개발자 첨부 자료에 의한 트로이 목마 대응 기법
(An Anti-Trojan Horse Mechanism with Attached Data from Developers)

조 은 선 † 예 홍 진 ††† 오 세 창 †† 홍 선 호 †† 홍 만 표 †††
(Eun-Sun Cho) (Hong-Jin Yeh) (Se-Chang Oh) (Sun-Ho Hong) (Manpyo Hong)

요 약 '트로이 목마 프로그램 (Trojan-horse program)' 이란 겉으로는 유용한 일을 하는 것처럼 보이지만 사용자 모르게 악성 행위를 하는 프로그램을 의미한다. 본 논문에서는 개발자의 첨부 자료에 의거하여 트로이 목마 프로그램을 감지할 수 있는 방법을 제안한다. 이 방법에서 코드는 개발자에 의해 자원 접근 정보를 가지는 첨부자료와 함께 배포되며 사용자는 첨부 자료를 통해 코드의 악성 여부를 1차 판단한다. 이 때 정상 코드로 판명되어 수행이 허가된 코드는 수행 중에 감시 시스템에 의해 자신의 첨부자료에 위배되지 않는 다는 것을 2차로 감시 받게 된다. 이로써 알려지지 않는다는 트로이 목마 프로그램의 감지가 가능함과 동시에 기존 감시 시스템 기법들에 비해 사용자 측의 정책 설정 및 판단 부담을 줄여주는 특징을 가진다. 본 논문에서는 제안된 방식을 형식 언어로 기술하고 그 안전성도 함께 보인다.
키워드 : 트로이 목마, 접근 제어 정책, 감시 시스템

Abstract Trojan-horse programs are the programs that disguise normal and useful programs but do malicious thing to the hosts. This paper proposes an anti-Trojan horse mechanism using the information attached to the code by the developers. In this mechanism, each code is accompanied with the information on their possible accesses to resources, and based on this information users determine whether the code is malicious or not. Even in the case a code is accepted by users due to its non-malicious appearance, its runtime behaviors are monitored and halted whenever any attempts to malicious operations are detected. By hiring such runtime monitoring system, this mechanism enables detecting unknown Trojan horses and reduces the decision-making overhead being compared to the previous monitoring-based approaches. We describe the mechanism in a formal way to show the advantages and the limitations of the security this mechanism provides.
Keyword : Trojan Horse, Access Control Policy, Monitoring System

1. 서 론

트로이 목마 프로그램(Trojan-horse program, 이하 트로이 목마) 이란 겉으로는 유용한 일을 하는 것처럼 보이지만 사용자 모르게 패스워드 유출이나 파일 파괴 등의 나쁜 행동을 하는 코드이다. 트로이 목마는 그 형태와 목적이 방대하고 다양하기 때문에, 극단적으로는 바이러스(virus) 나 웜(worm) 등의 악성 코드들이 거의 모두 트로이 목마에 속한다고 정의할 수도 있다.

트로이 목마는 인터넷상에서 내려 받거나 이-메일을 통해 전해지기도 하고, 일반적인 프로그램들 중에서도 트로이 목마가 발견되기도 하는데, 외부의 침입등에 의해 기존의 프로그램이 변조되는 경우도 여기에 포함된다. 대부분의 사용자는 개발자가 붙인 프로그램의 이름이나 프로그램에 관한 설명 등을 통해서 해당 프로그램의 수행 결과를 직관적으로 판단하고 수행하므로, 악의를 가진 개발자에게 쉽게 속을 수 있다.

본 논문은 주어진 프로그램이 트로이 목마인지를 판별하여 수행을 거부하거나 수행 중 제지를 하는 시스템인 'SKETHIC(Secure Kernel Extension against

† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
eschough@cbucc.chungbuk.ac.kr
†† 비 회 원 : 아주대학교 정보통신전문대학원 교수
sechang@ajou.ac.kr
sunhong@ajou.ac.kr
††† 종신회원 : 아주대학교 정보통신전문대학원 교수
hjyeh@ajou.ac.kr
mphong@ajou.ac.kr
논문접수 : 2001년 1월 27일
심사완료 : 2002년 1월 22일

Trojan Horses with Information-carrying Codes)' 을 제안한다. SKETHIC은 기존의 트로이 목마 대응 방법과는 달리 사용자의 프로그램에 대한 인식과 프로그램의 실제 행동간의 차이를 파악하여, 이미 알려진 트로이 목마 뿐 아니라 알려지지 않은 트로이 목마들을 판별하고 대응하는 것에 중점을 둔다. 프로그램은 코드 부분 외에도 수행 중 접근할 자원의 목록과 함께 배포되며, 사용자는 이 자료를 근거로 트로이 목마인지를 1차 판단한다. 이러한 사용자의 허가를 받아 수행 중인 프로그램이 만일 첨부 자료에 명시 된 것 외의 자원 접근을 시도한다면 감시 시스템이 2차로 판별하여 수행을 중지시키게 된다. 이 방식은 기존의 시그니쳐 기반 정적 판별 기법에 비해 미리 정보가 알려지지 않은 트로이 목마에 대해서도 대응할 수 있다는 장점이 있고, 또 기존의 감시 시스템 방식과 비교할 때에는 사용자가 낯선 프로그램들을 위한 접근 제어 정책을 일일이 세워야 하는 부담을 덜 수 있다는 장점을 가진다.

본 논문의 구성은 다음과 같다. 2 장에서는 기존의 트로이 목마 감지 기법들에 관해 살펴보고, 3 장에서는 SKETHIC에 대한 개괄적인 소개를 한다. 4 장에서는 SKETHIC의 트로이 목마 대응 방식에 대해 형식적으로 정의하고 시스템을 안전하게 유지함을 보인다. 5 장에서는 토의 및 비교를 한 후, 6장에서는 현재 진행 중인 구현 내용을 간략히 소개 하며, 7 장에서 결론을 맺는다. 끝으로 부록에는 4 장에 소개된 정리들이 증명되어 있다.

2. 기존의 트로이 목마 대응 방안

현재 알려져 있는 트로이 목마 대응 방안들은 다음과 같이 세가지 종류로 분류해 볼 수 있다.

첫째, 해당 트로이 목마에 관한 정보가 미리 파악된 경우, 알려진 트로이 목마 프로그램들의 특성들이 주어진 프로그램 내에서도 발견된다면 트로이 목마로 간주하는 방법이다. 예를 들어 바이러스 백신에서처럼 알려진 트로이 목마들의 시그니쳐(signature)중 하나가 주어진 프로그램 속에 들어있는지를 보고 트로이 목마임을 판별하거나[1,2], 해당 프로그램을 정적으로 분석하여 기존의 악성 코드들이 가지는 특성을 지니는지 여부를 파악하기도 한다[3]. 또, 프로그램을 우선 수행 시키다가 기존 트로이 목마들 중 하나와 유사한 행동을 보이면 수행을 중단시키는 방법도 있다[4]. 이러한 방식들은 기존의 트로이 목마들에 대한 자료에 의거하기 때문에, 이미 알려진 트로이 목마로 그 대응 범위가 한정되며, 악성 코드의 종류가 다양하게 증가하는 최근 상황에는 적합한 해결책이라고 볼 수 없다.

둘째, 악성 코드 방지를 위해 의심이 가는 프로그램에 대해 접근 가능한 자원을 한정시켜 수행시키는 방식이 있다[5,6,7,8,9]. 본 논문에서 소개하고 있는 SKETHIC 도 이 방식을 따르는데, 프로그램의 수행을 감시 및 제어하는 감시(monitoring) 시스템이 있어서 주어진 접근 제어 정책(access control policy) 에 따라 해당 프로그램의 자원 접근을 제한한다. 이 때 접근 제어 정책은 해당 프로그램의 성격에 맞게 최소한으로 한정하여 정의하는 것이 이상적인데, 특히 트로이 목마는 그 대상 프로그램의 종류가 매우 다양하고 이질적이기 때문에, 자바 애플릿의 보안 모델[10]처럼 출신 주소나 개발자 별로 획일적으로 자원의 접근 권한을 한정하는 것은 적절하지 못하다[7]. 따라서 주로 시스템 호출 정도의 작은 단위의 제어 모델을 기반으로 하는 경우가 많으며[8,9], 상태 전이 기계(state transition machine) 와 같이 프로그램의 추상적인 의미를 접근 제어 정책에 반영하려는 시도도 있어왔다[5,6]. 이러한 접근 제어 정책은 해당 프로그램을 수행할 사용자가 설정하게 되는데, 이에 따라 처음 대하는 프로그램들의 자원 접근 내역을 자세히 파악해야 하는 사용자의 부담이 감시 시스템 방식의 주요 단점 가운데 하나이다[7]. 그러나, 기존에 알려지지 않은 악성 코드들에 대해서도 대응할 수 있다는 커다란 장점이 있다.

