



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

공학석사 학위논문

바이너리 기반 자동 취약점 분석
도구 구현 및 성능개선



정보보호학 협동과정

박 종 섭

공학석사 학위논문

바이너리 기반 자동 취약점 분석 도구 구현 및 성능개선

지도교수 이 경 현

이 논문을 공학석사 학위논문으로 제출함.

2019년 2월

부 경 대 학 교 대 학 원

정보보호학 협동과정

박 종 섭

박종섭의 공학석사 학위논문을 인준함.

2019년 2월



주심 공학박사 김 창 수 (인)

위원 이학박사 신 상 욱 (인)

위원 이학박사 이 경 현 (인)

차 례

그림 차례	ii
표 차례	iii
Abstract	iv
I. 서 론	8
1. 연구배경	8
2. 연구 내용 및 구성	11
II. 관련 연구	13
1. 퍼징(Fuzzing)	13
2. DBI(Dynamic Binary Instrumentation)	14
3. 동적 오염 분석	14
4. 오염 분석 최적화	15
5. Concolic Execution	15
III. 자동 취약점 분석 도구 최적화	17
1. 반복적으로 실행되는 코드를 블록 처리 및 최적화	18
가. 오염블록 범위 설정	13
나. 코드 최적화된 오염블록 생성	14
다. 오염객체 전파과정(Propagation) 분석	15
라. 캐시 활용	15
2. 사용자 입력 값에 영향을 받는 명령어만 Concolic Execution 수행	23
가. 명시적 오염 분석(Explicit Taint Analysis)	17
나. Concolic Execution 구성 요소	21
다. 바이너리 기반 Concolic Execution 구현	23
IV. 실험	32
1. 실험 환경	32
2. 오염블록 구현 및 실험 결과	32
3. Concolic Execution 구현 및 실험 결과	40
V. 결 론	41
참고 문헌	35

그림 차례

[그림 1] 바이너리 기반 자동 취약점 분석 도구 구성요소	10
[그림 2] 오염블록 생성 및 실행 순서도	12
[그림 3] 명시적 오염 분석 프로그램 예시	17
[그림 4] 암시적 오염 분석 프로그램 예시	17
[그림 5] 오염 분석 대상 프로그램 main 함수 소스	18
[그림 6] 오염 분석 대상 프로그램 main 함수 어셈블리 코드	19
[그림 7] 어셈블리 코드 오염 분석 예시	20
[그림 8] 취약점이 내포된 프로그램 소스	22
[그림 9] 취약점이 내포된 프로그램 어셈블리 코드	24
[그림 10] 오염블록을 사용한 성능 개선	26
[그림 11] gzip에서의 성능 개선	28
[그림 12] bsdtar에서의 성능 개선	29
[그림 13] gzip2에서의 성능 개선	30
[그림 14] md5에서의 성능 개선	32
[그림 15] 바이너리 기반 Concolic Execution 구현 결과	33

표 차례

<표 1> gzip 실험 결과	27
<표 2> bsdtar 실험 결과	29
<표 3> bzip2 실험 결과	30
<표 4> md5 실험 결과	31



Implementation and Performance Improvement of Binary-based Automatic Vulnerability Analysis Tool

Jong-Seop Park

Interdisciplinary Program of Information Security, The Graduate School,
Pukyong National University

Abstract

An automatic vulnerability analysis tool enables efficient analysis of binary code to automatically detect various types of security vulnerabilities. The dynamic binary analysis techniques used to implement these tools are more accurate than the static binary analysis techniques, but they are very resource intensive to execute and cause severe performance degradation. In particular, this performance degradation is due to the fact that all instructions affected by the input value must be traced during the analysis. In this paper, we present an automatic vulnerability analysis tool that alleviates previously introduced problems. The proposed tool implements two optimization techniques. First, it removes unnecessary operation by first identifying repeatedly executed instructions as blocks and safely removing them from analysis. Second, it implements Concolic Execution method in which only traced objects are set as symbols and included to the analysis. The proposed binary-based Concolic Execution, we claim, has another significance in providing user-friendly environment in that existing tools mandate users to understand and analyze binary code in order to efficiently use the analysis tools.

I. 서론

1. 연구배경

대부분의 소프트웨어는 악용될 수 있는 취약점을 내재하고 있으며, 소프트웨어의 규모가 커지고 복잡해짐에 따라 취약점 탐지에 많은 자원이 요구된다. 따라서 급격히 증가하는 소프트웨어 취약점을 규모 및 속도 면에서 효율적으로 탐지하기 위해 자동으로 취약점을 분석하는 도구가 필요하다. 또한 취약점 분석은 소프트웨어에 대한 소스 코드를 구하기 어렵기 때문에 배포된 바이너리 코드를 대상으로 분석해야 한다. 바이너리 코드 분석은 실행 없이 바이너리 코드를 분석하는 정적 분석과 주어진 환경에서 바이너리를 실행시켜 분석하는 동적 분석으로 나뉜다. 정적 분석은 실행 없이 바이너리 코드를 분석하기 때문에 런타임(Run Time)에 따른 오버헤드(Overhead)가 발생하지 않고 다중 경로를 분석할 수 있는 장점이 있지만, 간접 호출처리 한계에 정밀하지 못한 단점이 있다. 동적 분석은 바이너리를 실행하면서 분석을 같이 병행하기 때문에 하나의 경로에 정확한 값을 가지는 장점이 있지만, 런타임에 많은 오버헤드가 발생하고 단일 경로만 분석 가능한 단점이 있다.

취약점 분석을 위해 소프트웨어를 실행하면서 신뢰할 수 없는 입력 값의 추적 및 명령어 실행에 따른 입력 값의 전과 과정을 추적하는 동적 오염 분석(Dynamic Taint Analysis) [1]이 필요하다. 동적 오염 분석은 DBI(Dynamic Binary Instrumentation)를 기반으로 동작

한다. DBI를 기반으로 구현된 동적 오염 분석은 바이너리 코드를 분석 및 실행 하며, 동적 오염 분석 코드도 삽입하여 사용자가 정의한 입력 값을 추적하게 된다. 동적 오염 분석은 이전 상태를 모르는 상태에서는 오염 전파가 불가능하기 때문에 순차적으로 분석을 진행하여야 한다. 동적 오염 분석에서 순차적 분석으로 인한 성능저하 개선을 위해 멀티코어 환경에서 DBI와 동적 오염 분석을 분리하여 병렬로 수행하는 연구[2]도 진행된 바 있다. 하지만 프로세스 또는 스레드간의 빈번한 동기화 문제로 성능향상이 어렵다. 또한 동적 오염 분석의 성능개선을 위해 정적 오염 분석(Static Taint Analysis) [3]으로 소프트웨어를 사전에 분석하면 성능 개선을 할 수 있지만, 패킹(packing)된 소프트웨어 분석 시 부하가 매우 커져 성능 개선이 되지 않는다. 이러한 이유로 동적 오염 분석은 실제로 잘 사용되지 않아 성능개선이 더 필요하다.

동적 오염 분석을 사용하면 사전에 정의된 취약점 패턴과 비교하여 사용자 입력 값에 영향을 받는 취약한 명령어를 추출할 수 있다. 하지만 동적 오염 분석은 입력 값에 따른 하나의 경로만 분석 가능하다. 다중 경로 탐색을 위해 입력 값을 자동으로 생성하는 Concolic Execution[4]이 필요하다. Concolic Execution은 실제 수행(Concrete Execution)과 기호 실행(Symbolic Execution)의 합성어로, 취약점 분석 시 입력 값으로 심볼과 구체적인 값을 같이 사용하는 기법이다. angr[5]과 같이 바이너리 기반 Concolic Execution 기존 연구는 도구 사용 사전에 바이너리 코드를 분석하고 이해해야 하는 한계가 있다. 따라서 완전한 자동 취약점 분석 도구 구현을 위해서는 분석 대상 소프트웨어의 사전 지식 없이 손쉽게 사용 가능한 사용자의 편의성을 제공할 필요가 있다.

본 논문에서는 자동 취약점 분석을 위해서는 동적 오염 분석을 하면서 반복문과 같이 성능저하가 많이 발생하는 코드의 문제점과 사용자가 분석에 관한 사전 지식이 필요한 문제점을 인지하였다. 이를 개선하기 위해 두 가지 최적화 기법을 제안 및 구현한다. 첫 번째로 소프트웨어를 실행하면서 반복적으로 실행되는 코드를 블록 처리 및 최적화하여 동적 오염 분석의 성능을 향상시키는 오염블록(Taint Block) 기법을 제안 및 구현한다. 두 번째로 사용자가 입력 값 지정만으로 해당 입력 값이 영향을 미치는 객체만 심볼로 설정하여 다중 경로에서 취약점을 분석하는 Concolic Execution 기법을 구현한다.



