http://dx.doi.org/10.3745/KTCCS.2013.2.7.303

Theory and Implementation of Dynamic Taint Analysis for Tracing Tainted Data of Programs

Hyun-il Lim†

ABSTRACT

As the role of software increases in computing environments, issues in software security become more important problems. Dynamic taint analysis is a technique to trace and manage tainted data originated from unreliable sources during the execution of a program. This analysis can be applied to software security verification as well as software behavior understanding, testing unexpected errors, or debugging. In the previous researches, they focussed only to show the analysis results of dynamic taint analysis, and they did not logically describe propagation process of tainted data and analysis procedures. So, there were difficulties in understanding the analysis procedures or applying to other analysis. In this paper, by theoretically describing the analysis procedure, we logically show how the propagation process of tainted data can be traced, and present a theoretical model for dynamic taint analysis. In addition, we verify the correctness of the proposed model by implementing an analyser, and show that propagation of tainted data can be traced by the model. The proposed model can be applied to understand the analysis procedures of data flows in dynamic taint analysis, and can be used as an base knowledge for designing and implementing analysis method, which applies such analysis method.

Keywords : Dynamic Taint Analysis, Software Security, Software Analysis

프로그램의 오염 정보 추적을 위한 동적 오염 분석의 이론 및 구현

임 현 일†

요 약

컴퓨팅 환경에서 소프트웨어가 차지하는 역할이 커지면서 소프트웨어 보안은 더욱 중요한 문제가 되고 있다. 동적 오염 분석은 프로그램 실행 중에 신뢰할 수 없는 소스로부터 유래된 오염된 데이터의 이동을 추적하고 관리하는 분석 방법이다. 이 분석 방법은 소프트웨어의 보안 검증 뿐만 아니라 소프트웨어의 동작을 이해하고, 예상하지 못한 오류에 대한 테스팅 및 디버깅 등에서 활용할 수 있다. 기존에 이와 관련한 연구에서는 동적 오염 분석을 이용한 분석 사례를 보여주고 있지만, 동적 오염 분석에서 오염된 정보 전파 과정 및 동작 과정에 대해서 체계적이고 논리적으로 기술하지 못하고 있다. 본 논문에서는 이런 분석 과정을 이론적으로 기술함으로써 오염된 정보의 전파 과정을 어떻게 추적할 수 있는지 논리적으로 보여주고, 이를 응용할 수 있는 이론적 모델을 제시하고 있다. 본 논문에서 기술한 이론적 모델에 대해서 분석기를 구현하고 프로그램에 대한 분석 결과를 통해서 모델의 정확성을 검증한다. 그리고, 프로그램에 나타나는 오염 정보들의 전파 과정을 보이고 결과를 검증한다. 본 이론적 모델은 동적 오염 분석에서 자료 흐름의 분석 과정을 이해하고 이를 활용하는 분석 방법을 설계하거나 구현하는 기반 지식으로 활용될 수 있을 것이다.

키워드 : 동적 오염 분석, 소프트웨어 보안, 소프트웨어 분석

1. 서 론

인터넷 사용이 보편화되면서 네트워크를 통한 방대한 양의 정보들이 이동하게 되고, 이를 활용한 다양한 어플리케이션이 사용되고 있다. 그리고, 네트워크를 통한 정보 검색, 전달, 활용 등은 선택이 아니고 필수적인 연산이 되고 있다. 하지만, 수없이 많은 정보의 홍수 속에서 필요 없는 정보들을 어떻게 찾아내고 신뢰할 수 없는 정보들을 효과적으로

※ 본 연구는 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2010-0024658).
† 정 회 원 : 경남대학교 컴퓨터공학과 조교수
논문접수 : 2013년 1월 17일
수 정 일 : 1차 2013년 3월 22일
심사완료 : 2013년 3월 27일
* Corresponding Author : Hyun-il Lim(hilim@kyungnam.ac.kr)

차단할 수 있는가에 대한 문제는 현재의 컴퓨팅 환경에서 필연적으로 해결해야 하는 문제가 되고 있다.

인터넷 사용량이 증가함에 따라 소프트웨어 취약성에 기반한 악성 코드와 이로 인한 피해 사례는 지속적으로 증가하고 있다. 소프트웨어 취약성을 보완하고 잠재적인 악성 코드를 탐지할 수 있는 다양한 연구가 진행되고 있는데, 분석 과정에서 프로그램의 실행 여부에 따라 프로그램을 실행하지 않고 코드를 분석하는 정적 분석[1]과 프로그램을 실행하면서 실행 중의 상태를 분석하는 동적 분석[2]으로 나눌 수 있다. 그중에서 동적 오염 분석 (dynamic taint analysis)은 소프트웨어 취약점을 이용해서 나타날 수 있는 소프트웨어의 오류를 찾아내고 대처할 수 있는 분석 방법의 하나이다. 이 방법은 소프트웨어에서 사용하는 정보들의 신뢰성을 확보할 수 있도록 실행 중에 나타나는 정보의 흐름을 추적하고, 신뢰할 수 없는 오염된 정보(tainted information)가 어떻게 이동하고 사용되고 있는가를 관리한다. 그리고, 오염된 정보가 허가되지 않은 방법으로 사용되는 것을 방지함으로서 소프트웨어 보안을 향상시킬 수 있는 분석 방법이다.