그 밖에도 해당 프로그램이 트로이 목마로 변조되었는지 여부를 알아내기 위해 사용할 수 있는 방법으로 무결성 값을 비교하는 방법이 많이 사용되고 있다[11,12]. 정상적인 원본 프로그램이 명확히 존재하고 이에 관한 무결성 값(예를 들어 메시지 다이제스트(message digest)) 이 이미 알려져 있는 경우에, 이를 주어진 프로그램에서 계산한 무결성 값과 비교하여 판별하게 된다.

3. 제안된 트로이 목마 대응 방안

앞 장에서 언급 되었듯이 SKETHIC은 기존에 알려지지 않은 트로이 목마를 감지 할 수 있도록 하기 위해 악성 코드 감시 시스템을 가지는 방식을 채택하였다. 그러나, 기존의 방식들과 다른 점은 사용자가 가지는 접근 제어 정책 설정의 부담을 개발자로 이전시킴으로써 앞 장에서 지적된 감시 시스템 기반 방식의 문제점을 완화시킨다는 것이다. 따라서, SKETHIC에서는 개발자의 자원 접근에 관한 자료 첨부와 이 자료에 의거한 사용자의 판단, 그리고, 감시 시스템의 제어가 상호 보완적

인 형태를 갖는다. 이들 각각이 하는 일을 정리하면 다음과 같다.

• 프로그램 개발자 : 자신이 작성한 프로그램에서 필요로 하는 자원과 접근 형태들의 목록인 자원 접근 목록(resource access list) 을 프로그램과 함께 배포한다. 즉 프로그램이 코드와 자원 접근 목록의 결합 형태로 확장된다고 볼 수 있다. 따라서, 본 논문에서 프로그램은 $p=<m, c, l>$로 나타내며, 여기서 $m \in M$은 프로그램 이름 등의 식별자, $c \in C$는 프로그램 코드, $l \in L$은 필요로 하는 자원 접근 목록이다.

• 사용자 및 시스템 관리자 : 주어진 프로그램 $<m, c, l>$의 이름 m 및 첨부된 자원 접근 목록 l을 보고, 주관적인 기대에 비추어 악성 연산을 내포하였는지를 1차 판별한다. 즉 프로그램의 이름이나 기대되는 행동에 어긋나며 위험 소지가 있는 연산이 자원 접근 목록에서 유도될 수 있는지 검토한다. 이로써 트로이 목마라는 의심이 들지 않는 프로그램에 한해서 자신의 컴퓨터에 저장하거나 경우에 따라서는 코드 c를 곧바로 수행한다.

• 감시 시스템 : 코드 c의 수행 중에 개발자가 첨부한 자원 접근 목록 l에 기술된 것 외의 자원 접근을 시도하는지 계속 감시한다. 만일 그러한 자원 접근이 일어났다면 그 프로그램의 수행을 중지시키고 사용자에게 알린다. 이 부분의 작동 원리는 기존의 보안 운영 체제나 악성 코드 감시 시스템 방식과 동일하나, 그 기반이 되는 보안 정책을 개발자가 보내온 자원 접근 목록 l을 가지고 결정한다는 것이 다른 점이다.

전체적인 흐름은 그림 1에서 보이는 바와 같다. 먼저 프로그램 개발자는 자신이 개발한 코드에 자원 접근 목록을 첨부하여 배포한다. 이를 받은 사용자나 시스템 관리자는 그 목록을 보고 해당 프로그램이 자신의 컴퓨터에 유해한지를 판단한다. 만일 사용할 것으로 결정이 되면 코드와 첨부된 목록은 저장이 되며, 환경에 따라서는 즉시 수행되기도 한다. 수행 중인 코드는 시스템 호출 등을 통하여 컴퓨터의 자원을 접근하게 되는데, 이 때 운영체제 커널과 연계된 감시 시스템에서는 개발자에 의해 첨부된 목록에 명시된 것 이외의 자원 접근을 하는지 감시한다. 만일 그러한 일이 발생한다면 수행을 중단한다.

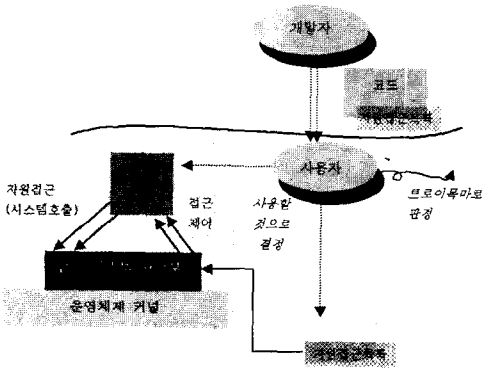

그림 1 SKETHIC의 트로이 목마 대응 방식

예를 들어 navidad.exe[13]와 같은 트로이 목마가 전달된 경우 자원 접근 목록을 사실대로 첨부한 경우와 거짓으로 첨부한 경우 두 가지로 나누어 생각해 볼 수 있다.

• 개발자가 자원 접근 목록을 해당 프로그램의 행동을 그대로 반영하여 사실대로 첨부하게 되면, 사용자는 이 프로그램이 자신의 파일을 접근하여 파괴하리라는 것을 알게 되므로 수행시키지 않을 것이다.

• 첨부된 자원 접근 목록이 화면에 출력하거나 임시 파일을 만드는 것 정도로 가장되어 거짓으로 첨부된다면 사용자나 관리자는 속아서 수행을 시킬 수 있다. 하지만, 수행 도중에는 자신의 자원 접근 목록에 명시된 것 이외의 자원 접근을 시도하게 되므로 감시 시스템의 접근 제어 기능에 의해 그 수행을 제지 당할 것이다.

제안된 방식이 기존의 다른 트로이 목마 대응 방법들에 비해 가지는 장점을 정리하면 다음과 같다. 첫째, 악성 코드 감시 시스템 방식을 채택하여 알려지지 않은 트로이 목마에 대한 대응도 가능하다는 점이다. 둘째, 앞서 지적되었던 감시 시스템 방식의 문제점인 사용자의 접근 권한 설정 부담을 해소 한다. 다시 말하자면, 사용자가 일일이 각 프로그램의 접근 권한을 설정할 필요 없이, 코드에 첨가된 자원 접근 목록을 보고 판단을 하는 것으로써 대신하게 된다. 셋째, 이미 사용자에 의해 1차 선별된 프로그램에 대해서만 감시 및 제어가 수행되므로 기존의 수행 중 제어 방식들에 비해 감시 시스템의 부담도 줄어들게 될 것으로 기대된다.

4. 형식적 기술

트로이 목마는 그 특성이나 행동이 매우 광범위하므로 트로이 목마에 대응한다는 것 또한 그 문제를 명확히 정의하는 것을 필요로 한다. 또, 제안하는 대응 방식의 작동 원리를 이에 맞추어 기술하고 그 효과를 보이는 것은 대응 방식의 장점과 한계를 파악하는 데에 유익한 작업이다[15]. 본 절에서는 트로이 목마가 파괴하는 안전성에 관한 정의와 이를 위한 SKETHIC의 대응 방안을 정

제된 형식으로 기술하고, 결과적으로는 SKETHIC이 주어진 문제의 해결 능력이 있음을 보인다.

1.1 문제의 정의

연산의 집합 O, 시스템의 자원 상태의 집합을 R, 사용자의 집합을 U라고 하자. 프로그램을 실제 수행시킬 때 실행될 수 있는 연산들의 순서열(sequence)을 '연산 수행 순서열'이라고 정의한다. 한 연산 수행 순서 열은 어떤 프로그램을 한 번 수행시켰을 때 일어날 수 있는 연산들을 순서대로 늘어놓은 것과 같다. 따라서, 하나의 프로그램은 여러 연산 수행 순서 열을 가지게 된다. 무한히 수행되는 프로그램들이 있으므로 순서열의 길이는 무한할 수 있다고 가정한다.

정의 1 연산 수행 순서열(operation execution sequence) $q=<o_1, \cdots, o_n> \in Q$, $o_i \in O$, $0<n\leq\infty$은 연산의 리스트(list)로 정의한다.

코드는 프로그램 자체를 나타내는 것으로 수행 시 나타날 수 있는 가능한 연산 수행 순서 열들의 집합으로 정의한다. 한 프로그램에서 수행 가능한 경로들의 집합이므로 이것의 원소 개수도 무한할 수 있다고 가정한다.

