Trusting the Network

(Extended Abstract)

Tom Chothia
Laboratoire d'Informatique (LIX)
École Polytechnique
91128 Palaiseau Cedex, France.

Dominic Duggan
Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ, USA.

Ye Wu
Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ, USA.

Abstract

Cryptography is often used to secure the secrecy and integrity of data, but its ubiquitious use (for example on every read and write of a program variable) is prohibitive. When protecting the secrecy and integrity of data, applications may choose to reply on the underlying runtime or network, or they may seek to secure the data themselves using cryptographic techniques. However specifying when to rely on the environment, and when to explicitly enforce security, is usually specified informally without recourse to explicit policies. This article considers an approach to making explicit when the runtime or network is trusted to carry data in cleartext, and when the data must be explicitly protected. The approach is based on associating levels of trust to parts of the system where data may reside in transit, and levels of relative sensitivity to data that is transmitted. The trust policy is enforced by a type system, one that distinguishes between security policies for access control and trust policies for controlling the safe distribution of data.

1 Introduction

This article addresses the question: When should one trust the network over which data is communicated?

What is a network? We take a very generic point of view: a network is any communication medium through which messages are exchanged between entities in a computing system. Examples include:

- hosts exchanging packets through the Internet;
- processes exchanging data across a virtual private network secured using IPsec;
- processes on the same machine communicating via IPC;
- threads in a single address space exchanging messages via a message queue.

We assume that the data being exchanged has both secrecy and integrity constraints associated with it. We say that a network is *trusted* if parties using that network to communicate rely on the network itself to realize the desired secrecy and integrity guarantees. For example in type-based information flow control systems, the "network" is represented by global shared variables, and a type system ensures that secrecy constraints are enforced for all participants. On the other hand, in distributed programming over the Internet, the network is not trusted; so cryptographic techniques are used to secure secrecy and integrity properties (encryption and signing, respectively). There is still some trust: cryptographic operations may not be performed until data is about to be output from the network card, since the operating system and its buffers are assumed to be trusted. Virtual private networks use cryptographic techniques to build a virtual trusted network that entities can use to communicate with assurance that the requisite properties are satisfied.

In most of these examples, the desired secrecy and integrity properties, and the level of trust in the network, are expressed relatively informally. There is then little hope of relating the two to make sure that there is enough trust in the network to enforce these properties, and if not to use cryptographic techniques to achieve those properties. In type-based information flow systems and distributed programming over the Internet, it appears clear where to place trust (everywhere and nowhere, respectively). But there are gradations between these extremes: typed threads may communicate through an operating system component that is outside the type system, or across an untrusted network; hosts may communicate across a trusted subnet behind a firewall.

In this article we provide formalizations of both secrecy and integrity properties, and of the trust placed in parts of the network, and we relate them in such a way that the properties are guaranteed to be achieved (provided, that is, one has not placed trust in an unreliable part of the network).

Various models have been proposed for specifying secrecy and integrity properties of information in the type system [15, 16, 8]. These systems associate sets of policies (called *labels*) with variables, along with their types. This article works with a much simpler model, where a label simply specifies a set of principals (actually four sets of principals, each for a different purpose). Nevertheless the system presented here could be applied to the aforesaid information flow control systems. Our system is simplified because we do consider notions such as declassification. The novelty of this work is in a model of trust for networks, and how that is related to secrecy and integrity constraints on data exchanged over a network.

Work has previously considered type-based APIs to cryptographic operations [1, 11, 8] that relate secrecy and integrity policies, as defined in program types, with the use of cryptographic operations to dynamically enforce those policies where necessary (when transmitting data over the Internet, for example). However absent from these models is when these operations *must* be performed. Requiring them to be always performed is in practice ridiculous: every write of a variable would require encryption and signing, and every read of a variable would require decryption and authentication. However sometimes these operations must be performed, e.g., when non-trivial policies must be enforced over a raw TCP/IP channel. Implicit in these extremes is where the trust is placed in the network.

