가상화를 이용하여 난독화된 바이너리의 제어 흐름 재건

Control Flow Reconstruction from Virtualization-Obfuscated Binaries

저자
(Authors)

황준형, 한태숙
Joonhyung Hwang, Taisook Han

출처
(Source)

정보과학회논문지 42(1), 2015.01, 44-53 (10 pages)
Journal of KIISE 42(1), 2015.01, 44-53 (10 pages)

발행처
(Publisher)

한국정보과학회
KOREA INFORMATION SCIENCE SOCIETY

URL

http://www.dbpia.co.kr/Article/NODE06071179

APA Style

황준형, 한태숙 (2015). 가상화를 이용하여 난독화된 바이너리의 제어 흐름 재건. 정보과학회논문지, 42(1), 44-53.

ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 42, No. 1, pp. 44-53, 2015. 1
http://dx.doi.org/10.5626/JOK.2015.42.1.44

가상화를 이용하여 난독화된 바이너리의 제어 흐름 재건

(Control Flow Reconstruction from Virtualization-Obfuscated Binaries)

황 준 형 †　　한 태 숙 ††
(Joonhyung Hwang)　(Taisook Han)

요 약 제어 흐름 정보는 프로그램이 실행되는 구조를 담고 있어 소프트웨어를 분석할 때 기준이 되고 소프트웨어를 서로 비교할 때에도 유용하게 쓰인다. 가상화를 이용한 난독화는 실제 기계의 명령을 구조가 숨겨진 가상 기계의 명령으로 바꾸어 프로그램의 제어 흐름 정보를 감춘다. 난독화가 적용된 바이너리에서는 가상 기계의 명령을 실행하는 인터프리터의 구조만 직접 드러난다. 이 논문에서는 가상화로 난독화된 바이너리를 실행해서 수행되는 명령들을 기록한 트레이스를 이용해 숨겨져 있는 프로그램의 본질적인 제어 흐름을 다시 만들어내는 방법을 제안한다. 트레이스를 기계 명령으로 이루어진 문자열로 보고 생성되는 트레이스들을 모두 받아들일 수 있는 오토마톤을 찾은 다음, 해당되는 제어 흐름 그래프를 만든다. 기계 명령의 수행은 오토마톤의 상태 전이에 대응하며, 이는 제어 흐름 그래프의 간선에 대응한다. 제안한 방법을 상용 가상화 도구로 난독화된 바이너리에 적용해 보았으며, 원본 바이너리와 유사한 제어 흐름 그래프가 생성되는 것을 확인하였다.

키워드: 가상화를 이용한 난독화, 제어 흐름 재건, 난독화 해제, 동적 분석

Abstract Control flow information is useful in the analysis and comparison of programs. Virtualization-obfuscation hides control structures of the original program by transforming machine instructions into bytecode. Direct examination of the resulting binary reveals only the structure of the interpreter. Recovery of the original instructions requires knowledge of the virtual machine architecture, which is randomly generated and hidden. In this paper, we propose a method to reconstruct original control flow using only traces generated from the obfuscated binary. We consider traces as strings and find an automaton that represents the strings. State transitions in the automaton correspond to the control transfers in the original program. We have shown the effectiveness of our method with commercial obfuscators.

Keywords: virtualization-obfuscation, control flow reconstruction, deobfuscation, dynamic analysis

· 본 연구는 2013년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF-2011-0023847)
· 이 논문은 2014 한국컴퓨터종합학술대회에서 '가상화를 이용하여 난독화된 바이너리의 제어 흐름 재건'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 한국과학기술원 전산학과(KAIST)
envia@envia.pe.kr
(Corresponding author임)
†† 종신회원 : 한국과학기술원 전산학과 교수
han@cs.kaist.ac.kr

논문접수 : 2014년 7월 28일
(Received 28 July 2014)
논문수정 : 2014년 10월 27일
(Revised 27 October 2014)
심사완료 : 2014년 11월 7일
(Accepted 7 November 2014)

정보과학회논문지 제42권 제1호(2015. 1)

1. 서 론

제어 흐름은 프로그램의 명령이 실행되는 순서를 뜻한다. 제어 흐름 정보는 프로그램 분석을 수행할 때 유용하게 쓰인다. 제어 흐름 정보를 이용하면 어떠한 명령이 어떠한 명령에 영향을 미치는지 알 수 있으며, 프로그램의 구조를 파악하고 프로그램 분석의 기준을 설정할 수 있다. 이를 통해 프로그램이 어떠한 일을 하며 어떤 특성을 만족하는지 알아낼 수 있다. 제어 흐름 정보가 없거나 부족하면 정확하고 정밀한 분석을 수행하기 어렵다.

제어 흐름 정보를 바탕으로 프로그램의 비교를 수행할 수도 있다[1]. 프로그램이 변형돼도 일관되게 추출되는 제어 흐름 정보가 있다면, 이를 바탕으로 프로그램이 유사한지 여부를 판단할 수 있다. 이는 프로그램의 도용 탐지나 악성 코드의 분류 작업에 이용할 수 있다.

난독화(obfuscation)는 프로그램의 구조를 파악하기 어렵게 만드는 기법이다. 프로그램의 핵심 알고리듬과 같은 부분을 제삼자가 알기 어렵게 하기 위해 쓰인다. 프로그램을 변경하기 위해서는 먼저 프로그램을 이해해야 하므로 난독화를 적용하면 프로그램에 대한 원하지 않는 변경을 막을 수도 있다. 난독화는 프로그램을 같은 일을 하지만 이해하기 어려운 프로그램으로 변경하는 방법으로 이루어진다.

가상화(virtualization)는 프로그램을 가상의 컴퓨터에서 실행하는 것과 같이 실행하는 기법인데, 난독화에도 적용할 수 있다. 프로그램의 명령을 가상 기계의 명령으로 전환하고 인터프리터를 통해 실행하면 원본의 명령을 노출하는 일을 막을 수 있다. 난독화가 적용된 프로그램을 분석하면 인터프리터의 구조만 직접적으로 나타나기 때문에 간단한 분석으로는 프로그램의 의미를 파악하기 어렵다.

가상화를 이용한 난독화는 악성 프로그램도 사용할 수 있으므로 이에 대한 분석 기술이 필요하다. 난독화된 바이너리로부터 핵심 특성을 추출할 수 있으면 이를 바탕으로 바이너리 분석을 효율적으로 수행하여 프로그램의 의도를 파악할 수 있을 것이다.

