프로그램의 구조와 상수 값을 이용하는 바이너리 실행 파일의 차이점 분석

(Analyzing Differences of Binary Executable Files using Program Structure and Constant Values)

박 희 완 [†] 최 석 우 [††] 서 선 애 [†††] 한 태 숙 [††††]

(Heewan Park) (Seokwoo Choi) (Sunae Seo) (Taisook Han)

요 약 바이너리 코드의 차이점 분석은 보안 패치와 같은 매우 유사한 두 프로그램 사이의 차이점을 구별해 주는 방법이다. 이전의 연구에서는 분석을 위하여 프로그램의 구조 또는 명령어의 세부 사항만을 각각 이용하였다. 프로그램의 구조를 이용하는 차이점 분석 방법은 제어 흐름의 변화는 잘 탐지해 낼 수 있지만, 버퍼 크기 변화와 같은 상수 값의 변화는 잘 찾아낼 수 없다. 명령어 기반의 차이점 분석 방법은 세부적인 값의 변화는 발견할 수 있으나 명령어 재배치와 같은 컴파일러에 의해 생성되는 불필요한 차이점을 결과로 낸다는 단점이 있다. 이 연구에서는 프로그램 구조를 이용한 비교 분석 방법에 상수 값의 변화를 함께 추적할 수 있는 방법을 제안하고 바이너리 차이점 분석 도구를 구현하였다. 구현된 도구는 윈도 보안 업데이트를 이용하여 평가하였다. 실험 결과 제안된 방법은 구조적인 차이점 분석과 같이 빠른 속도로 구조적인 변화를 찾아낼 뿐 아니라 상수 값의 변화까지 추적할 수 있다는 것을 보였다.

키워드 : 정적 분석, 역공학, 이진 코드 분석, 프로그램 차이점 분석

Abstract Binary diffing is a method to find differences in similar binary executables such as two different versions of security patches. Previous diffing methods using flow information can detect control flow changes, but they cannot track constant value changes. Diffing methods using assembly instructions can detect constant value changes, but they give false positives which are due to compiling methods such as instruction reordering. We present a binary diffing method and its implementation named SCV which utilizes both structure and value information. SCV summarizes structure and constant value information from disassembled code, and matches the summaries to find differences. By analyzing a Microsoft Windows security patches, we showed that SCV found necessary differences caused by constant value changes which the state-of-the-art binary diffing tool BinDiff failed to find.

Key words : static analysis, reverse engineering, binary code analysis, program difference analysis

· 본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원 사업의 연구결과로 수행되었음 (IITA-2008-C1090-0801-0020)

† 비 회 원 : 한국과학기술원 전산학전공
hwpark@compiler.kaist.ac.kr
†† 학생회원 : 한국과학기술원 전산학전공
swchoi@compiler.kaist.ac.kr
††† 비 회 원 : 한국과학기술원 전산학전공 박사후연구원
saseo@compiler.kaist.ac.kr
†††† 종신회원 : 한국과학기술원 전산학전공 교수
han@cs.kaist.ac.kr

논문접수 : 2007년 11월 19일
심사완료 : 2008년 5월 30일

1. 서 론

최근 들어 두 개의 유사한 바이너리 실행 파일의 차이점을 분석하는 도구가 많이 사용되고 있다. 바이너리 실행 파일의 차이점을 분석하는 목적은 악성 코드(malware) 분석[1], 운영체제 보안 패치(security patch) 분석, 그리고 프로그램 도용(code theft detection) 탐지 등이 있다. 악성 코드의 분석은 악성 코드에 의해 감염된 파일과 정상적인 파일을 비교하여 악성 코드의 특징을 발견하여 악성 코드를 탐지하고 감염된 파일을 치료하는 데 이용하기 위한 것이다. 운영 체제 보안 패치의 경우 대부분의 경우 패치를 통해 수정한 내용이 공개된다. 그러나 수정 내용의 공지 없이 패치만 이루어지는

경우가 많다. 또한 좀 더 구체적인 변경 내용을 분석하여 운영 체제 회사에서 지원하지 않는 운영 체제의 버전에 대한 상용 패치를 구현해야 할 경우가 존재한다. 이런 경우에 두 바이너리 실행 파일의 차이점을 분석해야 할 필요가 있다. 코드 도용 탐지를 위해서도 실행 파일의 차이점 분석이 이용된다. GPL(GNU Public License)와 같은 오픈 소스(open source) 프로그램은 소스 코드를 자유롭게 사용할 수 있지만 해당 소스 코드를 이용한 저작물에 대해서도 소스 코드를 공개할 것을 요구한다. 많은 경우 오픈 소스 코드를 이용해서 프로그램을 만들지만 소스를 공개하지 않는다. 바이너리 코드의 비교는 이런 경우에도 코드의 도용을 탐지할 수 있는 근거 자료를 얻는데 사용될 수 있다[2,3].

바이너리 실행 파일의 차이점을 분석하는 것은 단지 기계어 명령어 수준에서 차이점을 보여 주는 것이 아니다. 이 방법은 기계어 명령어의 차이점을 분석하여 소스 코드에서의 변화된 부분을 발견하려는 목적을 가지고 있다. 소스 코드는 컴파일 과정을 통해서 분석에 유용한 함수 또는 변수의 이름이나 타입과 같은 정보들을 잃게 된다. 따라서 기계어 코드로부터 소스 코드의 차이점을 역으로 찾아가는 과정은 소스 코드가 컴파일 되었을 때에도 기계어 코드에 여전히 남아 있는 제한된 정보를 이용하여 분석을 수행해야 하기 때문에 분석의 정확성이 떨어지게 된다. 하지만 바이너리 파일의 차이점 분석 도구는 보안 업데이트와 같은 작은 변화만 있는 매우 유사한 코드 사이에서 분석을 하게 되며, 대부분 같은 운영 체제와 같은 컴파일러를 가정하고 있기 때문에 분석의 정확성을 높일 수가 있다.

기존의 바이너리 실행 파일 차이점 분석 방법은 크게 두 가지로 나눌 수 있다. 첫째는 바이너리 코드의 명령어 단위에서 비교를 시작하여 차이점을 요약해 가며 전체 프로그램 단위까지 분석하는 상향식 분석(bottom-up analysis) 방법이 있다. 둘째는 바이너리 코드 전체를 함수 단위로 나누고 함수 단위에서 비교를 시작하여 명령어 단위까지 차이점을 세분화 하여 분석하는 하향식(top-down analysis) 방법이다.

상향식 분석 방법은 기계어 명령의 차이점과 같은 세밀한 분석을 할 수 있다. 반면 소스 코드에서는 변화가 없으나 컴파일 단계에서 다른 컴파일러를 사용한다거나 최적화 기법을 다르게 적용함을 통하여 발생할 수 있는 기계어 코드에서만 나타나는 불필요한 차이점을 찾아내는 문제점이 있다. 예를 들면 컴파일러나 최적화기법 등을 통해서 명령어 순서 변환(instruction reordering)이나 레지스터 할당(register allocation)에 의한 차이가 생기는 경우에 상향식 분석 방법은 코드에 변화가 있다는 불필요한 결과를 제공한다.

하향식 분석 방법은 소스 코드가 컴파일 되더라도 바이너리 실행 파일에서 프로그램 구조는 원래의 것과 큰 차이가 없다는 것을 이용하는 방법이다. 두 그래프 사이의 구조를 직접 비교할 수도 있지만 시간이 많이 걸린다는 단점이 있기 때문에 일반적으로는 프로그램 구조 정보를 요약하여 비교하게 된다. 먼저 함수 단위로 구조 정보를 요약하고, 두 프로그램 사이에 원래 같은 함수라고 여겨지는 함수 사이에 매칭(matching)이 이루어진다. 함수 단위의 매칭이 끝나면, 기본 블록(basic block) 사이의 매칭이 이루어지고, 마지막으로 명령어 단위의 차이까지 분석하게 된다. 이 방법의 단점은 프로그램의 구조 정보만을 이용하기 때문에 상수 값(constant value)의 변화와 같은 구조가 변하지 않는 차이점은 발견할 수가 없다는 것이다. 또한, 구조 요약 정보만을 이용하기 때문에 서로 다른 함수가 같은 요약 정보를 가지는 부정확한 분석 결과를 내는 경우가 생긴다.

