Secure Linking: a Framework for Trusted Software Components

Eunyoung Lee
Department of Computer Science
Princeton University
elee@cs.princeton.edu

Andrew W. Appel
Department of Computer Science
Princeton University
appel@cs.princeton.edu

Abstract

In linking together a software system from components in the presence of multiple versions, digital signatures, static type information, software fetched over networks, multiple vendors, local libraries, and so on, the policies guiding linking may be quite complex. We show how to describe such policies in a "linking logic" that is modular and provably sound. We show a prototype implementation, and we show that this logic is expressive and general enough to describe a real-world system: the Microsoft .NET "assembly" versioning system. The framework is general and expressive enough to represent other existing linking systems and to help different linking systems (and public key infrastructures) interoperate.

1 Introduction

Large software systems are often built from loosely-coupled subsystems. The relative independence of software components makes it possible for each component to be implemented separately and to communicate with other components only through its interface, regardless of its inner implementation. When a programmer uses a third-party software component as a building block of her system, she doesn't want the code she imports to break the whole system. She needs some methods guaranteeing that linking the foreign software component to her system is safe.

The most widely used methods for ensuring safe linking are type checking and code signing. Checking the type of the interfaces between two software components ensures that two components agree on the types they are using. Although type checking is quite strong and easy to use, it doesn't guarantee that the code will behave in an expected way. Code signing ensures that someone trustworthy trusts the code, but doesn't specify what properties of the code are certified by the signer: "never pops up misleading dialog box" or "immune to buffer overrun"?

To address this problem, stronger and more specific guarantees are needed. A code consumer may want to specify that a component must have certain properties to be linked safely and securely to its system. At the same time, the code consumer wants a code provider writing or supplying the component to prove it has the required properties.

We have developed a logical framework for linking systems providing stronger support for system safety and security. It is based on Proof-Carrying Authentication (PCA), an authentication/authorization framework based on higher-order logic [2]. In our framework, a code consumer announces its linking policy to protect itself from malicious code from outside. The policy can include certain properties required by the code consumer for system safety, such as software component names, application-specific correctness properties, version information of software components, etc. To link and to execute a component in the system of a code consumer, the provider of the component should submit a proof that the component has the properties specified in the code consumer's linking policy. The proof is formed by the basic logic and inference rules of the framework. After being submitted, the proof is checked by a small trusted proof checker in the code consumer, and if verified, the component is allowed to be linked to other components in the code consumer.

In this paper, we present the main design concepts of our proposed linking framework, and describe how they are represented in the underlying linking logic. We then show that the linking logic of the framework is general and powerful enough to give a formal description to other real linking systems by encoding the linking system of .NET in our logic, and discuss what we learned while formulating the .NET linking system.

Policy examples. One could use our framework, in building a system from components, to specify and enforce policies such as,

- `Game` imports (links to) `GUI` of version 1.3.
- `Compiler` links to any version of `SymbolTable` that has `efficient_lookup` property as certified

by `underwriters_laboratories`.

- `Compiler` links only to the particular implementation of `SymbolTable` whose machine code hashes to 8327518932.
- For version 1.6 through 1.9 of `GUI` it is OK to substitute version 2.4 (in .NET this is called a version redirection).
- All modules imported by `Game` must have been mechanically inspected by a virus detector.
- Certificate Authority `Alice` can vouch for the public key of the virus detector.
- `Game` links only to versions of `GUI` that support any superset of a particular COM interface.
- Untrusted modules must have been checked by a bytecode verifier to assure that they respect their interfaces.

The connection between the code, the property, and the property certifying agent is made by using cryptographic hashing and public-key encryption, as appropriate.

2 Related work

Component models. Restricting the communication between components is necessary to protect each component from misuse or malicious attack by other components. Traditionally, each program protects itself while communicating with other programs through abstract data types (ADT) or information hiding.

Object-based frameworks such as COM or languages such as Java provide component models based on *objects*. They permit access control at the level of class, method, or data field. The Compilation Manager (CM) of SML/NJ [6], enables access control of software components at a larger scale: CM makes it possible to describe a hierarchy of modules so that entire submodules are hidden from view except within their own component-group, and to export interfaces at the group level.

Bauer, Appel, and Felten extended the Java package mechanism and developed a linking system supporting hierarchical modularity similar CM [4]; Reid et al. propose a component model (called *units*) for the C language [14].

Bytecode verification. Java introduced the idea of static type-checking of program code by the "code consumer" just prior to linking; Necula [11] moved the checking from bytecode to machine code in *Proof-Carrying Code*. Devanbu et al. [7] suggested that if the proof-checking is too expensive to do on the "code consumer's" machine, then it could be done by a trusted coprocessor (on the code producer) that would produce a signed certificate.

