subject of future work is how well these algorithms and other ideas can be adapted to the history unification problem.

Given unify_τ and unify_H, the top-level algorithm $\mathit{unify}(E)$, which returns a set of unifying substitutions Φ, is defined as follows:

$$\begin{aligned} &\mathit{unify}(E) = \\ &\quad \text{let } \phi_1, E_H = \mathit{unify}_\tau(E) \text{ in} \\ &\quad\quad \text{let } \Phi_2 = \mathit{unify}_H(E_H) \text{ in} \\ &\quad\quad\quad \text{if } \Phi_2 \neq \emptyset \text{ then } \Phi_2 \circ \phi_1 \text{ else } \mathit{fail} \end{aligned}$$

(Here and below we will implicitly map substitution operators on to sets of substitutions, so $\Phi \circ \phi$ is for example $\{\phi_0 \circ \phi \mid \phi_0 \in \Phi\}$.)

4.1 Type Unification

The unification algorithm is standard except for the case

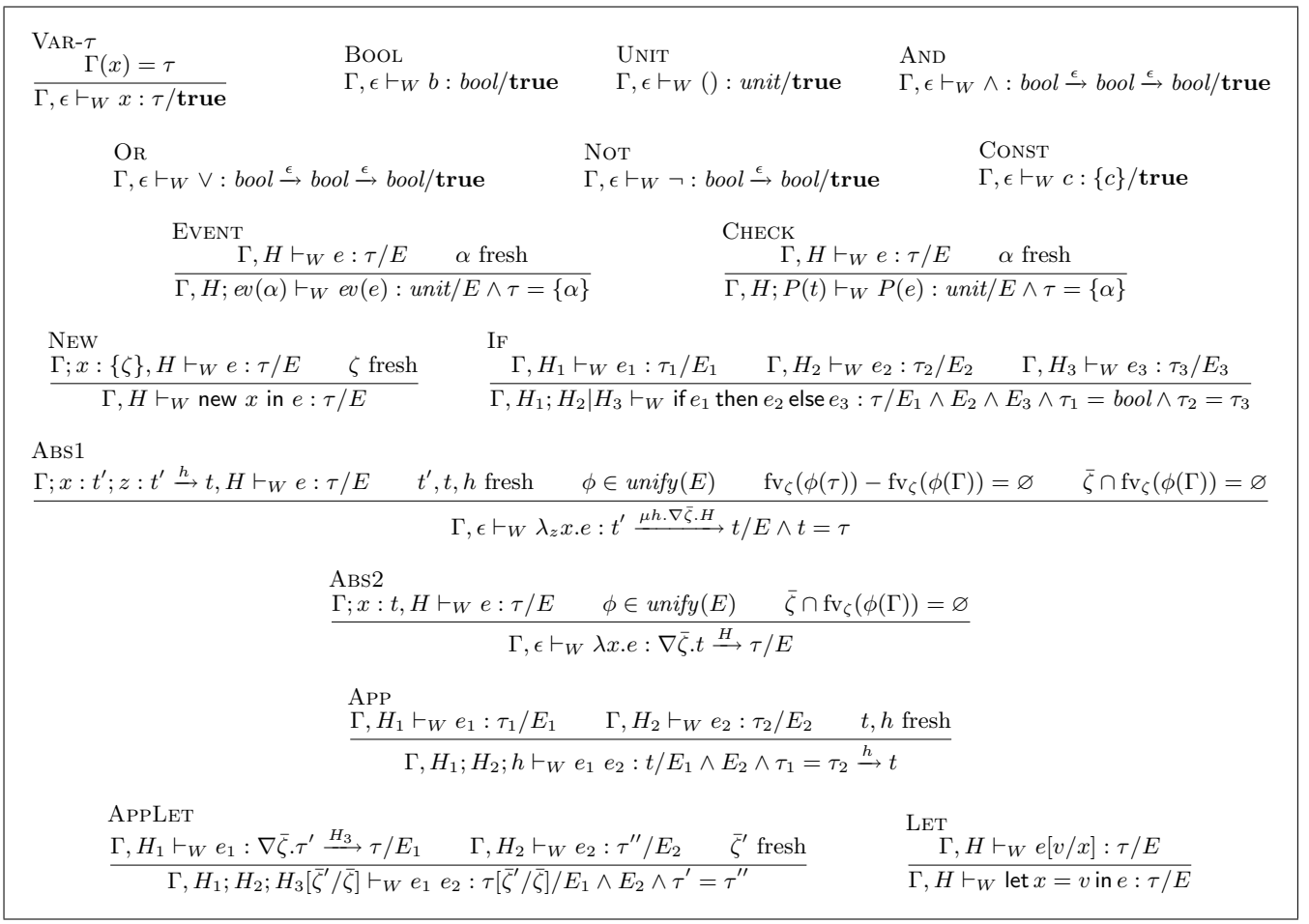

Figure 6: Type inference rules for λ_{hist} with generative constants

of the ∇-bound variables. Since each ζ is a fresh constant, they cannot be unified, unless it is in the context of unifying two function types, $\nabla\bar{\zeta}_1.\tau_1 \xrightarrow{H_1} \tau_1' = \nabla\bar{\zeta}_2.\tau_2 \xrightarrow{H_2} \tau_2'$. And in this case we don't know which ζ_1 in $\bar{\zeta}_1$ is to be matched with which ζ_2 in $\bar{\zeta}_2$ if the length of $\bar{\zeta}$ is greater than one. This matching must thus be done lazily as a post-processing operation after $unify_\tau$ has completed. To simplify the presentation here, we will restrict ourselves to function types with at most one ζ bound, allowing ourselves to skip the post-processing phase.

The type unification algorithm is given in Figure 7. It assumes each variable ζ in a ∇-bound in the original equations E is unique, and also distinct from any free ζ. And, we assume there a predicate $bound(\zeta)$ that holds iff ζ is bound and not free. The substitution ϕ produced, when applied, assumes ∇ does not bind $\bar{\zeta}$.

LEMMA 2. *$unify_\tau$ is a total, computable function.*

LEMMA 3 (SOUNDNESS OF $unify_\tau$). *$\phi(\tau_1) = \phi(\tau_2)$ for $\phi = unify_\tau(E)$ and $\tau_1 = \tau_2 \in E$.*

