Types and Abstract Interpretation for Authorization Hook Advice

Christian Skalka
Computer Science
University of Vermont
ceskalka@uvm.edu

David Darais
Computer Science
University of Vermont
david.darais@uvm.edu

Trent Jaeger
Computer Science and Engineering
Penn State University
tjaeger@cse.psu.edu

Frank Capobianco
Computer Science and Engineering
Penn State University
frank@cse.psu.edu

Abstract—**Authorization hooks are access control checks that prevent unauthorized principals from interacting with some protected resource, and are used extensively in critical software such as operating systems, middleware, and server programs. They are often intended to mediate information flow between subjects (e.g., file owners), but typically in an ad-hoc manner. In this paper we present a static type and effect system for detecting whether authorization hooks in programs properly defend against undesired information flow between subjects. A significant novelty of our approach is an integrated abstract interpretation-based tool that guides system clients through the information flow consequences of access control policy decisions.**

I. Introduction

This paper introduces a language-based approach to address a practical security problem in the design of software systems. Designers of software systems often wish to restrict information flows which occur during execution, e.g., flows between different users of the software, or flows between software components separated by module or function boundaries. Although there is extensive literature on enforcing these information flow policies using dynamic information flow monitors, in practice these flows are often restricted using access control checks, also called authorization hooks. In these systems, there is a disconnect between the placement of authorization hooks, and their enforcement (or non-enforcement) of the intended information flow policy. Our aim is to develop a tool for predicting whether an access control policy, instrumented in source code via authorization hooks, enforces an information flow policy during execution. Our main technical contribution is an approach that combines methods in type theory and abstract interpretation to statically and automatically predict the effects of authorization hooks on information flow in programs. Our results include a compositional, higher order type-and-effect system for approximating authorization hook events which occur during execution, and the design and implementation of an abstract interpreter which analyzes trace-effects synthesized in types to predict violations of a given information flow policy.

Authorization hooks are used in various practical settings such as X Server and the Linux Kernel, and have been previously studied from a systems and languages perspective in the research literature [10, 11, 40]. Authorization hooks are access control checks; they ensure that a given subject is authorized for specified privilege. The precise nature of subjects and resources can vary between applications—subjects may be Unix-style "users" controlling an I/O channel, or subjects may be an IP address at the other end of a socket. A privilege is typically an operation on a program object, e.g. read or write access to a specific memory location or object field. In any case, access rights are defined statically in an *access control policy*, and program execution is terminated if an authorization hook check fails at runtime.

Authorization hooks are typically reflective of an underlying access control policy. However, the larger aim of hook placements is typically to enforce an *information flow* policy, with protected operations mediating communication between subjects [17, 26, 38]. That is, resources should be protected by hooks in such a way that communication between certain subjects is disallowed. But because false positives are considered intolerable in certain non-critical systems [26] and programmers are often willing to sacrifice policy precision for simplicity, information flow policies can be violated in programs. Furthermore, no previous work exists on automated analysis of information flow policy enforcement via authorization hooks, so even state-of-the-art automated hook placement methods [26] are ad hoc with respect to information flow policies. Therefore, there is a practical opportunity for automated analyses to identify points in programs where violations can occur to aid programmers in mediating information flows purely through hook placement.

A. Motivations

As mentioned, the true point of authorization hooks is to enforce an information flow policy, even though hooks fail or succeed based on an access control policy. In fact, as described in previous work [17], access control *policies* (not hooks) are understood to statically *induce* an information flow policy. For example, suppose an access control policy allows a subject $\mathtt{s}_1$ to read/write from/to a file `fil`, and allows a subject $\mathtt{s}_2$ only to read from the file `fil`. This implicitly allows information to flow from $\mathtt{s}_1$ to $\mathtt{s}_2$, but not to any other subject, and does not allow flow from any subject to $\mathtt{s}_1$.

Depending on concerns, taint analysis [1] or even stricter noninterference may be the intended information flow semantics. In any case, it is critical to observe that *enforcing any*

This research was supported by the National Science Foundation (NSF) under Grant Numbers CNS-1408880 and CNS-1408801.

```
long sys_fcntl(unsigned int fd, unsigned int cmd,
    unsigned long arg) {
  struct file * filp;
  filp = fget(fd);
  err = security ops->file_ops->fcntl(filp, cmd,
      arg);
  err = do_fcntl(fd, cmd, arg, filp);
  ...
}
static long do_fcntl(unsigned int fd, unsigned int
    cmd, unsigned long arg, struct file * filp) {
  ...
  switch(cmd) {
   ...
   case F_SETLK:
    err = fcntl_setlk(fd, ...);
   case F_SETOWN:
    lock_kernel();
    err = security ops->file_ops->setown(filp); ...
    filp->f_owner.pid = arg; ...
   case F_SETLEASE:
    err = fcntl_setlease(fd, filp, arg);
   break;
  }
  ...
}
int fcntl_setlease(unsigned int fd, struct file
    *filp, long arg) {
  ... filp->f_owner.pid = current->pid; ...
}
```

Fig. 1. Linux 2.4.9 Code with Security Vulnerability.

access control or information flow policy with authorization hooks is not automatic, and must be accomplished by correct instrumentation of code with authorization hooks. This is because authorization hooks are manually placed in an ad-hoc manner, and operations on objects are not necessarily mediated even if they are specified in an access control policy. For example, a write to file `fil` is not automatically checked, rather a write authorization hook must be explicitly inserted in code before it. This is basically due to efficiency and programmer understanding of policies. For example, if $n > 1$ writes to `fil` occur in sequence, it is more efficient to just insert a single initial `fil` write privilege check, instead of n checks. Indeed, significant previous work has focused on minimization authorization hook placements, and programmers are often willing to tolerate false negatives in lieu of false positives and complexity of policies [26]. But this manual instrumentation method is error-prone.

A common source of error is programmers simply overlooking certain crucial placements. Listing 1 shows a vulnerability in the original Linux Security Modules implementation [10], for protecting access to files' `f_owner.pid` field when it is updated via the `fcntl` system call. This field tells the kernel which process to send signals to regarding I/O on that file, creating a flow of sensitive information between the subjects that can access the file and the specified file owner process via the kernel. There is an authorization hook for the SETOWN privilege under the F_SETOWN case in `do_fcntl`, but not in the F_SETLEASE case, where the `f_owner.pid` field of `filp` file is set to the current process id. This allows any process that has an FCNTL privilege over a file to create an information flow that should be protected by the stronger SETOWN privilege. We provide a model of this example in our core language in Section VI.

B. Technical Challenges and Strategies

More recent work [38] has considered examples such as web server code, that illustrate several technical challenges our system needs to address. Consider the following server loop code represented in a typical functional style[1]. Here, *socket* is a socket used to communicate with arbitrary external clients, *pwds* is a filehandle to the passwords file, and *fpage* is a filehandle to a public front page. In this code, a function *rpwds* is defined, that protects a read of *pwds* with an authorization hook checking whether the given credential *auth* establishes password file read authorization. The server receives authorization data *cred* over the socket, and if the client is "authorized" will make a call to *rpwds* to retrieve the password data, providing the credential `ext_hi` for the encapsulated hook. This data is then written to the socket. If the client is not authorized, then harmless data from *fpage* is written to the socket.

```
let socket = ... in      // external connection
let pwds = ... in       // passwords file
let fpage = ... in      // harmless front page
let rpwds = λauth.(hook(auth, pwds.fname, r); pwds.data.read()) in
let loop = λ_z x.
    let cred = socket.data.read( ) in
    let html = if authorized(cred) then rpwds(ext_hi) else fpage in
    let data = html.data.read() in
    socket.data.write(data); z x
in loop()
```

This example highlights several technical challenges for our static analysis. The call to the authorization hook occurs before the write to the socket, so analysis must be *flow-sensitive* to detect that the hook can protect against unsafe information flow. When analyzing the code, programmers may want to explore possible execution paths depending on the outcome of the authorization hook, thus *path-sensitivity* is also a desirable feature of the analysis. Noting that information flows of interest occur inside the non-terminating control loop, the analysis must be able to compute a *fixpoint* of loop executions. And while the check of the client-provided *cred* value and the subsequent authorization hook presumable defends against unsafe information flow, the socket must still be approximated as an external channel in the static analysis and thus can show up as a *false positive*.

To address these challenges, in this paper we develop an analysis that combines a type theory with abstract interpretation. The type system statically computes a conservative prediction of program event traces as an effect, where events are either direct information flows or authorization hook checks. The abstract interpretation provides interactive tool support with flow and path sensitivity, allowing programmers to explore different execution paths based on different authorization hook outcomes. The abstract interpretation also supports

[1] This syntax is consistent with the syntax formally defined in Section II.

fixpoint computation for event traces of recursive functions. And while the abstract interpretation is able to highlight potentially undesirable information flows, it does not reject unsafe programs but instead directs programmers to potential problems.

Both of these motivating examples (the server code above and the Linux code in Figure 1) will be reconsidered in Section VI, where we will reformulate them in our language model and discuss actual analysis results.

C. Paper Overview and Contributions

A body of previous work exists on static enforcement of information flow, including classic [16] and more recent [21] flow-sensitive analyses. However, none of these consider the interaction of information flow and authorization hooks, and as discussed in Section I-A various technical nuances prevent a simple retrofit of previous approaches to address the problem. Our analysis also has other distinct benefits in this application space, including the ability to identify specific problematic information flows between subjects, and the ability to identify conditions of an access control policy that may allow or disallow insecure information flows between subjects in a path-sensitive manner. Extended examples that highlight and illustrate these points are discussed in Sections I-A and VI. We also discuss related work in Section VIII.

In this paper we establish foundations for our analysis in a core language model that captures essential problem features. These foundations include a new higher-order language model (Section II) that incorporates a notion of communication *channels* between *subjects* as an abstraction of, e.g., socket or file interactions. The language model also includes a formal notion of access control policy and authorization hooks. We define a static type analysis (Section III) that generates a *trace effect* (Section III-A) approximating the effect of channel communications and authorization hook checks- crucially, the interpretation of trace effects is flow sensitive, so in particular we can predict whether suspect communication events occur before or after authorization checks. We formulate a logical interpretation of trace effects that simulates program execution, and allows the statement and proof of formal properties (Section IV). We implement this interpretation as an abstract interpretation (Section V).

