Policy-Enforced Linking of Untrusted Components (Extended Abstract) *

Eunyoung Lee Andrew W. Appel

Department of Computer Science
Princeton University

{elee,appel}@cs.princeton.edu

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, assertion checkers*

General Terms

Languages, Security, Verification

Keywords

Linking, component composition, formal logic, proof-carrying

1. INTRODUCTION

Large software systems are often built from loosely-coupled subsystems. When a programmer uses a third-party software component as a building block of her system, she doesn't want the code she imports to break the whole system. She needs some methods guaranteeing that linking the foreign software component to her system is safe.

The most widely used methods for ensuring safe linking are type checking and code signing. Checking the type of the interfaces between two software components ensures that two components agree on the types they are using. Although type checking is quite strong and easy to use, it doesn't guarantee that the code will behave in an expected way. Different static checking mechanisms have been suggested to address specific security properties of programs: a security-sensitive type system [5], wrappers which encapsulate untrusted programs and implement security-concerned properties [7], and so on. 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.

*This material is based upon work supported by the National Science Foundation under Grant No.9974553 and by DARPA award F30602-99-1-0519.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland

Code signing ensures that someone trustworthy trusts the code, but it is not always enough for guaranteeing system safety since trusted software companies or software developers unintentionally make mistakes. Software signed by trusted companies can still cause security holes in users' systems. For example, in November 2002 Microsoft released a badly coded ActiveX control, signed (as usual) with Microsoft's code-signing key; the bug led to a security vulnerability. Because Microsoft's code-signing protocol is insufficiently expressive, Microsoft was faced with the choice of 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. Microsoft chose the latter; it recommended that users who desired a secure system should remove *Microsoft* from Internet Explorer's Trusted Publisher List [4].

We propose Secure Linking (SL), a flexible way of allowing software component users to specify their security policy at link time, giving the users more control than type-checking or traditional digital signing. Our Secure Linking mechanism would not prevent bugs in ActiveX in the previous example, but it would give the software provider and the software consumer finer-grain control of the meaning of certificates they use. We have developed a logical framework for SL, providing stronger support for system safety and security, and we have implemented a prototype system using the framework.

With the SL framework, a code consumer can establish a linking policy to protect itself from malicious code from outside. The policy can include certain properties which the code consumer thinks useful for system safety: software component names, application-specific correctness properties, version information of software components, and so on. To link and to execute a component in a SL-enabled system 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.

In this extended abstract, we present the main concept of our proposed Secure Linking, and outline the design of the

framework and the underlying linking logic. We also briefly discuss the expressiveness of the linking logic in terms of its extensibility and interoperability.

2. SECURE LINKING

Suppose that Bob finds a seemingly interesting HTML document from Alice's system, and is asked to download a JavaScript program from Alice's system in order to read that document. Should Bob download and install Alice's code or not? Is it possible for Bob to specify and enforce his own linking policy, which is more flexible than type-checking and more expressive than traditional code-signing at link time? We will describe how our Secure Linking works to allow Bob to do what he wants. Bob plays as a *code consumer*, in this scenario, because he wants to link outside software components to his system, and Alice plays as a *code provider* because she is providing suspicious-before-verified code to Bob.

Secure linking policies. A code consumer can specify what kind of useful behavior he expects from outside software components in his linking policy. Foreign software components are allowed to run in the code consumer's system, only when they can prove that they have certain properties specified in the linking policy.

Code consumers can specify the policies such as: "`Game` imports (links to) only version 1.3 of `GUI`," "`Compiler` links to any version of `SymbolTable` that has `efficientLookup` property as certified by `underwritersLab`," "`Compiler` links only to the particular implementation of `SymbolTable` whose machine code hashes to 8327518932," "All modules imported by `Game` must have been mechanically inspected by a virus detector," "Certificate Authority `Alice` can vouch for the public key of the virus detector," "Untrusted modules must have been checked by a bytecode verifier to assure that they respect their interfaces."