4.2 Soundness of Type Inference

LEMMA 4 (SOUNDNESS OF TYPE INFERENCE). *Suppose $\Gamma, H \vdash_W e : \tau/E$ and $\Phi = unify(E)$; then, $\phi(\Gamma), \phi(H) \vdash e : \phi(\tau)$ is derivable for each $\phi \in \Phi$.*

The inference algorithm is not complete in two respects, as alluded to above: history unification is undecidable and thus $unify_H$ must be incomplete, and functions returning fresh constants in their return types, $\{\zeta\}$, cannot be used in a fully higher-order manner.

5. APPLICATIONS

We have thus far treated predicates abstractly; for a concrete predicate language the type system must be equipped with a sound decision procedure for determining their validity. Recall that the predicates $P(c)$ are subterms of the final history type H, and the history type is valid iff all histories that could precede $P(c)$ hold for $P(c)$. Since history typings approximate all possible histories that may transpire at run-time and history types may contain subterms $\mu h.H$, there may be an infinite number of histories, and it may be nontrivial to decide the validity of the predicates over this infinite set. Fortunately, the theory is over-expressive in that for many cases it is not difficult to compute history validity. For many cases it suffices to show that a finite subset of $[\![\mu h.H]\!]$ is valid with respect to concrete history predicates; the full interpretation then holds by implication. One principled approach here is to approximate history type interpretations from the bottom up—to consider the shortest histories contained therein (the *basis)*, followed by the

$$\mathit{unify}_\tau(\emptyset) = \emptyset, \emptyset$$

$$\mathit{unify}_\tau(E \cup \{\tau_1 = \tau_1\}) = \mathit{unify}_\tau(E)$$

$$\begin{array}{l}\mathit{unify}_\tau(E \cup \{\{s_1\} = \{s_2\}\}) = \\ \quad \text{if } s_1 = s_2 \text{ then } \mathit{unify}_\tau(E) \\ \quad \text{else if } s_1 = \alpha_1 \text{ and } s_2 = \alpha_2 \text{ then } \mathit{unify}_\tau(E[\alpha_1/\alpha_2]) \circ [\alpha_1/\alpha_2] \\ \quad \text{else } \mathit{fail}\end{array}$$