We formalize this notion of trust using a notion of *zones*. A zone is an abstraction of any notion of network location, be it a process address space, a host, or a physical or virtual network. Entities specify that certain zones are trusted for certain communications. If data is transmitted across a zone that is trusted, then policies can be left to be enforced by the type system. For example, a general may send a message to soldiers in the field commending them or informing them of promotion; type-based techniques can guarantee that the general is the originator of the message. If data is transmitted across an untrusted zone, then the policies must be cryptographically enforced. For example, the general may place less trust in the network when sending a command to field commanders for a coordinated attack. On the other hand, if the general is in conference with field commanders within a secure conference room, it may be sufficient to rely on lightweight type-based security for such local communications.

We use the term *trust policy* to distinguish it from *security policy*. We use the latter to refer to access control policies to enforce secrecy and integrity constraints. We use the former to refer to levels of trust that are placed in parts of the network to respect such security policies. There should be no confusion with the term "trust management," although our approach could obviously be enriched with notions of delegation.

Trust and security policies are largely orthogonal. Security policies specify fine-grained read and write permissions for data. Trust policies specify the relative trustworthiness of parts of the network for enforcing those security policies, without application intervention using cryptography. Both forms of policies are specified using a type system. The type system ensures that security policies are respected by all well-typed processes. It also ensures that data with non-trivial security policies is not transmitted in cleartext over parts of the network that are not trusted to only contain well-typed processes.

1. A principal may be included in the trust secrecy policy for a piece of data but not the security secrecy policy. This simply means that processes for that principal may execute in zones where the principal is not allowed

to access the data; the principal is "trusted" to respect the security policy (as far as that piece of data is concerned).

2. A principal may be included in the security secrecy policy for a piece of data but not the trust secrecy policy. The principal is allowed to access the data, but cannot execute at any zones where the data is transmitted in cleartext. This is admittedly strange, but harmless, and demonstrates the orthogonality of trust and secrecy policies. There are extensions of our system, involving declassification, where this scenario may be useful.

We give an informal introduction to zones and trust policies in the next section. In Sect. 3 we provide a formalization of the type system. Finally Sect. 4 considers related work, while Sect. 5 provides conclusions.

2 Informal Motivation

The type system we introduce incorporates notions of *principals*, *labels* and *policies*. Every program variable has both a type and a label. A label $L = (\pi_1, \pi_2)$ is a pair of *policies*, one for secrecy and the other for integrity. A policy $\pi = (\{\overline{P_1}\}, \{\overline{P_2}\})$ is a pair of sets of principals:

1. The first set specifies the *security policy*, the set of principals that are allowed to access that variable (reading the variable in the case of a secrecy policy, and writing into that variable in the case of an integrity policy).

2. The second set specifies the *trust policy*, the set of principals that are expected to respect the security policy. The security policies are only enforced statically at sites that are accessible to principals included in the corresponding trust policies. At all other sites, cryptographic techniques must be used to enforce the security policies dynamically.

The security policy is enforced by the type system. At load time (e.g., via bytecode verification in a Java Virtual Machine), programs are checked by type-checking to ensure that the security policies are respected. We are assuming a model where code is signed and the loader checks code signatures before running them. Thus a process does not attempt to read a variable unless the principal for which it executes (the principal that signed its code base) is in the set of principals specified by the read security policy. Any program for which this check fails is rejected by the loader.

In a perfect world, all running processes are typed and security policies are always respected. Zones allow relative levels of trust to be placed on parts of the network. If data is transmitted through a zone that is not trusted, then cryptographic techniques must be used to protect it. Therefore, orthogonal to security policies, data also has trust policies associated with it.

Zones are an abstractions for network and network trust. A zone is abstractly a location that contains processes and messages. A process may only receive messages that are in its own zone. A process can send a message locally (within its own zone). A process may also send a message to another zone (routing). Routers are modelled just as normal user-space programs that accept messages and forward them to other zones. For simplicity we assume a fully-connected graph of network connections, though it would be trivial to enforce more restricted connectedness.