기존 연구[2,3]에서는 주로 가상 기계의 구조를 분석한 후에 이를 바탕으로 제어 흐름 그래프(control flow graph, CFG)를 만들어냈다. 하지만 가상 기계의 구조를 분석하는 데 많은 노력이 필요하며, 새로운 구조의 가상 기계가 나타나면 새로운 방법을 개발해야 하는 어려움이 있다. 다양한 가상 기계를 사용하여 프로그램 변경을 어렵게 하는 연구가 실제로 수행되었다[4].

본 연구에서는 가상화로 난독화된 프로그램의 제어 흐름을 재건하기 위해 프로그램의 실행 트레이스를 문자열로 보고, 문자열에 해당하는 유한 상태 오토마톤을 찾아내며, 이를 바탕으로 제어 흐름 그래프를 만들었다.

제안하는 방법이 효과적인지 확인하기 위해 상용 가상화 난독화 도구인 VMProtect[5]와 Code Virtualizer[6]를 적용한 바이너리를 대상으로 실험을 수행하였다. 실험 결과 난독화된 분기(branch)와 반복(loop)을 성공적으로 탐지할 수 있었다. 본 논문에서 제안하는 기법은 구조 파악 없이 제어 흐름 그래프를 만드는 작업을 수행할 수 있으므로 기존 방법보다 적은 비용으로 빠르게 제어 흐름 정보를 알아내어 분석에 이용할 수 있을 것으로 기대한다.

논문의 구성은 다음과 같다. 2장에서는 가상화를 이용한 난독화 기법을 소개한다. 3장에서는 트레이스를 이용한 분석과 관련된 용어들을 정의한다. 4장에서는 제어 흐름 재건 방법을 소개하며, 5장에서는 제어 흐름 그래프를 정리할 때 사용하는 휴리스틱을 소개한다. 6장에서는 예제 프로그램을 대상으로 한 실험 결과를 소개한다. 7장에서는 관련 연구를 소개하며, 8장에서는 결론을 내리면서 앞으로 연구할 과제들을 검토한다.

2. 가상화를 이용한 난독화

가상화는 원본 프로그램의 기계 명령들을 바이트코드로 변환한다. 난독화를 위해 가상화를 사용할 때에는 바이트코드를 보는 것만으로는 프로그램의 의미를 알 수 없도록 임의로 생성된 가상 기계를 사용한다. 원본 바이너리의 기계 명령들은 변환되어 사라지고 바이트코드의 의미는 숨겨져 있기 때문에 바이너리의 의미를 분석하기 어려워진다.

가상화를 이용하여 난독화를 수행하면 원본 프로그램의 구조가 사라지게 된다. 그림 1은 가상화를 이용한 난독화를 적용한 사례를 나타내었다. 인터프리터에 해당하는 부분이 I이고 A, B, C, D에 해당하는 핸들러가 각각 A_H, B_H, C_H, D_H이다. 원래 프로그램을 분석하면 B, A, C로 구성된 반복 구조를 파악할 수 있지만, 난독화된 프로그램에서는 인터프리터의 구조만 드러난다. 인터프리터의 구조만 보고 프로그램의 의미를 알아내는 것은 쉽지 않다.

가상화를 이용하여 난독화를 수행하면 도메인 평탄화(domain flattening)도 일어나 여러 프로그램 위치가 하나로 인식된다[7]. 예로서 그림 1의 원래 프로그램에서는 A가 2개 있는 것을 구분할 수 있지만, 난독화된 프로그램에서는 둘을 구분할 수 없다. 이러한 상황에서 프로그램 분석을 수행하면 분석의 정확도가 떨어지게 된다.

가상화를 이용하여 난독화를 수행하면 원하지 않는 프로그램의 변경을 막을 수 있으며, 프로그램의 기밀을 보호할 수 있다. 하지만 악성 코드가 가상화를 이용하면

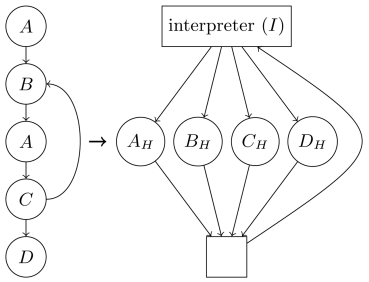

그림 1 가상화 난독화 전후의 제어 흐름 그래프

Fig. 1 Control flow graphs before and after virtualization-obfuscation

문제가 되며, 소스 코드 도용을 숨기기 위해 가상화를 사용할 수도 있다. 일부 보안 프로그램은 난독화된 프로그램 자체를 악성 코드로 분류하기도 하지만, 난독화된 프로그램에 대한 분석 기법을 개발하는 것이 더욱 바람직한 접근일 것이다.

3. 트레이스를 이용한 분석

본 논문에서는 난독화된 바이너리를 분석하기 위해 트레이스를 이용한 동적 분석 방법을 사용한다. 실제 프로그램 실행 없이 바이너리의 정보를 이용하여 분석을 수행하는 정적 분석 방법은 모든 실행 가능성을 다루어 안전성을 보장할 수 있다는 장점이 있지만, 실제로 일어나지 않을 수 있는 상황까지 포함하기 때문에 정확도가 떨어진다. 동적 분석 방법은 모든 가능성을 다루지는 못한다는 한계가 있지만, 실제로 실행된 결과를 바탕으로 분석을 수행하기 때문에 구체적이고 현실적인 분석을 수행할 수 있다. 난독화된 바이너리에는 정적 분석을 어렵게 하는 기법이 다양하게 적용되어 있으므로 동적 분석 방법을 사용해서 실용적인 분석 결과를 얻는다.

동적 분석 방법을 사용할 때에는 실제로 실행한 부분만 반영이 되므로 프로그램의 핵심을 실행하는 것이 중요하다. 여기서는 바이너리를 분석하는 사람이 프로그램의 핵심을 실행한 트레이스를 얻어낼 수 있다고 가정하였다. 난독화된 프로그램은 구조가 숨겨져 있으므로 핵심에 해당하는 부분을 직접 선택하는 것은 쉽지 않지만, 프로그램의 실행 결과를 관찰하면 핵심에 해당하는 부분이 실행되었는지 여부는 판단할 수 있다고 생각하였다. 예를 들어 난독화된 바이너리에서 인증에 해당하는 코드가 어디에 있는 파악할 수 없더라도 실행 결과 실제로 인증이 이루어졌다면 인증에 해당하는 코드가 실행되었다는 것은 알 수 있다. 이를 바탕으로 트레이스를 분석하면 인증에 해당하는 부분이 어디인지도 파악할 수 있을 것이다. 일부 악성 코드는 실행 환경을 관찰하면서 자신의 핵심 기능을 숨기는데, 이를 극복하는 방법에 대한 연구도 진행되고 있다[8].