$$\begin{array}{l}\mathit{unify}_\tau(E \cup \{t = \tau\}) = \\ \quad \text{if } \tau \text{ is } t \text{ then } \mathit{unify}_\tau(E) \\ \quad \text{else if } t \text{ occurs in } \tau \text{ then } \mathit{fail} \\ \quad \text{else let } \phi, E_H = \mathit{unify}_\tau(E[\tau/t]) \text{ in } \phi \circ [\tau/t], E_H\end{array}$$

$$\begin{array}{l}\mathit{unify}_\tau(E \cup \{\nabla\bar{\zeta}_1.\tau_1 \xrightarrow{H_1} \tau_1' = \nabla\bar{\zeta}_2.\tau_2 \xrightarrow{H_2} \tau_2'\}) = \\ \quad \text{let } \phi, E_H = \mathit{unify}_\tau((E \cup \{\tau_1 = \tau_2, \tau_2 = \tau_2')[\bar{\zeta}_1/\bar{\zeta}_2]) \text{ in} \\ \qquad \phi, E_H \cup \{\nabla\bar{\zeta}_1.H_1 = \nabla\bar{\zeta}_2.H_2\}\end{array}$$

Figure 7: Type unification algorithm

histories built on those, etc. Recalling that histories $\mu h.H$ occur as annotations on recursive functions f, the basis of $\mu h.H$ is informally those histories in its interpretation that result when f executes its basis. Informally, this set is obtained by disjoining all the h-variable-free "paths" through H; for example:

$$\mathit{basis}(\mu h.ev_1 \mid h \mid ev_2; (\mu h'.ev_3 \mid ev_4; h')) = ev_1 \mid ev_2; ev_3$$

A normal form of histories, called μ-dnf, is useful in defining a basis. (For the remainder of this section, we work in the basic theory λ^c_{hist} for simplicity.)

DEFINITION 9 (μ-DNF). *A history H is in μ-dnf iff $H = H_1|\ldots|H_n$, at most one H_i may be ϵ, and each other H_i is of the form $H_{i1};\ldots;H_{in_i}$ and each H_{ij} is either $ev(s)$, $P(s)$, h, or $\mu h.H'$, for H' inductively in μ-dnf.*

DEFINITION 10 ($\mu dnf(H)$). *Let $\mu dnf(H)$ be a function that maximally rewrites H according to the following rules and their symmetrical counterparts:*

$$\begin{array}{rcl}(H_1|H_2); H_3 & \Rightarrow & H_1; H_3|H_2; H_3 \\ H_1; \epsilon & \Rightarrow & H_1 \\ \epsilon|\epsilon & \Rightarrow & \epsilon\end{array}$$

LEMMA 5. *$\mu dnf(H)$ is in μ-dnf and $[\![\mu dnf(H)]\!] = [\![H]\!]$.*

Some auxiliary definitions are used to define the basis.

1. *$\mu h.H$ is μ-innermost* iff H contains no μ-abstracted subhistories.

2. The *bases* of a history disjunction is the disjunction of the h-free histories in it:

$$\mathit{bases}(H_1 \mid \cdots \mid H_n) = H_1' \mid \cdots \mid H_j'$$

such that $\forall 0 < i \leq j.H_i' \in \{H_1, \ldots, H_n\} \wedge \mathrm{fv}(H_i') = \varnothing$. If $j = 0$, then the value is a new distinguished history type $\uparrow$; for closure we also add $\mathit{bases}(\uparrow) = \mu dnf(\uparrow) = \uparrow$.

We now can define the basis of a history type.

