19-1-R02

안전한 소프트웨어 콤포넌트 조합을 위한 논리적 프레임워크

이 은 영
동덕여자대학교 정보과학대학 컴퓨터전공
elee@dongduk.ac.kr

요 약

소프트웨어 콤포넌트를 이용하여 시스템을 구성하는 경우, 서로 다른 버전들과 디지털 서명, 정적인 타입 정보나 네트워크로 전송된 소프트웨어, 그리고 서로 다른 판매자에 의한 콤포넌트들이 혼재함으로 인해서, 콤포넌트를 링크하는 과정은 그다지 간단하지 않을 수도 있다. 링크과정에 적용될 수 있는 정책의 수립이 이러한 복잡함을 해결할 수 있는 좋은 수단이 된다. 본 논문에서는 링크 시 적용될 수 있는 정책을 명시하는데 쓰일 수 있는 프레임워크를 소개한다. 프레임워크 Secure Linking은 “Linking Logic”이라고 명명한 higher-order logic에 기반하고 있으며, Linking Logic의 soundness와 modularity는 사용자들이 링크 정책을 세우고 적용하는 것을 용이하게 한다. 본 논문에서는 링크 정책과 증명을 생성하는데 바탕이 되는 Linking Logic을 정의하고, 이를 기반으로 한 프레임워크를 구현하였다. 그리고 구현된 Secure Linking 프레임워크를 이용하여 Microsoft의 .NET에서 사용되는 assembly의 링크 과정을 설명함으로써 Linking Logic이 실제로 사용되는 링크 시스템을 나타낼 수 있을 만큼 풍부한 표현력을 가지고 있음을 증명하였다.

1. 서 론

1.1. 연구 배경

거대한 소프트웨어 시스템은 대체로 서로 독립적인 서브시스템을 바탕으로 만들어진다. 각각의 콤포넌트들이 가지는 상대적인 독립성은 각각의 콤포넌트가 개별적으로 개발되고, 실제의 구현내용에 구애받지 않고 공개된 인터페이스만을 통해서 상호작용을 하는 것이 가능하게 만든다. 만약 어떤 프로그래머가 외부에서 제작된 소프트웨어 콤포넌트를 구입해서 자신이 만드는 시스템의 일부로 사용한다며, 그 프로그래머는 외부에서 들여온 코드가 잘 못 동작하여 자신의 시스템에 해를 입히는 것을 원치 않을 것이다. 다시 말해서 코드를 들여오는 프로그래머는 외부 소프트웨어 콤포넌트를 자신의 시스템에 링크시키는 것이 안전하다는 것을 보장해 줄 수 있는 어떤 장치를 원하게 된다.

안전한 링크를 위해서 현재 가장 널리 쓰이고 있는 방법은 타입 체크와 코드 서명이다. 2개의 소프트웨어 콤포넌트를 링크하기 전에 인터페이스의 타입을 체크하는 것은 해당 콤포넌트들이 그들이 사용하는 타입에 있어서는 일치한다는 것은 보장한다. 그렇지만 사용하기 편리하고 그로 인

해서 보장되는 내용이 강력함에도 불구하고, 타입 체크 자체는 코드가 프로그래머나 사용자가 예상하는 방향으로 동작할 것이라는 것을 보장해주지는 못한다는 단점을 가지고 있다.

코드 서명을 사용하는 경우는 코드가 사용자가 믿을 수 있는 소스로부터 왔다는 것은 보여줄 수 있으나, 서명자가 서명을 함으로써 코드의 어떤 속성을 보장했는지 명시적으로 보여주지는 못한다: “잘못된 대화 상자를 절대 띄우지 않는 프로그램임을 보장”하는 것인가, 아니면 “절대로 버퍼 오버런(buffer overrun)이 일어나지 않는 프로그램임을 보장”하는 것인가?

이러한 문제는 해결하기 위해서는 지금까지 사용하던 방법보다 더 강력한 보장을 해줄 수 있는 방법이 필요하다. 코드 사용자(code consumer)는 자신의 시스템에 링크하고자 하는 모든 콤포넌트들이 가져야 할 속성을 명시할 수 있게 되기를 원할 것이다. 동시에 코드 사용자는 코드 제공자(code provider)가 자신의 대신하여 콤포넌트의 요구된 속성을 증명해 주기를 바랄 수도 있다.

본 논문에서는 링크 시에 이제까지 사용되었던 어떤 방법보다 시스템 안전과 보안을 보장해 줄 수 있는 방법을 제안하고, 이를 구현할 수 있는 논리적 프레임워크를 제안하고자 한다. 논리적 프레임워크는 Secure Linking이라고 불리며, higher-order logic의 일종인 Proof-Carrying Authentication(PCA) 로직[2]을 기반으로 하고 있다. Secure Linking 프레임워크를 이용하여 코드 사용자는 링크 시에 자신의 시스템을 외부의 악의적인 코드로부터 보호하는 정책을 만들어 미리 알릴 수 있다. 사용자의 정책은 사용자가 시스템의 보호를 위하여 필요하다고 생각하는 특정한 속성을 포함한다. 예를 들면 소프트웨어 콤포넌트의 이름, 어플리케이션에 특화된 correctness 속성, 혹은 소프트웨어 콤포넌트들의 버전 정보 등이 이러한 속성에 해당된다고 볼 수 있다. 코드 사용자의 시스템에 주어진 콤포넌트를 링크시키고 실행시키기 위해서, 코드 제공자는 콤포넌트와 함께 주어진 콤포넌트가 사용자가 링크 정책에서 요구하는 속성을 가지고 있다는 증명을 함께 보내야한다. 여기에서 요구되는 증명은 프레임워크에서 제공되는 기본 로직과 추론 규칙(inference rules)을 통해서 만들어진다. 코드 제공자가 보낸 증명을 받은 사용자는 작고 믿을 수 있는 체커(checker)를 이용해서 증명을 검사하고, 만약 증명이 참이라고 검증되면 주어진 콤포넌트를 자신의 시스템에 있는 다른 콤포넌트들과 링크시킨다.

