情報保護學會論文誌
第16卷 第4號, 2006. 8

실용적인 시스템을 위한 안전한 소프트웨어 컴포넌트 조합

이 은 영
동덕여자대학교

Secure Component Composition for Practical Systems

Eunyoung Lee
Dongduk Women's University

요 약

소프트웨어 컴포넌트를 이용하여 시스템을 구성하는 경우 그리 간단하지 않은데, 그것은 링크 과정 자체가 서로 다른 버전들과 디지털 서명, 정적인 타입 정보나 네트워크로 전송된 소프트웨어, 그리고 서로 다른 판매자에 의한 컴포넌트들을 모두 포함하는 복잡한 과정이기 때문이다. 만약 링크과정에 적용될 수 있는 링크 정책을 수립하고 이를 링크 시에 적용할 수 있는 방법이 있다면 이러한 복잡함을 해결하는 좋은 수단이 된다. 시큐어 링킹(Secure Linking)은 사용자가 안전한 링크를 위한 정책을 만들고 이를 링크 시에 적용할 수 있도록 해주는 새로운 링크 프로토콜이며, 시큐어 링크 프레임워크(Secure Linking Framework)는 시큐어 링크 시스템 구현을 위한 논리적 프레임워크이다. 본 논문에서는 시큐어 링크 프레임워크를 이용하여 마이크로 소프트의 닷넷(.NET)에서 사용되는 어셈블리의 링크 과정을 설명함으로써 시큐어 링킹이 실제로 사용되는 링크 시스템을 나타낼 수 있을 만큼 풍부한 표현력과 실용성을 가지고 있음을 증명한다. 또한 이 과정에서 나타난 어셈블리 코드 서명의 문제점에 대한 논의를 통해서 논리에 기반을 둔 링크 프레임워크가 가지는 장점을 보이고자 한다.

ABSTRACT

When building a software system out of software components, the composition is not simple because of the complexity caused by diverse versions, digital signatures, static type information, and off-the-shelf components from various vendors. Well-established linking policies are one of the best solutions to solve the complexity problem at linking time. Secure Linking (SL) enables users to specify their linking policies which can be enforced at link time. Secure Linking framework is a framework based on a higher-order logic in order to help build a SL system. This paper shows that the Secure Linking logic is expressive enough to describe a real-world component composition system, the linking protocol of .NET. The paper also demonstrates the advantage of the logic-based linking framework by discussing the weakness of the code signing protocol in .NET which was found while we encoded the assembly linking system of .NET.

Keywords : *component composition, logical framework, code security*

Ⅰ. 서 론

안전한 링크를 위해서 현재 가장 널리 쓰이고 있

접수일: 2006년 4월 12일 : 채택일: 2006년 7월 24일
† 주저자, elee@dongduk.ac.kr

는 방법은 타입 체크와 코드 서명이다. 2개의 소프트웨어 컴포넌트를 링크하기 전에 인터페이스의 타입을 체크하는 것은 해당 컴포넌트들이 그들이 사용하는 타입에 있어서는 일치한다는 것은 보장한다. 그렇지만 사용하기 편리하고 그로 인해서 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. 컴포넌트의 사용자는 코드와 함께 주어진 서명이 이 코드가 "잘못된 대화 상자를 절대 띄우지 않는 프로그램임을 보장"하는 것인가, 아니면 "절대로 버퍼 오버런(buffer overrun)이 일어나지 않는 프로그램임을 보장"하는 것인지의 여부를 판단할 수가 없다.

대규모의 소프트웨어 시스템을 구성하는 경우, 한 회사의 개발자가 사용되는 모든 모듈이나 컴포넌트를 개발하는 경우는 그렇게 많지 않으며 효율적이지도 않다. 대부분의 경우, 다른 외부 회사로부터 컴포넌트를 구입하거나, 혹은 회사 내부에서 이미 개발되어 있는 컴포넌트들을 새로 개발한 컴포넌트와 링크, 새로운 시스템을 구성하는 경우가 일반화되어 있다. 이렇게 현대의 소프트웨어 개발 과정에서 링크나 소프트웨어 컴포넌트의 조합이 중요한 위치를 차지하고 있음에도 불구하고, 이 분야의 이론적인 면에 대해서는 상대적으로 연구가 부진한 편이었다. 특히나 링크가 시스템 보안과 관련되는 경우에는 몇몇 연구가 진행되었음에도 불구하고, 극히 초보단계에 머물고 있다[16,17].

본 저자는 소프트웨어 컴포넌트를 이용하여 개발되는 소프트웨어 시스템의 안전성을 보장하기 위한 링크 프로토콜인 시큐어 링킹(Secure Linking)을 제안하고, 시큐어 링킹을 위한 프레임워크인 시큐어 링크 프레임워크(Secure Linking Framework)를 구현하였다.[4] 본 논문에서는 시큐어 링킹과 시큐어 링크 프레임워크, 그리고 프레임워크의 핵심인 링크 논리(Secure Linking Logic)에 대하여 간단하게 소개하고, 그 이후 시큐어 링크 프레임워크를 이용하여 닷넷(.NET)의 링크 시스템을 정형적인 방법으로 정의할 수 있다는 것을 보이고자 한다. 이것은 프레임워크가 포함하고 있는 링크 논리가 실제 현장에서 운영되고 있는 링크 시스템을 표현할 수 있을 만큼 강력하다는 것을 의미한다. 그리고 마지막으로 닷넷의 링크 시스템을 정형적인 방법으로 정의하면서 알게 된 닷넷 시스템의 문제점에 대하여 논의하도록 하겠다. 프레임워크의 구성과 시큐어 링크 논리에 대한 내용은 저자의 다른 논문에서 자세히 논의되었으므로[16] 본 논문에서는 개념을 위주로 소개하도록 하겠다.

II. 관련연구

1. 컴포넌트 모델

컴포넌트 간의 상호작용을 일정한 수준으로 제한하는 일은 컴포넌트 간의 잘못된 사용이나 악의적은 공격을 막기 위해서 반드시 필요하다. 전통적으로 이제까지 프로그램은 추상적 데이터 타입(Abstract Data Type)이나 정보은닉(information hiding)을 통하여 다른 프로그램과의 상호 작용에서 보호되었다.

객체지향 프레임워크인 COM이나 객체지향 언어인 자바의 경우, 컴포넌트는 오브젝트(object)에 바탕을 두고 있다. 이들은 클래스나 메소드, 멤버 데이터 등을 접근의 단위로 삼는다. SML/NJ에서 제공하는 컴파일러 매니저 (Compiler Manager, CM)를 사용하면 소프트웨어 컴포넌트에 대한 접근 단위를 확장시키는 것이 가능하다. SML/NJ의 컴파일러 매니저 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 SML/NJ의 컴파일러 매니저의 계층적 모듈을 자바의 패키지에서 이용할 수 있도록 하는 시스템을 개발하였다.[4] 또한 Reid 등은 C언어를 위한 컴포넌트 모델인 유닛(units)을 제안하였다.[14]

2. 바이트코드 검증

자바는 코드 사용자가 실행될 코드에 대한 타입 체크를 링크 직전에 하도록 하는 첫 번째 시스템이었다. Necula는 증거운반코드(Proof-Carrying Code)에서 체크의 단위를 바이트 코드에서 기계어로 바꾸었다.[11] Necula는 증거운반코드라는 방법을 통하여 기계어 형태의 프로그램에 어떻게 안전성(safety property)을 표현하고 검증하는 방법을 제기하고

있다. 증거운반코드 프로그램의 제공자는 반드시 실행 가능한 기계어 코드와 기계로 검사 가능한 (인간이 검사하는 것이 아닌) 증명을 함께 제공해야만 한다. 프로그램 제공자가 제출하는 증명은 주어진 기계어 코드가 사용자의 컴퓨터가 가지는 안전성 정책(safety policy)을 위반하지 않는다는 것을 보장해야 한다. 그러나 많은 장점에도 불구하고 Necula의 증거운반코드는 모듈에 대한 개념을 포함하고 있지 않다는 한계점을 가지고 있다. 모듈의 개념이 전혀 고려되지 않은 기계어 수준의 코드에 대해서만 검증을 하기 때문에, 모듈이나 컴포넌트가 개발과 업데이트가 주요 단위를 이루는 현재의 소프트웨어 개발 경향을 제대로 반영하지 못한다는 약점을 가지고 있다.

