Collaborative End-to-end Enforcement of Fine-grained Information Sharing Policies in Distributed Systems

Nikhil Swamy
Microsoft Research

Executive Summary

Reliable and timely sharing of information across a community of collaborating principals is an integral part of Microsoft's vision of the "new world of work" [30]. Examples of Microsoft's investment in this vision abound. For one, *Sharepoint* specifically aims to *share information assets across teams, departments, and organizations while maintaining IT control*. Tools like *One Note* allow information from disparate software applications to be conveniently aggregated in a form that can be shared across a community of users. Longer-range initiatives like *HealthVault* target the sharing of patient medical records across a wide array of organizations, ranging from hospitals and insurance provides to employers and the patients themselves. Microsoft's partnership with consortiums like the *Trans-global Secure Collaboration Program* aim to build a platform for multi-national secure information sharing between government and industry.

Despite its prevalence, widespread information sharing is, clearly, a two-edged sword. While ready access to relevant information can make a collaboration more effective, uncontrolled sharing of digital assets raises many security concerns, e.g., the unintended dissemination of *HealthVault* records can compromise a patient's privacy. This paper describes work currently underway that has as its goal the formal verification of security properties for distributed information-sharing applications. Our work applies to the setting where principals have incentives (such as legal contracts) to abide by the security policies placed by data custodians. In this setting, we wish to make it possible for principals to collaborate with each other (e.g., by sharing security-critical software) and enforce a system-wide security policy with a high-degree of assurance.

We aim to address a number of concerns. As a first measure, we control information sharing by protecting resources by a formally specified claims-based access-control policy. Going further, we also address the specification and enforcement of usage-control policies so that a custodian of a resource can retain some control over how a resource is used after access has been granted, e.g., to prevent further dissemination of data. In order to promote as much information sharing as possible without compromising security, policies are applicable at a fine granularity, e.g., it will be possible to apply security controls to small fragments of documents rather than only to entire documents. As the number of data sources grows, keeping track of data dependences becomes important—both for enforcing security policies as well as assisting users with making sense of complex data sets. We aim to provide principled ways of tracking data provenance so that accurate records about where a piece of data originated and how it changed can be reliably maintained. Other concerns include making sure that security mechanisms like cryptography are properly used.

Concretely, our work consists primarily of three aspects: the formal specification of security policies; verifying the security of source programs; and, proving the security of low-level code.

Formal specification of information-sharing policies. In order to construct formal proofs of security, we require a means of specifying of a security policy. Towards this end, we have been investigating the use of SecPAL [7] and DKAL [25], two closely related authorization logics developed at MSR, for the specification of fine-grained access- and usage-control policies. While there are subtle differences, both these languages make it possible to formally specify intricate authorization requirements in a concise and intuitive manner.

Programming languages to verify the enforcement of security policies. We are currently implementing a new dialect of the F# programming language (called FINE) which will make it possible to prove that a system implementation correctly enforces a security policy. While FINE is general-purpose language, our intention is for FINE to be used primarily in the construction of security monitors, i.e, software components that mediate access to system resources by interposing the appropriate security enforcement code on each read/write to these resources.

Type-preserving compilers for verifiably secure low-level code. We are developing a compiler that translates FINE source programs to .NET assemblies in a provably secure manner. Our compiler uses a "proof-

carrying" methodology, which makes it possible for a code consumer to verify (using a simple type checker, similar to the existing .NET bytecode verifier) that a .NET assembly correctly meets a security policy of the code consumer's choosing.

The successful completion of these three strands of work will open the door to novel system architectures for enforcing fine-grained information sharing policies across a distributed system. One architecture we propose exploring is for principals to construct security monitors by combining code that enforces their own policies with the proof-carrying modules that enforce the usage control policies of other principals. Although developed piecemeal by principals with competing interests, our tools will be able to prove that security monitors correctly enforce end-to-end information-sharing policies throughout a distributed system.

1 Introduction

Whether due to economic incentives, legislation, or political realities, large organizations, often with competing interests, must share sensitive information across administrative boundaries in order to carry out their daily operations. For example, in the aerospace industry, military contracts are often shared between rival manufacturers requiring organizations to share critical elements of designs across administrative domains, while still protecting their trade secrets [46]. Similar situations arise in the management of medical records [33], in e-commerce, in the outsourcing of software development, and in military coalitions [18, 11].

While protecting their own information assets from improper use is the primary goal, principals are also expected to respect the security policies of their partners, e.g., a partnership may be bound by legal contracts that place restrictions on how shared data is to be used. Failure to properly carry out their contractual obligations (e.g., due to faults in the software that manipulates shared data) can have practical consequences, such as costly legal action. As such, organizations require assurance that their software correctly enforces both their own security policies as well as those of their partners. Towards this end, we advocate a program of research with the aim of formally verifying that security-critical software, potentially assembled from components authored by multiple principals, is in compliance with a security policy.

An example scenario. Consider a situation in which a defense contract for the manufacture of an aircraft is shared between two companies, *Lockheed* and *Airbus*. In this scenario, *Lockheed* appoints the services of a third party, *TechWriters*, to assist with documentation for this project. Employees of *TechWriters* require access to critical design documents from both companies. However, each company would like to ensure that its contributions to the project are properly accounted for in all documentation produced by *TechWriters*, i.e., that software used by *TechWriters* should track information flows so that documents always include metadata indicating the source (or provenance [14]) of their data. Thus, even after releasing information to a partner, the owner of the information wishes to exercise some control over how that information is used. Our example, though fictitious, is inspired by some of the requirements of the *Trans-global Secure Collaboration Program* [46].

We wish to make it possible for each principal in this scenario to verify that their software is in compliance both with their own policy and with those of its partners. As such, we intend to provide tools that, say, *Lockheed* can use to verify that its in-house policy enforcement software is in compliance with its own security policy. Additionally, we aim to make it possible for principals to confidently integrate third-party software with their security critical code so as to enforce originator controls [36]. For example, a reference monitor implemented by *Lockheed* can use libraries provided by *Airbus* to allow *Airbus* to control how *Lockheed* uses data owned by *Airbus*.

1.1 Outline of proposed work

The *Trans-global Secure Collaboration Program* (TSCP) is a government-industry partnership chartered with "developing secure solutions for today's most critical Aerospace and Defense issues: affordably mitigating multi-national compliance and IT security risks inherent in large-scale, collaborative programs." We intend to draw on concrete scenarios from the TSCP to guide our research.

Formalizing authorization policies in DKAL. (Section 2.1) Any software verification task begins with a formal specification of requirements. Over the course of the last fifteen years, researchers have proposed a variety of policy languages [2, 51, 50, 1, 3, 7, 25, 23, 10, 29, 27, 22, 15] to formally specify security policies in distributed systems. Despite promising to simplify the management and enforcement of policies, these frameworks have yet to see widespread use. There are many possible reasons: some languages have inconveniently verbose syntax (XACML or XrML), others have inefficient decision procedures or are simply undecidable (PolicyMaker, AF logic), while most include complex notions of trust and delegation that are hard to use correctly.

New authorization logics directly address many of these concerns. For example, SecPal [7] and DKAL [25] both have simple, terse syntax and are polynomially decidable. Additionally, a useful fragment of DKAL has been shown to be decidable in near-linear time. DKAL is also attractive for its ability to easily derive notions like trust and delegation rather than include these as primitives. As such, we propose to ground our work by using DKAL to specify distributed authorization policies. However, much of the remainder of our work should apply equally to policies specified in logics like SecPal, DCC [1], and others.

Modular enforcement of label-based policies. (Section 3) Policies like provenance tracking and dissemination controls are most naturally specified by tagging sensitive data with security labels. We propose to build upon an approach developed in the context of the FABLE calculus [41] to allow the semantics of security labels to be specified using code libraries that are called *enforcement policies*. Enforcement policy libraries define an API that limits how labeled data can be used; a type checker ensures that application programs always use this API correctly. This approach has been shown to be powerful enough to encode a range of policy idioms, including information flow controls, provenance tracking, and automaton-based information release policies [40].

We propose an architecture in which principals collaboratively enforce each others' label-based usage-control policies. Principals that wish to share labeled data with others, while still exercising originator controls on that data, can publish enforcement policy libraries that implement the controls they wish to apply. Recipients of labeled data can link these with their software in order to comply with the policies of the data source. This approach has the added benefit of enabling the integration of label-based policies from multiple principals, each with its own syntax and semantics for labels, e.g., the different hierarchies of multi-level confidentiality labels used by the US [12] and UK[1] governments could each be specified by separate enforcement policy libraries which could then be applied in concert to a piece of software.

