Policies and Proofs for Code Auditing

Nathan Whitehead[1], Jordan Johnson[1], and Martín Abadi[1,2]

[1] University of California, Santa Cruz
[2] Microsoft Research

Abstract. Both proofs and trust relations play a role in security decisions, in particular in determining whether to execute a piece of code. We have developed a language, called BCIC, for policies that combine proofs and trusted assertions about code. In this paper, using BCIC, we suggest an approach to code auditing that bases auditing decisions on logical policies and tools.

1 Introduction

Deciding to execute a piece of software can have substantial security implications. Accordingly, a variety of criteria and techniques have been proposed and deployed for making such decisions. These include the use of digital signatures (as in ActiveX [12]) and of code analysis (as in typed low-level languages [5, 9, 10]). The digital signatures can be the basis of practical policies that reflect trust relations—for instance, the trust in certain software authors or distributors. The code analysis can lead to proofs, and thereby to proof-carrying code [11]. Unfortunately, neither trust relations nor proofs are typically sufficient on their own. Trust can be wrong, and code analysis is seldom comprehensive.

We are developing a system for defining and evaluating policies that combine proofs and trusted assertions about code [18–20]. The core of the system is a logical query language, called BCIC. BCIC is a combination of Binder [4], a logic-programming language for security policies in distributed systems, with Coq's Calculus of Inductive Constructions (CIC) [3], a general-purpose proof framework.

Whereas the focus of most previous work (including our own) is on the decision to execute pieces of code, similar considerations arise in other situations. For instance, from a security perspective, installing a piece of code can be much like executing it. Further upstream, auditing code is also critical to security. Auditing can complement other techniques for assurance, in the course of software production or at various times before execution. Although humans perform the auditing, they are often guided by policies (e.g., what aspects of the code should be audited) and sometime supported by tools (e.g., for focusing attention on questionable parts of the code).

In this paper, using BCIC, we suggest an approach to code auditing that bases auditing decisions on logical policies and tools. Specifically, we suggest that policies for auditing may be expressed in BCIC and evaluated by logical means. Thus, this approach leverages trust relations and proofs, but it also allows

auditing to complement them. We recognize that this approach is still theoretical and probably incomplete. Nevertheless, it emphasizes the possibility of looking at techniques for verification and analysis in the context of policy-driven systems, and the attractiveness of doing so in a logical setting.

We present two small examples. The first example concerns operating system calls from an application extension language. With a BCIC policy, every operating system call must be authorized by an audit. A policy rule can allow entire classes of calls without separate digital signatures from an authority. In the second example, we consider an information-flow type system [14], specifically a type system that tracks trust (much like Perl's taint mode [6], but statically) due to Ørbæk and Palsberg [13]. The type system includes a form of declassification, in which expressions can be coerced to be trusted (that is, untainted). If any program could use declassification indiscriminately, then the type system would provide no benefit. With a BCIC policy, a trusted authority must authorize each declassification. In both examples, security decisions can rely on nuanced, fine-grained combinations of reason and authority.

We treat these examples in Sections 2 and 3, respectively. We consider implementation details in Section 4. We conclude in Section 5.

2 Example: Auditing Function Calls

In this example we consider the calling behavior of programs in a managed environment of libraries. A base application may allow extensions that provide additional functionality not only to the user but also to other extensions. The extensions may come from many different sources, and accordingly they may be trusted to varying extents. By constraining calls, the security policy can selectively allow different functionality to different extensions.

2.1 Language

For simplicity, we study an interpreted extension language. Specifically, we use an untyped λ-calculus with a special `call` construct that represents operating system calls and calls to other libraries. All calls take exactly one argument, which they may ignore. In order to allow primitive data types, we also include a representation for data constructors (`constr0`, `constr1`, and `constr2`, for constructors that take zero, one, and two arguments respectively). These constructors are enough to handle all the data types that appear in our implementation, including natural numbers, pairs, and lists. Destructors have no special syntax, but are included among the calls.

In Coq notation [2, 3, 16], the syntax of the language is:

```
Inductive exp : Set :=
| var : nat -> exp
| abs : exp -> exp
| app : exp -> exp -> exp
```

```
| call : funcname -> exp -> exp
| constr0 : constrname -> exp
| constr1 : constrname -> exp -> exp
| constr2 : constrname -> exp -> exp -> exp.
```

A detailed knowledge of Coq is not required for understanding this and other definitions in this paper. This definition introduces a class of expressions, inductively by cases with a type for each case; expressions rely on De Bruijn notation, so variables are numbered and binding occurrences of variables are unnecessary. Similarly, other definitions introduce other classes of expressions and propositions, and some parameters for them.