본 논문에서는 우선 Secure Linking 프레임워크의 디자인 결정에 대하여 논의하고, 그 뒤에 이러한 디자인이 어떻게 링크 로직으로 표현되었는지를 설명한다. 그 이후 Secure Linking 프레임워크을 이용하여 .NET의 링크 시스템을 이론적으로 정의할 수 있다는 것을 보임으로써 프레임워크가 포함하고 있는 링크 로직이 강력하다는 것을 증명할 것이다. 그리고 마지막으로 .NET의 링크 시스템을 이론적으로 정의하면서 알게 된 .NET 시스템의 문제점에 대하여 논의하도록 하겠다.

2. 관련연구

2.1 콤포넌트 모델

콤포넌트 간의 상호작용을 일정한 수준으로 제한하는 일은 콤포넌트 간의 잘못된

사용이나 악의적은 공격을 막기 위해서 반드시 필요하다. 전통적으로 이제까지 프로그램은 추상적 데이터 타입(Abstract Data Type)이나 정보 은닉(information hiding)을 통하여 다른 프로그램과의 상호 작용에서 보호되었다.

객체지향 프레임워크인 COM이나 객체지향 언어인 자바의 경우, 콤포넌트는 오브젝트(object)에 기반하고 있다. 이들은 클래스나 메소드, 멤버 데이터 등을 접근의 단위로 삼는다. SML/NJ에서 제공하는 Compiler Manager(CM)를 사용하면 소프트웨어 콤포넌트에 대한 접근 단위를 확장시키는 것이 가능하다. CM 내부에서 프로그램 모듈은 계층을 이루는데, 계층 안의 모듈은 자신의 하위의 모든 모듈에 대한 정보를 감추고 자신이 원하는 인터페이스만을 그룹(group)이라는 단위를 통해서 공개할 수 있다.

Bauer와 Appel, Felten은 CM의 이러한 계층적 모듈을 자바의 패키지에서 이용할 수 있도록 하는 시스템을 개발하였다[4]. 또한 Reid 등은 C언어를 위한 콤포넌트 모델인 유닛(units)을 제안하였다[14].

2.2 바이트코드 검증

자바는 코드 사용자가 실행될 코드에 대한 타입 체크를 링크 직전에 하도록 하는 첫 번째 시스템이었다. Necula는 Proof-Carrying Code[11]에서 체크의 단위를 바이트 코드에서 기계어로 바뀌었으며, Devanbu는 [7]에서 코드 사용자가 증명에 대한 검증을 하는 것이 비효율적이라고 생각되는 경우, 검증을 코드 제공자의 믿을 수 있는 보조프로세서(coprocessor)에서 수행하고, 그 결과를 디지털 서명이 된 증명서와 함께 보내는 방법에 대하여 논의하였다.

2.3 .NET 프레임워크

.NET 프레임워크는 마이크로소프트가 개발한 플랫폼으로 인터넷을 중심으로 한 고도로 분산된 환경에서의 컴퓨팅을 위해서 개발된 환경이다[13].

.NET의 Common Language Runtime (CLR)은 자바 언어 스타일의 바이트코드 검증기법을 채택하고 있다. 또한 .NET은 어셈블리(assembly)라고 불리는 새로운 환경 관리(configuration management) 방식을 제안하고 있다. 어셈블리는 링크의 기본 단위가 되며, 그 내부에 버전 정보를 비롯한 링크에 필요한 다른 콤포넌트들에 대한 정보를 관리하고 있다.

2.4 Proof-Carrying Authentication

Secure Linking 프레임워크는 Appel과 Felten이 제안한 PCA 로직[2]에 바탕으로 만들어졌다. Proof-Carrying Authentication (PCA)은 Necula가 제안한 proof-carrying 방식[11]으로 만들어진 분산 환경에서의 인증 프레임워크(authentication framework)이다. PCA는 기존의 분산 인증 프레임워크들과 2가지 면에서 차이점을 보이고 있다. 첫째, PCA는 higher-order logic을 사용하여 좀 더 일반적이고 다용적이다. 둘째, 서버가 클라이언트의 요구에 대한 승인을 내리기 위해서 복잡한 결정과정을 모두 거칠 필요가 없다. 대신 클라이언트는 자신이 원하는 리소스에 대한 권한이 있음을 서버에게 보이는 책임을 진다.

분간 인증 프레임워크나 프로토콜들은 꾸준하게 정형 로직(formal logic)을 이용해 오고 있으며[1], 그 대표적인 예가 Taos 분산 운영체제이다. Taos 시스템[15]의 인증 로직은 propositional calculus를 바탕으로 만들어졌으며, 로직의 soundness가 증

명되었다. 또한 실제 구현에서는 제안된 인증 로직의 decidable subset만을 사용함으로써 클라이언트의 요청에 대한 승인 요청에 대한 검증이 항상 결정이 나도록 고안되었다.

결정이 가능한 로직(decidable logic)은 일반적인 로직보다 표현할 수 있는 범위가 좁기 때문에, 이의 사용은 인증 로직 자체의 표현 범위를 좁히는 결과를 가져왔다. 또 기존의 프레임워크들은 응용프로그램에 특화된 로직을 만들기 위해서는 새로운 추론 규칙들을 더하고, 확장된 로직의 soundness를 새로이 증명해야하는 불편함을 가지고 있었다.