Type-preserving compilation of security-typed languages. (Section 4) Of course, prior to linking third-party libraries with their security-critical code, principals will want assurance that these libraries are safe. We plan to use type-preserving compilation [35, 31] (a technique to ensure the safety of low-level code) to make it possible for principals to confidently share code modules and collaboratively enforce end-to-end policies. Type-preserving compilation has matured to the extent that it has now been used to certify that a large .NET compiler preserves memory safety [16]. We aim to advance this work so as to be able to verify that low-level code (such as an enforcement policy library) correctly enforces a high-level security policy. In addition to enabling cross-domain sharing of enforcement policy code, type-preserving compilation produces verified code with a small trusted computing base. With this technology, bugs in a compiler implementation no longer pose a threat to system security.

Verifying the enforcement of rich authorization policies, even at the level of source programs, has traditionally been out of reach. This is largely due to the wide gap between the abstractions used in a policy specification and those used by a system implementation. However, recent work in security-typed programming has produced new techniques [41, 40, 26, 8] that allow rich authorization logics to be seamlessly integrated into a programming language, thereby making it possible to verify, for example, that a reference monitor correctly enforces an authorization policy specified in DKAL.

Of these new security-typed programming languages, the F7 language [8], through its reliance on an external theorem prover, is the most convenient for source-level programming. We intend, first, to extend F7

[1]http://en.wikipedia.org/wiki/Classified_information_in_the_United_Kingdom

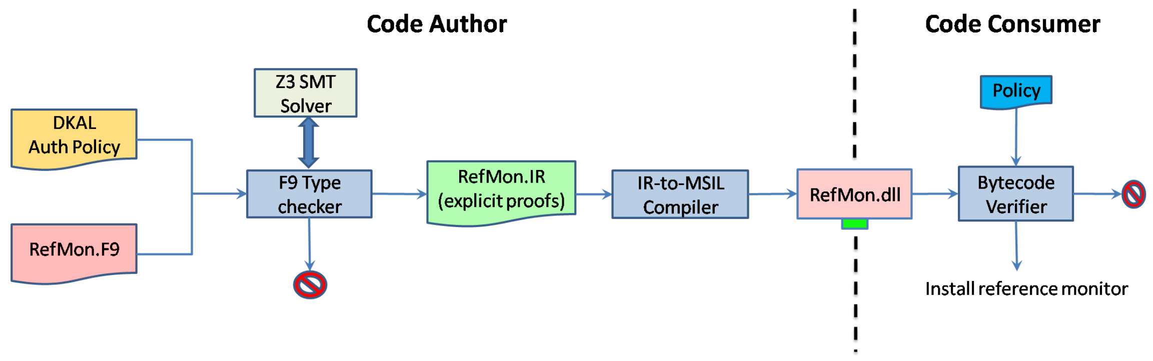


Figure 1: Verification workflow

with support for additionally enforcing label-based security policies in the style of FABLE. We then propose to develop a tool chain in which source programs in this extension of F7 can be compiled to an intermediate representation based on a calculus called FLAIR [40], a generalization of FABLE. From this intermediate representation, we propose to enhance existing work on type-preserving compilation to produce binaries that can be easily checked (e.g., with a lightweight type-checker) to satisfy the security properties verified for source programs.

Verified implementations of claims-based identity protocols. (Section 5) Identity in loosely-federated distributed systems can be hard to pin down. The lack of a widespread identity infrastructure may have been one stumbling block in the adoption of authorization logics—authorization without authentication (to which identity is central) makes little sense. The appearance of mainstream claims-based federated identity systems (including Windows Cardspace[2] and the Geneva framework[3]) opens the door to building an authorization management system upon a flexible infrastructure for claims-based authentication.

A key element of our proposed work is to integrate federated claims-based identity networks with the enforcement of DKAL authorization policies. However, the flexibility afforded by frameworks like Geneva comes at the price of complexity—verifying that these multi-party protocols are correctly implemented and provide the necessary security guarantees poses a significant challenge. Again, recent results in symbolic verification of cryptographic protocols are encouraging—verified implementations of InfoCard, the protocol suite underlying Windows Cardspace have been developed [9]. We aim to adapt these results and incorporate them into a common framework for type-based verification of implementations.

In summary, we contend that several recent advances in the theory of programming languages and the design of authorization logics mean that the time is ripe for a renewed effort to tackle distributed system security. This paper proposes to bring together several threads of research in a comprehensive framework to reliably enforce end-to-end security policies in distributed systems. Figure 1 presents an overview of our verification workflow. The remainder of the paper describes each of the elements in detail.

2 A Simple Distributed Authorization Scenario

In this section we present a small DKAL policy that formalizes the authorization requirements for the example scenario described in Section 1 . We then sketch an architecture for a system in which this policy is

[2]http://msdn.microsoft.com/en-us/library/aa480189.aspx
[3]https://connect.microsoft.com/site/sitehome.aspx?SiteID=642

(T1)	*TechWriters*	:	*Alice* MemberOf *JSFTeam*
(L1)	*Lockheed*	:	*Lockheed* Shares *Design.doc*
(L2)	*Lockheed*	:	*TechWriters* TrustedOn (p MemberOf *JSFTeam*)
(L3)	*Lockheed*	:	p MemberOf *JSFTeam* to $p \leftarrow p$ MemberOf *JSFTeam*
(L4)	*Lockheed*	:	p TrustedOn q Said p MemberOf *JSFTeam*
(L5)	*Lockheed*	:	p CanDownload Labeled(x, Provenance(*Lockheed*)) $\leftarrow$ p MemberOf *JSFTeam*, *Lockheed* Shares x
(A1)	*Airbus*	:	*Airbus* Shares *Part1.gif*
(A2)	*Airbus*	:	*Lockheed* TrustedOn (p MemberOf *JSFTeam*)
(A3)	*Airbus*	:	p TrustedOn q Said p MemberOf *JSFTeam*
(A4)	*Airbus*	:	p CanDownload Labeled(x, Provenance(*Airbus*)) $\leftarrow$ p MemberOf *JSFTeam*, *Airbus* Shares x

Figure 2: A small example policy in DKAL

to be enforced, including an example workflow that results in *Alice*, an employee of *TechWriters*, successfully downloading data from *Lockheed* and *Airbus*. The architecture and workflow also serve to illustrate a number of threats to the security of the system. This section concludes with an outline of the main security threats to the system and describes how our proposed work addresses each of these concerns.

2.1 Specifying an Authorization Policy in DKAL

DKAL is a new authorization language that exceeds many prior authorization languages in expressiveness, while still admitting efficient decision procedures for authorization queries. This section informally illustrates some of the features of DKAL by example.

2.1.1 Basic DKAL Concepts

DKAL posits the notion of an *infon* as the basic unit of information. Statements made by principals represent infons, e.g., the statement "*Lockheed* Said *Alice* CanDownload *XYZ*" is an infon which can be communicated to *Alice*. On receiving such an infon i from *Lockheed*, *Alice* simply concludes that *Lockheed* said i, but she does not necessarily believe that i is true—the separation of speech from knowledge is a key feature of DKAL.

DKAL incorporates a notion of knowledge by using a relation Knows between principals and infons, e.g, (Knows *Alice* i) is a fact about *Alice*'s knowledge of the infon i. A principal p knows infons i either by directly asserting them or by using the TrustedOn relation. DKAL includes the inference rule *(TdOn)* below to connect the TrustedOn and Knows relations:

$$\textit{(TdOn)} \qquad \mathsf{Knows}(q, p\ \mathsf{TrustedOn}\ i)\ \text{AND}\ \mathsf{Knows}(q, p\ \mathsf{Said}\ i) \Rightarrow \mathsf{Knows}(q, i)$$

This rule states that if a principal q trusts another principal p to make statements i; and, if at some point q knows that p did in fact say i; then, q knows (believes) i to be true. The example policy of the next section will illustrate a use of TrustedOn to delegate the task of authenticating users from one principal to another.

2.1.2 A Simple DKAL Policy for Cross-Domain Information Sharing

Figure 2 shows a simple DKAL policy that is intended to capture the informal requirements of the example scenario described in Section 1. The policy consists of a number of statements of the form $p{:}A$, each of which represents an assertion A made by a principal p. The policy of the *TechWriters* principal is particularly simple. Assertion (T1) just states that *Alice* is a member of a group *JSFTeam*.