본 논문에서는 동적 오염 분석 방법을 체계화하고 이론적으로 모델링한 결과를 제시한다. 동적 분석 방법을 효율적으로 모델링하기 위해서 프로그램의 실행 시간 동작을 잘 표현할 수 있는 이론적 기반이 될 수 있는 연산 의미론(operational semantics)을 활용하였다. 연산 의미론은 프로그램이 가지는 명령어들의 동작을 상세하게 표현할 수 있으며, 동적 분석을 이론적으로 기술하기 위해서 적절한 기술 방법이다. 본 논문에서 제시한 모델은 예제를 통해 오염 정보의 전파 과정을 보여준다. 그리고, 분석 결과의 정확성을 검증하기 위해서 분석기를 구현하고 실행 결과를 보여준다.

본 논문의 기여도는 다음과 같다.

기존의 논문에서 기술하지 못한 동적 오염 분석의 이론적 모델을 제시한다. 기존의 연구 사례는 동적 오염 분석의 원리에 대해서 간단하게 언급하고 있을 뿐 실행 과정에 대해서 체계적인 절차나 알고리즘 및 분석 과정 등을 제시하지 않고 있다. 본 논문에서는 동적 오염 분석에 대한 기존의 논문들에서 제시하지 않은 이론적인 토대를 마련함으로써 오염된 정보의 전파 과정 및 동적 오염 분석의 동작 과정을 체계적으로 보여주고 있다. 이는 향후 동적 오염 분석 및 이를 활용한 분석 방법을 적용하거나 설계할 수 있는 기반 지식이 된다.

본 논문에서는 동적 분석을 이론적으로 기술하기 위해서 연산 의미론을 이용한다. 연산 의미론은 프로그램에 정의된 명령어의 동작 과정을 이론적으로 기술할 수 있으며 이는 프로그램의 실행 시간 움직임을 표현하기에 적절하다. 연산 의미론을 이용해서 프로그램에 포함된 명령어의 동작 과정과 함께 오염 정보의 전파 과정을 함께 표현할 수 있음을 보이고 동적 분석 과정을 이론적으로 기술하는 방법을 제시하고 있다.

본 논문에서는 예제를 통해서 오염된 정보가 전파되는 과정과 이를 분석하는 절차를 보이고 있다. 이를 통해 본 논문에서 제시한 모델이 오염된 정보의 흐름을 올바르게 추적할 수 있음을 보인다. 그리고, 분석기의 구현 및 실험 결과를 통해서 본 모델의 정확성을 검증한다.

본 논문의 구성은 다음과 같다. 2장에서는 동적 오염 분석과 관련한 기존의 연구 사례를 소개한다. 3장에서는 동적 오염 분석을 기술하기 위한 언어를 정의하고 연산 의미론을 이용해서 모델링한다. 4장에서는 3장의 모델링을 이용해서 오염된 정보를 분석하는 사례를 보여주고, 구현한 분석기를 통한 실험 결과를 통해 본 논문에서 제시한 이론적 접근 모델을 검증한다. 5장에서 결론을 맺는다.

2. 관련 연구

동적 오염 분석은 동적 분석의 하나로 프로그램의 실행 시간에 나타나는 동작을 추적해야 하기 때문에 분석을 위한 실행 환경이 필요한데, Newsome 등[3] 은 동적 오염 분석 방법을 통해 오염된 데이터의 분석에 대한 연구를 제안하였다. 이 방법은 프로그램의 실행을 추적하면서 악의적인 입력(malicious input)을 통해 생길 수 있는 비정상적인 제어 흐름의 변경과 같은 보안 문제점을 찾아낼 수 있다. 그래서, 소프트웨어의 취약점을 통해 발생할 수 있는 외부의 공격을 감지하고 방어하는 분석 도구로 사용될 수 있다.

Cheng 등[4], Bai 등[5], Clause 등[6]은 이와 관련한 동적 오염 분석 사례를 소개하고 있으며, 이 방법을 활용한 악성 코드 탐지와 같은 응용 사례[7]에 대한 연구도 있다. Cheng 등[4]은 데이터의 흐름을 추적할 수 있는 보안 시스템을 개발하였다. 이 시스템은 버퍼 오버플로나 포맷 문자열 공격(format string attack)과 같은 여러 형태의 네트워크를 통한 공격을 감지할 수 있다. 이 방법에서는 직접 메모리 매핑(direct memory mapping)을 이용해서 실행 시간 오버헤드를 줄이고 성능을 향상시킬 수 있었다. Bai 등[5]은 네트워크 접속시에 나타날 수 있는 데이터 보안 문제를 검출할 수 있도록 동적 분석 방법을 기술하였다. 이 방법은 x86 가상 머신인 QEMU에 기반해서 만들어졌으며, 가상 머신에서 나타나는 데이터의 흐름을 기록하면서 프로그램의 제어 흐름을 바꿀 수 있는 프로그램의 취약점이 이용되고 있는지를 검출한다. Clause 등[6]은 위와 비슷한 동적 오염 분석 방법을 통해서 범용적으로 적용할 수 있는 프레임워크를 개발하였다. 이 분석 방법에서는 일반적으로 자료 흐름을 통해 전파되는 오염 정보 뿐만 아니라 제어 흐름을 통해 나타나는 오염 정보의 전파도 추적할 수 있다.

Yin 등[7]은 동적 오염 분석의 한 응용 사례로 악성 코드를 탐지하기 위해서 신뢰할 수 없는 오염 데이터의 흐름을 추적하고 이를 바탕으로 테인트 그래프를 생성하는 방법을 제시하였다. 이 방법은 기존의 분석들이 오염된 데이터가 전파되고 사용될 때 프로그램의 취약점을 공격할 수 있는지를 예측하는데 한정된 것에 반해서 테인트 그래프를 통해서 데이터 흐름이 사용자의 민감한 정보를 훔치거나 이용하려는 행동이 아닌지 분석하고 악성코드를 찾아내는데 이용하였다.