PCA 로직은 술어논리(predicate logic)에 quantification을 더함으로써 표현의 다양성을 얻을 수 있었다. 따라서 이 인증 프레임워크는 하나의 세트로 이루어진 추론 법칙들을 가지고 모든 응용 프로그램 특유의 인증 논리를 표현할 수 있다는 장점을 가지고 있다. PCA에서 응용프로그램의 인증 법칙들은 레마의 형태로 표현되고 증명된다.

이와 같은 PCA 접근 방식의 또 다른 장점은 응용 프로그램들이 같은 추론 법칙들을 공유함으로써, 상호변환이 손쉬워졌다는 점이다. 이것은 PCA가 기존의 어떤 인증 프레임워크보다 더욱 일반적이라는 것을 증명한다. 그렇지만, 클라이언트의 요청에 대한 해답을 찾는 것이 항상 가능하지만은 않다는 단점을 PCA는 가지고 있다. 그것은 PCA가 사용하고 있는 higher-order 로직의 한계점이기도 하다. 이 문제를 해결하기 위해서 PCA는 요청에 대한 증명을 클라이언트가 하고, 서버는 단지 그 증명을 검증하는 방식을 택하고 있다. 이것은 Necula의 proof-carrying code의 연장선으로 볼 수 있을 것이다. PCA에 기반을 둔 시스템으로는 Bauer와 Schneider, Felten이 개발한 웹 인증 시스템을 들 수 있다[5].

3. 예 제

이 장에서는 예제를 통하여 Secure Linking 프레임워크의 디자인에 대하여 간단히 설명하고자 한다. Alice라는 주체(principal)가 compiler라는 이름의 콤포넌트를 가지고 있다고 가정해 보자. Alice는 콤포넌트를 작성한 프로그래머일 수도 있고, 외부에서 그 콤포넌트를 구입한 단순한 사용자일 수도 있다. Alice는 자신이 가지고 있는 compiler 콤포넌트를 또 다른 주체인 Bob에게 보내서 Bob의 시스템에서 실행시키기를 원한다. 한편 Bob은 자신의 시스템이 외부의 프로그램으로부터 공격당하는 것을 원치 않으며, Alice의 프로그램을 전적으로 신뢰할 수 없기 때문에 Alice가 그녀의 프로그램이 Bob에 시스템에 링크되어도 안전하다는 증명을 함께 보내줄 것을 요구하게 된다. 증명을 위해서 Bob은 자신의 링크 정책을 Alice에게 알려준다. 링크 정책은 크게 3가지 부분으로 나누어진다: Bob의 시스템 내부에 존재하는, 공개된 소프트웨어 콤포넌트(라이브러리라고 불린다), Bob이 외부 소프트웨어 콤포넌트에 요구하는 일련의 속성들, 그리고 Bob이 신뢰하는 외부 권위자들의 이름이다.

3.1 속성(Properties)

코드 사용자는 자신의 시스템을 보호하기 위하여 자신이 미리 선언한 속성들을 외부에서 오는 콤포넌트들이 가지고 있기를 원한다. Secure Linking에서 콤포넌트의 속성은 콤포넌트들이 가지는 행동에 대

한 확인의 형태로 표현된다. 시스템을 악의적인 공격으로부터 막는데 유용한 속성들은 여러 가지가 있을 수 있다. 예를 들면 “이 소프트웨어 콤포넌트는 타입 체크되었다”라든지 “이 소프트웨어 콤포넌트는 절대로 할당된 메모리 영역 외의 범위는 액세스하지 않는다” 혹은 “이 소프트웨어 콤포넌트는 사용자의 파일 시스템에서 어떤 정보도 읽거나 쓰지 않는다” 등의 속성들이다.

이 장에서 제시된 예제에서 Bob은 외부의 소프트웨어 콤포넌트를 받아들일 때는 항상 콤포넌트가 타입 체크되었다는 의미를 가진 prp_type_checked를 요구한다고 가정하자. 이제 Alice는 자신이 가지고 있는 콤포넌트 compiler를 Bob의 시스템에 링크시키기 위해서는 prp_ type_checked라는 속성을 가지고 있음을 보여야만 한다.

3.2 속성검증 권위자

위의 예제에서 사용되는 타입 체크에 대한 속성들처럼 어떤 속성들은 신뢰할 수 있는 컴파일러에 의해서 보장될 수 있지만, 어떤 속성들은 그렇게 쉽게 보장될 수 없다. 그렇지만, Secure Linking에서는 그러한 속성들도 믿을 수 있는 권위자들에 의한 확인이 주어진다면, 주어진 콤포넌트의 속성으로 받아들일 수 있다. 여기에서 사용되는 권위자라는 용어는 소프트웨어 검사(software audit)이나 소프트웨어 엔지니어링에서 사용되는 여러 가지 종류의 소프트웨어 검증 방법들을 지칭할 수도 있다. 이런 권위자들은 속성 검증 권위자(property authorities)라도 불리고, 권위자들이 만드는 확인은 디지털 서명을 포함된 보증서의 형태를 가지게 된다. 속성검증 권위자가 보낸 보증서는 보증서에 포함된 디지털 서명을 검증함으로써 진위가 검증된다. 즉 다시 말해서 디지털 서명이 검증되면, 코드 사용자는 주어진 콤포넌트가 보증서의 명시된 속성을 가지고 있다고 간주한다.