A zone has a *level*, reflecting the level of trust that is placed in it. A level is denoted simply as a set of principals. The only processes that can send and receive messages in a zone are those executing for the principals in the corresponding zone level.

Zone levels effectively enforce which principals are allowed access to a zone, for sending or for receiving. An insider attack consists of running "untyped" code in a zone. For an outside attacker to access a zone, it must compromise a principal allowed in that zone and then mount an insider attack. Network trust policies can then be defined based on (a) which principals can execute processes in a zone and (b) how susceptible those principals' processes are to insider or outsider attacks. There is implicit in this model some notion of authenticating the right of code to execute in a zone. We simplify our model by assuming that the only code that runs in a zone is

that which is allowed by the zone's level. The actual details of authentication can be assumed to be part of the trusted computing base (TCB). An interesting direction for further work is to see how this authentication could be reflected from application programs into the TCB.

2.1 Examples

Fig. 1 gives an example of processes executing for principals, and implicitly the zones for those principals. We assume a zone level CJIS (Criminal Justice Information Division) partitioned into two sublevels: NCIC (National Crime Information Center) and NICS (National Instant Criminal Background Check System). NCIC allows processes for two principals, A and B, to execute there, while NICS allows processes for principal C. Another zone level, PAPD (Port Authority Police Department), allows processes for principals X and Y to execute there. Processes for principals B and X are required to be able to share data, therefore we establish an intermediate zone level CPD (Central Police Desk) for just those two principals. CPD describes all sites on which only processes for those principals can execute. This could be implemented as a trusted path, a physical firewall, or a distributed firewall or VPN established using cryptographic techniques. The framework introduced here does not depend on the details of how access to zones is established. Processes for B may execute in the same zones as processes for A and C (at levels NCIC and NICS). Similarly processes for X may execute in the same zone as processes for Y (at level PAPD). Messages exchanged between processes for B and processes for X may be passed in cleartext in zones of level CPD.

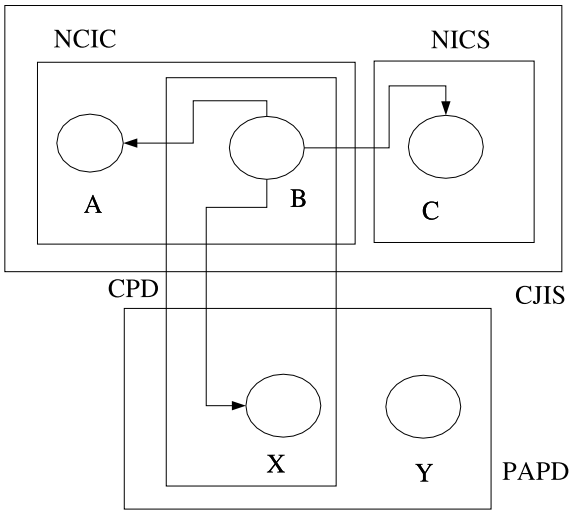

Figure 1: Principals and Zone Levels

Now whether messages exchanged between B and X may be passed in cleartext in zones of level NCIC and PAPD, for example, depends on the relative sensitivity of the data and the relative level of trust placed in such zone levels. We can label data with the level of its relative sensitivity, and then compare this with the level of a zone through which the data is transmitted.

We assume that policies are specified as sets of principals. This is clearly a simplifying assumption that could be removed by introducing some notion of levels or roles in the security system, and relying on authorization certificates to allow people to assume particular roles. Then policies could be specified in terms of such roles or levels. However for simplicity in the article we stay with sets of principals. For the example above we can define the following:

```
prinset NCIC = {A,B};
prinset NICS = {C};
prinset CJIS = NCIC ∪ NICS;
prinset PAPD = {X,Y};
prinset CPD = CJIS ∩ PAPD;
prinset World;
```

We can treat `World` as a special marker for the set of all principals. Now using these definitions of sets of principals, we can specify the following policies:

```
policy TopSecret = ( CPD, CPD );
policy FBI_Secret = ( CPD, NCIC ∪ {X} );
policy Internal = ( NCIC ∪ PAPD, CJIS ∪ PAPD );
policy Public = ( World, World );
```

For example FBI_Secret defines information that is only intended for CPD personnel to see, but we trust members of the FBI to carry this information without violating confidentiality.

