Verified Enforcement of Stateful Information Release Policies

Nikhil Swamy Michael Hicks
University of Maryland, College Park
{nswamy,mwh}@cs.umd.edu

Abstract

Many organizations specify *information release* policies to describe the terms under which sensitive information may be released to other organizations. This paper presents a new approach for ensuring that security-critical software correctly enforces its information release policy. Our approach has two parts. First, an information release policy is specified as a security automaton written in a new language called AIR. Second, we enforce an AIR policy by translating it into an API for programs written in λAIR, a core formalism for a functional programming language. λAIR uses a novel combination of dependent, affine, and singleton types to ensure that the API is used correctly. As a consequence we can certify that programs written in λAIR meet the requirements of the original AIR policy specification.

1. Introduction

Many organizations, including financial institutions, healthcare providers, the military, and even the organizers of this conference, wish to specify the terms under which sensitive information in their possession can be released to their partners, clients, or the public. Such a specification constitutes an *information release policy*.

These policies are often quite complex. For example, consider the policy that regulates the disclosure of military information to foreign governments as defined by the United States Department of Defense [1992]. This policy includes the following provisions: a release must be authorized by an official with disclosure authority who represents the "DoD Component that originated the information"; the system must "edit or rewrite data packages to exclude information that is beyond that which has been authorized for disclosure"; a disclosure shall not occur until the foreign government has submitted "a security assurance [...] on the individuals who are to receive the information"; and, that the release must take place in the Foreign Disclosure and Technical Information System in which both approvals and denials of a release request must be logged.

We would like to ensure that software systems that handle sensitive data—including military systems, but also programs like medical-record databases, online auction software, and network appliances—correctly enforce such a high-level policy. As a concrete example, consider a specific kind of application called a *cross-domain guard*. These are programs, like network firewalls, that mediate the transfer of information between organizations at different trust levels. Commercial guards, e.g., the Data Sync guard produced by BAE [Focke et al., 2006], do not enforce high-level policies but rather implement low-level "dirty keyword" filters. Neither has the research community considered verified enforcement of high-level release policies. FlowWall [Hicks et al., 2007] is a firewall which, by virtue of being built with the Jif programming language [Chong et al., 2006], is sure to enforce a low-level filtering policy, but it does not appeal to high-level information release criteria.

To fill this gap, this paper presents a methodology for building highly-assured software that acts in accordance with an information release policy. Our approach has two parts. First, we define AIR, a formal language for defining information release policies. AIR's design follows from the observation that an information release policy is a kind of stateful authorization policy naturally expressed as an automaton [Schneider, 2000] (AIR stands for *automata for information release*). Satisfaction of a release obligation advances the state of the automation, and once all obligations have been fulfilled, the automaton reaches the accepting state and the protected information can be released. AIR allows one to express such automata in a natural way.

Second, we define λAIR, a core formalism for a programming language in which type-correct programs can be shown to correctly enforce an AIR policy. λAIR has three elements.

• First, λAIR provides *singleton types* to allow a programmer to associate sensitive data with an AIR automaton that protects that data. For example, an object x implementing a security automaton is given type $!Instance^N$, where N is a type-level name unique to x. Then, an integer i protected by x would be given type *Protected Int N*, which is essentially a kind of dynamic labeling [Zheng and Myers, 2004]. While

the state of an automaton can change, its association to a protected value will not change until all policy obligations have been fulfilled and the data is released, thus ensuring a kind of *complete mediation*. Prior work on verified enforcement of security automata via type checking [Walker, 2000] or inlined reference monitors [Erlingsson, 2004] is less flexible and/or has a larger trusted computing base (Section 6).

• Second, a λAIR program can express a release obligation with a *dependent type*, where an object having that type serves as a proof that the obligation has been fulfilled. For example, data could be released to a principal p only if p acts for some principal q (where p and q are program variables that store public keys). A proof of this fact could be represented by an object with type *ActsFor p q*. Generally speaking, proof objects represent certificates which are used to produce a *certified evaluation* of stateful policy logic—every authorization decision is accompanied by a proof that all obligations mandated by the high-level policy have been met. Certificates can be produced locally (e.g., by calling a function *acts_for* with type $(x{:}Prin) \to (y{:}Prin) \to$ *ActsFor x y* which succeeds if and only if the acts-for relationship holds) or remotely. As a final wrinkle, we use *affine types* to ensure that stale evidence about old policy states are never used in authorization decisions.

• Finally, given these mechanisms, we provide a way to compile an AIR policy to an API in λAIR, where each API function corresponds to a automaton transition such that the type of that function precisely expresses the evidence necessary for a transition to succeed. Thus type-correct λAIR programs must use the compiled AIR API correctly and, as a consequence, meet the specifications of the high-level policy. More precisely, we prove that the sequence of events produced by a program's execution is a word in the language accepted by the AIR automaton.

Using our techniques, one could build a cross-domain guard that adheres to high-level policy prescriptions; e.g., it would release information only after confirming that appropriate security assurances have been received, that to-be-released data packages have been rewritten appropriately, and that audit logs have been updated.

Our use of AIR policies for information release departs from prior work on declassification policies in that we do not focus on establishing a noninterference-like property for programs. However, our work complements noninterference-oriented interpretations of information release. For example, additional support for robust declassification [Zdancewic and Myers, 2001] could ensure that an adversary never causes information to be released, and furthermore, when it is released, it always follows the prescription of the high-level AIR policy.

2. AIR: Automata for Information Release

This section presents AIR, our language for expressing information release policies as automata.

Metavariables

id	class and rule ids	P	principals
$\mathscr{C}$	state constructors	n, i, j	integers
x, y, z	variables		

Core language

Declarations	D	::=	class id = (principal:P; states: $\overrightarrow{S}$; $\overrightarrow{R}$)
States	S	::=	$\mathscr{C}$ | $\mathscr{C}$ of $\overrightarrow{t}$
Rules	R	::=	id : R | id : T
Release	R	::=	When G release e with next state A
Transition	T	::=	When G do e with next state A
Guards	G	::=	x requested for use at y and $\overrightarrow{\exists x{:}t.C}$
Conditions	C	::=	A_1 *IsClass* A_2 | A_1 *InState* A_2
		|	A_1 *ActsFor* A_2 | $A_1 \leq A_2$
Atoms	A	::=	n | x | id | P | $\mathscr{C}(\overrightarrow{A})$ | $A_1 + A_2$
		|	Self | Class(A) | Principal(A)

e is an expression and t is a type in λAIR. (cf. Figure 4)

Figure 1. Syntax of AIR

2.1 Overview and Formal Syntax

An AIR policy consists of one or more *class declarations*. A program will contain *instances* of a class, where each instance protects some sensitive data via a labeling. Protected data can be accessed in two ways. First, each class C has an *owning principal P* such that P and all who *act for P* may access data protected by an instance of C. Second, each class defines a *release policy* by which its protected data can be released to an instance of different class.

The release policy is expressed using rules that define a security automaton, which is a potentially infinite state machine in which states represent security-relevant configurations. In the case of AIR, the security automaton defines conditions that must hold before data can be released. Each class instance consists of its current state, and each condition that is satisfied transitions the automaton to the next state. These transitions ultimately end in a release rule that allows data to be released to a different class instance, potentially in a modified form. Because sensitive data is associated with instances rather than classes, multiple resources may be governed by the same policy template (i.e., the automaton defined by the class) but release decisions are made independently for each resource. Dually, related resources can protected by the same instance, thereby allowing release decisions made with respect to one resource to affect the other.

The formal syntax for AIR policies is presented in Figure 1. We explain the syntax while stepping through our running example, shown in Figure 2. Throughout, we use the notation $\overrightarrow{a}$ to stand for the ordered sequence $a_1, \ldots, a_n$. Where the context is clear, we will also treat $\overrightarrow{a}$ as the set $\{a_1, \ldots, a_n\}$, or the infinite series $a_1, a_2 \ldots$.