Aside from the technical novelty of the synergy of type theory and abstract interpretation, this approach has the benefit of being (1) modular, (2) flexible in terms of the expressiveness and precision of the abstract interpretation including path-sensitive analysis of hook placement, and (4) an immediately realizable implementation of a tool for auditing hook placements by allowing programmers to explore the consequences of hook placement/removal. This tool is provided in an accompanying public GitHub repository [2] available as open source.

$$
\begin{array}{rcl}
n & \in & \mathbb{Z},\ b \in \{\mathsf{true}, \mathsf{false}\},\ \mathbf{c} \in \mathcal{C},\ p \in \mathcal{L} \\
v & ::= & x \mid \lambda_z x.e \mid b \mid n \cdot p \mid \mathbf{c} \mid \langle \mathrm{s} \rangle \mid \{f = v; \ldots ; f = v\} \mid () \\
\oplus & ::= & + \mid \cdots \\
e & ::= & v \mid e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{let}\, x = v\, \mathsf{in}\, e \mid \{f = e; \ldots ; f = e\} \\
 & & e.f \mid n \mid e \oplus e \mid e.\mathsf{read}() \mid e.\mathsf{write}(e) \mid e{\downarrow_p}\mid \mathsf{hook}(e, e, e) \\
E & ::= & [\,] \mid v\, E \mid E\, e \mid \mathsf{let}\, x = E\, \mathsf{in}\, e \mid E.f \mid \\
 & & \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e \mid E \oplus e \mid v \oplus E \mid E{\downarrow_p}\mid \\
 & & \mathsf{hook}(E, e, e) \mid \mathsf{hook}(v, E, e) \mid \mathsf{hook}(v, v, E) \mid \\
 & & \{f = E; \ldots ; f = e\} \mid \cdots \mid \{f = v; \ldots ; f = E\}
\end{array}
$$

Fig. 2. Λ_{hook} source and runtime language syntax

Formal highlights include Theorem 4.5 in Section IV which establishes that our (non-algorithmic) *concrete* interpretation of trace effects derived by typing soundly predicts information flow violations. In Theorem 5.2 in Section V we establish that our algorithmic *abstract* interpretation correctly approximates the concrete one. The consequence of combining these results is Theorem 5.4 which shows our abstract interpretation of trace effects derived by typing soundly predicts all possible violations of information flow.

In addition to soundness, we account for the algorithmic complexity of our abstract interpretation algorithm, which is polynomial in the size of the program and subjects additively for the basic analysis, and exponential in the nesting depth of access control checks (a low constant in practice) for a more precise path sensitive instantiation of our analysis.

II. Core Language Model

In this Section we define a core language model called Λ_{hook} intended to capture critical elements of our intended application space. A main problem we are concerned with is detecting flows of information between *subjects* that violate information flow policy, due to weaknesses in access control policy. By subjects we mean external program actors such as Linux users, who are able to communicate via I/O *channels* such as I/O streams, sockets, etc. Subjects can potentially communicate in programs via normal program data flow, but this communication can be interrupted by failing authorization hooks that block computation.

For the purposes of our metatheory, we define our core language model, called Λ_{hook}, to include a dynamic direct information flow analysis, aka a dynamic taint analysis. This allows us to demonstrate a type safety result showing that our static analysis safely predicts information flow violations at runtime (Theorem 4.5). However, it is important to note that because our analysis is static, the intended object language need not include a taint analysis.

A. Access Control Policy

To model authorization hooks, access control policies must also be included in our model. We follow the style of previous formalizations [11], where access control policies define how subjects can access objects—in particular, access control policies are sets of 3-tuples $(\mathrm{s}, \mathbf{o}, \mathbf{c})$ specifying which subjects s can access which objects $\mathbf{o}$ via which operation $\mathbf{c}$, and hooks check a given access request for membership in the access

[2] https://github.com/uvm-plaid/hook-ai

control policy. For example, if `root` should be granted `read` access to the `passwds` file, then $(\texttt{root}, \texttt{passwds}, \texttt{read}) \in \mathcal{A}$. In practice, subjects are often represented by UIDs, objects are represented by fixed identifiers, and operations are represented as e.g. enumerated types. Hooks themselves have a dynamic flavor—for example in the `fcntl` authorization check in `sys_fcntl` in Listing 1, parameters `filp`, `cmd`, and `arg` refine the `fcntl` privilege, and the owner of the `current` process is the implicit subject parameter of the check.

B. Information Flow Policy

The language model must also include an *information flow policy*. Subjects are assumed to be allowed a certain *level* of knowledge, in the style of recent approaches to characterizing information flow semantics [28]. We posit a security lattice $\mathcal{L}$ with points aka levels p, an ordering $\preccurlyeq$, and meet $\wedge$ and join $\vee$ operations. The lattice can be interpreted as a confidentiality lattice or dually as an integrity lattice in the usual manner, though in this presentation we generally assume that the concern is confidentiality.

Subjects $\mathtt{s}$ are associated with levels in $\mathcal{L}$ depending on what they "know"—to capture this formally we let $\mathcal{K}$, called *subject knowledge*, range over mappings from subjects to $\mathcal{L}$. An assumed *subject information flow policy* $\mathcal{K}_{pol}$ posits "intended" security levels for subjects—that is, the level of information that is allowable to communicate to each subject.

As discussed informally in Section I-A, in practice this is typically induced by the access control policy. Formally, following previous work [17], we could formalize this as follows. Under the assumption that operations include just read $\mathbf{r}$ and write $\mathbf{w}$, and introducing a mapping g from objects to $\mathcal{L}$ just for the purposes of the definition, we can define $\mathcal{K}_{pol}$ and g to be the least mappings satisfying the following rules:

$$\frac{(\mathtt{s}, \mathbf{o}, \mathbf{w}) \in \mathcal{A}}{\mathcal{K}_{pol}(\mathtt{s}) \preccurlyeq g(\mathbf{o})} \qquad\qquad \frac{(\mathtt{s}, \mathbf{o}, \mathbf{r}) \in \mathcal{A}}{g(\mathbf{o}) \preccurlyeq \mathcal{K}_{pol}(\mathtt{s})}$$

However, for generality we will not assume that $\mathcal{K}_{pol}$ must be derived from $\mathcal{A}$ in this way, nor that mediated operations only include read and write.

Depending on information flows during computation, a subject's knowledge may evolve beyond $\mathcal{K}_{pol}$. Thus, we need to consider a semantics of "knowledge evolution" during runtime, similar to e.g. attacker knowledge [1, 28]. To formally represent extensions of knowledge, we write $\mathcal{K}[\mathtt{s} \mapsto p]$ to denote the mapping that agrees with $\mathcal{K}$ on all points except it maps $\mathtt{s}$ to p. Intuitively, a program has an information flow violation iff a subject $\mathtt{s}_1$ communicates information to another subject $\mathtt{s}_2$ at a lower security level. More specifically, an information flow between $\mathtt{s}_1$ and $\mathtt{s}_2$ is disallowed by the policy $\mathcal{K}_{pol}$ when $\mathcal{K}_{pol}(\mathtt{s}_1) \not\preccurlyeq \mathcal{K}_{pol}(\mathtt{s}_2)$. As the program executes, information flows occur between subjects, and we write $\mathcal{K}$ as the final (or intermediate) mapping from subjects to the information level which flows to them (i.e., the highest level of any information they have "learned"). An information flow violation is detected for some $\mathcal{K}$ when $\mathcal{K}(\mathtt{s}) \not\preccurlyeq \mathcal{K}_{pol}(\mathtt{s})$ for some $\mathtt{s}$ i.e., a subject learned more information than the policy allows.

$$\begin{array}{rcll}
\mathcal{K}, (\lambda_z x.e)v & \rightarrow & \mathcal{K}, e[v/x][\lambda_z x.e/z] & (\beta) \\
\mathcal{K}, n & \rightarrow & \mathcal{K}, n \cdot \bot & (\mathit{ConstTaint}) \\
\mathcal{K}, (n_1 \cdot p_1) + (n_2 \cdot p_2) & \rightarrow & \mathcal{K}, (n_1 + n_2) \cdot p_1 \vee p_2 & (\mathit{PlusProp}) \\
\mathcal{K}, \mathsf{if\ true\ then}\, e_1 \,\mathsf{else}\, e_2 & \rightarrow & \mathcal{K}, e_1 & (\mathit{IfT}) \\
\mathcal{K}, \mathsf{if\ false\ then}\, e_1 \,\mathsf{else}\, e_2 & \rightarrow & \mathcal{K}, e_2 & (\mathit{IfF}) \\
\mathcal{K}, \mathsf{let}\, x = v \,\mathsf{in}\, e & \rightarrow & \mathcal{K}, e[v/x] & (\mathit{Let}) \\
\mathcal{K}, \{\ldots, f = v, \ldots\}.f & \rightarrow & \mathcal{K}, v & (\mathit{Select}) \\
\mathcal{K}, \langle \mathtt{s} \rangle.\mathsf{read}() & \rightarrow & \mathcal{K}, n \cdot \mathcal{K}(\mathtt{s}) & (\mathit{ReadChan}) \\
\mathcal{K}, \langle \mathtt{s} \rangle.\mathsf{write}(n \cdot p) & \rightarrow & \mathcal{K}[\mathtt{s} \mapsto (\mathcal{K}(\mathtt{s}) \vee p)], () & (\mathit{WriteChan}) \\
\mathcal{K}, n \cdot p{\downarrow_{p'}} & \rightarrow & \mathcal{K}, n \cdot p \wedge p' & (\mathit{Downgrade}) \\
\mathcal{K}, \mathsf{hook}(\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c) & \rightarrow & \mathcal{K}, () \quad \text{if } (\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c) \in \mathcal{A} & (\mathit{Hook}) \\
\mathcal{K}, E[e] & \rightarrow & \mathcal{K}', E[e'] \quad \text{if } \mathcal{K}, e \rightarrow \mathcal{K}', e' & (\mathit{Context})
\end{array}$$

Fig. 3. Λ_{hook} operational semantics

C. Source Language Syntax

The language Λ_{hook}, defined syntactically in Figure 2, includes a normal λ-calculus with first class functions $\lambda_z x.e$ with "self" variable z for recursion, and let-expressions to support Hindley-Milner polymorphism with the value restriction. We additionally define $\lambda x.e$ as syntactic sugar for $\lambda_z x.e$ with z not free in e, sequencing $e_1; e_2$ as syntactic sugar for $(\lambda x.e_2)e_1$ with x not free in e_2, and let-expressions of the form $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ (where e_1 is not a value) as syntactic sugar for $(\lambda x.e_2)e_1$ when e_1 is not a value.

The language also includes an abstract notion of *channels*, that may be read as $e.\mathsf{read}$, and written to as $e_1.\mathsf{write}(e_2)$ where e_1 is the channel being written to and e_2 is the information being written. For simplicity in this presentation we assume that only integral values are read from and written to channels but this could be generalized. Channels are identified by *subjects* $\mathtt{s}$ that are determined statically, though channels are first class values. We imagine that channels are associated e.g. with files or sockets, and subjects identify their owners. Also included is declassification $e{\downarrow_p}$, that allows downgrading of the security level of information in the valuation of e. We note that this is purely an operational effect—typically declassification would be invoked upon termination of a robust and trusted sanitizer.

To model authorization hooks, the syntax of Λ_{hook} includes first-class *constant* values $\mathbf{c}$ that are provided as arguments to authorization hooks during computation. Authorization hooks themselves are represented by the expression form $\mathsf{hook}(e, e, e)$.