We focus in this example on secrecy. We define the following labels where we elide the integrity policies:

```
label TopSecretL = ( TopSecret, ...  );
label FBI_SecretL = ( FBI_Secret, ...  );
label InternalL = ( Internal, ...  );
```

A packet where the payload has label TopSecretL and the address has label InternalL expresses that the payload is top secret, but we allow the address to be visible within the organization, so it may be viewed for example by couriers that carry information back and forth. This example requires the payload to be encrypted outside CPD. A more lenient policy retains the notion that the payload is top secret, but enforces this based on the trustworthiness of the principals in the environment rather than relying on encryption. If we label the payload as FBI_SecretL and the address as InternalL, then we allow top secret information to be transmitted by processes for FBI (CJIS) personnel who may not have authority to read the information that they are conveying, however we do not trust unauthorized PAPD processes to respect the access restrictions.

In these examples the message-passing communication system checks the access control specifications on the data being transmitted with the mapping from zones names to sets of principals, to detect violations of the policy specified in the types. Since the former mapping will change after programs have been compiled, some amount of dynamic checking is unavoidable[1]. Nevertheless in the remainder of the article we rely on static checking using a type system. This at least formulates the properties that any kind of type checking is seeking to achieve, and leaves open the possibility of some hybrid regimen of static and dynamic policy checking.

To model attacks by adversaries who do not respect the restrictions of the type system, we assume a special principal Eve. There are many possibilities for the kinds of attacks we can allow this adversary to mount. At one extreme processes executing for Eve may be completely untyped. However it becomes difficult (not impossible, but somewhat complex) to say useful things about the ability of the system to withstand such attacks. In any case, untyped attacks based on bypassing the type system may often be caught in this message-passing context during the unmarshalling stage of communication. We assume a weaker attack model therefore: Eve respects the type system but can ignore the labels on the types of data and she can ignore the access control checks, both for secrecy and integrity. For example, Eve may mount an attack on Alice by sending her a channel with the payload label

$$((\{\mathsf{Alice},\mathsf{Bob}\},\ldots),(\{\mathsf{Bob}\},\ldots))$$

The security parts of the policies assure Alice that this is a private channel between her and Bob, created by Bob. We consider in the next section how Alice can defend herself against this attack using trust policies. For now we note that the type system ensures that if Eve does obtain access to a piece of data, then Eve must have been one of the principals allowed in the original trust policy. In other words, if an attacker obtains access to a data by subverting the type system, the original owner of the data must have mistakenly placed its trust in an unreliable principal.

3 Type System: Formal Description

The syntax of types is provided in Fig. 2. In this language without cryptographic operations, there are two kinds of data: message channels (for message-passing communication) and tuples (for data structures). Data have both types and *labels*, where a label is a pair of *policies*: the *secrecy policy* and the *integrity policy*. Each of these policies is in turn separated into security and trust components, which in this system are simply sets of principals.

[1]This dynamic checking would consist of checking that the trust restrictions on data being routed to a different zone are satisfied by the set of principals that are allowed to execute at that zone.