구체적인 분석을 다루기 전에 정적 분석 논문들[9,10]이 사용하는 정의를 응용하여 트레이스를 정의한다. 먼저 메모리를 주소와 값 사이의 함수로 정의한다. 주소와 값은 모두 자연수 집합 $\mathbb{N}$의 원소인데, 프로그램 실행 과정에서 다른 의미로 해석될 수 있다. 상태 $\sigma\in\Sigma$는 메모리 상태 $m\in\mathbb{N}\to\mathbb{N}_\perp$과 프로그램 카운터 주소 $a\in\mathbb{N}$의 순서쌍인 (m,a)로 정의한다. 프로그램 P는 프로그램 초기 상태들의 집합으로 정의하는데, 모든 입력은 이미 메모리에 들어있다고 가정한다. 기계 명령 $\iota\in\mathbb{I}$는 메모리에 자연수 형태로 저장되었다가 실행되는데, 함수 encode$\in\mathbb{I}\to\mathbb{N}$는 명령에 해당하는 자연수를 계산하고, 함수 decode$\in\mathbb{N}\to\mathbb{I}_\perp$는 자연수를 명령으로 풀어낸다. 기계 명령은 시스템 상태를 받아서 다음 상태를 주는 함수로 정의되는데, 함수 $[\![\ ]\!]\in\mathbb{I}\to\Sigma\to\Sigma_\perp$는 기계 명령의 의미를 정의한다.

상태 트레이스 $\tau\in\Sigma^*$는 프로그램이 실행되면서 나타나는 상태들의 열(sequence)로 정의하며, 명령 트레이스 $\hat{\tau}\in\mathbb{I}^*$는 프로그램이 실행하는 명령들의 열로 정의한다. 상태 트레이스 $(m_1,a_1)\ldots(m_{n-1},a_{n-1})(m_n,a_n)$ 에서는 $(m_1,a_1)\in P$가 성립하고, 모든 자연수 i에 대해 $(m_{i+1},a_{i+1})=[\![\text{decode}(m_i(a_i))]\!](m_i,a_i)$가 성립한다. 또한 상태 트레이스 $(m_1,a_1)\ldots(m_n,a_n)$ 에 대응하는 명령 트레이스가 $\iota_1\ldots\iota_n$ 이라면 모든 i에 대해 $\iota_i=\text{decode}(m_i(a_i))$가 성립한다.

가능한 모든 상태 트레이스의 집합은 T로, 가능한 모든 명령 트레이스의 집합은 $\hat{T}$ 또는 $\hat{T}^\natural$ 로 나타내며, 실제 수행을 바탕으로 생성한 명령 트레이스들의 집합은 $\hat{T}^\flat$ 으로 나타낸다. 이때 $\hat{T}^\flat$ 은 $\hat{T}$ 의 부분집합이다. 모든 가능한 명령 트레이스에 해당하는 제어 흐름 그래프는 $\hat{T}^\#$ 로 나타내는데, 노드 집합과 간선 집합의 순서쌍 (N,E)로 정의한다. 여기서 N은 $\mathbb{I}^*$의 원소들과 시작 노드 n_i, 종료 노드 n_f를 원소로 한다. 또한 E는 $N\times N$의 부분집합이며 $\hat{T}$의 모든 원소 $\hat{\tau}$에 대해 $\hat{\tau}=n_1n_2\ldots n_{n-1}n_n$ 을 만족하는 (n_i,n_1), (n_1,n_2), …, (n_{n-1},n_n), (n_n,n_f)가 E의 원소이다.

이렇게 정의하면 트레이스를 이용한 제어 흐름 재건은 실행을 관찰하거나 인스트루멘테이션 도구를 이용하여 $\hat{T}^\flat$ 을 생성하고 이로부터 $\hat{T}^\#$ 를 계산하는 과정으로 볼 수 있다. 여러 휴리스틱을 이용하여 이 작업을 수행하는데, 자세한 휴리스틱은 5장에서 다룬다.

4. 제어 흐름 재건

트레이스를 이용하여 제어 흐름을 다시 만들기 위해 먼저 트레이스에서 휴리스틱을 통해 분기, 반복과 같은

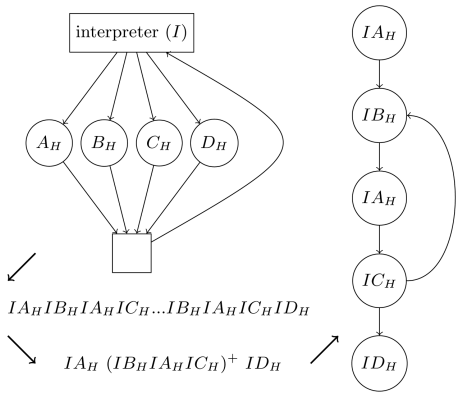

그림 2 제어 흐름 재건 방법
Fig. 2 Control flow reconstruction

구조를 찾아내고 이를 반영하는 오토마톤을 만든다. 트레이스에서 명령 A 다음에 B가 실행되면 정규 표현식 AB에 해당하는 오토마톤을 만들고, A와 B로 분기되면 $(A|B)$에, A가 반복되면 A^+에 해당하는 오토마톤을 만든다. 트레이스를 문자열로 보고 기계 명령을 알파벳으로 보면 만들어진 오토마톤은 트레이스를 받아들이게 된다.

오토마톤을 생성한 다음에는 이를 바탕으로 제어 흐름 그래프를 만든다. 오토마톤의 상태 전이가 제어 흐름 그래프의 노드가 되며, 오토마톤의 상태는 제어 흐름 그래프의 간선이 된다. 상태 전이에 해당하는 기계 명령이 그래프 노드에 해당하는 기계 명령이 된다.