A class declaration consists of a class identifier id, a principal identifier P for the owning principal, a list $\overrightarrow{S}$ of the automaton states, and finally a sequence of rules $\overrightarrow{R}$ that define the automaton transitions. Our example declares a

single class with identifier US_Army_Confidential, owned by the principal *US_Army*. This class defines a policy that protects confidential data owned by the U.S. Army. For simplicity, our examples use a flat namespace for class identifiers, and abstract names to represent principals.

Automaton states $\overrightarrow{S}$ are represented by terms constructed from an algebraic datatype whose constructors are drawn from the set $\mathscr{C}$. The example has two kinds of states. The nullary constructor Init represents the initial state of the automaton; all classes must have this state. The other kind of state is constructed by applying the unary constructor Debt to an argument of type *Int*. Constructors of the form $\mathscr{C}$ of $\overrightarrow{t}$ may carry data as indicated by the types $\overrightarrow{t}$. Types t (such as *Int*) are drawn from the programming language λAIR in which programs using AIR policies are written; λAIR is discussed in the next section.

Each rule in the rule set $\overrightarrow{R}$ is given a name *id*, and is either a *release rule* R or a *transition rule* T. In both cases, a rule begins with a guard G.

A guard G always begins by stating that a rule applies when data x is requested for use at the protection level described by an instance d of some class (typically d is different than the one being defined). G also includes a conjunction of condition expressions C that restrict the applicability of a rule; we discuss these in more detail below. Following its guard, the rule specifies an λAIR expression e that can either release information (perhaps after downgrading it by filtering or encryption) or do some other action (like logging), depending on whether the rule is a release rule or a transition rule. A rule concludes with the next state of the automaton following the release, specified as an atom A.

The first rule in our example US_Army_Confidential class is a release rule Conf_secret. The initial part of the rule's guard indicates that it applies when data protected by an instance of this class is to be released to some class instance d. This is allowed under the condition Class(d) *IsClass* US_Army_Secret, which states that d must be an instance of the US_Army_Secret class. When this condition is true, the release expression e is simply the variable x, which indicates that the protected data can be released without modification. Finally, the automaton transitions to state Self, the state it is already in.

We have picked a small ontology for condition expressions C based on integers, principals, classes and their instances. In a list of condition expressions of the form $\exists x_1{:}t_1.C_1, \ldots, \exists x_n{:}t_n.C_n$, each x_i is a variable of type t_i and is in scope as far to the right as possible, until the end of the rule. We will omit the $\exists x{:}t$ prefix if x is not free in some condition C. Condition expressions C are typed binary predicates over atoms A. For example, A_1 *ActsFor* A_2 is defined for *Principal*-typed atoms A_1 and A_2, and asserts that A_1 acts for A_2 according to some acts-for hierarchy among principals (not explicitly modeled here). Atoms include integers n, variables x, identifiers *id*, principal constants P, state literals constructed from an application of a state constructor $\mathscr{C}$ to a list of atoms, addition of integers and the implicit variable Self. We also include two operators: Class(A) retrieves the class of its argument A, a class instance, and Principal(C) retrieves the principal that owns the class C.

```
class US_Army_Confidential =
  principal : US_Army;   states : Init, Debt of Int;

  Conf_secret :
    When x requested for use at d and
      Class(d) IsClass US_Army_Secret
    release x with next state Self

  Conf_init :
    When x requested for use at d and
      Self InState Init
    do _ with next state Debt(0)

  Conf_coalition :
    When x requested for use at d and
      Principal(Class(d)) ActsFor Coalition,
      ∃count:Int.Self InState Debt(count),
      count ≤ 10
    release
      (log(...x...d); encrypt (pubkey (principal (class d))) x)
    with next state Debt(count + 1)
```

Figure 2. A simple risk-adaptive policy in AIR

2.2 A Risk-adaptive policy in AIR

The remainder of the class declaration in Figure 2 illustrates the other features AIR. Taken as a whole, the class US_Army_Confidential can be thought of as implementing a simple kind of *risk-adaptive* access control [Cheng et al., 2007], in which information is released according to a *risk budget*, with the intention of quantifying the risks vs. the benefits of releasing sensitive information.

This class associates a fixed risk debt with each secure resource, as reflected in the state Debt of *Int*. Each time information about the resource is released we estimate the risk associated with that release, and add that risk to our debt. When the total risk debt exceeds a threshold then further releases to coalition partners are not allowed. The two remaining rules in the policy, Conf_init and Conf_coalition implement this behavior.

The Conf_init transition rule applies when processing a release request from any instance d. This rule can be thought of as the class "constructor," applying when the automaton is in the Init state (condition Self *InState* Init). The "do" expression initializes the risk budget to 0 by transitioning the automaton to the Debt(0) state. The Conf_coalition rule allows information to be released to a coalition partner. In particular, if the release target class is owned by a principal that acts for the *Coalition* (expressed by Principal(Class(d)) *ActsFor Coalition*), then information can be released only if the current risk debt has not exceeded the budget, as expressed in the latter two conditions. The first of these requires the current state of the

```
1   let x_a1, a1 = get_secret_file_and_policy () in
2   let a2, channel = get_request () in
3    (* generating evidence of policy compliance *)
4   let a2, a2_class = get_class a2 in
5   let ev1 = acts_for (principal a2_class) Coalition in
6   let a1, Debt(debt), ev2 = get_current_state a1 in
7   let ev3 = leq debt 10 in
8    (* supplying evidence to policy API and releasing data *)
9   let a1', a2, x_a2 = Conf_coalition a1 x_a1 a2 ev1 debt ev2 ev3 in
10    send channel x_a2
```

Figure 3. Programming with an AIR policy

automaton to be Debt(*count*), where *count* is variable with type *Int* which tracks the current risk debt. The last condition requires that *count* is not above the preallocated risk budget of 10. With these conditions satisfied, Conf_coalition logs the fact that a release has been authorized and permits release of the data after it has been downgraded using an encryption function. In this case, the downgrading expression encrypts x with the public key of the principal that owns the instance d. Unlike releases to the US_Army_Secret class which do not alter the risk debt of the automaton, Conf_coalition increments the risk debt by transitioning to the Debt$(count + 1)$ state, indicating that releases to the *Coalition* are more risky than upgrading to a higher classification level of the same organization (via rule Conf_secret).

AIR as presented here is particularly simple. We anticipate extending AIR with support for more expressive condition ontologies and release rules. For instance, instead of a fixed set of ontologies, we could embed a stateful authorization logic (say, in the style of SMP [Becker and Nanz, 2007]) to allow custom ontologies and release rules to be programmed within an AIR class.

3. A Programming Model for AIR

Given a particular AIR policy, we would like to do two things. First, we must have a way of reflecting an AIR policy in a program by protecting sensitive resources with instances of an AIR class. Second, we must ensure that all uses of protected data adhere to the prescriptions of the AIR policy. Taken together, we can then claim that an AIR policy is correctly enforced by a program. To achieve these goals, we have defined a formal model for a language called λAIR in which one writes programs that use AIR policies. λAIR's type system ensures that these policies are used correctly. The rest of this section defines the programming model for this language and the next two sections flesh out its syntax and semantics. Section 5.4 proves that type-correct programs act only in accordance with their AIR policies.

The programming model for using AIR policies has two elements. First, programmers tie an AIR policy to data in the program by constructing instances of AIR classes and labeling one or more pieces of data with these instances. This association defines (1) the set of principals that may view the data (in particular, the principal P that owns the class, and any principals that may act for P), and (2) the rules that allow the data to be released. As in other security-typed languages, the labeling specification (expressed using type annotations) is part of the trusted computing base. One might adapt the ideas of Hicks et al. [2007] to ensure the initial labeling of data is consistent with the high-level system security goals.