$T \in \text{Type}$	::=	$\mathsf{Chan}(LT)$	Channel
	$\mid$	$\langle LT_1, \ldots, LT_k \rangle$	Tuple
$\pi \in \text{Policy}$	::=	$(\{\overline{P_1}\}, \{\overline{P_2}\}$	
$L \in \text{Label}$	::=	(π_1, π_2)	
$LT \in \text{Labelled type}$	::=	T^L	

Figure 2: Syntax of Sensitivity Types

$v \in \text{Value}$	::=	w, x, y, z	Variable
	$\mid$	a, b, c, n	Channel name
	$\mid$	$\langle V_1, \ldots, V_k \rangle$	Tuple
	$\mid$	V^L	Annotated value
$R \in \text{Process}$	::=	stop	Stopped process
	$\mid$	$\mathsf{let}_L\ x = \langle v_1, \ldots, v_k \rangle; R$	Create tuple
	$\mid$	$\mathsf{let}\ \langle x_1, \ldots, x_k \rangle = v; R$	Decompose tuple
	$\mid$	$\mathsf{let}_L\ x = v; R$	Annotate value
	$\mid$	$\mathsf{receive}\ c?x;\ R$	Message receive
	$\mid$	$\mathsf{send}\ v!v$	Message send
	$\mid$	$\mathsf{send}\ \ell.v!v$	Message route
	$\mid$	$\mathsf{new}(a : LT); R$	New channel
	$\mid$	$(R_1 \mid R_2)$	Parallel composition
$N \in \text{Network}$	::=	empty	Empty network
	$\mid$	$(P, \ell)[R]$	Located process
	$\mid$	$\mathsf{new}(a : LT)N$	Channel binding
	$\mid$	$(N_1 \mid N_2)$	Wire

Figure 3: Syntax of Values, Processes and Networks

For a trust policy, the set of principals in the secrecy component denotes those principals who are *trusted to honor secrecy restrictions for the data*, while the set of principals in the integrity component denotes those who are *trusted to honor integrity restrictions for the data*. We highlight this explanation because *trust policies do not themselves specify secrecy and integrity restrictions*. These restrictions should be expressed using the security policy. It is straightforward to extend our simple security policies with the other kinds of policy languages.

The syntax of values, processes and networks is given in Fig. 3. Values include variables, channel names and tuples. Processes include parallel composition (for forking new processes), new channel creation, operations for building and taking apart tuples, and basic message-passing operations (blocking receive and non-blocking send). This is a two-level syntax: every process executes under the authority of a principal and at a particular zone (network location). So the network is the parallel composition of a collection of located processes, each of the

$VE \vdash v : T^L$	Well-formed value
$VE \vdash (P, \ell)[R]$	Well-formed process
$VE \vdash N$	Well-formed network

Figure 4: Judgements for Type System

form $(P, \ell)[R]$ where P is the principal, ℓ the zone and R the process. The purpose of this two-level syntax is twofold:

- to enforce security policies based on the authority of the principal under which a process executes; and
- to support a *routing operation* that allows a process at one zone to route a message to another zone.

We provide a type system using judgements of the form given in Fig. 4. The type system for processes uses judgements of the form $VE \vdash (P, \ell)[R]$ to check that the process R is well-formed, under the assumption that it will be evaluated (executed) under the authority of the principal P, at the zone ℓ, with free names bound in the environment VE. An environment is a sequence of pairs, binding variables or names to types:

$$VE \in \text{Value Env} \quad ::= \quad \{\} \mid \{(x : LT)\} \mid \{(a : LT)\} \mid VE_1 \cup VE_2$$

The types of values, processes and networks are provided in the full version of the paper [9].

As noted, we denote attacker processes as those executing for the special principal Eve. An example was provided in Sect. 2. Eve is still subject to checks on where in the network her processes can execute. Alice can prevent the forgery attack described in Sect. 2 by only accepting a private channel sent in a zone that does not allow Eve to execute processes (in the integrity part of the trust policy). But then Alice and Bob can only communicate in the same trusted space. The more interesting cases then are where cryptography is used to secure communication over untrusted spaces, and where Eve may still attempt to mount attacks based on interception and forgery of cryptographic keys.