The *Lockheed* principal's policy aims to restrict access to resources owned by *Lockheed* to principals who are known to be part of a privileged group, *JSFTeam*. *Lockheed*'s policy begins with a set of facts of the form (L1) asserting that *Lockheed* is willing to share specific documents.

Assertions (L2-L4) have to do with how *Lockheed* manages claims about the identities of principals. The assertion (L2) states that the contractor *TechWriters* is trusted to state that a principal p (presumably one of its employees) is a member of the *JSFTeam*. That is, using the inference rule *(TdOn)*, if *Lockheed* can conclude that *TechWriters* Said *Alice* MemberOf *JSFTeam*, then, *Lockheed* believes this fact to be true. The assertion (L3) is a rule that states that *Lockheed* is willing to certify that p is a member of the *JSFTeam* (to p), so long as *Lockheed* knows this fact to be true. Assertion (L4) may appear strange—it states that any principal p can claim that another principal q certified that p is a member of the *JSFTeam*. Obviously, assertions like (R3) must be used with care—*Lockheed* must be careful to check (using, say, digital signatures) that p does not falsely claim that it was granted a permission by q. However, the DKAL policy leaves unspecified the implementation details that ensure the authenticity of claims.

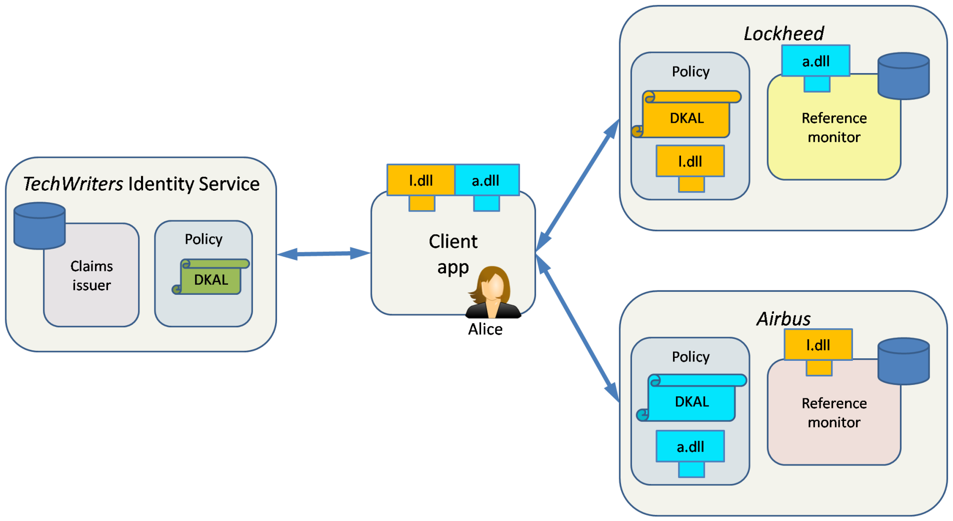
The final rule (L5) states that all files that *Lockheed* serves to members of the *JSFTeam* should be tagged with a security label of the form Provenance(*Lockheed*). The intention is that recipients of labeled data will ensure that all operations on labeled data properly keep track of metadata recording that this data originated from *Lockheed*. Of course, proper enforcement of such a label-based policy requires first, that the recipient understand what the label means, and that its interpretation of the meaning be consistent with the intended meaning of *Lockheed*. The DKAL policy itself is silent on these enforcement details, much as the DKAL policy is silent about the mechanism by which cryptography ensures the authenticity of policy statements. In Sections 2.2 and 3 we discuss how the semantics of label-based policies can be specified and enforced.

In this example, the policy of *Airbus* is similar to *Lockheed*'s policy. In general, of course, *Airbus*' policy may be completely different. The assertion (A1) states that an image, *Part1.gif*, is shared by *Airbus*. The assertion (A2) states that *Airbus* only trusts *Lockheed* to define membership in the *JSFTeam* group—*TechWriters*, being a contractor to *Lockheed*, may not even be known to *Airbus*. The assertion (A3) is analogous to *Lockheed*'s assertion (L5)—*Airbus* must use cryptography to protect against forgery. And, finally, (A4) requires all downloaded resources to be tagged with *Airbus*' provenance label.

2.2 An Architecture for Collaborative Policy Enforcement

Figure 3 (a) depicts each of the principals in our scenario. At the right we show the systems of *Lockheed* and *Airbus*. Each of these systems implements a reference monitor that protects access to a database of sensitive resources, i.e., *Lockheed*'s database contains *Design.doc*, while *Airbus*' database contains *Part1.gif*.

Each system is configured using a security policy and the reference monitors are expected to enforce these policies. The policy of each principal includes the appropriate assertions from the DKAL policy of Figure 2. In addition, each policy also includes a code library that defines the security controls on labeled data once that data leaves the direct control of its owner. For example, the library l.dll is *Lockheed*'s implementation of provenance tracking for its data. This library may include functions that allow strings to be copied out of *Design.doc*. These functions would be careful to ensure that the copied string is also tagged with the label Provenance(*Lockheed*) so as to indicate the source of that string. The corresponding



2 (a)

(1)	*Alice* to *Lockheed*	:	Get(*Design.doc*)
(2)	*Lockheed* to *Alice*	:	RequireClaim(*TechWriters* said *Alice* MemberOf *JSFTeam*)
(3)	*Alice* to *TechWriters*	:	RequestClaim(*Alice* MemberOf *JSFTeam*)
(4)	*TechWriters* to *Alice*	:	IssueClaim(*Alice* MemberOf *JSFTeam*)
(5)	*Alice* to *Lockheed*	:	PresentClaim(*TechWriters* said *Alice* MemberOf *JSFTeam*)
(6)	*Lockheed* to *Alice*	:	Response(LabeledContent(*Design.doc*, Provenance(*Lockheed*)))
(7)	*Alice* to *Airbus*	:	Get(*Part1.gif*)
(8)	*Airbus* to *Alice*	:	RequireClaim(*Lockheed* said *Alice* MemberOf *JSFTeam*)
(9)	*Alice* to *Lockheed*	:	RequestClaim(*Alice* MemberOf *JSFTeam*)
(10)	*Lockheed* to *Alice*	:	IssueClaim(*Alice* MemberOf *JSFTeam*)
(11)	*Alice* to *Airbus*	:	PresentClaim(*Lockheed* said *Alice* MemberOf *JSFTeam*)
(12)	*Airbus* to *Alice*	:	Response(LabeledContent(*Part1.gif*, Provenance(*Airbus*)))

2 (b)

Figure 3: An architecture and workflow to authenticate *Alice* and to authorize her to access shared resources

enforcement policy for *Airbus*' labels is shown as a.dll.

Since *Lockheed* may have to manipulate *Airbus*' data, and vice versa, the reference monitors of each principal are shown to be linked with the enforcement policy libraries of the other. The client application of *Alice*, an employee of *TechWriters*, is shown in the middle of the figure. Since she must work with documents from both *Lockheed* and *Airbus*, her application is shown to be linked with enforcement policy libraries from both principals. Of course, enforcement policy libraries can simply be treated as another form of resource. The means by which these libraries are distributed to other principals can also be governed by a security policy. Although not depicted, *Alice*'s client may also have its own DKAL policy to protect her personal data.

At the left of the figure, we show an identity service run by *TechWriters*. This service is only meant to issue claims to authenticate its employees to *Lockheed* and *Airbus*, e.g., claims that attest that *Alice* is a member of the *JSFTeam*. As such, it does not require the enforcement policies of the other principals since it is not supposed to receive their data. It also does not publish any enforcement policies.

2.3 Authorization Workflow

The process by which *Alice* is authorized to download shared resources is an instance of Appel and Felten's *proof-carrying authorization* approach [3]. When *Alice* requests a resource she is challenged to produce a claim that proves that she is authorized to access that resource. Obtaining claims can themselves result in other challenges being posed to *Alice*. A reference monitor must check that the proofs of claims presented by principals are valid before granting access to a resource. Figure 3 (b) shows a workflow as a series of messages.