Second, programmers manipulate data protected by an AIR class instance through a class-specific API that is generated by compiling each AIR class definition to a series of program-level definitions. For example, each AIR class's release and transition rules are compiled to functions that can be used to release protected data. The types given to these functions ensure that a caller of the function must always provide evidence that the necessary conditions to release protected data have been met.

Figure 3 illustrates a program using the AIR policy of Figure 2, written using a ML-like notation. At a high level, this program processes requests to release information from a secret file. The files are stored on the file system together with a policy label that represents a particular AIR class instance. Before disclosing the information, the program must make sure that the automaton that protects the data is in a state that permits the release. The first two lines set up the scenario. At line 1, we read the contents of a secret file into the variable x_a1 and the automaton that protects this file into the variable a1. Initially, only the principals that act for the owner of the class of a1 can view these secrets. At line 2, the program blocks until a request is received. The request consists of an output channel and another automaton instance a2 that represents the policy under which the requested information will be protected after the release. In effect, the information, once released, will be under the protection of the principal that owns the class of a2.

Prior to responding to the request, on lines 4-7 we must establish that a1 is in a state that permits the release. At line 4, we extract the class of the instance a2. At line 5, we check that the owner of a2's class acts for the Coalition principal and, if this check succeeds, we obtain a certificate ev1 as evidence of this fact. At line 6, we extract the current state of the automaton a1, use pattern matching to check that it is of the form Debt(debt) (for some value of debt) and receive an evidence object ev2 that attests to the fact that a1 is currently in this state. At line 7, we check that the total debt associated with the current state of the automaton is not greater than 10 and obtain ev3 as evidence if the check succeeds.

At line 9 we call Conf_coalition, a function produced by compiling the AIR policy. We pass in the automaton a1 and the secret data x_a1; the automaton a2 to which x_a1 is to be released; and the certificates that serve as evidence for the release conditions. Conf_coalition returns a1' which represents the next state of the automaton (presumably in the Debt(debt+1) state); a2 the unchanged destination automaton; and finally, x_a2, which contains the suitably down-

graded secret value. On the last line, we send the released information on the channel received with the request.

For programs like our example, we would like to verify that all releases of information are mediated by calls to the appropriate transition and release rules as defined by the AIR policy (functions like Conf_coalition). Additionally, we would like to verify that a program satisfies the mandates of an AIR policy rule by presenting evidence that justifies the appropriate release conditions. This evidence-passing style supports our goal of certifying the evaluation of all authorization decisions, while being flexible about the mechanism by which an obligation is fulfilled. To return to the DoD example from the introduction, this design gives us the flexibility to allow release authorizations to be obtained in one part of the system and security assurances from the recipient to be handled in another; the cross-domain guard must simply collect evidence from the other components rather than performing these operations itself. λAIR's type system is designed so that type correctness ensures these goals are satisfied, i.e., a type-correct program uses its AIR policy correctly. The type system has three key elements:

Singleton types. First, in order to ensure complete mediation, we must be able to correctly associate data with the class instance that protects it. For example, Conf_coalition expects its first argument to be an automaton and the second to be data protected by that automaton. In an ML-like type system, this function's type might begin with $\forall\alpha.\mathit{Instance} \rightarrow \alpha \rightarrow \ldots$ But such a type is not sufficiently precise since it does not prescribe any relationship between the first and second argument, allowing the programmer to erroneously pass in a2 as the first argument, rather than a1, for example. To remedy this problem, we can give Conf_coalition a type like the following (as a first approximation):

$$\forall\alpha,N,M.\mathit{Instance}^N \rightarrow \mathit{Protected}\ \alpha\ N \rightarrow \ldots$$

Here, N is a unique type-level name for the class instance provided in the first argument. The second argument's type *Protected* α N indicates it is an α value protected by the instance N, making clear the association between policy and data. We can ensure that values of type *Protected* α N may only be accessed by principals P that act for the owner of the class instantiated by the instance named N. This approach is more flexible than implicitly pairing each protected object with its own (hidden) automaton. For example, with our approach one can encode policies like secret sharing, in which a set of related documents are all protected by the same automaton instance. Each document's type would refer to the same automaton, e.g., *Protected Doc N*. Information released about one document updates the state of the automaton named N and can limit releases of the other documents.

Dependent types. Arguments 4-7 of Conf_coalition represent evidence (proof certificates) that the owner of class instance a2 acts for Coalition, and that a1 is in a state authorized to release the given data. We give types to these arguments that reflect the propositions that they are supposed to witness. For example, we give the seventh argument (ev3) to Conf_coalition the type *LEQ debt* 10 where *LEQ* is a dependent type constructor applied to two *expressions*, debt and 10, where each has type *Int*. Data with type *LEQ n m* represents a certificate that proves $n \leq m$. If we allow such certificate values to only be constructed by trusted functions that are known to correctly implement the semantics of integer inequality, then we can be sure that functions like Conf_coalition are only called with valid certificates—i.e., type correctness guarantees that all certificates are valid proofs of the propositions represented by their types and there is no need to inspect these certificates at run time. If we interface with other programs, we can check the validity of proof certificates at run time before allowing a call to proceed. Either way, the type system supports an architecture that enables certified evaluation of an AIR policy.

Affine types. The final piece of our type system is designed to cope with the stateful nature of an AIR policy. The main problem caused by a state change is illustrated by the value returned by the Conf_coalition function. In our example, a1' represents the state of the policy automaton that protects x_a1 *after a release* has been authorized. Thus, we need a way to break the association between x_a1 and the old, stale automaton state a1. We achieve this in two steps. First, even though our type system supports dependent types, as shown earlier, we use singleton types to give x_a1 the type *Protected* α N, where N is a singleton type name for a1 (rather than giving x_a1 a more-direct dependent type of the form *Protected* α a1). The second step is to use *affine types* (values with an affine type can never be used more than once) to consume stale automaton values, so that at any program point, there is only one usable automaton value that has the type-name N. Thus, we give both a1 and a1' the type $!\mathit{Instance}^N$, where $!t$ denotes an affinely qualified type t. Once a1 is passed as an argument to Conf_coalition (which constitutes a use) it can no longer be used in the rest of the program; a1' is the only automaton that can be used in subsequent authorization checks for x_a1. Thus, a combination of singleton and affine types transparently takes care of relabeling data with new automaton instances. (One might also wonder how we deal with proof certificates that can become stale because of the changing automaton state; we discuss this issue in detail in Section 5.1.)

To illustrate how these singleton, dependent, and affine types interact we show a part of the type of Conf_coalition below (the full type is discussed in Section 5.2).

$$\begin{aligned}\forall\alpha,N,M.\ \ & !\mathit{Instance}^N \rightarrow \mathit{Protected}\ \alpha\ N \rightarrow !\mathit{Instance}^M \rightarrow \\ & \ldots \rightarrow (\mathsf{debt} : \mathit{Int}) \rightarrow \ldots \rightarrow (\mathit{LEQ}\ \mathsf{debt}\ 10) \rightarrow \\ & (!\mathit{Instance}^N \times !\mathit{Instance}^M \times \mathit{Protected}\ \alpha\ M)\end{aligned}$$

The first three arguments are the *affine* source automaton (a1), the data it protects (x_a1), and the *affine* destination automaton (a2). On the next line, we show the dependent type