D. Operational Semantics

We define a dynamic taint analysis Λ_{hook} in a standard manner [29]. For clarity in this presentation we only track taint on numeric values, and therefore include *tainted numbers* $n \cdot p$ that are labeled with their security level p in the language of values. Computational continuations are represented as evaluation contexts E defined also in Figure 2.

An operational semantics for Λ_{hook} is defined in Figure 3. The semantics are defined in a small-step style, with $\rightarrow$ a *reduction* relation on *configurations* $\mathcal{K}, e$ where $\mathcal{K}$

represents the current state of subject knowledge. Some of the reduction rules are standard, but several bear discussion. In rule ConstTaint, we specify to assign $\bot$ to lexical constants in programs—this could be generalized, so that different constants get different security levels, but in this presentation we aim to focus on the effects of channel interactions. In the rule PlusProp, we specify taint propagation in the usual manner, by joining the security levels of the operands to the security level of the result. In rule ReadChan, the result of reading from a channel is an arbitrary integer n that is tainted with the security level associated with the subject $\mathtt{s}$'s *current state of knowledge* $\mathcal{K}(\mathtt{s})$. In rule WriteChan, writing a value n at security level p has the effect of extending $\mathtt{s}$'s current knowledge, as $\mathcal{K}[\mathtt{s} \mapsto \mathcal{K}(\mathtt{s}) \vee p]$. Note that if $p \not\leq \mathcal{K}_{pol}(\mathtt{s})$, this has the effect of raising $\mathtt{s}$'s level of knowledge beyond acceptable bounds. This is the main safety property we are concerned with, defined formally as follows:

Definition 2.1: A program e has an *information flow violation* iff $\mathcal{K}_{pol}, e \to^* \mathcal{K}, e'$ where there exists $\mathtt{s}$ such that $\mathcal{K}(\mathtt{s}) \not\leq \mathcal{K}_{pol}(\mathtt{s})$.

However, observe that interceding hooks can prevent unsafe information flows from occurring—for any program execution, we assume that $\mathcal{A}$ is defined, and any authorization hook check $\mathsf{hook}(\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c)$ will block unless $(\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c) \in \mathcal{A}$, as specified by the operational semantics rule *Hook* defined in Figure 3.

E. An Example

Here is an example illustrating how authorization hooks prevent information flow violations, and how Λ_{hook} can model the application setting. The following code snippet creates a *socket* object and a filehandle *fh* object. The former includes a `public` communication channel and includes a *port* field. The latter includes the filename `passwds` and the owner `root` of the file. We will assume that $(\mathtt{http}, \mathtt{passwds}, \mathtt{read}) \notin \mathcal{A}$, i.e. the password file should not be readable by a public `http` channel. Also we assume that $\mathcal{K}_{pol}(\mathtt{public}) = \bot$, i.e. the `public` principal is allowed to only have the lowest level of knowledge, whereas $\mathcal{K}_{pol}(\mathtt{root}) = \top$.

$$\begin{array}{l}
\mathsf{let}\, \mathit{socket} = \{\mathit{data} = \langle \mathtt{public} \rangle; \mathit{port} = \mathtt{http}\}\, \mathsf{in} \\
\mathsf{let}\, \mathit{fh} = \{\mathit{data} = \langle \mathtt{root} \rangle; \mathit{fname} = \mathtt{passwds}\}\, \mathsf{in} \\
\mathsf{let}\, \mathit{data} = \mathit{fh}.\mathit{data}.\mathsf{read}()\, \mathsf{in}\, \mathit{socket}.\mathit{data}.\mathsf{write}(\mathit{data})
\end{array}$$

This code has an information flow violation since, letting e be the above snippet, computation will lead to the point where a value n read from the `passwds` file, which is a channel to/from `root` and hence tainted with $\mathcal{K}_{pol}(\mathtt{root}) = \top$, and provided as a value to the `public` socket write:

$$\mathcal{K}_{pol}, e \to^* \mathcal{K}_{pol}, \{\mathit{data} = \langle \mathtt{public} \rangle; \mathit{port} = \mathtt{http}\}.\mathit{data}.\mathsf{write}(n \cdot \mathcal{K}_{pol}(\mathtt{root}))$$

And thus the full terminating computation of this example is $\mathcal{K}_{pol}, e \to^* \mathcal{K}, (\,)$ where $\mathcal{K}(\mathtt{public}) = \top$, which violates the information flow policy. However, by inserting the appropriate authorization hook before the file read of data that would be destined for writing to the `public` channel, we can block offending computation:

$$\begin{array}{llr}
\alpha \in \mathcal{V}_{\mathit{PC}}, \gamma \in \mathcal{V}_{\mathit{Subj}}, t \in \mathcal{V}_{\mathit{Type}}, h \in \mathcal{V}_{\mathit{Eff}} & & \textit{type variables} \\
\beta \in \mathcal{V}_{\mathit{PC}} \cup \mathcal{V}_{\mathit{Subj}} \cup \mathcal{V}_{\mathit{Type}} \cup \mathcal{V}_{\mathit{Eff}} & & \textit{scheme variables} \\
\varsigma \ ::= & \mathtt{s} \mid \gamma & \textit{subjects in types} \\
\kappa \ ::= & \chi \mid \bar{\mathbf{c}} & \textit{constants in types} \\
\mathit{ev} \ ::= & \ell \rightsquigarrow \varsigma \mid \varsigma \rightsquigarrow \alpha \mid \mathit{check}(\kappa, \kappa, \kappa) & \textit{events in types} \\
\ell \ ::= & \alpha \mid p \mid \ell \sqcup \ell \mid \ell \sqcap_p & \textit{program labels} \\
H \ ::= & \epsilon \mid h \mid \mathit{ev} \mid H; H \mid H|H \mid \mu h.H & \textit{trace effects} \\
\tau \ ::= & t \mid \kappa \mid \{f : \phi; \ldots; f : \phi\} \mid \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid \phi \to \phi, H & \textit{types} \\
\phi \ ::= & \tau^{\ell} & \textit{labeled types} \\
\sigma \ ::= & \forall \bar{\beta}.\tau & \textit{type schemes} \\
\Gamma \ ::= & \varnothing \mid \Gamma; x : \sigma & \textit{type environments}
\end{array}$$

Fig. 4. Λ_{hook} type syntax

$$\begin{array}{l}
\mathsf{let}\, \mathit{socket} = \{\mathit{data} = \langle \mathtt{public} \rangle; \mathit{port} = \mathtt{http}\}\, \mathsf{in} \\
\mathsf{let}\, \mathit{fh} = \{\mathit{data} = \langle \mathtt{root} \rangle; \mathit{fname} = \mathtt{passwds}\}\, \mathsf{in} \\
(\mathsf{hook}(\mathit{socket}.\mathit{port}, \mathit{fh}.\mathit{fname}, \mathtt{read}); \\
\;\mathsf{let}\, \mathit{data} = \mathit{fh}.\mathit{data}.\mathsf{read}()\, \mathsf{in}\, \mathit{socket}.\mathit{data}.\mathsf{write}(\mathit{data}))
\end{array}$$

Note that in this case, letting e' be the above snippet, the full terminating computation of e' is:

$$\mathcal{K}_{pol}, e' \to^* \mathcal{K}_{pol}, E[\mathsf{hook}(\mathtt{http}, \mathtt{passwds}, \mathtt{read})]]$$

where E is the continuation of computation containing the socket write. Computation terminates here because the authorization hook check fails, so subject knowledge does not change during execution of this instrumented expression, and the information flow policy is not violated.

III. Typing

The type theory we present here is based on the type system in [32], where a sound and complete inference algorithm is presented. Since core elements of inference have already been studied, in the current presentation we focus on a logical typing specification for simplicity.

In the Λ_{hook} type system, types reflect security levels in programs and type judgements reflect the consequences of channel interactions. We define a type system with information flow features reminiscent of standard systems, where types τ are endowed with *labels* ℓ that reflect the security level at various program points, and may be either lattice elements p or abstract α. In addition, our type system generates a trace effect H as an artifact of any typing, that predicts channel interaction events in a temporal manner. As we show in Section IV, this prediction of events can be used to predict possible evolution of knowledge during program execution. The metatheory for our type theory combined with this interpretation is presented in Section IV-B.

For example, returning to the code introduced in II-E, our static analysis generates the following trace of events for the first code snippet: $\mathtt{root} \rightsquigarrow \alpha; \alpha \rightsquigarrow \mathtt{public}$. This sequence says that knowledge from `root` flows into the program point α (associated with the variable data), and then knowledge from point α flows to `public`. The analysis generates the following trace of events for the second, instrumented code sequence: $\texttt{root} \rightsquigarrow \alpha; \mathit{check}(\texttt{http}, \texttt{passwds}, \texttt{read}); \alpha \rightsquigarrow \texttt{public}$. In our interpretation of events (Section IV), the hook event will block subsequent events (simulating the operational semantics). Thus, in the first case our system will predict an information flow violation, whereas the second case is correctly verified as safe.

A. Trace Effects

Our approach to static analysis of information flow in programs is to treat communication with subjects and authorization hooks as *events*, to approximate the sequence of events that a program can produce via a labeled transition system (LTS), called *trace effects*, and subsequently define an interpretation of these events that simulates their effects at run-time. We will use a type system to reconstruct trace effects, which constitute the approximation.

In essence, trace effects H conservatively approximate traces θ that may develop during execution, by representing a set of traces containing at least θ. Trace effects are generated by the grammar defined in Figure 4. A trace effect may be the empty effect ϵ, an effect variable h, an event ev (the definition of which we specialize for Λ_{hook} as discussed below), a sequencing of trace effects $H_1; H_2$, a nondeterministic choice of trace effects $H_1|H_2$, or a recursively bound trace effect $\mu h.H$ that finitely represents the set of possibly infinite traces that may be generated by recursive functions. Noting that the syntax of traces θ is the same as sequenced, variable-free trace effects, we abuse syntax and let θ also range over sequenced, variable-free trace effects, interpreting traces θ as the identical trace effect.

Trace effects denote sets of traces. More precisely, we define an LTS interpretation of trace effects as sets of strings over the alphabet of events plus a $\downarrow$ symbol to denote termination; abusing terminology, we also call these strings traces. Traces may be infinite, because programs may not terminate.

Definition 3.1: We write ϑ to denote possibly $\downarrow$ terminated strings over the alphabet of events:

$$\theta ::= \mathit{ev} \mid \epsilon \mid \theta\,\theta \qquad\qquad \vartheta ::= \theta \mid \theta{\downarrow}$$

We endow strings with an equational theory to interpret ϵ as the empty string and string concatenation as usual:

$$\theta\,\epsilon = \theta \qquad\qquad \epsilon\,\theta = \theta \qquad\qquad (\theta_1\theta_2)\theta_3 = \theta_1(\theta_2\theta_3)$$

The symbol Θ is defined to range over prefix-closed sets of traces.

Trace effects generate traces by viewing effects as programs in a simple nondeterministic transition system.