Devanbu는 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.[7]

3. 증거운반인증

시큐어 링크 프레임워크는 Appel과 Felten이 제안한 증거운반인증 논리를 바탕으로 만들어졌다.[2] 증거운반인증(Proof-Carrying Authentication)은 Necula가 제안한 증거운반 코드 방식으로[11] 만들어진 분산 환경에서의 인증을 위한 프레임워크(authentication framework)이다. 증거운반인증은 기존의 분산 인증 프레임워크들과 2가지 면에서 차이점을 보이고 있다. 첫째, 증거운반인증은 고차 논리(higher-order logic)를 사용하여 좀 더 일반적이고 범용적이다. 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.

분산 인증 프레임워크나 프로토콜들은 꾸준하게 정형 논리(formal logic)를 이용해 오고 있으며,[1] 그 대표적인 예가 Taos 분산 운영체제이다. Taos 시스템의 인증 논리는 명제논리(propositional calculus)를 바탕으로 만들어졌으며, 논리의 정당성이 증명되었다.[15] 또한 실제 구현에서는 제안된 인증 논리의 결정 가능한 부분집합(decidable subset)만을 사용함으로써 클라이언트의 요청에 대한 승인 요청에 대한 검증이 항상 결정이 나도록 고안되었다. 결정이 가능한 논리(decidable logic)는 일반적인 논리보다 표현할 수 있는 범위가 좁기 때문에, 이의 사용은 인증 논리 자체의 표현 범위를 좁히는 결과를 가져왔다. 또 기존의 프레임워크들은 응용프로그램에 특화된 논리를 만들기 위해서는 새로운 추론 규칙들을 더하고, 확장된 논리의 정당성을 새로이 증명해야 하는 불편함을 가지고 있었다.

증명운반인증이 사용하는 논리는 술어논리(predicate logic)에 양화사(quantification)를 더함으로써 표현의 다양성을 얻을 수 있도록 설계되었다. 따라서 이 인증 프레임워크는 하나의 세트로 이루어진 추론 법칙들을 가지고 모든 응용 프로그램 특유의 인증 논리를 표현할 수 있다는 장점을 가지고 있다. 증거운반인증에서 응용프로그램의 인증 법칙들은 보조 정리의 형태로 표현되고 증명된다. 이와 같은 증거운반인증 접근 방식의 또 다른 장점은 응용 프로그램들이 같은 추론 법칙들을 공유함으로써, 상호변환이 손쉬워졌다는 점이다. 이것은 증거운반인증 이 기존의 어떤 인증 프레임워크보다 더욱 일반적이라는 것을 보여준다. 그렇지만, 클라이언트의 요청에 대한 해답을 찾는 것이 항상 가능하지만은 않다는 단점을 증거운반인증은 가지고 있다. 그것은 증거운반인증이 사용하고 있는 고차 논리의 한계점이기도 하다. 이 문제를 해결하기 위해서 증거운반인증은 요청에 대한 증명을 클라이언트가 하고, 서버는 단지 그 증명을 검증하는 방식을 택하고 있다. 이것은 Necula의 증거운반코드의 연장선으로 볼 수 있을 것이다. 증거운반인증에 기반을 둔 시스템으로는 Bauer, Schneider, Felten이 개발한 웹 인증 시스템을 들 수 있다[5].

Ⅲ. 시큐어 링킹과 프레임워크

서론에서 지적한 바와 같이 현대의 소프트웨어 개발 과정에서 링크나 소프트웨어 컴포넌트의 조합이 중요한 위치를 차지하고 있음에도 불구하고, 링크나 컴포넌트 조합의 이론적인 면에 대한 연구는 상대적으로 부진한 편이었다. 특히나 링크가 시스템 보안과 관련되는 경우에는 몇몇 연구가 진행되었음에도 불구하고, 극히 초보단계에 머물고 있다[16,17].

시큐어 링킹은 코드 사용자가 링크 시에 자신의 시스템을 외부의 악의적인 코드로부터 보호할 수 있도록 링크 정책(linking policy)을 미리 수립하고, 그것을 이용해서 악의적인 외부 컴포넌트나 모듈로부터 자신의 시스템을 보호하는 것이 가능하게 하는 링

킹 프로토콜(linking protocol)이다. 시큐어 링킹에서 사용자의 정책은 사용자가 시스템의 보호를 위하여 필요하다고 생각하는 특정한 속성을 포함한다. 예를 들면 소프트웨어 컴포넌트의 이름, 어플리케이션에 특화된 정확도(correctness) 속성, 혹은 소프트웨어 컴포넌트들의 버전 정보 등이 이러한 속성에 해당된다고 볼 수 있다. 코드 사용자의 시스템에 주어진 컴포넌트를 링크시키고 실행시키기 위해서, 코드 제공자는 컴포넌트와 함께 주어진 컴포넌트가 사용자가 링크 정책에서 요구하는 속성을 가지고 있다는 증명을 함께 보내야 한다. 이와 같이 시큐어 링킹 프로토콜을 이용하여 컴포넌트 조합이 이루어진 소프트웨어 시스템은 그렇지 않은 링크 시스템을 이용한 소프트웨어 시스템보다 더 안전하다고 말할 수 있다. 다시 말해서 시큐어 링킹 프로토콜에 의해서 허가된 안전한 컴포넌트만을 이용하여 프로그램을 구성함으로써 새로운 프로그램의 조합에 사용된 외부의 컴포넌트가 사용자의 기존 시스템 내부에 존재하는 취약점을 공격하는 것을 피할 수 있게 되며, 이와 같은 관점에서 새롭게 만들어진 프로그램이 더욱 안전해지는 것이다.

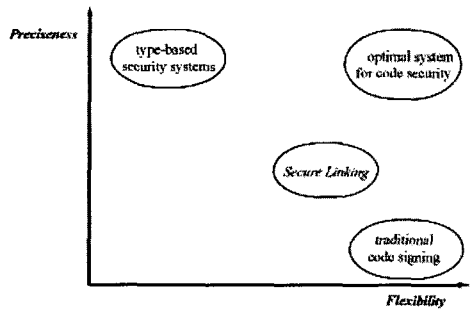

그림 1. 기존 코드보안 방식과의 비교

시큐어 링킹의 장점은 현재 코드 보안은 위하여 가장 많이 사용되고 있는 다른 방식과의 비교를 통해서 논의될 수 있다. 시큐어 링킹은 타입체크와 비교할 때, 다양한 종류의 속성들을 쉽게 표현할 수 있는 유연성(flexibility)을 가지고 있다. 또한 시큐어 링킹은 코드 서명이 보장하는 서명자에 대한 단순한 믿음보다는 구체적으로 명시된 속성에 대한 보장만을 택한다는 점에서 단순한 코드 서명보다 나은 정확한 표현력(preciseness)를 가지고 있다. [그림 1]은 기존의 타입체크 방식과 코드서명 방식을 시큐어 링킹과 비교한 결과를 보여준다. 정확한 표현력과 표현에 있어서의 유연성을 모두 가진 코드 보안 방식을 이상적인 형태라고 가정했을 때, 타입체크 방식은 높은 정확도를 가진다. 다시 말해서, 만약 주어진 프로그램이 타입 체크된다면 그 프로그램은 타입 시스템에서 보장하는 속성을 가지고 있다는 것이 보장된다는 말이다. 그렇지만 주어진 타입 시스템이 처음 의도된 속성 외의 다른 속성을 판단하게 하는 일은 그리 간단하지 않다. 그런 면에서 타입 체크 방식은 매우 낮은 유연성을 가진다고 할 수 있다. 이에 반해서 코드서명 방식은 이미 지적했듯이 서명 자체가 무엇을 의미하는지 정확히 규정되어 있지 않다. 이것은 코드서명 방식이 매우 높은 유연성을 가진 반면, 코드 보안에서 요구되는 또 다른 특성인 정확성을 거의 가지지 못한다는 것을 의미하게 된다. 본 논문에서 제안한 시큐어 링킹은 [그림 1]에서 제시된 것처럼 코드서명 방식이 가지는 정확도에서의 단점을 보완하고, 타입체크 방식이 가지는 유연성에서의 한계를 보완하고자 노력하였다.

