Secure Linking: A Logical Framework for Policy-Enforced Component Composition

Eun-Young Lee

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 2004

Abstract

In this thesis I propose a flexible way of allowing software component users to specify their security policies, and to endow digitally signed certificates with more expressive power at link time. Secure Linking (SL) is more flexible than type-checking or other static checking mechanisms because it allows users the freedom to specify security policies at link time. In addition, Secure Linking is more expressive than simple digital signing by restricting the scope of guarantees made by digitally signed *certificates*. Secure Linking would not prevent bugs in a software component, but it gives people signing a software component finer-grain control of the meaning of their certificates.

The linking logic in the Secure Linking framework is based on Proof-Carrying Authentication (PCA), a distributed authentication/authorization framework. In Secure Linking, a code consumer establishes a 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, and so on. In order for a software component to run in the system of a code consumer, there must be a machine-checkable proof that the component has the properties specified in the code consumer's linking policy. This proof might be provided by the code provider, or might be produced by an untrusted proving algorithm that runs on the code consumer's machine. The proof is formed using the 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.

I demonstrate that the Secure Linking logic is flexible enough to interoperate with other application-specific and security-concerned logics. I show that the Secure Linking logic is expressive enough to describe real-world linking systems. I also describe a prototype implementation of Secure Linking for Java components.

Acknowledgments

Many people have helped me finish my dissertation at Princeton with their support, advice, and care. Most of all, a large part of this credit should go to my advisor Andrew Appel. He has been a really patient and wise advisor. I have always been impressed by his sharpness in understanding my ideas on research, and his dependable advice and encouragement. I feel lucky to be one of his students. I am grateful to my thesis readers Iliano Cervesato and Ed Felten for their very helpful and insightful comments. I would like to thank Melissa Lawson, Becci Davies, and the CS staff members to maintain a wonderful environment for my research.

The research in this dissertation was supported by NSF grants CCR-9870316 and CCR-9974553, DARPA grant F30602-99-1-0519, and the fellowship from the Korea Foundation for Advanced Studies.

The past five years have been the most challenging period for me. I moved to a foreign country, to a different culture, to a different academic environment to the one I used to. All this change was not always exciting, but the people I met here gave me power for me to go through it. I would also like to thank all the friends who have made my stay at Princeton so enjoyable.

I owe everything to my family. My parents' pride and hope in me are the persistent driving force for my struggles in carrer in life. My husband and best friend, Hyong-Soon helped me survive all those struggles with his care, sacrifice and confidence in me. The completion of this dissertation will be a contribution to the family which he and I are going to build together.

To my husband, Hyong-Soon

Contents

Chapter 1

Introduction

1.1 Code-Signing Protocols

Large software systems are often built from loosely coupled subsystems; a large system developed by a team is divided into small pieces, and each of the pieces is assigned to a developer in the team. Some pieces of the large system can be off-the-shelf components from third-party companies. When composing a system out of software components from various sources (developers in a project team or software component vendors), the system composer doesn't want an imported component to break the whole system. She needs some guarantee that linking the foreign software component to her system is safe.

Enriched content of web pages is also a source of foreign software components. Web pages consist of downloadable controls as well as traditional static data (such as text or images), and web surfers are asked to download the controls and to *plug in* the control components to their web browsers to fully enjoy the content of the pages they encounter. The surfers don't want the downloaded components to crash their systems or read their private or security-sensitive data.

Awareness of the importance of system security has increased, and diverse research areas in system security have been studied. This thesis focuses on preserving system security when composing components.

The one of the most popular ways to assure the system security at component level is *code signing*. Code signing enables software developers and/or venders to include information about themselves with their code. When users buy or download digitally signed software, they can be assured:

- that the software really comes from the publisher who signed it, and
- that the software has not been altered or corrupted since it was signed.

Users benefit from this software accountability because they know who published the software and that the code hasn't been tampered with. In the extreme case that software performs unacceptable or malicious activity on their computers, users can also pursue recourse against the publisher. This accountability and potential recourse serve as a strong deterrent to the distribution of harmful code.

Java of Sun Microsystems provides a mechanism with which users sign their own code [46, 45, 36, 47]. Sun Java Signing relies on industry-standard cryptographic technique such X.509 v3 Code Signing IDs and Public Key Cryptography Standards (PKCS) #7 (for encrypted key specification) and PKCS #10 (for certificate request formats) signature standards. A *JAR* file in Java is signed with the private key of the creator of the JAR file and the signature is verified by the recipient of the JAR file with the public key in the pair. *JAR signer* is a command line tool for signing and verifying the signature on JAR files, and *keystore* is a command line tool for managing key certificates.

Internet Explorer (IE) of Microsoft uses *Authenticode* technology to help users sign their own code [27, 28, 25]. Authenticode also identifies the publisher of signed software and verifies the code's integrity. Authenticode relies on digital certificates and is based on specifications that have been used in the industry for some time, including PKCS #7, PKCS #10, X.509 (for certificate specification), and Secure Hash Algorithm (SHA) and MD5 hash algorithms.

When Internet Explorer downloads potentially dangerous content, it checks to see whether the code is digitally signed by a trusted publisher. In addition to verifying digital signatures of code, Internet Explorer has another mechanism called the *kill bit*. The kill bit is a method by which an ActiveX control can be prevented from ever being invoked via Internet Explorer, even if it's present on the system or even if it has a valid digital signature.

Although it is popular in the industry, code signing has a critical weakness. It is not clear what a signature on code guarantee except for identifying the signer. The following real example shows that a security hole caused by a weak code signing protocol at link time might lead to a system vulnerability. In November 2002, Microsoft put an article on their Security Bulletin, warning about Window's system vulnerability due to a buffer overrun in some versions of Microsoft Data Access Components (MDAC), one of the ActiveX control components [29].

The article strongly recommends customers using Microsoft Windows, particularly those who operate web sites or browse the Internet to download a patch Microsoft developed and apply it immediately. The article also says that this vulnerability is critical because if the security hole is exploited by an attacker, it will result in running arbitrary code of the attacker's choice.

The solution of this critical problem looks quite straightforward: get a patch and apply it to the hole. The article, however, contains an important caveat associated with the patch Microsoft provides. The caveat says:

> ***What caveats are associated with the patch?***
>
> *Although the patch does address the vulnerability, there is a niche scenario through which a patched system could, under unusual conditions, be made vulnerable again. This scenario results because it is not possible to set the kill bit used by one of the vulnerable components.*

Why isn't it feasible to set the kill bit in this case?

The ActiveX control involved in these vulnerabilities is used in many applications and web pages to access data. Many applications, including third-party applications, contain hard-coded references to it; if the patch set the kill bit, the web pages would no longer function at all even with the new, corrected version. As a result, the patch updates the control to remove the vulnerabilities, but does not provide a brand-new control and set the kill bit on the old one.

A warning message is generated whenever there's an error associated with a digital signature or the signer isn't trusted. But in this case, the digital signature on the old version of the control is still valid, and the signer is Microsoft and depending on the circumstances, many users may have chosen to trust this particular Microsoft certificate (since it is used to sign a number of different Microsoft ActiveX controls). Because of this, many users would not see a warning message of any kind if the old control was re-introduced.

Why not revoke the certificate that was used to sign the control? The certificate that was used to sign the control is still valid?

The problem lies in the control, not the certificate. In addition, a number of controls have been signed using the same certificate, and revoking the certificate would cause all of them to become invalid.

What steps could I follow to prevent the control from being silently re-introduced onto my system?

The simplest way is to make sure you have no trusted publishers, including Microsoft. If you do that, any attempt by either a web page or an HTML mail to download an ActiveX control will generate a warning message. ... The best criterion to use is whether you trust the web site or the sender of the HTML mail. If you don't trust the web site offering the control, cancel the download.

Microsoft was faced with the two choices: setting a *kill bit* so that no browser would run the control—thereby disabling thousands of websites, even ones containing no security-critical data, or not setting the bit—thereby continuing to *endorse* the product and run the risk.

The old buggy control could be re-introduced by an attacker even after the patch was installed, because Microsoft signed a number of different Microsoft ActiveX controls including the buggy component with one certificate, and if a user chose to trust any of those ActiveX controls, it results in trusting the buggy control too.

Microsoft recommended that users who desired a secure system should remove *Microsoft* from Internet Explorer's Trusted Publisher List. Microsoft Internet Explorer provides a way of maintaining digital certificates for clients and certification authorities (CAs),[1] and a *Trusted Publisher List* is a list of publishers (people, companies, or organizations) whose content (webpages and/or controlling code) can be downloaded without user intervention. So, if a software company is listed in the Trusted Publisher List of a user, the user would not see a warning message of any kind when the browser downloads a piece of code from the software company.

After getting rid of *Microsoft* from the Trusted Publisher List, users will see a pop-up dialog box asking whether they are going to trust components to be downloaded which are signed by Microsoft. It might give a warning to users, but it doesn't solve the problem. If a user decides to download any of the ActiveX controls which share the same certificate with the badly coded ActiveX control, it will immediately open a security hole to attackers explained above.

This critical situation is caused because Microsoft's code-signing protocol is insufficiently expressive, and because Microsoft does not have a way of telling a component from another component which has the same name, but different version.

[1] A certification authority (CA) is a trusted organization that issues a public key certificate.

1.2 Goals

The example of Section 1.1 demonstrates the need for a more expressive way to guarantee security at link time. 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 (also called *sandboxing*) is quite strong and easy to use, it doesn't guarantee that the code will behave in an expected way. Different static checking mechanisms have been suggested to address specific security properties of programs: a security-sensitive type system [18, 32, 42], and wrappers which encapsulate untrusted programs and implement security-concerned properties [43, 44]. They give the users better facilities to address security properties than typical type-checking does, but they still suffer from a lack of expressiveness since their security or linking policies are fixed and encoded in their type- or logic systems.

Code signing ensures that someone trustworthy trusts the code. It, however, is not always enough to guarantee system safety since trusted software companies or software developers unintentionally make mistakes. Software signed by trusted companies can still leave security holes in users' systems.

We propose a flexible way of allowing users of software components to specify their security policies, and to endow digitally signed certificates with more expressive power at link time. Secure Linking is more flexible than type-checking by allowing informally specified properties, but by providing a way of reasoning about combinations of those properties in a formal way. In addition, Secure Linking is more expressive than simple digital signing by restricting the scope of guarantee made by digitally signed *certificates*. Secure Linking would not prevent bugs in a software component, but it would give people signing a software component finer-grain control of the meaning of their certificates.

Figure 1.1 compares the mechanisms for link-time security with respect to flexibility

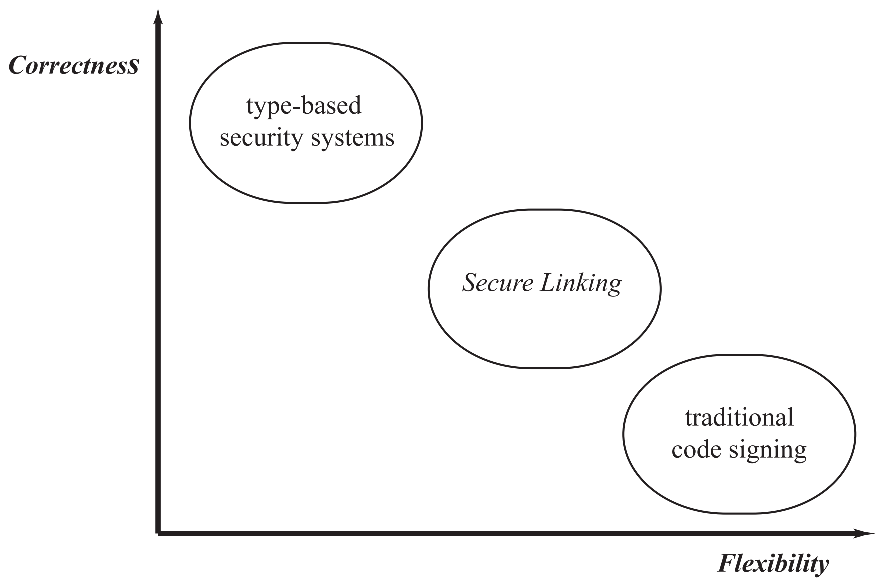

Figure 1.1: Different Static Checking Mechnisms

and correctness. The term *flexibility* is used for describing the degree of freedom establishing a security policy. Suppose that a user has a human-oriented security policy such as "a foreign component must not pop up a misleading dialog box." This policy can be guaranteed for a software component if someone investigates the component and makes an assurance. However, we don't know how to specify this property in formal logic. The term *correctness* is used for describing how complete an assurance made by a security system is. For example, if a type system makes an assurance that "this component never accesses the memory outside the memory protection barrier." after type checking, a user will know there will be no memory access violation while running the component.

Traditional code signing provides high flexibility, but weak correctness guarantees. For example, suppose that users purchase software components digitally signed by a big software vender. After verifying the digital signature, users get to know that the software component is coming from a trusted source and to expect the components *to behave in a good way without causing any harm to their system.* The property code signing could guarantee is implicit and very informal. The software components, however, could have bugs due to the mistakes of trusted software developers, and these bugs could result in security holes. Secure Linking lies between type-based approaches and traditional code signing. SL gains more flexibility than type-checking by providing a way of establishing users' own security policies, and gains more correctness than traditional code signing by providing a way of clarifying what is guaranteed by a signed certificate.

We have developed a logical framework for Secure Linking providing stronger support for system safety and security, and we have implemented a prototype system using the framework. The linking logic in the Secure Linking framework is based on Proof-Carrying Authentication (PCA), a distributed authentication/authorization framework [3]. In Secure Linking, a code consumer establishes a 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, and so on. In order for a software component to run in the system of a code consumer, there must be a machine-checkable proof that the component has the properties specified in the code consumer's linking policy. This proof might be provided by the code provider, or might be produced by an untrusted proving algorithm that runs on the code consumer's machine. The proof is formed using the 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.

Going back to the problem addressed in Section 1.1, a couple of remedies can be suggested for resolving this problem:

- Users adopt a linking system in which the identity of a software component includes its version information, and allows users to indicate which version of a given component should be run at an application level or a user's machine level.

- Otherwise, users can feel safe if a trusted authority, after inspecting a software component to be downloaded, makes an assurance that the software component is not at all using the badly coded ActiveX control, or an assurance that the component does not do any things harmful though it calls the buggy ActiveX control. That is, legitimate use of the buggy ActiveX control is still allowed as long as the call to the buggy ActiveX control does not exploit the known security hole maliciously.

Microsoft tackles the problem with the first approach; it introduced a version information as a part of component identity in the .NET framework. This approach could be overkill since it excludes all the code making unharmful use of the buggy ActiveX control. The second approach is more expressive than the first one in that it doesn't exclude the safe use of the old version, and more general in that assurances from trusted authorities can describe much more various kinds of software behavioral properties than simple version information. Secure Linking can express and enforce either of these policies.

I will explain the fundamental ideas of Secure Linking in Chapter 2 by describing how Secure Linking implements the second solution. Later I will also show that Secure Linking is expressive enough to implement the first solution of the example as a case study in Chapter 8. The premilinary result of this thesis can be found in [21], and an extended abstract of part of this thesis was published in [22].

1.3 Related Work

1.3.1 Proof-carrying authentication

Proof-Carrying Authentication (PCA) is a distributed authentication/authorization framework based on proof-carrying mechanism, introduced by Appel and Felten [3]. 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 server need not execute a complicated decision procedure to grant a client's request.

A client 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 [48]. Taos has a logic of authentication on top of propositional calculus, which are proved to be sound.

In Taos, a code consumer given an access request has to decide whether to grant the request. Wobber et al. [48] chose to implement only a decidable subset of their authentication logic since they want a decision procedure (i.e. a finite algorithm) to be able to decide whether to grant a request. Decidable logics are weaker than general logics, so this makes the authentication logic less flexible. If a user wants to extend the Taos' authentication logic for his specific system, some application-specific rules should be added to a given set of basic inference rules. But it makes the soundness proof of the Taos' logic invalid, therefore the soundness of the extended logic should 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. To describe an application-specific security policy, a programmer only needs to pick an application-specific decidable subset of the underlying general logic of PCA.

Since all the application-specific logics are expressed using the same general inference rules, they can interoperate with each other easily. This makes PCA more general than other previous authentication logic. 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 [31]. 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 [8].

1.3.2 Component models

A fundamental principle of software engineering is to divide a large-scale program into relatively independent and small subprograms. With this strategy, communication between the subprograms and seamless integration of them becomes important in order to protect each subprogram from malicious attack or inadvertent misuse of other subprograms, and eventually in order to make the large program work. Traditionally, each program protects itself while communicating with other programs through abstract data types (ADT) or information hiding.

Some languages, for example Standard ML and its associated Compilation Manager (CM) [9], support more facilities than simple ADTs by making it possible to structure modules hierarchically. Rather than having a flat and simple space of modules, CM makes it possible to build a hierarchy of modules by describing the relationship between them with its own descriptive language. Furthermore, it provides the facility of restricting view of modules and the facility to be able to see modules through a richer interface. Within the module hierarchy, modules in lower levels can communicate across more expressive interfaces, and modules in higher levels can enforce more restrictive ones. Bauer, Appel and Felten extended the Java package mechanism and developed a linking system supporting hierarchical modularity similar to that of Standard ML and CM [7].

1.3.3 Digital signatures

A digital signature scheme consists of a signature algorithm and a verification algorithm. A digital signature of a document is a value depending on the contents of the document and on some secret only known to the signer, i.e., a private signature key, that associates the document with an entity, i.e., a public verification key. The verification algorithm usually takes the document and the public verification key as input, but in exceptional cases the document - or parts of the document - can be recovered from the signature and the document does not have to be provided signature verification. The property that a third party can resolve the validity of a digital signature without having the signer's private key is called the *verifiability* property of digital signatures.

Digital signatures support *nonrepudiation*. Public-key cryptography is a natural source for digital signature schemes. In such a scheme, it is computationally infeasible to derive the signature key from the verification key. Despite the similarities in the underlying mathematical techniques cause confusion between two schemes, these schemes have fundamentally different purposes. Encryption protects the confidentiality of a message and has to be reversible. Digital signatures provide data origin authentication and nonrepudiation. The El Gamal signature scheme [15] and one of its successful successors, the digital signature algorithm (DSA) [33] demonstrate that signing does not have to be encryption with a private key. These schemes sign on a message with a private signature key without encrypting the whole message or a part of the message. In contrast, another widely used digital signature scheme, the RSA algorithm [41] allows users to use the same algorithms for signing and for encryption.

Another important topic in digital signatures is to guarantee the authenticity of public keys. How do verifiers know that the public verification key they are using to check signatures indeed correspond to the right party? There has to be a reliable source that links 'identities' with cryptographic keys. This trusted source of key binding is called a *certification authority* (CA). CAs guarantee the link between user and cryptographic key by a signing a document (called a *certificate*) that contains user name, key, name of the CA, expiry date, etc. There exist several proposals specifying the precise format of a certificate, most notably the certificates used in the X.509 Directory Framework [11]. To verify a certificate, verifiers need a public verification key to check the CA's signature, and the key can be guaranteed by another CA; then it forms a chain of certification. Public verification keys can be obtained by a chain of certificates, but in any case, verifiers must know at least one public key, whose authenticity is proved by some other way, for example, by being sent from an authorized company, or by being obtained in person.

Due to its simplicity and strengths explained above, the digital signature schemes are very popular, and widely used. Considering that program code itself is simply a text before it is interpreted as a list of machine instructions, it is not at all unusual to treat program code as a document, sign it using one of the digital signature schemes, and verify the signature. As mobile code and commercial software components are becoming prevalent, signing code with one of the digital signature schemes play an important role in giving users some trust in the code. In other words, users can tell whether the code is coming from someone they can trust or not after verifying the signature of code. Despite its popularity, code-signing suffers from a critical weakness, the lack of expressiveness. The example in Section 1.1 illustrated the weakness could cause a security problem in a real world.

1.3.4 Security-concerned linking

While type theory is a well developed field, there has been relatively little work on the semantics of linking, and less work where linking is a security-critical operation.

Cardelli addresses type-safety issues with separate compilation and linking [10]. He introduces a simple language, the simply typed λ-calculus, with a primitive module system that supports separate compilation. He then informally, but rigorously, proves that his linking algorithm terminates, and if the algorithm is successful, that the resulting program will not have a type error. (Here a type error means calling a function with the wrong number or type(s) of arguments, or using a number as a function.) However, it assumes that all types are known at link time, and does not address (mutually) recursive modules.

Dean's work [13] mentions the necessity of handling dynamic linking for secure systems. Although dynamic linking itself is an old idea (appearing in Multics [34], among other systems), it has become more important to reason about the behavior (or the semantics) of dynamic linking as mobile code has gained its popularity. Dean addresses the design of a type-safe dynamic linking system, and describes how static typing and dynamic linking interact in a security-relevant way. Dean also gives a formal specification of his model written in PVS [35] to prove his design to be safe.

Zaremski and Wing proposed *specification matching* of software components [49]. Specification matching is a process of determining if two software components are related when retrieval (from a software library), reuse, substitution or subtyping of a component is considered. Two components are compared based on description of the component's behaviors as well as their syntax appearances. The behavior of a software component is formally modeled using pre- and postconditions written as predicates in first-order logic. They uses the Larch prover [20] for their implementation, because theorem proving is required to determine match or mismatch of two software components.

Recent practical work for secure systems prefers interposing security code at the operating system boundary to observe and modify the data passing through. For software component composition, untrusted components are encapsulated in *wrapper programs*, and these programs have full control over the interactions between encapsulated components and operating systems, or over the interactions among components [14, 43, 44]. The code of a wrapper can perform access control checks, audit, attempt to detect intruders, and even monitor covert channels.

Fraser, Badger and Feldman [14] proposed a system in which the body of wrapper programs are written in a variant of C (called the Wrapper Definition Language) and the dynamic aspects of creating wrappers and executing components are specified in their framework (called the Wrapper Life Cycle framework). Sewell and Vitek abstracted information flow properties of wrappers in a process calculus [44]. They designed a language for composing concurrently-executing components, including primitives for encapsulating components and controlling their interactions. Their language is based on the *box-*π calculus [43], focusing on the rigorous formal statement and proofs of information flow properties.

Chapter 2

The Design of SL

In this chapter, I will describe how Secure Linking works and what SL can guarantee at link time, with a simple example. The chapter begins with a brief description of a situation which simplifies the real world problem discussed in Section 1.1. It is followed by the explanation how SL solves the problem.

2.1 A Simple Example

As an analogy of the real world problem of Microsoft, suppose that a code consumer Bob has a buggy component `sharedCom` in his web browser, causing a system hole vulnerable to security attacks.

Bob finds a seemingly interesting HTML document from Alice's system, and he is asked to download a JavaScript program from Alice's in order to read that document. Should Bob download and install Alice's code or not? How does Bob assure that the code from Alice would not attack his system's known weakness? Bob might feel safe with Alice's code if Bob can get the guarantee that Alice's code makes safe use of the buggy component `sharedCom`. Let's call this behavioral property of a software component *safeUseOfSharedCom*.

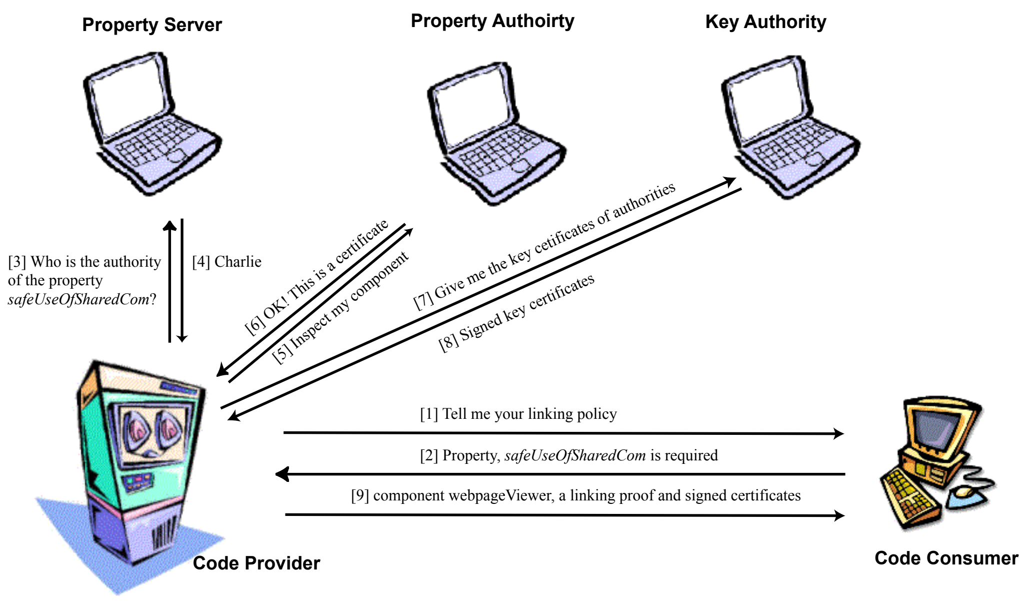
Property Server
Property Authoirty
Key Authority
[3] Who is the authority of the property *safeUseOfSharedCom*?
[4] Charlie
[6] OK! This is a certificate
[5] Inspect my component
[7] Give me the key certificates of authorities
[8] Signed key certificates
[1] Tell me your linking policy
[2] Property, *safeUseOfSharedCom* is required
[9] component webpageViewer, a linking proof and signed certificates
Code Provider
Code Consumer

Figure 2.1: Interaction of principals in Secure Linking

Bob is going to depend on Secure Linking to verify whether or not his download from Alice is safe. In Secure Linking composing software components is a complicate task in which many principals and chains of trust are involved, and where they heavily interact with each other. Figure 2.1 illustrates the interaction among principals involved in Secure Linking. I'll explain how a principal who gets a piece of suspicious code verifies the trustworthiness of the code step by step.

2.2 Code Provider

Alice in this example is called a *code provider* because Alice is providing code which is considered suspicious until it is proved to be safe. Alice could be a programmer who wrote the code for herself, or a software vender who sells the code. Alice could be even a colleague who is working together on a large software project.

Just as a code provider comes from various sources, suspicious-before-verified code is transferred and linked via various ways to Bob's system. As in the previous example, code might be a plug-in module transferred by network for a web browser. Code might also be a commercial off-the-shelf component purchased from third-party software vendors, or a software component written by co-workers. The transferred code can be linked either dynamically (like a web browser's plug-in module) or statically.

2.3 Code Consumer

Bob in this scenario is called a *code consumer*. A code consumer is the principal who uses outside software components in his system, and wants to protect his system by verifying the components at link time according to his own linking policy.

A code consumer can establish his own linking policy which can be consulted by code providers and used for component verification at link time. For example, Bob's linking policy might say that "my system is going to accept only the outside code making safe use of `sharedCom`." Bob also specifies that he is going to trust a certificate from a principal Charlie, about a snippet of code, saying that "this code doesn't call at all `sharedCom`," or "this code makes calls to `sharedCom`, but doesn't exploit the security hole maliciously." To help code providers build a linking proof, a code consumer specifies some information other than required properties in his linking policy, such as enumerating a list of names of trusted authorities, or a list of components in his system, which are visible to a foreign software component.

A code consumer's linking policy should be available to code providers to give hints to build a linking proof in advance; then a code consumer asks code providers to hand in a proof whenever they send a software component. For example, Bob asks Alice to show him a proof that her JavaScript program is safe to link other components in his browser. In other words, Bob requires a proof that Alice's program has the properties specified in his linking policy.

After getting a linking proof and the code from Alice, Bob checks the proof. If the proof is verified, Bob will believe that the code is safe and run the code; otherwise, he will refuse to execute the downloaded code and give up reading the HTML document.

2.4 Secure Linking Policies

A code consumer's linking policy usually consists of three parts: a list of useful properties the code consumer requests from outside software components, a list of names of trusted authorities, and the description of pre-installed library components in his system.