위와 같이 동적 오염 분석기의 개발과 관련된 연구 사례가 소개되고 있지만, 기존의 연구 사례에서는 분석 원리에 대해서 간단하게 언급하고 있을 뿐 오염된 정보의 전파 과정이나 분석 방법에 대한 상세한 절차는 기술하지 않고 있다. 본 논문에서는 동적 오염 분석의 이론적 모델을 기술함으로써 향후 여러 응용 과제에서 기반이 되는 이론으로 적용될 수 있을 것이다.

3. 동적 오염 분석의 모델링

3.1 언어의 정의

동적 오염 분석을 기술하기 위해서 이를 위한 기본 언어는 필수적인 프로그램 구문을 포함하는 명령형 언어(imperative language)인 WHILE[8]에 기초를 두었다. 그리고, 추가적으로 필요한 명령어를 보완해서 동적 오염 분석을 위한 필수적인 명령어 구문들을 포함하도록 구성하였다.

Table 1. The syntax of basic language

program ::= **program** declaration* **begin** statement* **end**	
declaration ::= **proc** identifier (identifier) **begin** statement* **end**	//proc. decl.
statement ::= **skip** |	
identifier = exp |	//assignment
if exp **then** statement* **else** statement* |	//if stmt
while exp **do** statement* |	//while stmt
call identifier (exp) |	//proc. call
identifier = **load** (exp) |	//load
store (exp, exp) |	//store
taintcheck (exp)	//taint check
exp ::= identifier |	
true | **false** |	//boolean
constant |	//constant
input (src) |	//input data
exp op_b exp |	//binary op.
op_u exp	//unary op.

Table 1에서 본 논문에서 사용하는 기본 언어의 구문(syntax)을 보여주고 있다. 전체 프로그램은 먼저 프로시저들이 정의된 후에 필요한 명령어 구문들이 나타난다. 프로시저 정의는 프로시저의 이름과 호출시 필요한 인자를 명시하고, 프로시저의 바디에는 필요한 명령어 구문들이 정의된다.

명령어 구문(statement)은 일반적인 명령형 언어에서 포함하는 구문인 값 할당(assignment), if 조건문, while 반복문, 프로시저 호출 등을 포함하고 있다. load 와 store 명령어는 메모리에 데이터를 직접 읽어 오거나 쓰는 명령어이다. 이 명령어는 프로그램의 동적 분석에서 메모리에 직접 접근하는 명령어들을 분석하고 표현하기 위해서 사용된다. taintcheck 명령어는 주어진 수식 또는 변수에 대해서 이 값이 오염(taint) 되었는지 아니면 신뢰할 수 있는지를 확인하는 명령어이다. 분석 결과를 통해서 명령어의 데이터 신뢰성 여부를 확인하고, 소프트웨어의 취약점을 공격할 수 있는 명령어를 방지할 수 있다.

표현식(exp)은 변수 또는 프로시저 등의 이름을 표현하는 identifier, true 또는 false 값을 가지는 불린값, 상수, 이항 연산, 그리고 단항 연산 등으로 구성된다. input 명령어는 프로그램 외부로부터 정보를 입력으로 받는 명령어이며, 이 명령어를 통해서 사용자나 네트워크 등의 외부 입력을 표현식으로 받아들일 수 있다.

3.2 연산 의미론 (Operational Semantics)

연산 의미론[9,10]은 프로그램 또는 시스템의 동작 절차를 정확하게 기술할 수 있는 방법을 제공한다. 연산 의미론은 프로그램의 동작을 이해하거나 분석하고 검증하기 위한 수학적인 모델로 사용된다. 따라서 프로그램의 동적 분석 과정을 논리적이고 체계적으로 표현하기 위한 방법으로 적당한 모델이다. 연산 의미론은 명령어의 실행 동작을 다음과 같은 규칙으로 표현한다.

$$\frac{computation\ result}{< S_1 > com_1 \Rightarrow < S_2 > com_2}$$

이 규칙은 이전 프로그램의 상태인 S_1에서 명령어 리스트 com_1을 실행할 때, computation result 와 같은 연산 결과가 나온다면 상태는 S_2로 전이되고, 명령어 리스트 com_2로 전이(transition) 된다는 것을 나타낸다. 여기서 상태는 프로그램에서 명령어를 실행함으로써 나타나는 변화를 논리적으로 기술하기 위해서 쓰이는데, 명령어 실행을 통한 메모리, 스택, 저장 공간 등의 현재 상태와 변화된 상태 등을 나타내게 된다. 그리고, 프로그램의 명령어 리스트는 전체 프로그램을 구성하는 각 명령어 구문의 리스트에서 실행중인 명령어들로 구성되며, 각 명령어는 리스트 연결 연산(concat, ::)으로 연결되어 있다.

3.3 동적 오염 분석의 모델링

1) 환경 변수의 정의

동적 오염 분석을 연산 의미론으로 모델링하기 위해서 명령어를 실행할 때 나타나는 상태의 변화를 표현하기 위한 상태 변수를 정의해야 한다. 불린 또는 상수 값의 집합 V, 변수의 집합 X, 메모리 주소의 집합 L, 프로시저의 집합 P에 대해서 상태를 표현하는 환경 변수는 다음과 같이 $< \sigma, M, \gamma >$의 쌍으로 정의 된다.