The workflow begins with *Alice* requesting *Design.doc* from *Lockheed* and, at step 2, she is challenged by *Lockheed* to produce a claim from *TechWriters* attesting that *Alice* is a member of *JSFTeam*. At step 3, *Alice* approaches her employer's identity service to request this claim and the identity server issues the necessary claim at step 4. After presenting this claim to *Lockheed* at step 5, *Alice* receives the content of *Design.doc* at step 6, labeled with *Lockheed*'s provenance label.

Alice's interaction with *Airbus* begins at step 7 when she requests *Part1.gif*. *Airbus* challenges her to produce a claim from *Lockheed*, and after fetching the claim from *Lockheed* (in steps 9 and 10), *Alice* presents the issued claim to *Airbus* in step 11. Finally, *Airbus* responds with the requested resource, but makes sure to label the content Provenance(*Airbus*), as required by the policy.

This workflow does not illustrate the manner in which the enforcement policies installed on *Alice*'s machine track provenance. Section 3 describes the working of enforcement policies is greater detail.

2.4 Mitigating Threats to Security

In this section we mention a number of threats and describe how we intend to counter them. We also mention threats that are outside the scope of the proposed research.

System security is contingent upon each principal implementing reference monitors that perform the appropriate policy checks before granting access to resources. A failure to perform the appropriate check constitutes a failure of complete mediation—there are number of ways in which this could happen.

- **Failure to issue appropriate challenges in response to a request.** Due to a software fault, *Lockheed* could respond directly to *Alice*'s request with the content of *Design.doc*. Or, *Lockheed* could challenge *Alice* to present a claim that is unrelated to the policy to be enforced. Programming errors that result in these access control bypasses are not uncommon in large systems [39]. Our use of security-typed programming languages to embed the authorization logic within a program will ensure that a reference monitor always presents the appropriate challenge.

- **Failure to properly check proofs of claims.** Issuing the appropriate challenges is only worthwhile if we can guarantee that the proofs of claims presented are valid. The soundness proofs of our type systems will guarantee that every proof is checked properly.

- **Installing malicious/incorrect enforcement policy code.** Installing third-party enforcement policy libraries can compromise the security of a reference monitor, e.g., a malicious library could provide a backdoor to sensitive data store that bypasses access control checks. Through the use of type-preserving compilation, we will allow principals to easily verify the safety of libraries before they install them.

- **Failure of end-to-end enforcement due to improper use of an enforcement policy.** Our collaborative scheme for the end-to-end enforcement of label-based policies depends on using enforcement policy libraries in accordance with their APIs. Our type-based verification technique will guarantee that application code built using our tools will always use these APIs as intended. The same mechanisms will also be used to ensure that resources are labeled properly, e.g., that *Lockheed* always serves content tagged with the Provenance(*Lockheed*) label.

Principals that choose not to use our tools may be vulnerable to each of these above threats. Of course, regardless of whether a principal is negligent or malicious, it is important to ensure that a single rogue principal cannot compromise the security of the entire system. We address these issues within the broader context of outsider threats.

Outsiders are principals that cannot be authenticated and should not have access to any part of the system. Outsiders stand to gain illegitimate access to the system by intercepting communication between principals, or by injecting messages into the network. For example, although it is convenient for DKAL policies to include statements like q TrustedOn (p said q CanDownload x), system security crucially depends on it being impossible for q to fraudulently claim that a statement was made by p. An attacker who is able to forge such a claim will be able to gain unauthorized access. We aim to defend against such threats by relying on verified implementations of cryptographic protocols. The techniques we intend to use provide secrecy and authenticity properties which generally take into account the abilities of an active adversary that may have succeeded in compromising one or more insiders.

While cryptography helps prevent outsiders from gaining access to a system, attackers may still compromise system security by exploiting other vulnerabilities, e.g., bugs in an operating system or web browser. Outsiders may also be able to violate security by exploiting side channels, e.g., network traffic, timing, or power consumption patterns. Both these kinds of threat are outside the scope of our proposed work.

3 Modular Enforcement of Label-based Security Policies

In this section we describe how principals can author libraries to enforce security policies on labeled data. Through the use of a novel type system called FABLE, we can ensure that programs use these libraries correctly, and, as a consequence enjoy useful security properties. We begin with a brief review of label-based security policies and their enforcement using type systems.

In her classic work Denning [21] developed a canonical form of label-based security policies to enforce multi-level confidentiality policies. Nearly two decades later, Volpano et al. [49] showed how, by refining the types of a programming language to include security labels, such policies could be enforced by a compiler. For example, the type $\texttt{int}^{\texttt{High}}$ represents the set of all `High`-security integers; $\texttt{File}^{\texttt{Low}}$ could represent the type of a `Low`-security file that can be read by all users. Volpano et al. define a type system that tracks the flow of information through the constructs of a core programming formalism and can detect programs that reveal information about their `High`-security inputs on `Low`-security outputs. A large body of work [38] has extended these basic ideas of security typing to incorporate information flow analyses of the programming constructs of real languages (e.g., exceptions, higher-order functions, objects etc.).

3.1 Enforcing Label-based Policies in FABLE

In our work on FABLE, we observed that a wide variety of security policies are enforceable by associating labels with data in the types, where the label expresses the security policy for that data. What varies among policies is the *specification* and *interpretation* of labels, in terms of the actions that are permitted or denied.

By allowing the syntax and semantics of labels to be defined in libraries, we benefit from the high degree of assurance provided by security typing while still retaining the flexibility to enforce a range of policies.

Enforcing a label-based policy in FABLE proceeds in two steps. First, a policy designer defines custom security labels and associates them with the data they protect using *dependent types* [4]. Next, rather than "hard-code" their semantics, policy enforcement is parametrized by a programmer-provided interpretation of labels, specified in a privileged part of the program—we call this privileged part of the program the *enforcement policy*. The type system forbids application programs from manipulating data with a labeled type directly. Instead, in order to use labeled data, the application must call the appropriate functions in the enforcement policy that interpret the labels. By verifying the interpretation of labels, and relying on the soundness of the type system, policy implementers can prove that type-correct programs enjoy relevant security properties.

For instance, a programmer can define a label High, and give a high-security integer value a type that mentions this label, Labeled Int High. As another example, the programmer could define a label ACL(Alice, Bob) to stand for an access control list and give an integer a type such as Labeled Int (ACL(Alice, Bob)). For the policy of Figure 2, we could use labels of the form Provenance(Airbus) and give values Labeled types that mention these labels. Programmers define the interpretation of labels in an enforcement policy, a library of privileged functions distinguished from the rest of the program. Thus, in order to capture the intuition that an integer with the type Labeled Int (ACL(Alice, Bob)) is only to be accessed by Alice or Bob, one writes an enforcement policy function like the following:

```
policy access_simple (acl:Lab, x:Labeled Int acl) = if (member user acl) then unlabel x else −1
```

Here, access_simple takes a label acl as its first argument (like ACL(Alice, Bob)), and an integer protected by that label as its second argument. If the current user (represented by the variable user) is a member of x's access control list acl (according to some function member, not shown), then x is returned with its label removed, expressed by the syntax unlabel x, which coerces x's type to Int so that it can be accessed by the main program. If the membership test fails, it returns -1, and x's value is not released.

The slightly more complex enforcement policy function below shows how to track provenance when concatenating strings.

```
policy prov_cat (p:Lab, x:Labeled String p, q:Lab, y:Labeled String q) = relabel (strcat(unlabel x, unlabel y)) (Union(p,q))
```

This function takes the labeled strings x and y as arguments, tagged with the labels p and q respectively. In the body of the function, we unlabel x and y, coercing their types to String before calling the strcat function from the standard library to actually concatenate the strings. To ensure that the result records the provenance of its arguments, we use the relabel operator to tag the result with the union of the two argument labels.

By preventing the main program (i.e., the non-policy part) from directly examining data with a labeled type, we can ensure that all its operations on data with types like Labeled String (Provenance(Airbus)) are preceded by calls to the appropriate policy function, which performs any authorization checks and propagates the labels as necessary to the results of the operation , i.e., the type system guarantees complete mediation.

We have shown that this basic idea in FABLE is powerful enough to encode the enforcement of various styles of access control, data provenance, and information flow policies. A generalization of FABLE, a calculus called FLAIR, goes further to handle the enforcement of policies that deal with mutable state. We discuss FLAIR in greater detail in Section 4.2.