예제에서 Charlie는 소프트웨어 콤포넌트가 prp_type_checked라는 속성을 가지고 있는지를 검정하는 속성검증 권위자이다. Bob이 자신의 링크 정책에서 Charlie를 prp_type_checked 속성에 대한 속성검증 권위자로서 신뢰한다고 명시하였기 때문에, 이제 Alice는 Charlie로부터 그녀의 콤포넌트 compiler가 prp_type_checked라는 속성을 가지고 있다는 보증서를 얻어야만 한다.

3.3 키분배 권위자(Key Authorities)

속성검증 권위자가 발행하는 보증서는 디지털 서명과 함께 오기 때문에, 이를 검증하기 위해서 코드사용자는 서명자(여기서는 속성검증 권위자)의 공개키(public key)를 알아야만 한다. 공개키에 대한 보증서는 종종 체인 형태로 검증된다. 따라서 코드 사용자는 적어도 하나의 신뢰할 수 있는 키분배 권위자와 그의 공개키는 알고 있어야 한다. 키분배 권위자는 주체(principal)의 이름과 그의 공개키에 대한 보증서를 발행한다.

주어진 예제에서 Bob은 자신이 신뢰하는 하나 이상의 키분배 권위자를 가지고 있어야하며, 그에 대한 정보를 링크 정책에 명시하여야 한다. Charlie에게서 속성에 대한 보증서를 받고 검증하기 위해서, Bob은 Charlie의 공개키를 미리 알고 있을 필요가 없다. 단지 자신의 신뢰하는 키분배 권위자에게 Charlie의 공개키에 대한 보증서를 요구하고, 이 보증서를 검증하여 Charlie의 공개키를 알아낼 수 있다.

3.4 속성분배 서버(Property Server)

Bob은 자신이 신뢰하는 키분배 권위자의 공개키 외에는 어떤 공개키도 기억할 필요가 없는 것과 마찬가지로, 주어진 어떤 속성에 대하여 누가 그것을 검증해 줄 속성검증 권위자인지도 기억하고 있을 필요가 없다. 단지 그에게 속성검증 권위자에 대한 정보를 알려줄 수 있는, 믿을 수 있는 서버를 하나만 알고 있으면 된다. Secure Linking에서 이런 서버는 속성분배 서버(property server)라고 불린다. 속성분배 서버를 이용함으로써 코드 사용자는 속성검증 권위자의 이름을 링크 정책 내부에 언급할 필요가 없어지고, 또 속성과 그것의 검증 권위자의 바인딩이 변하는 경우에도 링크 정책을 수정하거나 새로 공표하는 수고를 덜게 된다.

<그림 1>의 예제에서 Bob은 자신이 신뢰하는 속성분배 서버(예를 들어, Emily)를 링크 정책 안에 명시하고, Alice는 Emily에게 자신의 콤포넌트 compiler의 속성을 검사해줄 수 있는 속성검증 권위자가 누구인지 질의하게 된다. 그러면 주어진 예제에서 Emily는 Charlie가 prp_ typed_checked를 검증해 줄 수 있는 권위자임을 Alice에게 알려줄 것이다.

3.5 라이브러리(Library)

하나의 소프트웨어 콤포넌트가 필요한 모든 코드를 포함하도록 프로그래밍할 수도 있으나 이것은 그리 바람직하거나 자연스러운 일은 아니다. 대신 대부분의 소프트웨어 콤포넌트는 빈번하게 다른 소프트웨어 콤포넌트를 호출하게 된다. 따라서 코드 사용자가 자신의 시스템에서 외부 사용자가 사용할 수 있는 콤포넌트들을 선별적으로 공개할 수 있다. 이것을 Secure Linking에서 라이브러리라는 이름으로 불린다. 동시에 코드 사용자는 외부에서 들어오는 모든 콤포넌트들에게 자신들이 호출하고자하

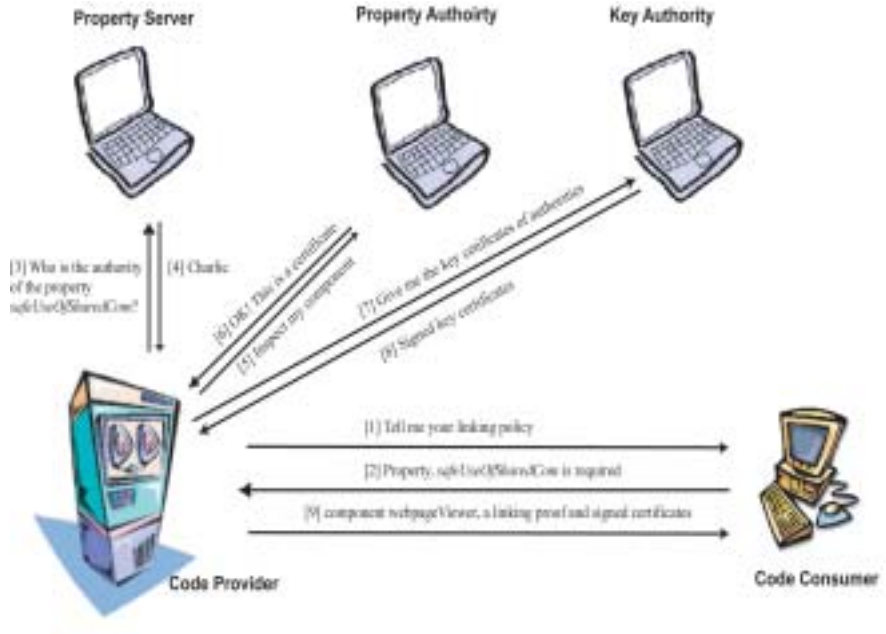

그림 1. Secure Linking의 예

는 콤포넌트들을 모두 명시하도록 요구하게 된다. 그리고 링크를 결정하는 과정에서 코드 사용자는 외부 콤포넌트들이 단지 자신이 공개한 라이브러리만을 호출하는지 검사한다. 만약 외부 콤포넌트들이 라이브러리에 포함된 콤포넌트 이외의 콤포넌트를 호출한다면, 링크는 거부된다.

