By Reason and Authority: A System for Authorization of Proof-Carrying Code

Nathan Whitehead
Department of Computer Science
University of California, Santa Cruz
nwhitehe@cs.ucsc.edu

Martín Abadi
Department of Computer Science
University of California, Santa Cruz
abadi@cs.ucsc.edu

George Necula
Computer Science Division
University of California, Berkeley
necula@cs.berkeley.edu

Abstract

We present a system, BLF, that combines an authorization logic based on the Binder language with a logical framework, LF, able to express semantic properties of programs. BLF is a general system for specifying and enforcing policies that rely on both reason and trust. In particular, BLF supports extensible software systems that employ both digitally signed code and language-based security, especially proof-carrying code. We describe BLF, establish some of its fundamental properties, and explain its use.

1. Introduction

Modern software systems are often extensible, through software upgrades, with third-party add-ons or components, or with applets. Thus, the extension techniques and policies vary significantly, and so does the level of trust in the producer or distributor of the extension code. Despite this diversity, there is much interest in general mechanisms that can be used to allow system extensions, even untrusted ones, without compromising the stability and security of the host system.

Mechanisms based on digital signatures (e.g., [13, 22]) can ensure that the untrusted code, or data, is endorsed by a trusted party. Digital signatures are relatively easy to use and are fairly well understood. On the other hand, digital signatures have several limitations with respect to mobile-code security. Each safety policy based on digital signatures is likely to name only a small number of principals as trusted to provide code with certain safety properties. The presence of a digital signature from somebody we do not know is not very useful. Moreover, a digital signature, even from somebody we trust, does not ensure the lack of mistakes in the code—it only ensures that the mistakes came from the expected principal.

Language-based security mechanisms (e.g., [8, 17, 18, 20]) overcome some of these limitations of digital signatures. They are largely agnostic on the origin of the code, thus can be used on code coming from unknown principals. They are based on the semantics of the code, thus they can prevent even unintentional mistakes. However, this benefit does not come for free. Type-based mechanismscan enforce only predetermined classes of type safety policies. Proof-carrying code (PCC) is more open-ended and can be used to enforce, in principle, any desired semantic property. A PCC checker is parameterized by a description of the safety conditions for certain operations (e.g., the preconditions for a memory operation or an `open` system call to be safe), and by a set of proof rules that establish the acceptable ways to construct the proofs of the safety conditions. Existing work on PCC [3, 5, 20, 21] assumes that the untrusted proof producer and the receiving host negotiate beforehand a mutually acceptable set of proof rules. Moreover, as the complexity of the property being verified increases, so does the opportunity for error and the proof generation burden. Even though this burden is entirely on the untrusted code producer, who is assumed to have more intimate knowledge of the code, purely semantic mechanisms can sometimes be impractical and perhaps even inappropriate.

Since digital signatures can verify where the code is from but not what the code does, and language-based mechanisms can verify what the code does but not where it is from, it is reasonable to expect that a synergistic combination of such mechanisms would be useful. In particular, the availability of digital signatures can provide a way to control and leverage the PCC proof effort: the vast majority of the safety conditions can be proved as usual, but a

few of the more complex properties (perhaps beyond the scope of previous PCC systems) can be "proved" by signature from a trusted party. The trusted party does not have to certify the entire program, but needs to sign only the validity of precise statements about a small number of points in the program. For example, these points might be where sensitive information is declassified in apparent violation of an information-flow policy (e.g., casts in Jif [19]). Moreover, a public-key infrastructure can be extended to work as a proof-rule infrastructure. Such a proof-rule infrastructure provides a framework through which principals come to trust proof rules and reasoning systems, for instance the proof rules of a particular type system. Even without such an infrastructure, digitally signed statements can serve for establishing safety policies to be used with PCC.

In this paper we present a system that combines a general logical framework, LF [11], with an authorization logic based on Binder [7]. We call our system BLF. Terms in LF that represent programs, properties, and proofs are embedded in the authorization logic. A code consumer constructs a policy in the logic which describes trust relationships with other principals, defines predicates declaratively, and may include axioms for reasoning. Statements made by other principals are imported into the local policy. Reasoning in the authorization logic then determines whether a query (request) given to the resulting policy is derivable (authorized).

One may imagine alternative systems that integrate digital signatures and proofs—and we have tried several, as we explain below. We have chosen BLF because it is intuitive (in our opinion), and flexible enough to support a great variety of scenarios; it is also amenable to formal analysis, as we demonstrate. In particular, we establish a conservativity result and develop an algorithm for reasoning about well-behaved policies.

We have implemented the main components of BLF, such as an LF typechecker, a decision algorithm, a frontend, and mechanisms for importing statements, without however linking the implementation to a reference monitor. The implementation served us in validating elements of our design, but did not lead to any surprises.

Contents In section 2 we present examples introducing the main features of our system. Section 3 presents a syntax and proof rules. It also presents our two main technical results about BLF: a theorem that well-formed policies conservatively extend LF and an algorithm for deciding whether a formula is derivable from a given set of facts. Section 4 includes discussion about the design of the system and describes some of the variations we considered. Section 5 concludes. An appendix contains proofs and describes an additional example.

Related work Combinations of reason and trust appear throughout computer security, though with different purposes, characteristics, power, and generality. For instance, the processing of chains of X.509 certificates requires both certificate chaining (a form of reasoning) and trust in certification authorities.

Another combination of reason and trust arises in interesting work on proof-carrying authentication (PCA) [4, 6]. There, reasoning about authentication and authorization is carried out in a logic similar to the one that underlies Binder. Requests are accompanied by explicit proofs in that logic. PCA is not directly concerned with certifying semantic code properties in extensible software systems (unlike PCC). In contrast, BLF, like PCC, relies on explicit proofs at the LF level and, like Binder, BLF uses a decision procedure for reasoning about authorization. Thus, although both PCA and BLF deal with proofs and logic-based authorization, they address different problems, with different techniques.

Recent work further defines a linking logic on top of PCA [14]. This logic is tailored for supporting the secure linking of untrusted components; in particular, the logic can be used in the formal description of .NET linking policies. The logic relies on code properties to determine whether components may be linked and executed. For each property, a set of property authorities may be trusted to decide whether code satisfies the property. Those authorities are basically black boxes. Although the linking logic is compatible with the use of PCC [14, section 6.1], it does not subsume PCC. In contrast, BLF includes a way to reason logically about properties, thus supporting and extending classical PCC.

2. The system in action

In this section we introduce BLF through three examples. The first is an extended example that allows us to introduce and illustrate the main features of the system. The second example shows how rulesets may be treated as first-class objects. The third example shows how trust annotations may be added to reasoning chains.

2.1. Cell phone/game device example: informal description

Consider the situation in which a combination cell phone/game device[1], referred to as a *host*, desires to download a game from an untrusted developer. The user, or *code consumer*, requires memory safety of programs run on the host. A program that violates memory safety might overwrite data such as phone numbers and appointments stored on the device. Suppose that the cell phone service also offers network services and charges a fixed price for every packet sent. A rogue program could be very expensive.[2]

1 The N-GAGE by Nokia is a real-life example of this type of device.

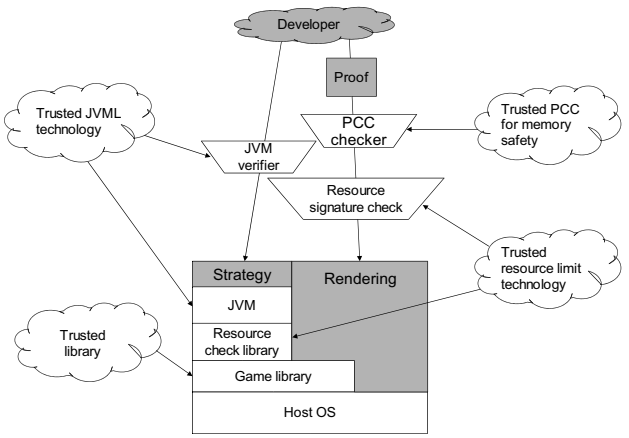

Figure 1. Cell phone/game device example

To protect the user, the host operating system requires all untrusted code to be memory safe and to send less than X packets per second. The host OS allows for multiple ways in which these properties can be established satisfactorily. For memory safety, the host accepts either code written in the Java Virtual Machine Language (JVML) [17], or code carrying proofs of memory safety with respect to a certain set of proof rules. For network usage limits, the host is not going to require proofs, since these are typically hard to construct for interesting programs. It does accept, however, certificates of safety from a trusted principal. Alternatively, code linked with a library that performs dynamic checking to enforce resource quotas is also believed to satisfy the network quota.