시큐어 링크 프레임워크는 시큐어 링킹 프로토콜을 지원하는 논리적 프레임워크로, 고차 논리인 증명 운반인증 논리를 기반으로 하고 있다. 시큐어 링킹에서 요구되는 증명은 프레임워크에서 제공되는 기본 논리와 추론 규칙(inference rules)을 통해서 만들어진다. 코드 제공자가 보낸 증명을 받은 사용자는 작고 믿을 수 있는 검사기(checker)를 이용해서 증명을 검사하고, 만약 증명이 참이라고 검증되면 주어진 컴포넌트를 자신의 시스템에 있는 다른 컴포넌트들과 링크시킨다.

Ⅳ. 시큐어 링킹의 동작원리

이 장에서는 예제를 통하여 시큐어 링킹 프로토콜을 간단히 설명하고자 한다. [그림 2]는 이 장에서 사용될 간단한 예제를 그림으로 보여준다. 만약 앨리스(Alice)라는 주체(principal)가 compiler라는 이름의 컴포넌트를 가지고 있다고 가정해 보자. 앨리스는 컴포넌트를 작성한 프로그래머일 수도 있고, 외부에서 그 컴포넌트를 구입한 단순한 사용자일 수도 있다. 앨리스와 같은 주체를 코드 공급자(code provider)라고 부른다. 앨리스는 자신이 가지고 있는 compiler 컴포넌트를 또 다른 주체인 봅(Bob)에게 보내서 봅의 시스템에서 실행시키기를 원한다. 봅과 같이 다른 사람의 컴포넌트를 받아들여서 실행시키는 주체를 시큐어 링킹에서는 코드 사용자(code

consumer)라 지칭한다. 당연히 봅은 자신의 시스템이 외부의 프로그램으로부터 공격당하는 것을 원치 않으며, 동시에 앨리스의 프로그램을 전적으로 신뢰할 수도 없기 때문에, 봅은 앨리스의 프로그램이 봅에 시스템에 링크되어도 안전하다는 증명을 함께 보내줄 것을 앨리스에게 요구하게 된다. 증명을 위해서 봅은 자신의 링크 정책을 앨리스에게 알려준다. 링크 정책은 크게 3가지 부분으로 나누어진다. 우선 봅이 외부 소프트웨어 컴포넌트에 요구하는 일련의 속성들, 봅이 신뢰하는 외부 권위자들의 이름, 그리고 봅의 시스템 내부에 존재하는, 공개된 소프트웨어 컴포넌트(라이브러리)이다.

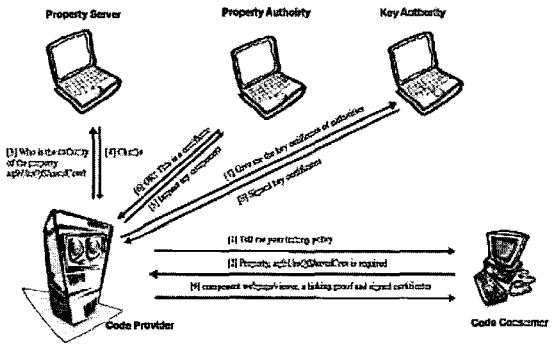

그림 2. 시큐어 링킹의 개념적 구조

1. 속성 (Properties)

코드 사용자는 자신의 시스템을 보호하기 위하여 자신이 미리 선언한 속성들을 외부에서 오는 컴포넌트들이 가지고 있기를 원한다. 시큐어 링킹에서 컴포넌트의 속성은 컴포넌트들이 가지는 행동에 대한 확인의 형태로 표현된다. 시스템을 악의적인 공격으로부터 막는데 유용한 속성들은 여러 가지가 있을 수 있다. 예를 들면 "이 소프트웨어 컴포넌트는 타입 체크되었다"라든지 "이 소프트웨어 컴포넌트는 절대로 할당된 메모리 영역 외의 범위는 액세스하지 않는다" 혹은 "이 소프트웨어 컴포넌트는 사용자의 파일 시스템에서 어떤 정보도 읽거나 쓰지 않는다" 등의 속성들이다.

이 장에서 제시된 예제에서 봅은 외부의 소프트웨어 컴포넌트를 받아들일 때는 항상 컴포넌트가 자신이 믿을만한 컴파일러(Sun Microsystems의 자바 컴파일러나 SML/NJ 컴파일러 등등)에 의해서 타입 체크되었다는 의미를 가진 prp_type_checked를 요구한다고 가정하자. 이제 앨리스는 자신이 가지고 있는 컴포넌트 compiler를 봅의 시스템에 링크시키기 위해서는 자신의 컴포넌트가 prp_type_checked라는 속성을 가지고 있음을 보여야만 한다.

2. 속성검증 권위자

위의 예제에서 사용되는 타입 체크에 대한 속성들처럼 어떤 속성들은 신뢰할만한 컴파일러에 의해서 보장될 수 있지만, 어떤 속성들은 그렇게 쉽게 보장될 수 없다. 그렇지만, 시큐어 링킹에서는 그러한 속성들도 믿을 수 있는 권위자들에 의한 확인이 주어진다면, 주어진 컴포넌트의 속성으로 받아들일 수 있다고 전제한다. 여기에서 사용되는 권위자라는 용어는 소프트웨어 검사(software audit)이나 소프트웨어 엔지니어링에서 사용되는 여러 가지 종류의 소프트웨어 검증 방법들을 지칭할 수도 있다. 혹은 소프트웨어 개발자나 역량 있는 판매자들을 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 컴포넌트가 보증서의 명시된 속성을 가지고 있다고 간주한다.

예제에서 찰스(Charlie)는 소프트웨어 컴포넌트가 prp_type_checked라는 속성을 가지고 있는지를 검정하는 속성검증 권위자이다. 봅이 자신의 링크 정책에서 찰스를 prp_type_checked 속성에 대한 속성검증 권위자로서 신뢰한다고 명시하였기 때문에, 이제 앨리스는 찰스로부터 그녀의 컴포넌트 compiler가 prp_type_checked라는 속성을 가지고 있다는 보증서를 얻어야만 한다.

3. 키분배 권위자(Key Authorities)

속성검증 권위자가 발행하는 보증서는 디지털 서명과 함께 오기 때문에, 이를 검증하기 위해서 코드 사용자는 서명자(여기서는 속성검증 권위자)의 공개키를 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 주체의 이름

과 그의 공개키에 대한 보증서를 발행한다.

주어진 예제에서 봅은 자신이 신뢰하는 하나 이상의 키분배 권위자를 가지고 있어야 하며, 그에 대한 정보를 링크 정책에 명시하여야 한다. 찰스가 발행한 속성에 대한 보증서를 검증하기 위해서, 봅은 찰스의 공개키를 미리 알고 있을 필요가 없다. 단지 자신의 신뢰하는 키분배 권위자에게 찰스의 공개키에 대한 보증서를 요구하고, 이 보증서를 검증하여 찰스의 공개키를 알아낼 수 있다.

4. 속성분배 서버(Property Server)

봅은 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. 시큐어 링킹에서 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 2]에서 봅은 자신이 신뢰하는 속성분배 서버(예를 들어, 에밀리(Emily))를 링크 정책 안에 명시하고, 앨리스는 에밀리에게 자신의 컴포넌트 compiler의 속성을 검사해줄 수 있는 속성검증 권위자가 누구인지 질의하게 된다. 그러면 주어진 예제에서 에밀리는 찰스가 prp_typed_checked를 검증해 줄 수 있는 권위자임을 앨리스에게 알려줄 것이다.