Metavariables

B	Base terms	T	Type constructors	α, β, γ	Type vars

Core language

Terms	e	::=	$x \mid \lambda x{:}t.e \mid e\ e \mid \Lambda\alpha{::}k.e \mid e\ [t] \mid B$
		$\mid$	$e\ \{e\} \mid \mathsf{case}\ e\ \mathsf{of}\ \overrightarrow{x{:}t.e} : e\ \mathsf{else}\ e \mid \bot \mid \mathsf{new}\ e$
Types	t	::=	$(x{:}t) \xrightarrow{\varepsilon} t \mid \alpha \mid \forall\alpha{::}k \xrightarrow{\varepsilon} t \mid T$
		$\mid$	$t \Rightarrow t \mid q\ t \mid t\ t \mid t\ e \mid t^{\alpha}$
Affinity	q	::=	$! \mid \cdot$
Simple kinds	k	::=	$U \mid A \mid N$
Kinds	K	::=	$k \mid k \to K \mid t \to K$
Effects	ε	::=	$\cdot \mid \alpha \mid \varepsilon \uplus \varepsilon \mid \varepsilon \cup \varepsilon$

Signatures and typing environments

Signatures	S	::=	$(B{:}t) \mid (T{::}K) \mid S, S$
Type env.	Γ	::=	$x{:}t \mid \alpha{::}k \mid \Gamma, \Gamma$
Affine env.	A	::=	$x \mid A, A$

Figure 4. Syntax of λAIR

given to the evidence that the current debt of the automaton is not greater than 10. Finally, consider the return type of Conf_coalition. The first component of this three-tuple is a class instance with the same name N as the first argument. This returned value is the new state of the automaton named N—it protects all existing data of type *Protected* α N (such as `x_a1`). The second component of the three-tuple is the unchanged target automaton. The third component contains the data ready to be released—its type *Protected* α M indicates that it is now protected by the target automaton instance M.

Adhering to the constraints of λAIR's type system is surely more burdensome and difficult than when using a more typical programming language. Thus λAIR may be most appropriate for the security-critical kernel of an application, or even as the (certifiable) target language of a program transformation for inline reference monitoring. We leave to future work an exploration of support—e.g., type inference—for improving λAIR's usability.

4. Syntax and Semantics of λAIR

λAIR extends a core System F^{ω} [Mitchell, 1996] with support for singleton, dependent, and affine types. λAIR is parameterized by a *signature* S that defines base terms B and type constructors T—each AIR class declaration D is compiled to a signature S_D that acts as the API for programs that use D. All AIR classes share some elements in common, like integers, which appear in a *prelude* signature S_0. We explain the core of λAIR using examples from the prelude. The next section describes the remainder of the prelude and shows how our example AIR policy is compiled.

4.1 Syntax

Figure 4 shows the syntax of λAIR. The core language expressions e are mostly standard, including variables x, lambda abstractions $\lambda x{:}t.e$, application $e\ e'$, type abstraction $\Lambda\alpha{::}k.e$ and type application $e\ [t]$. Functions have dependent type $(x{:}t) \xrightarrow{\varepsilon} t'$ where x names the argument and may be bound in t'. Function arrows are decorated with an effect ε that records a set of type names given to automaton instances that are created when the function is applied; we discuss these later. Type variables are α and universally quantified types $\forall\alpha{::}k \xrightarrow{\varepsilon} t$ are standard; the latter stands for a type t that is universally quantified over all types α of kind k. Like functions, type abstractions can also have an effect when they are applied; when the effect is empty we will write a universally quantified type as $\forall\alpha{::}k.t$.

The signature S defines the legal base terms B and type constructors T, mapping them to their types t and kinds K, respectively. The prelude S_0 defines several standard terms and types, such as support for integers and pairs, which we use to illustrate λAIR's other type and term constructs.

The prelude includes the constructor *Int* to represent the type of integers, giving it kind U, written $Int{::}U$. Kind U is one of three simple kinds k. Types with simple kind A are affine in that the typing rules permit affinely-typed terms to be used at most once. Affine types are written $!t$, which is an instance of the form $q\ t$ where $q = !$. Terms whose types have kind U are unrestricted in their use. Kind N is given to type names, which we explain shortly.

The prelude also defines two base terms for constructing integers: $\mathsf{Zero} : Int$ represents the integer 0, while $\mathsf{Succ} : Int \Rightarrow Int$ is a unary *data constructor* that produces an *Int* given an *Int*. Data constructor application is written $e\ \{e\}$; thus the integer 1 is represented $\mathsf{Succ}\ \{\mathsf{Zero}\}$ (but we write $0, 1, 2$ etc. for brevity). Programs can use the expression form $\mathsf{case}\ e\ \mathsf{of}\ \overrightarrow{x{:}t.e} : e\ \mathsf{else}\ e$ to destruct data constructor values using pattern matching. This is essentially standard; details are in our technical report [Swamy and Hicks, 2008].

In addition to simple kinds k, kinds K more generally can classify functional type constructors, using the forms $k \to K$ and $t \to K$. A type constructor t_1 having the first form can be applied to another type using $t_1\ t_2$ to produce a (standard) type, while one of the second form can be applied to a term using $t\ e$ to produce a dependent type. As an example of the first case, the prelude defines a type constructor $\times{::}U \to U \to U$ to model pairs; $\times\ Int\ Int$ is the type of a pair of integers (for clarity, from here on we will use infix notation and write a pair type as $t \times t'$). The prelude also defines a base term Pair which has a polymorphic type $\forall\alpha, \beta{::}U.\alpha \Rightarrow \beta \Rightarrow \alpha \times \beta$ for constructing pair values.

Evidence for condition expressions in an AIR policy are typed using dependent types. For instance, the prelude provides means to test inequalities $A_1 \leq A_2$ that appear in a policy and generate certificates that serve as evidence for successful tests:

$$(LEQ{::}Int \to Int \to U),$$
$$(leq{:}(x{:}Int) \to (y{:}Int) \to LEQ\ x\ y)$$

LEQ is a dependent type constructor that takes two expressions of type *Int* as arguments and produces a type having kind U. This type is used to classify certificates that witness the inequality between the term arguments. These cer-

tificates are generated by the *leq* function, which has a dependent type: the labels x and y on the first two arguments appear in the returned type. Thus the call *leq* 3 4 would return a certificate of type *LEQ* 3 4 because 3 is indeed less than 4. An attempt to construct a certificate *LEQ* 4 3 by calling *leq* 4 3 would fail at run time, returning $\perp$ in our semantics, which has the effect of terminating the program (we could use option types or add support for for exceptions to handle failures more gracefully). The signature does not include a data constructor for the *LEQ* type, so its values cannot be constructed directly by programs—the only way is by calling the *leq* function.

We discuss the remaining constructs—including kinds N, singleton types t^α, affine type constructors new e, and effects ε—in conjunction with the types rules next.

4.2 Static semantics

Figure 5 shows the main rules from the static semantics of λAIR, which consists of two judgments. The full semantics can be found in our technical report. Both judgments are parameterized by a signature S. The main judgment giving an expression e a type t is written $\Gamma; A \vdash_S e : t; \varepsilon$ where Γ is the standard typing environment, A is a list of affine assumptions, and effect ε is the set of fresh type names generated in e. The second judgment, $\Gamma \vdash_S t :: K$ states that a type t is well-formed at kind K in the environment Γ. Recall that the type system must address three main concerns. First, in order to ensure complete mediation, we must correctly produce singleton type names to associate data with automaton instances. Next, for certified evaluation we must be able to properly construct evidence using dependent types. Finally, in order to cope with automaton state changes, we must prevent stale automaton instances or evidence from being reused via affine types. We consider each of these aspects of the system in turn, starting with the type judgment and then proceeding to the kinding judgment.

We construct new automaton instances using the new e construct. We adapt an approach used by Pratikakis et al. [2006] in order to give a unique type-level name to these instances. Thus, (T-NEW) assigns the name α to the type in the conslusion, ensuring (via $\alpha \uplus \varepsilon$) that α is distinct from all other names ε that have already been used. Recall from Section 3 that protected values will refer to this name α in their types (e.g., *Protected Int* α). The resulting type, $!t^\alpha$ is also affinely qualified; we discuss this shortly.