그림 2는 제안하는 해결 방법을 단순화시켜 나타낸 것이다. 원래 프로그램에서 A를 실행한 다음 B, A, C를 반복하고 D를 실행하면서 종료한다면, 난독화된 프로그램에서도 A에 해당하는 부분인 A_H를 실행한 다음 B_H, A_H, C_H 를 반복하고 D_H를 실행한 다음 종료할 것이다. 핸들러 실행 전에 인터프리터 I를 실행한다면 트레이스에는 $IB_HIA_HIC_H$ 가 반복될 것이다. 반복이 있다는 것을 이용해서 트레이스를 포함하는 언어에 해당하는 정규 문법 $IA_H(IB_HIA_HIC_H)^+ID_H$ 를 찾아내고, 이에 해당하는 오토마톤을 얻은 다음, 이를 바탕으로 제어 흐름 그래프를 만든다.

실제로 제어 흐름 재건 과정을 구현할 때에는 각 단계를 따로 구현하지 않고, 한 번에 바로 그래프를 생성하게 하였다. 정규 문법, 오토마톤, 제어 흐름 그래프 사이의 변환은 간단하게 수행할 수 있다.

5. 제어 흐름 그래프 정리

제어 흐름 그래프를 만들 때에는 먼저 쉽게 생성할 수 있는 형태의 그래프를 만들고 이를 정리하는 방법을 이용하였다. 실제로 수행된 모든 명령을 노드로 하고, 시작 노드와 종료 노드를 추가하며, 실행 순서대로 간선을 추가하였다. 트레이스가 여럿일 때에는 각각의 그래프를 만든 다음 시작 노드와 종료 노드를 합쳤다.

이렇게 만든 그래프는 노드와 간선의 수가 많고 프로그램의 구조를 잘 반영하지 못하므로 정리할 필요가 있다. 정리를 하면서 노드와 간선을 합치면 그래프는 간단해지고 그래프에 해당하는 정규 언어의 크기는 커진다. 그래프가 너무 간단해지면 분석의 정밀도가 떨어지므로 적절한 수준까지 정리해야 한다. 예를 들어 정리한 그래프가 인터프리터의 구조만을 나타내거나 정리한 그래프에 해당하는 정규 언어가 Σ^*가 되면 그래프를 다른 분석에 활용하기 어려워진다.

제어 흐름 그래프를 정리하기 위해서는 크게 두 가지 기법을 사용하였다. 먼저 분기되는 부분을 탐지하기 위해 각 노드 다음 노드에 공통되는 접두부(prefix)가 존재하거나 각 노드 이전 노드에 공통되는 접미부(suffix)가 존재하면 이들을 묶어 하나의 노드로 만들었다. 접두부와 접미부를 묶을 때에는 가능한 길게 묶었다. 그림 3은 접두부를 탐지하는 함수인 prefix(N,E)를, 그림 4는 접미부를 탐지하는 함수인 suffix(N,E)를 나타낸 것이다. 함수는 각 노드에 대해 바로 다음 노드의 가장 긴 접미부를 찾거나 바로 이전 노드의 가장 긴 접두부를 찾아서 새로운 노드로 만들고, 옛 노드에서 새로운 노드에 해당하는 부분을 제거한 다음, 간선을 추가해 주는 작업을 수행한다. 여기서 dsucc는 노드의 바로 뒤 노드(direct successor)를, dpred는 노드의 바로 앞 노드(direct predecessor)를 찾는 함수이다. 그림 5는 실행 결과의 예를 나타낸 것이다. 두 트레이스에 공통으로 있는 A와 D를 묶어서 하나로 만들었다.

한편 반복되는 부분을 탐지하기 위해 각 노드에 해당하는 트레이스를 더 짧은 트레이스가 반복되는 형식으로 나타낼 수 있는지 여부를 조사한다. 짧은 트레이스의 후보가 여럿 있을 때에는 가장 짧은 트레이스를 선택한다. 그림 6은 반복을 탐지하는 함수인 loop(N,E)를 나

```
for n ∈ N do
  s ← longest common prefix of dsucc(n)
  if |s| > 0 then
    make new node n′ with s
    N ← N ∪ {n′}
    for n″ ∈ dsucc(n) do
      remove prefix s from n″
      E ← E ∪ {(n, n′), (n′, n″)}
    end for
  end if
end for
```

그림 3 공통되는 접두부를 탐지하는 prefix(N,E) 함수
Fig. 3 Function prefix(N,E) identifying common prefixes

for $n \in N$ **do**
  $s \leftarrow$ longest common suffix of dpred(n)
  if $|s| > 0$ **then**
    make new node n' with s
    $N \leftarrow N \cup \{ n' \}$
    for $n'' \in$ dpred(n) **do**
      remove suffix s from n''
      $E \leftarrow E \cup \{ (n'', n'), (n', n) \}$
    end for
  end if
end for

그림 4 공통되는 접미부를 탐지하는 suffix(N,E) 함수
Fig. 4 Function suffix(N,E) identifying common suffixes

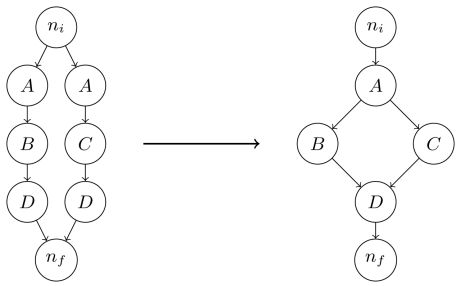

그림 5 분기 탐지
Fig. 5 Identification of a branch

for $n \in N$ **do**
  for $l \leftarrow 2$ **to** $\lfloor \frac{|n|}{2} \rfloor$ **do**
    if n is a repetition of $n_1 \ldots n_l$ **then**
      replace n with $n_1 \ldots n_l$
      $E \leftarrow E \cup \{ (n, n) \}$
      break
    end if
  end for
end for

그림 6 반복되는 부분을 탐지하는 loop(N,E) 함수
Fig. 6 Function loop(N,E) identifying loops

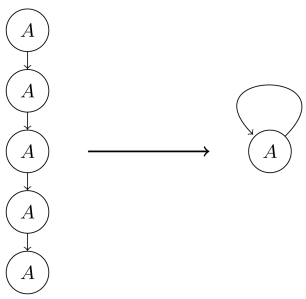

그림 7 반복 탐지
Fig. 7 Identification of a loop

타낸 것이다. 반복이 있으면 나머지 부분을 제거하고 반복에 해당하는 간선을 추가한다. 그림 7은 실행 결과의

$N \leftarrow \hat{T}^{\flat} \cup \{ n_i, n_f \}$
$E \leftarrow \{ (n_i, \hat{\tau}) \mid \hat{\tau} \in \hat{T}^{\flat} \} \cup \{ (\hat{\tau}, n_f) \mid \hat{\tau} \in \hat{T}^{\flat} \}$
repeat
  `loop`(N,E)
  `prefix`(N,E)
  `suffix`(N,E)
until no change is made

그림 8 제어 흐름 재건 알고리듬
Fig. 8 Algorithm for control flow reconstruction

예를 나타낸 것이다. A가 다섯 번 반복되는데 반복문 하나로 만들었다.