Linking decision. A code consumer must check that a foreign component provides all properties in his linking policy in order to decide to link the component to other components in his system. This sometimes costs the consumer's time and resources. Secure Linking requires the consumer only to verify the proof submitted by a code provider with the software component; code consumers are relieved from the burden of proving. For example, Bob checks the proof from Alice with the certificates by using a trusted proof checker, and links her component to other components in his system if the proof is valid; otherwise he rejects it. Since the certificates Alice submitted are digitally signed, all signatures are verified during the proof-checking time.

Properties. Properties a code consumer wants to enforce at link time are specified in his linking policy. A *property* of a component is an assertion of expected behavior from the component. There are many 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," or "this software component doesn't produce any arithmetic overflow or underflow."

Third-party authorities. Some properties, like the property of being type checked, can be guaranteed by a trusted compiler, while others cannot be proved easily. These properties, however, may be accepted 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. We call those assurances *property certificates*, and third-party authorities *property authorities*. The property statements in property certificates are accepted as true after the digital signatures on them are verified, and the components are considered to have the properties mentioned in those certificates.

Since all certificates from property authorities come with digital signatures, a code consumer should know the public keys of the signers in order to check the validity of the digital signatures. Otherwise, the code consumer must have at least one trusted authority who can provide the right public keys for verifying the signatures and her public key. These authorities are called *key authorities* (also known as certificate authorities).

Just as a code consumer doesn't have to know all the public keys of principals, he doesn't have to know which property authorities can guarantee his required properties. Instead a code consumer specifies that he trusts a principal as someone who will let him or any code provider know the property-authority bindings. These authorities are called *property servers*. In this way, the code consumer doesn't have to enumerate the names of property authorities for every required property in his linking policy, and he need not modify the linking policy whenever a property-authority binding changes.

Library. Although it is possible to keep a software component self-contained, it is very common for a software component to use pre-installed software components by importing them. A code consumer enumerates what library components he has and what properties are exported by each of those components. At the same time, a component from a code provider declares what components it imports and what properties are required for each of the imported components. The SL framework checks whether or not the import requirement of a foreign component is satisfied by the library components a code consumer provides.

In what follows, we will explain the SL framework and its linking logic. The SL framework is independent of programming languages or programming environments; thus, the explanation of the framework is language-neutral.

3. DESIGN OF THE FRAMEWORK

The framework consists of the linking logic, a proof checker, a linking policy description language and its parser, a component description language and its parser, a tactical prover and other proving-helping tools, as shown in Figure 1. In this section, we explain the main parts of the framework except for the linking logic. The linking logic is presented in detail in Section 4.

Linking policy. Linking policies are set by code consumers (e.g., system administrators or component integrators) and specify what software components may be linked together. The linking policy description language provides a simple and convenient way of stating linking policies. This language adopts XML syntax, and its parser produces the linking logic formulas from a linking policy. The syntax and semantics of the language are explained in our technical report [3].

A code consumer's linking policy usually consists of three

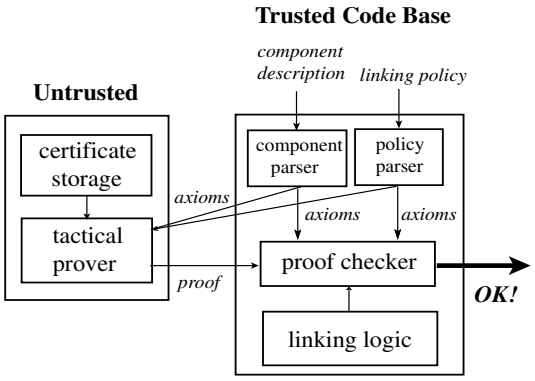

Figure 1: System Diagram

parts: a list of useful properties, the names of trusted authorities, and the description of pre-installed library components. The *required property* part is used for specifying the properties requested from outside components, and consists of a list of property names. The list of *trusted authority names* consists of the names of trusted key authorities and property servers. This list is used for getting public key certificates and property certificates from authorities to build a proof. The *library* part is a list of component descriptions, each of whose elements describes one library component of a code consumer's. By enumerating available library components explicitly, a code consumer can hide some security-critical software components from outside view; it prohibits outside components from accessing these security-critical components to attack the system.