Definition 3.2: The trace effect transition relation on closed traces effects is defined as follows:

$$\mathit{ev} \xrightarrow{\mathit{ev}} \epsilon \qquad\qquad H_1|H_2 \xrightarrow{\epsilon} H_1 \qquad\qquad H_1|H_2 \xrightarrow{\epsilon} H_2$$
$$\mu h.H \xrightarrow{\epsilon} H[\mu h.H/h] \qquad\qquad \epsilon; H \xrightarrow{\epsilon} H$$
$$H_1; H_2 \xrightarrow{\theta} H_1'; H_2 \quad \text{if } H_1 \xrightarrow{\theta} H_1'$$

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

Definition 3.3: The interpretation of trace effects is defined as follows:

$$[\![H]\!] = \{\theta_1 \cdots \theta_n \mid H \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_n} H'\} \cup \{\theta_1 \cdots \theta_n{\downarrow} \mid H \xrightarrow{\theta_1} \cdots \xrightarrow{\theta_n} \epsilon\}$$

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

Note that prefix closure does not cause any loss of information, since the postpending of $\downarrow$ to terminating traces allows them to be distinguished from their prefixes. In particular, this means that $(H_1; H_2) \neq H_1$ for arbitrary closed H_1 and $H_2 \neq \epsilon$.

Equivalence of trace effects is defined via their interpretation, i.e. $H_1 = H_2$ iff $[\![H_1]\!] = [\![H_2]\!]$. This relation is in fact undecidable: traces are equivalent to Basic Process Algebras (BPAs), as demonstrated in [32], and equivalence of BPAs is known to be undecidable [3]. However it has been shown in previous work [32] that a decidable fragment is sufficient for type reconstruction of core elements of the system we present here. In the remainder of the paper we write $H \sqsubseteq H'$ iff $[\![H]\!] \subseteq [\![H']\!]$, and consider trace effects as equivalent up to equivalence of their interpretations.

B. Type Syntax

For simplicity we assign the label $\bot$ to all program constants in the type system, so that data communicated over channels is the only way to affect security levels of program values. Labels can also be joins (represented syntactically) of security levels $\ell_1 \sqcup \ell_2$ e.g. to support propagation of information from both operands in operations such as $+$, as well as the meet of a label ℓ and a concrete security level p at declassification (represented syntactically). In order to generalize over the security level of values that can be passed to channels, and to establish placeholders for program points where information from subject can flow into programs, we also allow labels to be variables α. We let ϕ range over labeled types.

Types include function types of the form $(\phi_1 \to \phi_2, H)$ that include a trace effect H that approximates the events that can occur upon invocation. Events include the forms $\ell \rightsquigarrow \mathtt{s}$ and $\mathtt{s} \rightsquigarrow \ell$, which in the former case indicate a communication of information at level ℓ to the subject $\mathtt{s}$, and in the latter case indicate a communication of information from subject $\mathtt{s}$ to the program point ℓ. Record types are lists of labeled types indexed by fields. The types of channels are record types with read and write fields– the type of field read reflects flow of information to the relevant subject, and that of write reflects flow to the program from the subject. In types, the subject $\mathtt{s}$ can be a variable γ, allowing generalization and instantiation. Accordingly,a basic *channel type* abbreviation can be defined as follows:

$$\mathit{chan}[\varsigma, \ell_1, \ell_2, \ell_3] \triangleq \begin{array}{l} \{\mathrm{read} : (\mathit{unit}^{\bot} \to \mathit{int}^{\ell_1}, \varsigma \rightsquigarrow \ell_1)^{\bot}, \\ \;\;\mathrm{write} : (\mathit{int}^{\ell_2} \to \mathit{unit}^{\bot}, \ell_2 \rightsquigarrow \varsigma)^{\bot}\}^{\ell_3} \end{array}$$

Note also in these types that ℓ_1, ℓ_2, and ℓ_3 can be generalized as variables α allowing channel reads/writes of the same first-

Var
$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma \cdot \epsilon}$$

Unit
$$\Gamma \vdash () : \mathit{unit}^{\bot} \cdot \epsilon$$

Bool
$$\Gamma \vdash b : \mathit{bool}^{\bot} \cdot \epsilon$$

Int
$$\Gamma \vdash n : \mathit{int}^{\bot} \cdot \epsilon$$

TaintInt
$$\Gamma \vdash n \cdot p : \mathit{int}^{p} \cdot \epsilon$$

Chan
$$\Gamma \vdash \langle \mathtt{s} \rangle : chan[\mathtt{s}, \alpha_1, \alpha_2, \bot] \cdot \epsilon$$

If
$$\frac{\Gamma \vdash e_1 : \mathit{bool}^{\ell_1} \cdot H_1 \qquad \Gamma \vdash e_2 : \tau^{\ell_2} \cdot H_2 \qquad \Gamma \vdash e_3 : \tau^{\ell_2} \cdot H_3}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau^{\ell_2} \cdot H_1; (H_2 | H_3)}$$

Field
$$\frac{\Gamma \vdash e : \{\bar{f} : \bar{\tau^{\ell}}\}^{\ell} \cdot H \qquad f_i \in \bar{f}}{\Gamma \vdash e.f_i : \tau_i^{\ell_i} \cdot H}$$

Prop
$$\frac{\Gamma \vdash e_1 : \mathit{int}^{\ell} \cdot H_1 \qquad \Gamma \vdash e_2 : \mathit{int}^{\ell} \cdot H_2}{\Gamma \vdash e_1 + e_2 : \mathit{int}^{\ell} \cdot H_1; H_2}$$

Fix
$$\frac{\Gamma; x : \phi_1; z : (\phi_1 \to \phi_2, h)^{\bot} \vdash e : \phi_2 \cdot H}{\Gamma \vdash \lambda_z x.e : (\phi_1 \to \phi_2, \mu h.H)^{\bot} \cdot \epsilon}$$

App
$$\frac{\Gamma \vdash e_1 : (\phi \to \tau^{\ell_1}, H)^{\ell_0} \cdot H_1 \qquad \Gamma \vdash e_2 : \phi \cdot H_2}{\Gamma \vdash e_1 e_2 : \tau^{\ell_1} \cdot H_1; H_2; H}$$

Declassify
$$\frac{\Gamma \vdash e : \tau^{\ell} \cdot H}{\Gamma \vdash e\downarrow_p : \tau^{\ell \sqcap_p} \cdot H}$$

Const
$$\Gamma \vdash \mathbf{c} : \mathbf{c}^{\bot} \cdot \epsilon$$

Hook
$$\frac{\Gamma \vdash e_1 : \kappa_1^{\ell_1} \cdot H_1 \qquad \Gamma \vdash e_2 : \kappa_2^{\ell_2} \cdot H_2 \qquad \Gamma \vdash e_3 : \kappa_3^{\ell_3} \cdot H_3}{\Gamma \vdash \mathsf{hook}(e_1, e_2, e_3) : \mathit{unit}^{\bot} \cdot H_1; H_2; H_3; \mathit{check}(\kappa_1, \kappa_2, \kappa_3)}$$

Record
$$\frac{\Gamma \vdash e_1 : \phi_1 \cdot H_1 \quad \cdots \quad \Gamma \vdash e_n : \phi_n \cdot H_n}{\Gamma \vdash \{f_1 = e_1; \ldots; f_n = e_n\} : \qquad \{f_1 : \phi_1; \ldots; f_n : \phi_n\}^{\bot} \cdot H_1; \ldots; H_n}$$

Let
$$\frac{\Gamma \vdash v : \sigma \cdot \epsilon \qquad \Gamma; x : \sigma \vdash e : \phi \cdot H}{\Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \phi \cdot H}$$

Sub
$$\frac{\Gamma \vdash e : \phi_0 \cdot H \qquad \phi_0 \leq \phi_1}{\Gamma \vdash e : \phi_1 \cdot H}$$

∀-Intro
$$\frac{\Gamma, C \vdash_{\varnothing} v : \phi \cdot \epsilon \qquad \bar{\beta} \# \mathrm{fv}(\Gamma)}{\Gamma, C \vdash_{\varnothing} v : \forall \bar{\beta}.\phi \cdot \epsilon}$$

∀-Elim
$$\frac{\Gamma, C \vdash_{\varnothing} v : \forall \bar{\beta}.\phi \cdot \epsilon}{\Gamma, C \vdash_{\varnothing} v : \phi[\bar{\phi}/\bar{\beta}] \cdot \epsilon}$$

$$\phi \leq \phi \qquad \frac{\phi_0 \leq \phi_1 \qquad \phi_1 \leq \phi_2}{\phi_0 \leq \phi_2} \qquad \tau^{\ell_1} \leq \tau^{\ell_1 \sqcup \ell_2} \qquad \bar{\mathbf{c}}^{\ell} \leq \bar{\mathbf{c}}\mathbf{c}^{\ell} \qquad \tau^{\ell_2} \leq \tau^{\ell_1 \sqcup \ell_2} \qquad \frac{\bar{\phi_1} \leq \bar{\phi_2}}{\{\bar{f} : \bar{\phi_1}\}^{\ell} \leq \{\bar{f} : \bar{\phi_2}\}^{\ell}}$$

$$\frac{\phi_0' \leq \phi_0 \qquad \phi_1 \leq \phi_1' \qquad H \sqsubseteq H'}{(\phi_0 \to \phi_1, H)^{\ell} \leq (\phi_0' \to \phi_1', H')^{\ell}}$$

Fig. 5. Type Derivation (top) and Subtyping (bottom) Rules

class channel value to be "spliced in" to various program points via type instantiation.

C. Constants and Authorization Hook Types

To accurately type constants $\mathbf{c}$, we define *constants in types* κ which are either abstract χ or lists of constants– this accommodates either parametric or subtyping polymorphism. For example, we can assign the type $\mathbf{c}_1$ to the value $\mathbf{c}_1$, and the type $\mathbf{c}_2$ to the value $\mathbf{c}_2$, and the type $\mathbf{c}_1\mathbf{c}_2$ to either of these values and hence expressions of the form $\mathsf{if}\, e\, \mathsf{then}\, \mathbf{c}_1\, \mathsf{else}\, \mathbf{c}_2$. Based on this constant type form, we include an event form $check(\kappa, \kappa, \kappa)$ to record authorization hooks in trace effects.

D. Type Judgements

Type judgements are of the form $\Gamma \vdash e : \sigma \cdot H$, where Γ is a *type environment* mapping free expression variables in e to types, σ is a type scheme, and H is the top-level trace effect approximation the events that can occur during computation of e. We include monomorphic ϕ in type judgements when σ is a monomorphic (non-generalized) type scheme. The type derivation rules are given in Figure 5.

A main point to note is that rule App for function application will shift the trace effect annotation on the applied function to the top-level trace effect. For Λ_{hook}-specific constructs, we observe that due to the definition of channel types $chan[\cdot]$ noted above, the effects of read-ing and write-ing will be reflected via the normal typing machinery given the Field rule which is defined generally for record types with field names f, so these operations do not need their own rules. The TaintInt rule allows us to reflect the security levels of tainted values in integers, and the Declassify rule where we record the meet of the declassified expression and the given concrete security level p in the result type label.

a) Parametric and Subtyping Polymorphism: To support parametric let-polymorphism, we include ∀-Intro and ∀-Elim rules, and allow generalization over any sort of variables– including type variables t, label variables α, subject variables γ, and trace effect variables h. In type instantiation we require consistent substitution of type forms for variable sorts.