A code consumer could use the Secure Linking framework, in building a system from components, to specify and enforce policies such as,

- Bob requires all modules must have been mechanically inspected by a virus detector.
- Certificate Authority `Alice` can vouch for the public key of the virus detector.
- Foreign software components link 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.
- For version 1.6 through 1.9 of `GUI` it is OK to substitute version 2.4.

On the other hand, a code provider can specify properties expected from imported library components:

- `Game` requires (links to) exactly version 1.3 of `GUI`.
- `Compiler` requires (links to) any version of `SymbolTable` that has `efficientLookup` property as certified by `underwriters_laboratories.`
- `Compiler` requires an implementation of `SymbolTable` that has a property `typeSafety`.

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

2.5 Properties

A code consumer requires a software component to have certain 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 generally useful properties which help systems protect themselves from malicious outside codes, such as

- "this software component is type-checked,"
- "this software component never accesses outside of the memory assigned to it,"
- "this software component doesn't read any information from or write any information to the file system,"
- "this software component never reveals a secret of high-level processes to low-level processes," or
- "this software component doesn't produce any arithmetic overflow or underflow."

On the other hand, some properties are useful to fix specific problems. For example, consider the buggy shared component containing an inadvertent security flaw explained above. That is, a correct implementation of this shared component would permit the safe execution of untrusted (possibly malicious) JavaScript programs, but the buggy one might still be safe in connection with nonmalicious (trusted) code. Therefore, the consumer could establish a security policy that specifies, for this version-number example of the control, that the JavaScript linked to it must be certified as nonmalicious by a trusted party. In the above example, Bob specified this requirement in the form of required properties.

Note that these properties are meaningful only with the connection of the version-related problem though properties exemplified in this section address quite general problems for system security. This demonstrates the strength of Secure Linking over other traditional linking systems discussed in Section 1.3.4. Since those linking systems has their own security policies defined in advance, it is very important to determine a set of policies to support when designing the system. The policies should be general enough to satisfy the demand of large range of users, and powerful enough to protect the system from as many security attacks as possible. At the same time, however, those systems suffer from lack of composability. Because the systems have built-in linking policies, and are carefully designed to implement those policies, it is not always possible to combine different linking policies.[1] It is probable to build a system guaranteeing noninterference property of properly signed foreign component, but it will not be straightforward to build a type system supporting noninterference as well as checking array bounds out of two already existing type systems.

Secure Linking has an advantage over other security-concerned linking systems in this aspect. SL allows users to choose useful properties about software component behavior, and to enforce them at link time. Within SL, any individual property enforced by security-concerned systems, such as Java type safety, behavior subtyping, memory protection, array-bound check, or noninterference, can be uniformly treated as a property, and each system guaranteeing a specific property becomes an authority of that property; now, users just pick properties they think critical to protect their systems, and establish a Secure Linking policy out of those properties.

2.6 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

[1] to build a system guaranteeing useful properties from different security-concerned linking systems

as true if a software component has assurances made by trusted third-party authorities. The trusted authorities can generate assurances resulting from a software audit or some other verification processes for software engineering. Such assurances are usually encoded as digitally signed statements. Those authorities are called *property authorities* in Secure Linking, and the signed statements are called *certificates*. After verifying the digital signatures on the statements, the property certificates made by trusted property authorities are accepted as true and the components of those statements are considered to have the properties in those statements.

For example, Bob might trust a wide variety of well-known companies to claim "This module contains no malicious attacks and has been checked for common buffer-overrun problems," but might require more assurance for the claim "this module is implemented sufficiently carefully to be able to withstand linking with software that contains malicious attacks." If Bob wants to run a particular JavaScript program in an environment that has low resistance to attack, and if he trusts Charlie as a property authority for the *nonmalicious* property, then he should obtain a signed certificate from Charlie about the particular JavaScript program he's about to link and execute.

2.7 Library

Although it is possible to keep a software component self-contained, it is not unusual for a foreign software component to depend on other underlying, pre-installed software components[2] in a code consumer's system. The pre-installed components are trusted by the code consumer, and the foreign component will use the pre-installed components by importing them. These pre-installed components are called *library components* in Secure Linking.

To help a code provider build a proof, a code consumer declares what library components he has and what properties are exported by each of those components in his linking

[2] Consider software components provided by an operating system, or software components for GUI

policy. Code consumers list the visible library components explicitly in their linking policy; by simply not listing them code consumers can hide some security-critical components from outside, and thus prohibit security-critical components from being called in a malicious way by a foreign component. At the same time, a foreign component from a code provider declares what components it imports and what properties are required for each of the imported components. By addressing required properties as well as its name for an imported library component, a foreign component gives more detailed information about its imported library component.

At verification time, Secure Linking checks whether or not all the library components imported by a foreign component are provided by the code consumer and listed in the code consumer's linking policy.

2.8 Key Authorities

Since all certificates from property authorities come with digital signatures, a code consumer must know the signers' public keys in order to check the validity of the digital signatures. The code consumer must at least know who can provide the right public keys for verifying digital 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 its public key, and key certificates from a key authority are trusted after being verified with the public key of the key authority. The public key in the certificate can be used to verify the digital signature of another key certificate. This results in a chain of key certificates to get a trustable public key of a given principal as explained in Section 1.3.3.

Diane in Figure 2.1 is a key authority Bob trusts. Alice is responsible for sending Bob all key certificates needed to verify her linking proof; that means, Alice should get a key certificate of Charlie from Diane, and send the key certificate with her proof to Bob.

When verifying Alice's proof, Bob gets the public key of Charlie from the accompanying key certificate, and verifies the digital signature on Charlie's property certificate.

2.9 Property Servers

Just as a code consumer 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 a code consumer specifies that he trusts a principal as someone who will let him know the property-authority bindings. Therefore, the code consumer doesn't have to enumerate the names of property authorities for every required property in his linking policy, and it relieves him from modifying the linking policy whenever a property-authority binding changes.

In the example, Bob trusts Emily as a property server, and specifies it in his linking policy. Alice, when starting to build a linking proof, might not know who is the authority who can inspect her code and issue a property certificate about the *nonmalicious* property. The first thing she should to do is to consult Emily, the trusted property server, about this property-authority binding. Suppose that Emily makes a signed statement that a set of authorities including Charlie can make an assurance of the *nonmalicious* property. With this information, then, Alice can proceed to consult Charlie for a property certificate. The property-authority certificates from Emily are also submitted to Bob as well as a proof and key certificates from Alice.

2.10 Linking Decision

To decide whether it is safe 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. A code consumer checks the proof from a code provider with the certificates by using a trusted proof checker, and if the proof is valid, the code consumer accepts the foreign component to run with other components in his system; otherwise he rejects the foreign component. Since the certificates a code provider submits are all digitally signed, digital signatures are verified during the proof-checking time.

For example, Bob checks whether or not the component from Alice has the property *safeUseOfSharedCom* specified in his policy. The proof from Alice is verified by a trusted proof checker with certificates from a property authority *Charlie*, and other supporting authorities (such as a key authority *Diane* and a property server *Emily*).

Chapter 3

SL Interface

The user interface of the Secure Linking framework consists of two different languages, one for describing code consumers' linking policies, and one for describing components. The user interface languages adopts the XML syntax; therefore they can be parsed by any XML parser and modified even with a simple text editor. The Secure Linking framework has XML parsers for each language, producing logical formulas written in the Secure Linking logic from XML files. Since they must be trusted by users[1], the parsers are included in the TCB of the framework.

In this section, I will give the models of linking policies and components in Secure Linking, and explain the syntax and the semantics of the two description languages by way of the example given in Chapter 2.

3.1 Component Description Language

Going back to the example in Chapter 2, a code provider must describe her software component to a code consumer and to property authorities who are going to inspect her component. The component description is written down by using the user interface lan-

[1]Users believe that the logic formulas from the parsers will have adequately represented their linking policies or component descriptions.

guage of the Secure Linking framework, whose abstract syntax is illustrated in Figure 3.2. The result might look like one given in Figure 3.1.

When specifying a software component in the component description language of SL framework, a pair of XML tags, ⟨componentDsc⟩ and ⟨/componentDsc⟩ are used for marking the beginning of the description and the end of the description. A component description can be constructed by enumerating all the necessary information for a software component from the scratch (call it *basic component description*), or by combining two existing component descriptions (call it *combined component description*). In this section, I will give the model of basic component description with its related language syntax, followed by the explanation of combined component description.

A basic component description, $\mathcal{C}$, consists of four parts, $(\mathcal{N}, \mathcal{M}, \mathcal{E}, \mathcal{I})$, where

- $\mathcal{N}$ is the name of a software component. A tag, ⟨name⟩ and its counterpart, ⟨/name⟩ are used for specifying the name of a software component. A name is a local identifier of convenience for referring a software component.
- $\mathcal{M}$ is a list of modules making up of a software component.
- $\mathcal{E}$ describes what a software component makes visible from the outside.
- $\mathcal{I}$ enumerates other components which a given software component depends on.

3.1.1 Modules

The module part of the description specifies which code files compose a software component. Every entry of this part is given by its file name and the hash code of it.

Secure Linking requires that the cryptographic hash code of every binary file should be checked during the verification process. It prevents binary files from be tampered with after a component description and its accompanying proof have been generated; therefore, given binary files (programs or resources) and a component description, the

```
⟨componentDsc⟩
  ⟨name⟩ webpageViewer ⟨/name⟩

  ⟨modules⟩
    ⟨item hash = “194CA77319” ⟩ display.class ⟨/item⟩
    ⟨item hash = “EF41900142” ⟩ userInput.class ⟨/item⟩ ⟨/modules⟩

  ⟨exports⟩
    ⟨type⟩
      ⟨item⟩ class contentFrame⟨/item⟩
      ⟨item⟩ interface inputForm⟨/item⟩ ⟨/type⟩
    ⟨property⟩
      ⟨item⟩ safeUseOfSharedCom ⟨/item⟩ ⟨/property⟩ ⟨/exports⟩

  ⟨imports⟩
    ⟨component⟩
      ⟨name⟩ sharedCom ⟨/name⟩
      ⟨required⟩
        ⟨type⟩
          ⟨item⟩ class sharedActiveXControl ⟨/item⟩ ⟨/type⟩
        ⟨property⟩
          ⟨item⟩ prpTypeSafety ⟨/item⟩
        ⟨/property⟩ ⟨/required⟩ ⟨/component⟩ ⟨/imports⟩ ⟨/componentDsc⟩
```

Figure 3.1: An example of component description

ComponentDsc	::=	$\texttt{componentDsc}\ \mathit{Name}\ \mathit{DscBody}$
Name	::=	$\texttt{name}\ \mathit{NameId}$
DscBody	::=	$\mathit{Modules}\ \mathit{Exports}^*\ \mathit{Imports}^* \mid \mathit{CombineExp}$
CombineExp	::=	$\texttt{combine}\ \mathit{NameId}\ \mathit{NameId}$
Modules	::=	$\texttt{modules}\ \mathit{ModuleIdItem}^+$
Exports	::=	$\texttt{exports}\ \mathit{Types}^*\ \mathit{Properties}^*$
Imports	::=	$\texttt{imports}\ \mathit{ImportComponentId}^*$
ImportComponentId	::=	$\texttt{component}\ \mathit{Name}\ \mathit{RequiredPrps}^*$
RequiredPrps	::=	$\texttt{required}\ \mathit{Types}^*\ \mathit{Properties}^*$
NameId	::=	*String*
ModuleIdItem	::=	$\texttt{item}\ \mathit{HashCode}\ \mathit{ModuleName}$
ModuleName	::=	*String*
HashCode	::=	$\texttt{hash}\ \mathit{QuotedString}$
Types	::=	$\texttt{type}\ \mathit{TypeIdItem}^+$
TypeIdItem	::=	*String*
Properties	::=	$\texttt{property}\ \mathit{PropertyIdItem}^+$
PropertyIdItem	::=	*String*
String	::=	$(0 \mid 1 \mid \ldots \mid a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \mid\ \mid \_)^+$
QuotedString	::=	"*String*"

Figure 3.2: Abstract syntax of the component description language

Secure Linking framework checks the integrity of those code files by recalculating their cryptographic hash codes, and comparing with the hash codes stored in the component description.

A list of modules begin with a tag ⟨modules⟩, and ends with its counterpart tag ⟨/modules⟩. A tag ⟨item⟩ is used for providing the information of each module (that is, a file name of a binary module and its cryptographic hash code).

The example in Figure 3.1 shows that the component `webpageViewer` consists of two code files, named `display.class` and `userInput.class`, and their cryptographic hash codes are also provided in the component description.

3.1.2 Exports

Components can export two kinds of things: type identifiers (such as class names or function names) and properties. It is typical for a program to call functions, or to use classes or structures outside of its scope, and it has been linkers' responsibility to find out what functions, classes, or structures are really referenced. Linkers resolve the type identifiers by consulting the symbol tables handed to them by compilers.

In Secure Linking, a code provider can export properties as well as mere type identifiers of a software component. A property of a software component is a high-level description about the behavior of a software component, and can be enforced and verified at link time using digitally signed certificates from authorities as explained in Chapter 2.

As for the visibility of type identifiers, programming languages allow a programmer to control the visibility of identifiers by putting access modifiers in the program. For example, an ML programmer can hide some internal variables or types from outside view by defining a signature and binding it to an ML structure implementation. Similarly, with object-oriented programming languages such as Java or C++, programmers can put access modifiers on classes, functions or data members. In addition to class-level or structure-level access control, the Standard ML/Compilation Manager [9] supports a mechanism for specifying whether a type identifier is visible from outside or not by declaring a list of visible identifiers in a component[2] level. Java's package lacks this kind of control; it result in re-exporting all the type identifiers from dependent packages although it is not desirable.

The Secure Linking framework allows users to control the visibility of properties as well as the visibility of type identifiers. The export part is marked with a tag ⟨exports⟩ and its counterpart tag ⟨/exports⟩. Under the tag ⟨exports⟩, a tag ⟨type⟩ is used to enumerate type identifiers visible from outside, and a tag ⟨property⟩ is used to enumerate the properties names exported by a software component. Type identifiers are hidden from outside of a component if they are not listed in the *export* part, even if they are defined within the component or if they are visible from imported components. Imported properties[3] are hidden by default, and will not be exported unless they are listed in the

[2]It is called a *group* in SML/CM

[3]Exported properties from dependent software components

export part explicitly.

In the example of Figure 3.1, the component `webpageViewer` exports a property `safeUseOfSharedCom` saying that this component doesn't call the buggy component `sharedCom` at all, or call the component in a safe way. It also exports two type identifiers, `class contentFrame` and `interface inputForm`.

3.1.3 Imports

The import part specifies what other components a component depends on. Every entry of the import part consists of a brief description of an imported component. The import component description differs from the complete component description in terms that it doesn't have the module information, but only contains the name of a component and an export part. Note that this enables different implementations of a software component to be used as long as implementations has the same exporting interface. An import part of a component description is a list of imported component descriptions. The import part is marked with a tag ⟨imports⟩ and its counterpart tag ⟨/imports⟩ . Under this tag, each imported component description begins with a tag ⟨component⟩, and ends with a tag ⟨/component⟩. Each imported component description consists of

- the name of the imported component
- a list of properties required from the imported component
- a list of required type identifiers

When locating an imported component at link time, the Secure Linking framework searches and links a software component which has the name specified in the description, and exports all the required properties and type identifiers. Figure 3.1 shows that the component `webpageViewer` depends on a component named `sharedCom` which exports a property `prpTypeSafety` and a type identifier `class sharedActiveXControl`. The

Secure Linking framework will allow `webpageViewer` to import any implementation of `sharedCom` as long as it exports the given type identifiers and properties.

3.1.4 Combining component descriptions

A component description can be constructed from two existing component descriptions. It is very useful to make it possible to combine component descriptions, especially when considering digitally-signed certificates from property authorities. In Secure Linking, a component description with modules is required to be digitally signed by property authorities to prove its safety of linking. A property authority can guarantee only a set of properties, and announces this set in advance; when signing, it is reasonable for the property authority to sign on only the properties he can guarantee, rather than signing 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. This is the reason why the Secure Linking framework provides a way of combining component descriptions. After collecting component descriptions assured by property authorities, a code provider 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.

A combine component description consists of

- a new name of resulting component description specified by using the tag ⟨name⟩
- an expression which combines component descriptions. It begins with a tag ⟨combine⟩, followed by the names of two source component descriptions.

3.2 Linking Policy Description

3.2.1 Linking policy model

The linking policy description language allows code consumers to set Secure Linking policies that affect how applications run on their machines. A linking policy, $\mathcal{P}$, in Secure Linking is a four-tuple, $(\mathcal{R}, \mathcal{K}, \mathcal{S}, \mathcal{L})$, where

- $\mathcal{R}$ is a set of properties a policy writer (a code consumer) expects from foreign software components.
- $\mathcal{K}$ is a list of key authorities whom a policy writer trusts. The policy writer has the public keys of key authorities for verifying their digital signatures.
- $\mathcal{S}$ is a list of property servers whom a policy writer trusts as sources of providing bindings between a property and its property authorities.
- $\mathcal{L}$ is a list of library components which are visible to outside software components.

Figure 3.4 shows the abstract syntax of the user interface language for establishing a linking policy. The linking policy of Bob the example of Chapter 2 might look like one in Figure 3.3. Every linking policy begins with a root tag, ⟨linkingPolicy⟩, and ends with its counterpart tag, ⟨/linkingPolicy⟩. The four parts of a linking policy are given within these two tags, and the order in which each part appears doesn't matter.

3.2.2 Required properties

A tag ⟨requiredPrps⟩ and its counterpart tag ⟨/requiredPrps⟩ are used to specify the properties a code consumer requires from all the components from outside. Under this tag, the names of properties are enumerated by using a tag ⟨item⟩. In Figure 3.3, Bob specifies that he will accept software components with a property `safeUseOfSharedCom`.[4]

[4] Suppose that the property `safeUseOfSharedCom` stands for a software behavior that a software component doesn't call the buggy `sharedCom` at all or it makes only safe use of the buggy component.

```
⟨linkingPolicy⟩

 ⟨requiredPrps⟩
   ⟨item⟩ safeUseOfSharedCom ⟨/item⟩ ⟨/requiredPrps⟩

 ⟨keyAuth⟩
   ⟨item⟩ Diane ⟨/item⟩ ⟨/keyAuth⟩

 ⟨propertyServer⟩
   ⟨item⟩ Emily ⟨/item⟩
   ⟨item⟩ Ethan ⟨/item⟩ ⟨/propertyServer⟩

 ⟨library⟩
   ⟨component⟩
     ⟨name⟩ sharedCom ⟨/name⟩
     ⟨module⟩
       ⟨item hash = “9213317FCA”⟩ sharedComOld.class ⟨/item⟩ ⟨/module⟩
     ⟨exports⟩
       ⟨type⟩
         ⟨item⟩ class sharedActiveXControl ⟨/item⟩ ⟨/type⟩
     ⟨/exports⟩ ⟨/component⟩
   ⟨component⟩
     ⟨name⟩ sharedCom ⟨/name⟩
     ⟨module⟩
       ⟨item hash = “683EE18970”⟩ sharedComNew.class ⟨/item⟩ ⟨/module⟩
     ⟨exports⟩
       ⟨type⟩
         ⟨item⟩ class sharedActiveXControl ⟨/item⟩ ⟨/type⟩
       ⟨property⟩
         ⟨item⟩ prpTypeSafety ⟨/item⟩
       ⟨/property⟩ ⟨/exports⟩ ⟨/component⟩ ⟨/library⟩ ⟨/linkingPolicy⟩
```

Figure 3.3: An example of linking policy

$$
\begin{array}{lll}
\mathit{LinkingPolicy} & ::= & \texttt{linkingPolicy}\ \mathit{Library}^*\ \mathit{PropertyServer}^*\ \mathit{RequiredPrps}^* \\
\mathit{Library} & ::= & \texttt{library}\ \mathit{ExportComponentId}^+ \\
\mathit{KeyAuthority} & ::= & \texttt{keyAuth}\ \mathit{ServerIdItem}^+ \\
\mathit{PropertyServer} & ::= & \texttt{propertyServer}\ \mathit{ServerIdItem}^+ \\
\mathit{RequiredPrps} & ::= & \texttt{requiredPrps}\ \mathit{PropertyIdtem}^+ \\
\mathit{ExportComponentId} & ::= & \texttt{component}\ \mathit{Name}\ \mathit{Modules}\ \mathit{Exports}^* \\
\mathit{Name} & ::= & \texttt{name}\ \mathit{NameId} \\
\mathit{Modules} & ::= & \texttt{module}\ \mathit{ModuleIdItem}^+ \\
\mathit{Exports} & ::= & \texttt{exports}\ \mathit{Types}^*\ \mathit{Properties}^* \\
\mathit{NameId} & ::= & \mathit{String} \\
\mathit{ModuleIdItem} & ::= & \texttt{item}\ \mathit{HashCode}\ \mathit{ModuleName} \\
\mathit{ModuleName} & ::= & \mathit{String} \\
\mathit{HashCode} & ::= & \texttt{hash}\ \mathit{QuotedString} \\
\mathit{Types} & ::= & \texttt{type}\ \mathit{TypeIdItem}^+ \\
\mathit{TypeIdItem} & ::= & \mathit{String} \\
\mathit{Properties} & ::= & \texttt{property}\ \mathit{PropertyIdItem}^+ \\
\mathit{PropertyIdItem} & ::= & \mathit{String} \\
\mathit{ServerIdItem} & ::= & \mathit{String} \\
\mathit{String} & ::= & (0|1|\ldots|a|b|\ldots|z|A|B|\ldots|Z|\ |\_)^+ \\
\mathit{QuotedString} & ::= & \text{“}\mathit{String}\text{”}
\end{array}
$$

Figure 3.4: Abstract syntax of the linking policy description language

3.2.3 Key authorities

Tags ⟨keyAuth⟩ and ⟨/keyAuth⟩ are used for enumerating the names of trusted key authorities whom a code consumer trusts. Key authorities are responsible for providing the bindings between a principal and its public key. A code consumer must know the public keys of those key authorities, and public keys of any other principals should be accompanied by verifiable key certificates from trusted key authorities. Key certificates usually form a chain of trust, and this is why the names of trusted key authorities are exposed to code providers. In order to get any public keys used in their linking proofs, code providers must make it sure that the chain of their key certificates ends up with a certificate signed by any of the trusted key authorities of a code consumer.

In the example, Bob has one trusted key authority, *Diane*; therefore, Alice should get key certificates directly from Diane, or should build a chain of key certificates originated from Diane.

3.2.4 Property servers

As explained in Chapter 2, neither a code consumer nor a code provider has to know who is responsible for guaranteeing which properties; instead both parties depend on another kind of authorities called *property servers*. They certify the bindings between a property and its property authorities, and a code consumer trusts the certificates only from his trusted property servers after verifying their digital signatures. Since the assurances about property-authority bindings should be given together with a linking proof, a code consumer must specify at least one property server of his trust. Tags ⟨propertyServer⟩ and ⟨/propertyServer⟩ are used for enumerating the names of trusted property servers.

Bob enumerates two property servers, *Emily* and *Ethan*, in this linking policy by using a tag ⟨item⟩ under the ⟨propertyServer⟩ tag.

3.2.5 Library components

Code consumers can control the visibility of software components in their system (called *library components*) by enumerating, in their linking policies, only the library components which they want to allow to be visible outside, and by hiding the library components which they don't want to expose. A tag ⟨library⟩ is used for enumerating the library components which a code consumer wants to export.

A library component description, $\mathcal{L}$, is a tuple, $(\mathcal{N}, \mathcal{M}, \mathcal{E})$, where

- $\mathcal{N}$ is the name of a library component.
- $\mathcal{M}$ is a list of modules making up of a library component.
- $\mathcal{E}$ consists of a list of exported type identifiers and a list of exported property names.

Note that the library component description is different from the basic component description only in terms that it doesn't have an import part. Since the library components are already trusted as a part of a code consumer's system, the safety of a library component's importing other components doesn't need to be verified; indeed it is more important to specify what the library components provide to outside components of the system than what they require from other parts in the same system.

The library part of a linking policy consists of a list of library component descriptions, and each of which looks similar to basic component description. Each library component description is marked by tags ⟨component⟩ and ⟨/component⟩, and the syntax and semantics of sub-tags are the same to those used for basic component description.

In Figure 3.3, a code consumer has two different library component the name of both components is `sharedCom`. They have the same exported type identifiers as well as the same component name. These components could be different implementations of the same traditional component interface.[5] But one component can be distinguished from

[5] Traditional component interfaces reason about only type identifiers

the other within the Secure Linking framework since their exported properties are different. A foreign component described in Figure 3.1 is importing a component named `sharedCom` supplying a type identifier `class sharedActiveXControl` and a property named `prpTypeSafety`. Although Bob has two library components named `sharedCom`, the second library component will be picked for linking by the Secure Linking framework. It is because the second component exports the property `prpTypeSafety` required from the foreign component while the first one doesn't.

Chapter 4

SL Logic: Syntax

Linking policies of code consumers and component descriptions from code providers are written in the user interface languages explained in Chapter 3; before being verified, they are translated into the linking logic by the dedicated parsers. The linking logic plays as a core part in the Secure Linking framework. The linking decision process is expressed in the linking logic, and the decision about the safety of a foreign software component depends on the proof written in this logic. Hence, the logic should have clear interface for easy use and semantics powerful enough to cover different kinds of complicate linking decision processes.

The logic of the Secure Linking framework is a higher-order logic defined on top of PCA logic [3], and has a semantic model. In the Secure Linking logic, the meaning of each operator is defined in terms of the underlying PCA logic and its underlying higher-order logic, and its inference rules using operators are then proved as lemmas.

In this chapter, I explain the syntax and the inference rules of the linking logic. These two make up the interface of the linking logic. Understanding the interface of the linking logic gives users good intuition how to construct a linking proof written in the Secure Linking logic. The next chapter is dedicated to a more detailed explanation of the semantic model of the linking logic.

4.1 Term Constructors and Predicates

One of the basic units in the linking logic is a *logic term*. Terms are built by using *term constructors*, and various *predicates* are used to express the relation among logical terms. A formula in the Secure Linking logic is constructed by applying terms to a predicate. Since the Secure Linking logic adopts a semantic approach, each operator (term constructors or predicates) has its own definitions written in the underlying higher-order logic.

Term constructors build logical terms to represent the entities in Secure Linking (such as principals, component description, certificates, and so on) or fundamental structures (such as lists and sets). I will briefly explain the syntax of some terms in this section, but not of all.

Predicates are used for describing relations of SL logic terms. The predicates are categorized by their purposes in the linking logic: predicates for linking decision, predicates for certificates from various authorities, and so on. The predicates are categorized by the meaning of relation which each predicate stands for. Although the meaning of the predicates (or inference rules) is not the topic of this chapter, it would be helpful for understanding the intuitive roles of predicates in Secure Linking.

In this section, I will explain how to write formulas in the Secure Linking logic with using the examples given in Chapter 3, and give the full syntax of term constructors and predicates.

4.1.1 Translating linking policies

Linking policies are translated into the Secure Linking logic by the parser explained in Chapter 3. Recall the example of Alice and Bob given in Chapter 2, and how Bob's linking policy has been written in the linking policy description language in Chapter 3. The linking policy in Figure 3.3 is translated by the parser of the Secure Linking framework,

```
1   bob_n_prp1         ≡ mk_str(s, mk_str(a, ...)).
2                                  % a string ‘‘safeUseOfShareCom’’
3   bob_r_prp1         ≡ mk_prp(bob_n_prp1).
4   bob_rprps          ≡ set_union(set_singleton(rq_component_name_exists),
5                        set_union(set_singleton(rq_export_ids_exists),
6                        set_union(set_singleton(rq_code_hash_checkable),
7                        set_union(set_singleton(mk_prp_rq(bob_r_prp1)),
8                        set_empty)))).
9   diane :  worldview.
10  bob_ax_key_auth1   ≡ key_authority(diane).
11  bob_ax_keybind1    ≡ keybind(diane, ‘‘4920035A6...’’).
12  emily :  worldview.
13  bob_ax_prp_server1 ≡ prp_server(emily).
14  ethan :  worldview.
15  bob_ax_prp_server2 ≡ prp_server(ethan).
```

Figure 4.1: A linking policy written in the SL logic

resulting in formulas in Figure 4.1 and Figure 4.2.