The downloaded game is distributed as two modules, a *Strategy* module written in JVML and a *Rendering* module written in native code. The developer provides a proof that *Strategy* typechecks and a proof that *Rendering* is memory safe. The JVML code is compiled by the JVM, then linked with a library that dynamically enforces network resource limits. *Rendering* is asserted to satisfy the network resource limit by a trusted resource signer. Finally, both modules are linked with a trusted library *Gamelib* and executed on the host operating system. This situation is depicted in Figure 1.

PCC and digital signatures individually are not flexible enough to handle this example; we treat this example in BLF as an informal introduction to the main concepts and notations of the system.

2 There already exist malicious programs that secretly use the unsuspecting user's modem to dial high-cost international phone numbers.

2.2. BLF overview

To formalize the example we start with a brief introduction to BLF. Within BLF there are several types of objects. *Policies* are sets of formulas that express the trust assumptions and reasoning framework of the code consumer. *Principals* are public keys, and represent things that make statements. For simplicity we will refer to principals by name; in reality policies may refer to actual public keys. A *ruleset* declares constructors for LF terms along with proof rules; in LF terminology, a ruleset is an LF signature. We use the name ruleset to avoid confusion with digital signatures. *LF terms* represent anything described using constructors declared by a ruleset, including programs, properties, and proofs. *Predicates* are regular Prolog-style predicates of fixed arity, and *atomic statements* are terms consisting of a predicate with the correct number of arguments.

Binder is an extension of Datalog [25] that adds the `says` construct [1]. (Recall that Datalog is Prolog without function symbols.) Binder also defines a procedure whereby principals (called contexts in Binder) can make statements and export them to other contexts. A statement X signed by context U will be imported as U `says` X. Binder restricts `says` so that `says` can never be nested. This means that if V signs a statement U `says` X, the signed statement cannot be imported as V `says` U `says` X. The original signed statement X must be imported directly from U. BLF follows Binder in this respect.

To embed LF into Binder, we introduce two special predicates `sat` and `believe`. The `sat` predicate takes as argument an LF type `T`. Its interpretation is that there exists an object of type `T` within the current ruleset. The `believe` predicate is analogous, but the interpretation is that it is believed that there is an object of type `T`, usually because a trusted principal said so. Proofs and properties follow the judgments-as-types paradigm [11]. This means that a property about program `P`, e.g., `safe P`, is a type. An object of type `safe P` will be a proof that `P` is safe.

If a proof of `T` exists, then we conclude `sat T` and `believe T`. If a trusted principal tells us to believe that `T` is true, we conclude `believe T`. This allows us to maintain the distinction between actual existence and assumed existence of proof objects.

We also allow quantification of variables over LF terms and rulesets. When a variable is quantified over LF terms, the quantification includes a type for the variable. For this reason all quantification is written explicitly rather than in the Prolog style of implicit universal quantification.

2.3. Rulesets

Rulesets define constructors for LF terms and define the allowable proof rules. This section is a review of rulesets

and an explanation of how we use them. The reader unfamiliar with LF may find it helpful to review existing descriptions of LF and Twelf [2, 11, 24]. BLF uses Twelf syntax conventions. In particular, lambda expressions are denoted with square brackets and dependent types with curly brackets. An arrow notation, `->`, indicates functional types.

A simple example of a ruleset that defines natural numbers and proof rules for proving evenness is:

```
nat : type.
0 : nat.
s : nat -> nat.
even : nat -> type.
evenz : even 0.
evenss : {X:nat} (even X ->
                  even (s (s X))).
```

Terms of type `nat` include `0` and `(s (s (s 0)))`. The type `(even (s (s 0)))` represents a judgment; an object of type `(even (s (s 0)))` is a proof that 2 is even. The object `((evenss 0) evenz)` is such a proof. The subterm `(evenss 0)` has type `(even 0 -> even (s (s 0)))`, which when applied to `evenz` of type `(even 0)` yields the result `(even (s (s 0)))`. Note that a ruleset is a definition, neither right nor wrong on its own; the correctness of a ruleset (such as the one above) is dependent on the intended interpretation.

Since we use rulesets for encoding program properties, we must first explain how we represent programs. One approach is to require programs to be written in a type-safe programming language and then only check types at the source level and assume that executing a type-safe program is memory safe. An alternative approach is to check the actual native machine code that will run on the hardware. Code is written using a subset of the machine instructions and includes typing annotations. Programs written in such a typed assembly language (TAL) can be proven safe and executed without any assumptions about an interpreter or compiler. Our example includes both situations, so we must have multiple rulesets. One will allow us to specify programs in JVML, the other in typed assembly language.

Rulesets are quite complicated, so we will not present complete rulesets for our examples. The interested reader is referred to other work on PCC [20] and TAL [18], as well as the example in the appendix for more details. We assume that the ruleset `R_tal` describes constructors for typed assembly language programs with type `prg`, and that the ruleset `R_jvml` describes JVML programs with type `jprg`.

To allow us to reason about linking separate program modules together, the ruleset `R_tal` includes rules that define the predicate `link:prg->prg->prg->type`. The predicate `link P Q R` is satisfied when `R` is the result of linking `P` and `Q`. The ruleset `R_tal` also defines predicates representing memory safety and resource quota satisfaction, `safe:prg->type` and `economical:prg->type`. Rules for deriving `safe P` are included, but there are no rules for proving `economical P`.

The rules in `R_jvml` include definitions for the predicates `typechecks:jprg->type` and `compile:jprg->prg->type`. The predicate `typechecks J` is satisfied when `J` typechecks as a JVML program. The predicate `compile J P` is satisfied when `P` is a TAL program that corresponds to the JVML code `J`. Note that `compile` in `R_jvml` refers to TAL code, so `R_jvml` must be an extension of `R_tal`.

2.4. Policy

The goal of our policy is to describe when we have gathered enough confidence that programs are safe to execute. In the example, to run a program we must believe that it is memory safe and that it is economical with resources. The predicate `mayrun P` indicates that program `P` may execute.

Mentions of LF predicates and types (such as `prg` and `typechecks`) must somehow be linked to rulesets (LF type signatures). We have adopted the syntax `use R in ... end` to denote the scope of rulesets. Quantification over LF terms of type `T` is represented using `forallobj X:T` and `existsobj X:T`. Prolog quantification over atoms is represented using `forall X` and `exists X`. The remaining syntax is borrowed from Prolog. Conjunction is represented using commas, disjunction with semicolons, and reverse implication ("if") with the symbol `:-`. For example, we may write:

```
use R_tal in
  forallobj P:prg
    mayrun(P) :- believe(safe P),
                 believe(economical P).
end
```

We use `believe` instead of `sat` because we wish to allow digital signatures of these properties from trusted authorities. We also want to allow chains of reasoning that include statements believed based on authority and not proof. The conclusion of such a chain of reasoning will involve `believe` and not `sat` because `sat` is only derivable when there is an explicit proof object.