본 연구에서는 프로그램 구조 정보와 상수 값을 요약하여 차이점을 분석하는 바이너리 코드 비교 방법을 제안하고, 제안된 방법을 실제 바이너리 코드 비교에 적용할 수 있도록 구현한 도구를 소개한다. 본 연구에서 제안하는 바이너리 코드 비교 방법은 기존에 제안된 프로그램 구조 정보의 비교 뿐 아니라, 코드 내의 상수 값의 변화도 추적할 수 있으므로, 기존의 방법보다 좀 더 세밀한 분석을 할 수 있다. 우리는 제안된 방법을 기반으로 SCV라는 바이너리 코드 비교 분석 도구를 개발하였다. 이 도구를 실제 Microsoft Windows 원격 실행 취약점에 대한 보안 패치[1]에 적용하여 봄으로써 개발된 도구의 실용성 및 기존 연구 방법과의 차별성을 보이고자 한다.

논문의 구성은 다음과 같다. 제2장에서는 기존의 연구에 대해서 설명한다. 제3장에서는 본 논문에서 제안하는 상수 값과 구조에 기반을 둔 바이너리 코드 비교 방법에 대해 설명한다. 제4장에서는 개발된 도구인 SCV를 소개하고, 실제 MS 윈도 보안 패치에 적용한 결과를 제시한다. 제5장에서는 내용을 요약하고 앞으로 할 일에 대해 기술한다.

2. 관련연구

2.1 명령어 기반 비교 방법

T. Sabin은 그래프 일치(graph isomorphism)를 이용해서 바이너리 실행 파일을 비교하는 알고리즘을 제안하였다[4]. 이 방법은 먼저 디스어셈블러를 이용하여 바이너리 실행 파일의 코드와 데이타를 분리하고, 코드를 함수 단위로 분리한다. 함수는 각 명령어를 노드

1) 보안 패치 KB938827과 KB931906이다.

(node)로, 제어 흐름을 에지(edge)로 하는 그래프 형태로 표시된다. 두 프로그램으로부터 각각 한 개씩 함수를 선택하여 두 그래프 사이의 일치를 계산하게 된다. 함수에는 진입 지점(entry point)이 한 개 이상 존재하므로 진입 지점으로부터 시작하여 각 노드의 일치를 비교한다. 노드의 일치, 즉 명령어의 일치는 기계어 명령어 코드(opcode)의 비교를 통해서 이루어진다. 명령어 코드는 완전히 일치하는 의미를 가진 경우(semantically equivalent), 의미가 유사한 경우, 완전히 다른 경우의 세 가지 구분으로 일치를 계산한다. 두 함수 사이의 비교는 두 개의 큐(queue)를 이용하는 너비 우선 검색(breath-first-search) 방식을 사용한다. 이 방법은 명령어의 순서까지 고려하는 비교를 하기 때문에 의미가 같지만 명령어의 순서만 바뀌었을 때 변경된 부분으로 인식하는 단점이 있다. 그리고 함수 대 함수의 비교를 하는 방법은 제시되어 있지만 어떤 함수들을 선택하여 비교를 시작할지에 대해서는 제시하고 있지 않다.

DarunGrim[5](이하 다른그림)은 eEye Digital Security에서 개발한 바이너리 실행 파일 비교 프로그램이다. 다른그림은 기본 블록부터 비교를 시작하고 함수 단위 매칭으로 비교를 끝내는 상향식 분석 방법을 사용한다. 기본 블록의 비교는 분기문(branch instruction)을 제외한 명령어 코드의 값의 순서(sequence)에 의해 이루어진다. 따라서 컴파일러에 의한 명령어 선택과 명령어 순서에 의해 변경된 부분까지 차이점을 지적하기 때문에 사용자가 관심 없어 하는 차이점을 보여주는 단점이 있다.

2.2 구조 정보를 이용하는 비교 방법

T. Dullien은 실행 파일을 코드의 구조 정보를 바탕으로 비교하는 방법을 제안하였다[6]. 구조적인 정보란 각 함수 내의 제어 흐름 그래프와 함수 호출에 대한 요약 정보를 의미한다. 이 방법은 명령어 기반의 비교 방법보다 빠른 비교가 가능하다는 장점이 있다. 또한 명령어 기반의 비교 방법이 소스 코드는 같지만 컴파일러에 의한 명령어 재배치(instruction reordering)나 레지스터 할당(register allocation)에 의해 바이너리 실행 파일이 변경된 경우 두 파일이 다르다는 결과를 나타내던 단점을 개선하였다.

Dullien의 연구에서는 실행 파일이 패치 등에 의해 변형될 때 변형된 부분만 찾기 위하여 바이너리 파일을 함수 단위로 분리하여 비교한다. 구분된 각 함수는 각각의 어셈블리 코드를 고려하지 않고 제어 흐름만을 이용하여 요약한 후 그래프 동치 관계를 계산하여 두 바이너리를 비교한다.

T. Dullien과 R. Rolles는 그래프 제어 흐름 그래프 기반 비교 기법을 일반화하여 selector와 property 개념을 도입하였다[7]. Property를 사용하여 매칭 대상 범위를 좁히고, selector를 사용하여 매칭 대상을 선택한다. Property와 selector를 어떻게 정하느냐에 따라서 매칭 알고리즘의 속도와 정확도에 차이가 생긴다.

T. Dullien과 R. Rolles의 연구는 BinDiff[8]라는 상용 도구에 구현되었다. 본 연구에서 제안하는 바이너리 비교 방법론은 구조적인 비교 부분에 있어서 이들의 연구를 바탕으로 개발되었다.

3. 바이너리 코드 비교를 위한 알고리즘

이 장에서는 바이너리 코드 차이점 분석을 위해 우리가 제안한 방법을 소개한다. 제안된 바이너리 코드 차이점 분석은 기본적으로 구조적인 비교를 바탕으로 하고, 명령어 수준의 차이점까지 비교해 줄 수 있는 상수 단위 비교를 포함한다. 구조적인 비교는 프로그램을 구성하는 함수 단위 구조를 우선 비교하고, 함수의 매칭이 끝나면 함수를 구성하는 기본 블록 단위의 비교를 수행한다. 함수 매칭 및 기본 블록 매칭을 위해서는 각각을 표현하는 요약이 필요하다. 함수나 기본 블록의 비교를 위해서 사용되는 요약이 너무 성글게 될 경우 중요하지만 작은 변화를 감지할 수 없게 된다. 예를 들면 KB938827[2] 보안 패치의 경우 구조적인 정보가 완전히 일치하지만 상수 값은 다르기 때문에 구조적인 비교만을 이용하는 BinDiff를 통해서는 패치 후의 실행 파일에서 어떤 차이도 발견할 수 없다. 4.2.2절에서 이 보안 패치의 비교에 관해서 자세히 설명한다.

이런 코드의 차이점까지도 감지해 내기 위해서 우리는 상수 정보를 각 함수나 기본 블록의 요약에 포함하도록 한다. 바이너리 코드의 구조적인 정보를 추출하기 위해서 우리는 상용 디스어셈블러인 IDAPro[9]를 사용한다. IDAPro는 바이너리 실행 파일을 디스어셈블하며, 어셈블리 코드를 함수 단위로 구분해 주고, 함수간의 호출 관계 및 제어 흐름 그래프를 생성한다. 이 논문에서는 P, Q를 프로그램으로, f, g를 함수로, m, n를 기본 블록을 나타내는 기호로 사용한다.

3.1 바이너리 구조 정보의 요약

3.1.1 함수 요약하기

구조적인 바이너리 코드의 차이점 비교를 위해서 제일 먼저 함수 정보를 추출한다. 함수의 정보를 어떻게 추출하느냐에 따라 매칭의 정확도가 결정된다. 매칭을 빠르고 효율적으로 수행하기 위해서 함수를 요약한 정보를 사용한다. 함수 f에 대하여, 함수의 요약 정보 $\alpha(f)$는 세 정수의 튜플 (i, j, k)로 나타나는데 각각의 정수는 다음과 같다.