DEFINITION 11. *The basis of a history H, $\mathit{basis}(H)$, is defined as follows.*

$$\begin{array}{l}\mathit{basis}(H) = \\ \quad \textit{let } H' = \mu dnf(H) \textit{ in} \\ \quad H'[\mu\mathit{basis}(\mu h_1.H_1)/\mu h_1.H_1, \ldots, \mu\mathit{basis}(\mu h_n.H_n)/\mu h_n.H_n] \\ \qquad \textit{where } \mu h_1.H_1, \ldots, \mu h_n.H_n \textit{ are the } \mu\textit{-subterms in } H', \\ \qquad \textit{and where:}\end{array}$$

$$\mu\mathit{basis}(\mu h.H) = \begin{cases}\mathit{bases}(\mu dnf(H)) & \textit{if } H \textit{ is } \mu\textit{-innermost} \\ \mathit{bases}(\mathit{basis}(H)) & \textit{if } H \textit{ is not } \mu\textit{-innermost}\end{cases}$$

We now may define the nth *unrolling* of a μ-bound history $\mu h.H$, written $\circlearrowleft_n \mu h.H$, as follows. The nth unrolling intuitively represents the histories that may result in an n-deep recursive call tree of the function f that $\mu h.H$ annotates. In case the history has a basis, we consider that first and then subsequent, recursive paths, while if the history has no basis we consider unrollings of the recursive cases.

DEFINITION 12. *The nth* μ-unrolling $\circlearrowleft_n$ *is inductively defined as follows:*

$$\begin{array}{rcll}\circlearrowleft_0 \mu h.H & = & \mathit{basis}(H) & \textit{if } \mathit{basis}(H) \neq \uparrow \\ \circlearrowleft_0 \mu h.H & = & H[\uparrow/h] & \textit{if } \mathit{basis}(H) = \uparrow \\ \circlearrowleft_n \mu h.H & = & H[\circlearrowleft_{n-1} H/h] & \end{array}$$

Note that histories of the form $\uparrow; H$ are stuck with respect to the transition rules; this is just what we want, since when that symbol is encountered we are assuming it is a placeholder for a divergent history, so anything after that point cannot be reached.

Now, we define our approximation of a history H, written $\lfloor H \rfloor$, in terms of unrolling. We assume that H is in μ-dnf. In every case the function is homomorphic, except the μ case of course, which is defined as follows:

$$\lfloor \mu h.H_1 | \cdots | H_n \rfloor = \lfloor \circlearrowleft_0 \mu h.H_1 | \cdots | H_n \rfloor \mid \cdots \mid \lfloor \circlearrowleft_n \mu h.H_1 | \cdots | H_n \rfloor$$

Let's assume a history $\mu h.H_1|\cdots|H_n$ annotates a function f. Each of these H_i are history types associated with the control flow paths that that can be taken in a recursive call

tree of f, so if we consider those recursive call trees that are up to n-deep with respect to f, the interpretation of histories ensures we will include all possible orderings of each path H_i taken at least once. The significance of this is that some security properties remain invariant over multiple inclusions of the same groups of events, or groups of ordered events, in histories, as we discuss in more detail below. Thus, if we've seen the n-deep recursive call trees, we've essentially seen them all, since any deeper will ensure that some path H_i is taken more than once.

Now, we can observe that certain security paradigms possess the property that for predicates P and histories H in that paradigm, validity of $\lfloor H \rfloor$ implies validity of H, where n is the maximal degree of H. Since $[\![\lfloor H \rfloor]\!]$ is clearly finite, this yields a decidable validity check for history predicate types in models which are insensitive to repetitions of the same sequence of events.

5.1 Stack Inspection

Stack inspection is a well-studied security model, from both static and dynamic perspectives [10, 15]. Due to lack of space we assume the reader has some familiarity with it, and refer her to the previous citations otherwise. As an example to demonstrate the use and expressiveness of our system, we define an encoding of the stack inspection model λ_{sec} defined in [10]. We consider only unparameterized privileges, as in the current state-of-the-art for static stack inspection, but our parameterized events and predicates plus the singleton kind types should allow parameterized privileges to also be expressed, giving a more powerful analysis than [10].

We begin by instantiating our system with concrete events $push_r$, where r is an individual privilege, $push_p$, where p is a *principal* associated with a set of privileges, and *pop*. These events are added to the program history during execution, where they may be interpreted as stack operations by the function *stackify*, defined as follows:

$$\begin{aligned} \mathit{stackify}(\epsilon) &= \epsilon \\ \mathit{stackify}(\eta; \mathit{push}_r) &= \mathit{stackify}(\eta).r \\ \mathit{stackify}(\eta; \mathit{push}_p) &= \mathit{stackify}(\eta).p \\ \mathit{stackify}(\eta; \mathit{pop}) &= \text{let } s.x = \mathit{stackify}(\eta) \text{ in } s \end{aligned}$$

stackify takes a history and matches push with pop to produce the final stack state—this shows how having an event history gives strictly more information than an event stack. What remains is to define an appropriate encoding of λ_{sec} expressions. Recall that the informal semantics for a check on resource r is to walk back up the stack, making sure each principal p on the stack is authorized for r, until an enable of r is encountered. The predicate $\mathit{inspect}_r$ implements this policy by holding for stacks of that form only.