Notice in (T-NEW) that α must be a type variable that has been introduced into the context, which can either be generalized by a type abstraction, or is part of the top-level environment Γ. Extending λAIR with support for recursion would necessitate using existential quantification to abstract names in recursive data structures, as well as a means to forget names whose data goes out of scope (e.g., as in each iteration of a loop); supporting such extensions is straightforward [Pratikakis et al., 2006]. (T-DROP) allows the name associated with a type to be dropped. This is convenient for writing functions that need to inspect the state of an automaton without actually causing a transition. This rule is sound because although the name α of a type $!t^\alpha$ can be dropped, α cannot be reused as the type-level name of any other automaton (i.e., ε is unaffected).

In order to ensure that type-level names are not reused, we track the assignment of these names as effects through all the other rules. Thus, in (T-ABS) we associate the effect of the body of a function with that function's type. Since the body is itself suspended until the function is applied, the conclusion of (T-ABS) shows that the effect of the function itself is empty. However, when a function is applied, (T-APP) ensures that the effects of both subexpressions and the function's body itself are disjoint.

The dependent-typing feature of λAIR is illustrated by the types given to functions in (T-ABS) and the form of the (T-APP) rule. In a function type, $(x{:}t) \xrightarrow{\varepsilon} t'$, x names the formal parameter and is bound in t'. In the conclusion of (T-APP) we substitute the actual argument e' for x in the return type. Thus, given that a function f has the type $(\mathsf{debt} : \mathit{Int}) \to (\mathit{LEQ}\ \mathsf{debt}\ 10) \to t$, the application $(f\ 11)$ is given the type $(\mathit{LEQ}\ 11\ 10) \to t$. That is, the type of the second argument of *f depends on* the first argument. Note that although λAIR permits arbitrary expressions to appear in types, type checking in λAIR is decidable. This is because when enforcing an AIR policy, we never need to reduce expressions that appear in types.

Finally, we consider how the type system enforces the "use at most once" property of affine types. First, (T-NEW) introduces affine types by giving new automaton instances the type $!t^\alpha$. Values of affine type can be destructed in the same way as values of unrestricted type. For example, notice that (T-APP) allows e to be applied to e' as long as e has function type, whether or not this type is affine (q is unspecified). However, when a value with an affine type is bound to a variable, we must make sure that that variable is not used more than once. This is prevented by the type rules through the use of affine assumptions A, which lists the subset of variables with affine type in Γ which have not already been used. The use of an affine variable is expressed in the rule (T-XA), which types a variable x in the context of the single affine assumption x. To prevent variables from being used more than once, other rules, such as (T-APP), are forced to split the affine assumptions between their subexpressions. Affine assumptions are added to A by (T-ABS) using the function $a(x,k)$, where x is the argument to the function and k is the kind of its type. If the argument x's type has kind A then it is added to the assumptions, otherwise it is not. The function $p(A)$ is used to determine the affinity qualifier of the function's type: if no affine assumptions from the environment are used by the function (A is $\cdot$), then it is unrestricted; otherwise it has captured an assumption from the environment and should be called at most once. We include a weakening rule (T-WKN) that allows affine variables to re-

$\Gamma; A \vdash_S e : t; \varepsilon$ An expression e has type t in environment Γ with affine assumptions A and generates type names ε.

$$\frac{\Gamma \vdash_S \Gamma(x) :: U}{\Gamma; \cdot \vdash_S x : \Gamma(x); \cdot}\ \text{(T-X)} \qquad \frac{}{\Gamma; x \vdash_S x : \Gamma(x); \cdot}\ \text{(T-XA)} \qquad \frac{\Gamma; A \vdash_S e : t; \varepsilon \quad \Gamma \vdash_S t :: U \quad \Gamma(\alpha) = N}{\Gamma; A \vdash_S \mathsf{new}\ e : !t^{\alpha}; \alpha \uplus \varepsilon}\ \text{(T-NEW)}$$

$$\frac{\Gamma \vdash_S t_x :: k \quad \Gamma, x : t_x; A, a(x,k) \vdash_S e : t_e; \varepsilon \quad q = p(A)}{\Gamma; A \vdash_S \lambda x{:}t_x.e : q((x{:}t_x) \xrightarrow{\varepsilon} t_e); \cdot}\ \text{(T-ABS)} \qquad \text{where } \begin{array}{ll} a(x,A) = x & a(x,U) = \cdot \\ p(A) = ! & p(\cdot) = \cdot \end{array}$$

$$\frac{\Gamma; A \vdash_S e : q((x{:}t') \xrightarrow{\varepsilon} t); \varepsilon_1 \quad \Gamma; A' \vdash_S e' : t'; \varepsilon_2}{\Gamma; A, A' \vdash_S e\ e' : [x \mapsto e']t; \varepsilon \uplus \varepsilon_1 \uplus \varepsilon_2}\ \text{(T-APP)} \qquad \frac{\Gamma; A \vdash_S e : t; \varepsilon \quad \varepsilon' \subseteq \mathrm{dom}(\Gamma)}{\Gamma; A, A' \vdash_S e : t; \varepsilon \uplus \varepsilon'}\ \text{(T-WKN)} \qquad \frac{\Gamma; A \vdash_S e : t^{\alpha}; \varepsilon}{\Gamma; A \vdash_S e : t; \varepsilon}\ \text{(T-DROP)}$$

$\Gamma \vdash_S t :: K$ A type t has kind K.

$$\frac{\Gamma(\alpha) = k}{\Gamma \vdash_S \alpha :: k}\ \text{(K-A)} \qquad \frac{\Gamma \vdash_S t :: A \quad \Gamma \vdash_S \alpha :: N}{\Gamma \vdash_S t^{\alpha} :: A}\ \text{(K-N)} \qquad \frac{\Gamma \vdash_S t :: U}{\Gamma \vdash_S !t :: A}\ \text{(K-AFN)}$$

$$\frac{\Gamma \vdash_S t :: k \quad \Gamma, x : t \vdash_S t' :: k' \quad k, k' \neq N \quad \forall \alpha \in \varepsilon. \Gamma \vdash_S \alpha :: N}{\Gamma \vdash_S (x{:}t) \xrightarrow{\varepsilon} t' :: U}\ \text{(K-FUN)} \qquad \frac{\Gamma \vdash_S t :: t' \to K \quad \Gamma; \mathit{Affine}(\Gamma) \vdash_S e : t'; \varepsilon}{\Gamma \vdash_S t\ e :: K}\ \text{(K-DEP)}$$

Figure 5. Static semantics of λAIR (Selected rules)

main unused. This rule also allows additional effects to be added so long as they are disjoint from all other effects.

In the kinding judgment, the rule (K-A) is standard. (K-N) allows a name to be associated with any affine type t. (K-AFN) checks an affinely-qualified type: types such as $!!t$ are not well-formed. (K-FUN) ensures that neither the argument or the return type of a function has the kind of a type name. Type names are not inhabited by any value. (K-FUN) also ensures that no free names appear in the effect annotation ε. Finally, (K-DEP) checks the application of a dependent type constructor. Here, we have to ensure that the type of the argument e matches the type of the formal. In the second premise, $\mathit{Affine}(\Gamma)$ stands for *all* affine assumptions in Γ. Since e is a type-level expression which can be erased at run time, it is permitted to use affine assumptions that may have been used elsewhere.