In the extension of the system with cryptographic operations, there are now types for encrypted and signed data, $\mathcal{E}\{T\}$ and $\mathcal{S}\{T\}$ respectively. There are also types for public and private keys, for encryption and signing. Each of these key types is indexed by a policy. This reflects the intuition that a key fundamentally is used to enforce a policy across address spaces. If decryption of ciphertext succeeds, then the secrecy policy associated with the key type is re-established for the resulting cleartext. Similarly if authentication of signed ciphertext succeeds, then the integrity policy associated with the key type is re-established for the resulting cleartext.

Theorem 1 (Type Preservation) *Suppose* $VE \vdash N_1$ *and* $N_1 \xrightarrow{TE;VE} N_2$, *then* $VE \vdash N_2$.

A network N is "stuck" if no evaluation rule is applicable to it.

Theorem 2 (Progress) *If a network N is stuck, then the remaining processes are of the form:*

1. *A receive operation with no matching send message to synchronize with.*
2. *A decrypt operation where the decryption key is not the inverse of the encryption key for the ciphertext.*
3. *An authentication operation where the authentication key is not the inverse of the signing key for the ciphertext.*

In particular this justifies omitting run-time access checks in the semantics. Note that the checks for processes' permissions to execute in zones is determined statically in the type system, as mentioned earlier. This static restriction could be relaxed with the addition of zone variables to the language.

We still need to say something about the extent to which processes of Eve may subvert the security of the system. We focus on names (channel names and cryptographic keys) as the values whose secrecy is paramount. Say that a name is *leaked* if there is a free occurence as a subterm of a process for Eve, where that occurence does not occur as part of a piece of ciphertext.

Theorem 3 *Given VE and N, with* $(c : T^L) \in VE$ *and c is leaked in N. Then* $\mathsf{Eve} \in SLEV(L)$.

Intuitively the secrecy trust policy for the data reflects all possible principals in all possible zones where the data may be transmitted in cleartext. If the data is encrypted, then the type rule for encryption keys requires that the decryption key have at least as strict a secrecy trust level. If the data or key is transmitted directly from a process of Eve in the current zone, even with annotation, Eve must be included in the set of principals that can execute in the zone where the key was created. If the data or key is transmitted via an intermediary (trusted) process, then that process or some ancestor in a chain of intermediaries exchanged the data or key in a zone with Eve; and since none of the intermediaries have access to the annotation operation, the trust secrecy policy for the data or key must include Eve as a principal.

4 Related Work

The motivation for this work has been the need for proper programming abstractions for applications that must manage the task of securing their own communication. Much of the work on abstractions for Internet programming has focused on security, for example, providing abstractions of secure channels [4, 3], controlling key distribution [7], reasoning about security protocols [1, 5], tracking untrustworthy hosts in the system [14, 19], etc.

The work of Riely and Hennessy [14, 19] in particular has some relationship to this work. They provide a type system that reflects the relative level of trust in hosts in the network. They are motivated by ensuring that mobile agents do not migrate to untrusted hosts. An "untrusted" host in our system amounts to a zone that includes a process executing for the attacker Eve. This extra level of indirection is more than cosmetic, since it reflects our concern with enforcing access control policies through a combination of static and dynamic techniques, with trust policies used to determine when dynamic techniques (cryptography) must be used.

Abadi [1] considers a type system for ensuring that secrecy is preserved in security protocols. For securing communication over untrusted networks, he includes a "universal" type `Un` inhabited by encrypted values. His type system prevents "secrets" from being leaked to untrusted parties, but allows encrypted values to be divulged. In an analogous way, encrypted values in our type system provide a way to temporarily subvert the access controls in the type system, with the secrecy properties enforced by labels reasserted when the ciphertext is decrypted/authenticated. Gordon and Jeffrey [12, 13] have developed a type-based approach to verifying authentication protocols.