코드 사용자가 라이브러리를 선별적으로 공개할 수 있게 함으로써, Secure Linking은 사용자가 자신이 원하는 시스템 콤포넌트나 취약한 콤포넌트를 외부 소프트웨어 콤포넌트가 접근하지 못하도록 막을 수 있다.

3.6 링크 결정과정

외부 소프트웨어 콤포넌트를 자신의 시스템에 링크할 지의 여부를 결정하기 위해서 코드 사용자는 코드 제공자가 제출한 증명을 검증하여야 한다. 증명의 검증은 함께 제출된 속성에 대한 보증서, 공개키에 대한 보증서 등과 함께 검증된다. 만약 증명이 검증되면, 코드 제공자가 제출한 소프트웨어 콤포넌트는 신뢰할 수 있다고 간주되고 링크가 허락된다.

4. 링크 모델

Secure Linking 프레임워크는 링크 과정을 설명하는데 있어서 2개의 개념적인 모델을 사용하고 있다. 하나는 소프트웨어 콤포넌트와 포함된 속성을 기술하는 모델이고, 나머지 하나는 코드 사용자의 링크 정책을 기술하는 모델이다. 각각의 모델은 XML 문법형태로 고안된 기술언어를 가지고 있으며, 프레임워크의 사용자들은 이 언어들을 이용하여 자신의 콤포넌트와 링크 정책을 기술하게 된다. <그림 2>는 콤포넌트 기술언어로 작성된 콤포넌트의 예를 보여주고 있다.

이렇게 기술언어를 써서 작성된 콤포넌트나 링크 정책에 대한 설명은 프레임워크 내부에 구현된 파서(parser)에 의해서 링크 로직으로 변환된다. 콤포넌트 기술 언어나 링크 정책 기술 언어에 대한 자세한 내용은 [9]에 기술되어 있다.

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

그림 2. Component Description

단순한 클래스나 메소드의 이름을 공개하는 것뿐만이 아니라, 적절한 속성들을 공개하고 도입함으로써 Secure Linking 프레임워크는 타입 이상의 정보를 링크 시에 이용할 수 있으며, 이를 통해서 소프트웨어 콤포넌트의 조합을 좀 더 안전하게 만들 수 있다.

링크 정책은 시스템 관리자(혹은 콤포넌트를 조합하는 사람)에 의해서 설정되며, 어떤 속성을 가진 콤포넌트들이 자신의 시

스템에 링크될 수 있는지를 명시한다. Secure Linking 프레임워크가 제공하는 링크 정책 기술 언어는 사용자가 간단하고 편리한 방법으로 링크 정책을 기술하는 것이 가능하게 한다. 링크 정책 내부에서 사용자는 자신이 공개하는 라이브러리 콤포넌트에 대한 정보, 신뢰하는 키분배 권위자에 대한 정보, 속성분배 서버에 대한 정보, 그리고 외부 콤포넌트에게 요구하는 속성들에 대해서 기술해야한다.

링크 정책을 프레임워크 내부에 포함시키지 않고, 코드 사용자가 따로 명시할 수 있게 함으로써, Secure Linking 프레임워크는 보다 다양한, 응용프로그램 고유의 문제들을 링크 정책으로 표현하는 것이 가능하게 하였다.

5. 링크 로직(Linking Logic)

Secure Linking 프레임워크의 핵심인 링크 로직은 PCA 로직[2]에 기반을 둔 higher-order 로직이다. 따라서 링크 로직에 정의된 오퍼레이터의 내용(semantics)은 PCA 로직에 정의된 오퍼레이터와 식을 이용하여 표현된다. 링크 로직의 모든 추론 규칙들은 레마의 형태로 표현되고, 모두 증명되었다. 이 장에서는 어떻게 사용자가 기술한 콤포넌트에 대한 설명과 링크 정책이 링크 로직으로 변환되고, 또 링크 결정과정이 어떻게 링크 로직으로 표현되었는지 논의하도록 하겠다.

5.1 Soundness

링크 로직은 링크의 기본 동작은 정의하는 오퍼레이션과 주어진 링크 정책에 따른 목적(goal)을 증명하는데 사용할 수 있는 추론 규칙들로 이루어져 있다. Secure Linking에서 코드 사용자는 자신의 링크 정책에 대한 증명을 요구하고 링크 로직으로 쓰여진 증명을 검증하게 되는데, 만약 링크 로직의 soundness를 증명할 수 있다면 (다시 말해서, 참인 아닌 정리는 증명될 수 없다면), 이는 링크 정책 및 시스템의 안정성에 커다란 도움이 될 것이다.