4.3 Dynamic semantics

The dynamic semantics of λAIR defines a standard call-by-value, small-step reduction relation for a purely functional language, using a left-to-right evaluation order. The full definition can be found in our technical report. The form of the relation is : $M \vdash e \xrightarrow{l} e'$. This judgment claims that a term e reduces in a single step to e' in the presence of a model M that interprets the base terms in a signature. The security-relevant reduction steps are annotated with a trace element l, which is useful for stating our security theorem. In this section, we briefly discuss the form of the model M and the trace elements l and state our type soundness result.

Following a standard approach for interpreting constants in a signature [Mitchell, 1996], we define a model M by axiomatizing the reductions of base term applications. In practice, we would implement the model in a real programming language. For example, we could do this in FABLE [Swamy et al., 2008], a language we specifically designed for programming *policy functions* that may coerce one protected type to another (like Conf_coalition) or may produce unforgeable certificates (like acts_for).

A model M contains equations $B : \mathscr{D} \rightsquigarrow e$, where $\mathscr{D}$ is a sequence of types and values. We require the types of the expressions in these equations to be consistent with the type given to B in the signature. An example of an equation is $\mathit{leq} : \ 4,\ 3 \rightsquigarrow \bot$ indicating that the expression $(\mathit{leq}\ 4\ 3)$ reduces to $\bot$. We also need a mechanism to construct a value that represents a proof certificate for a valid inequality; i.e., a value that inhabits the type *LEQ* 3 4. In practice, one could either chose a concrete representation for these objects if proofs need to be checked at run time (for instance, when interfacing with type-unsafe code); or, if we are in a purely type-safe setting, we could chose an arbitrary value (like unit) to represent a proof certificate. In our technical report, we introduce a special value to stand for proof objects that facilitates our soundness proof.

The security-relevant actions in a program execution are the reduction steps that correspond to automaton state changes. As indicated earlier, each transition and release rule in a policy will be translated to a function-typed base term like Conf_coalition. Thus, every time we reduce an expression e using a base-term equation $B : \mathscr{D} \rightsquigarrow e'$, we record $l = B : \mathscr{D}$ in the trace: i.e., $M \vdash e \xrightarrow{B:\mathscr{D}} e'$.

The statement of our type soundness theorem is shown below; the proof is in our technical report.

Theorem (Type soundness)**.** *Given a set of name constants $\Gamma = \alpha_1 :: N, \ldots, \alpha_n :: N$ such that $\Gamma; \cdot \vdash_S e : t; \varepsilon$, and an interpretation M such that M and S are type-consistent, then $\exists e'.M \vdash e \xrightarrow{l} e'$ or $\exists v.e = v$, or $e' = \bot$. Moreover, if $M \vdash e \xrightarrow{l} e'$ then $\Gamma; \cdot \vdash_S e' : t; \varepsilon$.*

5. Translating AIR to λAIR

In this section, we show how we discuss how we translate an AIR class to a λAIR API, describe how that API is to be used, and state our main security theorem.

5.1 Representing AIR primitives

In order to enforce an AIR policy we must first provide a way to tie the policy to the program by protecting data with AIR automata. We must also provide a concrete representation for automata instances and a means to generate certificates that attest to the various release conditions that appear in the policy. These constructs are common to all λAIR programs and appear in the standard prelude, along with the integers and pairs discussed in Section 4.1.

Protecting data. As indicated in Section 3, we include the following type constructor to associate an automaton with some data: $(\mathit{Protected}{::}U \rightarrow N \rightarrow U)$. A term with type $\mathit{Protected}\ t\ \alpha$ is governed by the policy defined by an automaton instance with type-level name α. We would like to ensure that all operations on protected data are mediated by functions that correspond to AIR policy rules. For this reason, we do not provide an explicit data constructor for values of this type (ensuring that they cannot be destructed directly, say, via pattern matching). Values of this type are introduced only by assigning the appropriate types to functions that retrieve sensitive data—for instance, library functions that read secret files from the disk can be annotated so that they return values with a protected type.

In addition to functions corresponding to AIR class rules, we can provide functions that allow a program to perform secure computations over protected values. We have explored such functions in our work on FABLE and showed that computations that respect a variety of policies (ranging from access control to information flow) can be encoded [Swamy et al., 2008]; we do not consider these further here.

Next, we discuss our representation of an AIR automaton—these include representations of the class that the automaton instantiates and the principal that owns the class.

Principals. The nullary constructor *Prin* is used to type principal constants P; i.e., $(\mathit{Prin}{::}U), (P{:}\mathit{Prin})$. As with integers, we need a way to test and generate evidence for acts-for relationships between principals. We include the dependent type constructor and run-time check shown below.

$$(\mathit{ActsFor}{::}\mathit{Prin} \rightarrow \mathit{Prin} \rightarrow U)$$
$$(\mathit{acts\_for}{:}(x{:}\mathit{Prin}) \rightarrow (y{:}\mathit{Prin}) \rightarrow \mathit{ActsFor}\ x\ y)$$

AIR classes. A class consists of a class identifier *id* and a principal P that owns the class. The type constructors $(\mathit{Id}{::}U), (\mathit{Class}{::}U)$ are used to type identifiers and classes. Classes are constructed using the data constructor $(\mathsf{Class}{:}\mathit{Id} \Rightarrow \mathit{Prin} \Rightarrow \mathit{Class})$. The translation of an AIR class introduces nullary data constructors like $\mathsf{US\_Army\_Confidential}{:}\mathit{Id}$ and $\mathsf{US\_Army}{:}\mathit{Prin}$, from which we can construct the class $\mathsf{USAC} =$ $\mathsf{Class}\ \{\mathsf{US\_Army\_Confidential}\}\ \{\mathit{US\_Army}\}$. Finally, we use a dependent type constructor and run-time check to generate evidence that two classes are equal.

$$(\mathit{IsClass}{::}\mathit{Class} \rightarrow \mathit{Class} \rightarrow U),$$
$$(\mathit{is\_class}{:}(x{:}\mathit{Class}) \rightarrow (y{:}\mathit{Class}) \rightarrow \mathit{IsClass}\ x\ y)$$

Class instances. Instances are typed using the $\mathit{Instance}{::}U$ type constructor. Each instance must identify the class it instantiates and the current state of its automaton. For each state in a class declaration, we generate a data constructor in the signature that constructs an *Instance* from a *Class* and any state-specific arguments. For example, we have:

$$\mathsf{Init}{:}\mathit{Class} \Rightarrow \mathit{Instance}, \mathsf{Debt}{:}\mathit{Class} \Rightarrow \mathit{Int} \Rightarrow \mathit{Instance}$$

Thus the expression $\mathsf{new}\ \mathsf{Init}\ \{\mathsf{USAC}\}$ constructs a new instance of a class. According to (T-NEW), this expression has the affine type $!\mathit{Instance}^{\alpha}$, where the unique type-level name α allows us to protect some data with this automaton. Since we wish to allow data to be protected by automata that instantiate arbitrary AIR classes, we give all instances, regardless of their class, a type like $!\mathit{Instance}^{\alpha}$, for some α. This has the benefit of flexibility—we can easily give types to library functions that can return data (like file system objects) protected by automata of different classes. However, we must rely on a run-time check to examine the class of an instance since it is not evident from the type—the following two elements of the prelude accomplish this.

$$\mathit{ClassOf}{::}N \rightarrow \mathit{Class} \rightarrow U$$
$$\mathit{class\_of\_inst}{:}\forall\alpha{::}N.(x{:}!\mathit{Instance}^{\alpha}) \rightarrow (!\mathit{Instance}^{\alpha} * c{:}\mathit{Class} * \mathit{ClassOf}\ \alpha\ c)$$

The function *class_of_inst* extracts a *Class* value c from an instance named α and produces evidence (of type $\mathit{ClassOf}\ \alpha\ c$) that α is an instance of c. The return type of this function is interesting for two reasons. First, because the returned value relates the class object in the second component of the tuple to the evidence object in the third component, we give the returned value the type of a *dependently typed tuple*, (designated by the symbol $*$). Although we do not directly support these tuples, they can be easily encoded using dependently typed functions [Swamy et al., 2008]. Second, notice that even though *class_of_inst* does not cause a state transition, the first component of the tuple it returns contains an automaton instance with the same type as the argument x. This is a common idiom when programming with affine types; since the automaton instance is affine and can only be used once, functions like *class_of_inst* simply return the affine argument x back to the caller for further use.