The linking policy begins with a set of required properties from Bob. In this example, Bob requires the property `safeUseOfSharedCom` from foreign software components. To make a property request of the property `safeUseOfSharedCom`, Bob constructs an identifier `bob_n_prp1` by using a term constructor ***mk_str*** (at line 1), and constructs a property out of the identifier by using a term constructor ***mk_prp*** (at line 3).

Required properties from a code consumer are coded as a union set of property requests. Bob's required property `safeUseOfSharedCom` is unioned with three pre-defined properties in Secure Linking. Secure Linking requires a foreign software component to have a name, which is encoded as a property request `rq_component_name_exists` (at line 4). A foreign software component in Secure Linking must export a list of type identifiers (encoded as a property request `rq_export_ids_exists` at line 5), and the modules of the component must have valid cryptographic hash codes (encoded as a property request `rq_code_hash_checkable` at line 6). A property request for the property `safeUseOfSharedCom` is constructed by using a term constructor ***mk_prp_rq*** (at line 7). The term constructors for core property requests (*mk_prp_component_name*,

```
bob_n_libCom1         ≡ mk_str(s, mk_str(h, ...)).
                                % a string ‘‘sharedCom’’
bob_m1_n_libCom1      ≡ mk_str(s, mk_str(h, ...)).
                                % a string ‘‘sharedComOld.class’’
bob_m1_libCom1        ≡ mk_module(bob_m1_n_libCom1, mk_hcode(9213317FCA)).
bob_mlist_libCom1     ≡ list_cons(bob_m1_libCom1, list_nil).
bob_id1_libCom1       ≡ mk_str(c, mk_str(l, ...)).
                                % a string ‘‘class sharedActiveXControl’’
bob_idList_libCom1    ≡ list_cons(bob_id1_libCom1, list_nil).
bob_c_libCom1         ≡ list_cons(mk_prp_component_name(bob_n_libCom1),
                        list_cons(mk_prp_code_hash(bob_mlist_libCom1),
                        list_cons(mk_prp_export_ids(bob_idList_libCom1,
                        list_nil)))).
bob_ax_libCom1        ≡ library_dsc(bob_mlist_libCom1, bob_c_libCom1).
bob_n_libCom2         ≡ mk_str(s, mk_str(h, ...)).
                                % a string ‘‘sharedCom’’
bob_m1_n_libCom2      ≡ mk_str(s, mk_str(h, ...)).
                                % a string ‘‘sharedComNew.class’’
bob_m1_libCom2        ≡ mk_module(bob_m1_n_libCom2, mk_hcode(683EE18970)).
bob_mlist_libCom2     ≡ list_cons(bob_m1_libCom2, list_nil).
bob_id1_libCom2       ≡ mk_str(c, mk_str(l, ...)).
                                % a string ‘‘class sharedActiveXControl’’
bob_idList_libCom2    ≡ list_cons(bob_id1_libCom2, list_nil).
bob_prp1_libCom2      ≡ mk_prp(mk_str(p, mk_str(r, mk_str(p, ...)))).
bob_c_libCom2         ≡ list_cons(mk_prp_component_name(bob_n_libCom2),
                        list_cons(mk_prp_code_hash(bob_mlist_libCom2),
                        list_cons(mk_prp_export_ids(bob_idLIst_libCom2),
                        list_cons(bob_prp1_libCom2),
                        list_nil))).
bob_ax_libCom2        ≡ library_dsc(bob_mlist_libCom2, bob_c_libCom2).
bob_lib_modules       ≡ list_cons(bob_mlist_libCom1,
                        list_cons(bob_mlist_libCom2, list_nil)).
bob_lib_exports       ≡ list_cons(bob_c_libCom1,
                        list_cons(bob_c_libCom2, list_nil)).
```

Figure 4.2: A linking policy written in the SL logic (continued)

mk_prp_code_hash and *mk_prp_export_ids*) will be discussed in Chapter 7. Bob's required property set `bob_required_prps` is constructed out of four property requests by using set constructors ***set_union***, ***set_singleton*** and ***set_empty*** (at lines 4–8).

The second part of Bob's linking policy in Figure 3.3 enumerates the names of trusted authorities. Bob has one trusted key authority named Diane. In the linking policy written in the Secure Linking logic, Bob defines a principal `diane` of type ***worldview*** (at line 9). Principals in the Secure Linking logic (such as code consumers, code providers, and various authorities) are of type *worldview*, which is the type defined in PCA logic. Bob declares that he trust Diane as a key authority by using a predicate ***key_authority*** (at line 10). To verify key certificates, Diane's public key must be known to Bob in advance. Thus Diane's key binding is given by using a term constructor ***keybind*** as a part of the linking policy (at line 11).

Figure 3.3 shows that Bob has two trusted authorities as property servers. As explained in Section 2.9, property servers are principals who are providing the bindings between properties and property authorities. Bob's trust in a property server is encoded by using a term constructor ***prp_server***. Bob declares Emily (at line 13) and Ethan (at line 15) as his trusted property servers.

The remaining part of Bob's linking policy is used for describing the library component information which Bob provides. The resulting linking policy written in the Secure Linking logic is given in Figure 4.2. Each component in the list of library components of Figure 3.3 is translated into the Secure Linking logic one by one. The name of Bob's first library component `sharedCom` is translated into an identifier in the Secure Linking logic (at line 16).

The information of modules is also encoded in the logic as follows: The file name of a module `sharedComOld.class` is coded as an identifier (at line 18), and a module in the logic is constructed by using a term constructor ***mk_module*** out of the identifier (at line 18) and a cryptographic hash code given in the linking policy description (at line 20).

Because a library component usually consists of more than one modules, the modules of a component make up a list. Although Bob's first library component `sharedCom` consists of only one module, a list of modules `bob_mlist_libCom1` is constructed by using list constructors ***list_cons*** and ***list_nil*** (at line 21).

Figure 3.3 shows that Bob's first library component `sharedCom` exports a type identifier `class sharedActiveXControl`. The type identifier `class sharedActiveXControl` is converted into a identifier in the Secure Linking logic (at line 22), and comprises a list of exported identifiers `bob_idList_libCom1` (at line 24). The exported properties of Bob's first library component `sharedCom` consists of three default properties: a component name constructed by a term constructor ***mk_prp_component_name*** (at line 25), cryptographic hash codes of modules constructed by a term constructor ***mk_prp_code_hash*** (at line 26), and a list of exported type identifiers constructed by a term constructor ***mk_prp_export_ids*** (at line 27). All the properties that Bob's first library component exports are encoded in the form of a list by using *list_cons* and *list_nil* (at lines 25–28). In the same way, the second library component of Bob's is translated into the Secure Linking logic (at lines 30–44).

Since library components reside in Bob's system and must be trusted, a binding between modules and its export of a library component is given as an axiom. A binding between the library modules `bob_mlist_libCom1` and the export `bob_c_lib_Com1` (at line 29), and another binding between the library modules `bob_mlist_libCom2` and the export `bob_c_lib_Com2` (at line 45) are translated into axioms by a predicate ***library_dsc***.

Bob has more than one library components in his linking policy, so they are encoded as a list of library modules (at lines 46–47), and a list of library exports (at lines 48–49).

4.1.2 Translating component description

Alice must describe her component `webpageViewer` in the component description language to Bob, and the result is shown in Figure 3.1. The component description is translated

```
alice_n_dsc1           ≡ mk_str(w, mk_str(e, ...)).
                                 % a string ‘‘webpageViewer’’
alice_m1_n_dsc1        ≡ mk_str(d, mk_str(i, ...)).
                                 % a string ‘‘display.class’’
alice_m1_dsc1          ≡ mk_module(alice_m1_n_dsc, mk_hcode(194CA77319)).
alice_m2_n_dsc1        ≡ mk_str(u, mk_str(s, ...)).
                                 % a string ‘‘userInput.class’’
alice_m2_dsc1          ≡ mk_module(alice_m2_n_dsc, mk_hcode(EF41900142)).
alice_mlist_dsc1       ≡ list_cons(alice_m1_dsc,
                         list_cons(alice_m2_dsc, list_nil)).
alice_id1_dsc1         ≡ mk_str(c, mk_str(l, ...)).
                                 % a string ‘‘class contentFrame’’
alice_id2_dsc1         ≡ mk_str(i, mk_str(n, ...)).
                                 % a string ‘‘interface inputForm’’
alice_idList_dsc1      ≡ list_cons(alice_id1_dsc,
                         list_cons(alice_id2_dsc, list_nil)).
alice_prp1_dsc1        ≡ mk_prp(mk_str(s, mk_str(a, ...)))
                                 % a property ‘‘safeUseOfSharedCom’’
alice_export_dsc1      ≡ list_cons(mk_prp_component_name(alice_n_dsc),
                         list_cons(mk_prp_code_hash(alice_mlist_dsc),
                         list_cons(mk_prp_export_ids(alice_idList_dsc),
                         list_cons(alice_prp1_dsc, list_nil)))).
alice_n_imprtCom1      ≡ mk_str(s, mk_str(h, ...)).
                                 % a string ‘‘sharedCom’’
alice_id1_imprtCom1    ≡ mk_str(s, mk_str(h, ...)).
                                 % a string ‘‘sharedActiveXControl’’
alice_idList_imprtCom1 ≡ list_cons(alice_id1_imprtCom1, list_nil).
alice_prp1_imprtCom1   ≡ mk_prp(mk_str(p, mk_str(r, ...))).
                                 % a property ‘‘prpTypeSafety’’
alice_import_imprtCom1 ≡ list_cons(mk_rq_component_name(alice_n_imprtCom1),
                         list_cons(mk_rq_export_ids(alice_idList_imprtCom1),
                         list_cons(mk_prp_rq(alice_prp1_imprtCom1),
                         list_nil))).
alice_importList_dsc1  ≡ list_cons(alice_import_imprtCom1, list_nil).
alice_dsc1             ≡ mk_component_dsc(alice_export_dsc1,
                                          alice_importList_dsc1).
```

Figure 4.3: A component description written in the SL logic

into the Secure Linking logic by the parser of the Secure Linking framework, resulting in formulas in Figure 4.3.

The description of the component `webpageViewer` begins with the component name. It is encoded as an identifier (at line 1). The component `webpageViewer` consists of two module files `display.class` and `userInput.class`, and they are translated into a list of modules in the Secure Linking logic (at lines 3–10) in the same way explained in Section 4.1.1. Exported type identifiers forms a list after they are converted into identifiers in the Secure Linking logic (at lines 11–16), and the property `safeUseOfSharedCom` which Alice's component exports is translated into the Secure Linking logic by using *mk_prp* (at lines 17–18). The export of Alice's component `webpageViewer` consists of three core properties as well as the property `safeUseOfSharedCom`, making up a list (at lines 19–22). The detail is omitted because the translating procedure so far is exactly the same as the procedure given in Section 4.1.1.

A software component usually depends on other components, therefore the information about dependent components must be specified in the component description. Figure 3.1 shows that Alice's component relies on a software component named `sharedCom`. The name of the library component imported by Alice is encoded as an identifier (at line 23), and type identifiers required from the library component are encoded as a list of identifiers `alice_idList_imprtCom1` (at line 27). It is possible for code providers in Secure Linking to ask some useful properties from library components when they import library components. In case of Alice, she requires a property `prpTypeSafety` from the imported component `sharedCom`. The requirement for an imported component is encoded as a list of property requests: a request for the name of the library component (at line 30), a request for type identifiers (at line 31), and a request for the property `prpTypeSafety` (at line 32). It is usual for a component to import more than one library components; thus the specification of imported library components forms a list of imports, each element of which is requests for one library component. The import requests of Alice's component is encoded as a list `alice_importList_dsc1` (at line 34).

A component description consists of its export and import. The description of Alice's component `webpageViewer` is finally constructed by using a term constructor ***mk_component_dsc*** (at line 35).

4.1.3 Formal syntax

In the previous sections, I explained how to write a linking policy and a component description in the Secure Linking logic with an example. The full syntax of term constructors is given in Figure 4.4, and Figure 4.5 shows the syntax of predicates of the Secure Linking logic.

- A component description $\mathcal{C}$ can be made by using a term constructor ***mk_component_dsc*** taking an export part $\mathcal{C}_e$ and an import part $\mathcal{C}_i$. Otherwise, a component description can be a result of combining two component descriptions by a term constructor ***cdsc_combine***. For example, the component description from Alice *alice_dsc1* is constructed out of an export *alice_export_dsc1* and an import list *alice_importList_dsc1* at line 35 of Figure 4.3.

- A component description's *export* is a list of properties. It also can be extracted from a component description by using a term constructor ***cdsc_exprt***. In the example, the export part *alice_export_dsc1* of Alice's component, consisting of four different property requests, is defined at line 19.

- A component description's *import* is a list of imports, each element of which $\mathcal{R}_p$ is a set of property requests. The list can be extracted from a component description by using a term constructor ***cdsc_imprt_list***. At line 30 of Figure 4.3 shows that Alice's import is a list of only one element *alice_import_imprtCom1*.

- An element of a component description's import list is a set of property requests type $\mathcal{R}_p$. A set is constructed by using set constructors ***set_union***, ***set_singleton***

(Terms)	τ	::=	$\mathcal{C} \mid \mathcal{C}_e \mid \mathcal{C}_i \mid \mathcal{M} \mid \mathcal{M}_c \mid \mathcal{L}_m \mid \mathcal{L}_d$
			$\mid \mathcal{P} \mid \mathcal{R} \mid \mathcal{R}_p \mid \mathcal{I}_c \mid \mathcal{I}_s \mid \mathcal{H} \mid \mathcal{L}(\tau)$
			$\mid \mathcal{S}(\tau) \mid \mathcal{K} \mid \mathcal{N} \mid \mathcal{W} \mid \mathcal{F}$
(Component Description)	$\mathcal{C}$	::=	$\mathit{mk\_component\_dsc}(\mathcal{C}_e, \mathcal{C}_i)$
			$\mid \mathit{cdsc\_combine}(\mathcal{C}, \mathcal{C})$
(Component Export)	$\mathcal{C}_e$	::=	$\mathcal{L}(\mathcal{P}) \mid \mathit{cdsc\_exprt}\,(\mathcal{C})$
(Component Import lists)	$\mathcal{C}_i$	::=	$\mathcal{L}(\mathcal{R}_p) \mid \mathit{cdsc\_imprt\_list}(C)$
(Module)	$\mathcal{M}$	::=	$\mathit{mk\_module}(\mathcal{I}, \mathcal{H})$
(Component Modules)	$\mathcal{M}_c$	::=	$\mathcal{L}(\mathcal{M})$
(Library Modules)	$\mathcal{L}_m$	::=	$\mathcal{L}(\mathcal{M}_c)$
(Library Description)	$\mathcal{L}_d$	::=	$\mathcal{L}(\mathcal{C}_e)$
(Property)	$\mathcal{P}$	::=	$\mathit{mk\_prp}(\mathcal{I})$
(Property Requests)	$\mathcal{R}$	::=	$\mathit{mk\_prp\_rq}(\mathcal{P})$
(Required Properties)	$\mathcal{R}_p$	::=	$\mathcal{S}(\mathcal{R})$
(Characters)	$\mathcal{I}_c$	::=	$\mathit{mk\_ch}(\mathcal{N})$
(Identifiers)	$\mathcal{I}_s$	::=	$\mathit{mk\_str}(\mathcal{I}_c, \mathcal{I}_s)$
(Hash Codes)	$\mathcal{H}$	::=	$\mathit{mk\_hcode}(\mathcal{N})$
(Lists)	$\mathcal{L}(\tau)$	::=	$\mathit{list\_nil}$
			$\mid \mathit{list\_cons}(\tau, \mathcal{L}(\tau)) \mid \mathit{list\_cat}(\mathcal{L}(\tau), \mathcal{L}(\tau))$
(Sets)	$\mathcal{S}(\tau)$	::=	$\mathit{set\_empty}$
			$\mid \mathit{set\_singleton}(\tau) \mid \mathit{set\_union}(\mathcal{S}(\tau), \mathcal{S}(\tau))$
(Key)	$\mathcal{K}$	::=	{"011F9290E8821202", "433C0555D4920035", … }
(Numbers)	$\mathcal{N}$	::=	$\{0, 1, 2, \ldots\}$
(Worldview)	$\mathcal{W}$	::=	{Alice, Bob, … }

Figure 4.4: Syntax of term constructors

(Predicates)	$\mathcal{F}$	::=	$\mathcal{F}_s \mid \mathcal{F}_{pm} \mid \mathcal{F}_{ax} \mid \mathcal{F}_{ct} \mid \mathcal{F}_a \mid \mathcal{F}_c \mid \mathcal{F}_{ce} \mid \mathcal{F}_{ci}$
			$\mid \mathcal{F}_p \mid \mathcal{F}_i \mid \mathcal{F}_m \mid \mathcal{F}_{mc} \mid \mathcal{F}_l \mid \mathcal{F}_{st} \mid \mathcal{F}_{eq}$
(Linking predicates)	$\mathcal{F}_s$	::=	$ok\_to\_link(\mathcal{M}_c, \mathcal{C}, \mathcal{L}_m, \mathcal{L}_d, \mathcal{R}_p)$
			$\mid export\_required\_prps(\mathcal{R}_p, \mathcal{C}_e)$
			$\mid provide\_enough\_lib(\mathcal{C}_i, \mathcal{L}_m, \mathcal{L}_d)$
			$\mid satisfy\_imports(\mathcal{C}_i, \mathcal{L}_d)$
			$\mid imprt\_match(\mathcal{R}_p, \mathcal{L}_d)$
			$\mid valid\_library(\mathcal{L}_m, \mathcal{L}_d)$
(Property match predicates)	$\mathcal{F}_{pm}$	::=	$has\_property(\mathcal{R}_p, \mathcal{C}_e)$
			$\mid has\_prp(\mathcal{R}, \mathcal{C}_e)$
(Axiom predicates)	$\mathcal{F}_{ax}$	::=	$key\_authority(\mathcal{W}) \mid keybind(\mathcal{W}, \mathcal{K})$
			$\mid prp\_authority(\mathcal{W})$
			$\mid prp\_server(\mathcal{W})$
			$\mid library\_dsc(\mathcal{M}_c, \mathcal{C})$
			$\mid signed(\mathcal{K}, \mathcal{F}_{ct})$
(Certificate predicates)	$\mathcal{F}_{ct}$	::=	$keybind(\mathcal{W}, \mathcal{K})$
			$\mid guarantees(\mathcal{W}, \mathcal{S}(R))$
			$\mid module\_dsc(\mathcal{M}_c, \mathcal{C})$
(Authentication predicates)	$\mathcal{F}_a$	::=	$signed\_component\_dsc(\mathcal{M}_c, \mathcal{C})$
			$\mid all\_signed(\mathcal{M}_c, \mathcal{C}, \mathcal{C}_e)$
			$\mid signed\_prp(\mathcal{M}_c, \mathcal{C}, \mathcal{P})$
			$\mid signed\_by\_auth(\mathcal{M}_c, \mathcal{C}, \mathcal{P})$
			$\mid valid\_sig\_prp\_auth(\mathcal{W}, \mathcal{R}_p)$
			$\mid valid\_sig\_component\_dsc(\mathcal{W}, \mathcal{M}_c, \mathcal{C})$
			$\mid keycert(\mathcal{W}, \mathcal{K})$
(Component predicates)	$\mathcal{F}_c$	::=	$component\_dsc\_valid(\mathcal{C})$
			$\mid component\_dsc\_eq(\mathcal{C}, \mathcal{C})$
			$\mid sub\_component\_dsc(\mathcal{C}, \mathcal{C})$
			$\mid sub\_export(\mathcal{C}_e, \mathcal{C}_e) \mid sub\_import\_list(\mathcal{C}_i, \mathcal{C}_i)$
(Component export predicates)	$\mathcal{F}_{ce}$	::=	$export\_valid(\mathcal{C}_e) \mid export\_eq(\mathcal{C}_e, \mathcal{C}_e)$
(Component import predicates)	$\mathcal{F}_{ci}$	::=	$import\_list\_valid(\mathcal{C}_i) \mid import\_list\_eq(\mathcal{C}_i, \mathcal{C}_i)$
(Property predicates)	$\mathcal{F}_p$	::=	$prp\_valid(\mathcal{P}) \mid prp\_eq(\mathcal{P}) \mid prp\_match(\mathcal{R}, \mathcal{P})$
(Identifier predicates)	$\mathcal{F}_i$	::=	$ide\_valid(\mathcal{I}_s)$
(Module predicates)	$\mathcal{F}_m$	::=	$module\_valid(\mathcal{M}) \mid module\_eq(\mathcal{M}, \mathcal{M})$
(Component module predicates)	$\mathcal{F}_{mc}$	::=	$cmodule\_valid(\mathcal{M}_c) \mid cmodule\_eq(\mathcal{M}_c, \mathcal{M}_c)$
(List predicates)	$\mathcal{F}_l$	::=	$list\_valid(EQ(\tau), \mathcal{L}(\tau))$
			$\mid list\_is\_nil(\mathcal{L}(\tau))$
			$\mid list\_member(\mathcal{L}(\tau), \tau)$
			$\mid list\_eq(EQ(\tau), \mathcal{L}(\tau), \mathcal{L}(\tau))$
(Set predicates)	$\mathcal{F}_{st}$	::=	$set\_member(\mathcal{S}(\tau), \tau) \mid set\_is\_empty(\mathcal{S}(\tau))$
(Equality predicates)	$\mathcal{F}_{eq}$	::=	$validper(EQ(\tau)) \mid validper\_refl(EQ(\tau), \tau)$

Figure 4.5: Syntax of predicates

and ***set_empty***. In the example, an import *alice_import_imprtCom1* consists of a property request standing for an imported component's name, a property request for imported type identifiers and a property request for a property *prpTypeSafety*.

- A software component usually consists of a set of modules rather than only one module, so a list of modules $\mathcal{M}_c$ is used for a component description. A module is constructed by using a term constructor ***mk_module*** out of a binary file name and its cryptographic hash code. The component from Alice consists of two modules: one module *alice_m1_dsc1* is defined at line 5 and the other module *alice_m2_dsc1* is defined at line 8.

- A property $\mathcal{P}$ is made out of a name of the property by using a term constructor ***mk_prp***.

- A property request $\mathcal{R}$ is constructed from a property by using a term constructor ***mk_prp_rq***.

- A public key $\mathcal{K}$ is a string composed by hexadecimal numbers.

- Principals in the Secure Linking logic (such as code consumers, code providers, and various authorities) are of type ***worldview***, which is the type defined in the PCA logic.

4.2 Inference Rules

Inference rules describe the relation between predicates. The inference rules define the interface of the Secure Linking logic. To build a linking proof, users (or provers) are guided by inference rules rather than delving into the details of the logic. All the inference rules are, in fact, lemmas of the Secure Linking logic, and they are all proved to be machine-checkable. They are also a selected set of lemmas to hide the details of the Secure Linking logic from the outside view.

$$\frac{\begin{array}{l} signed\_component\_dsc(alice\_mlist\_dsc1, alice\_dsc1) \\ provide\_enough\_lib(cdsc\_imprt\_list(alice\_dsc1, bob\_lib\_modules, bob\_lib\_exports) \\ export\_required\_prps(bob\_rprps, cdsc\_exprt(alice\_dsc1) \end{array}}{ok\_to\_link(alice\_mlist\_dsc1, alice\_dsc1, bob\_lib\_modules, bob\_lib\_exports, bob\_rprps)}$$

$$\frac{\dfrac{has\_property(bob\_rprps, alice\_export\_dsc1)}{export\_required\_prps(bob\_rprps, alice\_export\_dsc1)} \quad \begin{array}{c} \vdots \\ export\_eq(cdsc\_prps(alice\_dsc1), alice\_export\_dsc1) \end{array}}{export\_required\_prps(bob\_rprps, cdsc\_prps(alice\_dsc1))}$$

Figure 4.6: Alice's Secure Linking proof

4.2.1 Building a proof

Alice in the example must build a proof saying that her component exports all the properties required by the code consumer Bob. This can be done by showing that the set of modules and the component description satisfy the predicate ***ok_to_link*** with respect to the linking policy specified by Bob. Let's call the formula of the predicate ***ok_to_link*** applied by a set of modules and a component description an ***SL theorem*** The rule ***ok_to_link_i*** in Figure 4.7 is the inference rule used for building a linking proof which a code provider should submit. Figure 4.6 shows a couple of derivations from the proof of the SL theorem with Bob's linking policy and Alice's component description.

In the example of Alice and Bob, Alice should prove that her SL theorem `ok_to_link(alice_mlist_dsc1,alice_dsc1,bob_lib_modules,bob_lib_exports,bob_rprps)` holds. To prove this theorem, Alice must show that its three sub-theorems are satisfiable (by the rule ***ok_to_link_i***):

- `signed_component_dsc(alice_mlist_dsc1, alice_dsc1)`,
- `provides_enough_lib(cdsc_imprt_list(alice_dsc1), bob_lib_modules, bob_lib_exports)` and
- `exports_required_prps(bob_rprps, cdsc_prps(alice_dsc1))`

The second derivation in Figure 4.6 shows how to prove the third sub-theorem. Proving the theorem `exports_required_prps(bob_rprps, cdsc_prps(alice_dsc1))` is reduced to proving two sub-theorems `export_required_prps(bob_rprps, alice_export_dsc1)` and `export_eq(component_dsc_prps(alice_dsc1), alice_export_dsc1)` by the rule ***export_rprp_congr*** in Figure 4.7; and then, proving the first sub-theorem `export_required_prps(bob_rprps, alice_export_dsc1)` is reduced to another sub-theorem `has_property(bob_rprps, alice_export_dsc1)` by the inference rule ***export_required_prps_i*** in Figure 4.7.

Each theorem is reduced to smaller sub-theorems, and reduction continues until there exists no more sub-theorems. The derivation of her SL theorem is the proof Alice must submit to Bob with her component. After getting the component and the proof, Bob checks if the derivation is valid with a trusted proof checker; and if the derivation is valid, Bob allows the component to be linked to his system.

In the following subsections, I will show inference rules used for making linking decisions and authentication, and give explanation how they work.

4.2.2 Linking

The inference rules in the *linking* group are the rules for proving an SL theorem at the first step. They introduce formulas of the predicates *provide_enough_lib*, *export_required_prps*, and other predicates used for proving their premises. The inference rules for the predicate *signed_component_dsc* are explained in Section 4.2.3.

The predicate *export_required_prps* holds if a given export C_e has enough properties to satisfy all the property requests in a set R_p (by the rule ***export_required_prps_i***). The predicate *export_required_prps* also holds for an export C_e and a set R_p if the predicate holds for another export $C_e{}'$, which is *export_eq* to C_e, and a set R_p.

The predicate *provide_enough_lib* holds if given library information L_m and L_d satisfies the predicate *valid_library*, and the predicate *satisfy_imports* is proved with an import list C_i and the library information L_d by the rule ***provide_enough_lib_i***.