3.2 Collaborative enforcement by sharing enforcement policies

It should be clear that properly enforcing end-to-end policies (like provenance tracking) requires intercepting all the operations that a program can perform on protected data. The semantics of such a policy is closely tied to specific operations of a program and, as such, these policies are most naturally specified as code. FABLE's enforcement policies enable just this kind of specification while the type system guarantees proper enforcement.

Our proposal to use enforcement policy libraries is similar in spirit to microkernel-style *library operating systems* where common services are linked as libraries rather than implemented in separate components. Principals implement reference monitors that enforce their own policies, and link in libraries to handle the enforcement of policies on data that belong to other principals. By allowing principals to publish enforcement policies, and to have these policies applied in the software systems of their partners, we effectively allow organizations to retain control over their data even after that data has left their system.

To make this approach work in practice, we must be careful to ensure that enforcement policy functions compose well. For example, multiple principals could use labels with the same syntax, but their corresponding enforcement policies could interpret these labels in conflicting ways. We have begun exploring simple techniques to ensure that policies always compose well [40]. However, more complex notions of composition may become necessary as policies grow more involved. Recent work on resolving conflicting policies using many-valued logics may provide some guidance [13].

Accounting for changes to policies will also require some care [43]. One advantage of specifying enforcement policies as code is that this code can periodically inspect policies published by the code authors and enforce the most recent version, e.g., *Airbus*' enforcement policy can poll *Airbus*' servers for the most current policy. Our verification methodology would have to ensure that unintended or malicious information leaks do not take place via this form of network communication. More significant updates to a policy may require the implementation and deployment of entirely different enforcement policies. Such updates could make use of existing patch-deployment infrastructure, or potentially even make use of dynamic software updates [34].

4 Type-preserving Compilation of Security-typed Languages

This section describes a tool chain that is intended to allow the construction of modular and verifiably secure reference monitors. We begin by briefly describing the F7 programming language and argue that, with extensions to support the enforcement of label-based policies, it is appropriate for verifying the enforcement of DKAL-like policies. However, F7 requires a large trusted computing base (TCB), including a state-of-the-art theorem prover. To reduce the TCB we propose translating F7 source programs to an intermediate representation called FLAIR, introduced earlier. In Section 4.2 we describe FLAIR and show how F7 programs that enforce DKAL policies can be translated to FLAIR. Finally, in Section 4.3 we discuss how to compile FLAIR programs to produce provably secure mobile code.

4.1 Verifying Security in F7 with Refinement Types

F7 is an extension of the F# programming language [44], itself a dialect of ML. Like other languages in the ML family, F7 is mostly functional and strongly typed. In this section, we sketch how to develop a small reference monitor in F7 that enforces a fragment of our example DKAL policy. Figure 4 shows an F7 listing, which we describe in several steps.[4]

[4]We alter the notation of F7 slightly to simplify the presentation and to avoid confusion with the notation of FABLE.

```
1  (* Basic DKAL constructs *)
2  type prin =
3    | Self of (pubkey × privkey)
4    | Other of pubkey
5
6  type infon =
7    | Said of prin × infon
8    | TrustedOn of prin × infon
9    | CanDownload of prin × string
10   | JSFTeam of prin
11   | Shares of string
12   | CanSayTo of prin × infon
13
14 type krel = Knows of prin × infon
15
16 (* Typed interface for Lockheed's reference monitor *)
17 val lockheed:prin
18 val techWriters:prin
19
20 type (α; p:prin) channel
21
22 type (α; uri:string) res = (id:{x:string | x=uri}, data:α)
23
24 val send: (α; p) channel →
25             {(α; uri) res |
26              Knows(lockheed, CanDownload(p, uri))} →
27             unit
28 (* Lockheed's policy *)
29 (* L1 *)
30 assume(Knows(lockheed, Shares(''Design.doc'')))
31 (* L2 *)
32 assume(forall p.
33          Knows(lockheed,
34            TrustedOn(techWriters, JSFTeam(p))))
35 (* L3 *)
36 assume(forall p.
37          Knows(lockheed, JSFTeam(p)) ⇒
38            Knows(lockheed,
39              CanSayTo(p, JSFTeam(p))))
40 (* L4 *)
41 assume(forall p,q.
42          Knows(lockheed,
43            TrustedOn(p, Said(q, JSFTeam(p)))))
44 (* L5 *)
45 assume(forall p,x.
46          Knows(lockheed, JSFTeam(p)) ∧
47          Knows(lockheed, Shares(x)) ⇒
48            Knows(lockheed, CanDownload(p, x)))
49
50 (* General DKAL inference rules *)
51 (* TdOn *)
52 assume(forall p,q,x.
53          Knows(p, TrustedOn(q, i)) ∧
54          Knows(p, Said(q, i)) ⇒ Knows(p, i))
```

Figure 4: An F7 interface to check the enforcement of *Lockheed*'s DKAL policy

4.1.1 Representing basic DKAL constructs

Figure 4 begins by defining datatypes for various basic DKAL structures like principals, infons, and knowledge. These are shown on lines 1-15. We do not make use of any F7-specific constructs in these types—similar types could be declared in any functional language.

We start by defining a representation for principals; for simplicity, we assume that all principals have public keys. The type prin is defined as an algebraic type (a kind of tagged union)—the Self constructor holds a public and private key pair, while the Other constructors holds only a public key. For example, *Lockheed*'s reference monitor would include a value Self(pk, sk) of type prin to represent the *Lockheed* principal. It would have values of the form Other(pk) to represent other principals like *TechWriters*.

Next, at line 6, we define the type infon to represent the DKAL concept of infons. This algebraic type is the type of values like Said(p, CanDownload(q, ''Foo'')), the our representation of DKAL statements like p Said CanDownload((q), "Foo"). Line 14 defines the krel type, a direct representation of DKAL's Knows relation.

4.1.2 A secure interface for sensitive operations

Lines 16-27 in Figure 2 defines an interface that must be satisfied by an implementation of *Lockheed*'s reference monitor. This interface gives special types to sensitive operations that the reference monitor may have to perform, e.g., allowing a principal to download sensitive data. These types include constraints derived from the DKAL policy and prior to performing a sensitive operation, an implementation must prove that these constraints have been met.

Lines 17 and 18 are straightforward. They simply state that *Lockheed*'s reference monitor will use the prin-typed values lockheed and techWriters to hold the key material of the corresponding principals.

Line 20 defines the type of a communication channel. This is an *indexed* type, which is parametrized by a type variable α and term variable p of type prin. That is, channel represents a family of types, where members of this family are types of the form (String, alice) channel, where alice is a prin-typed value. We will use types like (String, alice) channel to represent a network connection that can be used to send String data to the principal alice. The underlying representation of values of this type are unimportant—so, line 20 leaves the channel type abstract (i.e., there is no definition).

Next, we have to decide on a representation for resources, i.e., the data objects that are to be protected by the reference monitor. One could choose any of several representations for resources. Here, we choose to represent resources as a record with two fields, id and data. For example, the *Design.doc* resource would be represented as a record (id=''Design.doc'', data=0x0f...)—the id field contains the name of the resource and the data field contains, in this case, the raw bytes of the document.

The res type defined at line 22 is to be the type of resources. The definition is a little subtle because it makes use of F7's distinctive feature—refinement types. First, just like channel, res is really a family of types indexed by a type α and a string uri. For example, the type (bytes; ''Design.doc'') res will be the type we give to the record (id=''Design.doc'', data=0x0f...). As we will show shortly, by using the uri to index the type, we will be able to specify constraints on how resources are allowed to be used.

If we were to define res as a standard F# record, its definition would be (id:string, data:α). However, we want to ensure that the uri index of the res type is consistent with the actual uri stored in the id field of the record. We can use F7's refinement types to express this constraint. The set-comprehension notation {x:string | x=uri} stands for the type of a string value x for which the proposition x=uri is known to be true. The F7 type checker ensures that this constraint is always satisfied. For example, the only type that can be given to (id=''Design.doc'', data=0x0f...) is (bytes; ''Design.doc'') res, as desired.

Finally, at line 24, we give a type to the send function that a program can use to send resources that hold data of type α to a principal p. The first argument to send is channel of type (α, p) channel. The second argument is a resource of type (α, uri) res. However, not any resource of this type will do. We, of course, want to ensure that we only send resources that we know p is authorized to download. So, we use a refinement type again, and constrain the second argument of send to only be resources for which the formula Knows(lockheed, CanDownload(p, uri)) is known to be true—the next section describes how such formulas are proved. The send functions accepts these two arguments, sends the resource to p, and can returns a dummy Unit value. The types allow us to ensure that the necessary authorization checks have been performed.