그래프 정리는 더 이상 변화가 없을 때까지 반복한다. 그림 8은 전반적인 제어 흐름 재건 알고리듬을 나타낸 것이다. 실행을 통해 얻은 트레이스를 시작 노드와 종료 노드에 연결해서 초기 제어 흐름 그래프를 만든 다음, 변화가 없을 때까지 그래프를 정리한다.

6. 실험 결과

제안한 방법이 동작하는지 확인하기 위해 원본 바이너리와 난독화한 바이너리에 제안한 방법을 적용해 보고, 유사한 제어 흐름 그래프가 생성되는지 관찰하였다.

시스템의 구성은 그림 9와 같다. 먼저 바이너리로부터 트레이스를 여럿 생성하고, 그 트레이스를 이용하여 제어 흐름 그래프 재건을 수행한다.

트레이스의 생성은 Intel에서 나온 동적 바이너리 인스트루멘테이션(dynamic binary instrumentation) 도구인 Pin[11]을 통해 수행하였다. C++를 이용하여 구현하였으며, 트레이스 생성과 관련된 부분은 520줄, 제어 흐름 그래프 생성과 관련된 부분은 1210줄을 작성하였다. 실험은 32비트 윈도 XP와 Visual Studio 2010을 이용하여 수행하였다. 시스템의 CPU는 Intel Core i5-760을 사용하였으며, RAM은 4GB를 설치하였다. 가상화를 위해 VMProtect 2.12.2와 Code Virtualizer 1.3.9.10을 이용하였다.

그림 9 시스템 구성
Fig. 9 System overview

6.1 분기 탐지

분기를 탐지하는지 확인하기 위해 그림 10의 프로그램을 이용하였다. 삼각형을 분류하는 작업을 수행한다. 트레이스는 5 종류의 인수를 이용하여 생성하였다. 트레이스 생성 및 분석에 걸린 시간과 자료의 크기는 표 1과 같다. 그림 11은 제안한 방법을 이용하여 제어 흐름

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
/* Header for Marker */
int main(int argc, char *argv[])
{
  int t = 0; int a, b, c;
  char *s[4]={"scalene","isosceles","equilateral"};
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  c = atoi(argv[3]);
  /* Start Marker */
  if (a == b) t += 1;
  if (b == c) t += 2;
  if (c == a) t += 3;
  if (t == 0) t = 0;
  else if (t > 3) t = 2;
  else t = 1;
  /* End Marker */
  printf("%d, %d, %d: %s\n", a, b, c, s[t]);
  return 0;
}
```

그림 10 삼각형 분류 프로그램

Fig. 10 Triangle classification program

표 1 삼각형 분류 프로그램의 자료 크기 및 생성 시간

Table 1 Data size and generation time for the triangle classification program

Binary Type	Binary Size (B)	Arguments	Execution Time (s)	Trace Size (B)	Analysis Time (s)
Original	46,592	2 2 3	6.09	107,539,871	10.03
		2 3 2	5.96	107,541,295	
		2 3 4	5.96	107,519,541	
		3 2 2	6.01	107,540,442	
		3 3 3	5.94	107,561,920	
VMProtect	57,856	2 2 3	6.98	129,662,433	12.69
		2 3 2	6.98	129,650,084	
		2 3 4	6.67	123,823,778	
		3 2 2	6.91	129,638,607	
		3 3 3	7.13	133,363,767	
Code Virtualizer	64,676	2 2 3	6.34	114,671,896	10.51
		2 3 2	6.38	114,486,255	
		2 3 4	6.24	113,241,829	
		3 2 2	6.34	114,423,699	
		3 3 3	6.40	116,794,234	

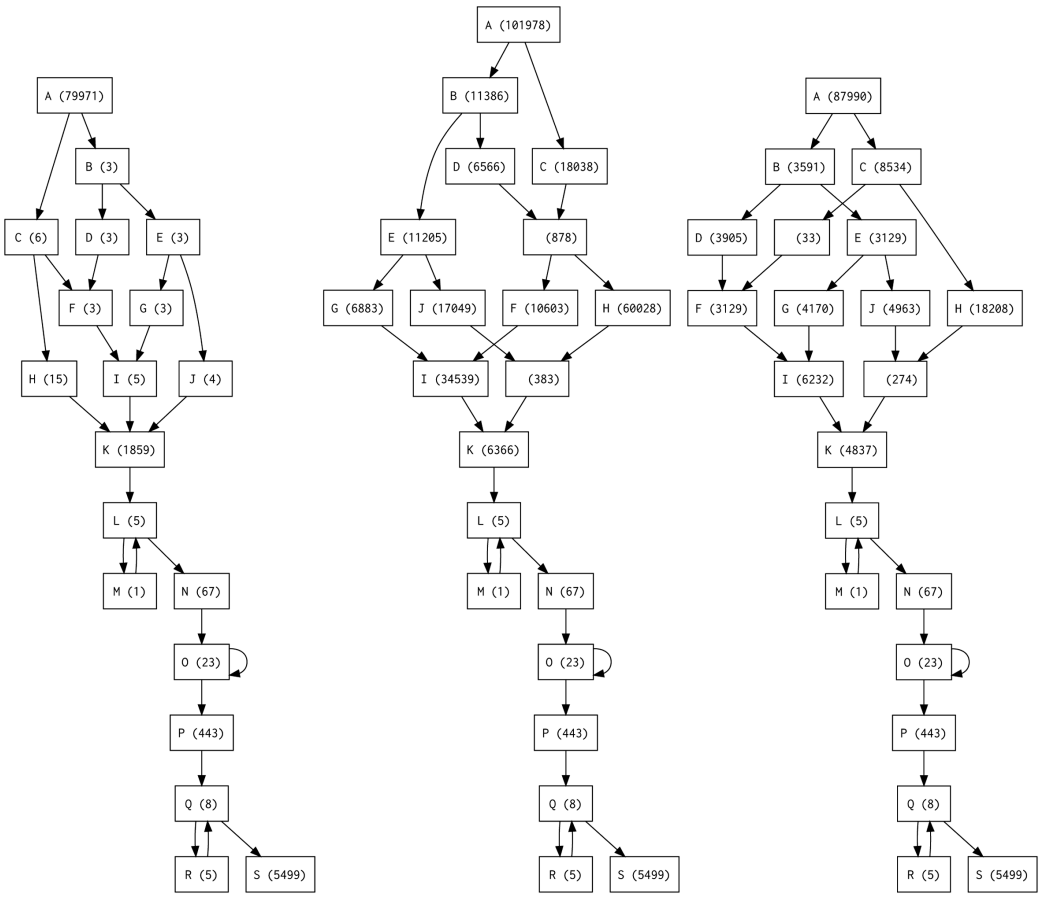

그림 11 삼각형 분류 프로그램의 제어 흐름 그래프 (원본, VMProtect, Code Virtualizer)

Fig. 11 Control flow graphs of the triangle classification program (Original, VMProtect, Code Virtualizer)

그래프를 생성한 것이다. 왼쪽부터 원본 바이너리, VMProtect로 난독화된 바이너리, Code Virtualizer로 난독화된 바이너리에 해당한다. 세 그래프가 유사하므로 난독화의 영향을 제거했음을 볼 수 있다.

원본의 노드에 알파벳으로 이름을 붙였고, VMProtect와 Code Virtualizer의 그래프에 해당되는 노드가 있으면 같은 알파벳으로 표시하였다. 알파벳이 없는 노드는 원본 그래프에 직접 대응하는 노드가 없는 노드이다. 가상화 과정에서 인터프리터가 추가 작업을 수행하거나 같은 의미를 가진 다른 명령을 통해 작업을 수행하면 이러한 노드가 생기게 된다.

괄호 안에 있는 수는 각 노드에서 실행된 명령의 수이다. 여기서는 B~J가 가상화된 부분인데 원본 노드에 비해 크기가 1,000배가량 증가한 것을 확인할 수 있다. 인터프리터가 포함되었고 분석을 번거롭게 하기 위해 무의미한 명령들이 다수 추가되었기 때문에 실행되는 명령의 수가 증가한 것이다. 가상화되지 않은 부분에 있는 명령의 수는 동일한 것을 확인할 수 있다.

6.2 반복 탐지

반복을 탐지하는지 확인하기 위해 정수 덧셈을 수행하는 프로그램을 사용하였다. 프로그램의 코드는 그림 12와 같다. 트레이스 생성을 위해 4 종류의 인수를 사용하였다. 트레이스 생성 및 분석에 걸린 시간과 자료의 크기는 표 2와 같다. 그림 13은 프로그램에 해당하는 제

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
/* Header for Marker */
int main(int argc, char *argv[])
{
  int i, n, sum;
  n = atoi(argv[1]);
  /* Start Marker */
  for (i = sum = 0; i <= n; i++)
    sum += i;
  /* End Marker */
  printf("0 + ... + %d = %d\n", n, sum);
  return 0;
}
```