The predicate *satisfy_imports* holds if every import of an import list C_i, the first input

$$\frac{\begin{array}{l} \mathit{signed\_component\_dsc}\,(M_c, C) \\ \mathit{provide\_enough\_lib}\,(\mathit{component\_dsc\_import}(C), L_m, L_d) \\ \mathit{export\_required\_prps}\,(R_p, \mathit{component\_dsc\_prps}(C)) \end{array}}{\mathit{ok\_to\_link}\,(M_c, C, L_m, L_d, R_p)}\ \textbf{(ok\_to\_link\_i)}$$

$$\frac{\mathit{has\_property}\,(R_p, C_e)}{\mathit{export\_required\_prps}\,(R_p, C_e)}\ \textbf{(export\_required\_prps\_i)}$$

$$\frac{\begin{array}{l} \mathit{export\_eq}(C_e, {C_e}') \\ \mathit{export\_required\_prps}(R_p, {C_e}') \end{array}}{\mathit{export\_required\_prps}(R_p, C_e)}\ \textbf{(export\_rprps\_congr)}$$

$$\frac{\mathit{set\_is\_empty}\,(R_p)}{\mathit{has\_property}\,(R_p, C_e)}\ \textbf{(has\_property\_set\_empty)}$$

$$\frac{\mathit{has\_prp}\,(R, C_e)}{\mathit{has\_property}\,(\mathit{set\_singleton}\,(R), C_e)}\ \textbf{(has\_property\_set\_singleton)}$$

$$\frac{\begin{array}{c} \mathit{has\_property}\,(R_p, C_e) \\ \mathit{has\_property}\,({R_p}', C_e) \end{array}}{\mathit{has\_property}\,(\mathit{set\_union}\,(R_p, {R_p}'), C_e)}\ \textbf{(has\_property\_set\_union)}$$

$$\frac{\begin{array}{c} \mathit{valid\_library}\,(L_m, L_d) \\ \mathit{satisfy\_imports}\,(C_i, L_d) \end{array}}{\mathit{provide\_enough\_lib}\,(C_i, L_m, L_d)}\ \textbf{(provide\_enough\_lib\_i)}$$

$$\frac{\mathit{list\_is\_nil}\,(C_i)}{\mathit{satisfy\_imports}\,(C_i, L_d)}\ \textbf{(satisfy\_imports\_nil)}$$

$$\frac{\begin{array}{c} \mathit{imprt\_match}\,(R_p, L_d) \\ \mathit{satisfy\_imports}\,(C_i, L_d) \end{array}}{\mathit{satisfy\_imports}\,(\mathit{list\_cons}(R_p, C_i), L_d)}\ \textbf{(satisfy\_imports\_cons)}$$

$$\frac{\begin{array}{c} \mathit{satisfy\_imports}\,(C_i, L_d) \\ \mathit{satisfy\_imports}\,({C_i}', L_d) \end{array}}{\mathit{satisfy\_imports}\,(\mathit{list\_cat}(C_i, {C_i}'), L_d)}\ \textbf{(satisfy\_imports\_cat)}$$

Figure 4.7: Rules for linking

argument of the predicate finds its match in the list of library information L_d. Hence, it has three introducing rules, *satisfy_imports_nil*, *satisfy_imports_cons*, *satisfy_imports_cat.* The rules are reasoning about the cases when the import list is nil, when the import list is constructed by *list_cons*, and when the import list is constructed by *list_cat* respectively.

The predicate *has_property* takes two arguments of type $\mathcal{R}_p$ and $\mathcal{C}_e$, and it has three introducing rules according to the structure of the first argument. The predicate *has_property* holds if the first argument is a empty set (rule ***has_property_set_empty***), if the first argument is formed by using a term constructor *set_singleton* and the property request R is satisfied by the second argument C_e (rule ***has_property_set_singleton***), or if the first argument is formed by using a term constructor *set_union* and each of constituent sets is satisfied by the second argument C_e (rule ***has_property_set_union***).

4.2.3 Authentication

The inference rules in this category is focusing on introducing a formula formed by the predicate *signed_component_dsc.* The rule ***signed_component_dsc*** holds if a component description C is valid, and if there exists another component description C' which is equal to the component description C and it satisfies the predicate *all_signed* with the module list of C and the export properties of C.

A property authority WV is the authority who can make an assurance of a property P, if the formula *valid_sig_prp_auth(WV, P)* is provable. The predicate *valid_sig_prp_auth* holds if there exist a key authority WV' and a property server WV'', the key authority *says* the public key of the property server is K, and a property-authority certificate *guarantees(WV, P)* is signed by K (rule ***valid_sig_prp_auth_i***).

A component description C and its related modules M_c are considered as properly signed by a property authority WV if the formula *valid_sig_component_dsc(WV, M_c, C)* is provable. The predicate *valid_sig_component_dsc* holds if WV is a property authority, if there exists a key authority WV' who *says* the public key of the property authority

$$\frac{\begin{array}{l} component\_dsc\_valid\,(C) \\ component\_dsc\_eq\,(C, C') \\ all\_signed\,(M_c, C', cdsc\_prps\,(C)) \end{array}}{signed\_component\_dsc\,(M_c, C)}\ \textbf{(signed\_component\_dsc\_i)}$$

$$\frac{list\_is\_nil\,(C_e)}{all\_singed\,(M_c, C, C_e)}\ \textbf{(all\_signed\_nil)}$$

$$\frac{\begin{array}{l} prp\_eq\,(P, P') \\ signed\_prp\,(M_c, C, P') \\ all\_signed\,(M_c, C, C_e) \end{array}}{all\_signed\,(M_c, C, list\_cons(P, C_e))}\ \textbf{(all\_signed\_cons)}$$

$$\frac{\begin{array}{l} all\_signed\,(M_c, C, C_e) \\ all\_signed\,(M_c, C, {C_e}') \end{array}}{all\_singed\,(M_c, C, list\_cat(C_e, {C_e}'))}\ \textbf{(all\_signed\_cat)}$$

$$\frac{\begin{array}{l} sub\_component\_dsc\,(C, C') \\ signed\_by\_auth\,(M_c, C', P) \end{array}}{signed\_prp\,(M_c, C, P)}\ \textbf{(signed\_prp\_i)}$$

$$\frac{\begin{array}{l} valid\_sig\_prp\_auth\,(WV, R_p) \\ set\_member\,(R_p, R) \\ prp\_match\,(R, P) \\ valid\_sig\_component\_dsc\,(WV, M_c, C) \end{array}}{signed\_by\_auth\,(M_c, C, P)}\ \textbf{(signed\_by\_auth\_i)}$$

$$\frac{\begin{array}{l} key\_authority\,(WV') \\ prp\_server\,(WV'') \\ says\,(WV', (keybind\,(WV'', K))) \\ signed\,(K, guarantees\,(WV, P)) \end{array}}{valid\_sig\_prp\_auth\,(WV, P)}\ \textbf{(valid\_sig\_prp\_auth\_i)}$$

$$\frac{\begin{array}{l} prp\_authority\,(WV) \\ key\_authority\,(WV') \\ says\,(WV', keybind\,(WV, K)) \\ signed\,(K, module\_dsc\,(Mc, C)) \end{array}}{valid\_sig\_component\_dsc\,(WV, M_c, C)}\ \textbf{(valid\_sig\_component\_dsc\_i)}$$

$$\frac{\begin{array}{l} key\_authority\,(WV') \\ says\,(WV', (keybind\,(WV, K))) \\ signed\,(K, Fct) \end{array}}{says\,(WV, Fct)}\ \textbf{(says\_i)}$$

Figure 4.8: Rules for authentication

$$\frac{}{\mathit{keybind}\,(WV, K)}\ \textbf{(keybind\_i1)}$$

$$\frac{\mathit{key\_authority}\,(WV') \quad \mathit{says}\,(WV', (\mathit{keybind}\,(WV, K)))}{\mathit{keybind}\,(WV, K)}\ \textbf{(keybind\_i2)}$$

Figure 4.9: Rules for key binding

is K, and if K *signed* that the component description C is the proper description of the modules M_c (rule ***valid_sig_prp_auth_i***).

The inference rules in Figure 4.9 shows how to introduce a binding between a principal and its public key. The key binding of a trusted key authority is given as an axiom (rule ***keybind_i1***). Except for the key bindings of key authorities, all the key bindings should be introduced by using the rule ***keybind_i2***.

Chapter 5

SL Logic: Semantic Model

When building a system in logic, there are two different approaches: syntactic or semantic. In this chapter, I will briefly explain two approaches to achieve the same goal with a very simple example. Then I will explain the semantics of the Secure Linking logic using the semantic approach. I will also discuss the soundness of the Secure Linking logic later in this chapter.

5.1 Syntactic vs. Semantic Approach

Suppose that we are going to build a very, very simple system in logic, which compares two natural numbers and determines if the first number is greater than the second number.

Figure 5.1 shows the syntax of simple-gt system. Natural numbers can be defined inductively with *zero* and a *successor* function. There is only one expression in the simple-gt system which is evaluated as true if the first argument is greater than the second argument. The inference rules of simple-gt system is given at Figure 5.2. The rule ***gt-zero*** shows that `zero` is the smallest natural numbers. The rule ***gt-succ*** is used for comparing two successor numbers. A natural number `(succ x)` is greater than a natural number `(succ y)` if the number `x` is greater than the number `y`.

$$\begin{array}{llll} \textit{type} & \tau & ::= & \textit{nat} \mid \textit{expr} \\ \textit{nat} & n & ::= & \texttt{zero} \mid \texttt{succ}\ n \\ \textit{expr} & e & ::= & \texttt{gt}\ n\ n \end{array}$$

Figure 5.1: The syntax of simple-gt system

$$\frac{}{gt\ (succ\, n)\ zero}\ [\textbf{gt-zero}]$$

$$\frac{gt\ x\ y}{gt\ (succ\, x)\ (succ\, y)}\ [\textbf{gt-succ}]$$

Figure 5.2: Inference rules of simple-gt system

With these rules, it is possible to prove a natural number 3 (in the form of `(succ (succ (succ zero)))`) is greater than a natural number 1 (in the form of `(succ zero)`). The whole proof is given in Figure 5.3. First, the rule *gt-succ* is applied to reduce an expression `gt (succ (succ (succ zero))) (succ zero)` into an expression `gt (succ (succ zero)) zero`. The derivation is completed by applying the rule *gt-zero*.

For a logic system, it is useful to show that the system is consistent: *what is provable in the system is logically valid.* Therefore, it should not be possible to prove both some formula and its negation [19]. If it were possible, it would result in proving `false`, and out of the proof of `false`, any arbitrary formula could be proved. If a logic system we would build is not consistent, it is not of much use, because any formulas can be proved in this logic system. Soundness of a logic is more powerful property than the consistency of a logic. In a sound logic, any formula provable by the sound logic will be evaluated as true in any model of the logic [19]. Thus, the soundness of a logic implies the consistency of the logic.

There are two different ways to prove the consistency of a logic system: the syntactic

$$\frac{\dfrac{}{gt\ (succ\ (succ\ zero))\ zero}}{gt\ (succ\ (succ\ (succ\ zero)))\ (succ\ zero)}$$

Figure 5.3: A proof derivation of simple-gt system

$$\frac{}{gt\ \ (succ\,n)\ \ zero}\ \boldsymbol{gt\text{-}zero}$$

$$\frac{\dfrac{}{gt\ \ (succ\,x)\ \ zero}\ \boldsymbol{gt\text{-}zero}}{gt\ \ (succ\,(succ\,x))\ \ (succ\,zero)}\ \boldsymbol{gt\text{-}succ}$$

$$\frac{\begin{array}{c}\dfrac{}{gt\ \ (succ\,n)\ \ zero}\ \boldsymbol{gt\text{-}zero}\\ \vdots\qquad\qquad \boldsymbol{gt\text{-}succ}\\ gt\ \ (succ\,x)\ \ (succ\,y)\end{array}}{gt\ \ (succ\,(succ\,x))\ \ (succ\,(succ\,y))}\ \boldsymbol{gt\text{-}succ}$$

Figure 5.4: Proof trees for simple-gt system

and the semantic approaches. When using the syntactic approach, consistency is proved by induction on the length of proofs. The consistency proof reasons about all proofs that can be built from a given set of inference rules. To prove that the simple-gt system is logically consistent, all the proofs which can be generated by the inference rules in Figure 5.2 must be considered. Consider, for instance, a syntactic consistency theorem saying that a false formula `gt n (succ n)` is not provable for any natural number `n`. Figure 5.4 shows possible proof trees in the simple-gt system. Every proof tree ends with the rule ***gt-zero*** because this is the only rule requiring no premises to be proved in the simple-gt system. The proof of the consistency theorem is built by induction on the length of derivation (that is, on the number of inference rules applied) in the simple-gt system.

The base case is when the length of derivation is 1. Applying the rule ***gt-zero*** once is the only possible way to construct a proof of length 1, and the first proof tree in Figure 5.4 illustrates this case. It is obviously impossible to prove the consistency theorem for the rule ***gt-zero*** when the proof length is 1; thus, for the base case, the consistency theorem holds.

For the inductive step, suppose that the consistency theorem for the rule ***gt-zero*** holds when the proof length is k. A proof of length $k + 1$ in the simple-gt system must be constructed by applying the rule ***gt-succ*** k-times, and following applying the rule ***gt-***

```
t: tp.
nat : tp = (t arrow t) arrow (t arrow t).

zero : tm nat = lam [f] lam [x] x.
succ : tm (nat arrow nat) = lam [n] lam [f] lam [x] f @ (n @ f @ x).

gt : tm (nat arrow nat arrow form) =
  lam [a] lam [b]
    forall [r]
      (forall [n] r @ (succ @ n) @ zero) imp
      (forall2 [x][y] r @ x @ y imp r @ (succ @ x) @ (succ y)) imp
      r @ a @ p.

gt-zero : pf (gt @ (succ N) @ zero) = ...
gt-succ : pf (gt @ X @ Y) -> pf (gt @ (succ X) @ (succ Y)) = ...
```

Figure 5.5: The definition of simple-gt system

zero once. After applying the rule ***gt-succ***, the formula `gt n (succ n)` is reduced to a formula `gt n' (succ n')` where `n` $\equiv$ `(succ n')`.[1] So, if it were possible to prove the formula `gt n (succ n)` in the simple-gt system, after applying the inference rules $k+1$ times, it should be also possible to prove the formula `gt n' (succ n')` with applying the inference rules k times. By the induction hypothesis, it is impossible to prove the formula `gt n' (succ n')` with k steps, therefore it is impossible to prove the formula `gt n (succ n))` with $k+1$ steps. By induction, the consistency theorem holds.

Another approach, which we adopt for the Secure Linking logic, is the semantic one. With the semantic approach, operators in a logic system are defined by using an underlying logic, and the logic system is called an *object logic* of the underlying logic. Each logic term of an object logic is reduced to a term of the underlying logic; thus the soundness of the object logic depends on the soundness of the underlying logic.

The semantic specification of simple-gt system is shown at Figure 5.5. Suppose that we have a type `t` defined in an underlying logic, which has infinitely many elements. To

[1] If `n` is `zero`, there exists no applicable rules to the formula, and it contradicts the premise that the proof length is $k+1$.

model natural numbers, we used Church numerals [12]. The type `nat` is defined as a higher-order function, which returns a function from type `t` to type `t` when applied by an input function from type `t` to type `t`. A natural number is defined by the number of times a function `f` is composed; the natural number `zero` is defined as an identity function (that is, composed 0 times), and the successor function returns a function of type `nat` after composing a function `f` once more. The predicate `gt` is defined on top of natural numbers as illustrated in Figure 5.5.

With the definitions of `nat`, `zero`, `succ` and `gt`, the inference rules ***gt-zero*** and ***gt-succ*** are defined as lemmas shown in Figure 5.5. The complete definition and the proved lemmas of simple-gt system are given at Appendix B.

5.2 SL Logic Semantics

The Secure Linking logic adopts the semantic approach. Every operator is defined in terms of operators of PCA logic and its underlying higher-order logic. The Secure Linking logic consists of 65 definitions and 186 lemmas. The definitions must be trusted by a code consumer, therefore be included in Trusted Computing Base (TCB) of Secure Linking. The complete definition of the Secure Linking logic is given at Appendix C. The Secure Linking logic also contains lemmas which is considered as useful for proving Secure Linking theorems. All the lemmas in the Secure Linking logic are proved and the proofs amount to about 2720 lines in the underlying higher-order logic. Since the lemmas are not trusted, they should be checked by a trusted proof checker if they are used for proving Secure Linking theorems.

The underlying Core Logic on which the Secure Linking logic relies is defined as an object logic in Twelf [38]. The Twelf system is one of the implementations of the logical framework LF [17], which allows the specification of logics. Figure 5.6 shows some declarations. A metalogic (Twelf) type is a `type`, and an core-logic type is a `tp`.

```
tp: type.
tm: tp -> type.
form: tp.
arrow: tp -> tp -> tp.
pf: tm form -> type.
lam: (tm T -> tm U) -> tm (T arrow U).
@: tm (T arrow U) -> tm T -> tm U.
and: tm form -> tm form -> tm form.

and_i: pf A -> pf B -> pf (A and B) = ....
def_i: pf (B X) -> pf (lam B @ X) = ....
```

Figure 5.6: Core Logic definitions

Core Logic types are constructed from the type `form` of formulas of the Core Logic, and the `arrow` constructor. Core Logic level terms of type `T` has type `(tm T)` in the Twelf metalogic. Terms of type `(pf A)` are terms represented proofs of formula `A`.

The declarations beginning with `lam` introduce constants for constructing terms and formulas. Intuitively, `lam` is used for constructing a λ-expression of a higher-order logic, and `@` is used for applying Core Logic level terms to a function which is constructed by `lam`.

Figure 5.7 shows the definition of the predicate ***ok_to_link***; the predicate is defined using the Core Logic constructors `lam` and `and` as well as other SL predicates ***signed_component_dsc***, ***provide_enough_lib***, and ***export_required_prps***.

The lemma *ok_to_link_i* can be used for constructing a `pf` term of ***ok_to_link*** out of three `pf` terms. The lemma `def_i` is used to derive a `pf` term of the `lam` constructor, and the lemma `and_i` is used to derive a `pf` term of the `and` constructor (See Figure 5.6). Note that the lemma *ok_to_link_i* exactly looks like the inference rule ***ok_to_link_i*** in Figure 4.7.

Figure 5.8 shows the definitions of list constructors ***list_member*** and ***list_cat***, and the proof of a lemma *list_member_cat_i1*. Lists are defined in the Core Logic of the SL framework, consisting of constructors (such as *list_nil, list_cons, list_cat, sublist, list_mapfun,* and so on), predicates (such as *list_valid, list_length, list_is_nil,* and so on), and lemmas

```
ok_to_link: tm (module_list arrow component_dsc arrow
                (list module_list) arrow (list export) arrow
                (set prp_rq) arrow form) =
 lam [m] lam [cdsc] lam [lib] lam [libdsc] lam [rprps]
    signed_component_dsc @ m @ cdsc and
    provide_enough_lib @ (cdsc_imprt_list @ cdsc) @ lib @ libdsc and
    export_required_rprps @ (cdsc_exprt @ cdsc).

ok_to_link_i :
 pf (signed_component_dsc @ M @ Cdsc) ->
 pf (provide_enough_lib @ (cdsc_imprt_list @ Cdsc) @ Lib @ Libdsc) ->
 pf (export_required_prps @ Rprps @ (cdsc_exprt @ Cdsc)) ->
 pf (ok_to_link @ M @ Cdsc @ Lib @ Libdsc @ Rprps) =
 [p1][p2][p3]
 def_i (def_i (def_i (def_i (def_i (and_i p1 (and_i p2 p3)))))).
```

Figure 5.7: Secure Linking logic definition and lemma

```
list_member: tm (list T arrow set T) =
  lam2 [L][X] exists [I] list_nth @ L @ I @ X.

list_cat: tm (list T arrow list T arrow list T) =
 lam4 [L1][L2][I][X]
  exists [N]
   list_length @ L1 @ N and
   if (lt I N) (list_nth @ L1 @ I @ X) (list_nth @ L2 @ minus I N @ X).

list_member_cat_i1 :
 {n} pf (list_length @ L1 @ n) ->
 pf (list_member @ L1 @ X) ->
 pf (list_member @ (list_cat @ L1 @ L2) @ X) =
 [n][p2: pf (list_length @ L1 @ n)]
 [p1: pf (list_member @ L1 @ X)]
 exists_e (def2_e p1) [i][q1: pf (L1 @ i @ X)]
 cut (list_index_inrange p2 (list_nth_i q1)) [q2: pf (inrange @ n @ i)]
 def2_i (exists_i i
 (def4_i (exists_i n
  (and_i p2
   (if_i1 (inrange_e3 q2) q1))))).
```

Figure 5.8: SL logic definitions (continued)

describing the properties of lists. The Secure Linking logic uses only three constructors (*list_nil, list_cons and list_cat*), and a couple of predicates (*list_valid and list_is_nil*) among those constructors and predicates. The semantic definition of ***list_member*** says that a value `X` is an element of a list `L` if there *exists* an index `i` and `X` is the i^{th} element of the list `L`. A list constructor ***list_cat*** returns the i^{th} element of the fist list `L1` if the index i is smaller than the length `n` of `L1`; otherwise it returns the $(i-n)^{th}$ element of the second list `L2`.

The lemma ***list_member_cat_i1*** intuitively means that if a value `X` is a member of the list `L1`, then `X` is a member of the concatenated list from the lists `L1` and `L2`. The lemma takes two premises: the length of the list `L1` is `n` and the value `X` is a member of `L1`. The proof is constructed by proof constructors[2] in the Core Logic such as *exists_e, def_e, cut*, and so on. The complete proof of the lemma is given in Figure 5.8. With the semantic approach, the inference rules in the Secure Linking logic are coded as lemmas and all of the lemmas are proved.

5.3 Soundness of the SL Logic

With the syntactic approach, the soundness of a logic is typically proved by induction over all proofs that can be built from a given set of inference rules. However, with the semantic approach, the soundness of a logic can be proved in a different way. As already mentioned, PCA adopts a semantic model approach. That means each application-specific operator 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. This makes it easier to add new application-specific operators and rules as needed.

The soundness of the PCA logic has been proved relying on the soundness of its underlying higher-order logic. The PCA logic consists of a higher-order logic and ax-

[2] They are, in fact, proved lemmas in the Core Logic.

iomatic premises introduced by the predicate ***signed***, standing for cryptographic signatures. Bauer showed that the PCA logic is sound, and the way his access-control logic introduces a set of axiomatic premises (of the form ***signed****(A, F)*) does not compromise the soundness of the PCA logic [6, Section 3.3].

The same argument can be also applied to the soundness of the Secure Linking logic. We have chosen the PCA logic as the underlying logic of the Secure Linking logic, every term in the Secure Linking logic boils down to a term in higher-order logic, whose soundness was proved by Church [12]. With the semantic approach we adopt for the Secure Linking logic, the soundness of the underlying PCA logic and the higher-order logic guarantees the soundness of any lemma provable in the SL logic.

Adding a set of premises with no semantic definitions could make the higher-order logic inconsistent [6]. For example, if both F and $\neg F$ are added to the Secure Linking logic, one could derive `false`, and it would result in making it possible to prove any arbitrary software component is acceptable regardless of linking policies or signed certificates.

The Secure Linking logic have three different kinds of axioms: axioms built in the underlying logic, axioms added by trusted parties and and axioms added by untrusted parties. Axioms in the underlying logic are not of much interest when the soundness of the SL logic is discussed, because the set of axioms is fixed, and the soundness of the underlying logic including the axioms has been already proved.

The second group of axioms are introduced by trusted parties such as code consumers and the SL parsers. Three predicates are used for constructing axioms from trusted parties: *key_authority, property_server and library_dsc*. Axioms of those predicates are constructed from a code consumer's linking policy by the trusted SL parser. Axioms of the predicate *key_authority* is used for declaring key authorities who are trusted by a code consumer, and axioms of the predicate *property_server* is used for declaring trusted property servers. The predicate *library_dsc* is used for introducing the library information of a code consumer's to the Secure Linking logic. Those premises, however, must be generated by someone trusted (for example, a code consumer who establishes his own linking policy or trusted parsers in the framework), so it is not probable for the axioms in the second group having contradiction in order to interfere with the soundness of the Secure Linking logic.

The third group of axioms are introduced by untrusted parties such as code providers. The predicate ***signed*** in the PCA logic is the only constructor by which untrusted parties can introduce axioms into the Secure Linking logic. Bauer already showed that axioms introduced by ***signed*** cannot compromise the soundness of the PCA logic [6]. Therefore, axioms of the third group cannot cause any inconsistency in the Secure Linking logic.

Hence, adding a set of axioms that Secure Linking allows to add doesn't cause any inconsistency in the Secure Linking logic.

Chapter 6

Tactical Prover

We have developed a tactical prover in order to help code providers prove Secure Linking theorems. The tactical prover is a logic program running on the Twelf logical framework [38]. A Secure Linking theorem with the modules and the component description from a code provider is the goal to be proved, and axioms that are likely to be necessary in proving the SL theorem are defined before starting proving. The prover generates a derivation of the SL theorem; this is the proof that a code provider must send to a code consumer.

Code providers can build their own proofs from scratch, but it is not easy for someone who is not familiar with logic programming or with the Secure Linking logic to write down the proof by himself. The prover is a tool designed for making the proving process easy,

$$\frac{\begin{array}{l} \mathit{signed\_component\_dsc}(\mathit{alice\_mlist\_dsc}1, \mathit{alice\_dsc}1) \\ \mathit{provide\_enough\_lib}(\mathit{cdsc\_import}(\mathit{alice\_dsc}1, \mathit{bob\_lib\_modules}, \mathit{bob\_lib\_exports}) \\ \mathit{export\_required\_prps}(\mathit{bob\_rprps}, \mathit{cdsc\_exprt}(\mathit{alice\_dsc}1) \end{array}}{\mathit{ok\_to\_link}(\mathit{alice\_mlist\_dsc}1, \mathit{alice\_dsc}1, \mathit{bob\_lib\_modules}, \mathit{bob\_lib\_exports}, \mathit{bob\_rprps})}$$

$$\frac{\dfrac{\mathit{has\_property}(\mathit{bob\_rprps}, \mathit{alice\_export\_dsc}1)}{\mathit{export\_required\_prps}(\mathit{bob\_rprps}, \mathit{alice\_export\_dsc}1)} \quad \begin{array}{c} \vdots \\ \mathit{export\_eq}(\mathit{cdsc\_prps}(\mathit{alice\_dsc}1), \mathit{alice\_export\_dsc}1) \end{array}}{\mathit{export\_required\_prps}(\mathit{bob\_rprps}, \mathit{cdsc\_prps}(\mathit{alice\_dsc}1))}$$

Figure 6.1: Alice's Secure Linking proof (revisited)

$$\frac{\begin{array}{c} fp\_signed\_cdsc(W, M_c, C) \\ fp\_exp\_prps(R_p, C) \\ fp\_prv\_prps(C, L_m, L_d) \end{array}}{fp\_ok\_to\_link(W, M_c, C, L_m, L_d, R_p)} \; [R.1]$$

$$\frac{fp\_has\_property(R_p, C)}{fp\_exp\_prps(R_p, C)} \; [R.4]$$

$$\frac{\begin{array}{c} fp\_has\_property(R_p, C) \\ fp\_has\_property(R_p{}', C) \end{array}}{fp\_has\_propety(set\_union(R_p, R_p{}'), C)} \; [R.10]$$

$$\frac{fp\_has\_prp'(R, C)}{fp\_has\_property(set\_singleton(R), C)} \; [R.11]$$

$$\frac{fp\_emptyset(R_p)}{fp\_has\_property(R_p, C)} \; [R.12]$$

Figure 6.2: Rules in the SL tactical prover

but the tactical prover doesn't need to be trusted. Any proof generated by the prover should be verified by a trusted checker at link time, so any bugs causing the misbehavior of the prover will get caught when checking the resulting proofs.

In this chapter, I explain the structure of the tactical prover, and prove the soundness, the termination and the conditional completeness of the tactical prover.