Note that we have made a simplification here by removing the constraint that requires the resource to be labeled with *Lockheed*'s provenance label. Expressing and enforcing label-based policies in F7 is not particularly easy. Part of our proposed work is to extend F7 with this capability.

4.1.3 Proving the security of implementations

On its own, a formula like Knows(lockheed, CanDownload(p, uri)) has no meaning in F7. They can only be interpreted in the context of a set of rules describing the DKAL policy. On the left side of Figure 4, we show how assumptions about a DKAL policy can be introduced into an F7 program.

Lines 28-48 show assumptions that correspond to each of the assertions in the DKAL policy of Figure 2. Each assumption is a fairly direct translation from DKAL, so we do not describe them in detail. Lines 50-54 show a translation of the DKAL house rule (TdOn) that interprets the TrustedOn relation. A full implementation would include assumptions about the other house rules of DKAL. Of course, assumptions must be used with care—the validity of the various constraints induced by refinement types depends on the validity of the assumptions.

The snippet of code below illustrates how an implementation is checked.

```
val designDoc: (bytes; ''Design.doc'') res
val monitor: p:{ p:prin | Knows(lockheed, Said(p, Said(techWriters, JSFTeam(p))))} → (bytes; p) channel → unit
let monitor p c = send c designDoc
```

Here, we show a function monitor which is executed in a context where the variable designDoc has the type (bytes; ''Design.doc'') res. The monitor function expects two arguments, p and c. The first argument represents a principal, but the refinement type of p is very specific. In particular, p is a principal for which the formula Knows(lockheed, Said(p, Said(techWriters, JSFTeam(p)))) (call it ϕ) is known to be true. The second argument c is a channel to the principal p. We ignore for the moment the issue of how specific formulas like ϕ come to be known–in Section 5 we discuss how cryptography can be used to safely introduce such formulas as assumptions. The point here is to show how in the presence of general policy assumptions, and with specific facts like the formula ϕ, F7 proves that implementations are secure.

In the body of monitor function, we call the send function whose type was discussed in the previous section. For this program to be secure, we must be able to show that Knows(lockheed, CanDownload(p, ''Design.doc''). In order to establish this goal, F7 collects all the assumptions and known facts in the current context (i.e, assumptions L1-L5, *TdOn*, the specific formula ϕ, and the formulas that appear in the definitions of type constructors like res) and leaves it to the Z3 theorem prover [20] to decide if the goal can be proved from the assumptions. In this case, Z3 would successfully find a proof, using all the assumptions (except (L3)) in a chain of deductions that results in the goal.

The use of Z3 to discharge proof obligations is one of the reasons why programming in F7 is simpler than in other languages of comparable expressiveness, e.g., FABLE, FLAIR, or Aura [26]. Rather than explicitly construct proofs of the authorization decisions, programmers get to write relatively familiar code like send p c while Z3 takes care of cumbersome proof manipulations. However, this approach has some downsides. First, Z3 must be in the TCB, e.g., a bug in Z3 could compromise the security of the system. Second, in the mobile code setting (e.g., requiring principals to use each others' enforcement policy libraries), we would like to allow code consumers to be able to independently verify that the code they download is secure. Requiring them to implement state-of-the-art theorem provers to do so defeats the purpose. Finally, both ourselves [42] and others [48] have argued that it is useful to construct explicit authorization proofs. For instance, these proofs can be logged and inspected later if an audit becomes necessary. Additionally, in a proof-carrying authorization regime, principals may be required to present complete derivations of authorization decisions, rather than simply presenting digital certificates that attest to the satisfaction of a goal. We stand to gain the best of both worlds (i.e., ease of programming and a small TCB etc.) by compiling F7 source programs to FLAIR, a simpler intermediate language in which all authorization proofs are explicit.

4.2 FLAIR: A Core Calculus with Explicit Authorization Proofs

In this section we give a flavor of the FLAIR calculus and show how F7 source programs that enforce DKAL policies can be translated into a core calculus with explicit authorization proofs. FLAIR aims to be a minimal calculus, suitable as an intermediate representation for a compiler. To that end, the fragment of FLAIR shown here is a fairly standard dependently-typed second-order lambda calculus, i.e., System F(2) [24]. Versions of (non-dependently-typed) System F have previously been used in several type-preserving compilers, including the Glasgow Haskell Compiler [28], and the TIL [45] and TILT [37] compilers for Standard ML. We expect a type-preserving compiler for FLAIR to be able to profitably use this extensive body of prior work as a point of departure.

Signatures	S	$::=$	$B{:}t \mid T{::}K \mid S, S'$
Terms	e	$::=$	$x \mid B \mid \lambda x{:}t.e \mid e\ e' \mid \Lambda\alpha.e \mid e\ [t] \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \Rightarrow e_3\ \mathsf{else}\ e_4$
Types	t	$::=$	$\alpha \mid (x{:}t) \to t' \mid \forall\alpha.t \mid T \mid t \Rightarrow t' \mid t\ t' \mid t\ e$
Kinds	K	$::=$	$\star \mid \star \to K \mid t \to K$

Figure 5: Syntax of FLAIR (partial)

Figure 5 shows the syntax for a fragment for FLAIR. FLAIR programs are type checked in the presence of a signature S, which represents a standard prelude that gives types t to base terms B and also defines a set of type constructors T and their kinds K. The terms in the language include (in order) variables, base term constants, lambda abstraction, application, type abstraction, type application, and a pattern matching construct.

The syntax of types includes type variables α; dependent function types are written $(x{:}t) \to t'$, where x names the argument of type t and is in scope within the return type t'; $\forall\alpha.t$ is a type t universally quantified over all types α. We also include type constructors T which are organized into three kinds. Nullary type constructors are given the kind $\star$, the kind of normal types. For example, the nullary type constructor Int has kind $\star$ and can be used to classify base terms like O, the data constructor that stands for the integer 0. Data constructors themselves may take arguments, e.g, the integer 1 can be represented as $Succ(O)$, an application of the $Succ$ constructor to the term O. We give $Succ$ the type $Int \Rightarrow Int$ to indicate that it constructs a term of type Int from an argument of type Int. Of the remaining two kinds of type constructors, $\star \to K$ classifies type constructors t that can be applied to any normal type t' using the notation $t\,t'$. Finally, the kind $t \to K$ is the kind of a dependent-type constructor t'; these may be applied to any term term e of type t using the notation $t'\ e$. For example, the label terms like Acl(Alice, Bob) that we used in Section 3 can be given the type Lab which has kind $\star$. The Labeled type constructor, which we used to construct types like Labeled Int (Acl(Alice,Bob)) can be given the kind $\star \to$ Lab $\to \star$.

4.2.1 Translating F7 to FLAIR

Translating F7 to FLAIR essentially requires handling two constructs—refinements and assumptions. Here we briefly sketch the main ideas.

Recall that our objective is to make explicit in FLAIR those proof-related operations that are implicit in an F7 program. For example, the F7 refinement type {i:infon | Knows(lockheed, i)} is the type of infons i for which the proposition Knows(lockheed, i) can be proved by Z3 to be true. Our translation to FLAIR will the proof explicit by representing this type as a pair, where the first element of the pair is the infon i and the second element is a term that represents a proof of the relevant proposition. Our first task then is to give propositions a type in FLAIR and then, using the Curry-Howard isomorphism, represent proofs of these propositions (types) as terms.