Abadi and Blanchet [2, 6] have worked on analyzing security protocols, showing how it is possible to guarantee secrecy properties and then generalizing this to guarantee integrity. Their system uses a type of "secret," and a type system that ensures that secret items are never put on channels that are shared with untrusted parties. They can translate types in their system into logic programs that can then be used to check protocols for correctness. The emphasis of this work is somewhat different, since Bruno and Blancet work in a more "black and white" environment where there are trusted parties and untrusted parties. In contrast our interest is in a more refined type system where we allow certain parties to access certain data, and where different levels of trust are placed in different parts of the network.

Other work on security in programming languages has focused on ensuring safety properties of untrusted code [17] and preventing unwanted security flows in programs [10, 15, 21, 18]. Sabelfeld and Myers [20] provide an excellent overview of work in language-based information-flow security. Our security concerns have largely been with access control, but the work can be extended with ideas from the decentralized label model of JIF [16, 8].

Our work is clearly related to that of J/Split [9, 22]. That system partitions a sequential JIF program into a distributed system, where portions of the program run on hosts that are trusted for the principals for whom the code runs. There is then a tight relationship between the access restrictions on data (specified using the richer form of access restrictions allowed by JIF) and the hosts where data may be stored in cleartext. For two mutually distrustful principals engaged in a distributed game, for example, a trusted third party (the board) is responsible for communicating data from one party to the other. Network security (cryptographic operations) is implicitly part of the TCB.

Our model can be viewed as attempting to expose some of the TCB machinery of this approach to the application, with the eventual goal of modeling the J/Split runtime as application programs in our approach. Thus although hosts can be viewed as analogous to zones (both are simply abstractions of network locations, hardly a new concept), we decouple the security and trust policies because each means different things.

5 Conclusions

The ultimate goal of this research program is to provide a programming environment where secrecy and integrity requirements are specified explicitly in the type system, where these requirements are related to the relative trustworthiness of parts of the network. Security policy specifies what must be protected; trust policy specifies how it must be protected. Finally a type-based API to cryptographic operations relates the use of these operations to the requirements that they are intended to satisfy.

The antithesis of such an environment is one where all security requirements are enforced inside the runtime. This leads to a bloated trusted computing base (TCB) and flies in the face of the well-known end-to-end argument in system design. An interesting possible avenue for the application of our approach is in Web services authentication, where end-to-end security considerations predominate [5].

We have deliberately chosen a very simple language for presenting our approach. There are numerous avenues to pursue with this work. Clearly it can be combined with the full extent of KDLM [8], which provides a somewhat richer policy language, borrowing ideas from JIF [16] and adding declassification certificates. A more interesting direction to consider is allowing zone variables, so that processes can build routing tables and perform dynamic routing decisions, while continuing to perform static checking as much as possible. Zone types based on zone levels, perhaps building on the work of Riely and Hennessy [14, 19], appear to be a promising direction in this regard.

References

[1] Martin Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Science*, pages 611–638, 1997.

[2] Martin Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 33–44, 2002.

[3] Martin Abadi, Cedric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, 1999.

[4] Martin Abadi, Cedric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2000.

[5] Karthikeyan Bhargavan, Cedric Fournet, and Andrew D. Gordon. A semantics for web services authentication. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2004.

[6] Bruno Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, pages 242–259, 2002.

[7] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *Concurrency Theory (CONCUR)*. Springer-Verlag, 2000.

[8] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *Computer Security Foundations Workshop*, 2003.

[9] Tom Chothia, Dominic Duggan, and Ye Wu. Trusting the network. `http://guinness.cs.stevens.edu/~dduggan/Public/Papers/zones.pdf`, 2005.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 1977.

[11] Dominic Duggan. Cryptographic types. In *Computer Security Foundations Workshop*, Nova Scotia, Canada, 2002. IEEE Press.

[12] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2001.

[13] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[14] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[15] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, 1998.

[16] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.

[17] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, 1996.

[18] Francois Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of ACM International Conference on Functional Programming*, 2000.

[19] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

[20] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2002.

[21] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Springer-Verlag, 1997.

[22] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *Transactions on Computer Systems*, 20(3):283–328, 2002.