We also support subtyping via a subsumption rule and a subtyping relation $\leq$ defined in Figure 5. Subtyping allows weakening of trace effects on function types, depth subtyping of records, subtyping of constants, and allows us to weaken labels on types by joining with other labels. This latter ability of subtyping allows us to merge the labels of conditional branches, for example we could have:

$$\varnothing \vdash (\mathsf{if\,true\,then}\,0 \cdot p_1\,\mathsf{else}\,1 \cdot p_2) : \mathit{int}^{p_1 \sqcup p_2} \cdot \epsilon$$

The rules for record and constant subtyping, as well as $\forall$-INTRO and $\forall$-ELIM and other notation later in this paper, uses vector notation defined as follows.

Definition 3.4: Vectors, denoted $\bar{x}$, range over sequences of distinct elements $x_1 \cdots x_n$, with the empty vector denoted $\varnothing$ and singleton vector denoted x. (In this definition we use x to denote arbitrary elements, not just expression variables.) Vectors are equivalent up to reordering and we assume their elements are unique, hence, we may treat vectors $x_1 \cdots x_n$ as analogous to sets $\{x_1, \ldots, x_n\}$, in particular adapting notation $x \in \bar{x}$, $\bar{x}_1 \cup \bar{x}_2$, $\bar{x}_1 \cap \bar{x}_2$, and $\bar{x}_1 \times \bar{x}_2$ with the obvious meaning. We write $\bar{x}_1 \# \bar{x}_2$ iff $\bar{x}_1 \cap \bar{x}_2 = \varnothing$. We write $\bar{x}_1 \bar{x}_2$ to denote the vector $\bar{x}_1 \cup \bar{x}_2$ where $\bar{x}_1 \# \bar{x}_2$.

E. An Example

As a more complex example illustrating many of our type system features, consider the term $\lambda x.\lambda c.(c.\mathsf{write}(x))$, which has type:

$$\forall \alpha_1, \alpha_2, \alpha_3, \gamma, h.$$
$$(\mathit{int}^{\alpha_2} \to (\mathit{chan}[\alpha_1, \alpha_2, \alpha_3, \gamma] \to \mathit{unit}^{\perp}, \alpha_2 \rightsquigarrow \gamma)^{\perp}, \epsilon)^{\perp}$$

and thus the following typing can be assigned to a program that results in writing a value n tainted at level p to a channel that communicates with subject s:

$$\varnothing \vdash (\lambda x.\lambda c.(c.\mathsf{write}(x)))\,(n \cdot p)\,\langle \mathsf{s} \rangle : \mathit{unit}^{\perp} \cdot p \rightsquigarrow \mathsf{s}$$

Note in particular that the top-level trace effect $p \rightsquigarrow \mathsf{s}$ in this case represents the flow of information that will occur upon execution to the concrete subject s (not an abstract γ). In general, because top-level types are closed, these events will always be concrete in traces.

IV. INTERPRETATION OF TRACES AND METATHEORY

In this section we define an interpretation of the traces that approximate possible program executions. This interpretation is logical, not algorithmic, since trace effects as defined in Section III-A may be nonterminating and generate an infinite number of possible traces. However this interpretation is adequate to formulate the main formal properties of our analysis, in particular Theorem 4.5. In Section V, we will show how this interpretation can be implemented via abstract interpretation.

Consider the following program, assuming a simple security lattice with just top and bottom elements hi and lo, and $\mathcal{K}_{pol}(\mathsf{s}_1) = \mathsf{lo}$, $\mathcal{K}_{pol}(\mathsf{s}_2) = \mathsf{lo}$, and $\mathcal{K}_{pol}(\mathsf{s}_3) = \mathsf{hi}$:

$$\begin{array}{l} \mathsf{let}\, c_1 = \langle \mathsf{s}_1 \rangle\, \mathsf{in} \\ \mathsf{let}\, c_2 = \langle \mathsf{s}_2 \rangle\, \mathsf{in} \\ \mathsf{let}\, c_3 = \langle \mathsf{s}_3 \rangle\, \mathsf{in} \\ c_1.\mathsf{write}(c_2.\mathsf{read}()); c_2.\mathsf{write}(c_3.\mathsf{read}()) \end{array}$$

The level of knowledge $\mathcal{K}$ after execution of the above program has $\mathcal{K}(\mathsf{s}_2) = \mathsf{hi}$ but $\mathcal{K}(\mathsf{s}_1) = \mathsf{lo}$, since communication from s_2 to s_1 happens before communication from s_3 to s_2. The temporality of these events is reflected in the following top-level trace effect of the above program:

$$\mathsf{s}_2 \rightsquigarrow \alpha_1; \alpha_1 \rightsquigarrow \mathsf{s}_1; \mathsf{s}_3 \rightsquigarrow \alpha_2; \alpha_2 \rightsquigarrow \mathsf{s}_2$$

For precision we need to define an interpretation of these events that preserves temporality. If the analysis is not flow-sensitive, then it would predict $\mathcal{K}(\mathsf{s}_1) = \mathsf{hi}$ for the above program, which introduces false positives. Not only that, but it would not be possible to detect when an authorization hook placement prevents unsafe information flows by blocking them, which is the main goal of our analysis. Our approach will essentially be to treat event traces as simulations of run-time events, that accrue changes to knowledge in an ordered manner via a transition relation.

A. Traces as Knowledge Transition Machines

To interpret traces generated by trace effects, we give a more refined definition of events and traces, where the latter are really just lists of events and events have concrete subjects in them– this is guaranteed by the type system:

$$\mathit{ev} ::= \ell \rightsquigarrow \mathsf{s} \mid \mathsf{s} \rightsquigarrow \alpha \mid \mathit{check}(\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3) \qquad \theta ::= \epsilon \mid \mathit{ev}\,\theta$$

We introduce the following shorthand for instances in which the types of constants are approximated as vectors of singleton types with the following abbreviation:

$$\mathit{check}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2, \bar{\mathbf{c}}_3) \triangleq \mathit{check}(x_1) \mid \cdots \mid \mathit{check}(x_n)$$

where $x_1, \cdots x_n = \bar{\mathbf{c}}_1 \times \bar{\mathbf{c}}_2 \times \bar{\mathbf{c}}_3$.

We view traces themselves as a state transition machine, where events simulate the effect of events upon subject knowledge at run time, as well as the blocking effect of failing authorization hooks. To keep track of the level of label variables α, we extend $\mathcal{K}$ to be a mapping from label variables α to lattice elements p, and write $\mathcal{K}(p)$ to mean p and $\mathcal{K}(\ell_1 \sqcup \ell_2)$ and $\mathcal{K}(\ell \sqcap_p)$ to mean inductively $\mathcal{K}(\ell_1) \vee \mathcal{K}(\ell_2)$ and $\mathcal{K}(\ell) \wedge p$. The state transition relation $\to$ is defined as the least binary relation on pairs $(\mathcal{K}, \theta)$ satisfying the following rules:

$$\begin{array}{rcl} \mathcal{K}, (p \rightsquigarrow \mathsf{s})\theta & \to & \mathcal{K}[\mathsf{s} \mapsto \mathcal{K}(\mathsf{s}) \vee \mathcal{K}(p)], \theta \\ \mathcal{K}, (\mathsf{s} \rightsquigarrow \alpha)\theta & \to & \mathcal{K}[\alpha \mapsto \mathcal{K}(\mathsf{s}) \vee \mathcal{K}(\alpha)], \theta \\ \mathcal{K}, ((\mathit{check}(\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c))\theta) & \to & (\mathcal{K}, \theta) \quad \text{if } (\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c) \in \mathcal{A} \\ \mathcal{K}, ((\mathit{check}(\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c))\theta) & \to & (\mathcal{K}, \epsilon) \quad \text{if } (\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c) \notin \mathcal{A} \end{array}$$

Notation $\to^*$ is the Kleene closure of $\to$. Now we can specify what we mean when we say that a trace effect predicts knowledge:

Definition 4.1: We write $\mathcal{K}_1, \theta \vdash \mathcal{K}_2$ iff $\mathcal{K}_1, \theta \to^* \mathcal{K}_2, \epsilon$. We write $\mathcal{K}_1, H \vdash \mathcal{K}_2$ iff there exists $\theta \in [\![H]\!]$ with $\mathcal{K}_1, \theta \vdash \mathcal{K}_2$

B. Properties of the Analysis

The following theorems formally characterize the most important properties of our system. We take a standard strategy of proving a subject reduction result. The following definition captures the preserved invariant in subject reduction, which intuitively says that as a term reduces, its type stays the same, and its trace effect and label can only become more refined in

$$\mathcal{F}\llbracket\_\rrbracket \in \text{eff} \to (\mathcal{V}_{\text{eff}} \rightharpoonup \wp(\text{trace})) \to \wp(\text{trace})$$

$$\mathcal{F}\llbracket\epsilon\rrbracket(\rho) \triangleq \{\epsilon\} \quad \mathcal{F}\llbracket ev\rrbracket(\rho) \triangleq \{ev\} \quad \mathcal{F}\llbracket h\rrbracket(\rho) \triangleq \rho(h)$$

$$\begin{aligned}
\mathcal{F}\llbracket H_1 \mid H_2\rrbracket(\rho) &\triangleq \mathcal{F}\llbracket H_1\rrbracket(\rho) \cup \mathcal{F}\llbracket H_2\rrbracket(\rho) \\
\mathcal{F}\llbracket H_1 \mathbin{;} H_2\rrbracket(\rho) &\triangleq \mathcal{F}\llbracket H_1\rrbracket(\rho) \cup \{\theta_1\theta_2 \mid \theta_1{\downarrow} \in \mathcal{F}\llbracket H_1\rrbracket(\rho), \\
&\qquad\qquad \theta_2 \in \mathcal{F}\llbracket H_2\rrbracket(\rho)\} \\
\mathcal{F}\llbracket \mu h.H\rrbracket(\rho) &\triangleq \mu X.\ \mathcal{F}\llbracket H\rrbracket(\rho[h \mapsto X])
\end{aligned}$$

Fig. 6. Compositional Concrete Semantics of Trace Effects

their approximation of future events and the security level of the term.