6.1 Finding a proof

In Chapter 4, I showed how inference rules are used for proving a Secure Linking theorem. A proof of a Secure Linking theorem can be built out of inference rules and logic definitions by users, but writing a proof in formal logic could be overwhelming to users. Figure 6.1 shows the same proof tree given in Figure 4.6, and Figure 6.2 illustrates the rules related to the predicates ***ok_to_link*** ***export_required_prps*** and ***has_property***. With a SL theorem provided by users, the tactical prover starts to search a proof; the proving always starts with the rule [R.1] because it is the only rule applicable to a formula constructed by the predicate *ok_to_link*. By the rule [R.1], the goal is reduced to three different subgoals:[1] a subgoal `fp_signed_cdsc alice_mlist_dsc1 alice_dsc1`, a subgoal `fp_exp_prps bob_rprps alice_dsc1`, and a subgoal `fp_prv_prps alice_dsc1 bob_lib_modules bob_lib_exports`. Then the prover tries to find a proof of each subgoal.

For the subgoal `fp_exp_prps bob_rprps cdsc_prps(alice_dsc1)`, the prover has only one rule applicable, Rule [R.4]. By the rule [R.4], proving the goal `fp_exp_prps bob_rprps cdsc_prps(alice_dsc1)` is reduced to proving a subgoal `fp_has_property bob_rprps cdsc_prps(alice_dsc1)`. Figure 6.2 shows that there are 3 rules for the tactical ***fp_has_-property***. The structure of the first term (of type R_p) decides which rule should be applied to a given goal. Recall that the set of property requests from Bob, `bob_rprps` is constructed by set-unioning 4 singleton sets (in Figure 4.1). Hence, Rule [R.10] is chosen for proving this goal because `bob_rprps` of the goal is constructed by using the term constructor *set_union*. In this way, the prover searches a proof of a goal (that is, an SL theorem) until there exists no subgoal (in this case, the prover returns a derivation), or until there is no applicable rule (in this case, the prover reports a failure).

6.2 Tacticals and Tactics

The tactical prover of the Secure Linking framework consists of 41 tacticals and 73 tactics. When a prover searches for a proof, tactics are reducing goals to subgoals, and tacticals are providing primitives for combining tactics into larger ones that can give multiple proof-steps.

We designed the tactical prover in order that each tactical is related to at most one predicate; thereby a tactical finds proofs (or derivations) of formulas built only by the related predicate. On the other hand, every predicate used in the prover, except for the

[1] The subgoals means, informally, (1) the certificates describing the binding between `alice_mlist_dsc1` and `alice_dsc1` have digitally valid signatures, (2) the component description `alice_dsc1` exports all the properties specified in `bob_rprps alice_dsc1`, and (3) Bob provides enough library (represented by `bob_lib_modules` and `bob_lib_exports`) to satisfy the import requests of `alice_dsc1`.

Tactical	Predicate
ax_key_auth	key_authority
ax_key_bind	keybind
ax_prp_server	prp_server
ax_prp_auth	property_authority
ax_auth_sig_stmt	signed
ax_library_dsc	library_dsc
ax_cacl_hash	calc_hash

Table 6.1: Mapping between axiomatic tacticals and predicates

predicate *has_prp*, is related to only one tactical. That is, for a theorem formed by a given predicate, the proof of the theorem is derivable only by the related tactical in the prover. For example, as shown in Table 6.2, the predicate *ok_to_link* is related to only one tactical *fp_ok_to_link*, so trying to prove a Secure Linking theorem with the SL tactical prover would always end up with calling the tactical *fp_ok_to_link*. It is also possible to design a tactical prover of the same capability which has fewer tacticals, but we mapped each tactical to one predicate because this will make it easier to reason about the prover's properties.

The 41 tacticals of the Secure Linking prover are categorized into two groups:

- 7 tacticals which are used only for finding a proof from digitally signed axioms.
- 34 tacticals which are used for finding a proof of a formula, relying on the Secure Linking logic.

The digitally signed axioms are given by code consumers or authorities (key authorities, property authorities or property servers), and believed true if the digital signatures are valid. After being verified, the axioms are stored in the knowledge database of Twelf. Whenever a tactical in the first group is called, Twelf searches the knowledge database, and returns a matching proof if there exists any in the knowledge database. Tacticals in Table 6.1 comprise the first group. There are no tactics related to these tacticals because a goal with one of the axiom tacticals is provable only when the proof is already in the

Tactical	Predicate
fp_ok_to_link	ok_to_link
fp_prv_prps	provide_enough_lib
fp_exp_prps	export_required_prpps
fp_st_imprt	satisfy_imports
fp_st_imprt_cdsc	imprt_match
fp_has_property	has_property
fp_has_prp′	has_prp
fp_has_prp	has_prp
fp_valid_lib	valid_library
fp_signed_cdsc	signed_component_dsc
fp_all_signed	all_signed
fp_signed_prp	signed_prp
fp_signed_ma	signed_by_auth
fp_valid_sig_auth	valid_sig_prp_auth
fp_valid_sig_cdsc	valid_sig_component_dsc
fp_key_cert	keycert
fp_cdsc_valid	component_dsc_valid
fp_export_valid	export_valid
fp_import_valid	import_valid
fp_prp_match	prp_match
fp_prp_valid	prp_valid
fp_module_valid	module_valid
fp_cmodule_valid	cmodule_valid
fp_valid_chash	valid_chash
fp_valid_clist	valid_chash_list
fp_ide_valid	ide_valid
fp_ide_list_valid	list_valid(ide_eq)
fp_list_valid	list_valid
fp_list_member	list_member
fp_list_is_nil	list_is_nil
fp_set_member	set_member
fp_emptyset	set_is_empty
fp_validper	validper
fp_validper_refl	validper_refl

Table 6.2: Mapping between tacticals and prediccates

knowledge database of Twelf. A tactical in the second group breaks a goal into subgoals by using tactics, and tries to solve the subgoals by calling other tacticals. The success or failure of proving a goal depends on the success or failure of proving subgoals. Table 6.2 illustrates the tacticals in our tactical prover, and the mapping between tacticals and the predicates. The complete list of tactics and their related tacticals of the tactical prover is given at Appendix A.1.

The tactical prover in the Secure Linking framework is carefully designed to expedite the speed of finding a proof: by syntax-directness and by the order of tactics The prover is almost syntax-directed. A tactical prover is said to be syntax-directed if there exists only one tactic for a given input formula to the prover. For only a few formulas, the tactical prover has more than one matching tactics. In the case of multiple matching tactics, the prover tries the tactics one by one in the order that the tactics are defined. If the tactic fails, the prover back-tracks to the points it branched, and then tries the next tactics. Backtracking is one of the notorious factors which can make a tactical prover slow and impractical. We designed the prover in order to make it as syntax-directed as possible; only 6 pairs out of 73 tactics cause backtracking,[2] but for a given formula, there are at most two matching tactics in the Secure Linking prover. For example, the tactical *fp_list_member* has 4 related tactics. As explained in Chapter 4, the Secure Linking logic provides two list constructors: *list_cat* and *list_cons*. If a list is constructed from two input lists by using *list_cat*, a member of the list is a member of the first input list or a member of the second input list. The tactic [R.57] and the tactic [R.58] in Appendix A.1 are used for solving this case. First, the prover chooses the tactic [R.57], and will succeed if an element in question is a member of the first list in the concatenated list. Otherwise, the prover fails, and then backtracks to apply the tactic [R.58] instead. The success of proving at this time depends on whether or not the element in question is a member of the second list in the concatenated list. Another pair of tactics, [R.59] and [R.60], also

[2] Let's call these pairs *backtracking* pairs.

causes backtracking. These tactics are used for testing membership of a list constructed by *list_cons*. For a given formula, the tactical prover calls tactics in one of the two pairs, not in both of them. It is because the input domain of a backtracking pair is orthogonal to each other. Hence, backtracking happens only within a backtracking pair, not across a backtracking pair.

To speed up the proving process, the order of tactics is also considered. In the Secure Linking prover, the order of tactics doesn't affect whether a given formula is provable by the prover or not. The success of proving is completely independent of the order how the tactics are arranged. But proving can be accelerated if the tactics are carefully arranged. For example, when reasoning about the validity of a list, there are 3 cases to consider, and they result in 3 tactics in the prover related to the tactical *fp_list_valid*. A list is valid if two input lists of a concatenated list are valid ([R.54]), if a list is formed by using the constructor *list_cons* with a valid head element and a valid tail list ([R.55]), or if the list is nil ([R.56]). In our tactical prover, we put the tactic for a nil list at the last. The rule [R.56] is applied only when a given input formula doesn't match the two previous rules; this cuts off time-consuming evaluation and successive backtracking. Otherwise, every input list would be evaluated to see if the list is nil before applying the other rules.

6.3 Soundness

By showing the soundness of a tactical prover, 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.

Appel and Felty [4] 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 the Secure Linking prover is easily seen to be sound by the method of Appel & Felty.

Although a dependent type system is useful for showing the soundness of the Secure Linking prover, it doesn't guarantee that the prover is complete. The completeness of a prover, hence, should be proved in a different way. Later in this chapter, I will prove the conditional completeness of the Secure Linking prover. Proving soundness and the conditional completeness ensures the conditional consistency of the Secure Linking prover.

6.4 Termination

Proving the termination of the tactical prover of the Secure Linking framework is useful because it guarantees that the prover returns a result, regardless of an input. That means, for any formula the prover returns a derivation of the input formula or reports a failure without looping forever.

Twelf, on which the prover runs, provides a termination checker [38, 39]. Although, in general, termination is undecidable, it is possible to verify if a given type family, when interpreted as a logic program, always terminates on well-moded goals. The termination checker of Twelf requires a logic program being checked to be well-moded.

In logic-programming-language terminology, mode information refers the information about which arguments to a predicate should be considered input and output. To be well-moded, every argument of a predicate must be assigned its mode, and when the predicate is called, the input arguments and the output arguments of the predicate must be ground.[3]

The mode declaration of a few tacticals in the tactical prover is shown in Figure reffig:prover:mode. An argument of a predicate is assigned an identifier, and augmented by '+' if the argument is input, '–' if it is output, or '$*$' if its use is unrestricted. For example, the tactical ***fp_ok_to_link*** has 6 arguments, and all of them are inputs. In case of the tactical ***fp_valid_sig_auth***, it has 3 arguments; the first two arguments are inputs, but

[3] In Twelf terminology, *ground* objects are objects not containing any existential variables.

```
%mode    fp_ok_to_link      +CC +M +Dsc +L +Lc +R.
%mode    fp_exp_prps        +R +E.
%mode    fp_prv_prps        +I +L +Lc.

%mode    fp_valid_sig_auth  +CC +Ma -Rprps.
%mode    fp_valid_sig_cdsc  +CC -Ma +M +C.
```

Figure 6.3: Mode declaration of the Secure Linking tactical prover

```
%terminates     Ma        (fp_valid_sig_auth _ Ma _).
%terminates     C         (fp_valid_sig_cdsc _ _ _ C).

%terminates     I         (fp_prv_prps I _ _).
%terminates     [R E]     (fp_exp_prps R E).
%terminates     D         (fp_ok_to_link _ _ D _ _ _).
```

Figure 6.4: Termination declaration of the prover

the last argument is output. The mode of the Secure Linking tactical prover is given at Appendix A.2. It shows that all the tacticals in the prover are well-moded without any unrestricted use. That means, every argument in the tactical prover is input or output of a tactical.

In order for a Twelf logic program to be termination-checked, the mode information of the program of interest should be declared and be checked before termination is checked. And the termination declaration of the program must be given to the Twelf termination checker. Termination checking is based on lexicographic or simultaneous subterm ordering by programmers. The termination declaration specifies which arguments or predicates are associated to termination via call patterns.[4] During termination checking, the checker verifies if every termination argument of a predicate are reduced to its subterms whenever the predicate is called. If so, it becomes a simple sufficient condition to guarantee the termination of the logic program.

Figure 6.4 shows the termination declaration of the tacticals whose modes are given in

[4] Let's call these arguments *termination* arguments.

Figure 6.3. The tactical ***fp_valid_sig_auth*** has one termination argument, `Ma`;[5] whenever this tactical is called, a term for `Ma` is reduced to its subterm. The termination declaration also shows that the tactical ***fp_exp_prps*** has two lexicographic termination arguments, `R` and `E`. Therefore, whenever ***fp_exp_prps*** is called, either a term for `R` or a term for `E` is reduced to its own subterm.

The tactical prover of the Secure Linking framework is carefully designed to be well-moded and termination-checked. The mode of the prover is declared, and the termination call patterns are specified; it results in an automated, machine-generated proof that the prover always terminates. The full mode declaration and termination declaration of the Secure Linking prover is given at Appendix A.2.

6.5 Completeness

We also proved that the tactical prover in the Secure Linking framework is conditionally complete. We know from Gödel's incompleteness theorem, general higher-order logic is not complete. That means, in general higher-order logic, it is not always possible to find a proof of a true formula. Hence, it might not be possible to prove the completeness of the Secure Linking logic, which is based on a higher-order logic. It is, however, still possible to show that the Secure Linking tactical prover is complete, because we are considering only a subset of a general higher-order logic. In general, proving the completeness of a logic guarantees that there exists a derivation for every true formula formed by the logic. The completeness I discuss here is *conditional* in the sense that the prover always finds a proof (or derivation) of every true formula which is generated by the trusted parsers discussed in Chapter 3. The set of true formulas which the conditional complete prover covers is a subset of true formulas which the Secure Linking logic can express, and it is quite useful to prove the completeness of the prover on the set of true formulas generated

[5] In a termination declaration, the arguments, which have no effect on termination, can be omitted by using *underscore*.

by the Secure Linking parsers; it guarantees that the prover will not fail in finding a proof of a correct theorem originated from a code consumer's linking policy and a code provider's component description.

The formulas generated by the trusted parsers have restriction as follows:

- The congruence rules using equivalence relations (such as *export_eq*, *list_eq*, etc.) are removed from the prover.

- A list of library component modules and a list of library component descriptions are made out of the list constructor *list_cons*.

- Exports or import lists are constructed by using both *list_cons* and *list_cat*, but cannot be extracted from component descriptions (for example, by using a term constructor *component_dsc_exprt* or *component_dsc_imprt_list*).

- Module lists must be formed by using *list_cons*. The list constructor *list_cat* is not allowed to be used. Note that the list of modules are not concatenated or combined in any other way when combining two component descriptions. Two component descriptions must have the same list of modules to get combined.

The Secure Linking logic has congruence rules for equivalence relations such as *export_eq* or *list_eq*. Although congruence rules enrich the expressive power of the SL logic, they might prevent the prover (which is, in fact, a logic program) from terminating.[6] In the Secure Linking prover, we restricted the use of congruence rules in order to guarantee the termination of the prover. Instead, the exact match of terms is used for determining the equality of terms. However, this is not a critical loss in the expressive power of the SL logic, because the terms in a Secure Linking theorem are generated by the same trusted parsers either in the proving phase (at a code provider's site) or in the checking phase (at a code consumer's site), resulting in the exactly same structure.

[6] That means, the prover may loop forever without returning a derivation successfully or without reporting a failure.

The Secure Linking logic has two list constructors, *list_cons* and *list_cat*, and extensively uses lists in its definitions of Secure Linking formulas. The trusted parsers use restricted list constructors in the cases enumerated above to reduce the number of rules in the Secure Linking prover, and to improve the speed of the prover. Restricting list constructors does not weaken the expressive power of the SL logic since the equivalence between a list constructed by *list_cons* and a list constructed by *list_cat* can be easily proved; this restriction only cuts off a few proving steps, making the SL prover faster.

The ultimate goal of the prover is to prove Secure Linking theorems, that is, to find a derivation of a formula constructed by *ok_to_link* if it is true; Otherwise the prover must report failure.

Table 6.2 shows that the tactical *fp_ok_to_link* is the only applicable tactical to formulas formed by the predicate *ok_to_link*. Hence, the completeness of the prover can be stated as follows.

Proposition 1 *The tactical fp_ok_to_link finds a derivation for every parser-generated true formula constructed by the predicate ok_to_link.*

We will outline the proof of Proposition 1 here; the full proof is provided in Appendix A.3.

The completeness of a tactical depends on the completeness of the dependent tacticals for proving subgoals. To prove that applying every true formula generated by the Secure Linking parsers to the tactical *fp_ok_to_link* always results in a correct derivation, we have to show that each subsequent call to another tactical always returns a correct derivation. We can prove the completeness of a tactical by climbing up the call tree of subsequent tacticals with showing the completeness of each subsequent tactical one by one.

The pattern how a tactical calls other tacticals comes down to one of the following three cases:

- Tacticals with no tactics
- Tacticals with a tactic, but without any subgoals

- Tacticals calling only other tacticals to prove subgoals
- Tacticals making recursive calls

The conditional completeness of the Secure Linking prover can be proved by showing that all the tacticals of each group are conditionally complete. The tacticals of the first group are axiomatic tacticals discussed in Section 6.2. They are used for finding an axiom stored in Twelf's knowledge database. For a given formula, it is true as an axiom if the formula is believed as true by a code consumer, and stored in a prover's knowledge database. In other words, if a parsed formula is true as an axiom, the formula must reside in Twelf's knowledge database, and must be retrievable when the Secure Linking prover asks. This proves the conditional completeness of axiomatic tacticals. Table 6.1 shows 7 axiomatic tacticals of the prover.

The tacticals in the second group don't have any subgoals, so they work as terminating points of proof search in the prover. The 3 tacticals, *fp_list_is_nil, fp_emptyset* and *fp_validper*, fall into this category. The conditional completeness of these tacticals depends on whether or not the related tactics of a tactical in this group cover all the possible true input formulas. The complete proof is given at Appendix A.3.1.

The third group consists of tacticals each of which has subgoals to prove, but doesn't make any recursive calls to itself. The 14 tacticals, *fp_ok_to_link, fp_exp_prps, fp_signed_prp, fp_signed_ma, fp_valid_sig_auth, fp_valid_sig_cdsc, fp_key_cert, fp_export_valid, fp_import_-valid, fp_prp_match, fp_prp_valid, fp_module_valid, fp_valid_chash*, and *fp_validper_refl*, are included in this group. The conditional completeness of these tacticals rely on the conditional completeness of dependent tacticals (or subsequently called tacticals) and the completeness of the input domain covered by the tactics of each tactical. The complete proof for this group is given at Appendix A.3.2.

The remaining 17 tacticals fall into the fourth group. They have subgoals to prove, and each of them makes recursive calls to itself. The tacticals, *fp_prv_prps, fp_st_imprt,* *fp_st_imprt_cdsc, fp_has_property, fp_has_prp′, fp_has_prp, fp_valid_lib, fp_signed_cdsc, fp_all_signed, fp_cdsc_valid, fp_cmodule_valid, fp_valid_clist, fp_ide_valid, fp_ide_list_valid, fp_list_valid, fp_list_member*, and *fp_set_member*, make up the fourth group. The completeness proof is built by induction on the structure of input formulas of each tacticals, and/or induction by cases of term constructors used for input formulas. The complete proof for this group is given at Appendix A.3.3.

Chapter 7

Beyond SL logic

A strength of the the Secure Linking logic is that it can be extended to encode application-specific details. In this chapter I will explain how to extend the Secure Linking logic in order to represent some typical properties for linking in the Secure Linking logic. I will also discuss, in Chapter 8, that the Secure Linking logic is expressive enough for users to model a real world linking system by adding small extension to the logic.

Another strength is that the Secure Linking logic can interoperate with other application-specific logics concerning system security. Since they are built in the same higher-order logic, existing examples of application-specific logics are Foundational Proof-Carrying Code (FPCC) and the public key infrastructure using the PCA logic.

7.1 Extending the Secure Linking logic

The Secure Linking logic lies in a hierarchy of logic; it is based on the Proof-Carrying Authentication logic and core logic which encodes basic structures such as sets or lists. These logics are defined on top of a general higher-order logic. Figure 7.1 illustrates the hierarchy of those logics. The figure also shows that the Secure Linking logic can be extended in the same way to encode basic linking properties and real world linking

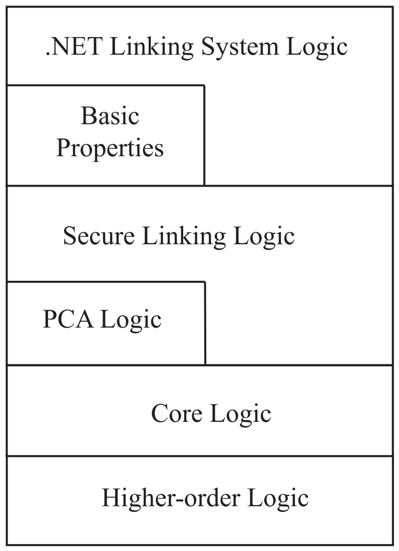

Figure 7.1: Hierarhcy of logics

systems.

The Secure Linking logic defines very general and abstract properties and property requests. Users have freedom and responsibility to write property definitions and their matching property requests with suitable semantics; but it would be not easy for users who are not used to logic or logic programming to write down property-request pairs by themselves. Hence, to support general users of the Secure Linking framework, it is necessary to choose a set of basic properties and a set of basic property requests and to include them as a part of the framework.

At link time, linkers usually check the names of components and the types of exported/imported identifiers. It is essential to support these functionalities in the Secure Linking framework if we want linkers based on the Secure Linking framework to work as an ordinary linker as well as to check more sophisticated properties.

We chose three core properties the Secure Linking framework should support:

- Names of components

$$\begin{array}{llll}
\text{(Property)} & \mathcal{P} & ::= & \mathit{mk\_prp}(\mathcal{I}) \\
 & & & \mid \mathit{mk\_prp\_cname}(\mathcal{I}) \\
 & & & \mid \mathit{mk\_prp\_export\_ids}(\mathcal{L}(\mathcal{I})) \\
 & & & \mid \mathit{mk\_prp\_code\_hash}(\mathcal{M}_c) \\
\text{(Property Requests)} & \mathcal{R} & ::= & \mathit{mk\_prp\_rq}(\mathcal{P}) \\
 & & & \mid \mathit{rq\_cname\_exists} \\
 & & & \mid \mathit{mk\_rq\_cname}(\mathcal{I}) \\
 & & & \mid \mathit{rq\_export\_ids\_exists} \\
 & & & \mid \mathit{rq\_code\_hash\_checkable} \\
\text{(Module predicates)} & \mathcal{F}_m & ::= & \mathit{module\_valid}(\mathcal{M}) \mid \mathit{module\_eq}(\mathcal{M}, \mathcal{M}) \\
 & & & \mid \mathit{is\_module\_fname}(\mathcal{M}, \mathcal{I}) \\
 & & & \mid \mathit{is\_module\_hcode}(\mathcal{M}, \mathcal{H}) \\
 & & & \mid \mathit{valid\_chash}(\mathcal{M}) \\
\text{(Component module predicates)} & \mathcal{F}_{mc} & ::= & \mathit{cmodule\_valid}(\mathcal{M}_c) \\
 & & & \mid \mathit{valid\_chash\_list}(\mathcal{M}_c) \\
\text{(Axiom predicates)} & \mathcal{F}_{ax} & ::= & \mathit{key\_authority}(\mathcal{W}) \mid \ldots \mid \mathit{calc\_hash}(\mathcal{I}, \mathcal{H})
\end{array}$$

Figure 7.2: Added term constructors and predicates

- Identifiers exported from/imported to a component
- Cryptographic hash codes of binary modules

The added term constructors and predicates to the Secure Linking logic are shown in Figure 7.2, and new inference rules in Figure 7.3. Three new constructors are introduced for the type *property*: *mk_prp_cname*, *mk_prp_export_ids*, and *mk_prp_code_hash*. A term constructor *mk_prp_cname* builds a property term out of an identifier, standing for a component name. A term constructor *mk_prp_export_ids* constructs a property term from a list of identifiers standing for exported type identifiers (such as class names, structure names, or function names). A term constructor *mk_prp_code_hash* is used for encoding cryptographic hash codes of a component's binary modules into a property term.

Coupling with newly introduced properties, four property request constructors are added to the extension of the Secure Linking logic. Term constructors *rq_cname_exists* and *mk_rq_cname* build property requests related to the component name property. A term constructor *rq_export_ids_exist* is used for the exported identifier property, and *rq_code_hash_*-

$$\frac{\mathit{ide\_valid}(I)}{\mathit{prp\_match}(\mathit{rq\_cname\_exists}, \mathit{mk\_prp\_cname}(I))}\ \textbf{(pmatch\_cname\_i1)}$$

$$\frac{\mathit{ide\_eq}(I, I')}{\mathit{prp\_match}(\mathit{mk\_rq\_cname}(I), \mathit{mk\_prp\_cname}(I'))}\ \textbf{(pamtch\_cname\_i2)}$$

$$\frac{\begin{array}{c}\mathit{list\_valid}(\mathit{ide\_eq}, L(I)) \\ \neg \mathit{list\_is\_nil}(L(I))\end{array}}{\mathit{rq\_export\_ids\_exists}(\mathit{mk\_prp\_export\_ids}(L(I))}\ \textbf{(pmatch\_ids\_i)}$$

$$\frac{\begin{array}{c}\mathit{cmodule\_valid}(M_c) \\ \mathit{valid\_chash\_list}(M_c)\end{array}}{\mathit{rq\_code\_hash\_checkable}(\mathit{mk\_prp\_code\_hash}(M_c)}\ \textbf{(pmatch\_chash\_i)}$$

$$\frac{\mathit{list\_is\_nil}(M_c)}{\mathit{valid\_chash\_list}(M_c)}\ \textbf{(valid\_chash\_list\_nil)}$$

$$\frac{\begin{array}{c}\mathit{valid\_chash}(M) \\ \mathit{valid\_chash\_list}(M_c)\end{array}}{\mathit{valid\_chash\_list}(\mathit{list\_cons}(M, M_c))}\ \textbf{(valid\_chash\_list\_cons)}$$

$$\frac{\begin{array}{l}\mathit{is\_module\_fname}(M, I) \\ \mathit{is\_module\_hcode}(M, H) \\ \mathit{calc\_hash}(I, H)\end{array}}{\mathit{valid\_chash}(M)}\ \textbf{(valid\_chash\_i)}$$

Figure 7.3: Added inference rules

checkable is used for the cryptographic hash code property of modules.

We designed a property request as a predicate accepting a property term, and reduced the decision process of property matching to a simple predicate evaluation. It is also possible to define more than one property requests for one property. Owing to our semantic model approach, users can assign different semantics to each property request. This makes the property matching of the Secure Linking framework more flexible.

For example, the two different property requests for the component name property in extended SL logic explain this flexibility. Usually there are two cases a component name is concerned about: users might want to check if a software component has a name without caring what the component name is. In the other case, however, users might want to check if a component has the exact name they wants. That means, users need two different ways of checking one property; this can be accomplished by providing two different property requests *rq_cname_exists* and *mk_rq_cname* for the component name property.

The inference rule ***pmatch_cname_i1*** in Figure 7.3 is used to check if an input property is of type *property*. The predicate *prp_match* holds if an input property is built by using the term constructor *mk_prp_cname*. It doesn't matter what exact component name the argument has. On the other hand, the inference rule ***pmatch_cname_i2*** in Figure 7.3 is used for exact matching. The predicate *prp_match* holds if and only if an input property is built from the same identifier (that is, a component name) to the identifier used for building the property request.

Secure Linking also verifies cryptographic hash codes of a component's binary modules to check if binary modules are tampered with after the component description has been built. This requirement is coded as a property request by using a term constructor *rq_code_hash_checkable*. The inference rules ***pmatch_chash_i***, ***valid_chash_list_nil***, ***valid_chash_list_cons*** and ***valid_chash_i*** shows how the hash code checking works. The property request *rq_code_hash_checkable* is satisfied if an input list of modules has valid cryptographic hash codes (rule ***pmatch_chash_i***), and a list of modules is considered to have valid cryptographic hash codes if every element of the list has a valid cryptographic hash code (rule ***valid_chash_list_nil*** and rule ***valid_chash_list_cons***).

7.2 Interoperating with other logics

7.2.1 Foundational proof-carrying code

Foundational Proof-Carrying Code (FPCC) is a proof-carrying code framework [2], in which safety proofs about machine-language programs can be stated and proved with respect to a minimum set of axioms. Safety properties such as memory safety ("this program accesses only addresses in a certain range") and type safety ("this program respects its Java interfaces") can be established in and enforced by FPCC.