2) http://www.microsoft.com/technet/security/Bulletin/MS07-051.mspx

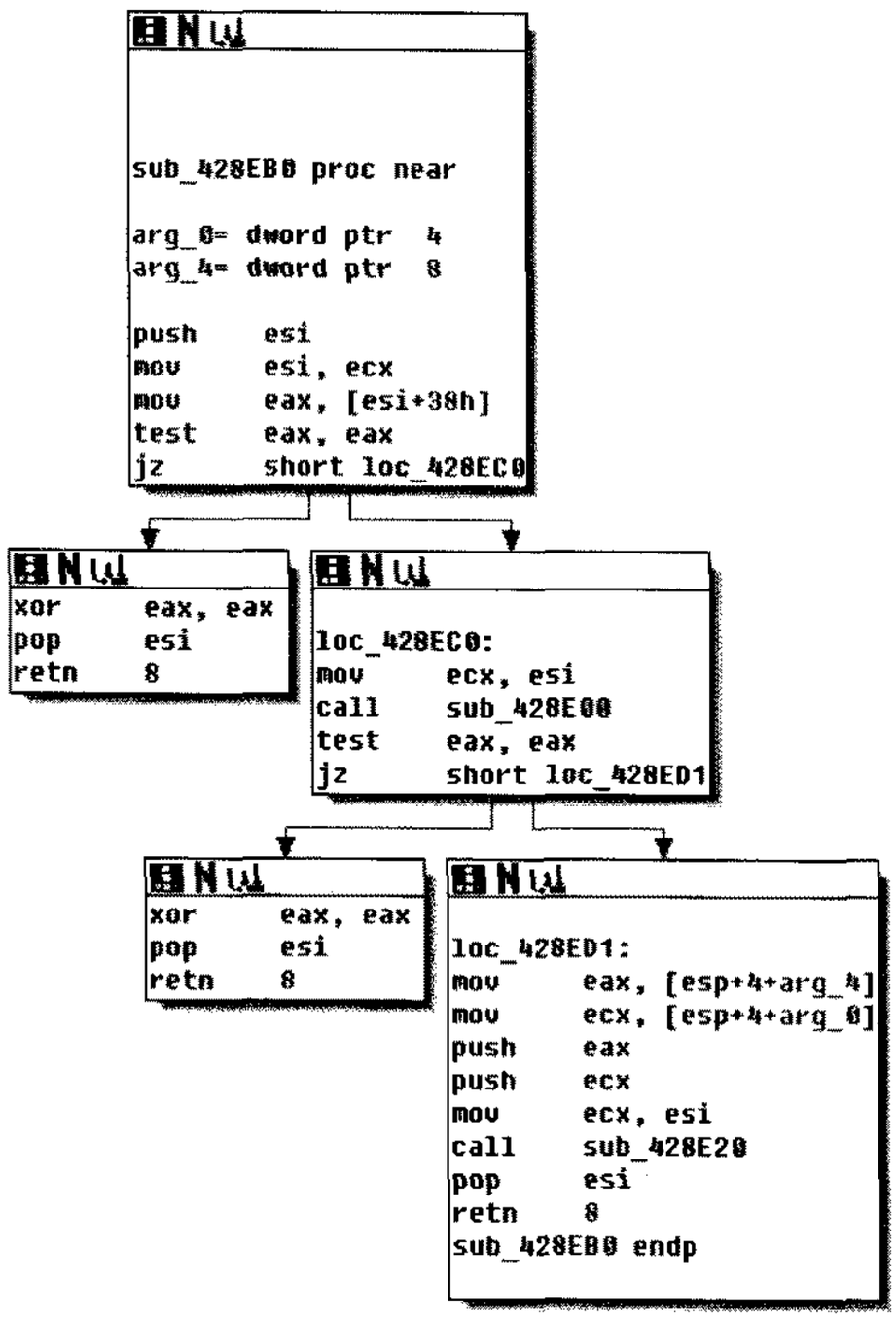


그림 1 IDAPro로 분석한 한 함수의 제어 흐름 그래프

- i : 함수에 포함된 기본 블록(basic block)의 개수
- j : 함수에 포함된 기본 블록 간의 연결선(link)의 개수
- k : 함수에 포함된 함수 호출(function call)의 개수

튜플 내 정수를 가리키기 위해서 우리는 $\alpha_1(f)$와 같은 표기법을 사용한다. 예를 들어 $\alpha(f)=(5,2,3)$이라면, $\alpha_1(f)=5$ 이고 $\alpha_3(f)=3$ 이다.

그림 1은 IDAPro를 이용해서 구조가 분석된 함수의 제어 흐름 그래프이다. 그림 1의 바이너리 코드는 5개의 기본 블록과 4개의 블록 간 연결선, 그리고 2개의 함수 호출을 가지고 있다. 그림에 나타난 함수의 α 값은 (5,4,2)이다.

3.1.2 기본 블록 요약하기

함수 단위 매칭이 이루어지면, 매칭이 이루어진 각 함수 쌍들의 명령어 비교를 수행해야 한다. 이때, 비교는 명령어 단위로 이루어지는 것이 아니고, 함수내의 기본 블록 단위로 수행된다. 각 함수의 기본 블록들은 제어 흐름 그래프를 따라서 표현되므로, 각 기본 블록의 정보를 적절히 추출하면 블록 단위 매칭이 이루어 질 수 있다. 기본 블록 매칭을 위해서 우리는 두 가지 요약 정보를 사용한다. 첫 번째는 블록 내의 함수 호출 개수와 함수 내에서 블록의 위치를 포함하는 요약 정보이고, 두 번째는 블록 내의 명령어 종류를 나타내는 요약 정보이다. 기본 블록 n에 대해서, 첫 번째 요약 정보는 $\beta(n)$으로 나타내고 세 개의 정수로 된 튜플 (i,j,k)로 정의된다. 각 정수의 의미는 다음과 같다.

- i : 함수의 시작 지점(entry point)에서 기본 블록에 도달할 때까지의 최단 거리를 이루는 블록의 개수
- j : 기본 블록에서 함수의 종료 지점(exit point)에 도달할 때까지의 최단 거리를 이루는 블록의 개수
- k : 기본 블록에서의 함수 호출 개수

$\beta(n)$ 정도의 블록 정보만을 가지고도 많은 매칭을 이룰 수 있지만, 이것만 가지고 정확한 기본 블록 매칭이 이루어지지 않는 경우가 있다. 그런 경우 블록 내 명령어들을 좀 더 자세히 살펴야 하는데 그것을 위해서 우리가 사용하는 것은 두 번째 블록 요약 정보인 핑거프린트이다. 핑거프린트 정의를 위해서 우리는 SPP(Small Primes Product) 알고리즘을 사용한다.

정의 3.1

$C=\{c_1,c_2,...,c_n\}$ 모든 명령어 집합;

$P=\{2,3,5,...p_n\}$ 모든 소수들의 집합;

각 명령어를 단 하나의 유일한 소수에 매핑하는 함수 $sp : C \rightarrow P$가 주어졌을 때, 명령어를 위한 SPP 값은 다음과 같다.

$$SPP(\{c_1,c_2,...,c_k\}) = \prod_{i=1}^{k} sp(c_i)$$

위에 정의된 SPP를 바탕으로 기본 블록의 핑거프린트가 정의된다. 어떤 블록 n이 $\{c_1,...,c_k\}$의 명령어들로만 구성되어 있다면, 기본 블록의 핑거프린트는 $\gamma(n)$이라 표시되고, 다음과 같이 정의된다.

$$\gamma(n) = SPP(\{c_1,...,c_k\})$$

SPP를 이용해서 블록 비교를 위한 핑거프린트로 사용하는 이유는 블록 내 명령어 비교를 쉽고 빠르게 수행할 수 있기 때문이다. 주어진 두개의 기본 블록에 대해서, 만일 두 값이 일치한다면 기본 블록은 같은 명령어들로 구성되어 있음을 알 수 있고, 매칭이 완성되게 된다. SPP 알고리즘의 결과로 나온 핑거프린트 값은 소수들의 곱으로 이루어져 있기 때문에 소인수 분해를 통해서 언제든지 원본 명령어 집합을 구할 수 있다는 장점이 있고, 명령어 순서에 상관없이 명령어 조합만 동일하면 항상 같은 핑거프린트 값을 가진다. 또한, 여러 명령어들로 이루어진 기본 블록을 숫자 하나로 요약해서 표현할 수 있기 때문에 기본 블록과 기본 블록 사이의 비교를 숫자의 비교로 바꿔치기 함으로, 비교 속도가 빠르다는 장점이 있다.