정의 2 코드는 $c=\{q_1, q_2, \cdots q_n\} \in C$, $q_i \in Q$, $0<n\leq\infty$로 정의한다.

또, 코드 c의 수행 중 나타날 수 있는 모든 연산 o들의 집합 $\{o \mid \exists q \in c. o \in q\}$을 함수 *operations* : $C \Rightarrow 2^O$를 써서 *operations*(c)로 나타낸다.

시스템은 프로그램들의 집합과 자원의 상태로 정의된다. 직관적으로 자원의 종류로는 파일시스템, 네트워크 포트 등을 들 수 있다.

정의 3 시스템은 $s \in S=<\{c_1, \cdots, c_n\}, r>$, $c_1, \cdots c_n \in C$, $r \in R$로 정의한다.

프로그램이 수행하는 연산에는 시스템 전체에 영향을 미치는 연산과 그렇지 않은 연산이 있다. 이를 판단하는 기준은 우선 자원의 상태와 정보 유출로 정의한다.

정의 4 영향 연산(effective operation) $e \in E$ 이란 다음과 같은 연산 중 하나이다.

- 시스템 자원의 상태를 바꾼다
- 시스템 내의 정보를 외부로 보낸다

시스템 보안에 영향을 미칠 수 있는 것은 영향 연산과 보안상의 헛점을 안고 있는 연산이다. 따라서 이러한 연산들을 '문제 연산'이라 이름 붙여 사용 한다.

정의 5 문제 연산(problematic operation) 이란

- 영향 연산(effective operation) 이거나
- 보안 취약부(security hole) 이다.

보안 취약부는 아직 정확한 기계적인 기준을 정하기가 어렵다. 주어진 시점에서는 그 취약성이 드러나지 않는 코드들도 앞으로 침입에 사용될 소지가 있기 때문에 코드 작성자 조차 보안 취약부인지 알 수 없다. 그러나, 여기서는 우선 일반적인 정의를 가정한다.

사용자가 특정 연산을 기대하는지 여부는 전적으로 사용자에게 달려있다. 다음의 세 함수는 코드 c 뿐 아니라 사용자 u 도 그 인자로 취하여 이를 반영한다.

정의 6 *overt* : U x $C \Rightarrow 2^O$는 사용자 u와 코드 c가 주어지면, *operations*(c) 중 u가 기대하고 있는 연산들의 집합을 결과로 하는 함수이다.

정의 7 *hcovert* : U x $C \Rightarrow 2^O$는 사용자 u와 코드 c가 주어지면, *operations*(c) 중 u가 기대하지 않고, 문제 연산에 속하는 연산들의 집합을 결과로 하는 함수이다(*hcovert*는 harmful covert operation을 의미) .

정의 8 *hlcovert* : U x $C \Rightarrow 2^O$는 사용자 u와 코드 c가 주어지면, *operations*(c) 중 u가 기대하고 있지 않지만, 문제 연산도 아닌 연산들의 집합을 결과로 하는 함수이다(*hlcovert*는 harmless covert operation을 의미).

사용자가 기대하지 않는 연산은 *hcovert*(u,c)와 *hlcovert*(u,c)의 합집합으로서 '물밑 연산(covert operation)'이라고 부른다. 물밑 연산은 문제 연산인지 여부에 따라 *hcovert*(정의 7)와 *hlcovert*(정의 8)의 결과로 나누어져 그 역할이 달라지는데, 여기서는 이 중 트로이 목마에 대한 대응책을 위해 초점이 맞추어지는 것은 *hcovert*(u,c) 이다.

여기서는 문제의 범위를 한정하기 위해, 사용자가 이미 파악하고 있는 연산에 대해서는 사용자 자신에게 전적인 책임을 준다. 즉, 사용자가 연산의 결과에 대해 이미 알고 있고 그것의 수행을 원한다면, 문제 발생 소지가 있다고 해도 사용자의 의도를 따른다. 따라서, 문제 연산 여부에 따른 별도의 세부 분류가 필요 없이 *overt* 함수의 결과에 속하는 연산은 모두 수행 거부를 받지 않게 된다. 이러한 연산들을 '기대 연산(overt operation)'이라고 부른다.

SKEPTHIC은 주어진 프로그램 c의 *hcovert*(u,c)를 알아내어 트로이 목마 여부를 판단한다. 그 과정을 정의하면 다음과 같다.

정의 9 사용자 u에 대해 트로이 목마란 *hcovert*(u,c) $\neq \varnothing$인 코드 c를 의미한다.

일반적으로 트로이 목마에 대하여 안전한 상태의 시스템을 트로이 목마가 전혀 없는 시스템으로 정의한다. 이를 위해서는 수행 전에 이미 트로이 목마인지를 판별하여 시스템에 설치되지 못하게 하는 것이 기존의 알려진 트로이목마 대응 기법들이 채택하고 있는 방법이다.

정의 10 시스템 s가 트로이 목마에 관해 안전한 상태

에 있다는 것은 모든 사용자 u에 대해, $fst(s)$에 u에 대한 트로이 목마 t가 존재하지 않는 상태를 의미한다.

본 연구에서 다루고 있는 대응책은 위의 정의 9에서 기술된 트로이 목마를 감지하고 그 수행을 막아 정의 10과 같은 안전한 상태를 유지하는 것이 목적이다. 다음 정리는 초기 시스템이 트로이 목마에 대해 안전하고, 트로이 목마가 확실히 아닌 프로그램만 추가된다면, 그 시스템은 트로이 목마에 관해 안전하게 유지될 수 있음을 나타낸다.

정리 1 트로이 목마에 관하여 안전한 상태의 시스템 $s=<cs, r>$와 트로이 목마가 아닌 코드 $c \notin cs$에 대해, $<cs \cup \{c\}, r>$은 트로이 목마에 대해 안전한 시스템이다.

증명 정의 9와 정의 10에서 자명하므로 생략한다. □

4.2 SKETHIC의 정의

앞 장에서 소개되었듯이 SKETHIC에서 프로그램의 정의는 코드 뿐 아니라 자원 접근 목록도 함께 필요로 한다.

정의 11 프로그램은 $p \in P = M \times C \times L$로 정의한다. (단 M은 프로그램의 이름이나 식별자, C는 코드, L은 자원 접근 목록)

이에 따라 트로이 목마의 정의도 다음과 같이 확장된다.

정의 12 프로그램 p가 사용자 u에 대해 $hcovert(u, snd(p)) \neq \emptyset$을 만족한다면 p를 트로이 목마라고 한다.

자원 접근 목록은 주어진 코드가 수행 중 행하는 자원 접근 연산들을 나타내고 있는 것으로 볼 수 있다. 주어진 자원 접근 목록에 의해 직/간접으로 기술되는 연산들은 그 목록에서 유도 되었다고 정의한다.

정의 13 $l \in L$, $o \in O$일 때 $l \Rightarrow o$는 주어진 자원 접근 목록 l이 연산 o를 명시하고 있는 것을 뜻하고, l이 o를 유도(imply) 한다고 한다.

또, $l \in L$, $os \in 2^O$ 일 때 $l \Rightarrow os$는 $o \Rightarrow os$ 인 모든 o에 대해서 $l \Rightarrow o$임을 의미한다.

자원 접근과 관련되지 않은 연산들은 모든 $l \in L$에서 유도된다고 가정한다.

코드 c의 올바른 자원 접근 목록은 c에서 수행될 모든 연산(즉, $operations(c)$)을 유도해야 한다. 또, 해당 코드가 수행하지 않는 관련 없는 연산들이 잘못된 자원 접근 목록에서 유도되는 것도 방지해야 한다. 특히 실제 코드에 없는 $hcovert(u,c)$가 잘못 유도된다면 정상 프로그램이 트로이 목마로 간주될 소지마저 있다. 따라서 본 논문에서 필요한 올바른 자원 접근 목록을 다음과 같이 정의한다.

정의 14 코드 $c \in C$의 올바른 자원 접근 목록 $l \in L$이란 다음 조건을 만족하는 것이다.

- $l \Rightarrow operations(c)$ 이고,
- $\forall o.(l \Rightarrow o \wedge o \in hcovert(u, c)) \geq o \in operations(c)$

본 논문에서 제안하는 시스템에서는, 나쁜 의도가 없는 개발자는 늘 올바른 자원 접근 목록을 첨부한다는 가정을 기반으로 한다. 즉, 정상 프로그램의 개발자들이 첨부한 자료는 항상 정확하고 유용한 정보를 내포한다는 의미이며, 이는 잘 고안된 정적 코드 분석기 등의 개발자 도구의 지원에 의해 실현될 것으로 본다. 반면, 트로이 목마의 자원 접근 목록에게는 이 성질의 보장을 기대할 수 없다고 가정한다.