.NET framework. The .NET framework is a computing platform developed by Microsoft targeting the highly distributed environment of the Internet [13].

The Common Language Runtime (CLR) of the .NET framework uses Java-style bytecode verification. It also has a new configuration-management mechanism called the *assembly*, which provides version-number information, as well as information about what version-numbers of other components are required for linking with the given assembly.

Tying it all together. There are so many mechanisms to choose from: class and method-level access control, supermodule-level access control, bytecode verification, delegation of verification to trusted hardware or some other authority, code auditing and certification by vendors or by independent third parties. All of these mechanisms may be useful. We have developed a framework in which system-builders can specify how the mechanisms should be combined; our implementation enforces such specifications.

PCA. Our framework is built on the PCA logic introduced by Appel and Felten [2]. Proof-Carrying Authentication (PCA) is a distributed authentication/authorization framework based the proof-carrying mechanism, originally introduced for Proof-Carrying Code by Necula [11]. PCA is different from previously existing authentication frameworks in two ways: it uses a higher-order logic that makes PCA more general and more flexible, and a code consumer need not execute a complicated decision procedure to grant the client's request. A code provider is responsible for proving her capability of access.

Authentication frameworks and protocols have been described using formal logic [1], for example in the Taos distributed operating system [15]. Taos has a logic of authentication on top of propositional calculus, which are proved to be sound. Wobber et al. [15] chose to implement only a decidable subset of their authentication logic since they want the decision procedure for granting a request to be decidable. Decidable logics are weaker than general logics, so this makes the authentication logic less flexible. To make an application-specific inference logic, some application-specific rules are added to a given set of basic inference rules and the soundness of the whole logic must be proved again.

PCA gains more flexibility by allowing quantification over predicates. Therefore the authentication framework has only one set of inference rules and all application-specific rules are proved as lemmas. In other words, for an application-specific security policy, users can define operators on top of the basic logic of PCA.

Since all the application-specific logics are expressed using the same general inference rules, they can interoperate

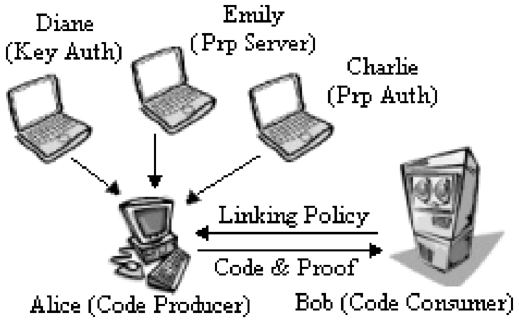

Figure 1. An example of secure linking

with each other easily. This makes PCA more general than other previous authentication logics. However, finding a proof for a request is not always possible because higher-order logic is not decidable. To get around this problem, PCA puts the burden of constructing proofs on the client and on the contrary the server simply checks that proof. This is in analogy with proof-carrying code [11]. Even in an undecidable logic, proof checking (not proving!) can be simple and efficient. Bauer, Schneider, and Felten developed an infrastructure for distributed authorization based on the ideas of PCA [5].

3 Example

Suppose that a principal Alice has a component named `compiler`. Alice could be a programmer who wrote the component, or she could be a customer who bought the component from a third-party developer. Now, Alice wants to send the component `compiler` to another principal named Bob, and Bob wants to execute `compiler` after linking it to other components in his system. Because Bob wants to keep his system secure and cannot trust the safety of the component Alice provides, he requires Alice to prove that her software component is safe enough to be linked to other components. To protect his system from all untrusted components from outside, Bob tells his linking policy to Alice. The linking policy usually consists of three parts: the description of software components Bob provides (call it a library), a list of useful properties Bob requires for outside software components, and the names of authorities trusted by Bob.

Properties. A code consumer requires a software component to have some pre-announced properties in order for the component to be linked to other components in his system. A *property* of a component is an assertion of expected behavior from the component. There are many useful properties which help protect systems from malicious outside codes, such as "this software component is type-checked," "this software component never accesses outside of the memory which is assigned to it," "this software component doesn't read any information from or write any information to the file system," or "this software component doesn't produce any arithmetic overflow or underflow." In our example, suppose that Bob requires that every component from outside should have a property called `prp_type_checked`, that means that Bob will allow only a type-checked component to be linked to his system. Now Alice must prove her component `compiler` has the property `prp_type_checked` in order to link the component to Bob's system.

Property authorities. Some properties, like the property of being type checked, can be guaranteed by a trusted compiler, while others cannot be proven easily. But these properties may be accepted as true if a software component has assurances made by trusted third-party authorities. The trusted authorities can make assurances resulting from a software audit or some other verification processes for software engineering. Such assurances are usually encoded as digitally signed statements. We call those authorities *property authorities*. After verifying the digital signatures on the statements, the property statements made by trusted property authorities are accepted as true and the components of those statements are considered to have the properties in those statements. In our example, as shown in Figure 1, Charlie is a property authority who examines a software component and determines if the component has the property `prp_type_checked`. Since Bob announced that he trusts Charlie as a property authority for `prp_type_checked`, Alice must get a digitally signed assurance from Charlie that the component `compiler` has the property `prp_type_checked`.