그림 12 정수 덧셈 프로그램
Fig. 12 Arithmetic series program

표 2 정수 덧셈 프로그램의 자료 크기 및 생성 시간
Table 2 Data size and generation time for the arithmetic series program

Binary Type	Binary Size (B)	Argument	Execution Time (s)	Trace Size (B)	Analysis Time (s)
Original	45,568	0	5.85	145,524,229	10.23
		1	5.91	145,528,404	
		2	5.93	145,531,041	
		9	5.94	145,573,476	
VMProtect	57,344	0	6.60	161,497,253	13.94
		1	6.96	168,694,196	
		2	7.19	175,889,601	
		9	9.26	226,281,412	
Code Virtualizer	63,340	0	6.29	152,474,865	11.98
		1	6.42	155,797,193	
		2	6.55	159,117,983	
		9	7.50	182,387,489	

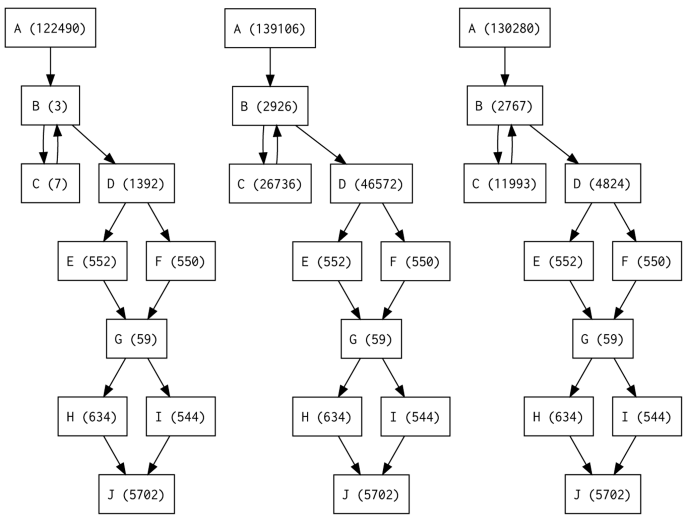

그림 13 정수 덧셈 프로그램의 제어 흐름 그래프(원본, VMProtect, Code Virtualizer)
Fig. 13 Control flow graphs of the arithmetic series program (Original, VMProtect, Code Virtualizer)

어 흐름 그래프이다. 원본 바이너리에 대해 생성한 그래프와 함께 VMProtect 및 Code Virtualizer로 가상화한 바이너리에 대해 생성한 그래프를 볼 수 있다. 세 그래프가 유사함을 확인할 수 있으며, B와 C로 구성된 루프를 볼 수 있다.

7. 관련 연구

7.1 가상화 난독화 분석

가상화 기법은 1960년대부터 컴퓨터 시스템을 효율적으로 활용하기 위한 기법으로 활용되어 왔으며, 2000년대 중반부터 프로그램의 분석과 수정을 어렵게 하기 위한 기법으로 활용되기 시작하였다[4]. 이러한 난독화 기법은 악성 코드도 이용하고 있으며, 시스템 보안을 위해서는 바이너리의 의미를 알아내는 능력이 필요하기 때문에 가상화가 적용된 바이너리에 대한 분석 기법이 연구되었다.