또, 나쁜 의도가 없는 프로그램에는 항상 올바른 자원 접근 목록이 첨부된다고 가정한다. 이는 개발자가 개발 단계에서 소스 코드에 대한 정확한 정보를 가지고 있으므로 자동화 분석 도구[3,14]의 도움을 받아 이루어 것으로 본다. 특히, 소프트웨어 공학에서 연구되는 취약성 검증 도구[14]나 타입 검사와 같은 프로그램 관련 검증 도구들에서도 변용 방법을 찾을 수 있을 것으로 기대된다.

본 논문에서 제안하는 SKETHIC은 상태와 이 상태에 적용될 수 있는 연산들로 기술할 수 있다. SKETHIC의 상태 T 는 $<u, d, k, s, f_{mc}, f_{ml}, x_1; \cdots ;x_n> \in U \times D \times K \times S \times F_{mc} \times F_{ml} \times List(X)$로 정의한다. 여기서 u는 사용자, d는 개발자, k는 감시 시스템을 의미하며, s는 시스템을 나타낸다. $f_{mc} \in F_{mc} : N \geq C$와 $f_{ml} \in F_{ml} : M \geq L$는 각각 프로그램 이름을 입력으로 하고 관련 코드나 첨부된 자원 접근 목록을 출력으로 하는 함수들이다. $List(X)$는 SKETHIC 연산 $x_i \in X$를 원소로 하는 순서화 집합으로, 리스트의 원소인 각 연산들은 ';'로 분리되어 있다.

SKETHIC의 연산 X는 SKETHIC 상태를 변화시키거나 코드에서의 연산 O를 수행시키는 것을 목적으로 하는 행위의 단위이다. 일반적으로 코드의 연산 O보다 수행 단위가 크며, $X=\{INSERT, DELETE, EXECUTE, run\}$로 정의된다. $INSERT(u, p)$는 사용자 u가 새로운 프로그램 p를 추가시키는 것을, $DELETE(u, m)$는 m으로 식별되는 프로그램을 삭제하는 것을 각각 의미한다. $EXECUTE(u, m)$는 m의 코드를 수행함을 의미하고, $run(k, o)$는 커널 k에서 프로그램의 연산 o를 수행함을 뜻한다. 현재는 자신이 추가시킨 프로그램의 코드만을 수행한다고 가정되었으나 추후 쉽게 확장될 수 있다. 연산의 의미는 상태를 어떤 식으로 변화시키는 지를 설명하는 규칙에 의해 정의된다. 따라서, run을 제외한 각 SKETHIC 연산의 정의는 다음과 같이 표현된다.

$<$*이전상태*$>\triangleright<$*이후상태*$>$ if $<$*조건*$>$

이것은 주어진 $<$*조건*$>$에서 $<$*이전 상태*$>$가 $<$*이후 상태*$>$로 바뀐다는 것을 나타낸다. 결국 $<$*이전 상태*$>$의 마지막 항(연산 열)에서 첫번째 연산의 의미(semantics)를 기술한 것이라고도 볼 수 있다.

새로운 프로그램이 삽입 될 때 자원 접근 목록으로부터 $hcovert(u,p)$에 속하는 연산이 유도되면(즉, $i \Rightarrow hcovert(u, c)$), 앞서 정의 9와 정의 12에 따라 그 프로그램은 트로이 목마로 간주, 시스템에 포함시키지 않는다(아래 [삽입 III]). 그렇지 않은 경우에는 기존에 해당 프로그램이 존재하는 경우(아래 [삽입 II])와 존재하지 않는 경우(아래 [삽입 I])로 구별해서 s, f_{mc}, f_{ml}에 넣는다. 기호 $f[x/y]$는 주어진 함수 f 를 $f(y)$ 값을 x로 하여 대입 또는 확장함을 의미한다. Ops는 x_1; … x_n과 같이 각 원소가 ; 로 구분된 SKETHIC 연산들의 리스트이다.

[삽입 I] $<u, d, k, s, f_{mc}, f_{ml}, INSERT(u, <m, c, l>)$; $Ops>$

$\triangleright$ $<u, d, k, <fst(s)\cup\{c\}, snd(s)>, f_{mc}[c/m], f_{ml}[l/m], Ops>$

if$\neg(l \Rightarrow hcovert(u, c)) \wedge \neg(\exists c'<m, c'> \in f_{mc})$

[삽입 II] $<u, d, k, s, f_{mc}, f_{ml}, INSERT(u, <m, c, l>)$; $Ops>$

$\triangleright$ $<u, d, k, <fst(s)-\{c'\}\cup\{c\}, snd(s)>, f_{mc}[c/m], f_{ml}[l/m], Ops>$

if$\neg(l \Rightarrow hcovert(u,c)) \wedge \exists c'<m, c'> \in f_{mc}$

[삽입 III] $<u, d, k, s, f_{mc}, f_{ml}, INSERT(u, <m, c, l>)$; $Ops>$

$\triangleright$ $<u, d, k, f_{mc}, f_{ml}, Ops>$

if $l \Rightarrow hcovert(u,c)$

프로그램의 수행을 의미하는 $EXECUTE$는 주어진 프로그램이 시스템에 존재하고, 코드를 구성하는 각 연산이 자원 접근 목록에서 유도 가능한 경우에만 정상적으로 수행된다(아래 [정상 수행]). 수행되는 연산 수행 순서 열은 시스템 s가 가지고 있는 자원의 상태에 따라 결정되는데 이것은 보다 구체적인 의미론의 정의를 요구하므로 본 논문의 범위를 넘는다. 따라서, 여기서는 시스템 s가 가지는 자원 상태에서 c를 수행할 때 실행되는 연산 열을 나타내는 $selected(snd(s), c)$ 함수로 대신한다. 개개의 프로그램 연산들을 수행하는 run의 의미도 운영체제가 수행하는 각 연산의 의미론과 관련 있는 부분으로써 본 논문의 범위를 벗어나므로 다루지 않는다. 단, run은 SKETHIC 상태를 변화시키지 않는다고 가정한다.

[정상 수행]$<u, d, k, s, f_{mc}, f_{ml}, EXECUTE(u, m)$; $Ops>$

$\triangleright$ $<u, d, k, s, f_{mc}, f_{ml}, run(k, o_1)\cdots;\ run(k,o_n)$; $Ops>$

if $f_{mc}(m)=c \wedge selected(snd(s), c)=<o_1,\cdots, o_n> \wedge$

$\forall o_i$ s.t. $1\le i\le n.\ f_{ml}(m)\Rightarrow o_i$

그러나, 코드를 구성하는 각 연산이 자원 접근 목록에서 유도되지 않는 경우(아래 [비정상 수행 I])에는 수행을 시도했던 프로그램이 시스템에서 삭제되며 프로그램은 비정상적으로 종료된다. 주어진 프로그램이 시스템에 제대로 설치되지 않은 경우 역시(아래 [비정상 수행 II]) 정상적으로 끝나지 못한다.

[비정상 수행 I]$<u, d, k, s, f_{mc}, f_{ml}, EXECUTE(u, m)$; $Ops>$

$\triangleright$ $<u, d, k, s, f_{mc}, f_{ml}, run(k, o_1)\cdots; run(k,o_{I-1}); DELETE(u, m)$; $Ops>$

if $f_{mc}(m)=c \wedge selected(snd(s), c)=<o_1,\cdots, o_n> \wedge$

$\exists o_i$ s.t. $1\le i\le n, \neg(f_{ml}(m)(o_i) \wedge$

$I=\min\{ j : 1\le j\le n \wedge \neg(f_{ml}(m)\Rightarrow o_j)\}$

[비정상 수행 II]$<u, d, k, s, f_{mc}, f_{ml}, EXECUTE(u, m)$; $Ops>$

$\triangleright$ $<u, d, k, s, f_{mc}, f_{ml}, DELETE(u, m)$; $Ops>$

if $\neg(\exists c\ f_{mc}(m)=c) \vee \neg(\exists\ c\ f_{ml}(m)=l)$

삭제($DELETE$)는 프로그램 수행 중 감시 시스템에 의해 트로이 목마로 밝혀졌을 때 수행 된다. 주어진 프로그램에 관련 코드 및 자원 접근 리스트에 관련된 자료들을 시스템, f_{mc}, f_{ml} 에서 찾아 삭제한다(아래 [삭제 I]). 인자로 주어진 코드가 기존에 없는 경우에는 무시된다(아래 [삭제 II]).