2. 연구 내용 및 구성

바이너리 기반 자동 취약점 분석 도구는 속도가 느리고 높은 테스트 커버리지 달성을 위한 구현이 어렵다. 따라서 동적 오염 분석에서 반복적으로 실행되는 코드를 블록 처리하여 분석 속도를 높이고, 사용자의 편의성과 바이너리 기반 자동 취약점 분석 도구 구현을 위해 사용자 입력 값만 심볼로 정의하여 Concolic Execution을 수행하는 방법을 제안한다.

취약점 분석을 위한 사용자 입력 값 추적은 간접 메모리 참조와 같이 추적되는 객체를 예측할 수 없는 명령어가 존재한다. 따라서 차례로 모든 실행 명령어를 분석해야 하므로 분석 속도가 매우 느리다. 특히 반복적으로 실행되는 코드에서 속도저하가 많이 발생하여 반복 코드영역을 블록으로 설정하여 최적화된 코드로 동적 오염 분석을 수행할 수 있도록 한다.

Concolic Execution에서 바이너리 기반 분석의 경우 변수, 타입, 함수의 정의가 어려워 어떤 객체를 심볼로 지정할지 문제가 발생한다. 때문에 사용자가 사전에 바이너리 코드를 분석하고 이해해야 하는 문제가 발생한다. 따라서 사용자가 정의한 입력 값만 심볼로 설정하여, 사용자 편의성을 높이고 바이너리 기반 다중경로 탐색이 가능한 Concolic Execution을 구현한다.

본 논문의 구성은 다음과 같다.

2장에서 자동 취약점 분석 도구 구현에 대하여 관련 연구를 살펴보고, 3장에서 성능 개선과 구현을 위한 최적화 기법 두 가지에 대해 설명한다. 4장에서 오염블록을 적용하여 수행 속도에서 성능개선이

이루어지는지와 바이너리 기반의 취약점 분석 도구에서 다중 경로 탐색을 위한 Concolic Execution 적용 방법을 보여준다.



II. 관련 연구

본 장에서는 자동 취약점 분석 구현과 성능개선에 대한 선행연구 내용에 대해서 간략히 살펴본다.

1. 퍼징 (Fuzzing)

퍼징 [6]은 소프트웨어 취약점 탐지의 한 기법으로, 대상 소프트웨어에 정상적인 입력뿐만 아니라 크래시가 발생 가능한 무작위 값을 입력하는 기법이다.

퍼징은 소프트웨어의 취약점과는 관계없는 입력 값도 같이 테스트 하는 경우에 취약점 탐지까지 많은 시간이 소요된다. BuzzFuzz [7]는 동적 오염 분석을 먼저 실시, 소프트웨어를 분석하여 취약점을 발생 시킬 수 있는 입력 값 위치를 확인한다. 확인된 입력 값 위치를 집중적으로 테스트하여 취약점 탐지에 소요되는 시간을 단축하였다.

또한 퍼징은 사전에 소프트웨어 바이너리 코드 분석이 없어 매우 간단한 결함만 찾는다. 일례로 소프트웨어에 체크섬 확인 코드가 포함된 경우 퍼징은 체크섬 확인 코드에 대해서만 취약점을 점검하게 된다. TaintScope [8]는 Concolic Execution 기법을 사용하여 제한된 취약점 점검 문제를 해결하였다.

이와 같이 퍼징은 동적 오염 분석, Concolic Execution과 같은 바이너리 코드 분석 기법과 연계하여 취약점 탐지 연구가 진행되고 있다.

2. DBI(Dynamic Binary Instrumentation)

DBI는 분석 대상 바이너리를 실시간으로 번역(Translation) 및 실행 하는 도구로 그 과정에 사용자가 정의한 코드를 삽입 할 수 있어 동적 오염 분석의 기반이 된다. 널리 사용되는 DBI 도구로는 Valgrind[9], Pin[10], DynamoRIO[11]가 있다. DBI는 기계어 코드 한개 마다 번역과 실행을 수행하기 때문에 심각한 속도저하를 초래한다. 해당 문제를 해결하기 위해 Valgrind와 DynamoRIO는 기본 블록(Basic Block) 단위로 번역을 하고 캐시에 저장하여 분석 대상 바이너리를 실행한다. Pin은 코드캐시(Code Cache)를 이용하여 번역된 코드를 저장시켜 가상머신에서 재사용한다.

3. 동적 오염 분석

동적 오염 분석은 신뢰할 수 없는 어떤 입력 값이 소프트웨어가 실행되면서 어떻게 전파되는지 추적한다. 실행되는 프로세스 내에서 동적으로 추적되는 오염객체(Taint Object)는 레지스터와 메모리이다. 동적 오염 분석은 신뢰되지 않은 입력 값이 어떻게 실시간으로 오염 객체에 전파되는지 전체과정을 분석 할 수 있어 보안 분야에서 많이 사용된다.

동적 오염 분석 코드는 DBI 도구가 실행시켜 주는 사용자 정의 코드부분에 삽입된다. 따라서 DBI 도구의 번역/실행에 동적 오염 분석 도구의 실행부하까지 더해져 분석속도가 매우 느려지는 문제가 발생

한다. 이러한 문제를 해결하기 위하여 선행 연구된 방법으로는 하드웨어 지원 방식[12]과 병렬처리 방식[13]이 있다.

4. 오염 분석 최적화

오염 분석의 성능 향상을 위해 오염 분석에 필요하지 않은 명령어 제거, 다른 명령어에서 같은 오염객체 검사인 경우 하나로 통일, DBI에서 코드 실행 시 빠른 코드로 대체와 같은 최적화 기법[14]이 연구되었다.

또한 동적 오염 분석 전에 실행 가능한 모든 경로를 고려하여 정적 오염 분석을 수행하면, 성능저하 없이 동적 오염 분석을 할 수 있다. 하지만 모든 경로에서 오염객체 정의가 정확하지 않으며, 패키징된 소프트웨어는 정적 오염 분석이 매우 어렵다는 문제가 있다.

5. Concolic Execution

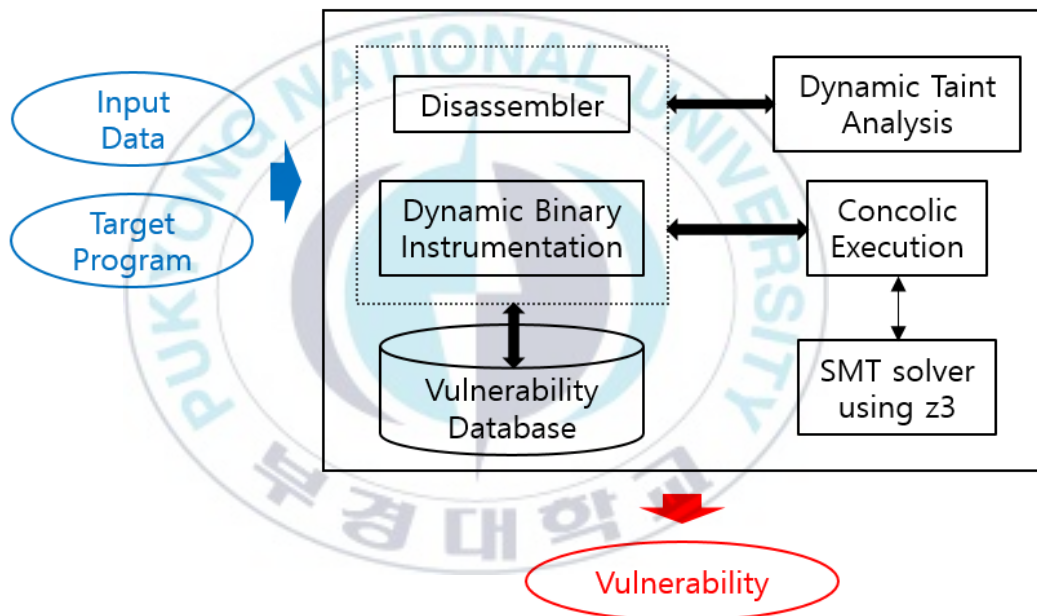
Concolic Execution은 실제 수행(Concrete Execution)과 기호(Symbolic Execution)의 합성어이다. 동적 분석의 문제점은 단일 경로에 대해서만 분석이 가능하다는 것이다. Concolic Execution은 동적으로 다중 경로가 실행 가능한 입력 값을 생성하는 기법이다. 소프트웨어의 실행경로는 분기 명령어 실행마다 2개씩 늘어나게 되어 모든 실행경로는 폭발적으로 증가하게 된다. 따라서 모든 경로를 실행하는 입력 값을 생성하는 것은 비효율적이다. BitBlaze[15]에서는 입력 값에 영향을 받는 분기 명령어에서만 실행경로를 생성하여 폭발

적 증가 문제를 해결한다. 다중 경로 탐색을 위한 Concolic Execution 프레임워크는 Triton[16], angr이 있다.