A policy can decide which calls any piece of code may execute. The policy can be expressed in terms of a parameter `audit_maycall`.

```
Parameter audit_maycall : exp -> funcname -> Prop.
```

According to this type, every audit requirement mentions the entire program `exp` that is the context of the audit. Mentioning a subexpression in isolation would not always be satisfactory, and it may be dangerous, as the effects of a subexpression depend on context. The audit requirement also mentions the name of the function being called. We omit any restrictions on the arguments to the function, in order to make static reasoning easier; we assume that the callee does its own checking of arguments. (Section 3 says more on going further with static analysis.)

The predicate `audited_calls` indicates that a piece of code has permission to make all the calls that it could make. This predicate is defined inductively by:

```
Inductive audited_calls : exp -> exp -> Prop :=
| audited_calls_var :
  forall e n,
    audited_calls e (var n)
| audited_calls_app :
  forall e e1 e2,
    audited_calls e e1 ->
    audited_calls e e2 ->
    audited_calls e (app e1 e2)
| audited_calls_abs :
  forall e e1,
    audited_calls e e1 ->
    audited_calls e (abs e1)
| audited_calls_call :
  forall f e e1,
    audited_calls e e1 ->
    audit_maycall e f ->
    audited_calls e (call f e1)
```

```
| audited_calls_constr0 :
  forall e cn,
    audited_calls e (constr0 cn)
| audited_calls_constr1 :
  forall e cn e1,
    audited_calls e e1 ->
    audited_calls e (constr1 cn e1)
| audited_calls_constr2 :
  forall e cn e1 e2,
    audited_calls e e1 ->
    audited_calls e e2 ->
    audited_calls e (constr2 cn e1 e2).
```

Crucially, the case of `audited_calls_call` includes an `audit_maycall` requirement. Every `call` statement requires an audit. A code producer may construct a proof that, given the right audits, the code satisfies the predicate `audited_calls`.

2.2 Policy

Audit statements do not have proofs. They are provided by authorities and trusted to be true through BCIC's policies. The policies employ the special predicate `sat` in order to connect Coq statements and assumptions with Binder rules. Basically, `sat(F)` will be derivable in BCIC when Coq formula `F` has a proof that has been imported into the local context.

The policy writer can express trust in statements that have no proof using a new `believe` predicate, and rules such as:

```
believe(F) :- A says believe(F), trusted(A).
```

This rule expresses that, when a trusted authority says to believe something, the entity that evaluates the policy (in other words, the reference monitor) believes it. In this rule, as in Binder, we use logic-programming constructs plus the special construct `says` [1, 8] for representing the statements of principals.

In order to allow reasoning on beliefs, some additional rules are useful:

```
believe(F) :- sat(F).
believe(Q) :- believe(P), believe(<~P -> ~Q>).
```

Here, the notation `<term>` includes a CIC term within the policy. The notation `~P` within an included CIC term indicates a free variable that is bound in the policy rule. With these rules, all formulas that have proofs are believed, and belief is closed under modus ponens. (A generalization of the second rule to dependent types, of which implication is a special case, might be attractive but is not needed for our examples.)

Using these constructs and rules, a complete policy of a code consumer may say that a program is allowed to run if it has been properly audited. The policy

can specify which principals are trusted for the auditing. In addition, the policy may collect calls into groups, thus authorizing sets of calls with a single statement from a trusted authority. The complete policy may therefore look as follows:

```
trusted(alice).
trusted(B) :- A says trusted(B), trusted(A).

believe(F) :- sat(F).
believe(Q) :- believe(P), believe(<~P -> ~Q>).
believe(F) :- A says believe(F), trusted(A).

believe(<audit_maycall ~P ~F>) :-
  A says believe(<audit_maycall ~P ~F>), trusted(A).

classify(setuid, dangerous).
classify(setgid, dangerous).
classify(sbrk, userlowlevel).
classify(printf, io).
classify(sprintf, dangerous).

believe(<audit_maycall ~P ~F>) :-
  classify(F, C), A says allowgroup(P, C), trusted(A).

mayrun(P) :- believe(<audited_calls ~P ~P>).
```

Many variants are of course possible. In particular, a policy may let the auditing requirements depend on CIC proofs about intrinsic properties of the code, with clauses of the form `believe(<audit_maycall ...>) :- sat(...)`. Furthermore, a policy may concern not only the decision to run a program but also any requirements for checks during execution.