FPCC logic and FPCC system are useful in building an extended Secure Linking system. For example, in the extended SL system, a code consumer may ask all incoming foreign software components to have FPCC proofs showing that they are obeying the safety theorem of FPCC. In this case, the FPCC logic and certifying compilers supporting FPCC can be considered as property authorities, whose certificates are written in logic, and machine-checkable. The certificates from FPCC (in fact, the safety proofs written in FPCC logic) are verified by a trusted proof checker; if valid, the Secure Linking framework generates an axiom constructed by the predicate *signed*. It is just like generating an axiom of the *signed* predicate after verifying digital signatures.

7.2.2 Public Key Infrastructure

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. For example, the security model of the .NET framework supports several standard public key certificate formats including X.509.

The Secure Linking framework requires all the certificates from property authorities to be digitally signed, and has intense use of signature verification modules and public key certificates. Those functionalities could be embedded in the linking logic, but tying public-key infrastructure so closely to the logic results in a less flexible system than expected; future users with a different PKI will not be able to fit in the linking logic without extending the logic, and not be able to take advantage of the soundness of the Secure Linking logic. For the scalability of key certificates, The Secure Linking framework separates authenticating with public key certificates from the secure linking logic in a more modular way.

To address principal-public key bindings, the Secure Linking framework has a built-in formula constructor *keybind* for translating various key certificate formats into a formula in the linking logic. The predicate *keybind* takes two arguments: the name of a principal, and its public key. The formula holds if and only if there exists a binding between the principal and the key.

Several different key-distribution protocols, such as SPKI or X.509, can be expressed as definitions (and machine-checked lemmas) within the PCA logic [3]. The PCA logic has the same underlying higher-order logic as the Secure Linking framework, and it enables Secure Linking to make use of almost any public-key protocol in a sound and secure way with no mismatching of protocol interfaces.

Chapter 8

Case Study

In this chapter, I will show the Secure Linking logic is general and expressive enough to express the linking protocol of the .NET framework. I also discuss what we learn about the properties of .NET while we're giving a formal specification of its linking procedure.

8.1 Overview

The .NET framework is a computing platform developed by Microsoft targeting the highly distributed environment of the Internet [40]. The main components of .NET are the Common Language Runtime and its class library.

The Common Language Runtime (CLR) uses Java-style bytecode verification, and has a new configuration-management unit called an *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.

CLR is responsible for execution-time management such as memory management, thread management and remote procedure calls. It could be compared to the Java Virtual Machine [23]. CLR provides its own intermediate language called Microsoft Intermediate Language (MSIL) and programs must be compiled to this intermediate language to be executed on CLR. CLR also implements a strict type- and code-verification infrastructure called Common Type System (CTS) and supports Just-In-Time (JIT) compiling for enhancing performance.

The .NET framework class library has a collection of reusable classes that tightly integrate with the Common Language Runtime. It provides a variety of classes, from the basic classes for graphical user interface to the more sophisticated classes and tools for development and consumption of Web services supporting the standards such as SOAP (standard object access protocol), XML (an extensible mark-up language).

8.1.1 Assemblies

An assembly is the logical unit of a program executable on the common language runtime. It is also a unit of security, a unit of type and a unit of version within the .NET framework, as well as a deployment unit during runtime execution. An assembly consists of four elements: the assembly manifest, type metadata, MSIL code, and a set of resources (such as `.bmp` or `.jpg` files).

An assembly manifest contains a collection of data that describes how the elements in an assembly relate to each other. This metadata includes the name of the assembly, version number, its cultural background (such as language), list of all files in the assembly, type reference information, and information on referenced assemblies. Assembly developers can add or change some information in the assembly manifest by putting assembly attributes declaratively in their source codes. An assembly manifest is created automatically by compilers or programming tools supporting the .NET framework. .NET framework provides an MSIL disassembler to view MSIL information in a file. If the file being examined is an assembly, this information can include the assembly's attributes, as well as references to other modules and other assemblies.

```
⟨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 8.1: A .NET configuration file

8.1.2 Versioning

An assembly is used as the unit of linking deployment and execution in the .NET framework. When a developer builds an application, the main assembly of the application is linked to other reference assemblies, each of which is identified by using the information such as the name of the assembly, the version number, the cultural information, and so on. However, sometimes the developer wants the application to run against a newer version of an assembly. .NET supports redirecting assembly versions through configuration files. Configuration files and decision procedures for version redirection will be discussed in the next section.

.NET also supports side-by-side execution. This is the ability to run multiple versions of assemblies of the same name simultaneously. Support for side-by-side storage and execution of different versions of the same assembly is an integral part of versioning, and is built into a part of the runtime. Treating an assembly's version number as part of its identity enables the runtime to store multiple versions of assemblies under the same name and distinguish them at run time.

8.2 Versioning

8.2.1 Version redirection in .NET

.NET framework treats an assembly's version number as part of the assembly's identity; this enables assemblies of the same name to co-exist within one system, and gives more control to developers or system administrators at link time. The runtime of the .NET framework allows developers or system administrators to specify the version of an assembly to bind and to choose a different version of an assembly of the same name at link time. In order to redirect binding of assemblies users can specify it in configuration files at different levels. Figure 8.1 illustrates an example of configuration files. Application configuration files, machine configuration files, and publisher policy files are used to redirect one version of an assembly to another. 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 .NET framework. The .NET linker decides which version of an assembly to bind with the following steps: first, the linker checks the original assembly reference to determine what version was 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.

8.2.2 Formal specification in the Secure Linking logic

In Chapter 7 I showed how to represent some basic properties such as component names as properties and property requests in the secure linking logic. In the same way, version information and redirection requests are coded as *properties* and *property requests*.

The linking decision procedure of the .NET framework is translated into a set of

$$\frac{\neg isempty(VP) \quad inrange(VP.range, V)}{ver\_policy\_effective(VP, V)} \ \textbf{(vp\_effective)}$$

$$\frac{ver\_policy\_effective(VP_{mch}, V_{org}) \quad version\_match\_policy(VP_{mch}, V_{new})}{ver\_redir(V_{org}, VP_{mch}, VP_{pub}, VP_{app}, V_{new})} \ \textbf{(machine\_redir)}$$

$$\frac{\neg ver\_policy\_effective(VP_{mch}, V_{org}) \quad ver\_policy\_effective(VP_{pub}, V_{org}) \quad version\_match\_policy(V_{pub}, V_{new})}{ver\_redir(V_{org}, VP_{mch}, VP_{pub}, VP_{app}, V_{new})} \ \textbf{(publisher\_redir)}$$

$$\frac{\neg ver\_policy\_effective(VP_{mch}, V_{org}) \quad \neg ver\_policy\_effective(VP_{pub}, V_{org}) \quad ver\_policy\_effective(VP_{app}, V_{org}) \quad version\_match\_policy(VP_{app}, V_{new})}{ver\_redir(V_{org}, VP_{mch}, VP_{pub}, VP_{app}, V_{new})} \ \textbf{(app\_redir)}$$

$$\frac{\neg ver\_policy\_effective(VP_{mch}, V_{org}) \quad \neg ver\_policy\_effective(VP_{pub}, V_{org}) \quad \neg ver\_policy\_effective(VP_{app}, V_{org}) \quad version\_match\_simple(V_{org}, V_{new})}{ver\_redir(V_{org}, VP_{mch}, VP_{pub}, VP_{app}, V_{new})} \ \textbf{(no\_redir)}$$

Figure 8.2: Rules for version redirection

$$\frac{\begin{array}{l} asm\_hash\_code(Asm, H) \\ valid\_hash\_code(Asm, H) \end{array}}{signed\_sname(Asm)} \ \textbf{(signed\_with\_self\_key)}$$

$$\frac{\begin{array}{l} key\_authority(WV) \\ says(WV, keybind(WV', K)) \\ signed(K, Asm) \end{array}}{signed\_certs(Asm)} \ \textbf{(signed\_with\_certs)}$$

$$\frac{signed\_sname(Asm) \bigvee signed\_certs(Asm)}{signed\_asm(Asm)} \ \textbf{(signed\_asembly)}$$

Figure 8.3: Rules for signing assemblies

inference rules in the secure linking logic, shown in Figure 8.2. The versions of two assemblies may match if they are identical *(ver_match_simple)*, or through a redirection (rule ***ver_match_policy***). The redirection may come from local configuration files (rule ***machine_redir***), from the software developer (rule ***publisher_redir***), or from the component integrator (rule ***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, it is provable that the predicate *ver_policy_effective* holds. The inference rule ***machine_redir*** shows the case when a version policy VP_{mch} from a machine configuration file is effective. If the version V of a target assembly matches the new version in the policy VP_{mch}, then 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.

8.3 Signing Assemblies

The runtime system of .NET requires that every assembly has a strong name [26]. 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 advised to sign the assembly with the private key corresponding to the public key stored in its assembly manifest. A code producer can sign an assembly in two different ways: with key certificates obtained from third-party key authorities or without key certificates. After signing an assembly the resulting signature is stored in its assembly manifest. Then, .NET runtime verifies the digital signature of an assembly using the public key stored in its assembly manifest (and key certificates if provided).

In formalizing the signed assembly feature of .NET, we were unable to prove a standard theorem in the Secure Linking framework that every public key is certified by at least one key authority. It turns out that .NET uses keys that need not be certified when it signs an assembly without key certificates from third-party key authorities. 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. Especially this kind of assembly signing doesn't guarantee the *non-repudiation* property which is commonly expected from digital signatures. In other words, 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.

Signing with a self-announcing public key doesn't provide more trust than hash verification does. Verifying a digital signature with a public key in an assembly manifest only guarantees that the assembly has not been tampered with after being signed. 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 the assembly signing in the Secure Linking logic, we replace signing on an assembly with a self-announcing public key by simple hash code verification. Figure 8.3 shows the inference rules of signing assemblies. An assembly is considered signed validly if it is signed with verified public key (rule ***signed_with_certs***), or if it has a valid cryptographic hash code (rule ***signed_with_self_key***).

8.4 Discussion

Secure Linking proposed to solve the weakness of the traditional code signing protocol: the traditional code signing protocol is not expressive enough for the current heavily connected computing environment. The .NET framework has been developed to give an answer to the same problem because Microsoft suffered from the system vulnerability caused by their less expressive code signing protocol as quoted in Section 1.1. Although Secure Linking and the .NET framework have been developed independently without knowing each other's existence, they aim for the same goal: to improve the expressive power of code signing. The .NET framework chose to support version information of a software component and to provide a way for users to specify version-specific configuration at link time. This approach works fine with solving the previously quoted problem successfully.

Compared to the .NET framework, the approach that Secure Linking provides is more general. Users can specify informally the properties they want to enforce (such as version information, verifiable cryptographic hash code, and so on), and certificates from one third-party authority is signed only on a property the authority can guarantee. As demonstrated in this chapter, Secure Linking is still expressive enough to encode .NET's version-redirection into an application-specific extension of Secure Linking.

Another thing I want to mention is the advantage of formal specification. We could find a small problem of code signing protocol Microsoft proposed while trying to give a formal specification to .NET's linking procedure. The problem of .NET's assembly signing protocol is small, but it could cause a system hole, or a bug in users' applications if the application designers understand the limitation and implication of the assembly signing protocol thoroughly. It is not a good design decision to leave a weakness in a system with expecting no bad things to happen We could get closer insight to the system while we tried to describe the system formally. This case study shows that the formalization of an existing system is a quite useful tool to analyze the system.

Chapter 9

Implementation

We have built a prototype Secure Linking system using the Secure Linking framework. Its system diagram is given in Figure 9.1. The prototype system implements a Java class loader running Sun Microsystem's Java virtual machine. It consists of: an XML parser for the linking policy description language, an XML parser for the component description language, the Secure Linking logic augmented by basic property and property requests, a tactical prover, a proof checker, and a Java class loading module.

9.1 Java Class Loaders

The class loader concept, one of the cornerstones of the Java virtual machine, describes the behavior of converting a named class into the bits responsible for implementing that class. Because class loaders exist, the Java runtime does not need to know anything about files and file systems when running Java programs [24]. The Java environment also allows users to provide customized class loaders. A user class loader gets the chance to load a class before the built-in original class loader does. Because of this, it can load the class implementation data from some alternate source (such as source repositories connected by the HTTP protocol), or it can perform some actions enhancing the security of a code

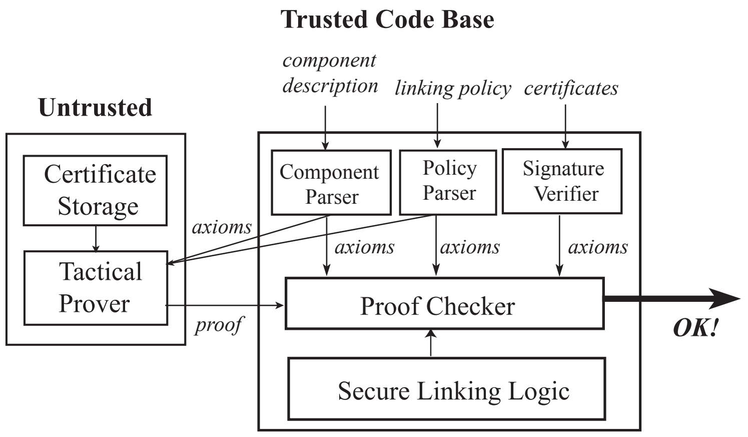

Figure 9.1: Prototype implementation

consumer's system. In the Secure Linking prototype system, we designed a class loader which checks a Secure Linking proof of a class file before the class is loaded.

An instance of the SL class loader is created with a linking policy of a code consumer. The linking policy is specified by using the description language explained in Chapter 3. If the SL class loader gets a request to load a class, it checks if the given binary class file comes with a component description in the SL component description language and a SL proof; if not, the class loader rejects the request. To check the SL proof from a code provider, the linking policy and the component description are translated into axioms in the Secure Linking logic by the trusted parsers of the framework. Beside trusted parsers, the SL class loader has a trusted digital signature verifier which produces axioms formed by the ***signed*** predicate after verifying the cryptographic signatures on the certificates. Currently, the SL class loader uses the Java package **java.security** for the trusted signature verifier. We chose the X.509 format [11] for public key certificates of the prototype SL class loader, SHA-1 [30] for the message digest algorithm, and DSA [33] for the digital signature algorithm.

After getting the axioms, the SL class loader starts the trusted proof checker; the Secure Linking logic and the axioms generated by the SL parsers and the trusted signature verifier are loaded into the checker. The SL proof from a code provider is verified by the loaded checker. If the proof is valid, the class of the request is loaded into the Java virtual machine; otherwise the request is rejected.

9.2 Proof Checker

Currently we're using the Twelf logical framework for code consumers to verify proofs from code providers. Twelf provides many useful features for theorem proving and proof verification, and they are very helpful when building a prototype system.

The proof checker in a secure linking system is an essential part of the trusted computing base (TCB), so a bug in the proof checker can be a security hole in the larger system. Twelf itself is a large program, and it has a problem of increasing the size of TCB. This problem can be avoided if we include only the core part of proof checking within the TCB rather than trust a software with all the redundant functionalities. Appel et al. [5] built a small trustworthy proof checker, which consists of only 800 lines of code. Their proof checker reduces the size of TCB significantly, and would be useful for linking systems using the Secure Linking framework.

Given a proof from a code provider, a code consumer must be able to verify the validity of the proof. The SL framework is built on the Twelf logical framework [38]. The Twelf system is one of the implementations of the logical framework LF [17], which allows the specification of logics as mentioned before. Since the Secure Linking logic is written on top of PCA, an object logic of LF, every term in the Secure Linking logic boils down to a term in the underlying LF logic. Therefore, the proof provided by a code provider 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 of the term. If the term is well typed, then the derivation is correct; hence, a code provider has succeeded in proving the proposition. If the proof of a code provider is checked, a secure linker of the SL framework will allow the component with the proof to be linked.

9.3 Certificate Storage

A few signed certificates are involved in Secure Linking when proving a SL theorem and checking a SL proof. Since a code provider is responsible for presenting certificates as well as an SL proof to a code consumer, getting certificates from various authorities could be a burden to a code provider. A code provider has collected lots of certificates for previous SL proofs, but she may not be sure if the certificates in need are in her collection or not. Therefore, a tool which manages digitally signed certificates and searches a queried certificate is quite useful for making the proving process easy.

Certificate management systems need not be trusted because signature verification or reasoning about security at link time would be done other components in the TCB of the Secure Linking framework. A certificate management tool in Secure Linking could be a file cabinet with printed certificates or a sophisticated database system. For the prototype system, we implemented a simple database which is based on a keyed hash table. However, there are some alternatives for the certificate storage; beyond database systems, some certificate management systems concern the system security when retrieving certificates. For example, Jim proposed a trust management system consisting of a high-level policy language, a local policy evaluator, and a certificate retrieval system, in which every answer of a query is checked if the answer follows the established security policy [16].

Chapter 10

Conclusion

I have shown a flexible way of establishing linking policies to ensure system security at link time. In this way we can give more expressive power to traditional code signing protocol.

I have proposed and implemented a logical framework supporting Secure Linking. 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 the Secure Linking framework, a code consumer announces its linking policy to protect its system from malicious code from outside. The policy can include properties, for example component names, valid hash code of programs, version information, and so on, which code consumers thinks important for the 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 Secure Linking logic consists of basic formula constructors and inference rules based on the logic of Proof-Carrying Authentication (PCA). Linking decision procedures of a code consumer, system-specific linking policies, and description of software components of code producers are translated into the linking logic. A proof of Secure Linking is formed out of the SL logic, and checked by a trusted proof checker in a code consumer. If the accompanying proof is valid, 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 the Secure Linking logic general and flexible. The Secure Linking logic is expressive enough to encode the linking procedures of a real-world linking system (the .NET framework). Due to this expressiveness, it is possible to encode various security models into the Secure Linking logic, and to enable different security models to interoperate conveniently. Trying to give a formal description of a real-world system gives us insight into the system. While trying to describe the .NET linking procedures in formal logic, we found that .NET's protocol of signing *assemblies* is not well-defined, and it might cause a weakness in a user's system.

Secure Linking is a way a code consumer can require high-level properties from outside software components and check the exported properties of a software component with certificates from third party authorities. Secure Linking enhances the security of a system during linking time, giving more control to a code consumer for linking decisions.

Appendix A

Tactical Prover

A.1 Tacticals and Tactics

This section shows the complete list of tactics of the Secure Linking tactical prover. See Chapter 6 for a detailed discussion about the prover.

Tactical fp_ok_to_link

$$\frac{\begin{array}{l} fp\_signed\_cdsc(W, M_c, C) \\ fp\_exp\_prps(R_p, C) \\ fp\_prv\_prps(C, L_m, L_d) \end{array}}{fp\_ok\_to\_link(W, M_c, C, L_m, L_d, R_p)} \; [R.1]$$

Tactical fp_prv_prps

$$\frac{\begin{array}{ll} fp\_cdsc\_valid(C) & fp\_prv\_prps(C, L_m, L_d) \\ fp\_cdsc\_valid(C') & fp\_prv\_prps(C', L_m, L_d) \end{array}}{fp\_prv\_prps(component\_dsc\_combine(C, C'), L_m, L_d)} \; [R.2]$$

$$\frac{\begin{array}{l} fp\_list\_valid(C_i) \\ fp\_valid\_lib(L_m, L_d) \\ fp\_st\_imprt(C_i, L_d) \end{array}}{fp\_prv\_prps(mk\_component\_dsc(C_e, C_i), L_m, L_d)} \; [R.3]$$

Tactical fp_exp_prps

$$\frac{fp\_has\_property(R_p, C)}{fp\_exp\_prps(R_p, C)} \; [R.4]$$

Tactical fp_st_imprt

$$\frac{\begin{array}{c} fp\_st\_imprt(C_i, L_d) \\ fp\_st\_imprt(C_i', L_d) \end{array}}{fp\_st\_imprt(list\_cat(C_i, C_i'), L_d)}\ [R.5]$$

$$\frac{\begin{array}{c} fp\_st\_imprt\_cdsc(R_p, L_d) \\ fp\_st\_imprt(C_i, L_d) \end{array}}{fp\_st\_imprt(list\_cons(R_p, C_i), L_d)}\ [R.6]$$

$$\frac{fp\_list\_is\_nil(C_i)}{fp\_st\_imprt(C_i, L_d)}\ [R.7]$$

Tactical fp_st_imprt_cdsc

$$\frac{\begin{array}{c} fp\_export\_valid(C_e) \\ fp\_has\_property(R_p, C_e) \end{array}}{fp\_st\_imprt\_cdsc(R_p, list\_cons(C_e, L_d))}\ [R.8]$$

$$\frac{\begin{array}{c} fp\_list\_valid(L_d) \\ fp\_st\_imprt\_cdsc(R_p, L_d) \end{array}}{fp\_st\_imprt\_cdsc(R_p, list\_cons(C_e, L_d))}\ [R.9]$$

Tactical fp_has_property

$$\frac{\begin{array}{c} fp\_has\_property(R_p, C) \\ fp\_has\_property({R_p}', C) \end{array}}{fp\_has\_propety(set\_union(R_p, {R_p}'), C)}\ [R.10]$$

$$\frac{fp\_has\_prp'(R, C)}{fp\_has\_property(set\_singleton(R), C)}\ [R.11]$$

$$\frac{fp\_emptyset(R_p)}{fp\_has\_property(R_p, C)}\ [R.12]$$

Tactical fp_has_prp′

$$\frac{fp\_has\_prp'(R, C)}{fp\_has\_prp'(R, component\_dsc\_combine(C, C'))}\ [R.13]$$

$$\frac{fp\_has\_prp'(R, C')}{fp\_has\_prp'(R, component\_dsc\_combine(C, C'))}\ [R.14]$$

$$\frac{\begin{array}{c} fp\_export\_valid(C_e) \\ fp\_has\_prp(R, C_e) \end{array}}{fp\_has\_prp'(R, mk\_component\_dsc(C_e, C_i))}\ [R.15]$$

Tactical fp_has_prp

$$\frac{fp\_has\_prp(R, C_e)}{fp\_has\_prp(R, list\_cat(C_e, {C_e}'))}\ [R.16]$$

$$\frac{fp\_has\_prp(R, {C_e}')}{fp\_has\_prp(R, list\_cat(C_e, {C_e}'))}\ [R.17]$$

$$\frac{fp\_prp\_valid(P) \quad fp\_prp\_match(R, P)}{fp\_has\_prp(R, list\_cons(P, C_e))}\ [R.18]$$

$$\frac{fp\_list\_valid(C_e) \quad fp\_has\_prp(R, C_e)}{fp\_has\_prp(R, list\_cons(P, C_e))}\ [R.19]$$

Tactical fp_valid_lib

$$\frac{\begin{array}{c} ax\_library\_dsc(M_c, C_e) \\ fp\_export\_valid(C_e) \\ fp\_cmodule\_valid(M_c) \\ fp\_valid\_lib(L_m, L_d) \end{array}}{fp\_valid\_lib(list\_cons(M_c, L_m), list\_cons(C_e, L_d))}\ [R.20]$$

$$\frac{\begin{array}{c} fp\_list\_is\_nil(L_m) \\ fp\_list\_is\_nil(L_d) \end{array}}{fp\_valid\_lib(L_m, L_d)}\ [R.21]$$

Tactical fp_signed_cdsc

$$\frac{\begin{array}{c} fp\_singed\_cdsc(W, M_c, C) \\ fp\_singed\_cdsc(W, M_c, C') \end{array}}{fp\_signed\_cdsc\,(W, M_c, component\_dsc\_combine(C, C'))}\ [R.22]$$

$$\frac{\begin{array}{l} fp\_export\_valid(C_e) \\ fp\_import\_valid(C_i) \\ fp\_all\_signed(W, M_c, mk\_component\_dsc(C_e, C_i), C_e) \end{array}}{fp\_singed\_cdsc(W, M_c, mk\_component\_dsc(C_e, C_i))}\ [R.23]$$

Tactical fp_all_signed

$$\frac{\begin{array}{c} fp\_signed\_prp(W, M_c, C, P) \\ fp\_all\_singed(W, M_c, C, C_e) \end{array}}{fp\_all\_signed(W, M_c, C, list\_cons(P, C_e))}\ [R.24]$$

$$\frac{fp\_list\_is\_nil(C_e)}{fp\_all\_signed(W, M_c, C, C_e)}\ [R.25]$$

Tactical fp_signed_prp

$$\frac{\begin{array}{l} fp\_cdsc\_valid(C) \\ fp\_signed\_ma(W, M_c, C, P) \end{array}}{fp\_signed\_prp(W, M_c, C, P)}\ [R.26]$$

Tactical fp_signed_ma

$$\frac{\begin{array}{l} fp\_valid\_sig\_cdsc(W, W', M_c, C) \\ fp\_valid\_sig\_auth(W, W', R_p) \\ fp\_set(R_p, R) \\ fp\_prp\_match(R, P) \end{array}}{fp\_signed\_ma(W, M_c, C, P)}\ [R.27]$$

Tactical fp_valid_sig_auth

$$\frac{\begin{array}{l} ax\_prp\_server(W, W_{ps}) \\ fp\_key\_cert(W, W_{ps}, K) \\ ax\_auth\_sig\_stmt(K, guarantees(W', R_p)) \end{array}}{fp\_valid\_sig\_auth(W, W', R_p)}\ [R.28]$$

Tactical fp_valid_sig_cdsc

$$\frac{\begin{array}{l} ax\_prp\_auth(W, W') \\ fp\_key\_cert(W, W', K) \\ ax\_auth\_sig\_stmt(K, module\_dsc(M_c, C)) \end{array}}{fp\_valid\_sig\_cdsc(W, W', M_c, C)}\ [R.29]$$

Tactical fp_key_cert

$$\frac{\begin{array}{l} ax\_key\_auth(W, W_{ca}) \\ ax\_key\_bind(W_{ca}, K_{ca}) \\ ax\_auth\_sig\_stmt(K_{ca}, keybind(W', K)) \end{array}}{fp\_key\_cert(W, W', K)}\ [R.30]$$

Tactical fp_cdsc_valid

$$\frac{\begin{array}{c} fp\_cdsc\_valid(C) \\ fp\_cdsc\_valid(C') \end{array}}{fp\_cdsc\_valid(component\_dsc\_combine(C, C'))}\ [R.31]$$

$$\frac{\begin{array}{c} fp\_export\_valid(C_e) \\ fp\_import\_valid(C_i) \end{array}}{fp\_cdsc\_valid(mk\_component\_dsc(C_e, C_i))}\ [R.32]$$

Tactical fp_export_valid

$$\frac{fp\_list\_valid(C_e)}{fp\_export\_valid(C_e)}\ [R.33]$$

Tacitcal fp_import_valid

$$\frac{fp\_list\_valid(C_i)}{fp\_import\_valid(C_i)}\ [R.34]$$

Tactical fp_prp_match

$$\frac{fp\_prp\_valid(P)}{fp\_prp\_match(mk\_prp\_rq(P), P)}\ [R.35]$$

$$\frac{}{fp\_prp\_match(rq\_cname\_exists, mk\_prp\_cname(I))}\ [R.36]$$

$$\frac{fp\_ide\_valid(I)}{fp\_prp\_match(mk\_rq\_cname(mk\_prp\_cname(I)), mk\_prp\_cname(I))}\ [R.37]$$

$$\frac{fp\_ide\_list\_valid(L(I))}{fp\_prp\_match(rq\_export\_ids\_exists, mk\_prp\_export\_ids(L(I)))}\ [R.38]$$

$$\frac{\begin{array}{c}fp\_valid\_clist(M_c)\\ fp\_cmodule\_valid(M_c)\end{array}}{fp\_prp\_match(rq\_code\_hash\_checkable, mk\_prp\_code\_hash(M_c))}\ [R.39]$$

Tactical fp_prp_valid

$$\frac{fp\_ide\_valid(I)}{fp\_prp\_valid(mk\_prp(I))}\ [R.40]$$

$$\frac{fp\_ide\_valid(I)}{fp\_prp\_valid(mk\_prp\_cname(I))}\ [R.41]$$

$$\frac{fp\_ide\_list\_valid(L(I))}{fp\_prp\_valid(mk\_prp\_export\_ids(L(I)))}\ [R.42]$$

$$\frac{fp\_cmodule\_valid(M_c)}{fp\_prp\_valid(mk\_prp\_code\_hash(M_c))}\ [R.43]$$

Tactical fp_module_valid

$$\frac{fp\_ide\_valid(I)}{fp\_module\_valid(mk\_module(I))}\ [R.44]$$

Tactical fp_cmodule_valid

$$\frac{\begin{array}{c} fp\_module\_valid(M) \\ fp\_cmodule\_valid(M_c) \end{array}}{fp\_cmodule\_valid(list\_cons(M, M_c))} \; [R.45]$$

$$\frac{fp\_list\_is\_nil(M_c)}{fp\_cmodule\_valid(M_c)} \; [R.46]$$

Tactical fp_valid_clist

$$\frac{\begin{array}{c} fp\_valid\_chash(M) \\ fp\_valid\_clist(M_c) \end{array}}{fp\_valid\_clist(list\_cons(M, M_c))} \; [R.47]$$

$$\frac{fp\_list\_is\_nil(M_c)}{fp\_valid\_clist(M_c)} \; [R.48]$$

Tactical fp_valid_chash

$$\frac{\begin{array}{c} ax\_calc\_hash(I, H) \\ fp\_ide\_valid(I) \end{array}}{fp\_valid\_chash(mk\_module(I, H))} \; [R.49]$$

Tactical fp_ide_valid

$$\frac{\begin{array}{c} fp\_ide\_valid(I_c) \\ fp\_ide\_valid(I_s) \end{array}}{fp\_ide\_valid(mk\_str(I_c, I_s))} \; [R.50]$$

$$\frac{}{fp\_ide\_valid(mk\_ch(N))} \; [R.51]$$

Tactical fp_ide_list_valid

$$\frac{\begin{array}{c} fp\_ide\_list\_valid(L(I)) \\ fp\_ide\_valid(I) \end{array}}{fp\_ide\_list\_valid(list\_cons(I, L(I)))} \; [R.52]$$

$$\frac{fp\_list\_is\_nil(L(I))}{fp\_ide\_list\_valid(L(I))} \; [R.53]$$

Tactical fp_list_valid

$$\frac{\begin{array}{c} fp\_list\_valid(EQ(\tau), L(\tau)) \\ fp\_list\_valid(EQ(\tau), L(\tau)') \end{array}}{fp\_list\_valid(EQ(\tau), list\_cat(L(\tau), L(\tau)'))} \; [R.54]$$

$$\frac{\begin{array}{c} fp\_validper\_refl(EQ(\tau), t) \\ fp\_list\_valid(EQ(\tau), L(\tau)) \end{array}}{fp\_list\_valid(EQ(\tau), list\_cons(t, L(\tau)))} \; [R.55]$$

$$\frac{\begin{array}{l} fp\_list\_is\_nil(L(\tau)) \\ fp\_validper(EQ(\tau)) \end{array}}{fp\_list\_valid(L(\tau))} \; [R.56]$$

Tactical fp_list_member

$$\frac{fp\_list\_member(L(\tau), t)}{fp\_list\_member(list\_cat(L(\tau), L(\tau)'), t)} \; [R.57]$$

$$\frac{fp\_list\_valid(L(\tau)', t)}{fp\_list\_member(list\_cat(L(\tau), L(\tau)'), t)} \; [R.58]$$

$$\frac{fp\_validper\_refl(Eq(\tau), t)}{fp\_list\_member(list\_cons(t, L(\tau)), t)} \; [R.59]$$

$$\frac{\begin{array}{c} fp\_list\_member(L(\tau), t') \\ fp\_list\_valid(EQ(\tau), L(\tau)) \end{array}}{fp\_list\_member(list\_cons(t, L(\tau)), t')} \; [R.60]$$

Tactical fp_list_is_nil

$$\frac{}{fp\_list\_is\_nil(list\_nil)} \; [R.61]$$

Tactical fp_set_member

$$\frac{fp\_set\_member(S(\tau), x)}{fp\_set\_member(set\_union(S(\tau), S(\tau)'), x)} \; [R.62]$$

$$\frac{fp\_set\_member(S(\tau)', x)}{fp\_set\_member(set\_union(S(\tau), S(\tau)'), x)} \; [R.63]$$

$$\frac{}{fp\_set\_member(set\_singleton(x), x)} \; [R.64]$$

Tactical fp_emptyset

$$\frac{}{fp\_emptyset(set\_empty)} \; [R.65]$$

Tactical fp_validper

$$\frac{}{fp\_validper(module\_eq)} \; [R.66]$$

$$\frac{}{fp\_validper(prp\_eq)}\ [R.67]$$

$$\frac{}{fp\_validper(cmodule\_eq)}\ [R.68]$$

$$\frac{}{fp\_validper(export\_eq)}\ [R.69]$$

Tactical fp_valdiper_refl

$$\frac{fp\_module\_valid(M)}{fp\_validper\_refl(module\_eq, M)}\ [R.70]$$

$$\frac{fp\_prp\_valid(P)}{fp\_validper\_refl(prp\_eq, P)}\ [R.71]$$

$$\frac{fp\_cmodule\_valid(M_c)}{fp\_validper\_refl(cmodule\_eq, M_c)}\ [R.72]$$

$$\frac{fp\_export\_valid(C_e)}{fp\_validper\_refl(export\_eq, C_e)}\ [R.73]$$

A.2 Termination Checking

A.2.1 Mode Declaration

Mode declaration in Twelf asks that every argument of a predicate must be decorated with its mode [37]. An argument of a predicate is assigned an identifier, and augmented by '+' if the argument is input, '−' if it is output, or '∗' if its use is unrestricted.

Twelf checks explicit mode declaration by the programmer against the signature and signals if the prescribed information flow is violated.

The full mode declaration of Secure Linking tactical prover is given below. This mode declaration passes Twelf's mode checking; thus the prover is well-moded.