$$\begin{aligned} (\!|\, \mathsf{check}\ r\ \mathsf{then}\ e \,|\!) &= \mathit{inspect}_r \circ \mathit{stackify}; (\!|\, e \,|\!) \\ (\!|\, \mathsf{enable}\ r\ \mathsf{in}\ e \,|\!) &= \mathit{push}_r; (\mathsf{let}\ x = (\!|\, e \,|\!)\ \mathsf{in}\ \mathit{pop}; x) \quad x \text{ fresh} \\ (\!|\, p.e \,|\!) &= \mathit{push}_p; (\mathsf{let}\ x = (\!|\, e \,|\!)\ \mathsf{in}\ \mathit{pop}; x) \quad x \text{ fresh} \\ (\!|\, \mathsf{fix}\ z.\lambda x.e \,|\!) &= \lambda_z x.(\!|\, e \,|\!) \\ (\!|\, e_1 e_2 \,|\!) &= (\!|\, e_1 \,|\!)(\!|\, e_2 \,|\!) \\ &\vdots \end{aligned}$$

Repeating a pattern of recursion already present on the stack will not affect the validity of stack inspection, thus we may soundly use the history approximation to test validity of stack typings:

LEMMA 6. *Let e be a λ_{sec} expression. If $\Gamma, H \vdash (\!|\, e \,|\!) : \tau$, then validity of $\lfloor H \rfloor$ implies validity of H.*

5.2 History-Based Access Control

In the history-based approach of Abadi and Fournet [1], regions of source code are associated with sets of privileges, and when region boundaries are crossed, a history tracks the security effect by intersecting the rights of the old and new regions; when a privilege is checked, it is in fact checked for membership in this intersection of rights. Here, we model their essential system, and define a static analysis for it, as an instance of our system.

First, we posit sets of atomic privileges p that may be associated with any source code; in particular, we imagine that these privileges annotate function bodies:

$$\lambda_z x.p.e$$

We implement any such function f in our system by an encoding $(\!|\, f \,|\!)$, which adds an event labeled with privileges p to the history:

$$(\!|\, \lambda_z x.p.e \,|\!) = \lambda_z x.ev_p; e$$

Furthmore, as was the case with *stackify* above, we define a function that converts a sequence of events in a history to its interpretation in the model:

$$\mathit{intersect}(ev_{p_1}; \ldots; ev_{p_n}) = p_n \wedge \cdots \wedge p_1$$

Finally, we may implement any privilege check demand_r, parameterized by constants c, as the composition of *intersect* and a membership check:

$$(\!|\, \mathit{demand}_r(c) \,|\!) = (\lambda x.r(c) \in x) \circ \mathit{intersect}$$

Since repeating the same event will have no effect on the validity of these intersections, we may also use the history approximations here:

LEMMA 7. *Let e be a history-based access control expression; if $\Gamma, H \vdash (\!|\, e \,|\!) : \tau$, then validity of $\lfloor H \rfloor$ implies validity of H.*

Acknowledgements

The authors would like to acknowledge Rao Kosaraju's contribution of a direct construction of the grammars used to prove undecidability of equivalence on context-free languages.

6. 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32:129–198, 1977.

[8] 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.

[9] Jeff Polakow. Ordered linear logic and applications. Technical Report CMU-CS-01-152, Carnegie Mellon University, 2001.

[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] K.U. Schulz. Makanin's algorithm: Two improvements and a generalization. In *Word Equations and Related Topics*, volume 572 of *Lecture Notes in Computer Science*, 1990.

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

[13] 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.

[14] 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.

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

APPENDIX

In this section we present the proofs of Theorem 2, Theorem 3, and Corollary 1. The technique is standard, with adaptations for histories; it is similar to a subject reduction approach, but reduction does not preserve history typings per se, rather we must define an invariant that holds.