III. 자동 취약점 분석 도구 최적화

[그림 1]은 바이너리 기반 자동 취약점 분석 도구의 구성요소이다. 사용자는 취약점 분석 대상 소프트웨어와 입력 값만 지정하면 자동으로 분석이 이루어진다. 디스어셈블러, DBI, 동적 오염 분석, SMT(Satisfiability Modulo Theories) solver, Concolic Execution를 사용하여 결과로 취약점을 제공한다.



[그림 1] 바이너리 기반 자동 취약점 분석 도구 구성요소

대상 소프트웨어가 바이너리이기 때문에 디스어셈블러로 바이너리 코드를 분석 가능한 어셈블리 코드로 변환한다. 변환된 어셈블리 코드를 DBI로 실행하면서 분석을 병행하기 위해 동적 오염 분석과 Concolic Execution을 수행한다. 동적 오염 분석으로 사용자가 정의한 입력 값을 추적하고, 다중경로를 수행하기 위해 Concolic Execution을 사용한다. Concolic Execution에서 분기 명령어 분석

시 다음 상태로 가기 위한 실제 값을 SMT solver로 구한다. SMT solver는 z3[17]를 사용한다. 다중경로에서 입력 값을 추적하면서 취약점 데이터베이스를 참조하여 사전에 정의한 취약점이 발생할 수 있는 경우 결과로 제공한다. 사전에 정의된 취약점 종류는 오버플로우(Overflow), 포맷스트링(Format String), 메모리 오염(Memory Corruption)과 같이 소프트웨어를 제어하여 원하는 코드를 실행할 수 있는 취약점이다.

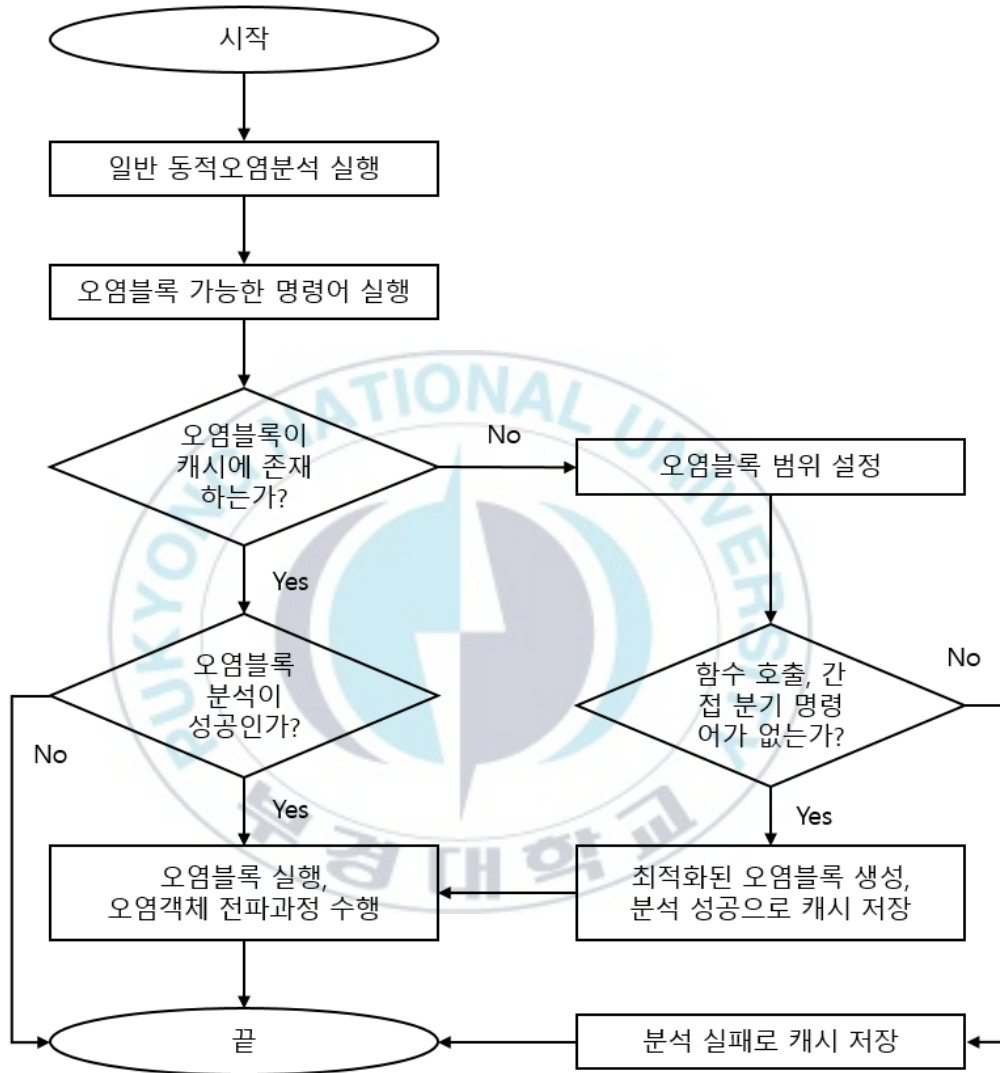
자동 취약점 분석 도구의 구성요소에서 대상 소프트웨어의 모든 어셈블리 명령어를 실행 및 분석하는 DBI와 동적 오염 분석에서 성능 저하가 가장 많이 발생하고, Concolic Execution을 사용하기 위해서는 사용자가 사전에 바이너리를 분석하고 해석하여 취약점 분석 도구를 설정해야 하는 문제가 있다. 본 장에서는 이러한 문제점 개선을 위해 두 가지의 최적화 기법을 제안한다.

1. 반복적으로 실행되는 코드를 블록 처리 및 최적화

[그림 2]는 제안 기법의 순서도(flowchart)이다. 먼저 동적 오염 분석을 수행하며 오염블록으로 구성하는 것이 가능한 명령어들을 찾게 되면 캐시를 검사한다. 캐시에 오염블록이 이미 존재하면, 오염블록 생성 성공 여부를 판별하여 동적 오염 분석을 수행한다. 캐시에 없으면 새로운 오염블록을 생성한다.

오염블록 생성은 먼저 반복적으로 실행되는 코드의 범위를 확인하여 블록의 범위를 명확하게 설정한다. 블록의 범위가 설정이 되면 블록에 포함된 명령어 중에 분석 불가능한 명령어가 존재하는지 검사한다. 검사 이후 블록이 분석 가능한 명령어로만 이루어진 것이 확인되

면 동적 오염 분석을 최소한의 코드만으로 수행할 수 있는 최적의 방식으로 오염블록을 생성하여 캐시에 저장하고 계속 실행한다. 오염블록 실행이 끝나면 저장된 오염객체의 전파과정 분석을 수행한다.



[그림 2] 오염블록 생성 및 실행 순서도

가. 오염블록 범위 설정

DBI 도구로 바이너리 코드를 번역/실행 하면서 낮은 주소로 분기하는 분기 명령어 또는 함수를 호출하는 명령어를 만나면 오염블록 범위 설정을 시작한다. 범위설정을 위해 분기명령어는 분기된 주소, 함수 호출은 호출된 주소의 명령어부터 분석을 시작한다. 분석시작주소 보다 낮은 주소를 실행하는 명령어가 나오면 해당 주소를 시작으로 블록 범위를 다시 설정한다. 블록의 끝은 처음 오염블록의 범위설정을 시작하게 된 명령어에 따라 다르게 설정되는데 분기명령어로 시작한 경우 처음 낮은 주소로 분기가 발생한 분기명령어의 주소이며, 함수 호출의 경우 호출된 마지막 명령어 주소로 설정된다. 오염블록에는 함수 호출 명령어 그리고 간접 분기명령어는 포함 될 수 없다. 포함 될 수 없는 명령어가 있으면 분석에 실패했다는 결과를 캐시에 저장한다.