It is also possible to extend the policy with actual user requests to run programs. We can include:

```
use R_tal in
  forall U
  forallobj P:prg
    run(P) :- mayrun(P), user(U),
              U says run(P).
end
```

In this case a user U requests to run a program P by signing the statement run(P). BLF then imports this statement as U says run(P); if BLF can derive run(P) then the program is run. Many other variants are possible. For example, the run predicate could take many arguments indicating various parameters about which capabilities should be given to the program, which user the program should run as, etc.. Determining which keys belong to authorized users can also be performed in the Binder part of BLF (see [7] for examples). For clarity, we limit this example and this paper to definitions of a simple mayrun predicate.

2.4.1. JVML typechecking Part of our policy states that we believe that if a program P written in JVML typechecks, then the compiled version of that program Q will be safe according to the PCC definition of safety.

```
use R_tal in
  forallobj Q:prg
  believe(safe Q) :-
    use R_jvml in
      existsobj P:jprg
        sat(typechecks P),
        sat(compile P Q).
    end
  end
```

Note how the R_jvml ruleset is nested inside the R_tal ruleset. Statements inside more than one ruleset are ignored if the rulesets conflict (that is, they attempt to define the same identifier with different types). If the rulesets do not conflict, the statements inside may use definitions from any of the enclosing rulesets. Note also how the scope of the inner ruleset R_jvml is arranged not to enclose the fragment believe(safe Q). If this were not true, the conclusion believe(safe Q) would be within the scope of both rulesets and could not directly be used to conclude mayrun(Q). The rule for mayrun (in section 2.4) requires safety to be proven in the scope of R_tal alone.

2.4.2. Trusted libraries The principal lib_signer says which libraries should be trusted. We believe that any library lib_signer certifies as trusted is safe and satisfies the resource quota.

```
use R_tal in
  forallobj L:prg
    believe(safe L) :-
      lib_signer says trusted(L).
    believe(economical L) :-
      lib_signer says trusted(L).
end
```

In a more elaborate version of this example, libraries may be deemed safe only when linked with programs that satisfy certain conditions. These conditions can be expressed by predicates defined using proof rules supplied by the library provider. Section 2.6 shows how rulesets may be endorsed by principals, making such a scheme practical.

There is another principal res_signer that we trust to sign programs or libraries that satisfy the correct resource quotas. We also trust res_signer to tell us of a library that dynamically enforces the quotas at run-time. We believe that any program linked with this library will satisfy the resource constraints.

```
use R_tal in
  forallobj P:prg
  forallobj L:prg
  forallobj R:prg
    believe(economical P) :-
      res_signer says
        believe(economical P).
    believe(economical R) :-
      sat(link P L R),
      res_signer says
        dynamic_enforcer(L).
end
```

Practical programming languages might require more than merely linking with one library; they might also require that the library be initialized at program startup. This could be represented by adding more predicates, but for simplicity we limit the requirements to a simple link.

2.4.3. Linking lemmas Suppose one can prove that safe programs linked together remain safe, and economical programs linked together remain economical. The proofs of these facts are not included directly in our policy, but after being checked they will allow us to derive the following statements:

```
use R_tal in
  sat({p:prg}{q:prg}{r:prg} safe p ->
      safe q -> link p q r -> safe r).
  sat({p:prg}{q:prg}{r:prg}
      economical p -> economical q ->
      link p q r -> economical r).
end
```

Recall that the notation {x:T} indicates a dependent type, and the notation U -> V indicates functional types. Treating LF terms as logic propositions, dependent types represent universal quantification and functional types represent implication.

2.5. Imported statements

The principals lib_signer and res_signer generate and sign several statements which are imported into the local policy. In reality each imported statement will have its

own ruleset scope; we present them here in an equivalent form with one ruleset scope for clarity. The imported statements will appear as:

```
use R_tal in
  lib_signer says trust(P_gamelib).
  res_signer says
    dynamic_enforcer(P_enforcer).
  res_signer says
    believe(economical P_render).
end
```

Several actual proofs are also provided by `res_signer` and `developer`. After being checked, the single proof provided by `res_signer` yields `sat(safe P_enforcer)`. The proofs from `developer` let us conclude:

```
use R_tal in
  sat(safe P_render).
  use R_jvml in
    sat(typechecks P_strategy).
    sat(compile P_strategy
                P_cstrategy).
  end
  sat(link P_cstrategy P_enforcer
      P_lcstrategy).
  sat(link P_lcstrategy P_gamelib
           P_linked).
  sat(link P_linked P_render P_final).
end
```

From our previously developed policy and these imported statements we can derive `mayrun(P_final)`. To show this we first consider the strategy module. Because `P_strategy` typechecks and is compiled into `P_cstrategy`, the single rule from section 2.4.1 implies `believe(safe P_cstrategy)`. A proof shows that `P_enforcer` is safe, so linking `P_cstrategy` with `P_enforcer` yields another program `P_lcstrategy` that is also safe (by the rules from section 2.4.3). Since `P_lcstrategy` is linked with the library `P_enforcer` from `res_signer` that dynamically enforces resource quotas, using the rules from section 2.4.2 we can derive `believe(economical P_lcstrategy)`. Thus, we conclude `believe(safe P_lcstrategy)` and `believe(economical P_lcstrategy)`.

The trusted library signer says that `P_gamelib` is a trusted library, so using the rules from section 2.4.2 we can conclude `believe(safe P_gamelib)` and also `believe(economical P_gamelib)`. Since the module `P_linked` is `P_lcstrategy` linked with the game library, `P_gamelib`, and we believe both `P_lcstrategy` and `P_gamelib` are safe and economical, by the linking rules in section 2.4.3 we also believe that `P_linked` is safe and economical. In addition, the developer provided a proof that `P_render` is safe, and the resource signer said it was economical, so we conclude `sat(safe P_render)` and `believe(economical P_render)`.

Since both `P_linked` and `P_render` are safe and economical, we conclude `believe(safe P_final)` and `believe(economical P_final)`. Finally we conclude `mayrun(P_final)` and the user is allowed to play the downloaded network game on the cell phone.

2.6. A second example: endorsing rulesets

Another powerful feature of BLF is that rulesets are first-class objects. A principal may digitally sign statements that tell the local policy which ruleset to use and which predicate should be applied to programs to check for safety. This local policy fragment shows how this can be done:

```
forallrules R
  use R in
    forallobj P:prg
    forallobj T:prg->type
      mayrun(P) :- sat(T P),
                   useruleset(R),
                   useprop(T).
  end
```

A local principal `Alice` can then construct a ruleset `R_0` which defines the property `safe`, and digitally sign the statements `useruleset(R_0)` and `useprop(safe)`.

Having rulesets and properties as first-class objects that can be communicated between principals allows the construction of systems that reason about which rulesets and properties to use. Thus, if the proof techniques or rules change because of bug fixes or new functionality, code consumers can be informed of the new rulesets dynamically without changing their existing policy. Libraries, for example, can provide explicit proof rules to define properties that code using the library should satisfy. Virtual machines can provide proof systems for proving program safety. Moreover, the explicit mention of rulesets may contribute to security by forcing the writers of policies to decide, consciously, on which rulesets to rely. Existing PCC systems generally embed the proof system in the proof checker and make the code consumer implicitly trust the hidden embedded proof system. Of course, usually, they also attempt to ensure that the proof system is small and reliable.

2.7. A third example: trust annotations

By default there are two distinctions made about statements from other principals: whether they come with valid proofs, and whether they should be believed. This might not be fine-grained enough for some applications. We might want to give principals varying levels of trust. The trust level

in any conclusion requiring many deductions should never exceed the trust we place in any one principal on which the deduction relies.

To this end, assertions and proofs can be annotated with trust levels from a lattice. The final trust level will be the greatest lower bound of trust we place in the derivation steps. For example, consider the three trust levels, untrusted, semi-trusted, and trusted (with trusted on top). The policy must define how to calculate the greatest lower bound of any two trust levels. Every predicate about code is extended to include a trust annotation. Instead of `safe(P)` indicating that `P` is safe, we define `safe(P,L)` which indicates that we believe `P` is safe and the derivation that `P` is safe is trusted with trust level `L`. We accept whatever any principal says, but annotate the predicate with the trust level we assigned to the principal. If an implication relies on several hypotheses, we assign the trust level of the conclusion to the greatest lower bound of trust levels in the hypotheses. If we actually have a proof, we annotate the predicate with `trusted`.