3.2 상수 정보의 요약

구조적 비교의 단점은 함수와 기본 블록을 제어 흐름 정보만을 이용해서 요약하기 때문에 버퍼 사이즈의 변

화와 같은 상수 정보에 대한 변화를 추적할 수 없다는 것이다. 따라서 우리는 구조적인 비교 방법에 추가적으로 상수 정보를 이용하여 바이너리 실행 파일의 차이점을 찾을 수 있게 하고 있다. 상수 정보는 함수 또는 기본 블록에 나타나나는 상수 값의 빈도(frequency)를 이용하여 나타낸다. 함수 또는 기본 블록의 상수 정보를 요약하기 위해서 우리는 $\delta(f)$ 라는 표기법을 사용한다.

$$\delta(f)=\{(v_1,n_1),(v_2,n_2),...,(v_n,n_n)\}$$

여기서 v_i는 상수 값, n_i는 블록 내에서 v_i가 나타나는 개수이다. 예를 들어 어떤 함수 f에서 상수 0×800가 2번 쓰이고 상수 0×1000가 1번 쓰였다. $\delta(f)=$ {(0×800,2), (0×100,1)} 이 된다.

3.3 바이너리 비교 알고리즘

3.3.1 함수의 매칭 알고리즘

함수의 기본 매칭은 제 3.1.1 장에서 소개한 함수로부터 추출된 함수 요약 정보를 가지고 시작한다. 함수의 매칭은 매핑 함수 p를 정의함으로써 이루어진다. 두 함수 f,g에 대하여, 그들의 요약인 $\alpha(f)$와 $\alpha(g)$의 값이 각 프로그램내의 함수 요약들 중에 유일하게 정의되고, 또 $\alpha(f)=\alpha(g)$로 일치한다면[3] 함수의 매칭이 이루어지게 된다.

정의 3.2 주어진 두 프로그램이 각각 n,m 개의 함수로 구성되어 있고, 그 함수들의 집합이 $\{f_1,f_2,...,f_n\}$과 $\{g_1,g_2,...,g_m\}$이라 할 때, 두 집합 사이의 매핑 p는 다음과 같이 정의된다.

$$p(f_i)=g_j \quad \stackrel{def}{=} \quad (\forall k\neq i:\ \alpha(f_i)\neq\alpha(f_k)) \wedge(\forall l\neq j:\ \alpha(g_j)\neq\alpha(g_l)) \wedge(\alpha(f_i)=\alpha(g_j))$$

이다.

만일 운이 좋아서 한 함수의 요약 정보가 다른 바이너리 코드에서의 함수 요약 정보와 정확하게 일치하고, 오직 하나만 존재한다면 두 함수 사이에서 매칭이 일어난다. 그러나 요약 정보가 같은 함수들이 여러 개일 경우가 발생할 수 있다. 또한 블록이나 링크가 추가되거나 함수 호출이 추가되어 요약 정보가 변경되었을 경우에는 요약 정보 값을 통하여 동일한 함수를 찾을 수 없는 경우도 발생한다.

함수가 포함하고 있는 기본 블록의 개수가 적을수록 같은 요약 정보를 가질 가능성이 많아진다. 예를 들면, 다른 함수 호출을 포함하지 않는 기본 블록 한 개 만으로 이루어진 함수일 경우 함수 요약 정보는 기본 블록 한 개로 밖에 표현될 수 없고, 이러한 종류의 함수들은 모두 같은 요약 정보를 가지게 된다. 이런 경우에는 매칭 대상 함수를 선택하는 것은 단순하게 요약 정보만으로 해결할 수 없다. 이때는 이미 매칭이 일어난 함수를 기준으로 함수 호출 그래프에서의 호출 관계 정보를 이용하여 매칭을 시도하게 된다. 이 매칭에 대해서는 향상된 매칭 알고리즘이라는 이름으로 다음에 더 자세히 설명이 된다.

함수의 기본 매칭 단계에서 가장 중요한 것은 가능한 많은 함수 매칭을 가능하게 만들어서 그 후의 알고리즘의 복잡도(complexity)를 줄이는 것이다. 함수 요약 정보를 사용한 매칭 기법이 충분히 많은 매칭을 성공시키지 못한다면 후의 과정이 어려워진다.

예를 들어, 두 개의 바이너리 프로그램 P와 Q가 있고, 각각은 $\{f_1,f_2,f_3,f_4\}$, $\{g_1,g_2,g_3,g_4\}$의 함수들로 구성되어 있다고 하자. 두 프로그램의 함수들이 다음과 같이 서로 같은 함수 요약값을 가지고 있다고 하자.

$$\alpha(f_1)=(14,20,5) \qquad \alpha(g_1)=(14,20,5)$$
$$\alpha(f_2)=(5,6,5) \qquad \alpha(g_2)=(5,6,5)$$
$$\alpha(f_3)=(75,94,30) \qquad \alpha(g_3)=(75,94,30)$$
$$\alpha(f_4)=(5,6,5) \qquad \alpha(g_4)=(5,6,5)$$

두 바이너리 코드에서 각각 4개의 함수 요약 정보를 추출하였고, f_1과 g_1은 각 코드에서 유일한 요약값 (14,20,5)를 가지고, f_3과 g_3도 (75,94,30)의 요약값을 유일하게 가지므로, $p(f_1)=g_1$, $p(f_3)=g_3$으로 매칭을 할 수 있다. 그러나 f_2, f_4, g_2, g_4는 모두 (5,6,5)의 값을 가지므로 이들 간에는 복수 개의 매칭이 존재한다. 이 경우, 어떤 매칭이 맞는 것인지 주어진 매칭 알고리즘으로는 판단할 수 없기 때문에, 좀 더 향상된 매칭 방법이 필요하다.

좀 더 향상된 함수 매칭을 위해서, 우리는 함수들의 요약값 뿐만 아니라, 함수들 간의 호출 관계를 고려한다. 어떤 함수에 대해서 함수 요약값이 같은 함수들이 두 개 이상이더라도, 두 함수가 항상 같은 호출, 피호출 관계를 가질 가능성은 매우 낮다. 따라서 함수의 호출 관계를 이용해서 매칭이 완성되지 않은 함수들끼리 매칭을 진행해 주면 더 많은 함수의 매칭을 찾을 수 있게 된다.

다음은 이미 정의된 기본 매칭 알고리즘을 확장한 향상된 매칭 알고리즘이다. 기본 매칭 알고리즘에서 매칭 함수 p를 미리 계산해서 제공한다고 가정하면, 향상된 매칭은 아래와 같이 재귀적인 매칭 함수 재정의를 통해서 완성된다.

$p_0=p.$
$p_{i+1}(f_a)=$

3) $\alpha(f)=\alpha(g)$ 는 $\forall i\in\{1,2,3\}:\alpha_i(f)=\alpha_i(g)$ 인 것과 같다.

$$\begin{cases} p_i(f_a) & \text{if } f_a \in dom(p_i), \\ & \quad (p_i(f_b) = g_d) \\ g_c & \text{if } \exists f_b \in dom(p_i): \wedge (f_b \rightarrow_f f_a) \\ & \quad \wedge (g_d \rightarrow_f g_c), \\ \textit{undefined} \text{ otherwise.} \end{cases}$$

위의 정의에서 $dom(p)$는 함수 p의 정의역, $\rightarrow_f$의 기호는 함수 호출 관계를 의미한다. 예를 들어, $f_1 \rightarrow_f f_2$는 함수 f_1에서 함수 f_2로의 호출이 존재함을 말한다.