[삭제 I] $<u, d, k, s, f_{mc},, f_{ml}, DELETE(u, m)$; $Ops>$

$\triangleright$ $<u, d, k, <fst(s)-\{c\}, snd(s)>, f_{mc}-\{<m, c>\}, f_{ml}-\{<m, l>\}, Ops>$

if $\exists\ c <m, c> \in f_{mc} \vee \neg\ l <m, l> \in f_{ml}$

[삭제 II] $<u, d, k, s, f_{mc},, f_{ml}, DELETE(u, m)$; $Ops>$

$\triangleright$ $<u, d, k, s, f_{mc}, f_{ml}, Ops>$

if $\neg(\exists\ c <m, c> \in f_{mc}) \vee \neg(\exists\ l <m, l> \in f_{ml})$

하나 이상의 단위 연산 수행이 이어지는 경우는 다음과 같이 '$\triangleright\triangleright$'으로 나타낸다. 이는 '$\triangleright$'를 통하여 재귀 규칙(reflexive)과 이행 규칙(transitivity) 으로 상태를 추론하는 것을 의미한다. 정의는 다음과 같은 귀납식으로 기술된다.

[연산들] $<u, d, k, s, f_{mc}, f_{ml}, x_1;x_2;..x_n;Ops> \triangleright\triangleright <u, d, k, s', f_{mc}', f_{ml}', Ops)>$

if 다음 중 하나일 때,

(i) $n=0 \wedge s=s' \wedge f_{mc}=f_{mc}' \wedge f_{ml}\ f_{ml}'$

(ii) $n=1 \wedge \langle u, d, k, s, f_{mc}, f_{ml}, x_1; Ops\rangle \triangleright \langle u, d, k, s', f_{mc}', f_{ml}', Ops\rangle$

(iii) $n > 1 \wedge$

$\langle u, d, k, s, f_{mc}, f_{ml}, x_1;x_2;..x_n;Ops\rangle \triangleright \triangleright \langle u, d, k, s'', f_{mc}'', f_{ml}'', x_n;Ops\rangle$

$\wedge \langle u, d, k, s', f_{mc}'', f_{ml}'', x_n; Ops\rangle \triangleright \langle u, d, k, s', f_{mc}', f_{ml}', Ops\rangle$

4.3 안전성

앞서 정의 10에서와 같이 트로이 목마에 대하여 안전한 상태의 시스템을 트로이 목마가 전혀 없는 시스템으로 정의한다면 이것은 수행 전에 이미 트로이 목마인지를 판별하여 시스템에 절대 들어오지 못하게 한다는 의미가 될 것이다. 하지만, 이는 알려지지 않은 트로이 목마에 대한 대응 방안이 되기에 현실적으로 매우 어렵다. 따라서 여기서는 다음과 같이 안전한 시스템의 정의를 완화시킨다.

정의 13 상태$\langle u, d, k, s, f_{mc}, f_{ml}, Ops\rangle$가 트로이 목마에 관해 안전한 상태에 있다는 것은 다음과 같이 정의된다.

모든 사용자 u에 대해, u에 대한 트로이 목마 t가 $fst(s)$에 존재한다면, 어떤 $o \in hcovert(u,t)$도 절대 수행되지 않는다.

다음은 앞서 기술된 정의에 비추어 SKETHIC이 시스템의 안전성을 보장함을 보이는 정리들이다. 자세한 증명은 부록에 있다.

정리 2 상태 $T1=\langle u, d, k, s, f_{mc}, f_{ml}, Ops\rangle$이 안전한 상태라고 하면, 사용자 u에 의해 수행되는 SKETHIC 연산 후의 상태 $T2=\langle u, d, k, s', f_{mc}', f_{ml}', Ops'\rangle$도 안전하다.

정리 3 초기 상태 $T0=\langle u, d, k, s, f_{mc}, f_{ml}, Ops\rangle$이 안전한 상태이면 앞서 정의된 연산들에 의해 관리되는 SKETHIC 시스템은 언제나 안전한 상태로 유지된다.

5. 비교 및 토의

소스 코드에 자원 접근 목록을 첨부해서 전달하는 접근 방법은 다소 비실용적으로 보일 수 있다. 그러나, 이미 메시지 다이제스트(message digest), 시그니쳐(signature) 등의 보안 관련 자료의 첨부가 이미 일반화 되어 있는데다[10], 증명 수반 코드(proof carrying code)[17]나 소프트웨어 검증(software verification)[6] 등에서 보다 일반적인 자료의 첨부도 활발하게 연구되고 있다. 더욱이 CASE 도구의 발달로 접근 제어 리스트를 첨부하는 개발자를 돕는 프로그램 분석 도구들[3,14]이 지원되는 것을 가정할 때 제안된 인프라의 보편화 가능성은 이미 무시할 수 없는 수준에 이르렀다고 보인다.

증명 수반 코드(proof carrying code) 방식에서도 SKETHIC과 비슷하게 특정 성질에 관련된 자료가 코드와 함께 첨가된다. 그러나, 단순히 성질 그 자체가 아닌 그 성질을 정적으로 판명하기 위한 증명이란 점이 그 특징이다. 그리고, SKETHIC과 달리 증명 수반 코드 방식에서의 사용자(관리자)는 프로그램이 수행되기 이전에 코드의 취함과 버림을 1차 판별하거나 하는 등의 개입이 없다. 이는 사용자의 인식에 위배되는지 여부가 판별의 중요한 요소로 작용하는 트로이 목마의 성격을 고려할 때 그다지 적절한 방법은 아니라고 볼 수 있다. 또, SKETHIC에서는 첨부 자료의 타당성을 수행 중 감시 시스템으로 확인하지만, 증명 수반 코드에서는 첨부된 증명이 맞았는지를 정적으로 확인하게 된다. 따라서 증명 수반 코드는 수행 전에 모든 판명이 완료되므로 수행 중 프로그램 성능은 높을 수 있으나 정적인 판명의 한계를 가지기도 한다.

이동 에이전트(mobile agent) 시스템에서의 코드는 각 컴퓨터를 항해하는 프로그램들을 지칭한다. 코드의 항해를 관장하는 관리 시스템 위에서 코드의 이동이 이루어지게 되는데, 이 때 각 코드마다 자원 소모에 관한 자료를 함께 첨부하여 이동시키는 예가 있다[18]. 항해 중인 코드가 해당 컴퓨터에서 수행될 수 있는지에 관한 판명은 참여 중인 모든 호스트들을 관리하는 중앙 관리 시스템에서 코드에 첨부된 자료에 의거하여 이루어진다. 이 경우에도 증명 수반 코드와 마찬가지로 프로그램의 악성 여부 판명에 사용자(관리자)의 개입이 없다.

SKETHIC 메커니즘 및 안전성을 기술하기 위해 본 연구에서는 형식적 프레임워크를 새로 정의하였다. 기존의 Timbley에 의해 제안된 프레임워크[15]는 악성 행위 수행 방지 측면보다는 원본 프로그램과의 차이와 무결성 검증 관점에서 트로이 목마, 바이러스 모두를 포함하는 보다 추상적이고 일반적인 정의로 구성되어 있는 관계로, SKETHIC의 기술을 위해 사용하게 되는 경우 불필요한 복잡도가 높아지게 된다는 단점이 있다.

본 논문의 방식은 사용자가 가지는 접근 제어 정책 설정의 부담을 개발자로 이전시킴으로써 사용자의 부담을 줄이지만, 이에 따르는 개발자의 부담이 상대적으로 늘어날 것으로 예상된다. 그러나, 개발자는 프로그램에

대해 이미 알고 있으므로 접근 제어 정책을 정의하는데 드는 부담이 사용자의 그것보다 매우 적을 것이다. 그리고, 개발자는 이진 수행 코드 외의 프로그램의 소스 코드를 가지고 있으므로 정적 분석 도구[3,14]의 도움을 효과적으로 받을 수 있다는 잇점도 있다.

트로이 목마가 아닌 경우에도 개발자가 만든 자원 접근 목록이 사용자가 보기에 트로이 목마로 비추어져 그 프로그램의 사용을 거부하는 경우가 발생될 소지가 없지는 않다. 그러나, 우선 개발자 도구 및 사용자 도구의 지원으로 이를 어느 정도 피할 수 있을 것으로 보이며, 장기적으로는 프로그램의 개발자들로 하여금 시스템의 내부적인 자원을 최소로 사용하도록 하는 것을 유도하여, 선량한 프로그램들이 트로이 목마 프로그램과의 변별력을 더 많이 가지도록 하는 효과를 얻는다.

6. 구 현

SKETHIC의 구현은 전체 자원 접근 목록에 대한 모델의 정의와 개발자 및 사용자 도구와 감시 시스템의 구현으로 볼 수 있다.