Definition 4.2: We write $\mathcal{K}_2 \preccurlyeq \mathcal{K}_1$ iff for all $\ell \in \text{dom}(\mathcal{K}_2)$, $\mathcal{K}_2(\ell) \preccurlyeq \mathcal{K}_1(\ell)$. We write $\mathcal{K}_2, H_2, \ell_2 \leq \mathcal{K}_1, H_1, \ell_1$ iff for all $\theta_2 \in \llbracket H_2 \rrbracket$ there exists $\theta_1 \in \llbracket H_1 \rrbracket$ such that:

$$\begin{aligned}
&\forall \mathcal{K}.\mathcal{K}_2, \theta_2 \vdash \mathcal{K} \Rightarrow \\
&\quad \exists \mathcal{K}'.(\mathcal{K}_1, \theta_1 \vdash \mathcal{K}') \wedge \mathcal{K} \preccurlyeq \mathcal{K}' \wedge \mathcal{K}(\ell_2) \preccurlyeq \mathcal{K}'(\ell_1)
\end{aligned}$$

Due to similarities in the type system, we are able to leverage several key results in [32] to prove subject reduction. These include normalization of type derivations which is necessary in light of the non-syntax directed typing rules, and treatment of CONTEXT reduction.

Theorem 4.3 (Subject Reduction): If $\varnothing \vdash e : \tau^{\ell} \cdot H$ and $\mathcal{K}, e \to \mathcal{K}', e'$ then $\varnothing \vdash e' : \tau^{\ell'} \cdot H'$ for some H' and ℓ' where $\mathcal{K}', H', \ell' \leq \mathcal{K}, H, \ell$.

Proof. By induction on the normalized derivation of $\varnothing \vdash e : \tau^{\ell_1} \cdot H$ and case analysis on e. On the basis of the subject reduction invariant, we can now show that changes in knowledge are predicted in a sound manner by our interpretation of trace effects.

Theorem 4.4 (Prediction of Knowledge): If $\varnothing \vdash e : \tau^{\ell} \cdot H$ and $\mathcal{K}_{pol}, e \to^* \mathcal{K}, e'$, then there exists $\mathcal{K}'$ where $\mathcal{K}_{pol}, H \vdash \mathcal{K}'$ and $\mathcal{K} \preccurlyeq \mathcal{K}'$.

Proof. By Theorem 4.3, Definition 4.2, and induction on the length of the reduction, if $\varnothing \vdash e : \tau^{\ell} \cdot H$ and $\mathcal{K}_{pol}, e \to^* \mathcal{K}, e'$, then there exists H' such that for all $\theta_1 \in \llbracket H' \rrbracket$ with $\mathcal{K}, \theta_1 \vdash \mathcal{K}_1$ there exists $\theta_2 \in \llbracket H \rrbracket$ such that $\mathcal{K}_{pol}, \theta_2 \vdash \mathcal{K}_2$ where $\mathcal{K}_1 \preccurlyeq \mathcal{K}_2$. In case $\theta_1 = \epsilon$, $\mathcal{K}_1$ is $\mathcal{K}$ and the result follows. □

A main result, that our analysis allows us to safely approximate information flow violations, is stated as follows.

Theorem 4.5 (Prediction of Flow Violations): If e has an information flow violation and $\varnothing \vdash e : \tau^{\ell} \cdot H$ then $\mathcal{K}_{pol}, H \vdash \mathcal{K}$ with $\mathcal{K}(\mathtt{s}) \not\preccurlyeq \mathcal{K}_{pol}(\mathtt{s})$ for some $\mathtt{s}$.

V. Abstract Interpretation of Trace Effects

As mentioned in Section III, a key component of our metatheory is that information flow violations do not occur if (1) the program is well typed with trace effect H, and (2) the trace effect H does not predict any invalid information flows. If an invalid information flow could occur, it will be the case that $\mathcal{K}_{\text{pol}}, H \vdash \mathcal{K}$ and $\mathcal{K}(\mathtt{s}) \not\preccurlyeq \mathcal{K}_{\text{pol}}(\mathtt{s})$ for policy $\mathcal{K}_{\text{pol}}$, possible resulting knowledge after running the program $\mathcal{K}$, and subject who learned more than allowed $\mathtt{s}$. In this section we describe a static analysis of trace effects which computes an over-approximation of all $\mathcal{K}$ which could result from $\mathcal{K}_{\text{pol}}, H \vdash \mathcal{K}$, and hence an over-approximation of resulting knowledge $\mathcal{K}(\mathtt{s})$ for any subject $\mathtt{s}$. The combination of type checking from Section III and static analysis from this section provide a guarantee that undesirable information flows cannot occur in any execution of the program. The formal guarantee we give is established at the end of this section as Theorem 5.4. Our algorithm is efficient, and has the added appeal of supporting helpful user interactions as discussed in Section VI-A.

Our approach to static analysis of trace effects is grounded in the tradition of *abstract interpretation* [5, 6, 7] which considers a "concrete semantics" for which computing the desired property is either inefficient or uncomputable, and proceeds through design of an "abstract semantics" for which the property is efficiently computable. To connect the abstract semantics and property to the concrete, a mapping between the domains is formed, called a Galois connection.

Following this recipe, we first present the concrete semantics of trace effects in the form of a compositional denotation function $\mathcal{F}\llbracket\_\rrbracket$ and prove it equivalent to the simpler labeled transition system $H \xrightarrow{\theta} H$ which is derived from prior work [31, 32] and described in Section III. This re-structuring is done to aid the design and proof of soundness of the abstract interpreter, which follows an analogous structure. Next, we design a finite abstract domain for traces $\text{trace}^{\sharp}$ which forms a Galois connection $\langle \dot{\alpha}, \dot{\gamma} \rangle$ with sets of concrete traces $\wp(\text{trace})$, and an abstract semantics for trace effects $\mathcal{F}^{\sharp}\llbracket\_\rrbracket$ which is sound w.r.t. the compositional concrete semantics $\mathcal{F}\llbracket\_\rrbracket$.

A. Concrete Semantics

We give a big-step account of trace effect semantics (w.r.t. the "ground truth" small-step system described in Section III) as a map from trace effects H directly into prefix-closed sets of (possibly terminating) traces, notated $\mathcal{F}\llbracket\_\rrbracket$ and shown in Figure 6. This definition uses an environment $\rho \in \mathcal{V}_{\text{eff}} \rightharpoonup \wp(\text{trace})$ which maps trace effect variables h to result sets, an important feature even for closed terms to interpret fixpoint expression $\mu h.H$.

B. Abstract Domain

The co-domain of the concrete semantics is unbounded and therefore uncomputable in general. However, we are not just interested in possible traces θ, but in the observable effects these traces have on knowledge $\mathcal{K}$. We therefore base our abstract domain on *flows between subjects*, defined as $\theta^{\sharp}$ in Figure 7, which are directly interpretable in terms of their effects on knowledge. An abstract trace is a mapping from source-sink pairs (indicating a flow from the source to the sink) to a declassification label which is associated with the flow. Note that because an abstract trace is a partial map, a single abstract trace $\theta^{\sharp}$ may contain multiple flows between distinct sources and sinks.

Abstract traces are understood formally via their abstraction mapping from concrete traces, shown as $\mathcal{E}^{\sharp}\llbracket\_\rrbracket$ also in Figure 7. In its definition, flow events between atomic entities ℓ^{o} and ℓ^{i}

$$\ell^o \in \text{source} ::= s \mid \alpha \mid p \qquad \ell^i \in \text{sink} ::= s \mid \alpha$$
$$\theta^\sharp \in \text{trace}^\sharp \triangleq \text{source} \times \text{sink} \rightharpoonup \text{security}$$

$$\mathcal{E}^\sharp\llbracket\_\rrbracket \in \text{event} \to \text{trace}^\sharp \quad \_\ltimes\_ \in \text{trace}^\sharp \times \text{trace}^\sharp \to \text{trace}^\sharp$$
$$\dot{\eta} \in \text{trace}{\to}\text{trace}^\sharp \quad \dot{\alpha} \in \wp(\text{trace}){\to}\text{trace}^\sharp \quad \dot{\gamma} \in \text{trace}^\sharp{\to}\wp(\text{trace})$$

$$\mathcal{E}^\sharp\llbracket\ell^o \rightsquigarrow \ell^i\rrbracket \triangleq \{\langle\ell^o,\ell^i\rangle \mapsto \top\}$$
$$\mathcal{E}^\sharp\llbracket\ell_1 \sqcup \ell_2 \rightsquigarrow \ell^i\rrbracket \triangleq \mathcal{E}^\sharp\llbracket\ell_1 \rightsquigarrow \ell^i\rrbracket \sqcup \mathcal{E}^\sharp\llbracket\ell_2 \rightsquigarrow \ell^i\rrbracket$$
$$\mathcal{E}^\sharp\llbracket\ell \sqcap_p\!\rightsquigarrow \ell^i\rrbracket \triangleq \{\langle\ell^o,\ell^{i\prime}\rangle \mapsto p'\sqcap p \mid \{\langle\ell^o,\ell^{i\prime}\rangle \mapsto p'\} \in \mathcal{E}^\sharp\llbracket\ell{\rightsquigarrow}\ell^i\rrbracket\}$$

$$\dot{\eta}(\epsilon) \triangleq \varnothing \quad \dot{\eta}(ev) \triangleq \mathcal{E}^\sharp\llbracket ev\rrbracket \quad \dot{\eta}(\theta_1\theta_2) \triangleq \dot{\eta}(\theta_1) \ltimes \dot{\eta}(\theta_2)$$

$$\dot{\alpha}(\Theta) \triangleq \bigsqcup_{\theta\in\Theta} \dot{\eta}(\theta) \qquad \dot{\gamma}(\theta^\sharp) \triangleq \{\theta \mid \dot{\eta}(\theta) \sqsubseteq \theta^\sharp\}$$

$$f_1 \ltimes f_2 \triangleq f_1 \sqcup f_2 \sqcup \bigsqcup_{\substack{\{\langle\ell_1,\ell_2\rangle\mapsto p_1\}\in f_1\\ \{\langle\ell_2,\ell_3\rangle\mapsto p_2\}\in f_2}} \langle\ell_1,\ell_3\rangle \mapsto p_1 \sqcap p_2$$

Fig. 7. Abstract Domain for Trace Effects

are abstracted as a singleton map from $\langle\ell^o,\ell^i\rangle$ to $\top$, indicating declassification to the top of the security lattice (that is, no declassification because $p \sqcap \top = p$ for all p). Compound flow events $\ell_1 \sqcup \ell_2 \rightsquigarrow \ell^i$ are interpreted as the (pointwise) join of decomposed events $\ell_1 \rightsquigarrow \ell^i$ and $\ell_2 \rightsquigarrow \ell^i$, and $\ell \sqcap_p\!\rightsquigarrow \ell^i$ as an extension of the interpretation of $\ell \rightsquigarrow \ell^i$ with the meet of p and any recursively appearing declassification labels p'.