향상된 매칭 알고리즘은 이미 기본 매칭 알고리즘으로 매칭이 끝난 집합에서부터 시작된다. 매칭이 된 기본 블록에서 향상된 매칭 알고리즘을 반복하여 적용하면서 매칭 집합을 계속 증가시킨다. 만일 더 이상 추가적인 매칭이 발생하지 않는다면 향상된 매칭 알고리즘의 수행이 완료된 상태이다. 향상된 매칭 알고리즘은 함수의 호출 관계를 함께 보기 때문에 기존의 매칭 알고리즘이 구별하지 못했던 함수를 추가적으로 매칭 시킬 수 있다. 그러나 함수의 호출 관계까지도 일치하는 경우 이 방법을 통해서 모든 함수를 구별해 낼 수는 없다. 이런 경우에는 다음 단계 매칭인 핑거프린트 매칭과 상수 정보 매칭을 추가적으로 이용하여 구별해 내야 한다.

앞서 소개된 두 바이너리 프로그램 P와 Q에 대해서 향상된 매칭 알고리즘을 적용해 보자. 이를 위해서는 우선 함수 호출 관계를 알아야 한다. 각각의 함수들에 대해서 다음과 같은 함수 호출 관계가 있다고 가정하면,

$$f_1 \rightarrow_f f_2,\ f_1 \rightarrow_f f_3,\ f_3 \rightarrow_f f_4$$

$$g_1 \rightarrow_f g_2,\ g_1 \rightarrow_f g_3,\ g_3 \rightarrow_f g_4$$

우리는 다음과 같이 최종적으로 매칭을 완성할 수 있다.

$$p_1(f_1) = g_1$$
$$p_1(f_2) = g_2$$
$$p_1(f_3) = g_3$$
$$p_1(f_4) = g_4$$

3.3.2 기본 블록의 매칭 알고리즘

기본 블록의 매칭 알고리즘은 제 3.1.2 장에서 소개된 기본 블록의 요약값 β와 γ를 바탕으로 이루어진다. 만일 한 블록의 요약 정보가 다른 바이너리 프로그램 내의 한 블록의 요약값과 일치하고, 그 요약값이 하나만 존재한다면 두 블록 사이에서 매칭이 완성된다.

정의 3.3 주어진 두 함수가 각각 u,v 개의 기본 블록으로 구성되어 있고, 그 블록들의 집합이 $\{n_1,n_2,\ldots,n_u\}$과 $\{m_1,m_2,\ldots,m_v\}$ 이라 할 때, 두 집합 사이의 매핑 q는 다음과 같이 정의된다.[4]

$q(n_i) = m_j$ 는

$$(\forall k \neq i:\ (\beta(n_i),\gamma(n_i)) \neq (\beta(n_k),\gamma(n_k)))$$
$$\wedge (\forall l \neq j: (\beta(m_j),\gamma(m_l)) \neq (\beta(m_j),\gamma(m_l)))$$
$$\wedge ((\beta(n_i) = \beta(m_j)) \vee (\gamma(n_i) = \gamma(m_j)))$$

인 것과 같다.

함수 매칭과의 차이점은 블록의 매칭에서는 두 종류의 요약값을 바탕으로 매칭을 수행한다는 점이다. 이 정의에서 우리는 두 블록이 매칭된다 함은 두 블록의 β 값이 같거나 혹은 γ 값이 같은 것을 말한다. 이렇게 두 가지 정보를 바탕으로 매칭을 수행하기 때문에 하나를 이용하는 것 보다 많은 매칭을 얻게 된다.

하지만, 주어진 블록 매칭 알고리즘으로도 해결하지 못하는 경우가 있다. 기본 블록에 대한 요약값이 유일하지 않을 경우가 그렇다. 이럴 때는 이미 매칭이 일어난 기본 블록을 기준으로 제어 흐름 그래프에서의 흐름 관계 정보를 이용하여 매칭을 다시 시도한다. 이 같은 블록 매칭을 위한 향상된 매칭 알고리즘은 함수의 향상된 매칭 알고리즘과 같다. 단지 함수의 경우 함수 호출 관계를 바탕으로 추가 매칭을 수행하고, 블록의 경우 제어 흐름 그래프 상의 제어 흐름 관계를 이용한다는 차이점이 있다.

다음은 이미 정의된 블록의 기본 매칭에서 확장한 향상된 매칭 알고리즘이다. 기본 매칭 함수 q를 미리 계산해서 제공한다고 가정하면, 향상된 매칭은 아래와 같이 재귀적인 매칭 함수 재정의를 통해서 완성된다.

$q_0 = q$

$$q_{i+1}(n_a) = \begin{cases} q_i(n_a) & \text{if } n_a \in dom(q_i), \\ & \quad (q_i(n_b) = m_d) \\ m_c & \text{if } \exists n_b \in dom(q_i): \wedge (n_b \rightarrow_c n_a) \\ & \quad \wedge (m_d \rightarrow_c m_c), \\ \textit{undefined} \text{ otherwise.} \end{cases}$$

위의 정의에서 $\rightarrow_c$의 기호는 제어 흐름이 있음을 의미한다. 예를 들어, $n_1 \rightarrow_c n_2$는 기본 블록 n_1에서 n_2로의 제어 흐름이 존재함을 말한다.

3.4 일치율 계산

3.4.1 구조적 일치율 계산

함수 및 기본 블록의 매칭이 끝났을 때 완전히 일치하는 경우도 있고, 향상된 매칭 알고리즘을 적용한 경우 일부가 일치하는 경우가 있다. 이 정보를 나타내기 위해서 일치율을 계산한다. 일치율은 코사인 측정법(cosine measure)을 사용한다. 함수의 요약값은 각각 $\alpha(f) = (i_1,j_1,k_1),\ \alpha(g) = (i_2,j_2,k_2)$ 라고 하고 각각을 3차원 벡터로 생각할 때 함수일치율은 다음과 같이 정의한다.

4) 쌍으로 된 정수들이 서로 다르다는 것은 다음과 같이 정의된다:
$(a,b) \neq (c,d)$는 $(a \neq c) \vee (b \neq d)$과 같다.