5. 라이브러리

하나의 소프트웨어 컴포넌트가 필요한 모든 코드를 포함하도록 프로그래밍할 수도 있으나 이것은 그리 바람직하거나 자연스러운 일은 아니다. 대신 대부분의 소프트웨어 컴포넌트는 빈번하게 다른 소프트웨어 컴포넌트를 호출하게 된다. 따라서 코드 사용자가 자신의 시스템에서 외부 사용자가 사용할 수 있는 컴포넌트들을 선별적으로 공개할 수 있다. 이렇게 선별적으로 선택되어 공개되는 컴포넌트들을 시큐어 링킹에서는 라이브러리라는 이름으로 부른다. 동시에 코드 사용자는 외부에서 들어오는 모든 컴포넌트들에게 자신들이 호출하고자 하는 컴포넌트들을 모두 명시하도록 요구한다. 그리고 링크를 결정하는 과정에서 코드 사용자는 외부 컴포넌트들이 단지 자신이 공개한 라이브러리만을 호출하는지 검사한다. 만약 외부 컴포넌트들이 라이브러리를 통해서 공개된 컴포넌트 이외의 컴포넌트를 호출한다면, 링크는 거부된다. 코드 사용자가 라이브러리를 선별적으로 공개할 수 있게 함으로써, 시큐어 링킹은 사용자가 자신이 원하는 시스템 컴포넌트나 취약한 컴포넌트를 외부 소프트웨어 컴포넌트가 접근하지 못하도록 막을 수 있다.

6. 링크 결정과정

외부 소프트웨어 컴포넌트를 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. 일단 보증서가 검증되면, 이들은 컴포넌트와 함께 제출된 증명을 검증하기 위한 전제로 사용된다. 주어진 링크정책과 전제로 사용되는 보증서들을 바탕으로 증명이 검증된다면, 코드 제공자가 제출한 소프트웨어 컴포넌트는 신뢰할 수 있다고 간주되고 링크가 허락된다.

V. 시큐어 링크 프레임워크

1. 기술언어

시큐어 링크 프레임워크는 링크 과정을 설명하는데 있어서 2개의 개념적인 모델을 사용하고 있다. 하나는 소프트웨어 컴포넌트와 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 컴포넌트와 링크 정책을 기술하게 된다. [그림 3]은 컴포넌트 기술언어로 작성된 컴포넌트의 명세를 보여주고 있다. 시큐어 링크에서 하나의 컴포넌트에 대한 기술은 컴포넌트의 이름과 이진 모듈(binary modules), 컴포넌트가 공개하는 속성들, 그리고 컴포넌트가 참조하는 다른 컴포넌트에 대한 기술로 이루어진다.

이렇게 기술언어를 써서 작성된 컴포넌트나 링크 정책에 대한 설명은 프레임워크 내부에 구현된 파서

(parser)에 의해서 링크 논리로 변환된다. 시큐어 링크 프레임워크는 사용자가 단순히 클래스나 메소드의 이름을 공개하는 것뿐만이 아니라, 적절한 속성들을 공개하고 도입할 수 있게 한다. 이를 통해서 타입 이상의 정보를 링크 시에 이용할 수 있으며, 소프트웨어 컴포넌트의 조합을 좀 더 안전하게 만들 수 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 3. 컴포넌트 명세서

링크 정책은 시스템 관리자(혹은 컴포넌트를 조합하는 사람)에 의해서 설정되며, 어떤 속성을 가진 컴포넌트들이 자신의 시스템에 링크될 수 있는지를 명시한다. 시큐어 링크 프레임워크가 제공하는 링크 정책 기술 언어는 사용자가 간단하고 편리한 방법으로 링크 정책을 기술하는 것이 가능하게 한다. 링크 정책 내부에서 사용자는 자신이 공개하는 라이브러리 컴포넌트에 대한 정보, 신뢰하는 키 분배 권위자에 대한 정보, 속성분배 서버에 대한 정보, 그리고 외부 컴포넌트에게 요구하는 속성들에 대해서 기술해야 한다. 링크 정책을 프레임워크 내부에 포함시키지 않고, 코드 사용자가 따로 명시할 수 있게 함으로써, 시큐어 링크 프레임워크는 보다 다양한, 응용프로그램 고유의 문제들을 링크 정책으로 표현하는 것이 가능하게 하였다.

2. 링크 논리

시큐어 링크 프레임워크의 핵심인 링크 논리는 증거운반인증 논리에[2] 기반을 둔 고차 논리이다. 따라서 링크 논리에 정의된 오퍼레이터의 내용(semantics)은 증거운반인증 논리에 정의된 오퍼레이터와 추론 규칙을 이용하여 표현된다. 링크 논리의 모든 추론 규칙들은 보조 정리(lemma)의 형태로 표현되고, 모두 증명되었다. 이 장에서는 구현된 링크 논리의 정당성과 링크 결정과정이 어떻게 링크 논리로 표현되었는지를 논의하도록 하겠다. 링크 논리와 컴포넌트/링크 정책 명세 언어에 대해서는 저자의 다른 논문에서 설명되었다.[9]

정당성(Soundness). 링크 논리는 링크의 기본 동작은 정의하는 오퍼레이션과 주어진 링크 정책에 따른 목적(goal)을 증명하는데 사용할 수 있는 추론 규칙들로 이루어져 있다. 시큐어 링킹에서 코드 사용자는 자신의 링크 정책에 대한 증명을 요구하고 링크 논리로 쓰여 진 증명을 검증하게 되는데, 만약 링크 논리의 정당성을 증명할 수 있다면 (다시 말해서, 참인 아닌 정리는 증명될 수 없다면), 이는 링크 정책 및 시스템의 안정성에 커다란 도움이 될 것이다.

일반적으로 어떤 논리의 정당성은 논리 안에 포함된 모든 추론 규칙들을 가지고 만드는 증명들에 대한 귀납법적 접근을 통하여 증명한다. 그렇지만 증거운반인증 논리의 경우는 약간 다른 방식으로 논리의 정당성을 증명하고 있다.[2] 증거운반인증에서 각각의 오퍼레이터는 기반이 되는 고차 논리의 오퍼레이터에 의해서 정의되고, 각각의 추론 규칙들은 바탕이 되는 고차 논리의 보조 정리로서 모두 증명이 되었다. 따라서 기반이 되는 고차 논리의 정당성이 증거운반인증 논리의 정당성을 설명하게 된다. 같은 맥락에서 링크 논리의 정당성 역시 증명될 수 있다. 링크 논리에 포함된 오퍼레이터들은 증거운반인증 논리와 증거운반인증 논리의 바탕이 된 고차 논리에 기반하고 있다. 또한 모든 추론 규칙들은 기반 논리의 정리의 형태로 정의되고 증명되었다. 따라서 이제까지 증명된 증거운반인증 논리와 바탕이 되는 고차 논리의 정당

성은 링크 논리의 정당성을 보장한다.

링크 결정과정. 코드 사용자의 시스템에 어떤 외부 소프트웨어 컴포넌트가 링크되기 위해서는 코드 제공자는 주어진 컴포넌트가 코드 사용자가 요구하는 모든 속성을 가지고 있다는 것을 반드시 보여야 한다. 링크 논리의 관점에서 봤을 때, 이것은 주어진 컴포넌트가 [그림 4]의 결과 부분에 명시된 시큐어 링크 정리(Secure Linking Theorem)를 만족시킨다는 증명을 찾아가는 과정이라고 할 수 있다. 시큐어 링크 정리에 사용되는 술어(predicate) ok_to_link는 앞에서 설명한 대로 외부로부터 주어진 소프트웨어 컴포넌트가 사용자가 명시한 링크 정책을 만족시켰을 때, 만족하게 된다. 술어 ok_to_link에 사용되는 인자는 왼쪽에서부터 각각 외부 컴포넌트를 이루는 이진 모듈의 집합, 링크 논리로 표현된 컴포넌트에 대한 명세, 사용자 컴퓨터에서 사용되는 라이브러리 모듈의 집합, 라이브러리 모듈에 대한 명세, 그리고 사용자가 링크 정책에서 명시한 요구 속성을 나타낸다.