일반적으로 어떤 로직의 soundness는 로직 안에 포함된 모든 추론 규칙들을 가지고 만드는 증명들에 대한 귀납법적 접근을 통하여 증명한다. 그렇지만 PCA 로직의 경우는 약간 다른 방식으로 soundness를 증명하고 있다[2]. PCA에서 각각의 오퍼레이터는 기반이 되는 higher-order 로직의 오퍼레이터에 의해서 정의되고, 각각의 추론 규칙들은 바탕이 되는 higher-order 로직의 정리로서 모두 증명이 되었다. 따라서 기반이 되는 higher-order 로직의 soundness가 PCA 로직의 soundness를 보장하게 된다. 같은 맥락에서 링크 로직의 soundness 역시 증명될 수 있다. 링크 로직에 포함된 오퍼레이터들은 PCA 로직과 PCA 로직의 바탕이 된 higher-order 로직에 기반하고 있다. 또한 모든 추론 규칙들은 기반 로직의 정리의 형태로 정의되고 증명되었다. 따라서 PCA 로직과 기본 higher-order 로직의 soundness를 바탕으로 링크 로직의 soundness를 증명할 수 있다.

5.2 로직을 이용한 링크 결정과정 표현

코드 사용자의 시스템에 어떤 외부 소프트웨어 콤포넌트가 링크되기 위해서는 코드 제공자는 주어진 콤포넌트가 코드 사용자가 요구하는 모든 속성을 가지고 있다는 것을 반드시 보여야 한다. 링크 로직의 관점에서 봤을 때, 이것은 주어진 콤포넌트가 <그림 3>에 결과 부분에 명시된 Secure

Linking Theorem을 만족시킨다는 증명을 찾아가는 과정이라고 할 수 있다. Secure Linking Theorem에 나타난 predicate ok_to_link의 semantics는 어떤 과정을 통해서 코드 사용자가 링크 의사결정을 내리는지를 보여준다. <그림 3>에 주어진 추론 규칙은 링크 결정 과정에 대한 직관적인 이해를 돕는데 도움이 될 것이라고 생각된다.

signed_component_dsc(*m,dsc,prqset*)
provides_enough_prps(*dsc,lib,libdsc*)
exports_required_prps(*prqset,dsc*)

---

ok_to_link(*m,dsc,lib,libdsc,prqset*)

그림 3. Secure Linking Theorem과 추론 규칙

우선 코드 사용자는 주어진 코드가 과연 함께 제출된 콤포넌트 기술에 나타난 코드와 일치하는 지를 검사한다. 만약 콤포넌트 기술과 코드가 일치하면 predicate signed_component_dsc가 만족하게 된다.

둘째로, 코드 사용자는 외부 콤포넌트가 코드 사용자가 링크 정책을 통하여 공개한 라이브러리 콤포넌트 이외의 다른 콤포넌트들을 호출하거나 참조하지 않는지의 여부를 검사한다. 만약 외부 콤포넌트가 이 제약을 충족시키면 predicate provides_enough_prps가 만족하게 된다.

마지막으로 코드 사용자는 외부 콤포넌트가 링크 정책에 명시된 모든 속성들을 가지고 있으며, 적합한 속성검증 권위자에게 검증을 받았는지를 함께 제출된 속성 보증서와 공개키 보증서 등과 함께 검증한다. 이것은 predicate exports_required_prps를 만족시키게 된다. 따라서 주어진 3개의 sub-goals가 만족되었으므로 predicate ok_to_link는 만족되고, Secure Linking Theorem에 대한 증명이 얻어진다.

5.3 증명 검증하기

코드 제공자가 코드와 증명을 함께 제출하는 경우, 코드 사용자는 제출된 증명을 검증할 방법이 필요하다. Secure Linking 프레임워크는 Twelf 로직 프레임워크[12]를 이용하여 구현되었다. Twelf는 로직 프레임워크인 LF[8]를 구현한 시스템으로, 사용자가 자신만의 고유한 로직을 디자인하는 것을 가능하게 해주는 시스템이다. 따라서 링크 로직의 모든 항(term)은 바탕이 되는 LF 로직의 항으로 귀착된다. 다시 말해서 코드 제공자가 제출한 증명은 다름 아닌 LF 로직의 항이라고 볼 수 있는 것이다. 항의 타입이 코드 제공자가 증명하려고 하는 목적이고, 항의 몸체가 증명의 유도가 된다. 또한 Curry-Howard isomorphism에 의해서 증명을 나타내는 항의 정확성을 체크하는 것은 그 항의 타입을 체크하는 것과 같은 과정이 된다.

이러한 이유 때문에 Secure Linking 프레임워크에서는 Twelf에서 제공하는 타입 체커를 증명 검증기로 사용하였다. 따라서 만약 제출된 증명이 옳다면, 이 증명(LF 로직 항)은 타입 체커를 통과하게 될 것이다. 증명을 검증한 이후 코드 사용자는 증명과 함께 제출된 콤포넌트를 안전하다고 간주하고 링크를 허락하게 된다.

6. Tactical Prover

Secure Linking 프레임워크는 링크 로직과 함께 프레임워크 사용자가 증명을 생성할 수 있도록 도와주는 로직에 기반을 둔 tactical prover를 포함하고 있다. Prover는 Twelf 로직 프레임워크[12]에 기반을 둔

로직 프로그램으로, 사용자의 편의를 돕기 위해서 포함되었을 뿐 프레임워크의 TCB (trusted computing base)에는 포함되지 않는다.

증명하고자 하는 목표는 정리(theorem)로 포현되고, 정리를 증명하는데 도움이 될 만한 사실들은 가정의 형태로 주어진다. Prover는 주어진 정리에 대한 유도(derivation)를 생성하는데, 이것이 바로 코드 제공자가 코드 사용자에게 보내야할 Secure Linking의 증명이 된다.

Secure Linking 프레임워크가 가지고 있는 prover는 30개의 tactical[1]과 58개의 tactic[2]으로 이루어져 있다. Secure Linking prover의 soundness를 증명하는 것은, prover를 이용하여 증명 가능한 정리는 항상 링크 로직에서 참이라는 것을 보장하게 된다.