The following example shows a policy with two predicates, `safe(P)` and `correct(P)`, whose conjunction yields `mayrun(P)`. The policy defines the trust lattice, the trust level of each principal, and the annotated predicates. The ruleset `R_0` defines two arbitrary properties `safe:prg->type` and `correct:prg->type`.

```
use R_0 in
  forall X, Y, Z
    lte(untrusted, semitrusted).
    lte(semitrusted, trusted).
    lte(X, X).
    lte(X, Z) :- lte(X, Y), lte(Y, Z).
    glb(X, Y, X) :- lte(X, Y).
    glb(X, Y, Y) :- lte(Y, X).

  trustlevel(alice, trusted).
  trustlevel(bob, semitrusted).

  forallobj P:prg
  forall L, K, U, V
    safe(P, trusted) :- sat(safe P).
    safe(P, L) :-
      K says believe(safe P),
      trustlevel(K, L).
    correct(P, trusted) :-
      sat(correct P).
    correct(P, L) :-
      K says believe(correct P),
      trustlevel(K, L).
    mayrun(P, L) :- safe(P, U),
                    correct(P, V),
                    glb(U, V, L).
end
```

3. Formal description of BLF

In this section we present a formal description of the syntax, proof rules, and the import function of BLF. We also present a result showing that BLF conservatively extends LF and describe a decision procedure for BLF formulas. Proofs of the theorems in this section appear in the appendix.

3.1. Syntax

The grammar for the syntax of the language is presented in Figure 2. The syntax is presented in BNF form using the same notational conventions as Binder [7]. The syntax defines a predicate Horn logic using d-formulas and g-formulas [9]. In Prolog terms, d-formulas represent *programs* and g-formulas represent *queries*. Any d-formula can be rewritten to standard Prolog clausal form. Organizing the syntax in this way simplifies the proof rules and implementation.

An additional syntax appears in Figure 3. Only rules that differ from Figure 2 are shown. This syntax annotates the `existsobj`, `forallobj`, `sat`, and `believe` predicates to explicitly record to which ruleset every LF term refers. Statements inside `use R in ... end` environments will be annotated with the ruleset `R`. Statements inside multiple ruleset scopes R_1 and R_2 will be annotated with the concatenation of the rulesets, $R_1; R_2$. An annotation operator $[\cdot]$ translates from the user syntax (Figure 2) to the annotated syntax (Figure 3). The annotation operator $[\cdot]_R$ is defined in Figure 4; the general annotation operator $[\cdot]$ is $[\cdot]_R$ with an empty R.

3.2. Proof rules

The proof rules are listed in Figure 5. The proof rules are written in the annotated syntax, and are based on a sequent presentation of predicate Horn logic [10]. In a sequent, d-formulas appear on the left of the sequent and g-formulas on the right. The first six proof rules are standard sequent rules for propositional logic. Rules 7 through 12 deal with quantification, and Rules 13 through 20 define `sat` and `believe`.

The notation $R \vdash_{LF} O : T$ indicates that object O has type T in LF signature R by the standard LF typing rules. The x not occurring in Rule 17 and Rule 18 is important, since x would normally be substituted with the object of type T, but we do not have explicit proof objects. One can still substitute when the type is actually dependent, but this requires Rule 14 or 16. Canonical forms in Rule 19 and Rule 20 refer to canonical forms in LF [11].

⟨gform⟩ ::= ⟨atomic⟩ | ⟨gform⟩, ⟨gform⟩ | ⟨gform⟩; ⟨gform⟩
| `exists` ⟨var⟩ ⟨gform⟩
| `existrules` ⟨rsvar⟩ ⟨gform⟩
| `existsobj` ⟨termvar⟩ : ⟨lfterm⟩ ⟨gform⟩
| `use` ⟨ruleset⟩ `in` ⟨gform⟩ `end`

⟨dform⟩ ::= ⟨atomic⟩ | ⟨dform⟩, ⟨dform⟩
| ⟨dform⟩ :- ⟨gform⟩ | `forall` ⟨var⟩ ⟨dform⟩
| `forallrules` ⟨rsvar⟩ ⟨dform⟩
| `forallobj` ⟨termvar⟩ : ⟨lfterm⟩ ⟨dform⟩
| `use` ⟨ruleset⟩ `in` ⟨dform⟩ `end`

⟨atomic⟩ ::= [⟨principal⟩ `says`] `sat`(⟨lfterm⟩)
| [⟨principal⟩ `says`] `believe`(⟨lfterm⟩)
| [⟨principal⟩ `says`] ⟨predicate⟩
([⟨argument⟩ [, ⟨argument⟩]*])

⟨var, rsvar, termvar, lfvar⟩ ::= ⟨identifier⟩

⟨policy⟩ ::= [⟨dform⟩.]+

⟨predicate⟩ ::= ⟨identifier⟩

⟨principal⟩ ::= ⟨key⟩ | ⟨var⟩

⟨argument⟩ ::= ⟨identifier⟩ | ⟨key⟩ | ⟨var⟩ | ⟨lfterm⟩
| ⟨ruleset⟩

⟨ruleset⟩ ::= ⟨rsvar⟩ | ⟨actualruleset⟩ | ⟨rsvar⟩; ⟨ruleset⟩

⟨actualruleset⟩ ::= `ruleset` ([⟨identifier⟩ : ⟨lfterm⟩.]*)

⟨lfterm⟩ ::= ⟨termvar⟩ | ⟨lfvar⟩ | `type` | ⟨lfterm⟩ ⟨lfterm⟩
| ⟨lfterm⟩ → ⟨lfterm⟩ | {⟨lfvar⟩ : ⟨lfterm⟩} ⟨lfterm⟩
| `[`⟨lfvar⟩ : ⟨lfterm⟩`]` ⟨lfterm⟩

Figure 2. Syntax

3.3. Import

There are two partial functions, $i_d : \langle\text{key}\rangle \times \langle\text{dform}\rangle \to \langle\text{dform}\rangle$ and $i_g : \langle\text{key}\rangle \times \langle\text{gform}\rangle \to \langle\text{gform}\rangle$ that import d-formulas and g-formulas from an outside context. The functions are partial because there can be only one level of quoting in formulas. Let U be the principal from which we import a statement. In Binder, clauses can be imported from U only if the head of the clause is not already quoted. If the original clause is H :- B, then the imported clause will be U `says` H :- B' where B' is the original body with every unquoted formula F replaced with U `says` F.