나. 코드 최적화된 오염블록 생성

오염블록의 범위가 정해지면 명령어의 피연산자(Operand)를 분석하여 코드 최적화를 한다. 최적화된 코드는 전파 가능한 오염객체를 최소한의 메모리와 레지스터를 사용하는 실행으로 전파 순서대로 메모리에 저장하게 한다. 사용자의 입력 값이 전파되는 오염객체인 피연산자 메모리는 오염블록 생성 시점에 정확한 주소를 인지할 수 없기 때문에 오염객체의 정보를 순서대로 저장하게 된다. 오염객체의 전파 정보를 메모리에 저장하는 이유는 많은 정보를 저장할 수 있는 저장장치 중에 접근이 가장 빠르기 때문이다. 하지만 메모리도 저장에 한계가 있기 때문에 초과하는 경우, 저장된 오염객체 전파과정을 분석 후 다시 오염블록을 실행할 수 있게 코드를 생성한다.

DBI 도구의 실행 코드와 동적 오염 분석에 필요한 오염객체 전파과정을 저장하는 최적화 코드가 완성되면 캐시에 저장한다. 오염블록을 재실행할 경우 캐시를 확인하여 기존에 생성한 오염블록 코드를 사용하기 때문에 DBI 도구의 분석 과정과 동적 오염 분석에 필요한 명령어 피연산자 분석과정이 필요치 않다. 그리고 최적화된 코드로 명령어 실행 및 동적 오염 분석에 필요한 오염객체를 저장하기 때문에 속도가 향상된다.

다. 오염객체 전파과정 (Propagation) 분석

오염블록의 실행이 종료되면 저장된 오염객체 전파과정을 한꺼번에 수행한다. 오염블록에서 오염객체를 저장할 때 전파과정 분석에서 최소한의 명령어로 사용할 수 있게 저장하여, 오염블록 실행에서 전파과정 분석으로 넘어가는 과정에 성능 부하를 야기하지 않는다. 오염객체 전파과정 분석은 일반적인 동적 오염 분석과 동일하여 성능 향상이 발생하지 않는다.

라. 캐시 활용

오염블록의 생성이 종료 되면 분석 성공/실패로 나누어 캐시에 저장한다. 다시 오염블록 생성 가능한 명령어 수행 시 먼저 캐시를 검사한다. 캐시에서 오염블록 분석 성공일 경우 오염블록을 재사용하기 때문에 성능이 향상된다. 분석 실패일 경우 일반적인 동적 오염 분석을 실행한다.

2. 사용자 입력 값에 영향을 받는 명령어만 Concolic Execution 수행

Concolic Execution을 수행하기 위해서는 심볼, 경로 조건, 다음 상태를 계속 저장해야 한다. Concolic Execution에서 소스 기반 분석의 경우 변수, 타입, 함수의 정의가 명확하기 때문에 심볼의 지정이 수월하다. 하지만 바이너리 기반에서 심볼을 지정하기 위해서는 사전 분석이 필요하다. 심볼의 생성과 전파 정의가 명확하지 않으면 분석이 불가능 할 정도로 많은 심볼이 생성된다.

간단한 설정만으로 사용가능한 자동 취약점 분석 도구 구현을 위해 Concolic Execution 수행 시 사용자가 정의한 입력 값만 심볼로 설정한다. 그리고 바이너리 기반 Concolic Execution 구현을 위해 심볼 생성과 심볼 전파 시 정확한 값의 전달만을 분석하는 명시적 오염 분석(Explicit Taint Analysis)만을 사용한다.

가. 명시적 오염 분석(Explicit Taint Analysis)

바이너리 기반에서는 사용자 입력 값에 영향을 받는 명령어만 정확하게 분석하기 위해 명시적 오염 분석을 사용한다. 명시적 오염 분석은 [그림 3]과 같이 변수 a가 오염객체일 때 변수 b는 a로부터 값을 전달 받기 때문에 b도 오염객체가 된다.

```
int a, b;
```

```
a = 2;  
b = a;
```

[그림 3] 명시적 오염 분석 프로그램 예시

암시적 오염 분석(Implicit Taint Analysis)은 명시적 오염 분석과 다르게 오염객체로부터 값을 바로 전달 받지 않는다. [그림 4]와 같이 암시적 오염 분석은 변수 a 오염객체가 제어 흐름에 영향을 미쳐 변수 b의 값에 영향을 미칠 때 변수 b도 오염객체가 된다.

```
int a, b;
```

```
a = 2;
```

```
if (a == 3)  
    b = 0;  
else  
    b = 1;
```

[그림 4] 암시적 오염 분석 프로그램 예시

소스코드 기반의 오염 분석은 변수의 정의가 명확하여 명시적 오염 분석과 암시적 오염 분석 두 개를 모두 사용할 수 있다. [그림 5]에서 보듯이 외부 사용자 입력 값을 파일(test.txt)로 정의한다. 소스에서는 변수 pf, buf, ret 모두 사용자 입력 값에 의해 전달 받기 때문에 명시적 오염 분석을 수행한다.

```
int main()
{
    FILE *pf;
    char buf[0x20];
    int ret;

    pf = fopen("test.txt", "rb");
    if (pf) {
        ret = fread(buf, 1, 0x10, pf);
        if (ret)
            test(buf);
    }
}
```

[그림 5] 오염 분석 대상 프로그램 main 함수 소스

[그림 6]은 [그림 5]의 소스 코드를 컴파일한 main 함수의 어셈블리 코드이다. 바이너리 코드 분석은 명시적 오염 분석만으로 가능한 소스 코드와는 다르게 암시적 오염 분석 부분도 나타난다. 11번째 어셈블리 코드는 fopen 함수의 반환 값에 의한 분기 명령어이다. 사용자의 입력 값에 의한 첫 번째 분기 명령어이다. 암시적 오염 분석도 허용할 경우 fread 함수를 호출하기 위해 세 번째 인자를 설정하는 14번째 어셈블리 코드 mov dword ptr [esp+8], 10h에서 피

연산자 [esp+8] 주소도 오염객체가 된다. 이와 같이 바이너리 코드 분석에서 암시적 오염 분석도 허용하게 되면 대부분의 명령어에서 피연산자는 오염객체가 된다. 따라서 분석이 불가능 할 정도의 오염객체가 생성되기 때문에 바이너리 기반 분석에서 암시적 오염 분석은 사용하지 않는다.

```

<main>:
(01) push    ebp
(02) mov     ebp, esp
(03) and     esp, 0FFFFFFF0h
(04) sub     esp, 40h
(05) call    ___main
(06) mov     dword ptr [esp+4], offset aRb
(07) mov     dword ptr [esp], offset aTestTxt
(08) call    _fopen
(09) mov     [esp+3Ch], eax
(10) cmp     dword ptr [esp+3Ch], 0
(11) jz      short loc_401608
(12) mov     eax, [esp+3Ch]
(13) mov     [esp+0Ch], eax
(14) mov     dword ptr [esp+8], 10h
(15) mov     dword ptr [esp+4], 1
(16) lea    eax, [esp+40h+var_28]
(17) mov     [esp], eax
(18) call    _fread
(19) mov     [esp+38h], eax
(20) cmp     dword ptr [esp+38h], 0
(21) jz      short loc_401608
(22) lea    eax, [esp+40h+var_28]
(23) mov     [esp], eax
(24) call    __Z4testPc
(25) mov     eax, 0
(26) leave
(27) ret

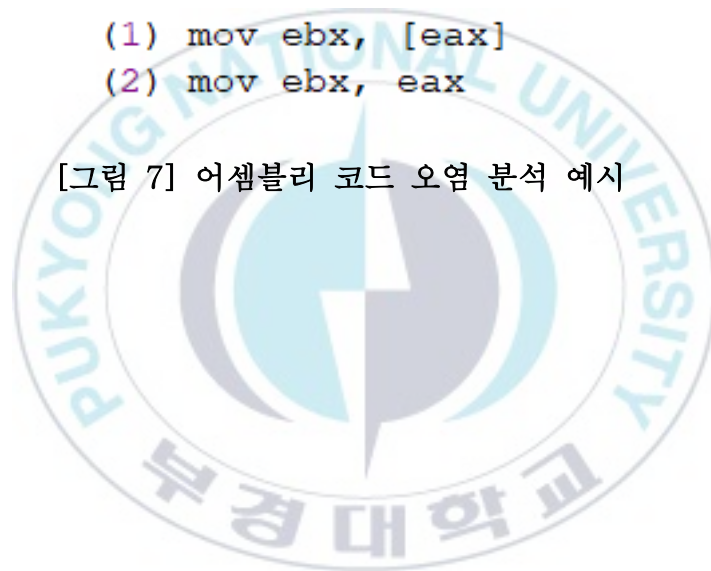
```