가상화가 적용된 바이너리를 분석하기 위해 가상 기계의 구조를 분석하는 방법이 연구되었다. Rolles는 가상 기계의 구조를 분석하고 이를 바탕으로 바이트코드를 기계 명령으로 옮기는 방법을 소개하였다[2]. 이 방법을 이용할 때에는 사람이 직접 가상 기계의 구조를 분석해야 하므로 많은 바이너리에 빠르게 대응하기는 어렵다. Sharif와 동료들은 가상 기계의 구조를 자동으로 분석하는 방법을 제안하였는데[3], 트레이스를 분석하여 가상 기계의 구조를 알아내고, 각 바이트 코드에 해당되는 실행을 찾아냈으며, 제어 흐름 그래프도 생성하였다. 이 방법은 기계의 구조를 먼저 분석하는데, 본 논문에서 제안하는 방법은 기계의 구조를 분석하지 않는다는 차이가 있다. Kinder는 가상 프로그램 카운터가 어떻게 동작하는지 알고 있을 때 도메인 평탄화 문제를 해결하는 방법을 소개하였다[7].

가상 기계의 구조를 분석하는 데 상당한 노력이 필요하기 때문에 가상 기계의 구조를 분석하지 않고 직접 바이너리의 의미를 알아내는 연구도 진행되었다. Coogan과 동료들은 가상 기계의 구조를 알아내지 않은 상태에서 트레이스를 이용하여 분석을 수행하는 방법에 대한 연구를 수행하였다[12,13]. 외부 환경에 영향을 주는 시스템 콜을 고르고 여기에 영향을 주는 명령들만 골라내어 분석하는 실행의 크기를 줄이는 방법을 제안하였다. 난독화된 실행에서는 실행되는 명령의 극히 일부분만이 외부 환경에 영향을 주기 때문에 이러한 방법으로 분석 작업의 효율성을 높일 수 있다.

7.2 트레이스 분석

트레이스를 이용한 분석은 디버그를 위해 쓰이거나 최적화를 위한 성능 분석에 쓰이며, 프로그램 이해를 위해서도 쓰인다. 본 논문에서는 트레이스로부터 오토마톤을 생성하는 작업을 수행하였는데, 이와 같이 주어진 문자열로부터 대표하는 문법을 찾아내는 작업은 문법 추론(grammatical inference)[14]으로 알려져 있다. 문법 추론은 자연어 처리나 생물 정보 처리를 위해 사용되며, Sequitur[15]를 비롯해 다양한 알고리듬이 개발되어 있다. 본 논문에서는 비교적 간단한 휴리스틱만을 이용하였으나 향후 이러한 알고리듬을 적용해 보고 유용성을 판단해 보고자 한다.

트레이스를 이용하여 분석을 수행하면 실제 실행되는 명령을 반영하기 때문에 정밀한 분석을 수행할 수 있다. 하지만 트레이스의 크기가 커서 사람이 이해하기 어렵거나 시간이 오래 걸리게 될 때도 많다. 이를 해결하기 위해 Bohnet과 동료들은 중요한 부분만 골라내고 반복되는 부분을 정리한 다음 시각화하는 작업을 수행하였다[16]. 이 연구는 소스 코드가 있는 상황에서 함수 호출 트레이스를 분석하였으며, 시각화하는 것에 초점을 맞추었다. 본 논문의 결과도 바이너리의 분석을 용이하게 하는 시각화에 적용할 수 있을 것으로 기대한다.

트레이스로부터 행동 정보를 요약하여 악성 코드 여부를 판단하는 기법도 연구되었다[17]. 함수 호출 트레이스를 오토마톤으로 정리하고 이를 악성 행동에 해당하는 오토마톤과 비교하는 방법으로 분석을 수행하였다. 제어 흐름 정보도 유사하게 활용할 수 있을 것으로 기대한다.

7.3 바이너리 특성 분석

제어 흐름 재건은 바이너리 분석 작업의 핵심 요소 중 하나이기 때문에 바이너리 분석 도구를 개발할 때 함께 연구되어 왔다. Brumley와 동료들이 연구한 BAP[18], Reps와 동료들이 연구한 CodeSurfer/x86[19], Kinder가 연구한 Jakstab[7,20,21] 등이 유명하며, Bardin과 동료들의 연구[22]도 있다. 이러한 연구들은 주로 정적 분석을 이용하여 제어 흐름 재건을 수행하며, 필요한 경우 동적 정보를 함께 사용할 때도 있다[21].

Lim과 동료들은 자바를 대상으로 제어 흐름 그래프를 만들고, 실행 가능한 간선들을 모아서 기준을 삼아 프로그램을 비교하였다[1]. 프로그램의 제어 흐름 정보가 있으면 제어 흐름 그래프를 비교할 수도 있지만, 이와 같이 다른 정보를 추출하여 비교하는 것도 가능해진다.

Preda와 동료들은 메타모픽 악성 코드의 특성을 추출하는 작업을 수행하였는데, 프로그램을 나타내기 위해 명령을 알파벳으로 하는 오토마톤을 사용하였다[9]. 이 연구에서는 정적인 분석에 초점을 맞춘 반면, 본 연구에서는 동적인 방법으로 트레이스를 분석한다.

본 논문에서 제안한 제어 흐름 재건 방법은 제어 흐름 평탄화(control-flow flattening)에도 적용할 수 있다. 제어 흐름 평탄화는 프로그램을 여러 조각으로 나누

고 한 곳에서 제어 흐름을 통제하도록 하는 기법이다[23]. 제어 흐름 평탄화 해제에 관한 기존 논문에서는 노드를 복제하여 후보 제어 흐름 그래프를 생성한 다음 실제 실행이 불가능한 간선을 제거하는 작업을 수행하였다[24]. 기존 방법은 정밀한 정적 분석이 가능하다는 것을 전제로 하고 있어 난독화 구조가 복잡할 때에는 적용하기 어려울 수 있다.