The sub-language of propositions that can appear in F7 refinements is untyped, so a specific type of propositions is not needed in F7. While this is convenient at the source level, it does permit writing down refinements that do not make sense, e.g, {x:string | x = 17}. FLAIR is stricter; so, when translating F7 programs that enforce DKAL policies to FLAIR, we will define the type constructor Knows::Prin $\to$ Infon $\to \star$ to represent the propositions about the knowledge of principals. Here, Prin and Infon are themselves nullary type constructors in FLAIR. For example, the F7 type {i:infon | Knows(lockheed, i)} will be translated to the FLAIR type $\forall\alpha$.((i:Infon) $\to$ (Knows lockheed i) $\to \alpha$) $\to \alpha$. This latter FLAIR type is simply a higher-order encoding of a pair type—using the more familiar notation of a dependent pair, it may be written (i:Infon $\times$ Knows lockheed i)

Next, we need a way to construct explicit proofs in FLAIR and to ensure that every well-typed proof is logically valid. To achieve this, we will translate every assumption in an F7 program into a term constant in FLAIR. For example, the DKAL inference rule for the TrustedOn relation will be represented in FLAIR using the following function-typed constant.

tdOn: p:Prin $\rightarrow$ q:Prin $\rightarrow$ i:Infon $\rightarrow$ Knows q (TrustedOn p i) $\rightarrow$ Knows q (Said p i) $\rightarrow$ Knows q i

Every implicit proof discharged by Z3 that relies on the (TdOn) assumption, will be translated into an explicit FLAIR term that includes an application of the tdOn function above. Similarly, each policy-specific assumption (e.g., L1-L5) will also be translated into term constants in FLAIR. Importantly, we need to ensure that the only way in which proof terms are built in FLAIR is through the use of these constants that represent policy rules or basic axioms. We will achieve this by ensuring that datatypes that will be used as propositions in proofs (like the Knows type) will have no data constructors that are usable in untrusted code.

4.3 Compiling FLAIR to Mobile Code

This final step in producing verifiably secure libraries that can be safely shared between (dis)trusting principals is to compile FLAIR programs to a low-level representation that still contains sufficient type information for verification. We are not the first to propose type-preserving compilation for security-typed code. There has been prior work on compiling security-typed dialects of Java to bytecode [5, 6] as well as more theoretical work on producing typed assembly language from a security-typed language [54, 53] and also to typed assembly languages. However, these efforts have focused mainly on the enforcement of noninterference-based information flow security policies. While these policies are of interest to us (we have shown that they can also be enforced in FLAIR), we aim to go beyond this.

Another shortcoming of prior work on type-preserving compilation of security-typed languages is that they assume that policies are statically known. However, most realistic policies are highly dynamic [47, 55], and FLAIR specifically targets dynamic label-based policies. Expressing these kinds of policies requires, at the very least, some simple forms of dependent types.

Dependently typed assembly languages have also been proposed before [52]. However, this line of work has focused primarily on enforcing memory safety properties, proving the correctness of array-bounds-check elimination optimizations etc. We are proposing to compile dependently languages to prove the correctness of security enforcement in mobile code. That is, we propose to define DTAL-S, a dependently typed assembly language for security.

We expect to have to overcome a number of challenges. For example, while it is relatively easy to include dependent types in functional source-level programs, working with dependent types in a computational model where side-effects are pervasive may be difficult. We might consider using existing work on Hoare type theory for this purpose [32]. Another concern will be efficiency of the generated code. While carrying explicit proofs in mobile code makes verification simple, it may not be so good for efficiently executable code. We imagine adapting work on using phantom types to maintain a strict phase separation between terms that have operational significance and those that do not [40], e.g., by isolating proof terms into a separate phase, these may be conveniently erased at run-time, if efficiency is a concern. Otherwise, these may be logged for auditing etc.

```
1  type (α;i:prin,u:prin) chan
2  type (;i:prin) speech = {x:infon | Said(i, x)}
3  type (;i:prin,u:prin) knowledge =
4                    {x:infon | Knows(i, Said(u, x))}
5
6  type αkey
7  type (;i:prin) skey = (;i) speech key
8  val sign: (;i) speech → (;i) skey → bytes
9
10 type (;i:prin,u:prin) vkey = (;i,u) knowledge key
11 val verify: (bytes ∗ infon) → (;i,u) vkey →
12                                  (;i,u) knowledge option
13 val send: (infon; i,u) chan → (;i) speech → (;i) skey → unit
14 let send c s signingKey =
15     let mac = sign s signingKey in
16     Net.send c (serialize(mac, s))
17
18 val recv: (infon; i,u) chan → (;i,u) vkey → (;i,u) knowledge
19 let read c verificationKey =
20     let (mac,i) = unserialize(Net.recv c) in
21     match (verify (mac,i) verificationKey) with
22       | Some k → k
23       | None → raise FailedSignatureAuth
```

Figure 6: A verifying the authenticity of messages (partial)

5 Verified Implementations of Cryptographic Protocols

In Section 4.1.3 we showed F7 can be used to check the security of implementations in the presence of types like, {p:prin | Knows(q, Said(p, Said(techWriters, JSFTeam(p))))}.In this section, as promised, we show how values of this type can be safely constructed using cryptography. Happily for us, the same mechanism in F7 that we used to check authorization policies can be applied to verifying authenticity properties. The construction we develop here follows a similar construction shown by Bengston et al [8].

5.1 A Simple MAC-based Protocol for Authenticity

Figure 6 shows a F7 listing that verifies the authenticity of messages using a simple protocol based on a message authentication code (MAC) using asymmetric keys. Our goal is ultimately to provide an interface that allows a reference monitor to receive messages m on a channel and safely conclude that m was from the intended sender. The first step towards this goal is to revise the type of channels that we used in Figure 4 so that it can be used to both send and receive messages. At line 1, we show the type (α;i,u) chan, which is the type of a channel on which the principal i can send α-typed messages to u and also receive messages of the same type from u, i.e., this is the type of a duplex channel between i and u.

Next, we define the type (;i) speech. This is the type of an infon that a principal i wants to send on a channel. Deciding that it is permissible for i to do so is a matter governed by i's authorization policy. The corresponding type on the side of the receiver of a message is (;i,u) knowledge. On receiving an infon x from u, i would like to conclude that u actually said x.

In order to safely construct values of the knowledge type, we will require messages to be signed when they are sent on a channel. Receivers will only admit a message into their knowledge after verifying the signature. At lines 5-12 we define the types of keys that will be used in this protocol. The type αkey is an abstract type of a key that can be used to sign or verify messages of type α. The type of a signing key is (;i) skey. Using the sign function a principal i can sign an infon and obtain the raw bytes of the MAC. The type of a verification key is (;i,u) vkey and is the key that the principal i uses to verify messages sent by u. The function verify takes a pair of a MAC and an infon as its first argument, a verification key as the second argument, and, if the verification succeeds, it returns a knowledge infon; if it fails, it returns a dummy value None.

At the right of Figure 6 we show the implementation of send and recv functions on channels. The first argument of send is an (infon;i,u) chan on which the principal i wishes to send a speech infon s (the second argument) to the principal u; the final argument is i's signing key. In the body of the function, at line 15, we construct a mac for the message by signing the speech infon and, at line 16, we convert the message to some low-level wire format and call the library function Net.send to send the message on c.

The recv function is used by i to receive infons from u and to admit these infons into its knowledge. The first argument of recv is the channel c and the second argument is the key i uses to verify messages from u (say, u's public key). At line 20, we call the Net.recv library function and block until a message is received. Once it is received, we unmarshall the message into a mac and an infon i. Next, we check the signature. If it succeeds, we return k, the knowledge infon; otherwise, we raise an exception.

5.2 Extensions

While this simple MAC-based protocol gives a flavor of the authenticity properties that can be checked in F7 (and, by translation in FLAIR etc.), a number of extensions are required to use this scheme in practice.

To begin with, our example protocol is not sufficiently strong to be used with DKAL. One of the key feature of DKAL is the ability of principals to quote other principals, e.g., principals can construct infons like Said(q, Said(p, x)). The protocol in Figure 6 simply requires an infon to be signed once, i.e., the principal q can construct an infon like Said(q, Said(p, x)) just by signing this with q's signing key. To prevent forgeries, we must require the nested infon Said(p,x) to contain a signature from p. Verifying a message would have to recursively check signatures within quoted infons.

In a situation where every principal can be identified by a public key, our MAC based protocol (extended to account for DKAL quotations) may be sufficient. However, in practice, we will need to support more flexible claims-based identity protocols, e.g., in our example policy, *Alice* is only identified to *Lockheed* and *Airbus* by claims that *TechWriters* issues about her and not a public key.

The protocols used in claims-based identity systems like Geneva and Windows Cardspace are substantially more complex than our simple MAC-based protocol. In a recent paper [9], we constructed formal models of the protocols underlying Windows Cardspace—some of the three-party authentication schemes require several rounds of communication, with some messages containing up to 20 different cryptographic operations like signatures, message digests, and encryptions. Verifying that these protocols are correctly implemented and that they give the desired authorization properties is decidedly non-trivial. Using a tool chain unrelated to F7, we have verified implementations of these protocols. We propose to adapt these results for use in a common framework for type-based verification in F7, FLAIR, and DKAL.