Appel과 Felty는 dependently typed 프로그래밍 언어가 정리 증명기(theorem prover)의 국지적인 정확성을 보장할 수 있다는 것을 보였다[3]. 다시 말해서, dependently typed 프로그래밍 언어로 쓰인 prover가 타입 체크되면, 그 prover가 생성한 모든 증명을 유효하다는 것이다. Secure Linking prover가 사용한 Twelf는 dependently typed 프로그래밍 언어이며, [3]에 의해서 prover의 soundness는 쉽게 증명될 수 있다.

7. 활용예제: .NET Framework

7.1 버전관리

.NET 프레임워크는 어셈블리의 버전 정보를 어셈블리 아이덴터티의 일부로 다루고 있다. 이러한 접근 방법은 같은 이름을 같지만 서로 다른 버전을 가진 콤포넌트들이 시스템 안에 공존하는 것을 가능하게 하며, 시스템 관리자나 개발자에게 링크 시에 보다 많은 재량을 준다는 장점을 가지고 있다. 또한 .NET 런타임 시스템은 개발자가 링크 시에 자신이 원하는 콤포넌트의 버전을 지정할 수 있도록 하고 있다.

사용자는 <그림 4>에 나타난 형태로 자신의 링크 환경을 설정할 수 있는데, 같은 시스템 내부에서도 서로 다른 권한의 환경을 설정하는 것을 허락하고 있다. 즉 .NET 내부의 링크 환경 설정은 어플리케이션 레벨, 개발자 레벨, 시스템 관리자 레벨의 3가지 단계로 나누어지고, 각 레벨을 서로 다른 권한을 가지고 있다.

```
<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>
```

그림 4. .NET version configuration

1) tactical은 다중 증명 단계에서 주어진 tactic으로 생성된 증명(혹은 유도)을 합쳐서 더욱 큰 형태의 증명(혹은 유도)을 만들어내기 위한 primitive이다.
2) tactic은 하나의 목적(goal)을 여러 개의 하부목적(sub-goals)으로 나누기 위해서 사용되는 primitive이다.

같은 이름을 가지고 있으나 서로 다른 버전을 가지고 있는 콤포넌트로의 링크를 binding redirection이라고 하며, <그림 4>에 보이듯이 <bindingRedirect>라는 태그 내부에 그 내용을 표시한다. 그리고 환경 설정 주체의 레벨에 따라 binding redirection 명령은 어플리케이션 정책, 개발자 정책, 시스템 정책이라는 이름으로 불린다.

Secure Linking 프레임워크 내부에서 binding redirection은 버전 정보를 콤포넌트의 속성으로 표현하고, 각각의 정책을 속성에 대한 predicate으로 표현함으로써 구현될 수 있다. 각각의 버전 정책 predicate은 버전을 나타내는 속성을 매개변수로 받아들여서, 주어진 버전 정보가 predicate 안에 표현된 redirection 요청에 적합한지를 판명한다. 만약 적합하다면 참을 리턴하고 그렇지 않으면 거짓을 리턴한다.

.NET binding redirection에 대한 결정 과정은 <그림 5>에 나타난 일련의 링크 로직 레마로 표현될 수 있다. .NET에서 링크 시 주어진 콤포넌트의 버전이 매치하는지의 여부는 두 개의 버전이 동일하거나(ver_match_simple), 아니면 binding redirection 정책에 의해서 결정된다(ver_match_policy). 그리고 binding redirection이 일어나는 경우, 서로 다른 레벨의 정책은 우선순위[3]에 따라 고려되어야 한다. 따라서 영향을 미치는 정책의 종류가 시스템 정책인지(machine_redir), 개발자 정책인지(publisher_redir), 아니면 어플리케이션 정책인지(app_redir)에 따라 서로 다른 추론 규칙이 적용된다.

ver_policy_effective(vrq.mch, vrq.org)
version_match_policy(vrq.mch,v)
———————————————— [machine_redir]
vrq(v)

¬ver_policy_effective(vrq.mch, vrq.org)
ver_policy_effective(vrq.pub, vrq.org)
version_match_policy(vrq.pub,v)
———————————————— [publisher_redir]
vrq(v)

¬ver_policy_effective(vrq.mch, vrq.org)
¬ver_policy_effective(vrq.pub, vrq.org)
ver_policy_effective(vrq.app, vrq.org)
version_match_policy(vrq.app,v)
———————————————— [app_redir]
vrq(v)

¬ver_policy_effective(vrq.mch, vrq.org)
¬ver_policy_effective(vrq.pub, vrq.org)
¬ver_policy_effective(vrq.app, vrq.org)
version_match_simple(vrq.org,v)
———————————————— [no_redir]
vrq(v)

그림 5. binding redirection을 위한 추론규칙

7.2 어셈블리와 디지털 서명

.NET 런타임 시스템은 각각의 어셈블리가 strong name을 가질 것을 요구하고 있다[10]. 어셈블리의 strong name은 어셈블리의 이름, 버전 정보, 문화적 배경(언어나 지역 등), 그리고 공개키와 디지털 서명으로 이루어진다. 그리고 각각의 어셈블리에 대한 이 정보는 assembly manifest라는 별도의 파일에 저장된다.

.NET은 어셈블리의 무결성을 위해 코드 제공자가 어셈블리에 디지털 서명을 할 것을 권장하고 있다. 코드 제공자는 2가지 방법을 통하여 어셈블리에 서명을 할 수 있다: 코드 제공자는 어셈블리에 서명을 하고 키분배 권위자를 통해서 받은 보증서를 함께 보내거나, 아니면 서명 후 공개키를

3) binding redirection 정책은 시스템 정책이 가장 우선하며, 다음이 개발자 정책, 마지막이 어플리케이션 정책의 순이다.

assembly manifest 내부에 저장하는 방법이다.