```
%mode   fp_ok_to_link         +CC +M +Dsc +L +Lc +R.
%mode   fp_prv_prps           +I +L +Lc.
%mode   fp_exp_prps           +R +E.
%mode   fp_valid_lib          +L +LC.
%mode   fp_st_imprt           +Ilist +LibDsc.
```

```
%mode   fp_st_imprt_cdsc      +I +L.

%mode   fp_has_property       +R +E.

%mode   fp_has_prp            +R +E.

%mode   fp_has_prp'           +R +C.

%mode   fp_cdsc_valid         +C.

%mode   fp_export_valid       +X.

%mode   fp_signed_cdsc        +CC +M +C.

%mode   fp_all_signed         +CC +M +C +E.

%mode   fp_signed_prp         +CC +M +C +P.

%mode   fp_signed_ma          +CC +M +C +P.

%mode   fp_valid_sig_auth     +CC +Ma -Rqset.

%mode   fp_valid_sig_cdsc     +CC -Ma +M +C.

%mode   fp_key_cert           +CC +P -Pkey.

%mode   fp_prp_match          +R +P.

%mode   fp_prp_valid          +P.

%mode   fp_module_valid       +M.

%mode   fp_cmodule_valid      +L.

%mode   fp_valid_clist        +L.

%mode   fp_valid_code_hash    +M.

%mode   fp_ide_valid          +X.

%mode   fp_ide_list_valid     +L.

%mode   fp_list_valid         +E +L.

%mode   fp_list_is_nil        +L.

%mode   fp_list_member        +L *X.

%mode   fp_validper_refl      +E +X.

%mode   fp_validper           +E.

%mode   fp_set                +S -X.

%mode   fp_emptyset           +S.

%mode   ax_key_auth           +CC -Ka.

%mode   ax_key_bind           +P -K.
```

```
%mode   ax_prp_server        +CC -P.
%mode   ax_prp_auth          +CC -Ma.
%mode   ax_auth_sig_stmt     +K -S.
%mode   ax_library_dsc       +L +LC.
```

A.2.2 Termination Declaration

A termination declaration of a predicate[1] consits of an order specification and a call pattern. A call pattern shows which arguments are required to be reduced whenever making a call to a predicate. An argument must get an identifier if it is used in the termination order specifiation. Otherwise, a place holder '_' can be used for an argument. The termination order specification part informs which arguments decreases at recursive calls.

Table A.2 gives the full termination declaration of Secure Linking prover. The prover and its termination declaration is checked successfully by the termination checker of Twelf; this can be used fo a machine-generated proof of the prover's termination.

```
%terminates    F          (ax_calc_hash F _).
%terminates    CC         (ax_key_auth CC Ka).
%terminates    P          (ax_key_bind P _).
%terminates    CC         (ax_prp_server CC P).
%terminates    CC         (ax_prp_auth CC Ma).
%terminates    K          (ax_auth_sig_stmt K _).
%terminates    [L LC]     (ax_library_dsc L LC).
%terminates    X          (fp_ide_valid X).
%terminates    M          (fp_module_valid M).
%terminates    L          (fp_list_is_nil L).
%terminates    L          (fp_ide_list_valid L).
```

[1] in Secure Linking prover, it is corresponding to a tactical.

```
%terminates     L               (fp_mlist_valid L).
%terminates     P               (fp_prp_valid P).
%terminates     X               (fp_export_valid X).
%terminates     E               (fp_validper E).
%terminates     X               (fp_validper_refl _ X).
%terminates     L               (fp_list_valid _ L).
%terminates     L               (fp_list_member L _).
%terminates     L               (fp_list_eq_refl _ L).
%terminates     S               (fp_set S _).
%terminates     S               (fp_emptyset S).
%terminates     M               (fp_valid_code_hash M).
%terminates     L               (fp_valid_clist L).
%terminates     R               (fp_prp_match R _).
%terminates     C               (fp_cdsc_valid C).
%terminates     [CC Ka]         (fp_key_cert CC Ka _).
%terminates     Ma              (fp_valid_sig_auth _ Ma _).
%terminates     C               (fp_valid_sig_cdsc _ _ _ C).
%terminates     C               (fp_signed_ma _ _ C _).
%terminates     C               (fp_signed_prp _ _ C _).
%terminates     E               (fp_all_signed _ _ _ E).
%terminates     C               (fp_signed_cdsc _ _ C).
%terminates     [L LC]          (fp_valid_lib L LC).
%terminates     E               (fp_has_prp _ E).
%terminates     E               (fp_has_prp' _ E).
%terminates     R               (fp_has_property R _).
%terminates     L               (fp_st_imprt_cdsc _ L).
%terminates     I               (fp_st_imprt I _).
%terminates     [R E]           (fp_exp_prps R E).
%terminates     I               (fp_prv_prps I _ _).
%terminates     D               (fp_ok_to_link _ _ D _ _ _).
```

A.3 Conditional Completeness Proof

A.3.1 Group 2: Tacticals with no subgoals

fp_list_is_nil:

As explained in Section 6.2, the only way of constructing a nil list is to use the pre-defined list constructor *list_nil*. The tactic [R.61] covers this case, so the tactical *fp_list_is_nil* is complete.[2]

fp_emptyset:

The only way the Secure Linking parsers to construct an empty set is to use the pre-defined set constructor *set_empty*. The tactic [R.65] covers this case, so the tactical *fp_emptyset* is complete.

fp_validper:

The tactical *fp_validper* is related to the predicate *validper*, and the tactic [R.56] is only one tactic calling the tactical *fp_validper*. The tactic [R.56] is related to the tactical *fp_list_valid*, which reasons about the validity of lists. Although the Secure Linking logic (in fact, its underlying Core Logic) allows to construct a list of any legitimate type, but only 4 types matter when considering the formulas generated by the trusted parsers. That means, only 4 different kinds of lists are taken into consider with respect to the tactical *fp_validper*. The tactics [R.66], [R.67], [R.68] and [R.69] cover the 4 cases which are produced by the Secure Linking parsers, therefore the tactical *fp_validper* is complete.

A.3.2 Group 3: Tacticals with no recursive calls

The completeness of these tacticals rely on the completeness of dependent tacticals (or subsequently called tacticals and the completeness of the input domain covered by the tactics of each tactical. The tacticals in this group can be subgrouped by the patterns they call dependent tacticals:

[2] Note that the word *completeness* in this proof refers to the conditional completeness on input terms parsed by the Secure Linking parsers.

- calling dependent tacticals with their original input terms.
- calling dependent tacticals after getting rid of term constructors.

When dependent tacticals are called with their original input terms, the completeness of these tacticals only depends on the completeness of dependent tacticals. The tacticals of this subgroup are *fp_ok_to_link, fp_exp_prps, fp_signed_prp, fp_signed_ma, fp_valid_sig_auth, fp_valid_sic_cdsc, fp_key_cert, fp_export_valid, fp_import_valid* and *fp_validper_refl*.

fp_ok_to_link:

The tactic [R.1] is the only rule related to the tactical *fp_ok_to_link*. The completeness of this tactical depends on the completeness of dependent tacticals *fp_signed_cdsc*, *fp_exp_prps* and *fp_prv_prps*, which are proved at Section A.3.3, Section A.3.2 and Section A.3.3. Therefore, the tactical *fp_ok_to_link* is complete.

fp_exp_prps:

The tactic [R.4] is the only rule related to the tactical *fp_exp_prps*. The completeness of this tactical depends on the completeness of a dependent tactical *fp_has_property*, which is proved at Section A.3.3. Therefore, the tactical *fp_exp_prps* is complete.

fp_signed_prp:

The tactic [R.26] is the only rule related to the tactical *fp_signed_prp*. The completeness of this tactical depends on the completeness of dependent tacticals *fp_cdsc_valid* and *fp_signed_ma*, which are proved at Section A.3.3, Section A.3.2 respectively. Therefore, the tactical *fp_signed_prp* is complete.

fp_signed_ma:

The tactic [R.27] is the only rule related to the tactical *fp_signed_ma*. The completeness of this tactical depends on the completeness of dependent tacticals *fp_valid_sig_cdsc*, *fp_set* and *fp_prp_match*, which are proved at Section A.3.2, Section A.3.3 and Section A.3.2 respectively. Therefore, the tactical *fp_signed_ma* is complete.

fp_valid_sig_auth:

The tactic [R.28] is the only rule related to the tactical *fp_valid_sig_auth*. The completeness of this tactical depends on the completeness of dependent tacticals *ax_prp_server*, *fp_key_cert* and *ax_auth_sig_stmt* which are proved at Section 6.5, Section A.3.2 and Section 6.5 respectively. Therefore, the tactical *fp_valid_sig_auth* is complete.

fp_valid_sic_cdsc:

The tactic [R.29] is the only rule related to the tactical *fp_valid_sig_cdsc*. The completeness of this tactical depends on the completeness of dependent tacticals *ax_prp_auth*, *fp_key_cert* and *ax_auth_sig_stmt* which are proved at Section 6.5, Section A.3.2 and Section 6.5 respectively. Therefore, the tactical *fp_valid_sig_cdsc* is complete.

fp_key_cert:

The tactic [R.30] is the only rule related to the tactical *fp_key_cert*. The completeness of this tactical depends on the completeness of dependent tacticals *ax_key_auth*, *ax_key_bind* and *ax_auth_sig_stmt* which are all proved at Section 6.5. Therefore, the tactical *fp_key_cert* is complete.

fp_export_valid:

The tactic [R.33] is the only rule related to the tactical *fp_export_valid*. The completeness of this tactical depends on the completeness of dependent tacticals *fp_list_valid*, which is proved at Section A.3.3. Therefore, the tactical *fp_export_valid* is complete.

fp_import_valid:

The tactic [R.34] is the only rule related to the tactical *fp_import_valid*. The completeness of this tactical depends on the completeness of dependent tacticals *fp_list_valid*, which is proved at Section A.3.3. Therefore, the tactical *fp_import_valid* is complete.

fp_validper_refl:

The tactical *fp_validper_refl* is related to the tactics [R.55] and [R.59]. Note that both of the tactics are reasoning about the lists in the Secure Linking logic. As mentioned above, the lists of only 4 types matter when considering the formulas generated by the trusted parsers. The tactics [R.70], [R.71], [R.72] and [R.73] cover the four cases which are produced by the Secure Linking parsers.

In the case of the rule [R.70], the completeness of this tactic depends on the completeness of the subsequent tactical *fp_module_valid*, which is proved at Section A.3.2. In the case of the rule [R.71], the completeness of this tactic depends on the completeness of the subsequent tactical *fp_prp_valid*, which is proved at Section A.3.2. In the case of the rule [R.72], the completeness of this tactic depends on the completeness of the subsequent tactical *fp_cmodule_valid*, which is proved at Section A.3.3. In the case of the rule [R.73], the completeness of this tactic depends on the completeness of the subsequent tactical *fp_export_valid*, which is proved at Section A.3.2.

The tacticals in the other subgroup, *fp_prp_match, fp_prp_valid*, *fp_module_valid* and *fp_valid_-chash*, call the subsequent tacticals after stripping term constructors. In this case, the completeness of these tacticals depends on how completely the tactics of a tactical cover all possible input terms as well as the completeness of the subsequent tacticals.

fp_prp_match:

In the Secure Linking logic, 5 term constructors are used to build property requests. The first term constructor *mk_prp_rq* is used to make a property request out of a property term. The tactic [R.35] is the rule related to *mk_prp_rq*. The completeness of dependent tactical *fp_prp_valid* is proved in this section, therefore the tactic [R.35] is complete. Terms constructed by *rq_cname_exists* are handled by the tactic [R.36]. This tactic has any subgoals, so it is complete. The tactic [R.37] is the rule related to the term constructor *mk_rq_cname*. The completeness of dependent tactical *fp_ide_valid* is proved at Section A.3.3, therefore the tactic [R.37] is complete. The tactic [R.38] is the tactic related to the term constructor *rq_export_ids_exists*. The completeness of dependent tactical *fp_ide_list_valid* is proved at Section A.3.3, therefore the tactic [R.38] is complete. The tactic [R.39] is the tactic related to the term constructor *rq_code_hash_checkable*. The completeness of dependent tacticals *fp_valid_clist* and *fp_cmodule_valid* are proved at Section A.3.3, therefore the tactic [R.39] is complete.

By induction on the cases, the tactical *fp_prp_match* is complete.

fp_prp_valid:

The Secure Linking logic has 4 term constructors for making property terms. The tactic [R.40] is the tactic related to the term constructor *mk_prp*. This is the term constructor making a property term out of an identifier standing for a property name. The completeness of dependent tactical *fp_ide_valid* is proved at Section A.3.3, therefore the tactic [R.40] is complete. The tactic [R.41] is the tactic related to the term constructor *mk_prp_cname*. This is the term constructor making a property term out of an identifier standing for a property name. The completeness of dependent tactical *fp_ide_valid* is proved at Section A.3.3, therefore the tactic [R.41] is complete. The tactic [R.42] is the tactic related to the term constructor *mk_prp_export_ids*. The completeness of dependent tactical *fp_ide_list_valid* is proved at Section A.3.3, therefore the tactic [R.42] is complete. The tactic [R.43] is the tactic related to the term constructor *mk_prp_code_hash*. The completeness of dependent tactical *fp_cmodule_valid* is proved at Section A.3.3, therefore the tactic [R.43] is complete.

By induction on the cases, the tactical *fp_prp_valid* is complete.

fp_module_valid:

The only way of constructing a module term is using the term constructor *mk_module*, and the tactic [R.44] handles this case. The completeness of dependent tactical *fp_ide_valid* is proved at Section A.3.3, therefore the tactical *fp_module_valid* is complete.

fp_valid_chash:

The tactical *fp_valid_chash* takes a module term as an argument, and the term constructor *mk_module* is the only way of constructing a module term in the Secure Linking logic. The tactic [R.49] shows the only case. The completeness of this tactical relies on the completeness of dependent tacticals *ax_calc_hash* and *fp_ide_valid*. The completeness of the tactical *ax_calc_hash* is discussed at Section 6.5, and the completeness of the tactical *fp_ide_valid* is proved at Section A.3.3. Therefore the tactical *fp_valid_chash* is complete.

A.3.3 Group 4: Tacticals with recursive calls

A tactical in this group make recursive calls to itself and subsequent calls to other tacticals. The completeness of a tactical making recursive calls can be proved by induction on the structure of input terms. Therefore the completeness of tacticals in this group is proved by using induction as well as other dependent tacticals' completeness. The tacticals are grouped again with respect that the types of input terms dependent tacticals are called with original input terms or their subterms. The subcategories are:

- tacticals making subsequent calls with subterms of type $\mathcal{C}$ (a type for component descriptions). This group consists of the tacticals *fp_prv_prps, fp_has_prp*$'$, *fp_signed_cdsc* and *fp_cdsc_valid*.
- tacticals making subsequent calls with subterms of type $\mathcal{L}(\tau)$ (a type for lists). This group consists of the tacticals *fp_st_imprt, fp_st_imprt_cdsc, fp_has_prp, fp_valid_lib, fp_all_signed, fp_cmodule_valid, fp_valid_clist, fp_ide_list_valid, fp_list_valid* and *fp_list_member*.
- tacticals making subsequent calls with subterms of type $\mathcal{S}(\tau)$ (a type for sets). This group consists of the tacticals *fp_has_property* and *fp_set_member*.
- tacticals making subsequent calls with subterms of type $\mathcal{I}$ (a type for identifiers). This group has only one tactical *fp_ide_valid*.

The tacticals in the first subgroup make recursive calls with subterms of type $\mathcal{C}$ (a type for component descriptions). In the Secure Linking logic, a component description can be constructed by using the term constructors *mk_component_dsc* and *component_dsc_combine*. The case in which a component description built from *mk_component_dsc* is the base case of the induction when proving the completeness of these tacticals. The term constructor *component_dsc_combine* is used for constructing a bigger component description term out of two component description terms. Hence, the tactics using *component_dsc_combine* cover the inductive step of the induction.

Following are the proofs of the tacticals in this subgroup one by one.

fp_prv_prps:

The base case of the tactical *fp_prv_prps* is when a component description is formed by using the term constructor *mk_component_dsc*. The tactic [R.3] handles this case, and the completeness of the tactic relies on the completeness of dependent tacticals *fp_list_valid*, *fp_valid_lib* and *fp_st_imprt*. The completeness of those tacticals are proved at Section A.3.3. Hence, the tactical *fp_prv_prps* is complete in the base case.

For the inductive step, suppose that the tactical *fp_prv_prps* is complete with component descriptions C and C'. The only way of constructing a component description out of two component descriptions is to use the term constructor *component_dsc_combine*. By the tactic [R.2], the completeness of the tactical *fp_prv_prps* depends on the completeness of calling itself with component descriptions C and C' (true by the induction hypothesis), the completeness of the tactical *fp_cdsc_valid* (proved at Section A.3.3). So the tactical *fp_prv_prps* is complete for the inductive step.

By induction, the tactical *fp_prv_prps* is complete.

fp_has_prp′:

The base case of the tactical *fp_has_prp'* is when a component description is formed by using the term constructor *mk_component_dsc*. The tactic [R.15] handles this case, and the completeness of the tactic relies on the completeness of dependent tacticals *fp_export_valid* and *fp_has_prp*. The completeness of the tactical *fp_export_valid* is proved at Section A.3.2, and the completeness of the tactical *fp_has_prp* is proved at Section A.3.3. Hence, the tactical *fp_has_prp'* is complete in the base case.

For the inductive step, suppose that the tactical *fp_has_prp'* is complete with component descriptions C and C'. The only way of constructing a component description out of two component descriptions is to use the term constructor *component_dsc_combine*. The tactic [R.13] shows the case in which the formula with the component description C is true. The completeness of the tactical *fp_has_prp'* depends on the completeness of calling itself with the component description C (true by the induction hypothesis). The tactic [R.14] shows the case in which the formula with the component description C' is true. The completeness of the tactical *fp_has_prp'* depends on the completeness of calling itself with the component description C' (true by the induction hypothesis). By induction on the cases, the tactical *fp_has_prp'* is complete for the inductive step.

By induction, the tactical *fp_has_prp'* is complete.

fp_signed_cdsc:

The base case of the tactical *fp_signed_cdsc* is when a component description is formed by using the term constructor *mk_component_dsc*. The tactic [R.23] handles this case, and the completeness of the tactic relies on the completeness of dependent tacticals *fp_export_valid*, *fp_import_valid* and *fp_all_signed*. The completeness of the tacticals *fp_export_valid* and *fp_import_valid* are proved at Section A.3.2, and the completeness of the tactical *fp_all_signed* is proved at Section A.3.3. Hence, the tactical *fp_signed_cdsc* is complete in the base case.

For the inductive step, suppose that the tactical *fp_signed_cdsc* is complete with component descriptions C and C'. The only way of constructing a component description out of two component descriptions is to use the term constructor *component_dsc_combine*. By the tactic [R.22], the completeness of the tactical *fp_signed_cdsc* depends on the completeness of calling itself with component descriptions C and C' (true by the induction hypothesis). So the tactical *fp_signed_cdsc* is complete for the inductive step.

By induction, the tactical *fp_signed_cdsc* is complete.

fp_cdsc_valid:

The base case of the tactical *fp_cdsc_valid* is when a component description is formed by using the term constructor *mk_component_dsc*. The tactic [R.32] handles this case, and the completeness of the tactic relies on the completeness of dependent tacticals *fp_export_valid* and *fp_import_valid*. The completeness of the tacticals *fp_export_valid* and *fp_import_valid* are proved at Section A.3.2. Hence, the tactical *fp_cdsc_valid* is complete in the base case.

For the inductive step, suppose that the tactical *fp_cdsc_valid* is complete with component descriptions C and C'. The only way of constructing a component description out of two component descriptions is to use the term constructor *component_dsc_combine*. By the tactic [R.31], the completeness of the tactical *fp_cdsc_valid* depends on the completeness of calling itself with component descriptions C and C' (true by the induction hypothesis). So the tactical *fp_cdsc_valid* is complete for the inductive step.

By induction, the tactical *fp_cdsc_valid* is complete.

The tacticals in the second subgroup make subsequent calls with subterms of type $\mathcal{L}(\tau)$ (a type for lists). The Secure Linking logic has 3 list constructors *list_nil*, *list_cons* and *list_cat*. A list in the Secure Linking logic can be formed by using the constructors *list_nil* and *list_cons*, or by using the constructors *list_nil* and *list_cat*. For each case, the base case is when a list is constructed by using the constructor *list_nil*. For the inductive steps, lists formed by the constructors *list_cons* and/or *list_cat* are considered. If two kinds of lists are considered, the completeness of the inductive step is proved by the induction on the cases.

Following are the proofs of the tacticals in this subgroup one by one.

fp_st_imprt:

The base case of the tactical *fp_st_imprt* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.7] handles this case, and the completeness of the tactic depends on the completeness of the dependent tactical *fp_list_is_nil*. The completeness of the tactical *fp_list_is_nil* is proved at Section A.3.1. Hence, the tactical *fp_st_imprt* is complete in the base case.

For the inductive step, suppose that the tactical *fp_st_imprt* is complete with lists C_i and $C_i{}'$. The Secure Linking has two list constructors forming a list out of given lists: *list_cat* and *list_cons*. The tactic [R.5] handles the case in which a list is constructed by using *list_cat*. The completeness of this tactic relies on the completeness of calling itself with lists C_i and $C_i{}'$ (true by induction hypothesis), so the tactic [R.5] is complete. The tactic [R.6] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with a list C_i (true by induction hypothesis), and the completeness of the dependent tactical *fp_st_imprt_cdsc* (proved at Section A.3.3. By induction on the cases, the tactical *fp_st_imprt* is complete for the inductive step.

By induction, the tactical *fp_st_imprt* is complete.

fp_st_imprt_cdsc:

The completeness of the tactical *fp_st_imprt_cdsc* is proved by induction on the length of an input list. As discussed in Chapter 6, any parsed true formula for the tactical *fp_st_imprt_cdsc* is constructed by using the list constructors *list_cons* and *list_nil* because a list of library descriptions is built by using only those 2 constructors.

The base case is when the length of the input list is one. That means, the input list is formed with a head element and a nil list. The tactic [R.8] is applied for this case. The completeness of this tactic relies on the completeness of dependent tacticals *fp_export_valid* and *fp_has_property*. The completeness of *fp_export_valid* and *fp_has_property* are proved at Section A.3.2 and at Section A.3.3. Hence, the tactical *fp_st_imprt_cdsc* is complete in the base case.

For inductive step, suppose that the tactical *fp_st_imprt_cdsc* is complete when an input list of length k. Then a list of length $k+1$ can be constructed by using *list_cons*. Note that the tactic [R.9] is applied only when applying the tactic [R.8] fails. If the tactic [R.8] is applied, the completeness of the tactical depends on the completeness of dependent tacticals *fp_export_valid* (proved at Section A.3.2) and *fp_has_property* (proved at Section A.3.3). So the tactic [R.8] is complete. If the tactic [R.9] is applied, the completeness of the tactical depends on the completeness of calling itself with a list of length k (true by induction hypothesis) and the completeness of the dependent tactical *fp_list_valid* (proved at Section A.3.3). So the tactic [R.8] is complete. By induction on the cases, the tactical *fp_st_imprt_cdsc* is complete for the inductive step.

By induction, the tactical *fp_st_imprt_cdsc* is complete.

fp_has_prp:

For any list formula which makes the tactical *fp_has_prp* true in the Secure Linking logic, the formula must be constructed by using the term constructors *list_cat* or *list_cons*.

If a list formula is built by *list_cat* and the list makes the tactical *fp_has_prp* true, the first part of the concatenated list must make the tactical *fp_has_prp* true [R.16], or the second part of the concatenated list must make the tactical *fp_has_prp* true [R.17]. If a list formula is built by *list_cons* and the list makes the tactical *fp_has_prp* true, the head element must make the tactical *fp_has_prp* true [R.18], or the tail list must make the tactical *fp_has_prp* true [R.19].

By induction on the cases, the tactical *fp_has_prp* is complete.

fp_valid_lib:

The base case of the tactical *fp_valid_lib* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.21] handles this case, and the completeness of the tactic depends on the completeness of the dependent tactical *fp_list_is_nil*. The completeness of the tactical *fp_list_is_nil* is proved at Section A.3.1. Hence, the tactical *fp_valid_lib* is complete in the base case.

For the inductive step, suppose that the tactical *fp_valid_lib* is complete with lists L_m and L_d. Note that a list of library descriptions is built by using only *list_cons* and *list_nil* by the Secure Linking parsers. The tactic [R.20] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with L_m and L_d (true by induction hypothesis), and the completeness of the dependent tacticals *ax_library_dsc* (discussed at Section 6.5), *fp_export_valid* (proved at Section A.3.2), and *fp_cmodule_valid* (proved at Section A.3.3). So the tactical *fp_valid_lib* is complete for the inductive step.

By induction, the tactical *fp_valid_lib* is complete.

fp_all_signed:

The base case of the tactical *fp_all_signed* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.25] handles this case, and the completeness of the tactic depends on the completeness of the dependent tactical *fp_list_is_nil*. The completeness of the tactical *fp_list_is_nil* is proved at Section A.3.1. Hence, the tactical *fp_all_signed* is complete in the base case.

For the inductive step, suppose that the tactical *fp_valid_lib* is complete with a list C_e. Note that the tactic [R.23] is the only one calling the tactical *fp_all_signed*, and an input argument comes from the argument of a component description. The Secure Linking parsers use only the list constructor *list_cons* when building a component description from a list of modules and a list of exported properties. Thus, any list input to the tactical *fp_all_signed* is always built by using *list_cons*. The tactic [R.20] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with C_e (true by induction hypothesis), and the completeness of the dependent tactical *fp_signed_prp* (proved at Section A.3.2). So the tactical *fp_all_signed* is complete for the inductive step.

By induction, the tactical *fp_all_signed* is complete.

fp_cmodule_valid:

The base case of the tactical *fp_cmodule_valid* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.46] handles this case, and the completeness of the tactic depends on the completeness of the dependent tactical *fp_list_is_nil*. The completeness of the tactical *fp_list_is_nil* is proved at Section A.3.1. Hence, the tactical *fp_cmodule_valid* is complete in the base case.

For the inductive step, suppose that the tactical *fp_cmodule_valid* is complete with a list M_c. Note that the Secure Linking parsers always use the list constructor *list_cons* for a list of modules. The tactic [R.45] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with M_c (true by induction hypothesis), and the completeness of the dependent tactical *fp_module_valid* (proved at Section A.3.2). So the tactical *fp_cmodule_valid* is complete for the inductive step.

By induction, the tactical *fp_cmodule_valid* is complete.

fp_valid_clist:

The base case of the tactical *fp_valid_clist* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.48] handles this case, and the completeness of the tactic depends on the completeness of the dependent tactical *fp_list_is_nil*. The completeness of the tactical *fp_list_is_nil* is proved at Section A.3.1. Hence, the tactical *fp_valid_clist* is complete in the base case.

For the inductive step, suppose that the tactical *fp_valid_clist* is complete with a list M_c. Note that the Secure Linking parsers always use the list constructor *list_cons* for a list of modules. The tactic [R.47] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with M_c (true by induction hypothesis), and the completeness of the dependent tactical *fp_valid_chash* (proved at Section A.3.2). So the tactical *fp_valid_clist* is complete for the inductive step.

By induction, the tactical *fp_valid_clist* is complete.

fp_ide_list_valid:

The base case of the tactical *fp_ide_list_valid* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.53] handles this case, and the completeness of the tactic depends on the completeness of the dependent tactical *fp_list_is_nil*. The completeness of the tactical *fp_list_is_nil* is proved at Section A.3.1. Hence, the tactical *fp_ide_list_valid* is complete in the base case.

For the inductive step, suppose that the tactical *fp_ide_list_valid* is complete with a list $L(I)$. Note that the Secure Linking parsers always use the list constructor *list_cons* for a list of exported identifiers. The tactic [R.52] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with $L(I)$ (true by induction hypothesis), and the completeness of the dependent tactical *fp_ide_valid* (proved at Section A.3.3). So the tactical *fp_ide_list_valid* is complete for the inductive step.

By induction, the tactical *fp_ide_list_valid* is complete.

fp_list_valid:

The base case of the tactical *fp_list_valid* happens when a list is constructed by the term constructor *list_nil*. The tactic [R.56] handles this case, and the completeness of the tactic depends on the completeness of the dependent tacticals *fp_list_is_nil* and *fp_validper*. The completeness of those tacticals are proved at Section A.3.1. Hence, the tactical *fp_list_valid* is complete in the base case.

For the inductive step, suppose that the tactical *fp_list_valid* is complete with lists $L(\tau)$ and $L(\tau)'$. The Secure Linking has two list constructors forming a list out of given lists: *list_cat* and *list_cons*. The tactic [R.54] handles the case in which a list is constructed by using *list_cat*. The completeness of this tactic relies on the completeness of calling itself with lists $L(\tau)$ and $L(\tau)'$ (true by induction hypothesis), so the tactic [R.54] is complete. The tactic [R.55] handles the case in which a list is constructed by using *list_cons*. The completeness of this tactic relies on the completeness of calling itself with a list $L(\tau)$ (true by induction hypothesis), and the completeness of the dependent tactical *fp_validper_refl* (proved at Section A.3.2). By induction on the cases, the tactical *fp_list_valid* is complete for the inductive step.

By induction, the tactical *fp_list_valid* is complete.

fp_list_member:

For any list formula which makes the tactical *fp_list_member* true in the Secure Linking logic, the formula must be constructed by using the term constructors *list_cat* or *list_cons*.

If a list formula is built by *list_cat* and the list makes the tactical *fp_list_member* true, the first part of the concatenated list must make the tactical *fp_list_member* true [R.57], or the second part of the concatenated list must make the tactical *fp_list_member* true [R.58]. If a list formula is built by *list_cons* and the list makes the tactical *fp_list_member* true, the head element must make the tactical *fp_list_member* true [R.59], or the tail list must make the tactical *fp_list_member* true

[R.60].

By induction on the cases, the tactical *fp_list_member* is complete.

The tacticals in the third subgroup make subsequent calls with subterms of type $\mathcal{S}(\tau)$ (a type for sets). The set constructs included in the Secure Linking logic are *set_empty*, *set_singleton* and *set_union*. Tacticals in this subgroup are *fp_has_property* and *fp_set_member*. Following are the proofs of the tacticals in this subgroup one by one.

fp_has_property:

A true set formula which makes the tactical *fp_has_property* is either an empty set or a nonempty set. In case of an empty set, the tactic [R.12] is applied. The completeness of this tactic relies on the completeness of the tactical *fp_emptyset* (proved at Section A.3.1); thus the tactic [R.12] is complete.

For a nonempty set, the base case occurs when the cardinality of the set is 1. The tactic [R.11] handles this case, and the completeness depends on the completeness of the tactical *fp_has_prp'* (proved at Section A.3.3); thus the tactical *fp_has_property* is complete in the base case.

Suppose that the tactical *fp_has_property* is complete with input sets R_p and $R_p{}'$ for the inductive step. The only way of building a bigger set from sets is using the set constructor *set_union*, and the tactic [R.10] handles this case. The completeness of the tactic [R.10] relies on the completeness of calling itself with R_p and $R_p{}'$ (true by induction hypothesis); thus the tactical *fp_has_property* is complete for the inductive step. By induction, the tactical *fp_has_property* is complete when an input set is nonempty.

By induction on the cases, the tactical *fp_has_property* is complete.

fp_set_member:

The base case of the tactical *fp_set_member* happens when a list is constructed by the term constructor *set_singleton*. The tactic [R.64] handles this case, and it requires no further calls to other tacticals. Hence, the tactical *fp_set_member* is complete in the base case.

For the inductive step, suppose that the tactical *fp_set_member* is complete with sets $S(\tau)$ and $S(\tau)'$. The Secure Linking has only one set constructor forming a set out of given sets: *set_union*.

If an element x is a member of a unioned set of two sets, x is a member of one set or the other. The tactics [R.62] and [R.63] cover this case. The completeness depends on the completeness of calling the tactical *fp_set_member* itself with sets $S(\tau)$ or $S(\tau)'$, and it is proved by induction hypothesis. Thus, the tactical *fp_set_member* is complete for the inductive step.

By induction, the tactical *fp_set_member* is complete.

The tactical in the fourth subgroup make subsequent calls with subterms of type $\mathcal{I}$ (a type for identifiers). There are two term constructors for identifiers: *mk_ch* and *mk_str*. The tactical *fp_ide_valid* is the only one in this subgroup.

The base case of the tactical *fp_ide_valid* happens when an identifier is constructed by the term constructor *mk_ch*. The tactic [R.51] handles this case, and it requires no further calls to other tacticals. Hence, the tactical *fp_ide_valid* is complete in the base case.

For the inductive step, suppose that the tactical *fp_ide_valid* is complete with identifiers I_c and I_s. The Secure Linking has only one term constructor forming an identifier from another identifier: *mk_str*. The tactic [R.50] handles this case. The completeness depends on the completeness of calling the tactical *fp_ide_valid* itself with I_c and I_s, and it is proved by induction hypothesis. Thus, the tactical *fp_ide_valid* is complete for the inductive step.

By induction, the tactical *fp_ide_valid* is complete.

Appendix B

Simple-Gt System