6 Project Plan

In this section, we outline a number of tasks that we intend to undertake in order to deliver the tools and technologies described in this paper. While there are some dependencies between the tasks, we expect several of these efforts to proceed in parallel.

I. Formalization of TSCP scenarios in DKAL. (Joint work with Yuri Gurevich.) We anticipate formalizing concrete descriptions of TSCP scenarios in DKAL. This formalization will have to unify a number of disparate policy elements into a single framework. For example, claims-based authentication policies in Geneva are currently specified using WSDL [17] and we will have to find ways to ensure that authentication elements in a DKAL policy are consistent with the corresponding elements of a WSDL policy. Additionally, end-to-end labeling policies like provenance tracking have, in prior work, not been integrated with authorization policies. We expect that a DKAL policy that elegantly brings together these three elements (traditional authorization, WSDL-style policies, and end-to-end labeling policies) will be of independent interest.

II. Implementation of DKAL libraries in F7. We plan continue development on a prototype that allows DKAL policies to be expressed and enforced in F7 programs. We aim for DKAL support to be provided mainly through libraries that implement core DKAL features (infons, inference rules etc.). However, some enhancements of the F7 compiler are expected. We also intend to provide library support for verified implementations of simple authentication policies, e.g., extending the MAC-based protocol described in Section 5.1 to properly handle quotations etc. We intend to use an elaboration of the example policy of Section 2.1 as a test case. It is possible that completing this work will require some substantial ingenuity and, again, may be of independent interest.

III. Formal foundations for secure compilation of source programs. This task represents the theoretical work that will be necessary before we can implement our type-preserving compilation. A paper about a dependently typed bytecode language and a preliminary implementation of a compiler is planned for submission to POPL in July 2009.

IV. Design and implementation of FINE: F7 augmented with labeling policies compiled with explicit proofs. (With Juan Chen and a summer '09 intern) With the assistance of an intern in the summer of 2009, we expect to complete an implementation of a compiler for a source language, tentatively called Fine. This compiler would also include a translation to the FLAIR intermediate language and emit .NET bytecodes that includes explicit proofs. We expect to use this compiler in the fall of 2009 and use Fine to build a prototype distributed reference monitor. A target application is a web-based document management system that enforces authorization and provenance policies. Our prior experience with SEWiki, a web-based document management system with provenance tracking [19], makes this a natural choice. One could imagine even enhancing the current implementation of SEWiki so as to make it compatible with a security infrastructure developed using our new tools. We plan for a submission to Oakland in November 2009.

V. Implementation of type-preserving compilers for F9. (Joint work with Juan Chen.) This is work that we expect to carry out in the latter part of 2009 and into 2010. This is likely to involve adapting Bartok, an existing type-preserving compiler for .NET MSIL to produce a dependently typed assembly language as output.

VI. Verified implementations of Cardspace protocols in Fine. Concurrent with the development of the Fine compiler, we plan to explore adapting prior results on the verification of the InfoCard protocol suite [9] in a manner that allows it to be integrated with the rest of our enforcement infrastructure. This work would require collaboration with members of the Samoa project in MSR Cambridge.

References

[1] M. Abadi. Access control in a core calculus of dependency. *SIGPLAN Not.*, 41(9):263–273, 2006.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th Conference on Computer and Communications Security*, pages 52–62. ACM Press, 1999.

[4] D. Aspinall and M. Hoffmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT Press, 2004.

[5] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *VMCAI*, pages 2–15, 2004.

[6] G. Barthe and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *European Symposium on Programming, Lecture Notes in Computer Science*. Springer, 2007.

[7] M. Becker, C. Fournet, and A. Gordon. Design and semantics of a decentralized authorization language. *Computer Security Foundations Symposium, IEEE*, 0:3–15, 2007.

[8] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *IEEE Symposium on Computer Security Foundations*, 2008.

[9] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *ACM Symposium on Information, Communication and Comunication Security (ASIACCS)*, pages 123–135, 2008.

[10] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 164, Washington, DC, USA, 1996. IEEE Computer Society.

[11] R. Boland. Network centricity requires more than circuits and wires. *SIGNAL*, Sept. 2006.

[12] R. H. Brown and A. Prabhakar. Standard security label for information transfer. Technical Report FIPS PUB 188, US Deparment of Commerce / National Institute of Standards and Technology, 1994.

[13] G. Bruns and M. Huth. Access-control policies via belnap logic: Effective and efficient composition and analysis. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 163–176, Washington, DC, USA, 2008. IEEE Computer Society.

[14] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 316–330, London, UK, 2001. Springer-Verlag.

[15] P. C. Chapin, C. Skalka, and X. S. Wang. Authorization in trust management: Features and foundations. *ACM Comput. Surv.*, 40(3):1–48, 2008.

[16] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 183–192, New York, NY, USA, 2008. ACM.

[17] E. Christensen, F. Curbera, G. Meredith, and S.Weerawarana. Web services description language (wsdl) 1.2, 2002.

[18] Computer World, June 2008. Top Secret: CIA explains its Wikipedia-like national security project.

[19] B. Corcoran, N. Swamy, and M. Hicks. Combining provenance and security policies in a web-based document management system. In *On-line Proceedings of the Workshop on Principles of Provenance (PrOPr)*, Nov. 2007.

[20] L. de Moura and N. Bjorner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

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

[22] J. DeTreville. Binder, a logic-based security language. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[23] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. IETF RFC 2693, 1999.

[24] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VI I, 1972.

[25] Y. Gurevich and I. Neeman. Dkal: Distributed-knowledge authorization language. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 149–162, Washington, DC, USA, 2008. IEEE Computer Society.

[26] L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, 2008.

[27] T. Jim. Sd3: A trust management system with certified evaluation. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 106, Washington, DC, USA, 2001. IEEE Computer Society.

[28] S. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The glasgow haskell compiler: a technical overview, 1993.

[29] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *IEEE Symposium on Security and Privacy*, 2002.

[30] Microsoft Sharepoint. Creating business value through better collaboration. Microsoft White Paper available at, 2005.

[31] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[32] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 229–240, New York, NY, USA, 2008. ACM.

[33] National Health Service. Spine. `http://www.connectingforhealth.nhs.uk/systemsandservices/spine`.

[34] I. Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, College Park, August 2008.

[35] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[36] J. Park and R. Sandhu. Originator control in usage control. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:0060, 2002.

[37] L. Petersen, P. Cheng, R. Harper, and C. Stone. Implementing the tilt internal language. Technical report, Carnegie Mellon University, 2000.

[38] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, Jan. 2003.

[39] SecurityFocus: Access control bypass vulnerabilities. `http://search.securityfocus.com/swsearch?metaname=alldoc&query=access+control+bypass`.

[40] N. Swamy. *Language-based Enforcement of User-defined Security Policies*. PhD thesis, University of Maryland, College Park, August 2008.

[41] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2008.

[42] N. Swamy and M. Hicks. Verified enforcement of stateful information release policies. In *Proceedings of the ACM SIGPLAN Workshop on Programming Langauges and Analysis for Security (PLAS)*, June 2008.

[43] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proc. Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006.

[44] D. Syme. The f# programming language. `http://msdn.microsoft.com/en-us/fsharp/default.aspx`, 2008.

[45] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til: a type-directed, optimizing compiler for ml. *SIGPLAN Not.*, 39(4):554–567, 2004.

[46] TSCP. The Trans-global Secure Collaboration Program (TSCP). `http://www.tscp.org/`, 2008.

[47] S. C.-T. Tse. *Dynamic security policies*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2007. Adviser-Stephan Zdancewic.

[48] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, Pittsburgh, PA, USA, June 2008.

[49] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[50] X. Wang, G. Lao, T. DeMartini, H. Reddy, M. Nguyen, and E. Valenzuela. Xrml – extensible rights markup language. In *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, pages 71–79, New York, NY, USA, 2002. ACM.

[51] XACML and OASIS Security Services Technical Committee. eXtensible Access Control Markup Language (XACML) Committee Specificaion 2.0.

[52] H. Xi and R. Harper. A dependently typed assembly language. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 169–180, New York, NY, USA, 2001. ACM.

[53] D. Yu. More typed assembly languaes for confidentiality. In *The Fifth ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 19–37, 2007.

[54] D. Yu and N. Islam. A typed assembly language for confidentiality. In *European Symposium on Programming*, pages 162–179, 2006.

[55] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST '04*. Springer, 2004.