First, we restate the usual substitution Lemma which is useful in the application case of our main utility Lemma (Lemma 12); the proof follows by structural induction on type derivations:

LEMMA 8. *If both* $\Gamma; x : \tau', H \vdash e : \tau$ *and* $\Gamma, \epsilon \vdash v : \tau'$ *are derivable, then so is* $\Gamma, H \vdash e[v/x] : \tau'$.

We must also state a separate substitution result for recursive substitution, since unrollings do not *quite* preserve history typings:

LEMMA 9. *If both* $\Gamma; z : \tau_1 \xrightarrow{h} \tau_2, H \vdash e : \tau_2$ *and* $\Gamma, \epsilon \vdash \lambda_z x.e : \tau_1 \xrightarrow{\mu h.H} \tau_2$ *is derivable, then so is* $\Gamma, H[\mu h.H/h] \vdash e[v/x] : \tau_2$.

We give a standard substitution result for evaluation contexts, which follows by structural induction on contexts:

LEMMA 10. *If* $\Gamma, H_1; H_2 \vdash E[e] : \tau$ *is derivable with subderivation* $\Gamma', H_1 \vdash e : \tau'$ *for* e *in the hole, and* $\Gamma', H_1' \vdash e' : \tau'$ *is derivable, then* $\Gamma, H_1'; H_2 \vdash E[e'] : \tau$ *is derivable.*

It will also be useful to observe the following property of typings, essentially that the history type associated with a redex in context will be the "earliest history" for the context. This is intuitively correct, and follows immediately by definition of evaluation contexts and associated typing rules, which place the histories associated with redex reduction leftmost in the history of larger contexts.

LEMMA 11. *If* $\Gamma, H \vdash E[e] : \tau$ *is derivable with* e *a redex, then* $H = H_1; H_2$ *with* $\Gamma', H_1 \vdash e : \tau'$ *a subderivation for* e *in the hole.*

Now, as alluded to above, we proceed by observing an invariant preserved during reduction. We argue that history types represent an approximation of the histories occurring during evaluation; in fact, as evaluation progresses, more and more of the "true" history is reified in the configuration, sans predicate annotations, with the associated approximation "chewed off" the history typings in resulting expressions. This intuitive description of the relevant invariant is formalized as the relation $\sqsupseteq$:

DEFINITION 13. *We write* $H' \sqsupseteq H$ *iff for all* $a_0 \cdots a_n\theta \in [\![H]\!]$ *there exists* $\theta'\theta \in H'$ *such that* $a_0; \ldots; a_n = \hat{\theta'}$.

The following consequences of this definition are noted; property (2) holds by property (1), since $[\![\mu h.H]\!] \supseteq [\![H[\mu h.H/h]]\!]$.

COROLLARY 2. *The following properties hold:*

1. *If* $[\![H]\!] \subseteq [\![H']\!]$ *then* $H' \sqsupseteq H$
2. *For closed* $\mu h.H$, *we have that* $\mu h.H \sqsupseteq H[\mu h.H/h]$.
3. *If* $H' \sqsupseteq H$ *then validity of* H' *implies validity of* H.

Now we may demonstrate our central utility Lemma, which uses the relation $\sqsupseteq$ to specify the relevant typing property preserved during evaluation. For brevity we restrict ourselves to the interesting cases:

LEMMA 12. *If* $\Gamma, H \vdash e : \tau$ *is derivable and* $\eta, e \to \eta', e'$, *then* $\Gamma, H' \vdash e' : \tau$ *is derivable with* $\eta; H \sqsupseteq \eta'; H'$.

PROOF. By case analysis on $\to$.

Case β. In this case the following assertions hold by assumption:

$$e = (\lambda_z x.e)v \qquad e' = e[v/x][\lambda_z x.e/z] \qquad \eta' = \eta$$

Furthermore, by definition of the typing rules we may partially reconstruct the derivation of $\Gamma, H \vdash e : \tau$ as follows, with $H = \mu h.H''$:

$$\dfrac{\dfrac{\Gamma; x : \tau'; z : \tau' \xrightarrow{h} \tau, H'' \vdash e : \tau \qquad h \text{ fresh}}{\Gamma, \epsilon \vdash \lambda_z x.e : \tau' \xrightarrow{H} \tau} \qquad \Gamma, \epsilon \vdash v : \tau}{\Gamma, H \vdash (\lambda_z x.e)v : \tau}$$

But then by Lemma 8 and Lemma 9 we have that:

$$\Gamma, H''[H/h] \vdash e' : \tau$$

is derivable, and by Corollary 2 we have $H \sqsupseteq H''[H/h]$, so clearly $\eta; H \sqsupseteq \eta'; H''[H/h]$, since $\eta = \eta'$.