[그림 6] 오염 분석 대상 프로그램 main 함수 어셈블리 코드

[그림 7]의 어셈블리 코드 오염 분석 예시로 바이너리 기반의 정확한 명시적 오염 분석에 대해 알아본다. eax 레지스터가 오염객체인 경우 첫 번째 명령어의 피연자인 ebx 레지스터는 오염객체가 되지 않는다. eax 레지스터가 주소를 가리키는 피연산자이기 때문에 eax 레지스터가 가리키고 있는 주소의 값이 오염객체인 경우 ebx는 오염객체가 된다. 두 번째 명령어는 eax 레지스터의 값을 그대로 ebx 레지스터에 전달하기 때문에 ebx 레지스터는 오염객체가 된다.

```
(1) mov ebx, [eax]  
(2) mov ebx, eax
```

[그림 7] 어셈블리 코드 오염 분석 예시



나. Concolic Execution 구성 요소

Concolic Execution은 하나의 분기문 실행/분석마다 두 개의 경로가 생성된다. 따라서 Concolic Execution을 구현하기 위해서는 분기문마다 심볼(Symbol), 경로 조건(Path Constraints), 다음 상태(Next Statement)를 계속 저장해야 한다. 심볼은 사용자가 정의한 입력 값이다. [그림 8]의 test 함수는 [그림 5]의 main 함수가 호출하는 test 함수이다. 따라서 파일(test.txt)의 내용이 사용자 입력 값으로 심볼이 된다. if (buf[0] == 'C' && 코드에서 첫 번째 분기문이 발생하고 경로조건은 첫 번째 분기문이기 때문에 참이다. 다음 상태는 buf[0] == 'C' 와 buf[0] != 'C' 두 개로 나눌 수 있다. buf[0] == 'C' 인 경우 두 번째 분기문은 buf[1] == 'E' 코드다. 두 번째 분기문에서 경로조건은 buf[0] == 'C' 이고 다음 상태는 buf[1] == 'E' 와 buf[1] != 'E' 두 개로 나눌 수 있다. 이와 같이 Concolic Execution 구현을 위해서는 모든 분기문에서 심볼, 경로 조건, 다음 상태가 필요하다.

[그림 8]의 test 함수는 취약점을 내포한 소스코드이다. buf의 시작 2바이트가 "CE" 이고 buf[2]의 값이 0xff이면 signed int 변수로 선언된 str_size의 값은 0xffffffff이 된다. str_size + 1 값을 인자로 하는 malloc 함수 호출 시 정수 오버플로우(Integer Overflow) 취약점 발생, 0 크기만큼 할당된 힙에 0xffffffff 크기를 복사하는 memcpy 함수 호출 시 힙 오버플로우(Heap Overflow) 취약점이 발생한다.

```

void test(char *buf)
{
    int str_size;
    char *p_str;

    if (buf[0] == 'C' &&
        buf[1] == 'E') {

        str_size = (int)buf[2];

        if (str_size < 0x20) {
            p_str = (char *)malloc(str_size + 1);
            memcpy(p_str, &buf[3], str_size);
            p_str[str_size+1] = '\x00';

            printf("Message : %s\n", p_str);
        }
    }
}

```

[그림 8] 취약점이 내포된 프로그램 소스

다. 바이너리 기반 Concolic Execution 구현

[그림 9]는 [그림 8]의 test 함수를 컴파일한 어셈블리 코드이다. 바이너리 기반의 Concolic Execution 구현이기 때문에 명시적 오염 분석만을 사용한다. (07) 어셈블리 코드에서 사용자 입력 값에 의해 첫 번째 분기문이 발생한다. 심볼은 [ebp+arg_0]의 값(buf[0])이고 크기는 1바이트(byte)이다. 경로 조건은 첫 번째 분기문이기 때문에 항상 참이고 다음 상태로 가기 위해 해당 값이 0x43인지 검사한다. 따라서 다음상태는 [ebp+arg_0]의 값이 0x43이 아니면 참이 되어 test 함수 실행을 종료하고, 0x43이면 조건이 거짓이 되어 다음 분기문이 (12) 어셈블리 코드를 분석 하게 된다. 두 번째 분기문의 심볼은 [ebp+arg_0+1]의 값이고 크기는 1바이트이다. 경로 조건은 [ebp+arg_0]의 값이 0x43이고 다음 상태로 가기 위해 [ebp+arg_0+1]의 값이 0x45 인지 검사한다.

바이너리 기반에서도 소스 기반과 같은 방법으로 심볼, 경로 조건, 다음 상태를 계속 저장하면서 Concolic Execution을 구현하면 힙오버플로우가 발생하는 (23) 어셈블리 코드까지 갈 수 있는 사용자 입력 값을 자동으로 생성 할 수 있다.

```

(01) push    ebp
(02) mov     ebp, esp
(03) sub     esp, 28h
(04) mov     eax, [ebp+arg_0]
(05) movzx   eax, byte ptr [eax]
(06) cmp     al, 43h
(07) jnz     short locret_401580
(08) mov     eax, [ebp+arg_0]
(09) add     eax, 1
(10) movzx   eax, byte ptr [eax]
(11) cmp     al, 45h
(12) jnz     short locret_401580
(13) mov     eax, [ebp+arg_0]
(14) add     eax, 2
(15) movzx   eax, byte ptr [eax]
(16) movsx   eax, al
(17) mov     [ebp+var_C], eax
(18) cmp     [ebp+var_C], 1Fh
(19) jg      short locret_401580
(20) mov     eax, [ebp+var_C]
(21) add     eax, 1
(22) mov     [esp], eax
(23) call    _malloc
(24) mov     [ebp+var_10], eax
(25) mov     eax, [ebp+var_C]
(26) mov     edx, [ebp+arg_0]
(27) add     edx, 3
(28) mov     [esp+8], eax
(29) mov     [esp+4], edx
(30) mov     eax, [ebp+var_10]
(31) mov     [esp], eax
(32) call    _memcpy
(33) mov     eax, [ebp+var_C]
(34) lea    edx, [eax+1]
(35) mov     eax, [ebp+var_10]
(36) add     eax, edx
(37) mov     byte ptr [eax], 0
(38) mov     eax, [ebp+var_10]
(39) mov     [esp+4], eax
(40) mov     dword ptr [esp], offset aMessageS
(41) call    _printf
(42) leave
(43) ret

```

[그림 9] 취약점이 내포된 프로그램 어셈블리 코드

IV. 실험

1. 실험 환경

본 장에서는 성능 개선을 위한 바이너리 기반 자동 취약점 분석 도구를 구현하고 실험 결과를 확인한다. 실험 환경은 다음과 같다.

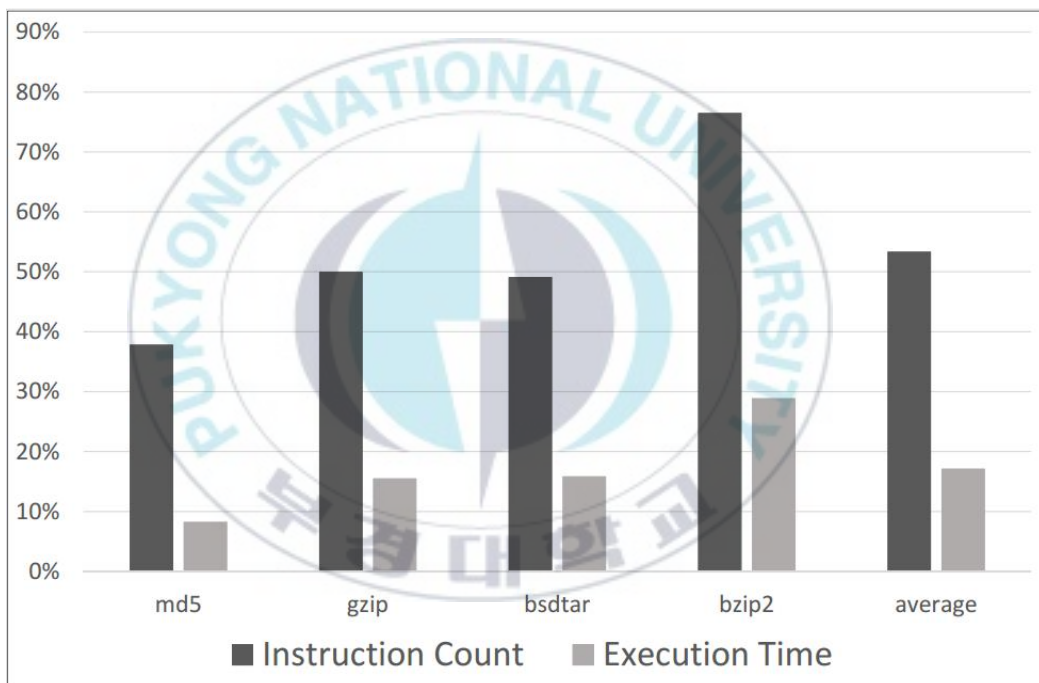
- Intel(R) Core(TM) i7-4980HQ CPU
@ 2.80GHz (8 CPUs), ~2.8GHz
- 16384MB RAM
- 1024GB SSD disk
- Windows 10 Enterprise 64bit