자원 접근 목록 형태는 접근 제어 정책을 표현할 수 있는 것이라면 모두 가능하다고 볼 수 있다. SKETHIC에서는 접근 제어 정책을 사용자가 설정하는 것이 아니므로, 다소 복잡한 접근 제어 모델의 적용도 용이하나, 우선 제안 된 메커니즘의 적용성을 보이기 위해 Java의 각 메소드에 명세에 해당 유한 상태 전이 기계를 구문[19] 형태로 바꾸어 첨부한다. 이에 따라 개발자 자료 첨부 도구는, 각 Java API 마다 기본적인 유한 상태 전이 기계가 할당된 상태에서, 각 사용되는 모듈의 첨부 자료를 취합하여 전체 프로그램의 첨부 자료를 얻는 방식으로 접근 하고 있다. 현재는 정적인 기계적 추론을 해내는 것만을 고려하고 있다.

사용자의 1차 판별을 돕는 도구로는 각 프로그램이 어떠한 분류에 속하는지 사용자로부터 입력 받아 해당 분류의 프로그램들에서 기대할 수 있는 행위에 위배되는 연산이 있으면 트로이 목마로 간주하는 방식을 구현하였다[7]. 감시 시스템은 기존의 보안 운영 체제와 같이 커널 단계에서 프로그램 수행을 방지하는 방식[8,9,16]으로 구현하였다. 첨부된 자원 접근 목록은 앞서 언급했듯이 구문 형태이므로 감시시스템의 판별에는 일반 파서[19]의 구문 분석과 동일한 노력이 든다고 볼 수 있다. 사용자 도구와 감시 시스템은 Linux 및 윈도우 2000 등을 기반으로 구현하였으나 현재 Java로 이전 중에 있다.

7. 요약 및 향후 연구 과제

SKETHIC은 알려지지 않은 트로이 목마를 탐지하는 기법으로서 개발자가 보내온 첨부 자료를 바탕으로 코드 수행 전에 안전한 프로그램인지를 선별하고 수행 중에는 행위를 감시하여 그 첨부 자료의 진위를 밝힌다. 이로써 사용자의 정책 선정 부담이 덜어지게 되며, 사용자에 의해서 받아들여진 프로그램만을 감시 대상으로 하기 때문에 감시 시스템의 부담도 줄어들게 된다고 볼 수 있다. 개발자로 전이된 접근 제어 정책 설정의 부담은 사용자의 부담에 비하면 작을 뿐 아니라, CASE의 발달 및 정적 분석 도구의 제공으로 해소 될 수 있다고 전망된다. 또 본 논문에서는 이러한 SKETHIC을 형식 언어로 기술하고 이 방식이 시스템을 안전하게 유지하는 지를 보였다.

현재는 앞서 6장에서 간단히 소개된 SKETHIC 메커니즘의 1차 구현을 진행 중에 있으며, 보다 효과적으로 트로이 목마에 대응할 수 있는 자원 접근 목록의 모델을 찾는 방안도 함께 모색하고 있다.

참 고 문 헌

[1] Symantec AntiVirus for Macintosh - knowledge-base - About Trojan Horses, http://service1.symantec.com/SUPPORT/num.nsf/d514450ab7fffddf852565a600636fb9/900ea4166e937d52852565a60063a176?OpenDocument

[2] Virus Protection, http://helpdesk.uvic.ca/how-to/support/virus.html#winfeatures

[3] R.W. Lo, K.N. Levitt, R.A. Olsson, "MCF: a Malicious Code Filter". *Computers & Security*, 1995, Vol.14, No.6, pp. 541-566

[4] Trojan Port List, http://www.glocksoft.com/trojan_port.htm

[5] C. Ko, G. Fink, K. Levitt , "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring". *Proc. of the 10th Annual Computer Security Applications Conference,* Orlando, FL, 5-9 Dec. 1994, pp. 134-144

[6] F. Schneider, "Enforceable Security Policies", Techbnical Report, TR98-1664, Dept of Computer Science, Cornell University, 1998

[7] A. Acharya, M. Raje, "MAPbox : Using Parameterized Behavior Classes to Confine Applications", Technical report, TRCS99-15, Dept. of Computer Science, University of California, Santa Barbara

[8] EROS: The Extremely Reliable Operating System, http://www.eros-os.org

[9] T. Mitchem and R. Lu and R. O'Brien, "Using Kernel Hypervisors to Secure Applications", *in the Proc. of the Annual Computer Security Application Conference(ACSAC97)*, 1997
[10] S. Oaks, "Java Security", O'Reilly, 1998
[11] "Trojan Horses", http://www.ladysharrow.ndirect.co.uk/Maximum%20Security/trojans.htm
[12] S. Mann, E. L. Mitchell, "Linux System Security : An Administrator's Guide to Open Source Security Tools", Prentice Hall PTR, 2000
[13] "Navidad.exe", http://home.ahnlab.com/virus-info/navidad.html, 2000
[14] J. Viega, J.T. Bloch, T. Kohno, G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code", In Proc. of the 16th Annual Computer Security Applications Conference(ACSAC'00), 2000
[15] H. Thimbleby, S. Anderson, P. Cairns, "A framework for modeling Trojans and computer virus infection", *Computer Journal*, Vol. 41 No. 7, pp444--458, 1999
[16] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, "A Secure Environment for Untrusted Helper Applications confining the Wily Hacker", Technical Report, University of California Berkeley, 1996
[17] G.C. Necula and P. Lee, "Safe Kernel Extensions Without Run-Time Checking", *in the Proc. of the Second Symposium on Operating Systems Design and Implementation(OSDI'96)*, 1996
[18] R. D. Nicola, G. Ferrari, R. Pugliese, "Types as Specifications of Access Polices", Available at http://rap.dsi.unifi.it/papers/html. To appear in Theoretical Computer Science, http://www.cs.engr.uky.edu/~lewis/essays/compilers/ll-lang.html
[19] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley 1986

부 록

다음은 본문 4장에서 소개된, SKETHIC이 시스템의 안전성을 보장함을 보이는 정리를 증명한다. 먼저, 다음과 같은 보조정리가 필요하다.

보조정리 1 트로이 목마 $<m, t, l>$이 SKETHIC에 상태에 존재한다면, 모든 $o \in hcovert(u, t)$에 대해 $\neg (l \Rightarrow o)$이다.

증명 앞서 SKETHIC 연산의 정의에 따르면, 프로그램을 시스템에 삽입시키는 연산은 *INSERT*가 유일하다. 따라서, $<m, t, l>$도 $INSERT(u, <m, t, l>)$을 통해 삽입되었음을 알 수 있다.

만일 프로그램 $<m, t, l>$에 대해 어떤 $o \in hcovert(u, t)$ 이 $l \Rightarrow o$라면, $<m, t, l>$을 삽입시킬 *INSERT* 연산은 [삽입 III] 규칙에 의해 동작하게 되며, 이는 삽입이 실패함을 나타낸다. 결국 $l \Rightarrow o$인 $o \in hcovert(u, t)$을 가지는 프로그램 $<m, t, l>$은 SKETHIC 상태에 결코 존재할 수 없다. □

다음 정리는 안전한 상태에서 정상적인 프로그램을 삽입했을 때 계속 안전한 상태를 유지함을 의미한다. 우선, *INSERT*를 수행해도 삽입되는 프로그램과 식별자가 다른 프로그램들은 그 수행에 영향을 받지 않는다는 것을 나타내는 보조정리들이 필요하다.

보조정리 2 상태 $S1=<u, d, k, s, f_{mc}, f_{ml}, Ops>$와 상태 $S2=<u, d, k, s', f_{mc}', f_{ml}', Ops'>$에서 $snd(s)=snd(s')$이고 $f_{mc}(m)=f_{mc}'(m)$이고 $f_{ml}(m)=f_{ml}'(m)$이면, 모든 프로그램에 m 대해, $S2$ 상태에서 $EXECUTE(u, m)$을 수행하는 것은 $S1$ 상태에서 $EXECUTE(u, m)$ 수행하는 것과 동일한 절차와 결과를 가진다.