2.3 Correctness

This example is simple enough that not too much theory is needed, but there are some subtleties (in part because functions can be higher-order, and recursion is possible). The main theorem about our analysis says that, if a piece of code passes the analysis and is executed, every actual call will have been audited (but it does not say anything about the appropriateness of particular BCIC policies such as that of Section 2.2). Proving this theorem requires defining the operational semantics of the language in such a way that a history of calls is kept. We define a state as a pair that consists of a list of calls and an expression, then define the single-step reduction relation $S \to S'$ in the usual way. The only nonstandard rule is for calls: $(l, \texttt{call}\ f\, v) \to (f :: l, O(f, v))$, where we let the expression $O(f, v)$ represent the return value of calling function f with value v. We assume that the returned values do not contain calls themselves. We obtain:

Theorem 1. *For all programs* `e`*, if* `audited_calls e e` *is true and* $([], \texttt{e}) \rightarrow^* (l, \texttt{e}')$*, then* `audit_maycall e f` *is true for all function names* `f` *in* l*.*

The proof relies on a lemma that says if a function name f appears in l then f appears in the program e. The proof of the lemma is by induction on reductions, then by case analysis of all the possible reductions. The substitutions that arise from β reductions constitute the only difficulty. The proof is in Coq.

3 Example: Trust in the λ-Calculus

Ørbæk and Palsberg define a simple type system for a λ-calculus with trust annotations [13]. In this system, one may for instance describe applets for a web server; the type system can help protect the applets from malicious inputs and the web server from malicious applets. The language includes a construct `trust` for untainting, thus supporting a form of declassification. (A dual form of declassification is turning secret data into public data.) In this example we study how to impose auditing requirements on declassifications. We started to consider this example in our work on BLF, a preliminary version of BCIC [20, Appendix B]; here we develop it further in BCIC.

3.1 Language

The type system allows a small set of built-in types and function types; in addition, types include the annotations `tr` or `dis`, which indicate trust and distrust, respectively. Data from unknown outside parties, annotated with `dis`, can be treated with the proper suspicion. For instance, $\texttt{int}^{dis}$ is the type of distrusted integers, while $(\texttt{int}^{dis} \rightarrow \texttt{int}^{tr})^{tr}$ is the type of trusted functions that take distrusted integer inputs and return trusted integer results.

Figure 1 defines the types, annotations, and relations between types. Types are defined as `bare_type`, which become `annotated_type` when annotated by `tr` or `dis`. Subtyping is defined straightforwardly for both bare and annotated types. Auxiliary functions `trust_lte` and `join` represent the partial order of the trust annotations and give the greatest lower bound of two trust levels, respectively.

Expressions in the language are similar to those of $\lambda_{<:}$, the simply typed lambda calculus with subtyping, with the addition of `trust`, `distrust`, and `check` expressions. Each `trust` is tagged with an identifier so it can be referenced by an auditor. We made typecasts explicit in both origin and destination types for simplicity in the typechecker. We also included `int` and `bool` as built-in types. Figure 2 shows the definition of expressions and typechecking. As usual, typechecking is done with respect to typing assumptions in a context.

3.2 Policy

By formalizing the type system in Coq, we can write BCIC security policies that rely on type safety according to this type system. A simple example is the

```
Inductive trust_type : Set :=
| tr : trust_type
| dis : trust_type.

Inductive bare_type : Set :=
| usertype : nat -> bare_type
| arrow : annotated_type -> annotated_type -> bare_type

with annotated_type : Set :=
| annote : bare_type -> trust_type -> annotated_type.

Inductive trust_lte : trust_type -> trust_type -> Prop :=
| trust_lte_refl : forall (T : trust_type), trust_lte T T
| trust_lte_trdis : trust_lte tr dis.

Inductive bare_lte : bare_type -> bare_type -> Prop :=
| bare_lte_refl :
  forall (T : bare_type), bare_lte T T
| bare_lte_arrow :
  forall (X Y A B : annotated_type),
    ann_lte Y B -> ann_lte A X ->
    bare_lte (arrow X Y) (arrow A B)

with ann_lte : annotated_type -> annotated_type -> Prop :=
| ann_lte_refl :
  forall (T : annotated_type), ann_lte T T
| ann_lte_annote :
  forall (A B : bare_type)(U V : trust_type),
    bare_lte A B -> trust_lte U V ->
    ann_lte (annote A U) (annote B V).

Definition join (U V : trust_type) : trust_type :=
  match U with
    | tr => V
    | dis => dis
  end.
```