2. 오염블록 구현 및 실험 결과

실험 대상 소프트웨어는 해시 프로그램 md5[18]와 압축 프로그램 gzip[19], bsdtar[20], bzip2[21] 이다. 실험 입력 값으로는 누구나 손쉽게 구할 수 있는 윈도우 실행파일, 리눅스 실행파일, 이미지파일 그리고 미디어파일이다. 10개의 파일을 입력 값으로 각각의 파일마다 10번의 실험을 수행해 그 결과의 평균을 사용한다.

[그림 10]은 오염블록을 적용한 경우, 각 실험 대상 소프트웨어

별로 동적 오염 분석도구가 분석해야 하는 명령어 수와 그에 따른 분석 시간이 적용하지 않은 경우에 대해 얼마나 차이가 나는지를 나타낸다. 오염블록 미적용 시 결과 값과 비교하여 보았을 때 분석하는 명령어 수와 동적 오염 분석 시간 모두 성능이 향상됨을 확인할 수 있다. 오염블록 적용 시 분석하는 명령어 수는 전체 평균 53%, 동적 오염 분석의 속도는 전체 평균 17% 향상된다. 또한 오염블록으로 처리하는 명령어 수에 비례하여 동적 오염 분석 속도가 개선되는 결과를 확인할 수 있다.

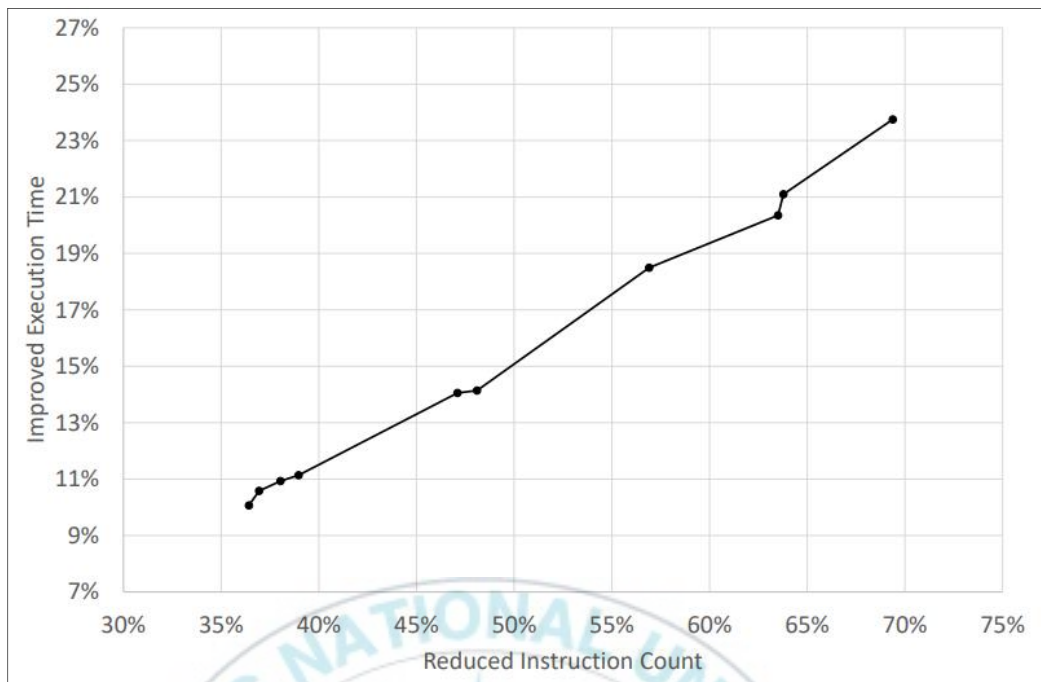


[그림 10] 오염블록을 사용한 성능 개선

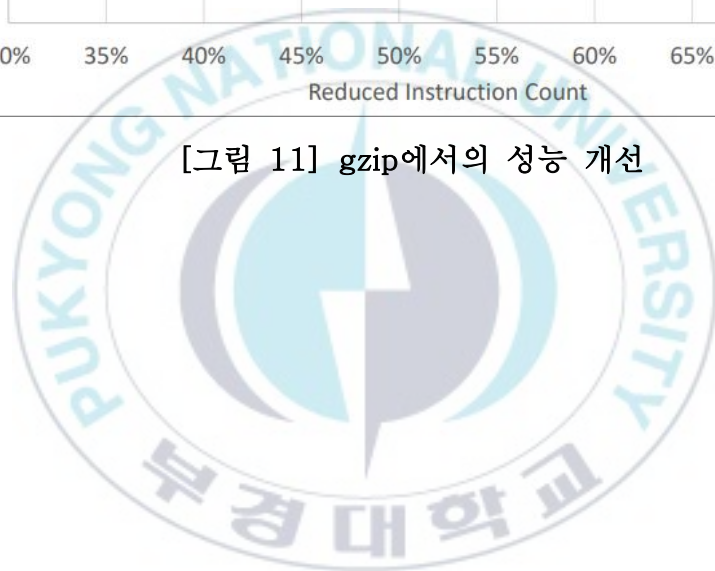
<표 1>은 [그림 10]의 결과 중 gzip 압축 프로그램에 대한 상세 결과로 10개의 입력 파일에 대한 실제 처리 결과를 나타낸다. [그림 11]은 gzip 압축프로그램에 대한 오염블록을 적용한 경우 줄어드는 명령어 분석 개수와 그에 따른 동적 오염 분석의 속도 향상과의 상관 관계를 나타낸다. 실험 대상 소프트웨어는 입력 파일을 제외하고는 이벤트가 발생되지 않는 소프트웨어이기 때문에 분석되는 명령어의 수는 항상 동일하다. 시간의 단위는 1/1000초(milliseconds)이며 평균값이다. 각각의 입력 파일 실험결과 동적 오염 분석 시간의 최소 값과 최대값의 차이가 무의미하여 평균값을 사용한다. [그림 11]을 통해 오염블록으로 처리하는 명령어 수의 감소와 동적 오염 분석 속도 개선은 근사하게 비례한다는 결과를 확인 할 수 있다.

<표 1> gzip 실험 결과

	Instruction Count		Execution Time(msec)	
	Native	Taint Block	Native	Taint Block
1	10866685	4682817	30423	24797
2	53466731	19509843	151996	121062
3	22396401	8111450	63054	49750
4	22769381	6972463	62979	48021
5	124434582	79116274	341575	307182
6	38492776	23494920	104932	93246
7	69683055	36164486	195820	168137
8	16697978	8832828	46589	40042
9	82491708	52014017	219670	196431
10	60194383	37299266	162054	144338