증명 $S2$에서 $EXECUTE(u, m)$를 수행할 때, 수행을 좌우하는 인자가 되는 것은 [정상 수행], [비정상 수행 I], [비정상 수행 II]의 정의에 의하면 $m, k, f_{mc}', f_{ml}', snd(s')$ 이다.(1)에 의해 이것은 $S1$에서의 인자들의 값 $m, k, f_{mc}, f_{ml}, snd(s)$와 동일하므로 $S1$에서 $EXECUTE$ (u, m)를 수행할 때와 절차와 결과가 다르지 않다. □

보조정리 3 상태 $S1=<u, d, k, s, f_{mc}, f_{ml}, Ops>$에 $INSERT(u, <m_1, c, l>)$을 수행한 결과가 $S2=<u, d, k, s', f_{mc}', f_{ml}', Ops'>$라 하자. 이 때, $\neg (m_1=m_2)$ 인 프로그램에 m_2 대해, $S2$ 상태에서 $EXECUTE(u, m_2)$을 수행하는 것은 $S1$ 상태에서 $EXECUTE(u, m_2)$ 수행하는 것과 동일한 절차와 결과를 가진다.

증명 *INSERT*에 관련된 연산 규칙에 따르면 $INSERT(u, <m_1, c, l>)$의 수행 후에도 $snd(s)$는 변하지 않고, f_{mc}, f_{ml}도 m_1에 관계 되는 부분만이 변화된다. 즉, $snd(s)=snd(s')$이고 $f_{mc}(m_2)=f_{mc}'(m_2)$이고 $f_{ml}(m_2)=f_{ml}'(m_2)$이다. 따라서 보조정리 2에 의해 $S2$에서 $EXECUTE(u, m_2)$를 수행할 때는 $S1$에서 $EXECUTE$ (u, m_2)를 수행할 때와 절차와 결과가 다르지 않다. □

보조정리 4 상태 $S1=<u, d, k, s, f_{mc}, f_{ml}, Ops>$이 안전한 상태라고 하면, 사용자 u가 트로이 목마가 아닌 프로그램 $p=<m, c, l>$를 삽입한 상태 $S2=<u, d, k, s', f_{mc}', f_{ml}', Ops'>$도 안전하다.

증명 $S2$에 트로이 목마 $p_t=<m_t, t, l_t>$가 존재한다면, 정리의 가정에 의해 p가 트로이 목마가 아니므로 p_t는 새로 삽입되는 p는 아니다. 따라서 이 p_t 는 $S1$에서부터 존재한 것으로 볼 수 있다.

(i) [삽입 I]에 의해 $INSERT(u, p)$가 수행된 경우(즉, $S1$에 m을 식별자로 하는 프로그램이 존재하지 않는 경우) :

$S1$에 m을 식별자로 하는 프로그램이 존재하지 않으므로, p_t는 m을 식별자로 할 수 없다. 즉, $\neg(m_t=m)$ 이다.

여기에 보조정리 1을 사용하면, $EXECUTE(u, m_t)$의 수행이 안전한 상태인 $S1$에서와 동일하다. 따라서, $S2$에서도 $o\in hcovert(u, t)$의 수행은 $S1$과 마찬가지로 불가능하게 된다.

(ii) [삽입 II]에 의해 $INSERT(u, p)$ 수행된 경우(즉, $S1$에 m을 식별자로 하는 프로그램이 존재하는 경우) :

$<m_t, t, l_t>$이 $S2$에 존재한다면, $EXECUTE(u, m_t)$에 의해 코드 t를 수행할 수 있는 경우를 의미한다.

그런데, $m_t=m$이면, 기존의 트로이 목마 프로그램을 정상 프로그램으로 덮어쓰는 것이므로, [삽입 II]의 정의에 의해 $f_{mc}(m_t)$이 t에서 정상 코드인 c로 바뀌는 $S2$ 상태에서는 $<m_t, t, l_t>$가 존재한다고 말할 수 없다.

따라서, $\neg(m_t=m)$이므로, (i)과 같이 보조정리 3에 의해 $EXECUTE(u, m_t)$의 수행이 안전한 상태인 $S1$에서와 동일하므로 $S2$에서도 $o\in hcovert(u, t)$의 수행은 불가능하게 된다.

결과적으로,(i),(ii)에 의해 $S2$에서도 $o\in hcovert(u, t)$의 수행이 불가능하여 $S2$는 정의 13의 안전한 상태의 정의를 만족하게 된다. □

다음 이어지는 정리는 안전한 상태에서 트로이 목마 프로그램을 삽입했을 때에도 계속 안전한 상태를 유지함을 의미한다. 이를 보이기 위해 먼저 안전한 상태에 삽입된 트로이 목마 프로그램 $<m, t, l>$의 $hcovert(u,t)$에 속하는 연산은 수행되지 않는다는 보조정리가 필요하다.

보조정리 5 안전한 상태 $S1=<u, d, k, s, f_{mc}, f_{ml}, Ops>$에서 사용자 u가 트로이 목마인 프로그램 $p=<m, c, l>$를 삽입한 상태 $S2=<u, d, k, s', f_{mc}', f_{ml}', Ops'>$라 하면, $S2$에서 $EXECUTE(u, m)$의 수행 시 어떤 $o\in hcovert(u, c)$도 수행되지 않는다.

증명 $selected(snd(s), c)=<o_1,\cdots, o_n>$ 이 $o=o_i\in hcovert(u, t)$를 포함하지 않는 경우에는 o의 수행이 안 되는 것이 자명하다.

$selected(snd(s), c)=<o_1,\cdots, o_n>$이 $o=o_i\in hcovert(u, t)$를 포함하는 경우에는 보조정리 1에서 $\neg(l\Rightarrow o)$인 o를 포함하게 되는데, $EXECUTE$의 정의에 의해 [비정상 수행 I]에 의해 수행된다. [비정상 수행 I]의 정의에서 run에 의해 수행되는 연산은 $I=\min\{j: 1\le j\le n \wedge \neg(l\Rightarrow o_j)\}$일 때, $o_1\cdots o_{I-1}$ 이며, 이것은 최초의 $\neg(l\Rightarrow o_j)$인 연산을 만나기 직전까지만 수행하게 된다. 따라서, $I\le i$이고 o_i는 수행되지 않는다. □

보조정리 6 상태 $S1=<u, d, k, s, f_{mc}, f_{ml}, Ops>$이 안전한 상태라고 하면, 사용자 u가 트로이 목마인 프로그램 $p=<m, c, l>$를 삽입한 상태 $S2=<u, d, k, s', f_{mc}', f_{ml}', Ops'>$도 안전하다.

증명 $S2$에 트로이 목마 $p_t=<m_t, t, l_t>$가 존재한다고 가정한다.

(i) [삽입 I]에 의해 $INSERT(u, p)$ 수행된 경우(즉, $S1$에 m을 식별자로 하는 프로그램이 존재하지 않는 경우) :

$p_t=p$일 때는 보조정리 5에 의해 $o\in hcovert(u, t)$는 수행되지 않는다.

p_t가 기존의 $S1$으로부터 존재하는 것일 때, $S1$에 m을 식별자로 하는 프로그램이 존재하지 않으므로$((m_t=m)$ 이고 보조정리 3에서 $o\in hcovert(u, t)$는 *$S2$에서 수행될 수 없다.*

(ii) [삽입 II]에 의해 $INSERT(u, p)$ 수행된 경우(즉, $S1$에 m을 식별자로 하는 프로그램이 존재하는 경우) :

$p_t=p$인 경우는(i)과 동일하게 보조정리 5에 의해서 증명된다.

p_t가 기존의 $S1$으로부터 존재하는 것일 때, 만일 $m_t=m$ 이라면, 보조정리 4의(ii)에서와 같이 기존의 트로이 목마 프로그램을 덮어쓰는 경우에는 [삽입 II]의 정의에 의해 $S2$ 상태에서는 $<m_t, t, l_t>$가 존재한다고 말할 수 없다. 따라서, $\neg(m_t=m)$ 이다. 그리고 이 때(i)과 동일하게 보조정리 3에서 증명된다.

결과적으로, (i), (ii)에 의해 $S2$에서도 $o\in hcovert(u, t)$ 의 수행이 불가능하여 $S2$는 정의 13의 안전한 상태의 정의를 만족하게 된다. □

다음 정리는 안전한 상태에서 삭제를 수행했을 때도 여전히 안전한 상태를 유지함을 보인다. 먼저 보조정리 3와 비슷하게 $DELETE(u,m)$를 수행해도 삭제되는 프로그램과 식별자가 다른 프로그램들은 그 수행에 영향을 받지 않는다는 것을 나타내는 보조정리를 정의한다.

보조정리 7 상태 $S1=<u, d, k, s, f_{mc}, f_{ml}, Ops>$에 $DELETE(u,<m_1, c, l>)$을 수행한 결과가 $S2=<u, d, k, s', f_{mc}', f_{ml}', Ops'>$라 하자. 이 때, $\neg(m_1=m_2)$인 프로그램에 m_2 대해, $S2$ 상태에서 $EXECUTE(u, m_2)$을 수행하는 것은 $S1$ 상태에서 $EXECUTE(u, m_2)$ 수행하는 것과 동일한 절차와 결과를 가진다.