시큐어 링크 정리에 나타난 술어 ok_to_link의 정의는 실제로 어떤 과정을 통해서 코드 사용자가 링크 의사결정을 내리는지를 보여준다. [그림 4]에 주어진 추론 규칙은 링크 결정 과정에 대한 직관적인 이해를 돕는데 도움이 될 것이라고 생각된다.

우선 코드 사용자는 주어진 코드가 과연 함께 제출된 컴포넌트 기술에 나타난 코드와 일치하는 지를 검사한다. 만약 컴포넌트 기술과 코드가 일치하면 술어 signed_component_dsc가 참값을 가지게 된다. 둘째로, 코드 사용자는 외부 컴포넌트가 코드 사용자가 링크 정책을 통하여 공개한 라이브러리 컴포넌트 이외의 다른 컴포넌트들을 호출하거나 참조하지 않는지의 여부를 검사한다. 만약 외부 컴포넌트가 이 제약을 충족시키면 술어 provides_enough_prp가 만족하게 된다. 마지막으로 코드 사용자는 외부 컴포넌트가 링크 정책에 명시된 모든 속성들을 가지고 있으며, 적합한 속성검증 권위자에게 검증을 받았는지를 함께 제출된 속성 보증서와 공개키 보증서 등과 함께 검증한다. 이것은 술어 exports_required_prps를 만족시키게 된다. 따라서 주어진 3개의 부목표들(subgoals)이 만족되었으므로 술어 ok_to_link는 만족되고, 시큐어 링크 정리에 대한 증명이 얻어진다.

3. 증명의 검증

코드 제공자가 코드와 증명을 함께 제출하는 경우, 코드 사용자는 제출된 증명을 검증할 방법이 필요하다. 시큐어 링크 프레임워크는 Twelf 논리 프레임워크를[12] 이용하여 구현되었다. Twelf는 논리 프레임워크인 LF를[8] 구현한 시스템으로, 사용자가 자신만의 고유한 논리를 디자인하는 것을 가능하게 해주는 시스템이다. 따라서 링크 논리의 모든 항 (term)은 바탕이 되는 LF 논리항으로 귀결된다. 다시 말해서 코드 제공자가 제출한 증명은 다름 아닌 LF의 논리항이라고 볼 수 있는 것이다. 항의 타입이 코드 제공자가 증명하려고 하는 목적이고, 항의 몸체가 증명의 유도가 된다. 또한 커리-하워드 동형(Curry-Howard isomorphism)에 의해서 증명을 나타내는 항의 정확성을 체크하는 것은 그 항의 타입을 체크하는 것과 같은 과정이 된다.

$$\frac{\textit{signed\_component\_dsc}(m,dsc,prqset) \quad \textit{provides\_enough\_prps}(dsc,lib,libdsc) \quad \textit{exports\_required\_prps}(prqset,dsc)}{\textit{ok\_to\_link}(m,dsc,lib,libdsc,prqset)}$$

그림 4. 시큐어 링크 정리와 추론 규칙

이러한 이유 때문에 시큐어 링크 프레임워크에서는 Twelf에서 제공하는 타입 검사기를 증명 검증기로 사용하였다. 따라서 만약 제출된 증명이 옳다면, 이 증명(LF 논리항)은 타입 검사기를 통과하게 될 것이다. 증명을 검증한 이후 코드 사용자는 증명과 함께 제출된 컴포넌트를 안전하다고 간주하고 링크를 허락하게 된다.

4. 택티컬 증명기(Tactical Prover)

시큐어 링크 프레임워크는 링크 논리와 함께 프레임워크 사용자가 증명을 생성할 수 있도록 도와주는 택티컬 증명기를 포함하고 있다. 증명기는 Twelf 논리 프레임워크에[12] 기반을 둔 논리 프로그램으로, 사용자의 편의를 돕기 위해서 포함되었을 뿐 프레임워크의 신뢰기반(Trusted Computing Base)에는 포함되지 않는다.

증명하고자 하는 목표는 정리(theorem)로 표현되고, 정리를 증명하는데 도움이 될 만한 사실들은 가정의 형태로 주어진다. 증명기는 주어진 정리에 대한 유도(derivation)를 생성하는데, 이것이 바로 코

드 제공자가 코드 사용자에게 보내야 할 시큐어 링킹의 증명이 된다.

시큐어 링크 프레임워크가 가지고 있는 증명기는 30개의 택티컬과 58개의 택틱으로 이루어져 있다. 시큐어 링크 증명기의 정당성을 증명하는 것은, 증명기를 이용하여 증명 가능한 정리는 항상 링크 논리에서 참이라는 것을 보장하게 된다. Appel과 Felty는 의존적 타입의 프로그래밍 언어 (dependently typed programming language)가 정리 증명기(theorem prover)의 국지적인 정확성을 보장할 수 있다는 것을 보였다.[3] 다시 말해서, 의존적 타입의 프로그래밍 언어로 쓰인 증명기가 타입 체크되면, 그 증명기가 생성한 모든 증명을 유효하다는 것이다. 시큐어 링크 증명기가 사용한 Twelf는 의존적 타입의 프로그래밍 언어이며, Appel과 Felty의 증명에 의해서[3] 증명기의 정당성은 어렵지 않게 증명된다.

Ⅵ. 닷넷 프레임워크의 버전관리

닷넷 프레임워크는 마이크로소프트가 개발한 플랫폼으로 인터넷을 중심으로 한 고도로 분산된 환경에서의 컴퓨팅을 위해서 개발된 환경이다.[13] 닷넷의 런타임 시스템(Common Language Runtime, CLR)은 자바 언어 스타일의 바이트코드 검증기법을 채택하고 있다. 또한 닷넷은 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 컴포넌트들에 대한 정보를 관리하고 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 5. 닷넷 버전 환경파일

닷넷 프레임워크는 어셈블리의 버전 정보를 어셈블리 아이덴터티(identity)의 일부로 다루고 있다. 이러한 접근 방법은 같은 이름을 갖지만 서로 다른 버전을 가진 컴포넌트들이 시스템 안에 공존하는 것을 가능하게 하며, 시스템 관리자나 개발자에게 링크 시에 보다 많은 재량을 준다는 장점을 가지고 있다. 또한 닷넷 런타임 시스템은 개발자가 링크 시에 자신이 원하는 컴포넌트의 버전을 지정할 수 있도록 하고 있다.

사용자는 [그림 5]에 나타난 형태로 자신의 링크 환경을 설정할 수 있는데, 같은 시스템 내부에서도 서로 다른 권한의 환경을 설정하는 것을 허락하고 있다. 즉 닷넷 내부의 링크 환경 설정은 어플리케이션 레벨, 개발자 레벨, 시스템 관리자 레벨의 3가지 단계로 나누어지고, 각 레벨을 서로 다른 권한을 가지고 있다. 같은 이름을 가지고 있으나 서로 다른 버전을 가지고 있는 컴포넌트로의 링크를 바인딩 방향전환(binding redirection)이라고 하며, [그림 5]에 보이듯이 <bindingRedirect>라는 태그 내부에 그 내용을 표시한다.[그림 5]에 주어진 링크 환경설정은 과거에 특정 컴포넌트의 버전 1.0.0.0과 1.9.9.0 사이의 구현을 사용했던 컴포넌트들은 모두 새로운 버전인 2.0.0.0을 사용하도록 규정하고 있다. 그리고 환경 설정 주체의 레벨에 따라 바인딩 방향전환 명령은 어플리케이션 정책, 개발자 정책, 시스템 정책이라는 이름으로 불린다. 각각의 링크 환경설정은 설정 주체에 따라 그 우선순위가 약간 다른데, 주어진 시스템에서 시스템 정책이 가장 높은 우선순위를 가지며, 그 다음이 개발자의 정책, 그리고 어플리케이션 단위로 설정된 정책이 가장 낮은 우선순위를 가진다.