The prelude also provides the following constructs that allow a program to inspect the current state of an automaton instance.

$$\mathit{InState}{::}!\mathit{Instance} \rightarrow \mathit{Instance} \rightarrow U$$
$$\mathit{state\_of\_inst}{:}\forall\alpha{::}N.(x{:}!\mathit{Instance}^{\alpha}) \rightarrow (z{:}!\mathit{Instance}^{\alpha} * y{:}\mathit{Instance} * \mathit{InState}\ z\ y)$$

These constructs are similar to the forms shown for examining the class of an instance, but with one important difference. Since the state of an automaton is transient (it can change as transition rules are applied), we must be careful when producing evidence about the current state. This is in contrast to the class of an automaton which never changes despite changes to the current state. Thus, we must ensure that stale evidence about an old state of the automaton can never be presented as valid evidence about the current state.

The distinction between evidence about the class of an automaton and evidence about its current state is highlighted by the first argument to the type constructor *InState*. Unlike the first argument of the *ClassOf* constructor (which can be some type-level name α::*N*), the first argument of *InState* is an *expression* with an affine type !*Instance* that stands for an automaton instance. (Notice that using (T-DROP), we can subsume the type !$Instance^{\alpha}$ of an automaton instance like new *e* to !*Instance*.)

As outlined in Section 3, and described further in the next subsection, functions that correspond to AIR rules take an automaton instance a_1 (say, in state Init) as an argument, and produce a new instance a_1' as a result (say, in state Debt(0)). Importantly, both a_1 and a_1' are given the type !$Instance^{\alpha}$—that is, the association between the type-level name α and the automaton instance is fixed; it is invariant with respect to changes to the state of the automaton. Since class of an automaton never changes (both a_1 and a_1' are instances of USAC) it is safe to give evidence about the class of an instance the type *ClassOf* α USAC—i.e., evidence about the class of an automaton can never become stale. On the other hand, evidence about the current state of the automaton can become stale. If we were to type this evidence using types of the form *InStateBad* α Init, then this evidence may be true of a_1 but it is not true of a_1'. Therefore, we make *InState* a dependent type constructor that references the particular state *variable* rather than indexing it with a singleton name. This illustrates an important distinction between singleton and dependent types in our system.

5.2 Translating rules in an AIR class

Our technical report defines a translation procedure from an AIR class declaration to a λAIR signature. Space constraints preclude a full presentation of the translation judgment here. Instead, we discuss the signature that is generated for the policy of Figure 2.

Release rules. Each release rule *r* in a class declaration is translated to a function-typed constant f_r in the signature. At a high-level, the rules have the following form. In response to a request to release data *x*, protected by instance a_1, to an instance a_2, the programmer must provide evidence for each of the conditions in the rule *r*. If such evidence can be produced, then f_r returns a new automaton state a_1', downgrades *x* as specified in the policy and returns *x* under the protection of a_2. As an example, consider the full type of the Conf_coalition rule shown below.

```
Conf_coalition :
1  ∀src::N.dst::N.∀α::U.
2  (a1:!Instance^src) → (x:Protected α src) → (a2:!Instance^dst) →
3  (e1:ClassOf src USAC) → (cd:Class) → (e2:ClassOf dst cd) →
4  (e3:ActsFor (principal cd) Coalition) → (debt:Int) →
5  (e4:InState a1 (Debt {USAC} {debt})) → (e5:LEQ debt 10) →
6      (!Instance^src × !Instance^dst × Protected α dst)
```

The first two lines of this type were shown previously—*x* is the data to be released from the protection of automaton *a*1 (with type-level name *src*) to the automaton *a*2 (with type-level name *dst*). At line 3, the argument *e*1 is evidence that shows that the source automaton is an instance of the USAC class; *cd* is another class object and *e*2 is evidence that the class of destination automaton is indeed *cd*. At line 4, *e*3 stands for evidence of the first condition expression, which requires that the owning principal of the destination automaton acts for the *Coalition* principal. Line 5 contains evidence *e*4 that *a*1 is in some state Debt(*debt*), where, from *e*5, $debt \leq 10$. The return type, as discussed before, contains the new state of the source automaton, the destination automaton *a*2 threaded through from the argument, and the data value *x*, downgraded according to the policy and with a type showing that it is protected by the *dst* automaton.

Transition rules. Each transition rule *r* in a class declaration is also translated to a function-typed constant f_r in the signature. However, instead of downgrading and coercing the type of some datum *x*, a transition function only returns the new state of the source automaton and an unchanged destination automaton. That is, instead of returning a three-tuple like Conf_coalition, a transition rule like Conf_init returns a pair (!$Instance^{src}$ × !$Instance^{dst}$), where the first component is the new state of the source automaton and the second component is the unchanged destination automaton threaded through from the argument.

5.3 Programming with the AIR API

The following example program, a revision of the program in Figure 3, illustrates how a client program interacts with the API generated for an AIR policy.

```
1  let a1:!Instance^src, x_a1 = get_usac_file_and_policy () in
2  let a2:!Instance^dst, channel = get_request () in
3  let a1,USAC,ca1_ev = class_of_inst [src] a1 in
4  let a2,ca2,ca2_ev = class_of_inst [dst] a2 in
5  let actsfor_ev = acts_for (principal ca2) Coalition in
6  let a1, Debt{USAC}{debt}, a1_state_ev = state_of_inst [src] a1 in
7  let debt_ev = leq debt 10 in
8  let a1',a2,x_a2 = Conf_coalition [src][dst][Int] a1 x_a1 a2
9                               ca1_ev ca2 ca2_ev actsfor_ev
10                              debt a1_state_ev debt_ev in
11   send [Int] [dst] channel x_a2
```

As previously, the first two lines represent boilerplate code, where we read a file and its automaton policy and then block

waiting for a release request. At line 3, we generate evidence a1_class_ev that a1 is an instance of the USAC class and at line 4 we retrieve a2's class ca2 and evidence ca2_ev that witnesses the relationship between ca2 and a2. At line 5, we check that the destination automaton is owned by a principal acting for the Coalition. At lines 6 and 7 we check that a1 is in the Debt{USAC}{debt}, for some value of debt ≤ 10. If all the run-time checks succeed (i.e., calls to functions like leq), then we call Conf_coalition, instantiating the type variables, passing in the automata, the data to be downgraded and evidence for all the release conditions. We get back the new state of the *src* automaton a1', a2 is unchanged, and x_a2 which has type *Protected Int dst*. We can give the channel a type such as *Channel Int dst*, indicating that it can be used to send integers to the principal that owns the automaton *dst*. The send function can be given the type shown below:

$$\mathsf{send}{:}\forall\alpha{::}U,\beta{::}N.\mathit{Channel}\ \alpha\ \beta \rightarrow \mathit{Protected}\ \alpha\ \beta \rightarrow \mathit{Unit}$$

This ensures that x_a1 cannot be sent on the channel. But, if the call to Conf_coalition succeeds, then the downgraded x_a2 has type *Protected Int dst*, which allows it to be sent.

5.4 Correctness of policy enforcement

In this section, we present a condensed version of our main security theorem and discuss its implications. The full statement and proof can be found in our technical report.