증명 $DELETE$에 관련된 연산 규칙에 따르면

DELETE(*u,m*)의 수행 후에도 *snd*(*s*)는 변하지 않고, f_{mc}, f_{ml} 도 m_1에 관계 되는 부분만이 변화된다. 따라서, *snd*(*s*)=*snd*(*s'*)이고 $f_{mc}(m_2)=f_{mc}'(m_2)$이고 $f_{ml}(m_2)=f_{ml}'(m_2)$이다. 따라서 보조정리 2에 의해 *S2*에서 *EXECUTE*(*u*, m_2)를 수행할 때는 *S1*에서 수행할 때와 절차와 결과가 다르지 않다.(

보조정리 8 상태 *S1*=<*u*, *d*, *k*, *s*, f_{mc}, f_{ml}, *Ops*>이 안전한 상태라고 하면, 사용자 *u*가 *m*으로 식별되는 프로그램을 삭제한 상태 *S2*=<*u*, *d*, *k*, *s'*, f_{mc}', f_{ml}', *Ops'*>도 안전하다.

증명 *S2*에서 트로이 목마 프로그램 p_t=<m_t, *t*, l_t>를 수행시킨다고 할 때.

(i) [삭제 I]에 의해 삭제되는 경우(즉, *m* 인 프로그램이 *s*에 존재하는 경우) :

*m*에 관련된 자료들은 *DELETE*에 의해 이미 *fst*(*s'*), f_{mc}', f_{ml}'에는 존재하지 않으므로 m_t 가 *m* 일수는 없다. 즉,($m=m_t$) 이다.

따라서 보조정리 7에 의해 *S2* 상태에서 *EXECUTE*(*u*, m_t)을 수행하는 것은 안전한 *S1* 상태에서 *EXECUTE*(*u*, m_t) 수행하는 것과 동일한 절차와 결과를 가지므로 *o*∈*hcovert*(*u,t*) 가 수행되지 않는다.

(ii) [삭제 II]에 의해 삭제되는 경우, 즉, *m*인 프로그램이 *s*에 존재하지 않는 경우 :

s, f_{mc}, f_{ml}가 불변하므로 따라서, 보조정리 2에 의해 모든 트로이 목마 프로그램 p_t에 대해 *S2*에서의 *EXECUTE*(*u*, *t*)의 절차와 결과가 안전한 상태인 *S1*에서와 동일하므로, *S1*에서와 같이 *o*∈*hcovert*(*u,t*)도 수행이 되지 않는다. □

보조정리 6와 보조정리 8을 바탕으로 다음과 같이 본문의 정리 2와 3이 유도될 수 있다.

정리 2 상태 *T1*=<*u*, *d*, *k*, *s*, f_{mc}, f_{ml}, *Ops*>이 안전한 상태라고 하면, 사용자 *u*에 의해 수행되는 SKETHIC 연산 후의 상태 *T2*=<*u*, *d*, *k*, *s'*, f_{mc}', f_{ml}', *Ops'*>도 안전하다.

증명 각 SKETHIC 연산 각각에 관해 성립함을 보이면 된다. 앞서 보조정리 6와 8에 의해 *INSERT*와 *DELETE*에 대해서는 성립한다. *EXECUTE*와 *run*은 정의에서 *snd*(*s*), f_{mc}, f_{ml}가 수행 전 후로 바뀌지 않는다. 따라서 보조정리 2에 의해서, 트로이 목마 프로그램 p_t=<m_t, *t*, l_t>에 대해 *S2*에서 *EXECUTE*(*u*, m_t)가 수행될 때는 안전한 *S1*에서 수행될 때와 동일하게 *o*∈*hcovert*(*u,t*)가 수행되지 않는다. 그러므로, 모든 SKETHIC 연산에 대해 성립한다. □

정리 3 초기 상태 *T0*=<*u*, *d*, *k*, *s*, f_{mc}, f_{ml}, *Ops*>이 안전한 상태이면 앞서 정의된 연산들에 의해 관리되는 SKETHIC 시스템은 언제나 안전한 상태로 유지된다.

증명 초기 상태 *T0*에서 *Ops*=x_1;… ; x_n;*Ops'*라 하고, x_1;… ; x_n을 수행 하고 난 현재 상태가 *Tn*=<*u*, *d*, *k*, *s'*, f_{mc}', f_{ml}', *Ops'*>라고 하자. *T0*▷▷*Tn*이므로 ▷▷의 정의에 의해 다음 세 가지 경우로 나눌 수 있다.

(i) *T0*=*Tn* : *T0*가 안전한 상태이므로 자명하다.

(ii) *T0*▷*T1*▷*Tn* : 정리 2에 의해서 *Tn*이 안전한 상태임이 증명된다.

(iii) 그 외 *T0*▷▷*Tn*이고 *n* > 1 :

▷▷의 정의에 의해 *T0*▷▷<*u*, *d*, *k*, s'', f_{mc}'', f_{ml}'', x_n;*Ops'*>이고, <*u*, *d*, *k*, s'', f_{mc}'', f_{ml}'', x_n; *Ops'*>▷*Tn* 인 *Tx*=<*u*, *d*, *k*, s'', f_{mc}'', f_{ml}'', x_n;*Ops'* >가 존재한다. *T0*▷▷*Tn에서 수행한* SKETHIC 연산의 수가 *I*라면, *T0*▷▷*Tx*에서 수행한 SKETHIC 연산의 수는 *I-1*이다.

만일 초기 상태로부터 수행된 연산의 수가 *I-1*일 때 본 정리가 성립한다면, *Tx*는 안전한 상태이고, 이 때 위 (ii)과 *Tx*(*Tn*에서 다시 *Tn*이 안전한 상태가 된다. 따라서, 초기 상태 이후 수행된 SKETHIC 연산의 수가 *I-1*인 경우 성립한다면, *I*인 경우에도 성립한다.

따라서, 초기 상태 이후 수행된 SKETHIC 연산의 수에 대해 귀납법(induction) 을 사용하면, 위(i),(ii)을 귀납 기반(base)으로 하고(iii)을 적용하여 *Tn*도 안전한 상태임이 증명된다. □

조 은 선

1991년 서울대학교 자연과학대학 계산통계학과 졸업. 1993년 서울대학교 자연과학대학 전산과학 석사. 1998년 8월 서울대학교 자연과학대학 전산과학 박사. 1999년 ~ 2000년 4월 한국과학기술원 전산학과 연구원. 2000년 5월 ~ 2002년 2월 아주대학교 정보통신전문대학원 조교수대우. 2002년 ~ 현재 충북대학교 전기전자컴퓨터학부 조교수.

예 홍 진

서울대학교 사범대학 수학교육과(1986) 석사 아주대학교 대학원 전자계산학과(1988) G.E.A. 프랑스 Grenoble 1 대학 응용수학과(1990) 박사 프랑스 Lyon 1 대학 전자계산학과(1993) 1993년 ~ 현재 아주대학교 정보 및 컴퓨터공학부 조교수. 관심분야는 컴퓨터 산술, 병렬 알고리즘과 구조, VLSI알고리즘

오 세 창

1988년 연세대학교 전산학과 졸업. 1990년 한국과학기술원 전산학과 석사. 1997년 한국과학기술원 전산학과 박사. 1995년 ~ 1999년 LG 종합기술원 선임연구원. 1999년 ~ 2000년 (주)이니트 이사. 2000년 ~ 현재 아주대학교 정보통신전문대학원 조교수대우.

홍 선 호

1986년 서울대학교 수학과 졸업. 1989년 오하이오 주립대학교 통계학과 석사. 1992년 남감리대학교 통계학과 박사. 1993년 ~ 1997년 세종대학교 응용통계학과 전임강사. 1997년 ~ 1999년 남감리대학교(Southern Mehodist University) 통계학과 방문교수. 2000년 ~ 현재 아주대학교 정보통신전문대학원 조교수 대우.

홍 만 표

서울대학교 자연과학대학 계산통계학과(1981) 석사 서울대학교 자연과학대학 계산통계학과(1983) 박사 서울대학교 자연과학대학 계산통계학과(1991) 1983년 ~ 1985년 울산공과대학 전자계산학과 전임강사. 1985년 ~ 1999년 아주대학교 정보및 컴퓨터공학부 교수. 1993년 ~ 1994년 미네소타대학 전자공학과 교환교수. 1999년 ~ 현재 아주대학교 정보통신전문대학원 교수. 2000년 ~ 2001년 조지와싱턴 대학교 컴퓨터과학과 교환교수. 관심분야는 병렬처리 및 컴퓨터 보안