$< \sigma, M, \gamma >$	상태 (state)
$\sigma : X \rightarrow (V, \tau)$	각 변수에 대해서 결과값과 테인트 정보를 가짐
$M : L \rightarrow (V, \tau)$	메모리 주소에 대해서 저장된 값과 테인트 정보를 가짐
$\gamma : P \rightarrow (X, body, \gamma)$	프로시저의 형식인자, 바디, 현재 프로시저 정의 상태 정보를 가짐
$\tau ::= True \mid False$	테인트 정보 : 정보 오염 여부

프로그램의 상태는 σ, M, δ 의 세 개의 환경 변수들로 표현할 수 있다. 환경 변수 σ는 프로그램에서 사용된 변수들의 정보를 프로그램의 상태로 나타내는 역할을 한다. 프로그램에 사용된 변수에 대해서 결과값과 정보의 오염 여부를 나타내는 테인트 값을 유지하고 관리한다. 환경 변수 M은 프로그램의 실행 시간에 메모리의 상태와 변화 정보를 보여준다. 메모리 공간의 주소에 대해 저장된 데이터와 이 정보의 오염 여부를 나타내는 테인트 값을 표현한다. 환경 변수 γ는 프로시저 정의 및 호출문이 있을 때 상태의 변화를 반영한다. 프로그램에 정의된 프로시저의 이름에 대해서 프로시저의 인수명, 프로시저 본문(body), 그리고 그 프로시저가 가지는 현재의 프로시저 정의 상태를 표현하고, 정의된 프로시저가 호출될 때 이 환경 변수로부터 필요한 명령을 수행할 수 있도록 한다.

2) 연산의 추론 규칙

데이터의 오염(taint) 여부를 추적하는 동적 오염 분석은 실행되는 각 명령어에 대해서 명령어가 실행되기 전과 후의 상태가 어떻게 변화하고 그 변화에서 오염된 데이터의 전이 여부가 결정되어야 한다.

Table 2. Inference rules of expressions

$$\frac{v :: input\,data}{<\sigma, M, \gamma> input(src) \Downarrow <v, T>}\ \text{INPUT}$$

$$\frac{}{<\sigma, M, \gamma> v \Downarrow <v, F>}\ \text{CONST}$$

$$\frac{}{<\sigma, M, \gamma> var \Downarrow \sigma(var)}\ \text{VAR}$$

$$\frac{e \Downarrow <v, t>}{<\sigma, M, \gamma> op_u\, e \Rightarrow <op_u\, v, t>}\ \text{UNARY}$$

$$\frac{e_1 \Downarrow <v_1, t_1> \quad e_2 \Downarrow <v_2, t_2>}{<\sigma, M, \gamma> e_1\, op_b\, e_2 \Downarrow <v_1\, op_b\, v_2, t_1 \vee t_2>}\ \text{BINARY}$$

Table 2에서는 앞에서 정의된 언어에서 나타나는 표현식(expression)에 대한 계산 규칙을 보여주고 있다. 표현식의 추론 결과는 그 식의 결과값과 함께 그 값의 오염 여부를 나타내는 테인트 값이 쌍으로 나온다. INPUT 규칙은 외부 장치로부터 데이터의 입력을 처리하는 규칙을 보여준다. 외부로 부터의 입력은 키보드 등을 통한 사용자의 입력, 네트워크를 통한 데이터 입력 등 입력 장치에 따라서 구분될 수 있으며, 검증되지 않은 입력 데이터라고 판단한다. 그래서, input() 명령어를 통해서 검증되지 않은 소스로부터 받은 입력은 오염된 상태로 결과를 추론하고 입력 값에 대한 테인트 값은 True로 추론된다. 반면 CONST 규칙에서 연산에 포함된 값은 오염되지 않은 상수값이므로 테인트 값은 False로 추론된다. VAR 규칙은 표현식에 사용된 변수를 계산하는 규칙을 보여주며 이는 이전에 계산된 값의 상태를 관리하는 환경 변수 σ로부터 나타나는 결과 값으로 추론된다. UNARY 규칙과 BINARY 규칙은 각각 단항 연산자와 이항 연산자를 포함하는 표현식을 계산하는 과정을 보여주고 있다. 이 규칙은 각각의 연산에 대한 결과값을 구하고, 테인트 값은 각각의 항들이 가진 테인트 값의 논리합으로 추론된다.

3) 동적 오염 분석의 단계적 분석 과정

표현식의 추론 규칙을 이용해서 Table 3에서는 동적 오염 분석을 통해 프로그램의 실행 과정에 나타나는 상태의 변화를 연산 의미론으로 표현한 결과를 보여준다. 프로그램을 구성하는 명령어들은 리스트 연결 연산으로 표현되고 있으며, $com_1 :: com_2$는 현재 실행되는 명령어는 com_1이며 그 이후의 명령어 리스트는 com_2라는 것을 의미한다.

SKIP 규칙은 skip 명령어에 대한 전이 과정을 보여주며, 상태의 변화 없이 다음 명령어를 실행한다. ASSIGN 규칙은 변수에 표현식의 결과를 대입하는 명령어를 실행하는 전이 규칙을 보여준다. 대입하는 표현식의 계산 결과값과 테인트 값을 환경 변수 σ에 반영한다. 여기서, $\sigma[var \leftarrow < var, t >]$는 환경 변수 σ에서 변수 var의 값은 <var, t> 로 저장한다는 것을 의미한다. IFT와 IFF 규칙은 if 조건문에 대해서 각각 조건문이 참인 경우와 거짓인 경우의 전이 규칙을 보여준다. 조건문이 참인 경우에는 동일한 상태에 대해서 then 구문이 실행되고, 거짓인 경우에는 else 구문이 실행되도록 전이한다. WHILET와 WHILEF 규칙은 각각 while 반복문에서 조건문이 참인 경우와 거짓인 경우의 전이 규칙을 보여준다. 조건문이 거짓인 경우에는 반복문을 더 이상 실행