Key authorities. Since all certificates from property authorities come with digital signatures, a code consumer must know what the signers' public keys are in order to check the validity of the digital signatures. The code consumer must at least know who is the authority providing the right public keys for verifying the signatures and what her public key is. These authorities are called *key authorities* (also known as certificate authorities). Key authorities are responsible for guaranteeing the bindings between a principal's name and a public key. Key certificates signed by a key authority are verified with her already known public key. Hence, it is not unusual for the key certificates to form a chain of trust. Diane in Figure 1, is a key authority in our example. When Bob gets a digitally signed property certificate from Charlie, Bob doesn't need to know in advance the public key of Charlie to verify the signature of that certificate. Instead Bob asks Diane for Charlie's key certificate (or Bob might ask Alice to get Charlie's key certificate from Diane to complete her proof); he gets Charlie's public key after verifying the key certificate with Diane's public key.

Property servers. Just as Bob doesn't have to know all the public keys of principals, he doesn't have to know all the bindings between properties and property authorities. Instead Bob announces that he trusts a principal as someone who will let him know the property-authority bindings. Therefore, Bob doesn't enumerate the names of property authorities for every required property in his linking policy, and it relieves him from modifying and re-announcing the linking policy whenever a property-authority binding changes. In Figure 1, Bob announces that he trusts Emily as a property server. So Alice should consult Emily to find out from whom she can get property statements for the component `compiler`.

Library. Although it is possible to write a software component to be self-contained, it is very natural for a software component to use already existing software components by importing them. A code consumer announces what components it has and what properties are exported by each of those components. At the same time, a component from a code provider declares what components it imports and what properties are required for each of them in its component description. For a foreign component to be linked safely, a secure linker checks whether or not the library of the code consumer provides all the software components that are required by the foreign component. Suppose that the component `compiler` in our example uses a hash table during its computation and imports a component called `hashTable` with a property `prp_efficient_search`. Bob's library could have several different components named `hashTable`. These are different from each other in terms of the properties they export. Because the component `compiler` requires that a component `hashTable` should have `prp_efficient_search`, only a component `hashTable` exporting the property `prp_efficient_search` would be linked to the component `compiler`.

Linking decision. To decide to link a component coming from outside to other components in the system, a code consumer must verify whether or not the component provides all the required properties. For example, Bob checks the proof from Alice with the certificates by using a trusted proof checker, and links the component `compiler` to other components in his system if the proof is valid; otherwise he rejects it. Since the certificates Alice provides are digitally signed, all signatures are verified during the proof-checking time.

```
⟨componentDsc⟩
  ⟨name⟩ compiler ⟨/name⟩
  ⟨modules⟩
    ⟨item hash = "194CA77319" ⟩ compiler.class ⟨/item⟩
    ⟨item hash = "EF41900142" ⟩ regAlloc.class ⟨/item⟩
  ⟨/modules⟩
  ⟨exports⟩
    ⟨type⟩
      ⟨item⟩ class compiler⟨/item ⟩
      ⟨item⟩ interface regAlloc⟨/item⟩
    ⟨/type⟩
    ⟨property⟩ ⟨item⟩ prp_type_safety ⟨/item⟩
    ⟨/property⟩ ⟨/exports⟩
  ⟨imports⟩
    ⟨component⟩
      ⟨name⟩ hashTable ⟨/name⟩
      ⟨required⟩
        ⟨type⟩ ⟨item⟩ class hashtable ⟨/item⟩ ⟨/type⟩
        ⟨property⟩
          ⟨item⟩ prp_type_safety ⟨/item⟩
          ⟨item⟩ prp_efficient_search ⟨/item⟩
        ⟨/property⟩ ⟨/required⟩ ⟨/component⟩ ⟨/imports⟩
⟨/componentDsc⟩
```

Figure 2. A component description in XML

In what follows, we will explain our framework using the example above. Our framework is independent of programming languages or programming environments; thus, the explanation of the framework is language-neutral.

4 Models for linking

Our framework has two conceptual models for linking: one for describing components and their properties, and one for describing a code consumer's linking policies. Each model has its own descriptive language adopting the XML syntax. We developed XML parsers for each language, producing our linking logic from XML description files. The resulting component description of the previous example is shown in Figure 2.

In this section, we will explain the models of components and linking policies in our framework. The complete syntax and semantics of the two description languages is presented in our companion technical report [9].