Analogously in BLF, g-formulas G with no `says` get translated to U `says` G. A g-formula of the form V `says` G remains unchanged in translation. A d-formula D without `says` gets translated to U `says` D, while

⟨gform⟩ ::= ⟨atomic⟩ | ⟨gform⟩, ⟨gform⟩ | ⟨gform⟩; ⟨gform⟩
| `exists` ⟨var⟩ ⟨gform⟩
| `existrules` ⟨rsvar⟩ ⟨gform⟩
| `existsobj′` ⟨ruleset⟩ ⟨termvar⟩ : ⟨lfterm⟩ ⟨gform⟩

⟨dform⟩ ::= ⟨atomic⟩ | ⟨dform⟩, ⟨dform⟩
| ⟨dform⟩ :- ⟨gform⟩ | `forall` ⟨var⟩ ⟨dform⟩
| `forallrules` ⟨rsvar⟩ ⟨dform⟩
| `forallobj′` ⟨ruleset⟩ ⟨termvar⟩ : ⟨lfterm⟩ ⟨dform⟩

⟨atomic⟩ ::= [⟨principal⟩ `says`] `sat′`(⟨ruleset⟩, ⟨lfterm⟩)
| [⟨principal⟩ `says`] `believe′`(⟨ruleset⟩, ⟨lfterm⟩)
| [⟨principal⟩ `says`] ⟨predicate⟩
([⟨argument⟩ [, ⟨argument⟩]*])

⋮

Figure 3. Internal syntax

$$[A, B]_R = [A]_R, [B]_R$$
$$[A; B]_R = [A]_R; [B]_R$$
$$[D \texttt{ :- } G]_R = [D]_R \texttt{ :- } [G]_R$$
$$[\texttt{forall } x\ D]_R = \texttt{forall } x\ [D]_R$$
$$[\texttt{exists } x\ G]_R = \texttt{exists } x\ [G]_R$$
$$[\texttt{forallrules } r\ D]_R = \texttt{forallrules } r\ [D]_R$$
$$[\texttt{existrules } r\ G]_R = \texttt{existrules } r\ [G]_R$$
$$[\texttt{forallobj } x : T\ D]_R = \texttt{forallobj}'\ R\ x : T\ [D]_R$$
$$[\texttt{existsobj } x : T\ G]_R = \texttt{existsobj}'\ R\ x : T\ [G]_R$$
$$[\texttt{use } R'\ \texttt{in } A\ \texttt{end}]_R = [A]_{R;R'}$$
$$[P\ \texttt{says}\ X]_R = P\ \texttt{says}\ [X]_R$$
$$[\texttt{sat}(T)]_R = \texttt{sat}'(R, T)$$
$$[\texttt{believe}(T)]_R = \texttt{believe}'(R, T)$$
$$[\texttt{P}(\alpha_1, \alpha_2, \ldots, \alpha_n)]_R = \texttt{P}(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

Figure 4. Annotation function

d-formulas of the form V `says` D are untranslatable.

Suppose the code consumer has a policy which is a d-formula D. If the d-formula D' is signed by key U, and $i_d(U, D')$ exists, then the code consumer extends D to $D, i_d(U, D')$. Once the code consumer has extended the policy with all visible signed statements, it may execute queries (g-formulas) against the policy.

These functions deal with formulas that contain rulesets, programs, and types. Rulesets and programs especially may be quite large, so always referring to them by value may re-

$$\frac{}{A, \Gamma \Rightarrow \Delta, A} \quad A \text{ is atomic} \tag{1}$$

$$\frac{\phi(\Gamma) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \phi(\Delta)}{\Gamma \Rightarrow \Delta} \quad \phi \text{ is a permutation} \tag{2}$$

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma, D \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, G} \tag{3}$$

$$\frac{\Gamma, D, D \Rightarrow \Delta}{\Gamma, D \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, G, G}{\Gamma \Rightarrow \Delta, G} \tag{4}$$

$$\frac{D_1, D_2, \Gamma \Rightarrow \Delta}{(D_1, D_2), \Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, G_1 \quad \Gamma \Rightarrow \Delta, G_2}{\Gamma \Rightarrow \Delta, (G_1, G_2)} \tag{5}$$

$$\frac{\Gamma \Rightarrow \Delta, G \quad D, \Gamma \Rightarrow \Delta}{D \text{ :- } G, \Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, G_1, G_2}{\Gamma \Rightarrow \Delta, (G_1; G_2)} \tag{6}$$

$$\frac{D[A/x], \texttt{forall}\ x\ D, \Gamma \Rightarrow \Delta}{\texttt{forall}\ x\ D, \Gamma \Rightarrow \Delta} \tag{7}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{exists}\ x\ G, G[A/x]}{\Gamma \Rightarrow \Delta, \texttt{exists}\ x\ G} \tag{8}$$

$$\frac{D[O/x], \texttt{forallobj}'\ R\ x : T\ D, \Gamma \Rightarrow \Delta \quad R \vdash_{LF} O : T}{\texttt{forallobj}'\ R\ x : T\ D, \Gamma \Rightarrow \Delta} \tag{9}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{existsobj}'\ R\ x : T\ G, G[O/x] \quad R \vdash_{LF} O : T}{\Gamma \Rightarrow \Delta, \texttt{existsobj}'\ R\ x : T\ .\ G} \tag{10}$$

$$\frac{D[R/r], \texttt{forallrules}\ r\ D, \Gamma \Rightarrow \Delta}{\texttt{forallrules}\ r\ D, \Gamma \Rightarrow \Delta} \tag{11}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{existrules}\ r\ G, G[R/r]}{\Gamma \Rightarrow \Delta, \texttt{existrules}\ r\ G} \tag{12}$$

$$\frac{R \vdash_{LF} O : T}{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, T)} \tag{13}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, \{x : T\}B) \quad R \vdash_{LF} O : T}{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, B[O/x])} \tag{14}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, T), \texttt{sat}'(R, T)}{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, T)} \tag{15}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, \{x : T\}B) \quad R \vdash_{LF} O : T}{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, B[O/x])} \tag{16}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, \{x : T\}B) \quad \Gamma \Rightarrow \Delta, \texttt{sat}'(R, T) \quad x \text{ does not occur in } B}{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, B)} \tag{17}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, \{x : T\}B) \quad \Gamma \Rightarrow \Delta, \texttt{believe}'(R, T) \quad x \text{ does not occur in } B}{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, B)} \tag{18}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, T') \quad T \text{ and } T' \text{ have the same canonical form in ruleset } R}{\Gamma \Rightarrow \Delta, \texttt{sat}'(R, T)} \tag{19}$$

$$\frac{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, T') \quad T \text{ and } T' \text{ have the same canonical form in ruleset } R}{\Gamma \Rightarrow \Delta, \texttt{believe}'(R, T)} \tag{20}$$

Figure 5. Proof rules

sult in an unacceptable performance penalty. For security reasons it is not enough to allow references by name to programs and rulesets; the mapping between names and objects must be trusted as much as the objects will be trusted. BLF itself does not provide any mechanism for avoiding this performance penalty, but standard techniques such as cryptographic hashes in place of actual objects solve this problem.

3.4. Conservativity

An important step in a convincing argument that BLF is a reasonable logic is a formalization of the relationship between BLF and LF. The behavior of the `sat` predicate is entirely determined by LF; we say that BLF is a conservative extension of LF on the `sat` predicate. A policy writer may like assurance that $\texttt{sat}(T)$ is derivable if and only if there actually is an LF object of type T. If there were a subtle flaw in the derivation rules of BLF this might not be true, and the policy writer would have no way of knowing what their policy containing `sat` meant and whether it was correct.

A *d-subformula* of a formula F is a subformula of F that is also a d-formula (with the restriction that atomic formulas are considered to be d-formulas or g-formulas depending on where they occur in F).

Definition 1 *A d-formula* $\texttt{P}$ *is* conservative *if no d-subformula of* $\texttt{P}$ *is an atomic formula of the form* $\texttt{sat}'(R, T)$.

The following theorem says that BLF is conservative over LF for conservative policies.

Theorem 1 *If* Γ *is a set of conservative d-formulas and* $\texttt{sat}'(R, T)$ *is derivable from* Γ*, then there exists an explicit LF proof object* O *such* $R \vdash_{LF} O : T$.

The restriction on Γ not having any d-subformulas that are atomic satisfaction formulas is necessary because an arbitrary Γ might contain "facts" of the form $\texttt{sat}'(R, T)$ which have no proof in LF. Rule 1 from Figure 5 would allow these unprovable facts to be derivable immediately.

3.5. Decision algorithm

An effective reasoning strategy is necessary if a system such as BLF is to be at all practical. Since BLF contains LF, a system capable of representing undecidable and arbitrarily complicated logics, we must limit in some way the kinds of reasoning our system performs. By considering examples such as those from section 2 and the appendix we can identify the requirements for a reasoning strategy. In particular, the reasoning strategy need not search for proof objects of LF types. In the PCC framework, proof objects will be provided (by code producers or other principals) and need only to be checked, not reproduced, by the code consumer.