Table 3. A modeling of dynamic taint analysis using operational semantics

$$\frac{}{<\sigma, M, \gamma> skip :: C \Rightarrow <\sigma, M, \gamma> C}\ \text{SKIP}$$

$$\frac{e \Downarrow <v, t>}{<\sigma, M, \gamma> var = e :: C \Rightarrow <\sigma[var \leftarrow <v, t>], M, \gamma> C}\ \text{ASSIGN}$$

$$\frac{e \Downarrow <true, t>}{<\sigma, M, \gamma> \text{if } e \text{ then } e_1 \text{ else } e_2 :: C \Rightarrow <\sigma, M, \gamma> e_1 :: C}\ \text{IFT}$$

$$\frac{e \Downarrow <false, t>}{<\sigma, M, \gamma> \text{if } e \text{ then } e_1 \text{ else } e_2 :: C \Rightarrow <\sigma, M, \gamma> e_2 :: C}\ \text{IFF}$$

$$\frac{e \Downarrow <true, t>}{<\sigma, M, \gamma> while\ e\ do\ e_1 :: C \Rightarrow <\sigma, M, \gamma> e_1 :: while\ e\ do\ e_1 :: C}\ \text{WHILET}$$

$$\frac{e \Downarrow <false, t>}{<\sigma, M, \gamma> while\ e\ do\ e_1 :: C \Rightarrow <\sigma, M, \gamma> C}\ \text{WHILEF}$$

$$\frac{e_1 \Downarrow <v_1, t_1> \quad e_2 \Downarrow <v_2, t_2>}{<\sigma, M, \gamma> store\,(e_1, e_2) :: C \Rightarrow <\sigma, M[v_1 \leftarrow <v_2, t_1 \vee t_2>], \gamma> C}\ \text{STORE}$$

$$\frac{e \Downarrow <v_1, t_1> \quad M(v_1) = <v_2, t_2>}{<\sigma, M, \gamma> var = load\,(e) :: C \Rightarrow <\sigma[var \leftarrow <v_2, t_1 \vee t_2>], M, \gamma> C}\ \text{LOAD}$$

$$\frac{}{<\sigma, M, \gamma> proc\ P(x) = e :: C \Rightarrow <\sigma, M, \gamma[P \leftarrow pd(x, e, \gamma)]> C}\ \text{DEF}$$

$$\frac{e \Downarrow <v, t> \quad \gamma(P) = pd(x, c, \gamma_1)}{<\sigma, M, \gamma> call\ P(e) \Rightarrow <\sigma[y \leftarrow <v, t>], M, \gamma> c[x/y]}\ \text{(for } New\ variable\ y\text{) CALL}$$

$$\frac{e \Downarrow <v, t>}{<\sigma, M, \gamma> taintcheck\ e :: C \Rightarrow (return\ t) :: <\sigma, M, \gamma> C}\ \text{TCHECK}$$

하지 않기 때문에 while 반복문을 종료하고 다음 명령어로 이동한다. 반면, 조건문이 참인 경우에는 반복문을 계속 실행해야 하기 때문에 반복문의 바디를 1회 실행한 다음 다시 반복문을 실행하도록 명령어 실행 순서가 변화된 구문으로 전이되었다. 그러면 1회의 반복문 바디를 실행한 이후 그 때의 상태 변수에 대해서 다시 반복문을 반복하게 되며, 조건문이 거짓이 될 때까지 반복 실행된다.

STORE 규칙은 표현식의 계산 결과를 메모리 위치에 저장하는 store() 명령어의 전이 규칙을 보여준다. 대입문의 좌변과 우변의 표현식을 계산한 결과를 이용해서 좌변의 결과값을 주소로 하는 메모리에 우변의 결과값을 대입하게 되고, 이는 메모리 상태를 표현하는 환경 변수 M에 반영된다. 그리고, 테인트 값은 저장하는 메모리 주소가 오염되지 않고, 대입하는 결과 값 또한 오염되지 않았을 때만 오염되지 않은 것으로 판정할 수 있고 신뢰할 수 있기 때문에 두 테인트 값의 논리합으로 반영한다.

LOAD 규칙은 메모리 위치로부터 값을 읽어오는 load() 명령어에 대한 전이 규칙을 보여준다. 메모리의 주소 값을 가지는 표현식을 계산한 결과 값의 메모리 정보를 읽어오기 위해서 메모리의 상태를 가지고 있는 환경 변수 M으로부터 나온 결과 값을 읽어오게 된다. 이 값은 다시 지정된 변수에 대입하고 이는 변수에 대한 상태를 가지고 있는 환경 변수 σ에 반영된다. 테인트 값은 메모리 주소 값의 테인트 여부와 메모리에 저장된 값의 테인트 여부에 의해서 결정되기 때문에 두 값의 논리합으로 정해진다.

DEF 규칙은 프로그램 내에서 호출하는 프로시저의 정의를 다루는 규칙이다. 프로시저가 정의되면 프로시저의 이름, 인자의 이름, 프로시저의 바디 등이 프로시저의 정보를 유지하는 환경 변수 γ에 저장된다. CALL 규칙은 정의된 프로시저를 호출하는 call 명령문의 전이 규칙을 보여준다. 호출된 프로시저의 정의를 γ로부터 찾아내고, 프로시저의 바디를 실행한다. 이 때 프로시저의 형식 인수는 동일한 이름을 가진 변수가 존재할 수 있기 때문에 새로운 이름의 변수 y를 생성해서 환경 변수 γ_2에서 수정 반영한다. 실인수 y의 값은 표현식 e를 계산한 결과 값으로 환경 변수 σ를 변경하고 프로시저의 바디를 실행한다. 여기서 $c[x/y]$는 수식 c에서 사용되는 변수 x를 변수 y로 바꾼 수식을 의미한다.