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

Component description. Component descriptions are prepared by code providers, giving the information of the components they submit. The component description language in the framework is designed to help code providers describe their components. It also adopts XML syntax, coming with a parser to the linking logic. The full description of its syntax and semantics is given in [3].

A component description in the SL framework consists of four parts: a component name, modules, exports, and imports. The *component name* is a local identifier of convenience for the component. The *module* part is a set of code files which implement the component. Each code file is represented by its file name and cryptographic hash code. The *export* part of a component description specifies what should be visible outside of the component. Components in the SL framework can export properties (by their names) as well as identifiers with type information (e.g., class and method names). The *import* part shows what other components a component depends on. An import request of a component consists of the name of a dependent component and some required properties. The framework locates a component matching an import request and checks if it exports all the required properties of the import request. By allowing exporting and importing properties as well as class and method identifiers, our framework gives a linker more information than types, and makes linking safer.

A component description can be built by combining other component descriptions. It is useful to make it possible to combine component descriptions, especially when reasoning with digitally-signed certificates from property authorities. When signing, it is reasonable for a property authority to want to sign only on the properties he can guarantee, rather than sign on all the properties a component description exports. After collecting component descriptions assured by property authorities, a code provider combines them, to build a complete component description. This frees the property authorities from a burden of assuring more properties of a component description than they want to.

Proof checker. Given a proof from a code provider, a code consumer must be able to verify the validity of the proof. The linking logic of SL is built on top of the PCA logic [1]. Since the PCA logic itself is an object logic of LF, a logical framework which allows the specification of logics [2], every term in the linking logic boils down to a term in the underlying LF logic. Therefore, the proofs written in the linking logic can be checked mechanically by any trusted checker implementing LF.

Tactical prover. We have developed a tactical prover for our linking logic. Note that the prover doesn't have to be trusted since a proof which is invalid will be rejected by the trusted proof checker. Although the prover itself is not contained in the trusted computing base (TCB) of the SL framework, it is worthwhile to develop a prover and to include as a part of the framework in order to help code providers to build proofs in the linking logic. The tactical prover in the SL framework is a logic program consisting of 30 tacticals and 58 tactics, running on Twelf [6]. The goal to be proved is encoded as the statement of a theorem, and axioms that are likely to be helpful in proving the theorem are added as assumptions. The prover generates a derivation of the theorem; this is the proof that a code provider must send to a code consumer.

We have shown that our prover is sound (i.e., it is guaranteed that every formula that is provable by the prover is true in the linking logic). Our prover is also complete for the set of true formulas generated by the linking policy description language and the component description language (i.e., it is guaranteed that every true formula in the set is provable by the prover). Although the underlying higher-order logic, in which the linking operators are encoded, is expressive enough to be undecidable, the logic covered by the tactical prover is an example of an application-specific sublogic which is decidable, and for which we have demonstrated a decision procedure. The complete proof can be found in our technical report [3].

4. LINKING LOGIC

In this section we explain a secure linking theorem of SL, description parsing, and the representation of properties; then we discuss the soundness of the linking logic.

Secure linking theorem. Before allowing linking, a code consumer requires a proof saying that the component from a code provider satisfies his linking policy. In other words, the proof must show that a set of modules and its component description satisfy the secure linking theorem of the SL framework. The secure linking theorem is written in the linking logic, and reflects the steps the SL framework follows to make a linking decision.

$$\frac{signed\_component\_dsc(m, dsc, rqSet) \quad safe\_imports(dsc, lib, libdsc) \quad exports\_required\_prps(rqSet, dsc)}{ok\_to\_link(m, dsc, lib, libdsc, rqSet)} \text{ [SL theorem]}$$