이와 같은 닷넷의 버전 정책은 시큐어 링크 논리를 이용하여 시큐어 링크 프레임워크 내부에서 구현될 수 있다. 링크 프레임워크에서 바인딩 방향전환은 버전 정보를 컴포넌트의 속성으로 표현하고, 각각의 정책을 속성에 대한 술어로 표현함으로써 구현된다. 각각의 버전정책 술어는 버전을 나타내는 속성을 매개변수로 받아들여서, 주어진 버전 정보가 술어 안에 표현된 바인딩 방향전환 요청에 적합한지를 판명한다. 만약 적합하다면 참을 반환하고 그렇지 않으면 거짓을 반환한다.

이와 같은 같은 속성과 술어들을 바탕으로 닷넷의 바인딩 방향전환에 대한 의사결정 과정은 [그림 6]에 나타난 일련의 링크 논리로 쓰인 보조 정리로 표현될 수 있다. 닷넷에서 링크 시 같은 이름을 가진 컴포넌

트가 서로 다른 버전을 가지고 있을 때, 어떤 버전의 컴포넌트를 사용할 지는 2가지 방법에 의해서 결정된다. 만약 적용 가능한 바인딩 방향전환 정책이 존재한다면, 그 정책에 의해서 어떤 컴포넌트를 사용할지 결정한다(ver_match_policy). 그렇지 않으면 기존에 사용되었던 버전과 정확히 일치하는 컴포넌트를 사용한다(ver_match_simple).

술어 ver_match_simple은 단순하게 2개의 같은 이름을 가진 컴포넌트가 같은 버전을 가졌는지 만을 검사한다. 그래서 버전이 일치하는 경우에만 참값을 가지게 된다.

$$\frac{ver-policy-effective(vrq.mch, vrq.org) \quad version-match-policy(vrq.mch, v)}{vrq(v)} \; [machine\ redir]$$

$$\frac{\neg ver-policy-effective(vrq.mch, vrq.org) \quad ver-policy-effective(vrq.pub, vrq.org) \quad version-match-policy(vrq.pub, v)}{vrq(v)} \; [publisher\ redir]$$

$$\frac{\neg ver-policy-effective(vrq.mch, vrq.org) \quad \neg ver-policy-effective(vrq.pub, vrq.org) \quad ver-policy-effective(vrq.app, vrq.org) \quad version-match-policy(vrq.app, v)}{vrq(v)} \; [app\ redir]$$

$$\frac{\neg ver-policy-effective(vrq.mch, vrq.org) \quad \neg ver-policy-effective(vrq.pub, vrq.org) \quad \neg ver-policy-effective(vrq.app, vrq.org) \quad version-match(vrq.org, v)}{vrq(v)} \; [no\ redir]$$

그림 6. 바인딩 방향전환을 위한 추론규칙

그렇지만 바인딩 방향전환이 일어나는 경우, 좀 더 복잡한 의사결정 과정을 따르게 된다. 우선, 서로 다른 레벨의 정책은 우선순위에 따라 고려되어야 한다. 위에서 언급했듯이 바인딩 방향전환 정책은 시스템 정책이 가장 우선하며, 다음이 개발자 정책, 마지막이 어플리케이션 정책의 순이다. 따라서 영향을 미치는 정책의 종류가 시스템 정책인지(machine_ redir), 개발자 정책인지(publisher_redir), 아니면 어플리케이션 정책인지(app_redir)에 따라 서로 다른 추론규칙이 적용된다. 추론 규칙 machine_redir은 시스템 정책이 존재하고, 그 정책이 현재의 컴포넌트에 영향을 미치는 경우에 적용되는 규칙이다. 일단 시큐어 링크 프레임워크는 원래 사용되던 버전이 시스템 정책에 의해서 영향을 받는지 검사한다. 만약 그렇다면 시큐어 링크 논리로 표현된 시스템 정책과 현재 어셈블리의 버전을 인자로 받는 술어 ver_policy_effective는 참값을 가지게 된다. 그 다음에는 링크의 목표가 되는 어셈블리가 시스템 정책에서 정의한 새로운 버전이 맞는지 확인하게 된다. 이것은 술어 version_match_policy로 구현되었으며, 주어진 목표 어셈블리의 버전이 시스템 정책 내부에 정의된 새로운 버전과 맞는 경우 술어는 만족된다.

$$\frac{1: \exists N.asmName(asm, N) \quad 2: \exists V.asmVersion(asm, V) \quad 3: \exists C.asmCulture(asm, C) \quad 4: \exists P.asmPubkey(asm, P) \quad 5: \exists T.asmSign(asm, T) \quad 6: signed(P, T)}{strongNamedAsm(asm)}$$

그림 7. 어셈블리 서명 검증

$$\frac{1: \exists WV.key-authority(WV) \quad 2: \exists WV.says(WV, keybind(WV, K)) \quad 3: SIGN(K, T)}{signed(K, T)}$$

그림 8. 서명 검증을 위한 추론 규칙

$$\frac{1: \exists N.asmName(asm, N) \quad 2: \exists V.asmVersion(asm, V) \quad 3: \exists C.asmCulture(asm, C) \quad 4: \exists P.asmPubkey(asm, P) \quad 5: \exists H.asmHashcode(asm, H) \quad 6: valid-hash-code(asm, H)}{strongNamedAsm(asm)}$$

그림 9. 공개키 보증서가 없는 어셈블리 서명검증

만약 시스템 정책이 없거나, 존재한다고 해도 링크에 사용될 어셈블리에는 특별한 영향을 미치는 않는 경우에는 개발자의 정책이 참조되고, 시큐어 링크 프레임워크에서는 추론규칙 publisher_redir이 적용된다. 추론규칙 publisher_redir의 첫 번째 전제조건은 시스템 정책이 현재의 어셈블리 링크에 영향을 미치고 있지 않다는 것을 나타낸다. 시스템 정책이 영향을 미치지 않는 경우, 프레임워크는 술어 ver_policy_effective를 사용하여 목표 어셈블리가 개발자 정책에 의해서 영향을 받는지 확인한다. 만약 술어 ver_policy_effective가 참이라며, 두 번째 전제조건이 만족하게 되고 목표 어셈블리는 개발자 정책의 영향을 받으므로, 마지막으로 주어진 목표 어셈블리의 버전이 시스템 정책 내부에 정의된 새로운 버전과 맞는지의 여부를 술어 version_ match_policy

를 이용하여 확인하게 된다.

시스템 정책과 개발자 정책이 모두 영향을 미치지 않는 경우에는 어플리케이션 정책이 목표 어셈블리의 링크에 영향을 미치는지의 여부가 확인되고, 영향을 받는 경우 위의 2가지 정책과 마찬가지의 방법으로 처리된다. 이 과정에서 사용되는 추론규칙 app_redir이 사용된다. 참조할 정책이 전혀 없거나, 혹은 정책은 있지만 목표 어셈블리의 링크에 영향을 미치지 않는 경우에는 단순한 버전의 일치만 확인되며, 이 과정을 나타내는 추론규칙이 [그림 6]에 나타나 있는 추론규칙 no_redir이다.

Ⅶ. 닷넷 어셈블리와 코드 서명

닷넷 런타임 시스템은 각각의 어셈블리가 스트롱 네임(strong name)을 가질 것을 요구하고 있다.[10] 어셈블리의 스트롱 네임은 어셈블리의 이름, 버전 정보, 문화적 배경(언어나 지역 등), 그리고 공개키와 디지털 서명으로 이루어진다. 그리고 각각의 어셈블리에 대한 스트롱 네임에 관한 정보는 어셈블리 목록(assembly manifest)라는 별도의 파일에 저장된다. 닷넷은 어셈블리의 무결성을 위해 코드 제공자가 어셈블리에 디지털 서명을 할 것을 권장하는데, 코드 제공자는 2가지 방법을 통하여 어셈블리에 서명을 할 수 있다. 첫 번째 방법은 코드 제공자가 어셈블리에 서명을 하고 키분배 권위자를 통해서 받은 보증서를 어셈블리, 그리고 어셈블리 목록과 함께 보내는 방법이다. 또 한 가지 방법은 개인키로 어셈블리에 서명을 한다는 것은 첫 번째 방법과 동일하나, 키분배 권위자가 발행한 보증서가 없는 경우이다. 이 경우에 코드 제공자는 서명 검증에 사용될 공개키를 어셈블리 목록 내부에 저장하고 어셈블리와 함께 보내게 된다.