Theorem (Security)**.** *Given all of the following: (1) an* AIR *declaration D of a class with identifier* C *owned by principal P, and its translation to a signature S_D; (2) a model M_D consistent with S_D; (3) $\Gamma = src{::}N, dst{::}N, s : !\mathit{Instance}^{src}$; (4) $\Gamma; s \vdash_{S_D} e : t; \varepsilon$ where $src \notin \varepsilon$; and (5) $M \vdash ((s \mapsto v)e) \xrightarrow{l_1} e_1 \ldots \xrightarrow{l_n} e_n$ where $v = \mathsf{new}\ \mathsf{Init}\ \{\mathsf{Class}\ \{\mathsf{C}\}\ \{P\}\}$. Then the string $l_1, \ldots, l_n$ is accepted by the automaton defined by D.*

The first condition relies on our translation judgment that produces a signature S_D from a class declaration D. The second condition is necessary for type soundness. Conditions (3) and (4) state that e is a well-typed expression in a context with a single free automaton $s : !\mathit{Instance}^{src}$ and two type name constants *src* and *dst*. By requiring that $src \notin \varepsilon$ we ensure that e does not give the name *src* to any other automaton instance. This theorem asserts that when e is reduced in a context where s is bound to an instance of the C class in the Init state, then the trace $l_1, \ldots, l_n$ of the reduction sequence is a word in the language accepted by the automaton of D.

The trace acceptance judgment has the form $A; D \models l_1, \ldots, l_n; A'$, which informally states that an automaton defined by the class D, in initial state A, accepts the trace $l_1, \ldots, l_n$ and transitions to the state A'. Recall that the trace elements l_i record base terms B that stand for security-relevant actions and sets of values that certify that the action is permissible. The trace acceptance judgment allows a transition from A to A' only if each transition is justified by all the evidence required by the rules in the class. This condition is similar to the one used by Walker [2000].

6. Related Work

The specification and enforcement of policies that control information release has received much recent attention. Sabelfeld and Sands [2005] survey many of these efforts and provide a useful way of organizing the various approaches. AIR policies address, to varying degrees, the *what*, *who*, *where* and *when* of declassification, the four dimensions identified by Sabelfeld and Sands. Most of this work approaches information release from the perspective of information flow policies [Denning, 1976], and most of the proposed security properties can be thought of as bisimulations. By contrast, our security theorem states that the program's actions are in accord with a higher-level policy, and not that these actions enforce an extensional security property (like noninterference). We believe that the two approaches are complementary. In combination, we could show a noninterference-like security theorem (e.g., noninterference until conditions [Chong and Myers, 2004], or robust declassification [Zdancewic and Myers, 2001]) while being able to reason that a high-level protocol for releasing information is correctly followed.

AIR policies are defined separately from programs that use them, allowing them to be reasoned about in isolation. Most related work embeds declassification policies within programs that use them, obscuring high-level intent. One exception is work on *trusted declassifiers* [Hicks et al., 2006]. Here, all possible information flows are specified as part of a graph in which nodes consist of either downgrading functions or principals, and edges consists of trust relationships. Paths through the graph indicate how data may be released. AIR classes generalize this approach in restricting which paths may occur in the graph, and in specifying release conditions in addition to downgrading functions.

Chong and Myers [2004] propose declassification policies as labels consisting of sequences of atomic labels separated by conditions c. Initially, labeled data may be viewed with the privileges granted by the first atomic label, but when a condition c is satisfied, the data may be relabeled to the next label in the sequence, and viewed at its privileges. Declassification labels are thus similar to AIR classes, with the main difference that our approach is more geared toward run-time checking: we support dynamically-checked conditions (theirs must be provable statically) and run-time labels (theirs are static annotations).

Security automata were first proposed by Schneider [2000] as a means of specifying and enforcing safety properties. AIR policies are actually a more general form of security automata called edit automata [Ligatti et al., 2003] because they may modify data before releasing it. To our knowledge, no prior work has used automata to specify the protection level and release conditions of sensitive data. Walker [2000]

defines a type-based approach for enforcing security automata policies in which the definition of a single automaton is embedded in the type-checking judgment. Our approach allows multiple automata policies to be easily defined separately. Automata policies have also been enforced using inlined reference monitors, as in SASI and PSLang/PoET [Erlingsson, 2004]. Our approach is in contrast with SASI in that we support local policy state—Erlingsson identifies SASI's global policy restriction as a main obstacle towards making it practical. PSLang/PoET does support local policy state, but unlike λAIR, PSLang/PoET augments the run-time representation of protected data to include the policy. Dynamic labels in λAIR are more expressive (as discussed in Section 3, we can easily enforce secret sharing policies on related data) and provide a way to verify that automata and protected data are always correctly manipulated. As such, one could imagine putting λAIR to use to certify that IRMs correctly enforce their policies.

There has also been much work on tracking the state of objects in types, dating back to Strom and Yemini [1986]. The calculus of capabilities [Crary et al., 1999] provides a way of tracking typestate, using singleton and linear types (a variant of affine types) to account for aliasing. The Vault [DeLine and Fähndrich, 2001] and Cyclone [Jim et al., 2002] programming languages implement typestate checkers in a practical setting to enforce proper API usage and correct manual memory management, respectively. λAIR's use of singleton and affine types is quite close to these systems. However, in these systems the state of a resource is a static type annotation, while in λAIR a policy automaton is first-class, allowing its state to be unknown until run time. Additionally, λAIR's use of dependent types permits more precise specifications, which is useful for certifying authorization decisions.

Finally, our automata-based polices are related to stateful authorization policies used in trust management [Chapin et al., 2008, Becker and Nanz, 2007]. Certified evaluation of authorization logic has also been advocated for trust-management systems [Jim, 2001]. While more investigation is required, we believe that the core constructs of λAIR are general enough to verify the enforcement of trust-management policies as well.

7. Conclusions

This paper has presented AIR, a simple policy language for expressing stateful information release policies. We have defined a core formalism for a programming language called λAIR, in which stateful authorization policies like AIR can be certifiably enforced. Our work takes an important step towards the construction of software that can be verified to correctly enforce high-level information-release policies. In future work, we plan to add support for λAIR-style policy enforcement to our secure web-programming language, SELINKS [Swamy et al., 2008].

References

M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. In *ESORICS '07*. Springer-Verlag, 2007.

P. Chapin, C. Skalka, and X. S. Wang. Authorization in trust management: Features and foundations. *ACM Computing Surveys*, 2008. To appear.

P.-C. Cheng, P. Rohatgi, C. Keser, P. A. Karger, G. M. Wagner, and A. S. Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *IEEE S & P*, 2007.

S. Chong and A. C. Myers. Security policies for downgrading. In *CCS*. ACM Press, 2004.

S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release, July 2006.

K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL*, 1999.

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *SIGPLAN Not.*, 36(5), 2001.

D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, 2004. Cornell University.

M. W. Focke, J. E. Knoke, P. A. Barbieri, R. D. Wherley, J. G. Ata, and D. B. Engen. Trusted computing system. United States Patent No. 7,103,914, 2006. issued to BAE Systems Information Technology LLC.

B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification:: high-level policy for a security-typed language. In *PLAS '06*, 2006.

B. Hicks, T. Misiak, and P. McDaniel. Channels: Runtime system infrastructure for security-typed languages. In *ACSAC*, 2007.

T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, 2001.

T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Tech. Conf.*, 2002.

J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *IJIS*, 2003.

J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *PLDI*, 2006.

A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05*, 2005.

F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.

N. Swamy and M. Hicks. Verified enforcement of automaton-based information release policies, 2008. CS-TR-4906, CS Dept., U. Maryland.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE S & P*, 2008.

United States Department of Defense. Department of defense directive number 5230.11, 1992.

D. Walker. A type system for expressive security policies. In *POPL*, 2000.

S. Zdancewic and A. C. Myers. Robust declassification. In *CSFW*, 2001.

L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST*, 2004.