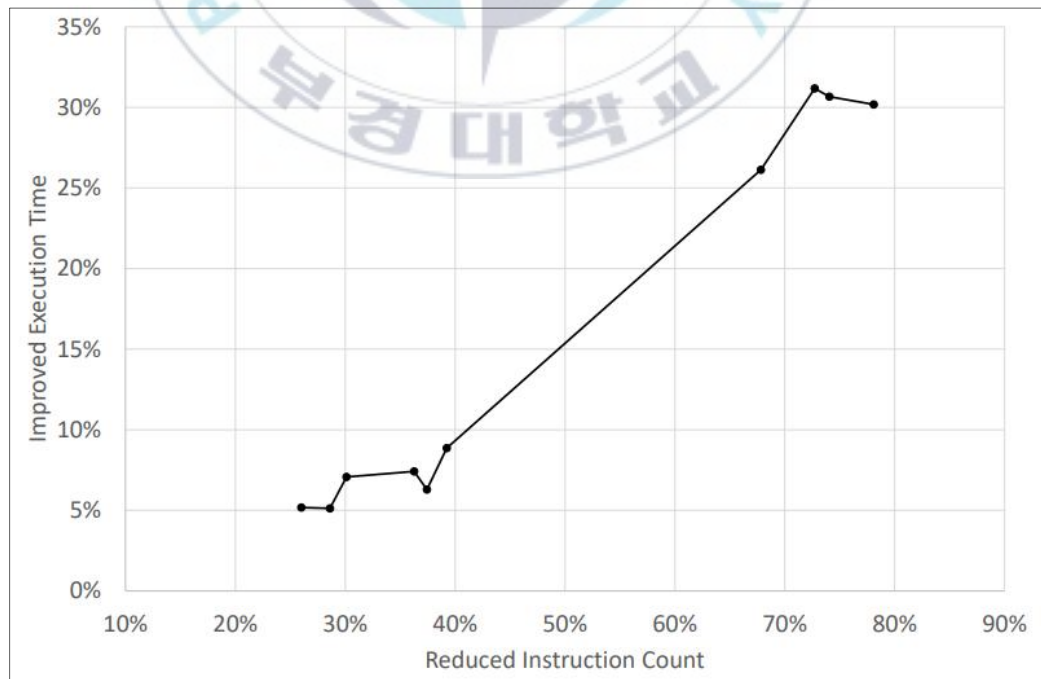
[그림 11] gzip에서의 성능 개선



<표 2>와 [그림 12]는 bsdtar 압축 프로그램에 대한 동적 오염 분석을 수행한 상세 결과 값이다.

<표 2> bsdtar 실험 결과

	Instruction Count		Execution Time(msec)	
	Native	Taint Block	Native	Taint Block
1	14280267	4594914	31860	23537
2	72535317	19783986	173704	119552
3	31217552	8098486	73621	51045
4	31702649	6944870	73235	51134
5	135234107	84613494	302658	283618
6	39795501	27812211	98392	91431
7	78177198	57847122	201592	191153
8	18645880	13312508	44968	42664
9	87869036	55993688	193304	178981
10	65834542	40004863	142760	130115

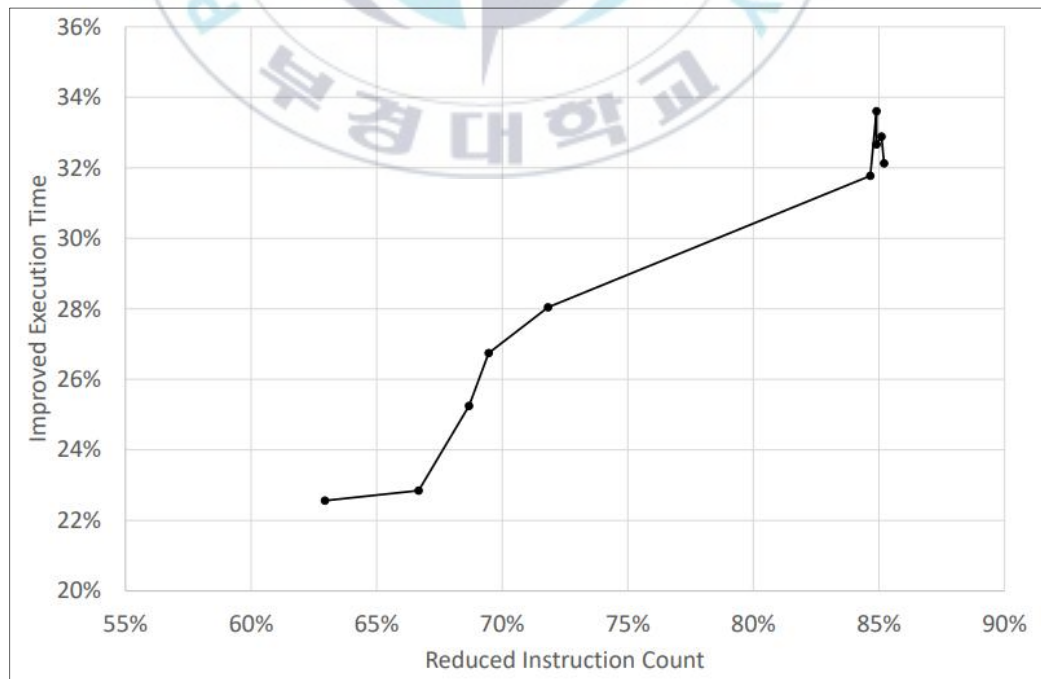


[그림 12] bsdtar에서의 성능 개선

<표 3>과 [그림 13]은 bzip2 압축 프로그램에 대한 동적 오염 분석을 수행한 상세 결과 값이다.

<표 3> bzip2 실험 결과

	Instruction Count		Execution Time(msec)	
	Native	Taint Block	Native	Taint Block
1	32286667	9859509	55017	40301
2	133140351	41703053	260690	194876
3	67638730	19057435	130303	93759
4	47247817	17510619	92581	71693
5	327375402	109118858	618476	477198
6	221675272	33031859	318547	213790
7	439667761	67465924	654249	446373
8	111560302	16843305	156033	103598
9	444652444	65796747	658784	447140
10	312766787	47220967	453951	305676

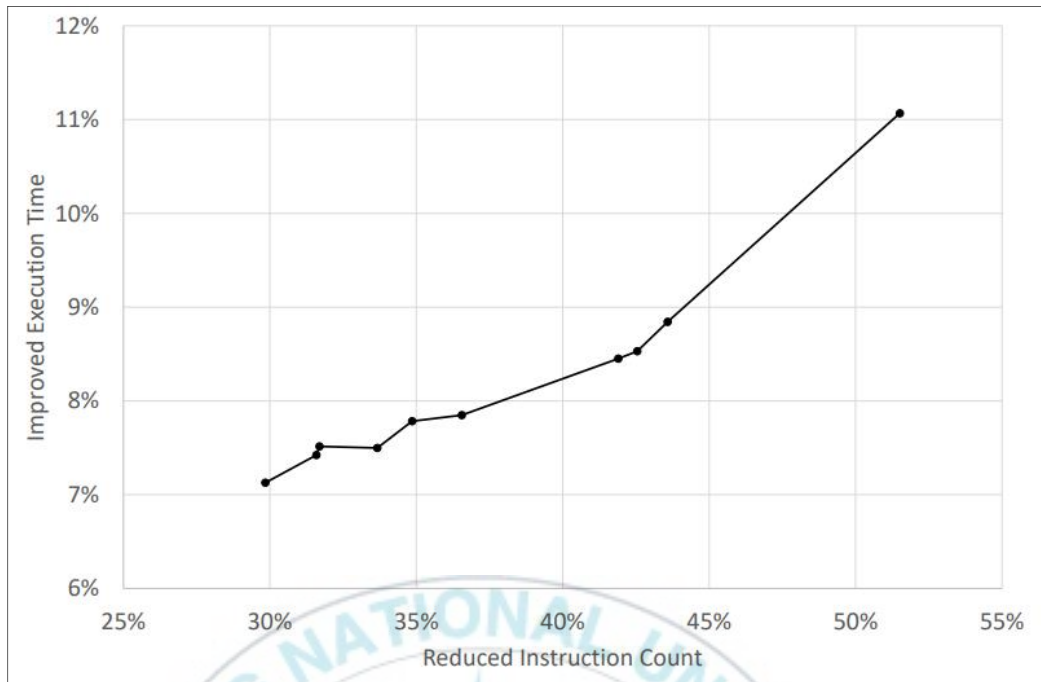


[그림 13] bzip2에서의 성능 개선

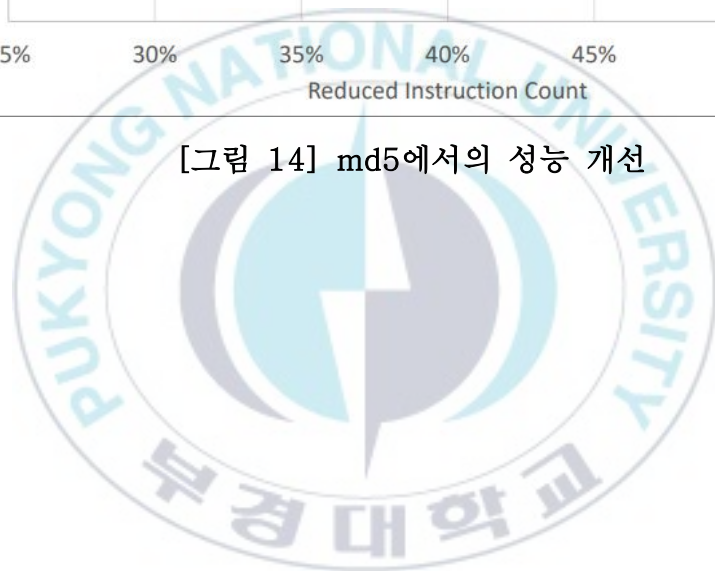
<표 4>와 [그림 14]는 md5 해시 프로그램에 대한 동적 오염 분석을 수행한 상세 결과 값이다. 이 프로그램의 경우 UPX로 패킹되어 있어 정적 오염 분석은 불가능하지만 우리가 제안한 기법은 동적 오염 기법을 기반으로 하기 때문에 다른 프로그램들과 동일하게 오염블록 기법이 적용될 수 있어 처리되는 명령어 수의 감소에 비례하여 동적 오염 분석의 속도가 개선되는 것을 확인할 수 있다.