Case *event*. In this case the following assertions hold by assumption:

$$e = ev(c) \qquad e' = () \qquad \eta' = \eta; ev(c)$$

Furthermore, by definition of the typing rules we can reconstruct the following derivation:

$$\dfrac{\Gamma, \epsilon \vdash c : \{c\}}{\Gamma, ev(c) \vdash ev(c) : \mathit{unit}}$$

Therefore the following assertions hold in this case:

$$H = ev(c) \qquad \tau = \mathit{unit} \qquad \Gamma, \epsilon \vdash e' : \mathit{unit}$$

and $\eta; ev(c) \sqsupseteq \eta'; \epsilon$ by implication and Corollary 2, so this case holds.

Case *check*. In this case the following assertions hold by assumption:

$$e = P(c) \qquad e' = () \qquad \eta' = \eta$$

Furthermore, by definition of the typing rules we can reconstruct the following derivation:

$$\dfrac{\Gamma, \epsilon \vdash c : \{c\}}{\Gamma, P(c) \vdash P(c) : \mathit{unit}}$$

Therefore the following assertions hold:

$$H = P(c) \qquad \tau = \mathit{unit} \qquad \Gamma, \epsilon \vdash e' : \mathit{unit}$$

and $\eta; P(c) \sqsupseteq \eta; \epsilon$ by definition, so this case holds.

Case *context*. In this case the following assertions hold by assumption and Lemma 11:

$$e = E[e_1] \text{ with } e_1 \text{ a redex} \qquad e' = E[e_2] \qquad \eta, e_1 \to \eta', e_2$$

$$H = H_1; H_2 \qquad \Gamma', H_1 \vdash e_1 : \tau'$$

Furthermore, by the other cases of this Lemma and Lemma 10 we have:

$$\Gamma', H_1' \vdash e_2 : \tau' \qquad \eta; H_1 \sqsupseteq \eta'; H_1' \qquad \Gamma', H_1'; H_2 \vdash E[e_2] : \tau$$

and $\eta; H_1; H_2 \sqsupseteq \eta'; H_1'; H_2$ by implication, so this case holds. □

Given the preceding, proofs of the results stated in the main paper text are straightforward:

Corollary 1. *If $\Gamma, H \vdash e : \tau$ and $\epsilon, e \to^\star \eta, v$ then $\eta \in [\![\hat{H}]\!]$.*

Proof. By Lemma 12 and induction on the length of the reduction we have that $H \sqsupseteq \eta$; since $\eta = [\![\eta]\!] = [\![\hat{\eta}]\!]$, the result follows. □

Theorem 2 (Progress). *If $\Gamma, H \vdash e : \tau$ is derivable and η, e is irreducible with $\eta; H$ valid, then e is a value.*

Proof. (Sketch): The proof proceeds by contradiction and case analysis on stuck expressions. In almost all cases, the proof follows in the same manner as analogous well-known results for functional calculi with constants; the novel case is $e = E[P(c)]$, which reduces to absurdity since validity of $\eta; H$ implies validity of $P(c)(\eta)$ and therefore reducibility of $\eta, E[P(c)]$. □

Theorem 3 (History Type Safety). *Well-typed expressions don't go wrong in λ^c_{hist}.*

Proof. Suppose on the contrary that $\epsilon, e \to^n \eta, e'$ with η, e' stuck. Since e is well-typed, there exists a derivable judgement $\Gamma, H \vdash e : \tau$ with valid H by definition. Furthermore, by Lemma 12 and induction on n we have that $\Gamma, H' \vdash e' : \tau$ such that $H \sqsupseteq \eta; H'$. But since H is valid, therefore $\eta; H'$ must also be valid by Corollary 2; hence e is a value by Theorem 2, which is a contradiction. □