Fig. 1. Types and relations between types.

```
Definition trustid : Set := nat.

Inductive expr : Set :=
| const_int : nat -> expr
| const_bool : bool -> expr
| var : nat -> expr
| abs : annotated_type -> expr -> expr
| app : expr -> expr -> expr
| trust : trustid -> expr -> expr
| distrust : expr -> expr
| check : expr -> expr
| cast : expr -> annotated_type -> annotated_type -> expr.

Inductive context : Set :=
| nilctx : context
| cons : annotated_type -> context -> context.

Inductive has_type : context -> expr -> annotated_type -> Prop :=
  | type_int : forall n C,
      has_type C (const_int n) (annote (usertype 0) tr)
  | type_bool : forall p C,
      has_type C (const_bool p) (annote (usertype 1) tr)
  | type_varz : forall T C,
      has_type (cons T C) (var 0) T
  | type_varn : forall n T T2 C,
      has_type C (var n) T ->
      has_type (cons T2 C) (var (S n)) T
  | type_abs : forall C E T T2,
      has_type (cons T C) E T2 ->
      has_type C (abs T E) (annote (arrow T T2) tr)
  | type_app : forall C E1 E2 T1 T2 U V,
      has_type C E1 (annote (arrow T1 (annote T2 U)) V) ->
      has_type C E2 T1 ->
      has_type C (app E1 E2) (annote T2 (join U V))
  | type_trust : forall C E T U id,
      has_type C E (annote T U) ->
      has_type C (trust id E) (annote T tr)
  | type_distrust : forall C E T U,
      has_type C E (annote T U) ->
      has_type C (distrust E) (annote T dis)
  | type_check : forall C E T,
      has_type C E (annote T tr) ->
      has_type C (check E) (annote T tr)
  | type_cast : forall C E T T2,
      has_type C E T ->
      ann_lte T T2 ->
      has_type C (cast E T T2) T2.
```

Fig. 2. Expressions in the language and typechecking rules.

following policy:

```
mayrun(C, P) :- sat(<has_type ~C ~P (annote (usertype 0) tr)>)
```

This policy says that program P may run in context C if it typechecks and has the trusted type $\texttt{int}^{tr}$. This policy does not exclude the possibility that this type is trivially obtained by an application of trust, however.

The next step is to require that all applications of the trust operator be audited. Since the trust operator is basically a way to escape the type system, the value of the type system for security would be questionable at best if any program could use trust indiscriminately, hence the desire for auditing.

Figure 3 defines a predicate trusts_audited. This predicate recursively traverses an expression while remembering the entire expression. In this definition, occurrences of trust impose extra requirements. In particular, every occurrence of trust must have a corresponding audit statement. The audit predicate is defined to be satisfied externally, in our case by assertions in the policy rather than by proofs.

```
Parameter audit : expr -> trustid -> Prop.
```

An expression E has all its trusts audited when trusts_audited E E holds.

As a small example, consider the context C that has two elements, a distrusted function f of type $(\texttt{int}^{tr} \rightarrow \texttt{int}^{tr})^{dis}$ and a trusted integer x. The expression $(\texttt{trust}^{0}\ \texttt{f})\ \texttt{x}$ (which is actually app (trust 0 (var 0)) (var 1) in our syntax but which we write as $(\texttt{trust}^{0}\ \texttt{f})\ \texttt{x}$ for simplicity) will pass trusts_audited when there is an audit statement $\texttt{audit}\ ((\texttt{trust}^{0}\ \texttt{f})\ \texttt{x})\ 0$. Suppose that the code producer bob provides the code $(\texttt{trust}^{0}\ \texttt{f})\ \texttt{x}$ and a proof of:

```
audit ((trust^0 f) x) 0
  ->
trusts_audited ((trust^0 f) x) ((trust^0 f) x)
```

A trusted authority alice signs the statement:

```
believe(<audit ((trust^0 f) x) 0>)
```