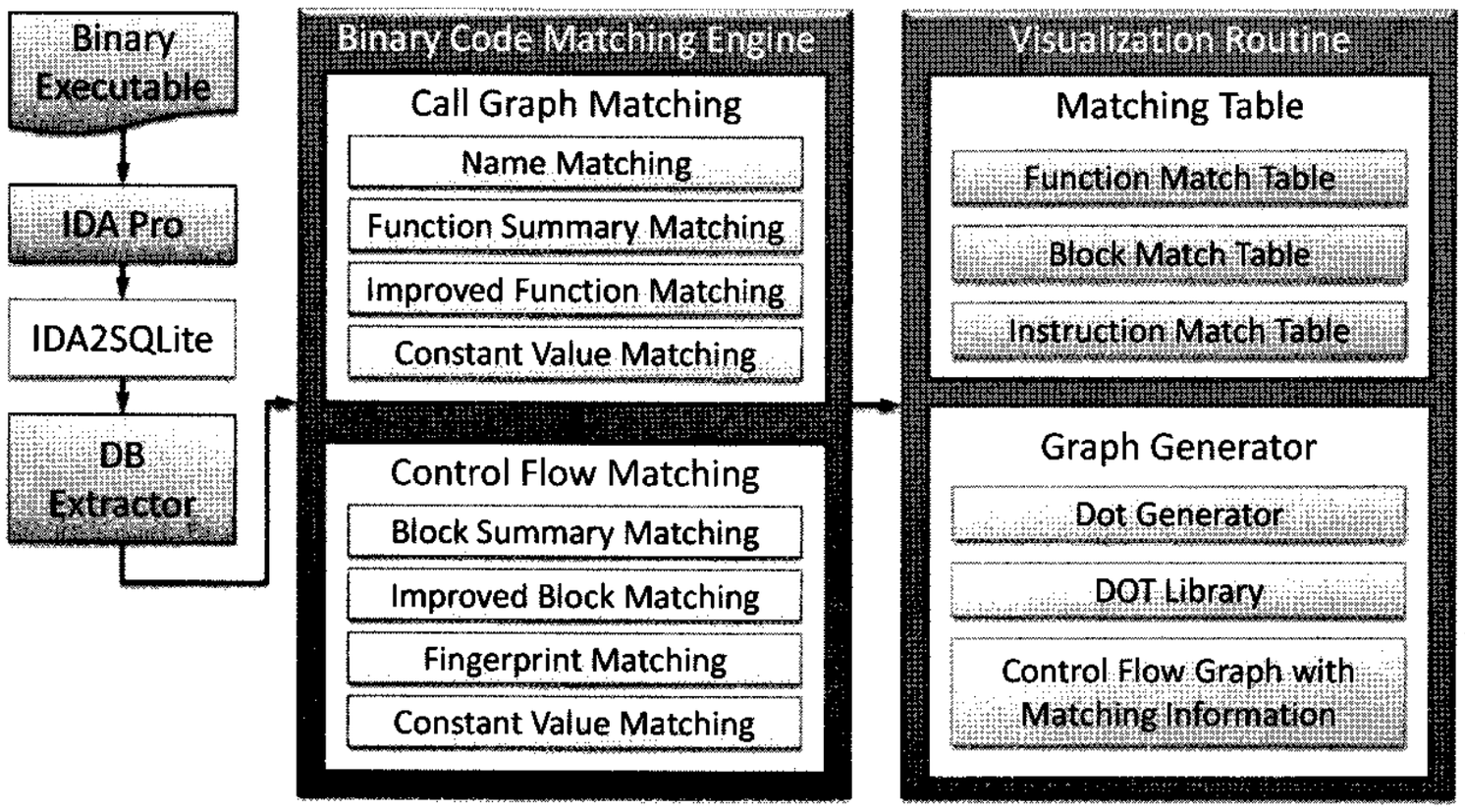


그림 2 SCV 시스템 구조도

$$\text{구조적일치율} = \frac{\alpha(f)\cdot\alpha(g)}{|\alpha(f)|\,|\alpha(g)|}$$

$$= \frac{i_1 i_2 + j_1 j_2 + k_1 k_2}{\sqrt{i_1^2 + j_1^2 + k_1^2}\ \sqrt{i_2^2 + j_2^2 + k_2^2}}$$

일치율은 두 벡터 사이의 코사인 각도를 의미하고 두 벡터의 원소가 전부 양수이기 때문에 값의 범위는 0.0 이상 1.0 이하이다.

3.4.2 상수 일치율 계산

구조적 매칭 값이 같더라도 버퍼 크기와 같은 상수 값만 변한 경우 구조적 비교만을 통해서는 변화된 값을 찾을 수가 없다. 따라서 이 논문에서는 상수 일치율을 계산함으로 상향식 바이너리 실행 파일 비교 방법의 장점을 함께 포함하려고 한다. 제 3.2 절에서 정의된 바와 같이 계산된 두 함수의 상수 요약값이 다음과 같이 $\delta(f)$, $\delta(g)$라고 하자.

$$\delta(f) = \{(v_1,q_1),(v_2,q_2),\ldots,(v_m,q_m)\}$$

$$\delta(g) = \{(w_1,r_1),(w_2,r_2),\ldots,(w_n,r_n)\}$$

이 때 두 함수에 대한 상수일치율은 다음과 같이 정의한다.

$$\text{상수일치율} = \frac{2|\delta(f)\otimes\delta(g)|}{|\delta(f)|+|\delta(g)|},$$

단, $|\delta(f)| = \Sigma_{i=1}^{m} q_i$, $|\delta(g)| = \Sigma_{i=1}^{n} r_i$ 이다.

이 식에서 상수일치율은 문서를 비교할 때에 사용되는 일종의 다이스 유사 계수(dice similarity measure)이다. 분자는 양 함수에서의 일치하는 상수의 총 개수를 의미하며, 분모는 각 함수의 상수의 개수의 합을 의미한다. 엄밀하게 정의하면 다음과 같다.

$$\delta(f)\otimes\delta(g) =$$
$$\{(v,\min(q,r))|((v,q)\in\delta(f))\wedge((v,r)\in\delta(g))\}$$

4. 평가

4.1 구현

바이너리 실행 파일 비교 분석 도구 SCV의 구조는 그림 2와 같다. SCV의 전단부는 기존의 바이너리 비교 도구인 다른 그림이나 BinDiff와 마찬가지로 IDAPro 디스어셈블러를 사용하였다. IDAPro는 바이너리 실행 파일을 분석하여 어셈블리 코드, 함수 정보, 기본블록 정보 등을 생성한다. 이 이 정보를 이용하기 위해서 우리는 IDA2SQLite 플러그인을 구현하였다. IDA2SQLite 플러그인은 IDAPro에서 생성된 정보를 SQLite[5] DB 형태로 저장한다.

DB Extractor는 DB에 저장된 정보를 이용하여 제어 흐름 그래프를 구성하고, 이 그래프와 어셈블리 코드를 분석하여 함수 요약, 기본 블록 요약, 상수 값 정보 요약을 수행한다.

바이너리 코드 매칭 엔진은 요약된 정보를 이용하여 함수 단위 매칭, 기본 블록 단위 매칭 그리고 상수 값 매칭을 수행한다. 매칭 결과는 매칭된 함수들과 함수 일치율, 함수 내의 매칭 된 블록들과 블록 일치율의 테이블로 저장된다.

시각화 엔진은 함수 단위의 비교 테이블을 일치율과 함께 나타내 주고, 함수를 선택했을 때 제어 흐름 그래프와 기본 블록의 매치에 대해 나타내 준다. 제어 흐름 그래프는 DOT 언어[10]로 변환된 후 GraphViz 라이브러리[11]를 이용하여 SVG로 변환된다[12]. SVG는 웹

5) http://www.sqlite.org/

표 1 바이너리 실행 파일 비교를 위한 패치 샘플

	KB931906 (MS 보안공지 MS07-046)	KB938827 (MS 보안공지 MS07-051)
내용	GDI 의 취약점으로 인한 원격 코드 실행 문제점	Microsoft Agent의 취약점으로 인한 원격 코드 실행 문제점
비교대상	GDI32.dll	agentdpv.dll
버전	5.1.2600.3099 / 5.1.2600.3159	2.0.0.3425 / 2.0.0.3426
크기(KB)	275 / 276	52 / 52