The reasoning system should, however, be able to combine true propositions using modus ponens and instantiate universally quantified propositions with the actual objects under consideration.

To this end we develop a decision algorithm for "well-behaved" policies. Intuitively, a policy is well-behaved if variables within LF terms do not appear in expressions. However, variables themselves may appear as arguments to predicates. This restriction ensures that the set of expressions under consideration does not change when variables are instantiated. This in turn allows us to develop a decision algorithm based on Datalog-style bottom-up evaluation that is both simple and terminating. Note that we place no restrictions on rulesets; any reasoning system with a representation in LF may be used in BLF.

Bottom-up Datalog evaluation is well-understood [25]. The procedure involves keeping track of a database of ground atomic formulas. The initially empty database is extended in stages by applying the rules in the Datalog program. When a fixed-point is reached and no more extensions are possible, the procedure answers the query using the database. Termination of the extension step is guaranteed because of the lack of function symbols.

The BLF procedure is similar. In BLF, atomic formulas may have an optional `principal says` prefix; this is easy to accommodate. The big difference is that BLF includes LF terms in its universe. The universe of known LF terms is increasing; we get a new term when we apply (`even 0` $\rightarrow$ `poweroftwo 0`) to `(even 0)`. To ensure termination of the BLF procedure we require that the universe of known LF terms be bounded. For well-behaved policies this guarantee is provided by Theorem 2.

Next we define our restriction on policies more precisely. A formula is *well-behaved* if every reference to an LF term is either an object of simple type, a simple type, an object of property type, or a property type. We now define these concepts.

A *constant* is either a reference to an entry in the LF signature, or an application of a constant to a constant. A *simple type* is a type entry in an LF signature of kind `type`, for example `nat`. A *simple constant* is a constant of simple type. For example, `0` and `(s (s (s 0)))` are simple constants. A *predicate type* is a type that has kind `type`, or kind $T \rightarrow A$ where T is a simple type and A is a predicate type. For example, `nat->type` is a predicate type. A *predicate name constant* is a constant with predicate type. For example, `even` is a predicate name constant.

A *base object* is either a simple constant or a variable. If it is a variable, the variable may be an ⟨lfvar⟩ (bound within an LF term), or a ⟨termvar⟩ (bound in a quantification over LF terms).

A *predicate* is an application of a predicate name constant with type $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow$ `type` to base objects of type $T_1, \ldots, T_n$. For example, `even P` and `even (s (s 0))` are predicates.

A *property type* is either a predicate A, a type of the form $\{x : T\}\ P$ where T is a simple type and P is a property type, or $A \rightarrow P$ where A is a predicate and P is a property type. In a property type, the variables in bindings for dependent types are bound to base object variables in arguments to predicates. For example, if the ruleset in section 2.3 included the entry `poweroftwo:nat->type`, then the LF type $\{x$ `: nat`$\}$ (`even` $x \rightarrow$ `poweroftwo` x) would be a property type. A type that is *not* a property type is $\{x$ `: nat`$\}$ (`even` $x \rightarrow$ `even (s (s` x`))`). This is not a property type because (`s (s` x`)`) is not a base object.

The basic operation of our algorithm is extending a universe of known LF types by application of existing types to one another.

Definition 2 *Let U be a set of objects with simple types. Let V be a set of property types such that all simple constants that appear in V also appear in U. The extension function η takes U and V as input and computes a new set of canonical property types derivable from U and V by application.*

To calculate $\eta(U, V)$, for all $p \in V$ let V_p be the set of types derivable from p defined as follows. If $p = \{x : T\}\ B$ then let $V_p = \{(p\ y) \mid y \in U \text{ and } y : T\}$. If $p = A \rightarrow B$ then check if $A \in V$. If so, let $V_p = \{B\}$; otherwise let V_p be empty. If $p = A$ where A is a predicate then let V_p be empty. Then $\eta(U, V)$ will be $\lfloor \cup_{p \in V} V_p \rfloor$, where $\lfloor \cdot \rfloor$ denotes conversion to canonical form including α-renaming all dependent variables to canonical names.

Theorem 2 *Given finite starting sets U of objects of simple type and V_0 of property types, and given $V_i = V_{i-1} \cup \eta(U, V_{i-1})$, there exists a fixed point V_n such that $V_n = V_n \cup \eta(U, V_n)$.*

We can now begin to construct a decision algorithm for BLF. Let P be a well-behaved policy (d-formula) and let Q be a query on the policy (g-formula). These are the inputs to the algorithm; the output will be a binary decision that determines if Q is derivable from P. First extract all rulesets from P and Q; let R represent the set of rulesets. There is no way to construct new rulesets that are not identical to existing rulesets, so R will remain constant. Next extract all ground ⟨argument⟩ constants from P and Q and denote them with A_0. Extract all LF constant objects of simple type into U. Extract all LF constant objects of property type (including provided proofs), and after checking their type store the property types in V_0, annotated with the ruleset to which they refer. The set V_i will represent property types that have explicit proof objects after i rounds of extension (for `sat`). Let $W_0 = V_0$. The set W_i will represent property types that are believed to have proof objects, again annotated with the ruleset to which they refer (for

believe). Let F_0 be the empty set; the set F_i will represent the database of atomic facts after i rounds.

The algorithm proceeds as follows. At stage i, first use the facts in F_{i-1} and the policy P to derive a set of extended facts F_i via standard Datalog resolution. If any conclusion is of the form `sat'(R,T)` then add `T` to V_i. If any conclusion is of the form `believe'(R,T)` then add `T` to W_i. Next extend the LF terms in V_{i-1} using η and set $V_i = \eta(U, V_{i-1})$. Do the same to W_{i-1} and set $W_i = \eta(U, W_{i-1})$. Ensure that $V_i \subseteq W_i$ by adding types to W_i from V_i if necessary. For every type in V_i, add the corresponding `sat` statement to F_i. For every type in W_i, add the corresponding `believe` statement to F_i. Finally, let $A_i = A_{i-1} \cup V_i \cup W_i$; this represents adding newly created LF terms from the LF extension to the Datalog reasoning universe. Repeat these steps until a fixed point is reached. The existence of the fixed point is a corollary of Theorem 2. The final database of facts, F_n, can then be used to answer the original query Q.

4. Design decisions

Our goals in designing BLF are to create a system that is practical, as simple to use as possible, and as extensible as possible while remaining logically grounded. During its development the system went through several variants that we ultimately rejected. In this section we discuss some of the design considerations that led to BLF.

Why LF and Binder We chose LF as the logical framework for proofs about programs because it is powerful and extensible enough to represent several proof systems of interest. There are also good tools based on LF [24]. LF is a standard choice for representing reasoning systems, and in particular is a standard choice for PCC.

Other languages and logics could be used in place of Binder. Some possibilities are logics allowing nested `says` [1], a higher-order logic [4], or other Prolog-like languages with additional capabilities [12, 15, 16]. The choice of Binder as the authorization language is not crucial; we choose Binder because it has the functionality we need and is simple.

Dependent proofs When a code producer proves safety but needs some extra assumptions, these could be represented by new unproven axioms in the signature or by dependent types in the formula itself. In the spirit of LF we use dependent types to model proof dependencies.

Mutual calls and reflection The current design of BLF is one-sided, with the Binder-like part making subcalls to an LF typechecker. An earlier idea was to have LF and Binder as more equal partners, mutually calling each other. LF by default has no notion of principals or Binder predicates so there must be some mapping from Binder to LF to allow LF to call Binder. Constructing a mapping that worked basically required implementing Binder using LF. This is not necessarily a problem, since LF is quite extensible.

Once Binder was implemented in LF, we tried to allow Binder to call back into LF. Since Binder was implemented in LF, this meant allowing reflection in LF. Again, LF was extensible enough and was able to model its own rules. We then had LF and Binder able to call each other, both embedded inside LF.