TCHECK 규칙은 수식에 대해서 그 식의 오염 여부를 확인하는 명령어 taintcheck에 대한 추론 규칙을 보여준다. 이 명령어의 결과는 표현식의 추론 결과가 가지는 테인트값으로 반환된다. 이 규칙을 통해서 명령어가 실행될 때 신뢰할 수 없는 오염된 정보가 포함되어 있지 않은지 분석 결과를 확인할 수 있다.

동적 오염 분석에서 정보의 테인트 값은 그 정보가 외부의 검증되지 않은 소스로부터 유래했을 때 오염된 것으로 분석한다. Table 3의 분석 결과로부터 오염된 정보는 프로그램 실행 중에 ASSIGN, STORE, LOAD, CALL 규칙을 통해서 다양한 경로로 여러 데이터에 전파될 수 있다는 것을 알 수 있다. 테인트의 전파 과정에서 오염되지 않은 데이터가 오염되기도 하고, 오염되었던 데이터가 정화되기도 한다. 오염되었던 데이터가 정화되는 경우는 여러 전파 경로 중에서 대입문에 의해서 오염된 변수에 오염되지 않은 값을 저장하는 경우에 나타날 수 있다.

4. 동적 오염 분석 모델의 구현 및 평가

4.1 동적 오염 분석 모델의 구현

본 절에서는 3절에서 제시한 동적 오염 분석의 모델이 정확한 결과가 나오는지 검증하기 위해서 동적 오염 분석기를 구현하고 평가한다. 구현 환경은 윈도우 XP 운영체제에서 구현하였으며, 함수형 언어인 하스켈 (Haskell) [11, 12] 언어를 이용하였다. 하스켈은 엄격한 타입 시스템 (strong type system)을 가지고 문자열을 다루는 고급 기능을 지원하기 때문에 프로그래밍 언어의 타입 추론, 해석 및 데이터 분석기 개발 등에 효과적으로 사용되는 언어이다. 본 논문에서 구현한 분석기는 Table 1에서 기술한 기본 언어로 작성된 프로그램을 입력으로 받고 3절에서 기술한 분석 모델을 기반으로 동적 오염 분석을 수행하도록 구현하였다.

Table 4. An example program

```
(1)   program begin
(2)       proc Fn(x) begin
(3)           z = load(x)
(4)           taintcheck(z)
(5)       end
(6)       a=input(network)  // taint input
(7)       b=10
(8)       if (a > b)
(9)           then c = a+b
(10)          else c = a-b
(11)      store(b, c)
(12)      taintcheck(b)
(13)      pbuf = 5000          // start address of buffer
(14)      length = 5           // size of buffer
(15)      px = pbuf+length     // buffer overflow
(16)      store(px, b)         // init value of px location
(17)      taintcheck(px)       // taint anal. of px location
(18)      index = 0
(19)      while (index <= length)
(20)          c = c + b
(21)          store ((pbuf+index), a)
(22)          index = index + 1
(23)      taintcheck(px)   // check after buffer overflow
(24)      call Fn(b)
(25)  end
```

Table 4는 본 분석기를 통해서 실험에 사용한 예제 프로그램을 보여주고 있다. 예제 프로그램은 본 논문에서 제시한 분석 규칙이 올바른지 검증하기 위해서 Table 2와 Table 3에서 기술한 분석 모델의 모든 분석 규칙이 실행 되도록 구성하였다. (2)~(5)에서 하나의 프로시저가 정의되는데, 이 프로시저는 인수의 값을 주소로 하는 메모리의 값을 읽어오는 작업을 한다. (6)~(10)에서 네트워크를 통해서 입력받은 오염된 정보 a와 상수 값을 가지는 변수 b를 더해서 변수 c

에 대입한다. (11)에서 b가 가리키는 메모리 주소에 저장한 후, (24)에서 프로시저를 호출한다. 그리고, 소프트웨어에서 흔히 나타나는 민감한 오류 중에 하나인 버퍼 오버플로가 나는 경우를 포함하고 있다. (13), (14)에서 pbuf는 5000번지에서 시작하고 크기가 5인 데이터로 정의하였으며, (15)에서 px는 pbuff 바로 다음에 위치하도록 정의하고 있다. (19)~(22)에서 반복문으로 버퍼 pbuf의 값을 오염된 변수 a의 값으로 초기화하며, 반복문의 반복 회수가 버퍼의 크기를 넘어가면서 버퍼 오버플로가 발생한다. (17)과 (23)에서는 오버플로가 나기 이전과 이후에 px의 테인트 정보를 각각 분석하고 있다.

Table 5. Execution results of dynamic taint analysis

```
---- TAINT Analysis Results ----
"Eid \"b\" : VConst 10 | Taint False"
"Eid \"px\" [Addr 5005] : VConst 10 | Taint False"
"Eid \"px\" [Addr 5005] : VConst 50 | Taint True"
"Eid \"z\" : VConst 60 | Taint True"

  [SIGMA] :
("z",(VConst 60,Taint True))
("Y001",(VConst 10,Taint False))
("index",(VConst 6,Taint False))
("px",(VConst 5005,Taint False))
("length",(VConst 5,Taint False))
("pbuf",(VConst 5000,Taint False))
("c",(VConst 120,Taint True))
("b",(VConst 10,Taint False))
("a",(VConst 50,Taint True))

  [MEMORY] :
(Addr 5004,(VConst 50,Taint True))
(Addr 5003,(VConst 50,Taint True))
(Addr 5002,(VConst 50,Taint True))
(Addr 5001,(VConst 50,Taint True))
(Addr 5000,(VConst 50,Taint True))
(Addr 5005,(VConst 50,Taint True))
(Addr 10,(VConst 60,Taint True))

  [GAMMA] :
("Fn",("x",[SLoad "z" (Eid "x"),STaintcheck (Eid "z")]))
```