외부에서 보내진 어셈블리를 사용하기 전에 모든 사용자는 어셈블리의 코드 서명을 검증하도록 닷넷은 권장하고 있다. 코드 서명은 코드 서명에 사용된 개인키에 대응하는 공개키를 사용하여 검증된다. 따라서 어셈블리 사용자는 어셈블리 목록 내부에 저장된 공개키로 어셈블리의 전자서명을 검증하고, 서명이 검증된 경우에만 어셈블리를 사용하게 된다.

1. 어셈블리 서명의 링크 논리 표현

전자서명 알고리즘을 이용한 어셈블리에 대한 코드 서명과 검증 과정 역시 닷넷의 링크 시스템의 일부이기 때문에, 닷넷의 링크 시스템을 링크 논리를 이용하여 표현한다면, 코드 서명에 관한 부분 역시 링크 프레임워크 내부의 시큐어 링크 논리를 이용하여 표현되어야 한다. 닷넷의 링크 시스템을 다루는데 가장 어려웠던 점은 닷넷 자체가 방대한 규모의 수많은 기능을 가진 시스템임에도 불구하고, 그 동작이나 구현에 대한 제대로 된 설명을 찾아보기 어려웠다는 점이다. 특히나 링크 시스템에 대한 내용은 사용자 매뉴얼 수준의 설명만이 주어졌을 뿐, 어떤 이론적인 내용도 공개되지 않았기 때문에 정형적인 방법을 적용하는데 적지 않은 어려움이 있었다.

닷넷의 링크 시스템을 표현하는 과정에서 우선은 2가지의 어셈블리 서명 방법에 차이를 두지 않고, 닷넷 매뉴얼에 주어진 링크 과정을 충실히 링크 논리를 이용하여 서술하였다. 이 과정은 별다른 어려움 없이 진행되었으며, 그 결과를 나타내는 추론규칙은 [그림 7]과 같다. [그림 7]에서 보이는 바처럼 어떤 어셈블리가 스트롱 네임을 가지고 있다고 말하기 위해서는 어셈블리 목록 안에 어셈블리의 이름, 버전, 문화적 배경, 그리고 제대로 된 공개키로 디지털 서명이 검증 가능하여야 한다. 이것이 추론규칙의 4가지 전제조건이 된다. 술어 signed는 링크 논리 내부에 정의된 술어로 2개의 매개변수를 취한다(추론 규칙의 6번째 줄). 첫 번째 매개변수는 공개키를, 두 번째 매개변수는 디지털 서명을 나타내며, 공개키로 술어 signed는 주어진 공개키로 디지털 서명이 검증 가능한 경우에만 참값을 가진다. [그림 7]의 추론 규칙은 주어진 4가지 전제 조건이 만족한다면, 주어진 어셈블리는 스트롱 네임을 가진다는 것을 의미한다. 다시 말해서 어떤 어셈블리가 스트롱 네임을 가진다면, 술어 strongNamedAsm은 참값을 가지게 된다. 링크 논리로 표현된 닷넷의 링크 시스템은 주어진 어셈블리가 술어 strongNamedAsm을 만족시키는 경우에만 링크를 허용한다. 다시 말해서 술어 strongNamedAsm는 "스트롱 네임을 가진다"라는 요구 속성을 나타내게 되는 것이다. 따라서 모든 어셈블리는 [그림 7]에 주어진 추론 규칙을 이용하여 자신이 술어 strongNamedAsm를 만족시킨다는 것을 증명해야 한다. 증명의 과정은 각각의 전제 조건을 새로운 목적으로 계속적으로 링크 논리의 추론 규칙을 적용함으로써 완성된다.

[그림 8]은 증명 과정에서 사용되는 스트롱 네임의 4번째 전제 조건에 사용된 술어 signed와 관련

된 추론 규칙이다. 링크 논리에서는 어떤 서명이 주어진 공개키에 의해서 검증 가능하다는 것은 주어진 공개키에 대하여, (1) 키 보증서를 검증하기 위한 믿을 수 있는 키분배 권위자가 있어야 하고 (1번째 줄), (2) 믿을 수 있는 키분배 권위자를 시작으로 어셈블리 목록에 저장된 공개키에 대한 신뢰체인(trusty chain)이 형성되어야 하며(2번째 줄), 마지막으로 어셈블리 목록에 저장된 공개키로 코드 서명이 검증 가능하여야 한다(3번째 줄).

앞 절에서 설명한 2가지 어셈블리 서명 방법 중에서 키분배 권위자로부터 보증서를 받는 경우에는 [그림 7]에 주어진 추론 규칙을 이용하여 어셈블리의 스트롱 네임을 충분히 표현할 수 있었다. 그렇지만 마이크로 소프트의 닷넷 매뉴얼에서는 같은 속성을 가진 것으로 표현된 2번째 서명 방법을 사용한 어셈블리의 경우에는 [그림 7]에 주어진 추론규칙으로 설명할 수 없음을 알게 되었다. 다음 절에서는 이 과정에서 발견한 닷넷 어셈블리 링크 시스템의 취약점과 대안에 대해서 보다 자세히 논의하도록 하겠다.

2. 어셈블리 서명의 취약점과 논의

앞 절에서 밝힌 바와 같이 닷넷의 어셈블리 서명 방법 중 하나가 링크 논리로 표현된 추론 규칙을 만족시키지 못하는 이유는 닷넷의 코드 서명 프로토콜이 링크 논리가 포함하고 있는 가장 기본적인 정리를 만족시키지 못하기 때문이었다. 링크 논리는 디지털 서명에 사용되는 모든 공개키들이 키분배 권위자의 보증서와 함께 제시될 것을 명시하고 있으며, 이것은 디지털 서명과 검증에서 가장 기본적인 전제 조건이 된다 ([그림 8] 1번째 줄과 2번째 줄). 그렇지만, 닷넷은 어셈블리에 행해진 코드 서명을 어셈블리 목록에 저장된 공개키로 검증하는 것을 허용하고 있다. 이것은 키 보증서를 통해서 공개키를 검증하는 과정을 생략한 프로토콜이라고 볼 수 있다. 따라서 이 경우는 모든 공개키가 키분배 권위자의 보증을 필요로 한다는 링크 논리의 조건을 전혀 만족시키지 못하고, 결과적으로 링크 논리로 만들어진 추론 규칙을 만족시키지 못하는 결과를 낳은 것이다.

본 연구진은 이와 같은 현상이 링크 논리의 취약점인지 아니면 닷넷 어셈블리의 코드 서명 프로토콜이 가지는 취약점인지 밝히기 위하여 많은 시도와 노력을 기울였다. 많은 논의의 결과, 우리는 현재와 같은 어셈블리의 코드 서명 프로토콜은 어셈블리 프로그래밍의 잠재적 취약점이 될 수 있다는 결론을 내리게 되었다. 언뜻 보기에 이와 같은 현상은 단순히 공개키와 이를 이용한 암호화의 잘못된 적용처럼 보인다. 그렇지만 이와 같은 오해는 나중에 닷넷을 바탕으로 개발된 시스템의 보안상의 허점으로도 악용될 소지를 분명히 가지고 있다.