One interesting advantage of this variant was that it had a way to reason about itself; the system was an LF signature, so this LF signature could be encoded and analyzed inside the system. This allowed a principal to extend the logical system itself and prove that derivations in the new system could be transformed back into the old system. A recipient of this proof could check it and then swap out the entire logical system for the new one.

On the other hand, as in other logical systems, reflection is a source of great difficulties, not to be adopted lightly. We found it hard to understand even the simplest reflective formulas, and to reason about what was provable. Moreover, in the examples we have considered, the use of reflection did not add practical expressiveness over our use of dependent types.

Explicit proof terms Another idea was to include proof terms in policies. Without explicit proof terms in the formulas it is sometimes difficult to tell who produces which proof in the examples. One might also want to attach trust levels to proofs as well as assertions because of the risk that proof systems themselves might be subtly inconsistent.

In the end we decided not to mention proof terms directly for several reasons. First, it simplified the syntax and proof rules of the logic. Also, if every logical statement about satisfaction includes a proof term, the issue of what proof term to use for asserted properties comes up. It is possible to add some sort of `hole` constructor that represents a missing term of a given type [2]. This raises questions about when the `hole` construct is allowed and generally complicates the logic. The strongest argument against including explicit proof terms in policies is that the meaning of policies should not depend on proof terms themselves, but merely on the fact that they exist or are assumed to exist.

5. Conclusion

In this paper we describe a system, BLF, that integrates a logic for expressing authorization policies (Binder) with a logical framework in which correctness properties can be specified and proved (LF). We had two main goals in developing BLF. First, we wanted to be able to supplement the security guarantees provided by digital signatures with semantic guarantees provided by safety proofs. Second, we wanted to simplify the use of safety proofs, by providing

a way to introduce trusted axioms based on digital signatures. These goals are important in the context of software systems built from components originating from a variety of sources. Some code sources are trusted to provide secure code, while others need to rely on proofs to gain our trust. By design, BLF supports both digitally signed code and language-based security, and goes beyond each of them taken in isolation. Unlike previous systems for language-based security, BLF does not suppose an a priori agreement between the code producer and the consumer on what proof rules to use. On the contrary, BLF provides a framework for reaching this agreement. In particular, BLF treats proof rules as first-order objects. Like Binder, BLF can serve for many kinds of authorization decisions (e.g., for file access) and, like LF, BLF can treat fairly arbitrary logical properties (e.g., logical properties of data to be written to a file). Thus, BLF is not limited to reasoning about code and whether it should be executed. Rather, it is a general system that enables security policies based on reason and authority.

6. Acknowledgments

Thanks to Allen Van Gelder for helpful ideas for the Prolog implementation. Thanks to Katia Hayati and the anonymous reviewers for reading this paper and offering suggestions for improvement. Nathan Whitehead's and Martín Abadi's work was partly supported by the National Science Foundation under Grants CCR-0204162 and CCR-0208800. George Necula's work was funded by the National Science Foundation under Grants CCR-9875171, CCR-0085949, CCR-0326577, and CCR-0081588.

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] Andrew W. Appel. Hints on proving theorems in Twelf, February 2000. http://www.cs.princeton.edu/∼appel/twelf-tutorial/.

[3] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[4] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.

[5] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, January 2000.

[6] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium 2002*, pages 93–108, 2002.

[7] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113, May 2002.

[8] ECMA. Standard ECMA-335: Common Language Infrastructure, December 2001. Available online at http://msdn.microsoft.com/net/ecma/.

[9] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[10] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1990. Translated and with appendices by Paul Taylor and Yves Lafont. Available online at http://nick.dcs.qmul.ac.uk/∼pt/stable/Proofs+Types.html.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[12] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, May 2001.

[13] Sebastian Lange, Brian LaMacchia, Matthew Lyons, Rudi Martin, Brian Pratt, and Greg Singleton. *.NET Framework Security*. Addison Wesley, 2002.

[14] Eunyoung Lee and Andrew W. Appel. Policy-enforced linking of untrusted components. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 371–374, September 2003.

[15] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, February 2003.

[16] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, May 2002.

[17] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison Wesley, 1997.

[18] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[19] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, January 1999.

[20] George C. Necula. Proof-carrying code. In *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[21] George C. Necula and Robert R. Schneck. A sound framework for untrusted verification-condition generators. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, July 2003.

[22] Microsoft Developer Network. About ActiveX controls, November 2003. http://msdn.microsoft.com/workshop/components/activex/controls.asp.

[23] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 3(2):75–85, 1997.

[24] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206. Springer-Verlag LNAI 1632, July 1999.

[25] J. D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 2. Computer Science Press, 1988.

Appendix

A. Proofs of theorems

Theorem 1 *If Γ is a set of conservative d-formulas and $\mathtt{sat}'(R,T)$ is derivable from Γ, then there exists an explicit LF proof object O such $R \vdash_{LF} O : T$.*

Proof. The proof of the theorem is by induction on derivations of $\Gamma \Rightarrow \mathtt{sat}'(R,T)$.

Consider which rules from Figure 5 would allow us to derive $\Gamma \Rightarrow \mathtt{sat}'(R,T)$. Several proof rules cannot be the last step in a derivation of $\Gamma \Rightarrow \mathtt{sat}'(R,T)$ and so do not need to be considered. Rule 3(right), Rule 5(right), Rule 6(right), Rule 8, Rule 10, Rule 12, Rule 15, Rule 16, and Rule 18 are of this form.

There are two base cases, Rule 1 and Rule 13. The atomic case is true because of the restriction on Γ having only conservative d-formulas. This means no entry of Γ can be $\mathtt{sat}'(R,T)$. The other case is true trivially; the O guaranteed by the theorem is the same O in the premise of the proof rule.

Several proof rules allow us to derive $\Gamma \Rightarrow \mathtt{sat}'(R,T)$ but trivially satisfy the theorem by the inductive hypothesis. Rule 2(left and right), Rule 3(left), Rule 4(left and right), Rule 5(left), Rule 6(left), Rule 7, Rule 9, Rule 11, and Rule 19 are of this form.

Consider Rule 14. By the inductive hypothesis, there exists a proof object O' such that $R \vdash_{LF} O' : \{x : T\}B$. By the second antecedent of the proof rule, we know $R \vdash_{LF} O : T$. The LF term $(O'\ O)$ thus has LF type $B[O/x]$, which satisfies the theorem.

Finally consider Rule 17. In this case the inductive hypothesis for the first antecedent of the proof rule says there is a proof object O' such that $R \vdash_{LF} O' : \{x : T\}B$. Applying the inductive hypothesis to the second antecedent yields an object O such that $R \vdash_{LF} O : T$. The LF object $(O'\ O)$ thus has LF type $B[O/x]$, and since x does not occur in B, $B[O/x] = B$. Thus the LF object $(O'\ O)$ satisfies the theorem.

Theorem 2 *Given finite starting sets U of objects of simple type and V_0 of property types, and given $V_i = V_{i-1} \cup \eta(U, V_{i-1})$, there exists a fixed point V_n such that $V_n = V_n \cup \eta(U, V_n)$.*

Proof. Let m be a function from property types to natural numbers that measures the arity of property types, defined as follows. For any predicate A, property type P, simple type T, and variable x, let $m(A) = 0$, $m(A \to P) = 1 + m(P)$, and $m(\{x : T\}\ P) = 1 + m(P)$. Let C_i be the maximum value m takes on property types in the set of types for V_i. Given a property type $P \in V_i$, new property types in $\eta(U, V_i)$ derived from P have arity $m(P) - 1$. Thus the maximum value m takes on the types of $\eta(U, V_i)$ is at most C_i. It follows that for all $i \in \mathbb{N}$, $C_i \leq C_0$.