The policy of a code consumer may be:

```
trusted(alice).

believe(F) :- sat(F).
believe(Q) :- believe(P), believe(<~P -> ~Q>).
believe(F) :- A says believe(F), trusted(A).

mayrun(C, P) :-
  believe(<has_type ~C ~P (annote (usertype 0) tr)>),
  believe(<trusts_audited ~P ~P>).
```

```
Inductive trusts_audited : expr -> expr -> Prop :=
  | audited_int :
    forall P n,
      trusts_audited P (const_int n)
  | audited_bool :
    forall P b,
      trusts_audited P (const_bool b)
  | audited_var :
    forall P n,
      trusts_audited P (var n)
  | audited_abs :
    forall P T E,
      trusts_audited P E ->
      trusts_audited P (abs T E)
  | audited_app :
    forall P E1 E2,
      trusts_audited P E1 ->
      trusts_audited P E2 ->
      trusts_audited P (app E1 E2)
  | audited_trust :
    forall P E id,
      audit P id ->
      trusts_audited P E ->
      trusts_audited P (trust id E)
  | audited_distrust :
    forall P E,
      trusts_audited P E ->
      trusts_audited P (distrust E)
  | audited_check :
    forall P E,
      trusts_audited P E ->
      trusts_audited P (check E)
  | audited_cast :
    forall P E T1 T2,
      trusts_audited P E ->
      trusts_audited P (cast E T1 T2).
```

Fig. 3. Auditing trusts predicate.

After importing `alice`'s statement, the code consumer will have:

$$\texttt{alice says believe(<audit ((trust}^{O}\ \texttt{f) x) 0>)}$$

After checking `bob`'s proof, it will have:

$$\texttt{sat}\left(\begin{array}{l}\texttt{<audit ((trust}^{O}\ \texttt{f) x) 0}\\ \quad\texttt{->}\\ \texttt{trusts\_audited ((trust}^{O}\ \texttt{f) x) ((trust}^{O}\ \texttt{f) x)>}\end{array}\right)$$

With a little reasoning, the code consumer obtains:

$$\texttt{mayrun}\left(\texttt{<(int}^{tr} \rightarrow \texttt{int}^{tr}\texttt{)}^{dis}\ \texttt{::}\ \texttt{int}^{tr}\ \texttt{::}\ \texttt{nil>, <(trust}^{O}\ \texttt{f) x>}\right)$$

Going further, as in Section 2, we have much flexibility in defining policies. In particular, each audit requirement need not be discharged with an explicit assertion specific to the requirement. We can allow broader assertions. In the extreme case, once a trusted authority vouches for a program, no auditing is required. We can express this policy with a BCIC rule:

```
mayrun(C, P) :- A says vouchfor(C, P), trusted(A).
```

We can also express policies that distinguish users. First, we modify `mayrun` to include an explicit user argument: `mayrun(U, C, P)` is satisfied when user `U` is allowed to run program `P` in context `C`. Since not all users might trust the same auditors and other authorities, we also introduce predicates `believes` and `trusts` as generalizations of `believe` and `trusted`, respectively, with user arguments. Because provability is absolute, not relative to particular users, no user annotation is needed for `sat`. With these variants, we may for example write the policy:

```
trusts(bob, alice).
trusts(charlie, bob).

believes(U, F) :- sat(F).
believes(U, Q) :- believes(U, P), believes(U, <~P -> ~Q>).
believes(U, F) :- A says believe(F), trusts(U, A).

mayrun(U, C, P) :-
  believes(U, <has_type ~C ~P (annote (usertype 0) tr)>),
  believes(U, <trusts_audited ~P ~P>).
```

However, some classes of policies require more advanced techniques. For instance, we may want to focus the auditing on expressions of certain types, or to exclude expressions that satisfy a given static condition established by a particular static-analysis algorithm. While static conditions can certainly be programmed as Coq predicates, once those predicates are present there need to be corresponding proofs.

Fortunately we have a way of encoding decision procedures in Coq signatures in order to allow BCIC to incorporate those decisions procedures. Our approach is based on proof by reflection (also known as proof by computation) [2, 7], a technique applicable to any theorem prover based on typing that includes convertibility to normal forms. The idea of this technique is that details related to computations are elided from proof terms themselves, and are instead handled automatically by the conversion rules. Using this technique, BCIC can integrate various static analyses. (Some details of our use of proofs by reflection appear in another document [17].)