Abstract traces represent *sets* of concrete traces, and we show mappings in each direction $\langle\dot{\alpha},\dot{\gamma}\rangle$ in Figure 7. The mapping functions are adjoint $\dot{\alpha}(\Theta) \sqsubseteq \theta^\sharp \iff \Theta \subseteq \dot{\gamma}(\theta^\sharp)$ and therefore form a Galois connection—these mappings are used in the statement of soundness for the abstract interpreter. Definitions of $\dot{\alpha}$ and $\dot{\gamma}$ rely on an elementwise-abstraction $\dot{\eta}$ which abstracts a single concrete trace to its most precise abstract trace. The empty trace is abstracted by the empty set of flows, a single event is abstracted by $\mathcal{E}^\sharp\llbracket\_\rrbracket$, and the sequencing of two traces is abstracted using an abstract sequencing operator $\ltimes$—we choose an asymmetric symbol to remind that the operator is not commutative. The abstract sequencing $\theta_1^\sharp \ltimes \theta_2^\sharp$ includes flows from $\theta_1^\sharp$ and $\theta_2^\sharp$ independently, as well as transitive flows $\langle\ell_1,\ell_3\rangle$ if $\langle\ell_1,\ell_2\rangle \in \theta_1^\sharp$ and $\langle\ell_2,\ell_3\rangle \in \theta_2^\sharp$ (modulo declassification).

C. Abstract Interpreter

In Figure 8 we define the abstract interpreter for computing an over-approximation of all prefix-closed concrete traces which occur in the operational semantics of a trace effect. The structure of the abstract interpreter mirrors that of the compositional concrete semantics effects in Figure 6, which simplifies the proof of soundness. The least fixed-point computation in the case of $\mu h.H$ trace effects is efficiently computable through Kleene fixed-point iteration.

$$\mathcal{F}^\sharp\llbracket\_\rrbracket \in \text{eff} \to (\mathcal{V}_{\text{eff}} \rightharpoonup \wp(\text{trace}^\sharp)) \to \wp(\text{trace}^\sharp)$$

$$\mathcal{F}^\sharp\llbracket\epsilon\rrbracket(\rho^\sharp) \triangleq \{\epsilon\} \quad \mathcal{F}^\sharp\llbracket ev\rrbracket(\rho^\sharp) \triangleq \{\mathcal{E}^\sharp\llbracket ev\rrbracket\} \quad \mathcal{F}^\sharp\llbracket h\rrbracket(\rho^\sharp) \triangleq \rho^\sharp(h)$$

$$\begin{aligned}
\mathcal{F}^\sharp\llbracket H_1 \mid H_2\rrbracket(\rho^\sharp) &\triangleq \mathcal{F}^\sharp\llbracket H_1\rrbracket(\rho^\sharp) \cup \mathcal{F}^\sharp\llbracket H_2\rrbracket(\rho^\sharp)\\
\mathcal{F}^\sharp\llbracket H_1 \,;\, H_2\rrbracket(\rho^\sharp) &\triangleq \mathcal{F}^\sharp\llbracket H_1\rrbracket(\rho^\sharp)\{\theta_1^\sharp{\ltimes}\theta_2^\sharp \mid \theta_1^\sharp{\in}\mathcal{F}^\sharp\llbracket H_1\rrbracket(\rho^\sharp),\\
&\qquad\qquad\qquad\qquad\qquad \theta_2^\sharp{\in}\mathcal{F}^\sharp\llbracket H_2\rrbracket(\rho^\sharp)\}\\
\mathcal{F}^\sharp\llbracket \mu h.H\rrbracket(\rho^\sharp) &\triangleq \mu X^\sharp.\ \mathcal{F}^\sharp\llbracket H\rrbracket(\rho^\sharp[h \mapsto X^\sharp])
\end{aligned}$$

Fig. 8. Compositional Abstract Semantics for Trace Effects

In our abstract interpreter, we add precision to the lattice of abstract traces $\text{trace}^\sharp$ by introducing a powerset of abstract traces $\wp(\text{trace}^\sharp)$. This necessitates another Galois connection sets of concrete and sets of abstract traces, which we construct as a standard (downward-closed) powerset "lifting" of the Galois connection between sets of concrete traces and individual abstract traces:

$$\tilde{\alpha}(\Theta) \triangleq \{\dot{\alpha}(\{\theta\}) \mid \theta \in \Theta\} \qquad \tilde{\gamma}(\Theta^\sharp) \triangleq \bigcup_{\theta^\sharp\in\Theta^\sharp} \dot{\gamma}(\theta^\sharp)$$

To establish soundness of the abstract interpreter w.r.t. the concrete semantics, it suffices to show that (1) $\mathcal{E}^\sharp\llbracket\_\rrbracket$ is a sound abstraction of the trace-interpretation of individual events, and (2) abstract sequencing ($\ltimes$) is a sound abstraction of concrete sequencing of traces. (1) is immediate by definition of abstraction $\dot{\alpha}$, and (2) is justified in the following lemma.

Lemma 5.1 (Abstract Sequencing Soundness): For concrete sets of traces the abstraction of sequenced traces is approximated by the abstract sequencing of abstraction of traces, that is, $\dot{\alpha}(\{\theta_1\theta_2 \mid \theta_1 \in \Theta_1, \theta_2 \in \Theta_2\}) \sqsubseteq \dot{\alpha}(\Theta_1) \ltimes \dot{\alpha}(\Theta_2)$

Proof. $\dot{\alpha}$ is homomorphic, so it suffices to demonstrate the property on singletons $\dot{\alpha}(\{\theta_1\theta_2\}) \sqsubseteq \dot{\alpha}(\{\theta_1\}) \ltimes \dot{\alpha}(\{\theta_2\})$ which is immediate by definition of $\dot{\alpha}$ and $\dot{\eta}$.

We now prove soundness of the abstract interpreter:

Theorem 5.2 (Abstract Interpreter Soundness): The abstract interpreter is an over-approximation of the concrete semantics, that is, for any trace effect H and concrete environment ρ: $\tilde{\alpha}(\mathcal{F}\llbracket H\rrbracket(\rho)) \sqsubseteq \mathcal{F}^\sharp\llbracket H\rrbracket(\tilde{\alpha} \circ \rho)$.

Proof. By induction on H and Lemma 5.1.

To bridge the soundness of the abstract interpreter to sound information flow prediction, we prove that a singleton abstract flow is a sound *under*-approximation of any program which semantically includes a flow between subjects:

Lemma 5.3 (Subject Flow Under-approximation): If an even trace results in increased knowledge for a subject, then the abstraction of that trace predicts a flow from some other subject, that is, if $\mathcal{K}_{pol}, \theta \to^* \mathcal{K}$ and $\mathcal{K}(s) \not\leq \mathcal{K}_{pol}(s)$ then there exists s' and $p \sqsupseteq \mathcal{K}(s)$ s.t. $\mathcal{K}_{pol}(s') = \mathcal{K}(s)$ and $\langle s', s\rangle \mapsto p \in \eta(\theta)$.

Proof. By induction on θ and monotonicity of $\mathcal{K}$ in knowledge semantics.

Finally, we conclude sound prediction of information flow violations as a consequence of abstract interpreter soundness, and type soundness from Section III.

Theorem 5.4 (Sound Information Flow Prediction): If an information flow violation occurs during the execution of a well-typed program, then the abstract interpretation of the trace effect will predict it, that is, if e has an information flow violation from s to s' w.r.t. policy $\mathcal{K}_{pol}$ and $\varnothing \vdash e : \tau^{\ell} \cdot H$ then there exists $p \sqsupseteq \mathcal{K}(s)$ and $\theta^{\sharp} \in \mathcal{F}^{\sharp}[\![H]\!](\varnothing)$ s.t. $\theta^{\sharp}(s', s) = p$ and $\mathcal{K}_{pol}(s') = \mathcal{K}(s)$.

Proof. By Theorem 4.5, definition of $\mathcal{K}_{pol}, H \vdash \mathcal{K}$, and Lemmas 5.2 and 5.3.

To verify a program, it therefore suffices to compute $\mathcal{F}^{\sharp}[\![H]\!](\varnothing)$ and check that the resulting set of abstract traces does not contain undesirable flows.

We have implemented our analysis in Haskell and will make it available as open source via a public GitHub repository [3]. We provide a number of examples in the repository, on all of which the analysis terminates in fractions of a second.

D. Algorithmic Complexity

The presented abstract interpreter executes in $O(S^6)$ time where S is the number of source and sink effect variables in the trace effect, which is in turn proportional size of the program from which the trace effect was generated plus the number of subjects. This bound is derived as follows: The join operator is a pointwise join computation over dictionaries linear in the size of the S, and is therefore $O(S)$. The abstract sequencing operator $\ltimes$ effectively computes the Cartesian product of its arguments using a nested loop, and is therefore $O(S^2)$. The abstract interpreter $\mathcal{F}^{\sharp}$ computes a least fixpoint which introduces a cubic factor around its inner loop, which consists of (linear) set unions and (quadratic) abstract sequencing which is ultimately $O(S^2)$. The entire fixpoint is therefore $O(S^{2^3}) = O(S^6)$. We also implement a *path-sensitive* variant of this analysis using BDDs in the representation of abstract effects which introduces a worst-case exponential factor $O(2^C)$ where C is the set of access control triples $(\mathbf{c}_s, \mathbf{c}_o, \mathbf{c}_c)$. However, in practice this is bounded by the nesting depth of access control checks in the program, which is a low constant for realistic programs.

VI. Applications in Practice

Here we consider analysis of the motivating examples introduced in Section I-A, and how our system discovers vulnerabilities in them and allows the programmer to explore possible executions based on authorization hook outcomes.

A. Missing Placements

In Figure 9, we recreate in Λ_{hook} the offending code presented in Listing 1 in Section I-A to demonstrate how our analysis applies to real application settings. In this example we posit constants `UID` and `ROOT` that identify the current user and root user ids. The records *cpid* and *rpid* are associated with the current process and root, and maintain a channel to those principles as well as identifying information. The `passwds` file is associated with the `root` subject, and has a *sigout* field that maintains a signal output channel, initially set to *rpid*.

```
let $uid = \mathtt{UID}$ in
let $cpid = \{data = \langle\mathtt{usr}\rangle;\ uname = uid\}$ in
let $rpid = \{data = \langle\mathtt{root}\rangle;\ uname = \mathtt{ROOT}\}$ in
let $passwds =$
    $\{data = \langle\mathtt{root}\rangle;\ fname = \mathtt{passwds};\ sigout = rpid\}$ in
let $setpid =$
    $\lambda f.\lambda c.\ \{data = f.data;\ fname = f.fname;\ sigout = c\}$ in
let $setlease = (\lambda f.setpid\ f\ cpid)$ in
let $dofcntl = \lambda f.\lambda cmd.$
    if $(cmd = 0)$ then $(setlease\ f)$ else
    $(\mathsf{hook}(uid, f.fname, \mathtt{setown}); setpid\ f\ cpid)$ in
let $sysfcntl = \lambda f.\lambda cmd.$
    $(\mathsf{hook}(uid, f.fname, \mathtt{fcntl}); dofcntl\ f\ cmd)$ in
let $passwds = (sysfcntl\ passwds\ 0)$ in
$passwds.sigout.data.\mathsf{write}(passwds.data.\mathsf{read}())$
```

Fig. 9. Λ_{hook} Model of Listing 1 Code

Each of the functions *setlease*, *dofcntl*, and *sysfcntl* are Λ_{hook} avatars of the `fcnt_setlease`, `do_fcntl`, and `sys_fcntl` functions defined in Listing 1, and allow resetting of the signal output channel for a file via the function *setpid*. The authorization hooks inserted in this code are faithful to the `fcntl` and `setown` hook implementations in the Linux security modules, which implicitly refer to the current user (via the user's `UID` available in the current process information) as the subject. In the last line, we read from the `passwds` file and output the result to the process channel in *sigout*, to "force the issue" of the potentially insecure information flow from the file to the signal output channel. As in Section I-A, we note that this program does have an information flow violation if $(\mathtt{usr}, \mathtt{passwds}, \mathtt{fcntl}) \in \mathcal{A}$, even if $(\mathtt{usr}, \mathtt{passwds}, \mathtt{setown}) \notin \mathcal{A}$.