8. 결 론

가상 기계의 구조를 분석하지 않고도 트레이스를 바탕으로 가상화로 난독화된 프로그램의 본질적인 제어 흐름을 재건하는 방법을 제안하였다. 상용 가상화 도구인 VMProtect 및 Code Virtualizer로 난독화된 바이너리를 대상으로 제어 흐름을 추출하였으며, 난독화 이전의 바이너리와 유사한 제어 흐름이 추출되는 것을 확인할 수 있었다. 추출해 낸 제어 흐름은 난독화된 프로그램을 분석하기 위한 기초 정보로 활용할 수 있으며, 프로그램의 비교나 분류 작업에도 활용할 수 있다. 제어 흐름 정보를 바탕으로 프로그램의 시각화를 수행할 수 있고, 제어 흐름 관련 정보를 비교하는 방법으로 프로그램의 유사도를 판단할 수 있다.

트레이스를 이용하여 분석을 수행할 때에는 실제로 실행된 부분만 반영된다는 한계가 있다. 이 논문에서는 바이너리를 분석하는 사람이 프로그램의 실행 결과를 관찰하여 프로그램의 핵심을 실행했는지 여부를 판단할 수 있다고 가정하였다. 앞으로 코드 커버리지 향상 기법을 적용하여 자동으로 프로그램의 핵심을 실행하는 작업을 수행해 보고자 한다. 또한 다양한 문자열 알고리듬을 추가로 적용하여 간접 분기(indirect branch)나 중첩된 반복문(nested loop)과 같이 복잡한 제어 흐름도 분석해 보고자 한다.

References

[1] H. Lim, H. Park, S. Choi, T. Han, "A Static Java Birthmark Based on Control Flow Edges," *Proc. of the 33th Annual IEEE Computer Software and Applications Conference (COMPSAC)*, pp. 413-420, 2009.

[2] R. Rolles, "Unpacking Virtualization Obfuscators," *Proc. of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[3] M. Sharif, A. Lanzi, J. Giffin, W. Lee, "Automatic Reverse Engineering of Malware Emulators," *Proc. of the 30th IEEE Symposium on Security and Privacy (SP)*, pp. 94-109, 2009.

[4] B. Anckaert, M. H. Jakubowski, R. Venkatesan, "Proteus: Virtualization for Diversified Tamper-Resistance," *Proc. of the 6th ACM Workshop on Digital Rights Management (DRM)*, pp. 47-58, 2006.

[5] VMProtect Software, VMProtect [Online]. Available: http://vmpsoft.com/

[6] Oreans Technologies, Code Virtualizer [Online]. Available: http://oreans.com/codevirtualizer.php

[7] J. Kinder, "Towards Static Analysis of Virtualization-Obfuscated Binaries," *Proc. of the 19th Working Conference on Reverse Engineering (WCRE)*, pp. 61-70, 2012.

[8] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, G. Vigna, "Efficient Detection of Split Personalities in Malware," *Proc. of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[9] M. D. Preda, R. Giacobazzi, S. Debray, K. Coogan, G. M. Townsend, "Modelling Metamorphism by Abstract Interpretation," *Proc. of the 17th International Static Analysis Symposium (SAS)*, LNCS 6337, pp. 218-235, 2010.

[10] L. Mauborgne, X. Rival, "Trace Partitioning in Abstract Interpretation Based Static Analyzers," *Proc. of the 14th European Symposium on Programming (ESOP)*, LNCS 3444, pp. 5-20, 2005.

[11] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pp. 190-200, 2005.

[12] K. Coogan, S. Debray, "Equational Reasoning on x86 Assembly Code," *Proc. of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 75-84, 2011.

[13] K. Coogan, G. Lu, S. Debray, "Deobfuscatoin of Virtualization-Obfuscated Software: A Semantics-Based Approach," *Proc. of the 18th ACM Conference on Computer and Communications Security (CCS)*, pp. 275-284, 2011.

[14] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammers*, Cambridge University Press, 2010.

[15] C. G. Nevill-Manning, I. H. Witten, "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research (JAIR)*, Vol. 7, pp. 67-82, 1997.

[16] J. Bohnet, M. Koeleman, J. Doellner, "Visualizing Massively Pruned Execution Traces to Facilitate Trace Exploration," *Proc. of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pp. 57-64, 2009.

[17] P. Beaucamps, I. Gnaedig, J.-Y. Marion, "Behavior Abstraction in Malware Analysis," *Proc. of the 1st International Conference on Runtime Verification (RV)*, LNCS 6418, pp. 168-182, 2010.

[18] D. Brumley, I. Jager, T. Avgerinos, E. J. Schwartz,

"BAP: A Binary Analysis Platform," *Proc. of the 23th International Conference on Computer Aided Verification (CAV)*, LNCS 6806, pp. 463-469, 2011.
[19] G. Balakrishnan, R. Gruian, T. Reps, T. Teitelbaum, "CodeSurfer/x86-A Platform for Analyzing x86 Executables," *Proc. of the 14th International Conference on Compiler Construction (CC)*, LNCS 3443, pp. 250-254, 2005.
[20] J. Kinder, H. Veith, "Jakstab: A Static Analysis Platform for Binaries," *Proc. of the 20th International Conference on Computer Aided Verification (CAV)*, LNCS 5123, pp. 423-427, 2008.
[21] J. Kinder, D. Kravchenko, "Alternating Control Flow Reconstruction," *Proc. of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS 7148, pp. 267-282, 2012.
[22] S. Bardin, P. Herrmann, F. Védrine, "Refinement-Based CFG Reconstruction from Unstructured Programs," *Proc. of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS 6538, pp. 54-69, 2011.
[23] C. Wang, J. Davidson, J. Hill, J. Knight, "Protection of Software-Based Survivability Mechanisms," *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pp. 193-202, 2001.
[24] S. K. Udupa, S. K. Debray, M. Madou, "Deobfuscation: Reverse Engineering Obfuscated Code," *Proc. of the 12th Working Conference on Reverse Engineering (WCRE)*, 2005.

황 준 형
2009년 KAIST 전산학전공 학사. 2011년 KAIST 전산학과 석사. 2011년~현재 KAIST 전산학과 박사과정. 관심분야는 프로그래밍 언어, 프로그램 분석

한 태 숙
1976년 서울대학교 전자공학과 학사. 1978년 KAIST 전산학과 석사. 1990년 Univ. of North Carolina at Chapel Hill 박사 1991년~현재 KAIST 전산학과 교수. 관심분야는 프로그래밍 언어, Secure 소프트웨어, 임베디드 시스템