Going back to the example in the previous section, Alice must describe the software component `compiler` to Bob's linking system. A component description in our framework consists of four parts: a component name, modules, exports, and imports. The *component name* is a local identifier of convenience for the component. The *module* part is a set of code files which implement the component. The logic represents each code file by its file name and cryptographic hash code; a secure linker makes sure (by checking hash codes) that the files linked are the ones that the policy accepted. The *export* part of a component description specifies what should be visible outside of the component. Usually the identifiers with type information have been exported from software components (e.g., class and method names). In our framework, we allow components to export properties as well as class and method identifiers. The *import* part of a component description shows what other components it depends on. An import request of a component consists of a component name and some required properties. A secure linker locates an imported component and checks that it exports all the required properties of an import request.

By allowing the export and import of properties as well as class and method identifiers, our framework gives a linker more information than types and enables component composition to be safer.

Linking policies are set by system administrators (or component integrators) and specify what software components may be linked together. The user interface language provides a simple and convenient way of stating linking policies: users can specify library components provided by a system (usually a system of a code consumer), required properties from foreign software components, the names of trusted key authorities, and the names of trusted property servers.

Separating linking policies from the framework gives more ability to code consumers to express their linking policies, and it makes our framework more general and flexible than frameworks with fixed linking policies.

5 Linking logic

The linking logic of the secure linking framework is a higher-order logic defined on top of Proof-Carrying Authentication (PCA) logic [2]; therefore the semantics of each operator are expressed in terms of the underlying PCA logic, and inference rules using operators are then proved as lemmas. In this section, we explain how we translate the software component description and linking policy from the user interface languages into the linking logic, and how we represent the linking decision procedure in the linking logic.

Soundness. A logic comprises *operators* (such as *keybind* in an authentication logic) and *inference rules*. The inference rules can be used to prove theorems; in our application the linker demands the proof of a linking theorem before running a software system built from the linked components. It is desirable to prove the logic *sound*, i.e., that untrue formulas cannot be proved.

Soundness is typically proved by induction over all proofs that can be built from a given set of inference rules. PCA takes a different approach, however: each application-specific operator (such as *keybind*) is defined in terms of the underlying operators of higher-order logic. Each inference rule is proved as a theorem of higher-order logic. Because each rule is proved sound independent of all the others, the system is more modular: it's easier to add new application-specific operators and rules as needed. This is important when making two systems (such as public-key infrastructure and safe linking, or two public-key infrastructures) interoperate. We have used PCA to prove the soundness of our logic.

$$\frac{prp\_kind(p, prp\_component\_name)}{rq\_component\_name\_exists(p)}\ name\_exists$$

$$\frac{\begin{array}{l} p\_name\_request = mk\_rq\_component\_name(p) \\ prp\_eq(p,q) \\ prp\_kind(q, prp\_component\_name) \end{array}}{p\_name\_request(q)}\ name\_match$$

Figure 3. Rules for component names

Representing properties. In our framework, a code consumer announces in advance which properties it requires a foreign component to have. Code providers are responsible for proving that their components have the properties required by the the code consumer. Therefore, an essential part of the logic is to check if two properties match each other.

In designing the logic for property matching, we had three purposes. We wanted: to easily check if two properties match each other, to make adding new properties simple, and to isolate the implementation details of *properties* from other parts of the linking logic.

We achieved the first goal by dividing property matching into two parts, *properties* and *property requests*. In our design, a property request is a predicate of type *prp_req* accepting an argument of type *property*; it returns true if the argument matches the request it implements, or returns false otherwise. By introducing a predicate type *prp_req* as well as the type *property*, we turn the procedure of checking property matching into simple evaluation of a predicate. Thus, the required properties in the linking policy of a code consumer are encoded in the form of type *prp_req*, and a code producer proves that the set of its exported properties includes all of those required to make the encoded property requests true.

As enumerated in Section 3, there exist many different kinds of properties. Therefore it is useful to make it simple to add a new kind of properties and property requests to the framework without changing other already existing properties in the linking logic. Our framework and its logic is designed for making this procedure as easy and simple as possible. For example, Figure 3 shows the inference rules used for matching properties for component names. Two different property requests on component names are typically used in linking. A code consumer may require a component to have a specific name p, by building a "name request" using the predicate *mk_rq_component_name*(p). An inference rule called *name_match* specifies how a name request can be satisfied (inference rules have names for convenience in referring to them in proofs). If a name request *p_name_request* is built from p using the the predicate, and if there is some name q equal to p, then *p_name_request* can be satisfied by q.

Another kind of request is simpler: The code consumer may require a component to have some name – any name. The inference rule *name_exists* can be used to build proofs of the *rq_component_name_exists* predicate.