The above rule shows the secure linking theorem. First, the framework examines if the logical description of a component and the set of modules have been tampered with by checking cryptographic hash codes and digital signatures; it generates a set of axioms asserting the binding between the modules and the component description. With these axioms, the code provider should prove that the predicate *signed_component_dsc* holds. Second, the code consumer wants the component to import only components visible in his linking policy; if so, it is possible for the code provider to prove that the predicate *safe_imports* holds. Last, the code consumer requires that the component description exports all the required properties; if so, the code provider can prove that the predicate *exports_required_prps* holds. If the component description and the set of modules satisfy the above three conditions, that is, if it is provable that those three predicates hold, then the secure linking theorem is provable, and linking is allowed; otherwise, linking is denied.

Description parsing. With a given linking policy, the parser turns the name bindings of property servers and of key authorities into axioms in the linking logic. A library component is encoded as a set of properties, which are exported by the library component. Each required property specified in a linking policy is translated into a *property request*, which will be explained later.

A component description is turned into formulas in the linking logic after checking the hash codes of binary modules. Digitally signed statements of certificates from key authorities, property authorities, or property servers are converted into axioms in the linking logic after verifying their signatures. A component description is encoded as a pair of its export and its import in the linking logic. The name of a component as well as the set of exported type identifiers and the set of exported properties are all treated as properties, making up the export part of the component. The import part of a component is encoded as a list of sets of property requests.

Properties. Property matching happens in two places at link time: a code consumer checks if a foreign software component exports all the properties in the linking policy, and if the foreign component imports only visible library components of the code consumer.

Since property matching occurs very frequently in proving the secure linking theorem, it is important to design the logic keeping property matching easy and simple. We achieved the goal by dividing property matching into two parts, *properties* and *property requests*. In our design, a property request is a predicate accepting a property as an argument; the predicate returns true if the argument matches the request it implements, or returns false otherwise. Therefore, checking property matching is turned into simple evaluation of a predicate.

We gained another benefit by designing property requests as predicates: it allows one property to have more than one property request with different semantics. It enriches the expressiveness of the SL framework. At link time, for example, component names are typically used in two different ways: a code consumer may require a component to have some name - any name - or to have a specific name for an exact match. These usages can be implemented in the linking logic by designing two different property requests working on one property, rather than designing two properties separately.

Soundness. It is desirable to prove our linking logic *sound*, i.e., that untrue formulas cannot be proved. Soundness is typically proved by induction over all proofs that can be built from a given set of inference rules. PCA takes a different approach, however: each application-specific operator is defined in terms of the underlying operators of LF higher-order logic [2], and each inference rule is proved as a theorem of the higher-order logic. Therefore, the soundness of PCA logic depends on the already-proven soundness of LF higher-order logic.

We have chosen PCA as the underlying logic of the linking logic and used the same approach that PCA took in order to prove that the linking logic is sound. In other words, the soundness of PCA and its underlying LF logic proves the soundness of our linking logic because all the operators in the linking logic are written in those two logics, and all the inference rules in the linking logic are proved as lemmas of those logics.

5. FURTHER DISCUSSION

The SL linking logic is so general that it is easily extended to give a formal description to real-world linking systems. We demonstrated the expressiveness of our linking logic by encoding the linking system of .NET as an example. We have modeled .NET assemblies as components, and .NET's strong naming, version-redirection, hash code verification of binary code files, and code-signing as properties and property requests in the linking logic. With these properties, the linking procedure of .NET assemblies is reduced to a case of Secure Linking requiring a specific set of properties. The detailed discussion can be found in our technical report [3]. We also discussed how our framework can interoperate other logic-based frameworks concerning system security, and what benefits can be gained.

6. REFERENCES

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

[2] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of ACM*, January 1993.

[3] E. Lee. *Secure Linking: A Logical Framework for Policy-Enforced Linking*. PhD thesis, Princeton University, to appear 2003.

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

[5] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. *Mobile Agents and Security*, Springer-Verlag, 1998.

[6] 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)*, July 1999.

[7] Sewell and Vitek. Secure composition of insecure components. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*, 1999.