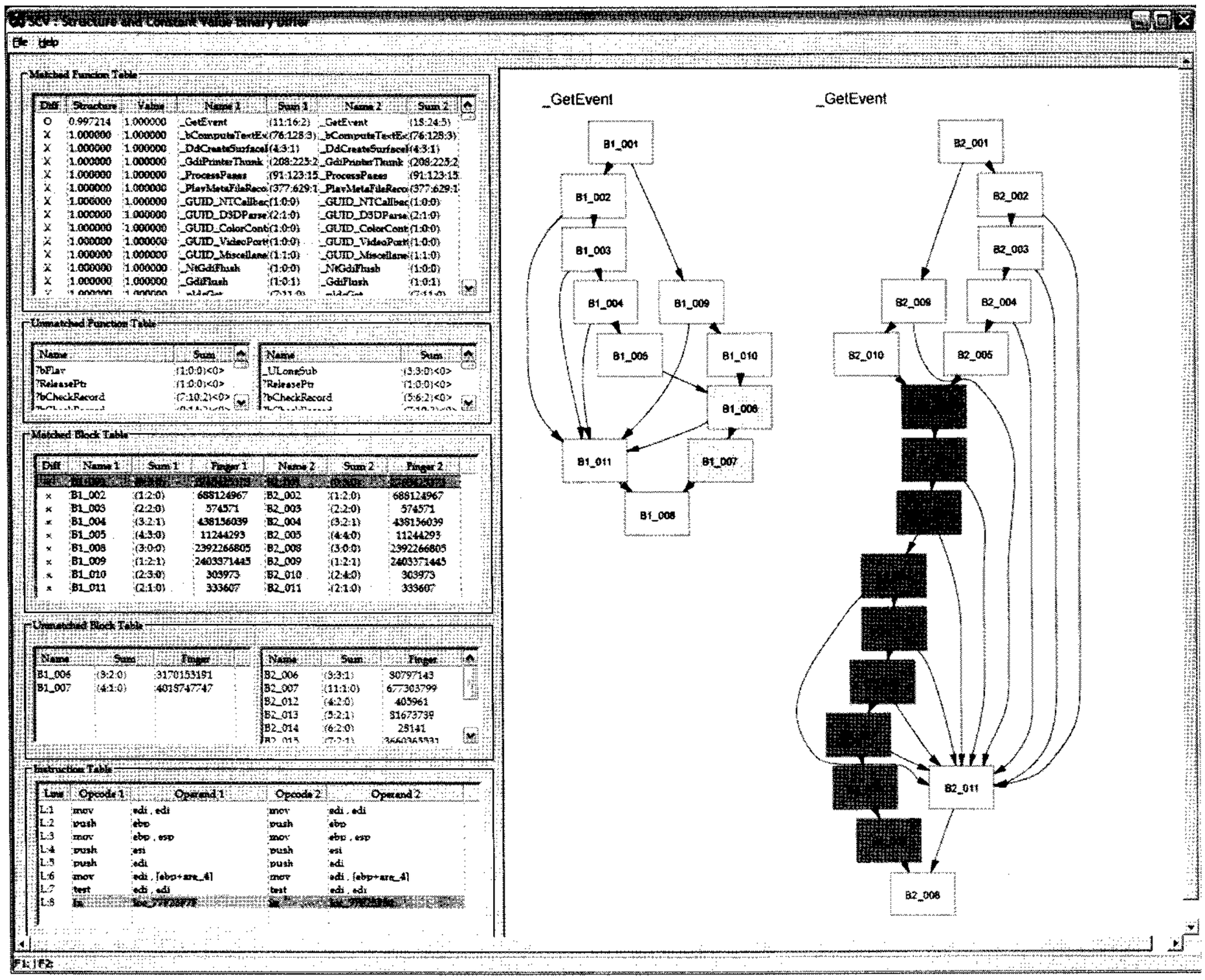


그림 3 SCV 시스템을 이용하여 GDI32.dll을 비교한 실행화면

브라우저 상에서 볼 수 있는데, 어도비의 SVG Viewer[6]에 의해 표현된다.

4.2 실험

구현된 시스템을 평가하기 위하여 MS 윈도 운영체제 보안 패치인 KB931906[7]과 KB938827을 사용하였다. 표 1은 비교를 위한 대상에 대해 설명하고 있다. 두 보안 패치를 우리의 도구인 SCV를 통해 분석하였고 바이너리 파일 비교 분석 도구 중 잘 알려진 Zynamics사의 BinDiff와 비교한다.

4.2.1 KB931906 패치

이 취약점은 이메일로 첨부된 이미지를 열 때 디코딩 과정에서 원격 코드를 실행할 수 있는 권한을 획득할 수 있도록 한다. 그림 3은 이 취약점을 보완한 코드인 GetEvent 함수의 변화를 보여 주고 있다.

그림에 표현된 정보는 함수 매치 테이블(Matched Function Table), 블록 매치 테이블(Matched Block Table), 명령어 테이블(Instruction Table) 그리고 제어 흐름 그래프이다. 그림의 왼편에 있는 다섯 개의 테이블은 각각 함수 매치 테이블, 매치가 되지 않은 함수 테이블(Unmatched Function Table), 블록 매치 테이블, 매치가 되지 않은 블록 테이블(Unmatched Block Table), 그리고 명령어 테이블이다. 그림의 오른편은 함수 매치 테이블에서 선택된 함수 쌍의 제어 흐름 그래프를 비교해서 보여준다. 제어 흐름 그래프에서 코드가 다른 부분은 다른 색으로 표현된다. 예를 들어, 그림 3의 Get-

6) http://www.adobe.com/svg/

7) http://www.microsoft.com/technet/security/bulletin/ms07-046.mspx

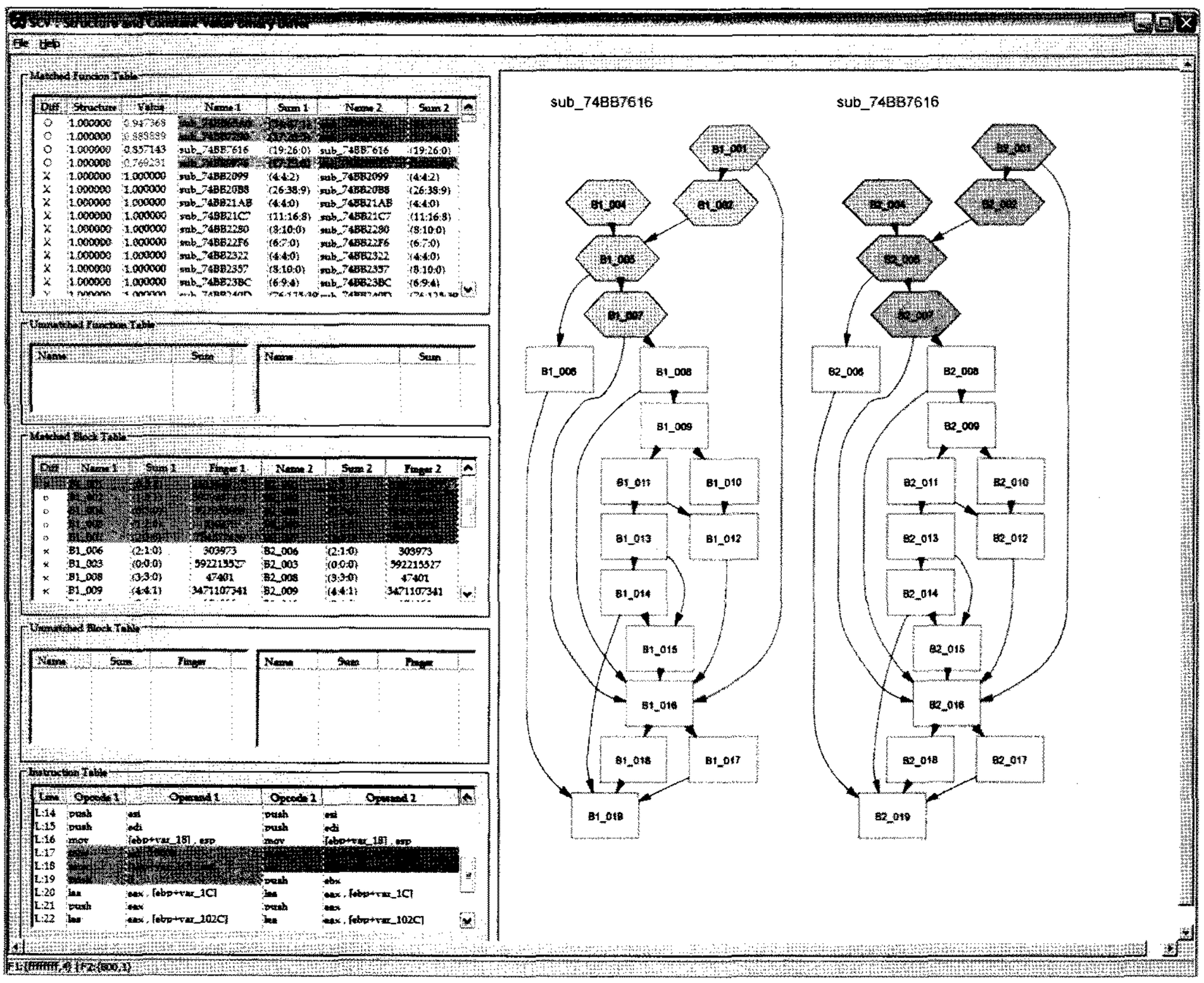


그림 4 SCV 시스템을 이용하여 agentdpv.dll를 비교한 실행화면

Event 함수의 제어 흐름 그래프의 경우, 두 개의 파란색 블록이 패치되어 9개의 붉은색 블록으로 변경되었다. MS는 이 함수의 취약점 보완을 위해서 여러 개의 조건 체크문을 추가하였다. 각 블록을 구성하는 명령어들은 명령어 테이블을 통해서 확인할 수 있는데, 어떤 조건 체크문이 추가되었는지는 이 명령어 테이블을 참조하면 알 수 있다.