An important design goal is to separate property definitions *(property)* and property requests *(prp_req)* from the other part of the linking logic. By separating these concerns, we increased the scalability and flexibility of the framework. During a linking procedure, property matching happens in two places: a secure linker checks if a foreign software component exports all the properties required by a code consumer, and if the library of the code consumer satisfies the foreign component's import requests. The predicates *export_required_prps* and *satisfy_import_req* address these requirements respectively in the linking logic.

The semantics of these decision predicates depend on a predicate *has_property*. Therefore, the predicate *has_property* is the only part concerned with the implementation of property matching. Furthermore, even the predicate *has_property* checks if a property satisfies a given property request. In other words, the predicate *has_property* depends only on the abstract part of types *property* and *prp_req*, not on the details of their implementation.

Component description. A component description is usually turned into formulas in the linking logic after checking the hash codes of binary modules. Digitally signed statements of certificates from key authorities, property authorities, or property servers are converted into axioms in the logic after verifying their signatures.

A component description is encoded in the form of a set of properties and a list of sets of property requests. The name of a component, the set of exported type identifiers, and the set of exported properties are all treated as properties and encoded into instances of type *property*. These make up the export part of the component.

The list of sets of property requests corresponds to the import part of a component. Each set of property requests stands for one imported component and includes the predicates for useful properties such as component names. Since a software component usually imports more than one components, the sets of property requests for imported components form a list. By using a list rather than using a set, we can handle some cases in which the order of importing the components is critical in linking decision.

Component descriptions can be combined by using the formula constructor *cdsc_combine*, which accepts two terms of type *component_dsc* and returns a term of type *component_dsc*. In addition, the linking logic provides a binary relation *sub_cdsc*, which determines one argument is a subcomponent of the other.

It is very useful to make it possible to combine component descriptions, especially when considering digitally-signed certificates from property authorities. When signing, it is reasonable for a property authority to want to sign only on the properties he can guarantee, rather than sign on all the properties a component description exports. For example, a trusted compiler can guarantee that modules with a given component description are type-safe, but doesn't want to, or is not able to, guarantee any other properties. After collecting component descriptions assured by property authorities, a code producer combines the small descriptions, and builds a complete component description. It frees the property authorities from a burden of assuring more properties of a component description than they want to.

Linking policy description. A linking policy is translated into two forms in the linking logic: axioms and formulas. The name bindings of property servers and of key authorities are turned into axioms.

The library components are encoded in formulas. A library component's name, its exported type identifiers, and its exported properties are encoded as a set of properties. Since a code consumer usually provides more than one library component, they are put into a list, each element of which is encoded as a set of properties, as explained.

At the same time, linking policies specify the properties required by the code consumer. Each required property is translated into a corresponding predicate of type *prp_req*. Together they form a set. For example, if a code consumer requires every foreign component to have a name, it is turned into the predicate *rq_component_name_exists* of Figure 3.

Making a linking decision. To link a component to other components of a code consumer, a code producer must show that her component exports all the properties required by the code consumer. This can be done by showing that a set of modules and its component description satisfy the predicate *ok_to_link* with the linking policy specified by the code consumer. The semantics of predicate *ok_to_link* shows what steps a secure linker should follow to make a linking decision. The following is the inference rule for a linking decision.

$$\frac{\begin{array}{l} \mathit{signed\_component\_dsc}(m, dsc, prqset) \\ \mathit{provides\_enough\_prps}(dsc, lib, libdsc) \\ \mathit{exports\_required\_prps}(prqset, dsc) \end{array}}{\mathit{ok\_to\_link}(m, dsc, lib, libdsc, prqset)}\ \mathit{ok\_to\_link\_i}$$

First, the linker examines if the given logical description `dsc` of a component really represents the given set of modules `m`; if so, the code producer can prove that the predicate *signed_component_dsc* holds. Second, the linker examines if the component can obtain all the imported components from the library `lib` and `libdsc` of the code consumer; if so, the code producer can prove that the predicate *provides_enough_prps* holds. Last, the linker examines if the component description exports all the required properties `prqset`; if so, the code producer can prove that the predicate *exports_required_prps* holds.

If the component description and modules satisfy the above three conditions, in other words, if the code producer can prove that those three predicates hold, linking is allowed; otherwise, it is denied. All the decision steps are addressed in the linking logic as operators and lemmas on top of PCA logic, and all the lemmas are proved.

Given a proof from a code producer, a code consumer must be able to verify the validity of the proof. Our framework is built on the Twelf logical framework [12]. The Twelf system is one of the implementations of the logical framework LF [8], which allows the specification of logics. Since our linking logic is written on top of PCA, an object logic of LF, every term in our linking logic boils down to a term in the underlying LF logic. Therefore, the proof provided by a code producer is encoded as an LF term. The type of the term is the statement of the proof; the body of the term is the proof's derivation.