3.3 Correctness

Much as in Section 2, the proof of correctness relies on an instrumented operational semantics for the language. In this semantics, annotations record `trust` operations. We define a state as a pair that consists of a list of trust identifiers, l, and an expression, e. We define the single-step reduction $(l, e) \rightarrow (l', e')$ mostly as usual, ignoring types and type annotations, and eliminating casts, `trust`s, `distrust`s, and `check`s. The rule for `trust` records the identifier of the trust: $(l, \texttt{trust}^{\texttt{id}}\ e) \rightarrow (\texttt{id} :: l, e)$. We obtain a correctness result that says that, when programs execute, they perform `trust` operations only as authorized:

Theorem 2. *For all programs* `e`*, if* `trusts_audited e e` *is true and* $([], \texttt{e}) \rightarrow^* (l, \texttt{e}')$*, then* `audit e id` *is true for all identifiers* `id` *in* l*.*

While helpful, this theorem does not aim to address the proper criteria for declassification. The study of those criteria, and the guarantees that they may offer, is an active research area (e.g., [15]).

4 Implementation

Our implementation of BCIC includes network communication, cryptographic primitives, a simple user interface, and the machinery for logical queries on policies [19]. We have used Coq's automatic program extraction from proofs to produce a certified correct implementation of the BCIC logic engine [18]. The examples we have presented in this paper are written and work in our implementation of BCIC.

In order to flesh out the examples, we have created corresponding proof generators. Given a target program, the proof generators analyze the program and construct Coq proof terms for it. For the example of Section 2, proof terms establish that strings of audit requirements imply `calls_audited` properties. For the example of Section 3, proof terms yield not only `trusts_audited` properties but also well-typing.

Further, in order to execute the programs that have been analyzed, we have created a simple run-time environment, basically a λ-calculus interpreter. During network communication, statements and proofs must be transmitted between principals. Given that audit statements include copies of programs, performance

could be problematic. Therefore, we replace CIC terms with their hashes. Thus, a set of statements about a single program will need to transmit the program only once, not once per statement. Additional performance improvements might be obtained by caching.

Despite poor, exponential bounds, queries in the certified implementation are reasonably fast in practice. Most policies we have studied require only one or two steps of reasoning before all possible conclusions have been drawn and a query can be answered. In practice the biggest bottleneck is typechecking terms in Coq, not logical deduction. Our implementation caches calls to Coq in order to avoid redundant typechecking.

The code samples in our implementation come from Scheme programs that do not use many Scheme language constructs. Going further, one may attempt to incorporate more of the Scheme language. For the example of Section 3, which requires static typing, a language such as ML may be a better match than Scheme. In this context, auditing may also be applied to special language features that allow a program to break the type system in a potentially unsafe way (e.g., *Obj.magic* in OCaml).

5 Conclusion

This paper develops the view that an audit assertion may encapsulate a non-formal judgment about code in the context of a formal reasoning system. While this judgment will typically be made by a human, perhaps incorrectly, the formal reasoning system provides the means to specify how it fits with security policies and with static code analysis. The formal reasoning system can thus guide the auditing work and ensure its completeness.

More broadly, our work with BCIC indicates the possibility of looking at techniques for verification and analysis in the context of logical security policies. Combinations of reason and authority arise in practice, for instance in systems with signed and verified code (e.g., [5]), though often with ad hoc policies and mechanisms. General theories and tools should support such systems.

Acknowledgments We are grateful to Andrei Sabelfeld and Steve Zdancewic for comments on this work. This work was partly supported by the National Science Foundation under Grants CCR-0208800 and CCF-0524078.

References

1. Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
3. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

4. John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113, 2002.
5. ECMA. C# and common language infrastructure standards, 2007. Online at `http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx`.
6. Perl Foundation. Perl 5.8.8 documentation: perlsec - Perl security. Online at `http://perldoc.perl.org/perlsec.html`.
7. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
8. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
9. Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison Wesley, 1997.
10. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
11. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL'97)*, pages 106–119, 1997.
12. Microsoft Developer Network. About ActiveX controls, 2007. Online at `http://msdn2.microsoft.com/en-us/library/Aa751971.aspx`.
13. Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.
14. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
15. Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations (CSFW'05)*, pages 255–269, 2005.
16. The Coq Development Team. The Coq proof assistant. http://coq.inria.fr/.
17. Nathan Whitehead. Towards static analysis in a logic for code authorization. Manuscript.
18. Nathan Whitehead. A certified distributed security logic for authorizing code. In *Proceedings of the 2006 TYPES Conference*, 2007. To appear.
19. Nathan Whitehead and Martín Abadi. BCiC: A system for code authentication and verification. In *Proceedings of the 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, pages 110–124, 2005.
20. Nathan Whitehead, Martín Abadi, and George Necula. By reason and authority: A system for authorization of proof-carrying code. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 236–250, 2004.