```
t: tp.
t_nonempty: pf (exists [x: tm t] true).
t_infinite:
 pf (exists [r: tm (t arrow t arrow form)]
     (forall2 [x][y] r @ x @ y equiv not (r @ y @ x)) and
     (forall3 [x][y][z] (r @ x @ y imp r @ y @ z imp r @ x @ z)) and
   (forall [x] exists [y] r @ x @ y)).

nat : tp = (t arrow t) arrow (t arrow t).
zero : tm nat = lam [f] lam [x] x.
succ : tm (nat arrow nat) =
 lam [n] lam2 [f][x] f @ (n @ f @ x).

gt : tm (nat arrow nat arrow form) =
 lam2 [a][b]
  forall [r]
    (forall [n] r @ (succ @ n) @ zero) imp
    (forall2 [x][y] r @ x @ y imp r @ (succ @ x) @ (succ @ y)) imp
   r @ a @ b.

gt-zero : pf (gt @ (succ @ N) @ zero) =
 def2_i
 (forall_i [r]
  imp2_i [q1: pf (forall [n] r @ (succ @ n) @ zero)][q2]
  forall_e q1 N).

gt-succ :
 pf (gt @ X @ Y) ->
 pf (gt @ (succ @ X) @ (succ @ Y)) =
 [p1: pf (gt @ X @ Y)]
 def2_i (forall_i [r] imp2_i
 [q1: pf (forall [n] r @ (succ @ n) @ zero)]
 [q2: pf (forall2 [x][y] r @ x @ y imp r @ (succ @ x) @ (succ @ y))]
 cut (imp2_e (forall_e (def2_e p1) r) q1 q2)
 [q3: pf (r @ X @ Y)]
 imp_e (forall2_e q2 X Y) q3).
```

Appendix C

Complete Secure Linking Logic

```
%%% linking definitions

library_dsc : tm (rel module_list export).

valid_library : tm (rel (list module_list) (list export)) =
 lam2 [mLst][cidLst]
  list_valid @ module_list_eq @ mLst and
  list_valid @ export_eq @ cidLst and
  (exists [n] list_length @ mLst @ n and list_length @ cidLst @ n) and
  forall3 [i][x][xid] (list_nth @ mLst @ i @ x and list_nth @ cidLst @ i @ xid) imp
   exists2 [x'][xid']
    module_list_eq @ x @ x' and export_eq @ xid @ xid' and
    library_dsc @ x' @ xid'.

has_prp : tm (rel prp_rq (list property)) =
 lam2 [rq][prps]
  exists [prp] list_member @ prps @ prp and rq @ prp.

has_property : tm (rel (set prp_rq) (list property)) =
 lam2 [rqs][prps]
   forall [rq] rqs @ rq imp has_prp @ rq @ prps.

imprt_match : tm (rel import (list export)) =
 lam2 [i][libdsc]
  exists [cid] list_member @ libdsc @ cid and has_property @ i @ cid.

satisfy_imports : tm (rel (list import) (list export)) =
 lam2 [importLst][libdsc]
  forall [i] list_member @ importLst @ i imp
   imprt_match @ i @ libdsc.

provide_enough_lib : tm (rel3 (list import) (list module_list) (list export)) =
 lam3 [imprt][lib][libdsc]
  valid_library @ lib @ libdsc and
  satisfy_imports @ imprt @ libdsc.
```

```
export_required_prps : tm (rel (set prp_rq) export) =
 lam2 [rprps][exprt] has_property @ rprps @ exprt.

ok_to_link : tm (module_list arrow component_dsc arrow
                (list module_list) arrow (list export) arrow (set prp_rq) arrow form) =
 lam5 [m][cdsc][lib][libdsc][rqs]
    signed_component_dsc @ m @ cdsc and
    provide_enough_lib @ (component_dsc_imprt_list @ cdsc) @ lib @ libdsc and
    export_required_prps @ rqs @ (component_dsc_exprt @ cdsc).

%%% component description definitions

export : tp = list property.
export_eq = list_eq @ prp_eq.
export_valid = list_valid @ prp_eq.

import : tp = set prp_rq.
import_eq = set_equiv.
imprt_list_eq = list_eq @ import_eq.
imprt_list_valid = list_valid @ import_eq.

component_dsc : tp = pair export (list import).

mk_component_dsc : tm (export arrow (list import) arrow component_dsc) =
 lam2 [id][imports] mkpair @ id @ imports.

component_dsc_exprt : tm (component_dsc arrow export) = fst.
component_dsc_imprt_list : tm (component_dsc arrow (list import)) = snd.

component_dsc_eq : tm (eqrel component_dsc) =
 lam2 [g1][g2]
 (export_eq @ (fst @ g1) @ (fst @ g2)) and
 (list_eq @ import_eq @ (snd @ g1) @ (snd @ g2)).

component_dsc_valid : tm (component_dsc arrow form) =
 lam [cdsc]
  list_valid @ prp_eq @ (component_dsc_exprt @ cdsc) and
  list_valid @ import_eq @ (component_dsc_imprt_list @ cdsc).

sub_export : tm (rel export export) =
 lam2 [lst1][lst2]
  forall [x] list_member @ lst1 @ x imp list_member @ lst2 @ x.

sub_import_list : tm (rel (list import) (list import)) =
 lam2 [lst1][lst2]
  list_is_not_nil @ lst1 imp
  exists2 [f][l]
   sublist_proper_range @ lst2 @ f @ l and
   list_eq @ set_equiv @ lst1 @ (sublist @ f @ l @ lst2).

sub_component_dsc : tm (rel component_dsc component_dsc) =
 lam2 [dsc1][dsc2]
  sub_export @ (component_dsc_exprt @ dsc1)
```

```
             @ (component_dsc_exprt @ dsc2) and
  sub_import_list @ (component_dsc_imprt_list @ dsc1)
                  @ (component_dsc_imprt_list @ dsc2).

export_combine : tm (export arrow export arrow export) = list_cat.
imprt_list_combine :
  tm (list import arrow list import arrow list import) = list_cat.

component_dsc_combine : tm (component_dsc arrow component_dsc arrow component_dsc) =
 lam2 [dsc1][dsc2]
  mk_component_dsc
      @ (export_combine @ (component_dsc_exprt @ dsc1) @ (component_dsc_exprt @ dsc2))
      @ (imprt_list_combine @ (component_dsc_imprt_list @ dsc1)
                             @ (component_dsc_imprt_list @ dsc2)).

%%% authentication definitions

controls : tm (rel worldview form) =
 lam2 [a][f] (says @ a @ f) imp f.

keybind : tm (rel worldview str) =
 lam2 [prn][pkey] speaksfor @ (key @ pkey) @ prn.

key_authority : tm (worldview arrow form) =
 lam [ca] forall2 [p][k] controls @ ca @ (keybind @ p @ k).

keycert : tm (rel worldview str) =
 lam2 [prn][pkey]
  exists [ca] key_authority @ ca and says @ ca @ (keybind @ prn @ pkey).

guarantees : tm (rel worldview (set prp_rq)).

prp_server : tm (worldview arrow form) =
 lam [pa]
  forall2 [ma][prpSet] controls @ pa @ (guarantees @ ma @ prpSet).

valid_sig_prp_auth : tm (rel worldview (set prp_rq)) =
 lam2 [ma][prpSet]
   exists2 [pa][paKey]
    prp_server @ pa and
    keycert @ pa @ paKey and
    signed @ paKey @ (guarantees @ ma @ prpSet).

module_dsc : tm (rel (list module) component_dsc).

property_authority : tm (worldview arrow form) =
 lam [ma]
  forall2 [m][dsc] controls @ ma @ (module_dsc @ m @ dsc).

valid_sig_component_dsc : tm (worldview arrow list module arrow component_dsc arrow form) =
 lam3 [ma][m][dsc]
  property_authority @ ma and
  exists [maKey]
```

```
    keycert @ ma @ maKey and
    signed @ maKey @ (module_dsc @ m @ dsc).

signed_by_auth : tm (rel3 (list module) component_dsc property) =
 lam3 [m][dsc][prp]
  exists3 [ma][prqSet][rq]
   valid_sig_prp_auth @ ma @ prqSet and
   prqSet @ rq and rq @ prp and
   valid_sig_component_dsc @ ma @ m @ dsc.

signed_prp : tm (rel3 (list module) component_dsc property) =
 lam3 [m][dsc][prp]
  exists [dsc'] sub_component_dsc @ dsc' @ dsc and signed_by_auth @ m @ dsc' @ prp.

all_signed :
 tm (rel3 (list module) component_dsc (list property)) =
 lam3 [m][dsc][plist]
  forall [p] list_member @ plist @ p imp
   exists [p'] prp_eq @ p @ p' and signed_prp @ m @ dsc @ p'.

signed_component_dsc : tm (rel (list module) component_dsc) =
 lam2 [m][dsc]
  component_dsc_valid @ dsc and
  exists [dsc'] component_dsc_eq @ dsc @ dsc' and
                all_signed @ m @ dsc' @ (component_dsc_exprt @ dsc).

%%% property definitions

property : tp = tree num.
prp_kind = tree_root.
prp_eq : tm (eqrel property) = tree_eq @ eq_arith.
prp_valid : tm (property arrow form) = tree_valid @ eq_arith.

mk_prp : tm (ide arrow property) = lam [p] p.

prp_rq : tp = property arrow form.
mk_prp_rq : tm (property arrow (property arrow form)) =
 lam [rprp]
  lam [sprp] prp_eq @ rprp @ sprp.

%%% basic property definitions

prp_component_name : tm num.

mk_prp_component_name : tm (ide arrow property) =
 lam [cname]
  mktree @ eq_arith @ prp_component_name
         @ (list_cons @ (tree_eq @ eq_arith) @ cname @ list_nil).

rq_component_name_exists : tm (property arrow form) =
 lam [prp]
  prp_kind @ prp @ prp_component_name.
```

```
mk_rq_component_name : tm (ide arrow prp_rq) =
 lam [cname]
  lam [sprp]
   prp_kind @ sprp @ prp_component_name and
   prp_eq @ (mk_prp_component_name @ cname) @ sprp.

prp_export_id : tm num.
mk_prp_export_id : tm (list ide arrow property) =
 lam [idlist]
  mktree @ eq_arith @ prp_export_id @ idlist.

rq_export_id_exists : tm (property arrow form) =
 lam [prp]
  prp_kind @ prp @ prp_export_id and
  list_is_not_nil @ (tree_list_of_subtrees @ eq_arith @ prp).

prp_code_hash : tm num.
mk_prp_code_hash : tm (list module arrow property) =
 lam [mlist] mktree @ eq_arith @ prp_code_hash @ mlist.

prp_code_hash_modules : tm (property arrow (list module)) =
 tree_list_of_subtrees @ eq_arith.

rq_code_hash_checkable : tm prp_rq =
 lam [prp]
  prp_kind @ prp @ prp_code_hash and
  valid_chash_list @ (prp_code_hash_modules @ prp).

%%% module definitions

module : tp = tree num.
mk_module : tm (ide arrow hash_code arrow module) =
 lam2 [fname][hcode]
  mktree @ eq_arith @ hcode
         @ (list_cons @ (tree_eq @ eq_arith) @ fname @ list_nil).

module_eq : tm (eqrel module) = tree_eq @ eq_arith.

get_module_name : tm (rel module ide) =
 lam2 [m][fname]
  tree_get_subtree' @ m @ (list_cons @ eq_nat @ zero @ list_nil) @ fname.

get_hash_code : tm (rel module hash_code) = tree_root.

valid_chash : tm (module arrow form) =
 lam [m]
  exists3 [fname][hcode][hcode']
   get_module_name @ m @ fname and
   get_hash_code @ m @ hcode and
   calc_hash @ fname @ hcode' and
   hash_code_eq @ hcode @ hcode'.

module_valid = tree_valid @ eq_arith.
```

```
module_list : tp = list module.
module_list_eq = list_eq @ module_eq.
module_list_valid : tm (module_list arrow form) = tree_valid_subtrees @ eq_arith.

valid_chash_list : tm (module_list arrow form) =
 lam [mlist]
  forall [m] list_member @ mlist @ m imp valid_chash @ m.

hash_code : tp = num.
hash_code_eq : tm (eqrel hash_code) = lam2 [h1][h2] eq_arith @ h1 @ h2.

calc_hash : tm (rel ide hash_code).

%%% identifier definitions

ch_a : tm num.
ch_b : tm num.
...
ch_y : tm num.
ch_z : tm num.
ch_underscore : tm num.
ch_dash : tm num.
ch_space : tm num.

ide = tree num.

mk_ch : tm (num arrow ide) =
 lam [ch] mktree @ eq_arith @ ch @ list_nil.

mk_str : tm (num arrow ide arrow ide) =
 lam2 [ch][str] mktree @ eq_arith @ ch
                       @ (list_cons @ (tree_eq @ eq_arith) @ str @ list_nil).

ide_valid = tree_valid @ eq_arith.
ide_eq = tree_eq @ eq_arith.
ide_list_valid : tm (list ide arrow form) = tree_valid_subtrees @ eq_arith.
```

Bibliography

[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. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science*, June 2001.

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

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

[5] A. W. Appel, N. G. Michael, A. Stump, and R. Virga. A trustworthy proof checker. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.

[6] L. Bauer. *Access Control for the Web via Proof-Carrying Authorization.* PhD thesis, Princeton University, 2003.

[7] 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.

[8] 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.

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

[10] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 266–277. ACM Press, January 1997.

[11] CCITT. Recommendation X.509: The directory - authentication framework. Technical report, CCITT Blue Book, 1989.

[12] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[13] D. Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, 1997.

[14] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[15] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete algorithms. *IEEE Transactions on Information Theory*, 31(4):469–472, June 1985.

[16] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software Practice and Experience*, 30(15):1609–1640, 2000.

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

[18] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[19] M. R. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[20] J.V.Guttag, J. Horning, S. Garland, K. Jones, A. Modet, and J.M.Wing. Larch: Languages and tools for formal specification, January 1993.

[21] E. Lee and A. W. Appel. Secure Linking: a framework for trusted software components (extended version). Technical Report CS-TR-663-02, Department of Computer Science, Princeton University, September 2002.

[22] E. Lee and A. W. Appel. Policy-enforced linking of untrusted components (extended abstract). In *Proceedings of the 9th European Software Engineering Conference and the 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Helsinki, Finland, September 2003.

[23] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addision Wesley, second edition, 1999.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[25] Microsoft. *Authenticode*. http://www.microsoft.com/technet/security/topics/-secapps/authcode.asp.

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

[27] Microsoft. *Introduction to Code Signing.* http://msdn.microsoft.com/workshop/-security/authcode/intro_authenticode.asp.

[28] Microsoft. *Signing and Checking code with Authenticode.* http://msdn.microsoft.-com/workshop/security/authcode/signing.asp.

[29] Microsoft, http://www.microsoft.com/technet/security/bulletin/MS02-065.asp. *Microsoft Security Bulletin MS02-065*, November 2002.

[30] National Institute of Standards and Technology. *SECURE HASH STANDARD*, April 1995. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

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

[32] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419, pages 61–91. Springer-Verlag, Berlin, 1998.

[33] N. I. of Standards and Technology. *Digital Signature Standard (DSS)*, May 1994.

[34] E. Organick. *The Multics System: An Examination of its Structure.* MIT press, 1972.

[35] S. Owre, N. Shankar, J. Rushby, and D. W. J. Stringer-Calvert. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001.

[36] M. Pawlan and S. Dodda. Signed applets, browsers, and file access, April 1999.

[37] F. Pfenning and C. Schuermann. *Twelf User's Guide, Version 1.4.* http://www-2.cs.cmu.edu/t̃welf/guide-1-4/twelf_toc.html, December 2002.

[38] 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.

[39] B. Pientka. Termination and reduction checking for higher-order logic programs. In *First International Joint Conference on Automated Reasoning(IJCAR)*, LNCS. Springer, 2001.

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

[41] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[42] A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.

[43] P. Sewell and J. Vitek. Secure composition of insecure components. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[44] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[45] R. N. Srinivas. *Java Security Evolution and Concepts, Part 3: Applet Security.* http://developer.java.sun.com/developer/technicalArticles/Security/% -applets, December 2000.

[46] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.3 Documentation.* http://-www.javasoft.com/j2se1.3/docs.html.

[47] VeriSign Inc. *Code Signing Digital IDs for Sun Java Signing*, 2001.

[48] 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.

[49] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.