By the Curry-Howard isomorphism, checking the correctness of deriving the term that represents a proof is equivalent to type checking the term. If the term is well typed, then the derivation is correct; hence, a code producer has succeeded in proving the proposition. If the proof of a code producer is checked, a secure linker of our framework will allow the component with the proof to be linked.

6 Tactical prover

We have developed a tactical prover for our linking logic. The prover is a logic program running on the Twelf logical framework [12]. The goal to be proved is encoded as the statement of a theorem, and axioms that are likely to be helpful in proving the theorem are added as assumptions. The prover generates a derivation of the theorem; this is the proof that a code provider must send to a code consumer.

Our tactical prover consists of 30 tacticals and 58 tactics: tactics, reducing goals to subgoals, and tacticals, providing primitives for combining tactics into larger ones that can give multiple proof-steps. By showing the soundness of logic, it is guaranteed that every formula that is provable (or derivable) by the prover (consisting of axioms and inference rules) is true in the logic.

```
<configuration>
 <runtime>
  <assemblyBinding
   xmlns="run:schemas-microsoft-com:asm.v1">
   <dependentAssembly>
    <assemblyIdentity name="hashTable"/>
     <bindingRedirect
      oldVersion = "1.0.0.0 - 1.9.9.0"
      newVersion = "2.0.0.0"/>
   </dependentAssembly>
  </assemblyBinding>
 </runtime>
</configuration>
```

Figure 4. A .NET configuration file

Appel and Felty [3] showed that using a dependently typed programming language can yield a partial correctness guarantee for a theorem prover: if it type-checks, then any proof (or subproof) that it builds will be valid. Twelf is such a higher-order dependently typed logic programming language, and our prover is easily seen to be sound by the method of Appel and Felty.

Although a dependent type system is very useful for showing the soundness of a prover, it doesn't guarantee that the prover is complete.

We also proved that our tactical prover always terminates and that it is complete. By proving the termination of the prover, we can guarantee that it halts, regardless of the input. By proving the completeness of the prover, we can make sure that it finds a derivation for every true formula formed by the linking logic. We will outline the proof in this section, and the complete proof can be found at [9].

To prove the completeness of the prover, we related each formula constructor in the linking logic to one tactical in the prover. That means, for a formula formed by a given constructor, the formula is derivable only by the related tactical in the prover. For example, the formula constructor *ok_to_link* is related to the tactical *findproof*. Therefore, any formula formed by *ok_to_link* is derivable only by using the tactical *findproof* in the prover. Since the goal of the tactical prover is to find proofs of formulas built from *ok_to_link*, we can prove that the tactical prover is complete and always terminates by showing that the tactical *findproof* always terminates and finds a proof of a true formula.

7 Case study: .NET framework

In this section, we will show our linking logic is general and expressive enough to address the linking decision procedure of the .NET framework.

7.1 Versioning

Version redirection in .NET. The .NET framework treats an assembly's version number as part of the assembly's identity; this enables assemblies of the same name to coexist within one system, and gives more control to developers or system administrators in the linking time. The runtime of the .NET framework allows developers or system administrators to specify the version of an assembly to be used for linking, and to use a different version of an assembly of the same name in the linking time. In order to redirect binding of assemblies users can specify it in configuration files in different levels. Application configuration files, machine configuration files, and publisher policy files are used to redirect one version of an assembly to another. An example of a configuration file redirecting assembly versions is shown in Figure 4. The information for each assembly that developers want to redirect is put inside the ⟨dependentAssembly⟩ tag and the information identifying the assembly is inside the ⟨assemblyIdentity⟩ tag.

Usually the original version of an assembly and the versions of dependent assemblies are recorded automatically in the assembly's manifest by programming tools or compilers supporting the .NET framework. A linker makes a decision of which version of an assembly is to be linked with the following steps: first, the linker checks the original assembly reference to determine what version is originally used. Second, it checks all available configuration files to find applicable redirection requests in a sequence of machine configuration files, publisher policy files and application configuration files. Last, it determines the correct assembly version that should be linked to the calling assembly, from the information of the original assembly reference and any redirection specified in the configuration files.

Translation into our linking logic. In Section 5, we explained how to represent some entities such as component names as properties and property requests in our logic. In the same way, version information and redirection requests are coded as *properties* and *property requests*.

Version information is of type *version*, alias of type *property*, and built by using the formula constructor *mk_prp_version*. The constructor *mk_prp_version* takes 4 numbers as arguments, each of which stands for a major version number, a minor version number, a build number and a revision number respectively.