Let P_C be the set of constants used as predicate names for predicates in the types of V_0. Because these are constant, at each stage constructing $\eta(U, V_i)$ from V_i can never introduce new predicate name constants in the types. So every predicate at every stage must use a predicate name constant from P_C.

Up to α-conversion, there are finitely many property types P with $m(P) \leq C_0$ with objects of simple type from U. A property type with $m(P) \leq C_0$ can have at most C_0 variables. Let $W = \{v_1, \ldots, v_{C_0}\}$ be canonical names for the possible variables. Then any predicate must be made up of a predicate name constant from P_C followed by constants from U or variables from W. Both U and W are finite, so there are a finite number of such predicates. Finally, constructing property types from these predicates allows finitely many choices C_0 times, so there are only finitely many property types P with $m(P) \leq C_0$. Let this number be B.

There are at most B distinct property types that may be in V_i, so $|V_i| \leq B$. The sequence V_i is monotonically increasing and bounded in size by B, so it must have a limit point V_n.

B. Example of trust in the λ-calculus

This example shows how one can use trust annotations and a type system to track trust in the λ-calculus. Here we use BLF to implement the proof system and first example from the work of Ørbæk and Palsberg [23].

In our example the code consumer is a company that wants to deploy a web server. The web server comes from a new start-up company (with key `wsc`) and is supposed to be ten times faster than standard web servers. Since the code consumer needs to process credit-card transactions online, she wants a high assurance that the web server operates securely. She hires an outside auditor (with key `auditor`) who verifies the trust assumptions in the web server code. Trust assumptions are places where the code would normally be distrusted, but is typecast into "trusted" because of knowledge outside the type system.

The base types for program data include `bool`, `unit`, `action`, and `request`. These types are annotated using superscripts with trust types *distrusted* and *trusted*, abbre-

viated *dis* and *tr*. For example, the type $\texttt{request}^{dis} \rightarrow \texttt{bool}^{tr}$ would be assigned to a function that takes untrusted requests and returns trusted booleans.

The following functions with the given annotated types are available to the web server.

$\texttt{perform_action} : \texttt{action}^{tr} \rightarrow \texttt{unit}^{tr}$

$\texttt{action_of_request} : \texttt{request}^{dis} \rightarrow \texttt{action}^{dis}$

$\texttt{verify} : \texttt{request}^{dis} \rightarrow \texttt{bool}^{tr}$

$\texttt{get_request} : \texttt{request}^{dis}$

$\texttt{error_action} : \texttt{action}^{tr}$

The web server code takes untrusted requests over the Internet, verifies they are legitimate requests, then performs an action associated with the request. Our example is slightly permuted from the original example of Ørbæk and Palsberg in order to show that the auditor does not need to examine all the code.

```
let x = get_request in
  perform_action
    (trust
      (if verify x then
         action_of_request x
       else
         error_action))
```

The code auditor looks at the following fragment of code:

```
if verify x then action_of_request x
else error_action
```

The auditor decides that since the untrusted action `x` is verified, the resulting return value of the code fragment may be trusted.

Here is the formalization of the typechecking rules in a ruleset `R_trust`.

```
nat : type.
zero : nat.
succ : nat -> nat.
expr : type.
ref : nat -> expr.
lam : expr -> expr.
app : expr -> expr -> expr.
trust : expr -> expr.
distrust : expr -> expr.
check : expr -> expr.

gt : nat -> nat -> type.
gt_zero : gt (succ N) zero.
gt_succ : gt A B -> gt (succ A) B.

baretype : type.
trusttype : type.
anntype : type.
annote : baretype -> trusttype ->
         anntype.
arrow : anntype -> anntype -> baretype.
trlte : trusttype -> trusttype -> type.
trlte_reflexive : trlte X X.
barelte : baretype -> baretype -> type.
barelte_reflexive : barelte X X.
annlte : anntype -> anntype -> type.
annlte_reflexive : annlte X X.
annlte_ax : barelte A B -> trlte X Y ->
            annlte (annote A X)
                   (annote B Y).
barelte_arrow : annlte Y B ->
                annlte A X ->
                barelte (arrow X Y)
                        (arrow A B).
join : trusttype -> trusttype ->
       trusttype -> type.
tr : trusttype.
dis : trusttype.
trlte_axiom : trlte tr dis.
jointr : join tr X X.
joindis : join dis X dis.

context : type.
emptycontext : context.
bnd : anntype -> context -> context.
hastype : context -> expr -> anntype ->
          type.
audit : expr -> type.
type_refz : hastype (bnd T C)
                    (ref zero)
                    T.
type_refn : hastype C (ref X) Z ->
            hastype (bnd T C)
                    (ref (succ X))
                    Z.
type_sub : hastype C E T ->
           annlte T T' ->
           hastype C E T'.
type_lam : hastype (bnd A C) X B ->
           hastype C (lam X)
             (annote (arrow A B) tr).
type_app : hastype C E1
             (annote (arrow A
               (annote T1 U)) W) ->
           hastype C E2 A ->
           join U W Z ->
           hastype C (app E1 E2)
                   (annote T1 Z).
type_trust : audit E -> hastype C E
                        (annote T U) ->
             hastype C (trust E)
               (annote T tr).
type_distrust : hastype C E
                  (annote T U) ->
                hastype C (distrust E)
                  (annote T dis).
type_check : hastype C E
               (annote T tr) ->
             hastype C (check E)
               (annote T tr).
```

```
bool : baretype.
action : baretype.
request : baretype.
```

Here are abbreviations to be syntactically substituted before typechecking.

```
error : anntype =
  annote action tr.
act_on : anntype =
  annote (arrow (annote request dis)
           (annote action dis)) tr.
verify : anntype =
  annote (arrow (annote request dis)
           (annote bool tr)) tr.
if : anntype =
  annote
    (arrow
      (annote bool tr)
      (annote
        (arrow
          (annote action dis)
          (annote
            (arrow
              (annote action dis)
              (annote action dis))
            tr))
        tr))
    tr.
context0 : context =
  (bnd error
     (bnd act_on
        (bnd verify
           (bnd if emptycontext)))).
```

Here are abbreviations that define the expression to be audited, and the final complete expression that must typecheck.

```
audit_expr : expr =
  (app
     (app
        (app
           (ref (succ (succ (succ
              (succ zero)))))
           (app
              (ref (succ (succ (succ
                zero))))
              (ref zero)))
        (app
           (ref (succ (succ zero)))
           (ref zero)))
     (ref (succ zero))).
full_expr : expr =
  lam (trust audit_expr).
```

The goal of the code producer is to prove that in the correct context, the web server takes untrusted requests and performs trusted actions. This prevents arbitrary requests from performing actions not authorized by the web server. Since the code needs to be audited first, the code producer proves that if the code is audited, then it typechecks and has the correct type.

The auditor audits the code, which is imported as:

```
use R_trust in
  auditor says
    believe(audit audit_expr).
end
```

The web server company provides a proof of:

```
(audit audit_expr) ->
  hastype context0 full_expr
    (annote (arrow
              (annote request dis)
              (annote action tr))
            tr)
```

The proof the web server company provides is:

```
[a:(audit audit_expr)]
   type_lam
   (type_trust
      a
      (type_app
         (type_app
            (type_app
               (type_refn
                  (type_refn
                    (type_refn
                      (type_refn
                         type_refz))))
               (type_app
                  (type_refn
                     (type_refn
                       (type_refn
                          type_refz)))
                  type_refz
                  jointr)
               jointr)
            (type_app
               (type_refn
                 (type_refn
                    type_refz))
               type_refz
               joindis)
            jointr)
         (type_sub
            (type_refn type_refz)
            (annlte_ax
               barelte_reflexive
               trlte_axiom))
         joindis)).
```

The code consumer combines the assertion from the auditor with the proof from `wsc` to yield a derivation that the web server code typechecks. The code consumer's policy states that any code which typechecks is safe to execute, so the web server code is safe to execute.