링크 로직을 이용하여 .NET의 strong name을 표현하면서, 링크 로직이 포함하고 있는 가장 기본적인 정리가 만족되지 않음을 발견하게 되었다. 링크 로직은 디지털 서명에 사용되는 모든 공개키들이 키분배 권위자의 보증서와 함께 제시될 것을 명시하고 있지만, .NET에서 어셈블리에 서명한 공개키를 assembly manifest에 저장하는 경우는 이런 조건을 전혀 만족시킬 수 없었다. 언뜻 보기에는 단순한 공개키와 이를 이용한 암호화의 잘못된 적용처럼 보이지만, 이와 같은 오해는 나중에 .NET을 바탕으로 개발된 시스템의 보안상의 허점으로도 악용될 소지를 가지고 있다.[4] 검증되지 않은 공개키를 이용한 디지털 서명을 해쉬 코드 검증보다 나은 시스템 상의 신뢰를 줄 수는 없다. 즉 다시 말해서 이 경우 서명이 검증되었다고 말하는 것은 서명 이후에 서명된 내용물이 변조되지 않았다는 것을 보장해 줄 뿐이며, 이것은 암호화된 해쉬 검증이 줄 수 있는 신뢰도와 크게 다르지 않다. 따라서 링크 로직을 가지고 어셈블리의 strong name을 표현하기 위해서 2개의 서로 다른 추론 규칙이 이용되어야만 했다. 하나는 기존의 공개키 보증서를 가지고 있는 경우를 다루는 규칙이었고, 다른 하나는 공개키 보증서가 없는 경우를 고려한 규칙이었다. <그림 6>에서 보이는 바와 같이 공개키 보증서가 없는 경우, 서명의 검증은 해쉬 코드 검정으로 대치되었다.

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

그림 6. 공개키 보증서가 없는 어셈블리 서명을 위한 추론 규칙

8. 결론

본 논문에서는 안전한 링크 시스템을 위한 프레임워크, Secure Linking을 제안하였다. Secure Linking에서 외부코드 사용자가 원하는 리소스에 대한 권한을 증명하는 것은 외부코드를 사용하는 사람에게 있는 것이 아니라, 외부코드를 제공하는 사람의 책임이 된다.

코드 사용자는 자신의 시스템을 외부로부터의 공격에서 보호하기 위해서 링크 시에 적용될 수 있는 링크 정책을 수립, 외부에 공표한다. 링크 정책은 코드 사용자가 시스템 보호를 위해서 필요하다고 생각되는 여러 가지 속성들을 포함한다. 예를 들면, 콤포넌트의 이름이나 해쉬 코드, 혹은 버전 정보들이 이러한 속성들에 포함된다. 코드 사용자의 시스템에서 어떤 외부 콤포넌트가 링크되고 실행되기 위해서 코드 공급자는 주어진 콤포넌트가 코드 사용자가 요구하는 속성들을 모두 가지고 있다는 증명을 콤포넌트와 함께 보내야한다.

Secure Linking의 링크 로직은 PCA를

4) 만약 사용자가 어셈블리에 사용된 서명을 manifest에 저장된 공개키로 검증하고 나서 어떤 속성에 대한 보증서로 사용한다고 가정해 보라. 이것은 심각한 보안의 허점이 될 수 있다.

기반으로 기본적인 포뮬러 생성자와 추론 규칙들로 이루어져 있다. 코드 사용자의 링크 결정과정과 링크 정책, 그리고 소프트웨어 콤포넌트 명세 등이 링크 로직으로 변환된다. 코드 제공자가 생성하는 증명 역시 링크 로직으로 쓰이고, 코드 사용자가 가지고 있는 믿을 수 있는 증명 검증기에 의해서 검증된다. 만약 콤포넌트와 함께 온 증명이 검증되면, 주어진 콤포넌트는 사용자 시스템에 링크되는 것이 허락된다.

본 논문에서는 링크 로직의 표현성을 실제로 사용되는 시스템인 .NET의 링크 시스템을 링크 로직을 가지고 설명함으로써 증명하였다. 상용화된 시스템에 로직을 이용하여 정형화된 명세(formal specification)를 주는 일련의 과정은 주어진 시스템에 대한 이해를 높이는데 큰 몫을 하였다. 특히 .NET의 경우, 공개키 보증서를 통해서 검증되지 않는 키를 어셈블리에 사용하는 것이 단순한 해쉬 코드 검사로 대체될 수 있는 것임을 밝혀낼 수 있었다. 이것은 단순한 디자인의 차이일 수도 있으나, 명확하지 않은 의미로 사용자에게 혼란을 줌으로써 잠재적인 보안상의 허점이 될 수도 있을 것이다.

또한 디지털 서명 검증이나 키 검증서의 관리를 어셈블리의 링크 시스템과 분리시키는 것이 .NET 시스템 전체의 scalibility와 interoperability를 향상시키는 방법이라고 생각한다.

참 고 문 헌

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

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

[3] A. W. Appel and A. P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, 14(1):3-19, January 2004.

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

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

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

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

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

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

Princeton University

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

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

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

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

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

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

이 은 영

1992년~1996년 고려대학교 전산학과 (학사)
1996년~1998년 고려대학교 전산학과 (석사)
1998년~2003년 Princeton University (박사)
2005년~현재 동덕여자대학교 컴퓨터학과 전임강사
관심분야 : 프로그래밍 언어, Mobile Code Security