Table 5는 구현된 분석기를 이용해서 예제 프로그램을 분석한 결과를 보여주고 있다. 수행 결과는 taintcheck 명령이 수행된 변수 및 메모리 위치에 대한 테인트 분석 결과를 출력하고 있으며, 프로그램의 분석을 수행한 후에 나타나는 환경 변수의 상태를 보여주고 있다. 분석한 변수 b와 z의 테인트 정보각각 False와 True로 분석되었다. 메모리 주소 5005 번지를 가리키는 px에 대한 분석 결과에서 초기값으로 테인트 값은 False 이다. 하지만, 버퍼 오버플로가 발생한 이후의 두 번째 분석 결과에서는 테인트 값은 True로 분석되었고, 본 분석 결과를 통해 버퍼 오버플로가 발생함을 확인하였다.

[SIGMA]는 프로그램 종료 후 변수의 값과 테인트 정보를 보여준다. 변수 z는 상수 60의 값을 가지고, 테인트 값은 True인 것을 확인할 수 있다. 반복문은 총 6회의 반복을 하고 사용된 변수 index는 6의 값을 가지고 있다. [MEMORY]는 프로그램 실행 후 메모리의 상태를 보여주고 있으며, 메모리 10번지에 정수 60을 값으로 가지고 테인트 값은 True이다. 시작 주소 pbuf와 크기가 5인 버퍼는 메모리 주소 5000에서 5004번지에 해당하며, 상수 값 50과 테인트 값 True로 분석되었으며, 버퍼 오버플로에 의해서 5005번지까지 같은 값으로 분석되었다. [GAMMA]는 프로그램에서 정의된 프로시저의 정의를 보여주고 있으며 함수 Fn 의 형식 인자와 프로시저 바디를 보여준다. 분석 결과는 프로그램의 각 단계에서 데이터의 오염 여부를 분석할 수 있으며, 오염 정보의 전파 과정을 분석할 수 있다.

4.2 실행 결과의 평가 및 검증

본 절에서는 4.1절에서 제시한 분석기의 실행 결과에 대해서 이론적인 모델링을 통해 올바르게 동작하는지를 검증한다. Table 6과 Table 7은 Table 4에서 제시한 예제 프로그램을 실행할 때 동적 오염 분석을 통해서 나타나는 환경 변수의 변화 및 자료의 오염이 전파되는 과정을 보여주고 있다. 이 과정은 Table 2와 Table 3에서 제시한 이론적 모델로부터 동일하게 유도된다.

Table 6. The change of environment variables and propagation of taint information during execution of the example program (before the call of procedure)

	σ	M	γ
(1)			
(2)~(5)			Fn←pd(x,body,{})
(6)	a←<50,T>		
(7)	b←<10,F>		
(8)~(10)	c←<60,T>		
(11)		10←<60,T>	
(12)	taintcheck(b) : F		
(13)	pbuf←<5000,F>		
(14)	length←<5,F>		
(15)	px←<5005,F>		
(16)		5005←<10,F>	
(17)	taintcheck(px) : F		
(18)	index←<0,F>		
(19)~(22)	c←<120,T> index←<6,F>	5000←<50,T> 5001←<50,T> 5002←<50,T> 5003←<50,T> 5004←<50,T> 5005←<50,T>	
(23)	taintcheck(px) : T		

Table 6에서는 프로시저 호출이 일어나기 전까지의 과정을 보여준다. (2)~(5)에서 프로시저가 정의되면 그 내용이 γ에 저장된다. (6)에서 외부의 네트워크로부터 값을 읽어 오는데 이 값이 50이라고 할 때, 이는 신뢰할 수 없는 오염된

데이터로 추론되고 <50,T> 값이 σ에 저장된다. (7)에서는 변수 b에 상수값을 대입하므로 오염되지 않은 값이기 때문에 <10,F>로 추론된다. (8)~(10)에서는 변수 a와 b의 값을 비교한 결과는 참이 되고, a와 b를 더한 값을 변수 c에 대입한다. 이때, 테인트 정보는 두 변수의 테인트 값의 논리합으로 분석되므로 <60,T>로 추론된다. (11)에서는 변수 b가 가리키는 메모리 주소 10번지에 변수 c의 데이터를 저장한 결과이다. (12)에서는 변수 b에 전파된 오염 정보를 분석한 결과를 출력하고 있으며, 오염되지 않았음을 확인할 수 있다. (13)~(15)에서는 버퍼의 시작 주소, 크기, 그리고 버퍼 다음에 위치하는 변수 px에 대한 초기값 설정 결과를 보여준다. (16), (17)에서는 버퍼 오버플로가 발생하기 전 px가 가리키는 메모리 주소에 오염되지 않은 변수 b를 저장한 후 테인트 분석 결과를 확인하고 있다. (18)에서는 반복문의 index를 0으로 초기화하고 있으며, (19)~(22)에서는 while 반복문의 실행 후 변수 및 메모리에 저장된 값을 보여주고 있다. 반복문은 총 6회 반복하며 메모리 주소 5000번지에서 5005번지까지 오염된 변수 a로 값을 변경하였다. (23)에서는 버퍼 오버플로가 발생한 후 px가 가리키는 메모리의 값이 오염된 것을 확인할 수 있다.