The following trace effect is assigned to this program by our analysis:

$$\begin{gathered} \mathit{check}(\mathtt{UID}, \mathtt{passwds}, \mathtt{fcntl});\ \mathtt{root} \rightsquigarrow \alpha; \\ ((\alpha \rightsquigarrow \mathtt{usr}) \mid (\mathit{check}(\mathtt{UID}, \mathtt{passwds}, \mathtt{setown}); \alpha \rightsquigarrow \mathtt{usr})) \end{gathered}$$

We again assume a security lattice with top and bottom elements hi and lo, and assume $\mathcal{K}_{pol}(\mathtt{root}) = \mathsf{hi}$ and $\mathcal{K}_{pol}(\mathtt{usr}) = \mathsf{lo}$. Whether or not this trace effect predicts an information flow violation depends on whether $(\mathtt{usr}, \mathtt{passwds}, \mathtt{fcntl}) \in \mathcal{A}$—if so, a violation is predicted. Our tool reports the following flows indicated by the trace effect, in decision-tree format.

$$\begin{aligned} \mathit{check}(\mathtt{UID}, \mathtt{passwds}, \mathtt{fcntl}) :&\quad \mathtt{root} \rightsquigarrow \mathtt{usr} \\ \neg\mathit{check}(\mathtt{UID}, \mathtt{passwds}, \mathtt{fcntl}) :&\quad <\!\mathtt{none}\!> \end{aligned}$$

Note that the tool isolates the scenario which could lead to the undesirable flow—when $\mathit{check}(\mathtt{UID}, \mathtt{passwds}, \mathtt{fcntl})$ succeeds. As mentioned previously, this tool has been implemented in Haskell and will be made available in a public GitHub repository [4].

B. Control Loops and Fixpoints

The web server example discussed in Section I-B is fully realized in our model in Figure 10. The *socket* and front page file *fpage* encapsulate `external` channels, while the password file *pwds* encapsulates a `root` channel.

The following trace effect is assigned to this program by our analysis:

$$\begin{aligned} &\mu h.\ ((\mathit{check}(\mathtt{ext\_hi}, \mathtt{passwds}, \mathtt{r}); \mathtt{root} \rightsquigarrow \alpha) \mid \mathtt{external} \rightsquigarrow \alpha); \\ &\quad \alpha \rightsquigarrow \mathtt{external}; h \end{aligned}$$

[3] https://github.com/uvm-plaid/hook-ai

[4] https://github.com/uvm-plaid/hook-ai

```
let socket = {data = ⟨external⟩; port = 80} in
let pwds = {data = ⟨root⟩; fname = passwds} in
let rpwds = λauth.(hook(auth, passwds, r); pwds.data.read()) in
let fp = {data = ⟨external⟩; fname = frontpage} in
let loop = λ_z x.
    let cred = socket.data.read( ) in
    let html = if authorized(cred) then rpwds(ext_hi) else fp in
    let data = html.data.read() in
    socket.data.write(data); z x
in loop()
```

Fig. 10. Λ_{hook} Model of Section I-B Web Server Example

This trace effect has no finite traces in its interpretation, reflecting the non-terminating behavior of the control loop. Our abstract interpreter tool reports the following to the user:

$$\mathit{check}(\mathtt{ext\_hi}, \mathtt{passwds}, \mathtt{r}) : \ \mathtt{root} \rightsquigarrow \mathtt{external}$$
$$\neg\mathit{check}(\mathtt{ext\_hi}, \mathtt{passwds}, \mathtt{r}) : \ \mathtt{<none>}$$

This example illustrates important technical features as discussed in Section I-B. It demonstrates how our abstract interpretation computes a fixpoint interpretation of the given trace effect. It illustrates flow-sensitivity, by recognizing that the authorization hook occur before the external socket write, and path-sensitivity since outcomes for both the successful and failing hook conditions are reported, depending on whether $(\mathtt{ext\_hi}, \mathtt{passwds}, \mathtt{r})$ is in the local access control list $\mathcal{A}$ or not. Also, while the flow from root to external in the former case is identified and reported by the analysis, the program is not rejected, allowing the programmer to determine that this is a false positive since that flow occurs only in the context where the authorization hook check succeeds.

VII. Related Work

a) Authorization Hooks.: A variety of analysis tools for authorization hook placements have been studied in previous work. In [11, 22] authorization hook placements are generated from specifications of known security-sensitive operations to mediate control flows to those operations. In [12, 13, 25, 34], heuristics for identifying "sensitive operations" are identified, which drives authorization hook placement choices. In [26] a method is developed for sound minimization of authorization hook placements, without introducing false positives. However, these techniques are all concerned strictly with mediating control flows to enforce an access control policy, not enforcing the information flows implied by the policy via authorization hooks. In [10, 33, 36] both a static and runtime analysis are considered for checking the consistency of authorization hook placements in enforcing access control policies—but again, information flow is not considered in this work.

Information flow integrity enforcement given access control policies is considered in [17] relative to a trusted computing base inferred from SELinux policies. This approach was the first to leverage the closure of access control policies relative to information flow, but did not consider program internals or authorization hook placements to enforce such policies.

b) Information Flow and Taint Analysis.: Information flow control—and the distinction between explicit and implicit flows—was first proposed by [9]. Static approaches to information flow security have been incorporated into usable language models by [18, 27, 39] (among many others). The metaproperty of information flow (noninterference) has been shown to be a hyperproperty by [4]. Static information flow is combined with dynamic labels in [41], which bears some resemblance to our setting, except our model is focused on access control policies and authorization hooks, and not dynamic security labels per se. In this paper, we show that a direct flow analysis is sound with respect to a dynamic taint analysis [29] augmented with access control checks. Like information flow, the metaproperty of taint analysis is a hyperproperty [28], but different than noninterference.

Flow sensitivity in information flow has also been considered in previous systems [16, 21]. However, these systems do not treat the main problem we are concerned with, which is to predict how enforcement of an access control policy will interact with and affect direct information flow between external subjects during computation. Also, none of these systems combine technical features such as compositionality, fixpoint computation for recursive higher-order languages, and path sensitivity in same way as our combination of typing and abstract interpretation. Our approach is also modular in the sense that precision of the abstract interpretation can be "dialed in" as appropriate for the application.

c) Trace Effects and Type-and-effect Systems.: Polymorphic effect systems were first proposed by [23] and extended to type-and-effect systems by [35]. A generic type and effect systems was proposed by [24], and a general syntactic framework for flow-sensitive effects was recently proposed by [15] based (in part) on the semantic foundations of [37] and [19]. Our work is based on *trace effects* [30, 31, 32]—a specific formulation of type-and-effect systems which characterizes program traces. It is unknown whether or not trace effects can be encoded using the more-recently developed generic effect theories. In particular, trace effects are modeled by basic process algebras, whereas the aforementioned approaches to effects are modeled by directed generalizations of join-semilattices (*e.g.*, effectoids, monoids or quantales). Efficient type reconstruction for trace effects is an essential aspect of our approach (due to [32]), and analogous results have yet to be demonstrated for these more general effect frameworks.

d) Abstract Interpretation.: Abstract interpretation [5, 6, 7] has been applied to both type systems [8] and information flow [2, 14, 20] directly. Our use of abstract interpretation differs in that the first phase of our analysis is based on trace effects imbued with information flow events—which we do not justify using abstract interpretation—and our second phase is an abstract interpretation of these type-level trace effects. We therefore require significantly less exotic modeling techniques in our abstract interpreter because the concrete model upon which we abstract (the trace effect) is already itself an abstraction of concrete execution.

VIII. Concluding Remarks

In this paper we presented a static analysis for predicting whether an access control policy, instrumented in source code via authorization hooks, enforces an information flow policy during execution. The analysis combines a type theory and abstract interpretation to statically and automatically predict the effects of authorization hooks on information flow in programs. Formal results establish soundness of our analysis for a core language model, while extended examples illustrate the applicability of our system to problems in practical settings. Our technique supports flow-sensitivity, allows exploration of computation paths depending on access control policy, and is applicable to higher-order languages with general recursion.

References

[1] Sepehr Amir-Mohammadian and Christian Skalka. In-depth enforcement of dynamic integrity taint analysis. In *PLAS*, 2016.

[2] Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 874–887, New York, NY, USA, 2017. ACM.

[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] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[5] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[8] Patrick Cousot. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 316–331, New York, NY, USA, 1997. ACM.

[9] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[10] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 225–234. ACM, 2002.

[11] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 330–339, New York, NY, USA, 2005. ACM.

[12] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229, May 2006.

[13] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, May 2007.

[14] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197, New York, NY, USA, 2004. ACM.

[15] Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[17] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 11th USENIX Security Symposium*, pages 59–74, August 2003.

[18] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–302, New York, NY, USA, June 2015. ACM Press.

[19] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645, New York, NY, USA, 2014. ACM.

[20] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 211–222, New York, NY, USA,

2013. ACM.

[21] Peixuan Li and Danfeng Zhang. Towards a flow- and path-sensitive information flow analysis. *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 53–67, 2017.

[22] Benjamin Livshits and Jaeyeon Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Proceedings of the 22th USENIX Security Symposium*, pages 113–130, 2013.

[23] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[24] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[25] Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. Leveraging "Choice" to automate authorization hook placement. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 145–156, October 2012.

[26] Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. Producing hook placements to enforce expected access control policies. In *International Symposium on Engineering Secure Software and Systems*, pages 178–195. Springer, 2015.

[27] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[28] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE EuroS&P*, pages 15–30, 2016.

[29] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, pages 317–331, 2010.

[30] Christian Skalka. Trace effects and object orientation. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 139–150, New York, NY, USA, 2005. ACM Press.

[31] Christian Skalka. Types and trace effects for object orientation. *Journal of Higher Order and Symbolic Computation*, 21(3):239–282, 2008.

[32] Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

[33] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. RoleCast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 1069–1084, 2011.

[34] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[35] 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.

[36] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In *Proceedings of the 17th USENIX Security Symposium*, pages 379–394, 2008.

[37] Ross Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

[38] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting resource access by inferring programmer expectations. In *USENIX Security Symposium*, pages 973–988, Berkeley, CA, USA, 2014. USENIX Association.

[39] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[40] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *USENIX Security Symposium*, pages 33–48, 2002.

[41] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Secur.*, 6(2–3):67–84, March 2007.