A redirection request for an assembly is a predicate of type *ver_req*, alias of type *prp_req*. It takes an argument of type *version* and returns true if the given version information satisfies the redirection request it implements. The version redirection information in configuration files has the type *ver_policy*, consisting of two parts, the affected old version (either a specific version or a range of versions) and a new target version.

$$\frac{\begin{array}{l}\neg isempty(vPolicy)\\ inrange(vPolicy.range, originalVer)\end{array}}{ver\_policy\_effective(vPolicy, orginalVer)}\ vp\_effective$$

$$\frac{\begin{array}{l}ver\_policy\_effective(vrq.mch, vrq.org)\\ version\_match\_policy(vrq.mch, v)\end{array}}{vrq(v)}\ machine\_redir$$

$$\frac{\begin{array}{l}\neg ver\_policy\_effective(vrq.mch, vrq.org)\\ ver\_policy\_effective(vrq.pub, vrq.org)\\ version\_match\_policy(vrq.pub, v)\end{array}}{vrq(v)}\ publisher\_redir$$

$$\frac{\begin{array}{l}\neg ver\_policy\_effective(vrq.mch, vrq.org)\\ \neg ver\_policy\_effective(vrq.pub, vrq.org)\\ ver\_policy\_effective(vrq.app, vrq.org)\\ version\_match\_policy(vrq.app, v)\end{array}}{vrq(v)}\ app\_redir$$

$$\frac{\begin{array}{l}\neg ver\_policy\_effective(vrq.mch, vrq.org)\\ \neg ver\_policy\_effective(vrq.pub, vrq.org)\\ \neg ver\_policy\_effective(vrq.app, vrq.org)\\ version\_match\_simple(vrq.org, v)\end{array}}{vrq(v)}\ no\_redir$$

Figure 5. Rules for version redirection

The linking decision procedure of the .NET framework is translated into a set of lemmas in our linking logic, shown in Figure 5.

Two versions may match if they are identical *(ver_match_simple)*, or through a redirection *(ver_match_policy)*. The redirection may come from local configuration files *(machine_redir),* from the software developer *(publisher_redir),* or from the component integrator *(app_redir)*.

A redirection request is effective, if the version of an original assembly to be linked is within the affected old version range specified in the redirection request. If so, the predicate *ver_policy_effective* holds.

The inference rule *machine_redir* shows the case when there exists a version redirection request `vreq`, and the version policy field `mch` of `vreq` is effective. If the version `v` of a target assembly matches the new version in the policy `mch`, then the version `v` satisfies the version request `vreq`, and the assembly of version `v` is used in the later phase of linking.

Other inference rules (*publisher_redir, app_redir,* and *no_redir*) show that local redirections override publisher redirections, and so on.

7.2 Strong naming

The runtime system of .NET requires that every assembly has a strong name [10]. A strong name of an assembly consists of the assembly's name, its version number, its culture information (such as languages), plus a public key and a digital signature. This information is stored in the assembly's manifest.

In order to guarantee the integrity of an assembly, a code producer is recommended to sign the assembly. A code producer can sign an assembly in two different ways: with a strong name or with key certificates obtained from third-party key authorities. Signing an assembly with a public key adds the encrypted public key and the resulting signature to its assembly manifest. Then, the .NET runtime verifies the digital signature of an assembly using the public key in its assembly manifest.

In formalizing the strong-naming feature of .NET, we were unable to prove a standard theorem in our system: that every public key is certified by at least one authority. It turns out that .NET's strong names use keys that need not be certified. At first glance, this appears to be a security hole, but in fact it is simply a harmless misapplication of public-key encryption. (However, it is a latent weakness if some future user could be misled into using this signature as a certificate of some property!)

Signing with a self-announcing public key doesn't provide more trust than a hash verification does. Verifying a digital signature with a public key in an assembly manifest only guarantees that the assembly has not been tampered after being signed. Without key certificates from trusted third-party key authorities, a digital signature on an assembly cannot give any assurance about the source of the assembly. It is exactly as strong as verifying a hash code of an assembly manifest because it is (assumed) impossible to change the content of data without changing the hash code of the data calculated by a cryptographic hash function (such as MD5 or SHA-1).

Digital signing is more complicated than hash code verification, and usually operates with hash code verification. Therefore, signing an assembly without key certificates in .NET seems redundant. In expressing strong-named assemblies in our logic, we replace signing on an assembly with a self-announcing public key by simple hash code verification. Following is the inference rule of strong-named assemblies.

$$\frac{\begin{array}{l}\exists N.asm\_name(asm, N)\\ \exists V.asm\_version(asm, V)\\ \exists C.asm\_culture(asm, C)\\ \exists P.asm\_pubkey(asm, P)\\ \exists H.asm\_hash\_code(asm, H)\\ valid\_hash\_code(asm, H)\end{array}}{strong\_named\_asm(asm)}\ strong\_named$$

An assembly `asm` has a strong name if there exist an assembly's name, its version, its culture information and its public key, and if a hash code accompanied with the assembly is valid.