Table 7. The change of environment variables and propagation of taint information during execution of the example program (after the call of procedure)

	σ	M	γ
(24)	y←<10,F>	10←<60,T>	Fn←pd(x,body,{})[x/y]
(3)	z←<60,T>		
(4)	taintcheck(z) : T		

Table 7은 프로시저 호출 이후의 실행 분석 과정을 보여준다. (24)에서는 프로시저가 호출되고 실행되기 전의 초기 상태를 보여준다. 호출된 프로시저가 실행되기 이전에 프로시저가 실행될 수 있는 환경 변수의 초기값이 설정되는데, 실인수 b는 새로운 형식인수 y가 만들어지고 y로 치환되서 그 값이 σ에 저장되고 메모리의 상태를 가져오게 된다. 프로시저의 본문으로 들어가서 (3)에서는 load 명령어로 메모리로부터 지정된 주소의 값을 가져오게 되고, 이 값은 변수 z에 전파된다. (4)에서는 변수 z에 대한 오염 정보를 분석하고 오염 정보가 전파되었음을 확인할 수 있다.

본 분석 과정을 통해서 동적 오염 분석을 통해서 오염된 데이터가 흘러가고 이를 동적 오염 분석을 통해서 기술할 수 있다는 것을 확인할 수 있다. 처음 네트워크를 통해서 a에 저장된 오염된 정보는 다양한 경로로 전파되고 있으며, 버퍼 오버플로가 발생함으로써 변수 px가 가리키는 메모리의 값이 오염된 정보로 변경됨을 확인하였다. 또한, 호출된 프로시저에서 로드된 변수 z의 값으로 전파되고 있다는 것을 알 수 있다.

프로그램에 대한 단계적 분석 결과는 Table 5에서 보여주는 분석기의 최종 분석 결과와 일치하고, 분석기를 통한 분석 결과가 정확함을 확인할 수 있다. 본 논문에서 기술한 동적 오염 분석의 모델은 오염된 정보의 전파 과정을 정확하게 분석할 수 있었고, 분석기의 구현을 통해서 그 결과를 검증하였다.

5. 결 론

컴퓨터와 인터넷 환경이 광범위하게 보급되면서 소프트웨어 보안에 관한 관심이 점점 커지고 있다. 동적 오염 분석은 오염된 정보의 전파를 추적하고 허가되지 않은 방법으로 사용하는 것을 방지함으로써 소프트웨어의 보안 강화를 위해서 활용되고 있다. 본 논문에서는 동적 오염 분석에 대한 이론적 고찰을 통해서 동적 오염 분석에서 오염된 정보를 추적하는 과정을 보여주고, 분석 과정을 이해할 수 있도록 논리적으로 기술하고 있다. 소프트웨어의 동적인 움직임을 기술하기 위해서 연산 의미론을 이용하였으며, 여기에 자료의 오염 여부 및 오염 자료의 전파 과정을 논리적으로 기술하고 있다. 본 논문에서 제시한 모델을 검증하기 위해서 분석기를 구현하였고, 분석된 결과를 통해서 오염 정보의 전파 과정 및 분석 결과가 정확하게 출력됨을 확인할 수 있었다. 본 논문의 결과는 향후 동적 오염 분석을 이용한 다양한 분석기의 설계 및 프로그램의 분석을 위한 응용 기술로 활용될 수 있을 것이라 기대된다.

참 고 문 헌

[1] Marco Pistoia, Satish Chandra, Stephen J. Fink, and Eran Yahav, A survey of static analysis methods for identifying security vulnerabilities in software systems, IBM Systems Journal, Vol.46, No.2, pp.265-288, 2007.

[2] Manuel Egele, Theodoor Scholte, Engin Kirda, Christopher Kruegel, A Survey on Automated Dynamic Malware Analysis Techniques and Tools, ACM Computing Surveys, Vol.44, No.2, February, 2012.

[3] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proceedings of the Network and Distributed System Security Symposium, 2005.

[4] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige, TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting, In Proceedings of 11th IEEE Symposium on Computers and Communications, pp.749-754, June, 2006.

[5] James Clause, Wanchun Li, and Ro Orso, Dytan: A Generic Dynamic Taint Analysis Framework, In Proceedings of the International Symposium on Software Testing and Analysis, pp.196-206, 2007.

[6] Zhiwen Bai, Liming Wang, Jinglin Chen, Lin Xu, Jian Liu, and Xiyang Liu, DTAD: A Dynamic Taint Analysis Detector for Information Security, In Proceedings of The Ninth

International Conference on Web-Age Information Management, pp.591-597, July, 2008.

[7] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda, Panorama: capturing system-wide information flow for malware detection and analysis, In Proceedings of the 14th ACM conference on Computer and communications security, pp.116-127, 2007.

[8] Syntax of the WHILE Language, http://pag.cs.uni-sb.de/while.html

[9] Kenneth Slonneger and Barry L. Kurtz, Formal Syntax and Semantics of Programming Languages, Addison Wesley, 1995.

[10] Sanjiva Prasad and S. Arun-Kumar, An Introduction to Operational Semantics, In the Compiler Design Handbook: Optimizations and Machine Code Generation, pp.841-890, CRC Press, 2002.

[11] The Haskell Programming Language, http://www.haskell.org/.

[12] Mark P. Jones, Typing Haskell in Haskell, In the Proceedings of Haskell Workshop, January, 1999.

임 현 일

e-mail : hilim@kyungnam.ac.kr

1995년 KAIST 전산학과(학사)

1997년 KAIST 전산학과(석사)

2009년 KAIST 전산학과(공학박사)

2010년~현 재 경남대학교 컴퓨터공학과 조교수

관심분야 : 소프트웨어 분석, 소프트웨어 보안, 소프트웨어 저작권 보호 및 도용 탐지 등