4.2.2 KB938827 패치

이 취약점은 MS Agent의 취약점으로 인한 원격 코드가 실행되는 문제점이다. 이 취약점의 경우, 패치 전후의 파일 크기가 동일하며, 구조적인 변경 사항이 전혀 없고, 단지 상수값만 변경된 것이 특징이다. 따라서 BinDiff 는 이 패치를 감지하지 못하고, 변경 사항이 없는 것으로 판단한다. 그러나 그림 4에서 보는 바와 같이 SCV는 상수값의 변화를 비교하기 때문에, 이 패치로부터 네 개의 함수가 변경된 것을 찾아낸다. 그림의 왼쪽 위에 있는 함수 매치 테이블에서 Structure 열의 일치율은 모두 1이지만, Value 열에서는 네 쌍의 함수가 1보다 작은 값을 가진 것을 알 수 있다.

그림의 오른쪽은 특히 네 함수 쌍 중에서 세 번째 함수의 제어 흐름 그래프를 비교한 것이다. 양쪽의 제어 흐름 그래프에서 상수값이 변경된 블록은 육각형 노드로 표현되어 있다.

5. 결론 및 향후 연구

본 연구에서는 보안 패치와 같은 유사한 형태의 바이너리 실행 파일 사이의 비교 방법을 제안하고 구현하였다. 우리는 프로그램의 구조 정보를 요약하고 그 정보를 이용하여 함수와 기본 블록 사이의 차이점을 분석하여 구조적인 매칭을 수행한다. 또한 구조적인 방법의 한계를 보완하기 위하여 상수 값 빈도수를 이용하여 요약하고 비교하는 방법을 사용하였다. 이 방법은 기존에 제안된 프로그램 구조 정보의 비교 뿐 아니라, 코드 내의 상수 값의 변화도 추적할 수 있으므로, 기존의 구조적 비교 방법을 통하여 찾아낼 수 없었던 버퍼 크기의 변화와 같은 핵심적인 정보까지 찾아낼 수 있다. 또한, 명령어 기반의 비교 방법의 단점인 명령어의 재배치나 레지스터 할당의 변화와 같은 불필요한 정보를 찾아내지 않는다는 장점이 있다. 우리는 제안된 방법을 기반으로 SCV라는 바이너리 코드 비교 분석 도구를 개발하였다.

이 도구를 실제 MS 윈도우즈 원격 실행 취약점에 대한 보안 패치에 적용하여 개발된 도구의 실용성 및 기존 연구 방법과의 차별성을 증명하였다.

이 방법은 IDAPro에 전단부를 의존하고 있기 때문에 악성 코드에서 많이 사용되고 있는 패킹(packing)이나 난독화(obfuscation)[13]된 코드에 대해서는 잘 적용할 수 없다는 단점이 있다. 이를 보완하기 위하여 언패커(unpacker)와 역난독화(deobfuscation) 도구가 개발될 필요가 있다. 이런 확장을 통하여 운영체제 패치 뿐 아니라 악성 코드 변종의 분석에 효과적으로 사용될 수 있을 것이다.

참 고 문 헌

[1] Using SABRE BinDiff v1.6 for Malware analysis. http://www.zynamics.com/content/_documents/bindiff_malware.pdf *last accessed 2008.3.21.*

[2] Using SABRE BinDiff for Code theft detection. http://www.zynamics.com/products/Code%20Theft.pdf *last accessed 2008.3.21.*

[3] Choi, S. and Park, H. and Lim, H and Han, T, "A Static Birthmark of Binary Executables Based on API Call Structure," LECTURE NOTES IN COMPUTER SCIENCE, Vol.4846, p.2, 2007.

[4] Sabin, T. 2004. Comparing binaries with graph isomorphisms. unpublished.

[5] eEye Digital Security. eEye Diffing Suite. http://research. eeye.com/html/tools/RT20060801-1.html *last accessed 2008.3.21.*

[6] Flake, H. 2004. Structural comparison of executable objects. *Proceedings of DIMVA 2004:Detection of Intrusions and Malware and Vulnerability Assessment.*

[7] Dullien, T. and Rolles, R. 2005. Graph-based comparison of executable objects. *Symposium sur la securite des technologies de l'information et des communications.*

[8] Zynamics BinDiff. 2007. http://www.zynamics.com/index.php?page=bindiff *last accessed 2008.3.21.*

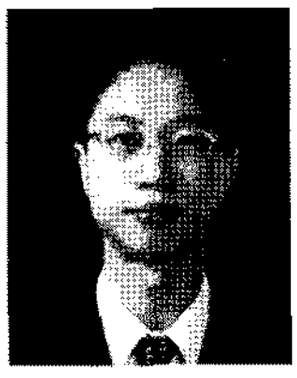
[9] Hex-Rays. The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/ *last accessed 2008.3.21.*

[10] Koutsofios, E. and North, S.C. 1993. Drawing graphs with dot. AT&T Bell Laboratories, Murray Hill, NJ.

[11] GraphViz. Graph vizualization package. AT&T Research.

[12] Ferraiolo, J. and Jun, F. and Jackson, D. 2003. Scalable vector graphics (SVG) 1.1 specification. W3C Recommendation 14.

[13] Linn, C. and Debray, S. 2003. Obfuscation of executable code to improve resistance to static analysis. *10th ACM Conference of Computer and Communications Security (CCS).*

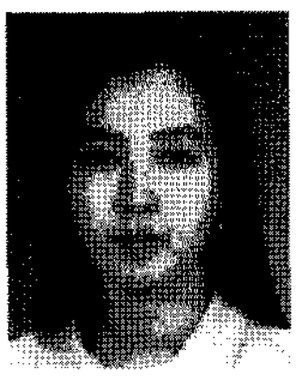


박 희 완

1997년 동국대학교 컴퓨터공학과 학사 1999년 KAIST 전산학과 석사. 2004년~2007년 삼성전자 책임연구원. 1999년~현재 KAIST 전산학과 박사과정. 관심분야는 프로그래밍 언어 및 컴파일러, 소프트웨어 공학

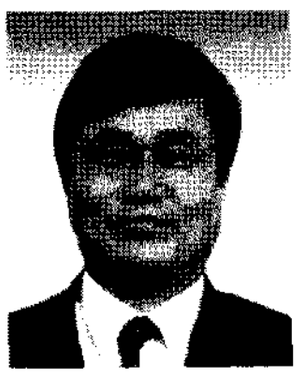
최 석 우

1998년 KAIST 전산학과 학사. 2000년 KAIST 전자전산학과 전산학전공 석사 2000년~현재 KAIST 전자전산학과 전산학전공 박사과정. 관심분야는 이진 프로그램 분석, 프로그램 포렌식스, 컴파일러



서 선 애

1998년 KAIST 전산학과 학사. 2000년 KAIST 전자전산학과 전산학전공 석사 2007년 KAIST 전자전산학과 전산학전공 박사. 2007년~현재 KAIST 전자전산학과 박사후 연구원. 관심분야는 프로그래밍 언어, 프로그램 분석, 임베디드 프로그램 분석



한 태 숙

1976년 서울대학교 전자공학과 학사. 1978년 KAIST 전산학과 석사. 1990년 Univ. of North Carolina at Chapel Hill 박사 1991년~현재 한국과학기술원 전자전산학과 교수. 관심분야는 프로그래밍 언어론, 함수형 언어, 임베디드 시스템 설계 및 분석