7.3 Key certificates

Code signing uses key certificates issued by trusted key authorities. Since different key authorities could use different formats for their certificates, it is necessary for a secure system to support as many formats as possible for interoperability.

The security model of the .NET framework supports several standard public key certificate formats including X.509. The key certificates are embedded in a specific location in an assembly manifest by a .NET-supporting compiler at compile time and then used by a code consumer to check the assembly's authenticity later.

The assembly-manifest format accommodates a specific set of key certificate formats. Tying public-key infrastructure so closely to version management results in a less flexible system than expected: future users with a different PKI will not be able to take advantage of code signing in .NET. Our linking logic can separate these issues in a more modular way.

To address principal-public key bindings, our linking framework has a built-in formula constructor *keybind* for translating various key certificate formats into a formula in the linking logic. The formula constructor *keybind* takes two arguments: the name of a principal, and its public key. The formula holds if and only if the second argument `pubkey` is the public key of the first argument `name`. The statement of `keybind(name, pubkey)` is made out of several key certificate formats by the trusted part of a code consumer. Since translating format-specific key certificates is not a part of the linking logic, a new key certificate format can be added to the linking framework later without the necessity of changing the linking logic. It increases the scalability of the linking framework.

Following is the complete rule for verifying digital signatures.

$$\frac{\begin{array}{l}key\_auth(ca)\\ keybind(ca, caKey)\\ signed(caKey, keybind(pname, pkey))\\ signed(pkey, stmt)\end{array}}{says(pname, stmt)}\ valid\_sig$$

This rule means that it is believed that the principal `pname` says `stmt` if `stmt` is signed with a key `pkey` and a key certificate, saying that the public key of the principal `pname` is `pkey`, is issued by a key trusted authority `ca`.

Having verified the digital signatures with this inference rule, none of the remaining part of linking procedures depends on the formulas built by *keybind* or *signed*. By introducing the format-neutral constructor *keybind* and letting only a small part of the logic use the signature-specific constructors, we separate the logic of verifying digital signatures from the rest of the linking logic, and make the framework work smoothly with different key certificates.

8 Conclusion

We have developed a framework for secure linking systems based on PCA. In this scheme, the burden of proving rights to access the shared resources of a code consumer is put on a code producer rather than on the code consumer, unlike in traditional distributed authentication frameworks.

In our framework, a code consumer announces its linking policy to protect its system from malicious code from outside. The policy can include properties, for example, class name, valid hash code of programs, version information, and so on, which the code consumer thinks important for system safety. To link a software component to other components in a code consumer and to execute it, a provider of the component should submit a proof that the component has the properties required by the code consumer.

The linking logic of our framework consists of basic formula constructors and inference rules on top of the PCA logic. Linking decision procedures of a code consumer, system-specific linking policies, and software component description of code producers are translated into the linking logic. A proof of secure linking is formed out of the linking logic, and checked by a trusted proof checker in a code consumer. If the accompanying proof is verified, a software component is allowed to be linked to other components in the system of the code consumer.

In addition, adopting the higher-order logic of PCA makes our linking logic general and flexible. Due to this expressiveness, it is possible to encode various security models into our logic, and to enable different security models to interoperate conveniently. We tested the expressiveness of our linking logic by encoding the linking procedures of the .NET framework. We showed how we formulate .NET's version redirection, strong-named assemblies, and digital signature verification on top of our linking logic.

Trying to give a formal description to a real-world system gives us insight into the system. In case of .NET, we found that its signing an assembly without key certificates is redundant and can be replaced by simple hash code verification. We also found that digital signature verification and its key certificate management would better be separated from other linking decision parts to enhance the scalability and interoperability of the .NET framework.

Using logic and formal methods has been helpful in two ways: formalization has helped us define abstractions in order to build a prototype language and system that would have been difficult to design otherwise; and in the process of formalizing an existing protocol (such as .NET) we find latent design flaws that might have otherwise gone undetected.

References

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

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[3] A. W. Appel and A. P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, Accepted for publication.

[4] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in java. Technical Report CS-TR-603-99, Department of Computer Science, Princeton University, July 1999.

[5] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[6] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21:812–846, 1999.

[7] P. T. Devanbu, P. W.-L. Fong, and S. G. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 1998 Internation Conference on Software Engineering*, pages 126–135, Los Alamitos, California, 1998.

[8] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:143–184, January 1993.

[9] E. Lee and A. W. Appel. Secure linking: a framework for trusted software components (extended version). Technical report, Department of Computer Science, Princeton University, To appear, 2002.

[10] Microsoft. *Inside the .NET framework*. http://msdn.microsoft.com/library/.

[11] G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, January 1997.

[12] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, July 1999.

[13] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2001.

[14] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the Usenix Conference on Operating System Design and Implementation*, pages 347–360, 2000.

[15] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.