만약 사용자가 어셈블리에 사용된 코드 서명을 우리가 아는 일반적인 코드 서명과 같은 속성을 가지고 있다고 믿고 다른 응용 프로그램을 개발한다고 가정해보자. 이미 밝혀진 바와 같이 디지털 서명은 여러 가지 장점을 가지고 있다. 인증, 무결성, 부인 금지 등이 그 장점 중에 속하며, 디지털 서명을 사용할 때 응용 프로그램 개발자는 디지털 서명이 가져다주는 이와 같은 모든 장점을 어셈블리의 코드 서명으로부터 기대한다고 볼 수 있다. 따라서 사용자는 어셈블리 목록에 저장된 공개키로 어셈블리의 코드 서명을 검증하고, 검증된 결과를 코드 서명이 부여하는 어떤 속성에 대한 보증으로 응용 프로그램에서 사용할 수도 있다. 예를 들어 응용 프로그램 개발자가 어셈블리 코드 서명을 부인 금지 속성에 대한 보증으로 사용한다면 키 보증서가 함께 주어지지 않은 경우, 이것은 심각한 보안의 허점이 된다. 왜냐하면, 키 보증서 없이 사용자는 어셈블리 목록의 저장된 공개키가 누구의 키인지 알 수 있는 방법이 전혀 없기 때문이다. 키 보증서와 신뢰 체인을 통해서 검증되지 않은 공개키를 이용한 코드 서명은 부인 금지라는 기능을 전혀 수행할 수 없다. 이 경우 서명이 검증되었다고 말하는 것은 서명 이후에 서명된 내용물이 변조되지 않았다는 것을 보장해 줄 뿐이다. 이것은 암호화된 해쉬 검증이 줄 수 있는 신뢰도와 크게 다르지 않다.

이러한 이유 때문에 링크 논리를 이용하여 어셈블리의 스트롱 네임을 표현하기 위해서는 2개의 서로 다른 추론 규칙이 이용되어야만 했다. 하나는 [그림 7]에 주어진 공개키 보증서를 가지고 있는 경우를 다루는 규칙이고, 다른 하나는 주어진 공개키 보증서가 없는 경우를 고려한 규칙으로 [그림 9]에 나와 있다. 위에서 설명한 바와 같이 [그림 9]에서 서명의 검증은 단순한 해쉬 코드 검정으로 대치되었다.

현재 공개된 닷넷 어셈블리의 코드 서명 프로토콜은 그 내용에 있어서 일관성(consistency)이 결여되어 있으며, 이는 추후에 어셈블리를 이용하는 시스템에 있어서 보안상의 허점으로 작용할 가능성이 높다. 예견되는 보안상의 허점을 막기 위해서 서로 다른 내용의 검증을 외견상 하나의 검증 방법으로 묶는

이와 같은 방법의 코드 보안 프로토콜 디자인은 지양되어야 한다고 생각한다. 이를 위한 대안으로는 공개키 보증서가 없는 어셈블리 코드 서명을 사용하지 못하게 하거나, 혹은 현재의 스트롱 네임이라는 이름으로 만들어진 코드 서명 프로토콜을 2개의 별도의 코드 서명 프로토콜로 분리는 방법이 고려될 수 있을 것이다. 또한 코드 서명 검증이나 키 검증서의 관리를 어셈블리의 링크 시스템과 분리시키는 것이 닷넷 시스템 전체의 확장성(scalibility)과 상호호환성(interoperability)을 향상시키는 방법이 될 수 있을 것이다.

Ⅷ. 결 론

프로그래밍 언어의 타입 이론은 많은 연구가 행해진 안정된 분야임에도 불구하고, 링크나 소프트웨어 컴포넌트 조합에 있어서는 상대적으로 연구가 부진한 편이었으며, 특히나 링크가 시스템 보안과 관련되는 경우에는 몇몇 연구가 진행되었음에도 불구하고, 극히 초보단계에 머물고 있다.

시큐어 링킹은 시스템 보안이 중요한 문제로 여겨지는 모바일 코드 환경에서 사용자가 자신의 링크 정책을 명시하고 그것을 링크 시에 적용하는 것이 가능하도록 한 컴포넌트 조합을 위한 링크 프로토콜이다. 또한 시큐어 링킹은 고차논리를 바탕으로 한 논리적 프레임워크를 만들고, 링크 과정을 논리를 이용하여 정형화시킴으로써 프로토콜의 정당성과 안전성을 향상시켰다.

시큐어 링킹에서 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. 시큐어 링킹에서 외부코드 사용자가 원하는 리소스에 대한 권한이 있는지의 여부를 증명하는 것은 외부코드를 사용하는 사람에게 있는 것이 아니라, 외부코드를 제공하는 사람의 책임이 된다. 다시 말해서, 코드 사용자의 시스템에서 어떤 외부 컴포넌트가 링크되고 실행되기 위해서 코드 공급자는 주어진 컴포넌트가 코드 사용자가 요구하는 속성들을 모두 가지고 있다는 증명을 컴포넌트와 함께 보내야 한다.

시큐어 링킹은 고차 논리인 링크 논리에 중심을 두고 있다. 코드 사용자의 링크 결정과정과 링크 정책, 그리고 소프트웨어 컴포넌트 명세 등이 링크 논리로 변환된다. 또한 코드 제공자가 생성하는 증명 역시 링크 논리로 쓰여 진다. 컴포넌트와 함께 제출된 증명은 코드 사용자가 가지고 있는 (혹은 시큐어 링크 프레임워크에서 제공되는) 믿을 수 있는 증명 검증기에 의해서 검증되고, 주어진 컴포넌트는 증명이 검증된 후에만 사용자 시스템에 링크되는 것이 허락된다.

본 논문에서는 마이크로소프트의 닷넷 링크 시스템을 링크 논리를 가지고 표현함으로써 링크 논리의 표현성과 실용성을 증명하였다. 상용화된 시스템에 논리를 이용하여 정형적으로 표현하는 일련의 과정은 주어진 시스템에 대한 이해를 높이는데 큰 몫을 하였으며, 표면적으로 분명히 들어나지 않는 시스템의 결함을 밝히는 데도 훌륭한 역할을 할 수 있음을 알 수 있었다. 본 논문에서는 공개키 보증서를 통해서 검증되지 않는 키를 사용하는 닷넷의 어셈블리 코드 서명이 일반적인 코드 서명이 보증하는 특성들을 보장하지 못한다는 것을 밝혀내었다. 이것은 단순한 디자인의 실수일 수도 있으나, 명확하지 않은 의미로 사용자에게 혼란을 줌으로써 잠재적인 보안상의 허점이 될 수도 있음을 논의하였다. 또한 이런 디자인상의 결점에 대한 몇 가지 대안을 제시하였다.

시큐어 링킹은 링크 시에 사용자가 자신만의 링크 정책을 통해서 보다 안전한 소프트웨어를 조합하는 것이 가능하게 하는 링크 프로토콜로, 소프트웨어 컴포넌트에 미리 정의된 속성을 요구하고 검증함으로써 조합의 안전성을 검증하게 된다. 링크 시에 사용 가능한 유용한 속성들에 대한 심화된 연구가 진행 중이며, 다양한 응용 프로그램 영역에 맞는 적절한 속성의 발견 및 논리적 정의가 시큐어 링킹의 유용성을 더욱 높이는 방법이 될 것이라도 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 International 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 Logical Framework for Policy-Enforced Component Composition (Extended Abstract). ACM SIGSOFT Symposium on the Foundations of Software Engineering, September, 2003.

[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 Taosoperating system. ACM Transactions on Computer Systems, 12(1):3-32, 1994.

[16] L. Cardelli. Program fragments, linking, and modularization. In Proceedings of ACM Symposium on Principles of Programming Languages, pages 266277. ACM Press, January 1997.

[17] 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.

〈著者紹介〉

이 은 영 (Eunyoung Lee) 정회원
1996년 2월: 고려대학교 전산과학과 (학사)
1998년 2월: 고려대학교 전산과학과 (석사)
2004년 1월: Princeton University,U.S.A.(박사)
2005년 3월~현재 동덕여자대학교 컴퓨터학과 전임강사
〔관심분야〕 프로그래밍언어, 코드보안, 컴파일러