<표 4> md5 실험 결과

	Instruction Count		Execution Time(msec)	
	Native	Taint Block	Native	Taint Block
1	2014221	976685	7665	6817
2	5770906	3759352	29526	27228
3	3099602	1780952	13939	12750
4	3224980	1873836	14660	13422
5	13087157	9181129	71968	66839
6	4844928	3074033	24168	22271
7	9090368	6219071	48900	45270
8	2917941	1646288	12934	11790
9	8914504	6089414	47862	44265
10	6645266	4407991	34668	32068



[그림 14] md5에서의 성능 개선

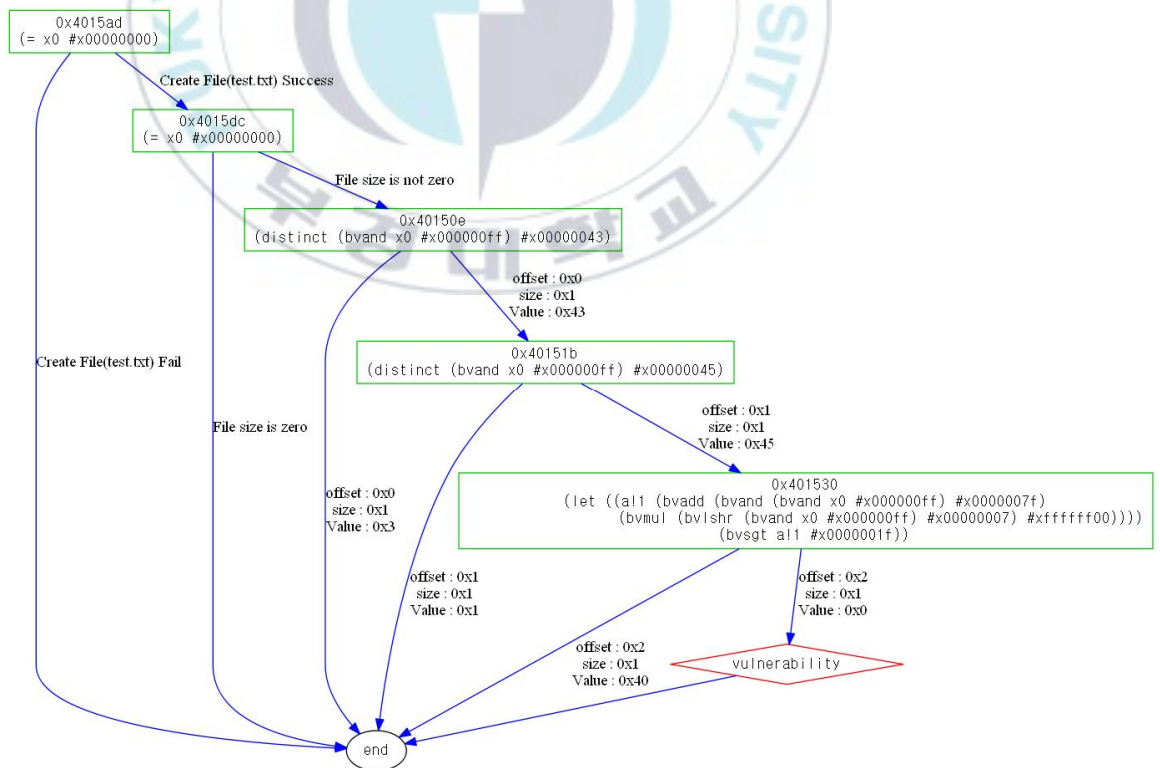


3. Concolic Execution 구현 및 실험 결과

실험 대상 소프트웨어는 [그림 5]의 main 함수와 [그림 8]의 test 함수를 컴파일한 바이너리이다. 입력값은 컴파일한 바이너리와 사용자 정의 입력 값 파일(test.txt) 두 개이다.

[그림 15]는 자동 취약점 분석 도구에서 Concolic Execution을 이용하여 다중 경로 탐색을 구현한 결과이다. 노드는 분기문이 발생하는 주소와 경로 조건이다. 노드와 노드를 연결하는 링크에는 다음 상태로 가기위한 정보를 나타냈다.

[그림 15]의 결과와 같이 두 개의 입력 값 지정만으로 취약점이 발생할 수 있는 입력 값을 자동으로 생성할 수 있다.



[그림 15] 바이너리 기반 Concolic Execution 구현 결과

V. 결 론

본 논문에서는 바이너리 기반 자동 취약점 분석 도구에서 동적 오염 분석의 속도 저하 문제점을 보완하고 최소한의 설정으로 동작하는 두 가지 기법을 제안하였다.

첫 번째로 제안한 오염블록 기법은 반복적으로 실행되는 코드를 블록으로 묶어 최적화된 코드를 생성해 추후 동적 오염 분석 시 반복되는 실행 코드와 오염객체를 저장하는 코드는 분석을 하지 않도록 한다. 실험 결과 DBI 도구와 동적 오염 분석에서 분석 과정이 한번만 수행되어 동적 오염 분석의 성능이 향상되는 것을 확인할 수 있었다. 또한 오염블록으로 수행되는 명령어 수에 비례하여 성능이 개선됨을 실험으로 확인하였다.

두 번째는 사용자 정의 입력 값만 심볼로 설정, 명시적 오염 분석만을 사용하여 Concolic Execution을 구현하는 기법이다. 실험 결과 바이너리 코드 사전 분석 없이 분석 대상 프로그램과 사용자 입력 값 두 개 설정만으로 다중 경로를 탐색하여 취약점을 탐지하는 것을 실험으로 확인하였다.

본 연구에서는 동적 오염 분석의 성능 향상을 위한 코드 최적화만 적용하였다. 향후 실행 및 오염객체를 저장하는 코드 최적화뿐만 아니라 오염객체간의 전파에 대해서도 코드 최적화를 적용하는 방안도 고려할 수 있을 것으로 판단된다. 또한 오염블록으로 처리하는 명령어의 수에 비례하여 성능이 개선되므로 더 많은 영역을 오염블록에 추가 할 수 있도록 코드 영역을 정의하는 연구도 같이 진행하여야 할 것으로 사료된다.

참고 문헌

- [1] J. Newsome and D. Song. “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.” In Proceedings of NDSS ’ 05, San Diego, (2005).
- [2] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. “Still: Exploit code detection via static taint and initialization analyses.” In Proceedings of the Annual Computer Security Applications Conference (ACSAC), (2008).
- [3] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. “ShadowReplica: Efficient parallelization of dynamic data flow tracking,” In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, ser. CCS ’ 13, (2013).
- [4] K. Sen, “Concolic Testing,” In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, (2007)
- [5] “angr” [Online]. Available: <https://github.com/angr>
- [6] B. P. Miller, L. Fredriksen, and S. Bryan. “An empirical study of the reliability of UNIX utilities.” Communications of the ACM, (1990).
- [7] V. Ganesh, T. Leek, and M. Rinard. “Taint-based directed whitebox fuzzing.” In Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, (2009).

- [8] T. Wang, T. Wei, G. Gu, and W. Zou, “Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection,” In Proceedings of the IEEE Symposium on Security and Privacy, (2010).
- [9] N. Nethercote and J. Seward. “Valgrind: A program supervision framework.” In Proceedings of the Third Workshop on Runtime Verification (RV’ 03), (2003).
- [10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. “Pin: Building customized program analysis tools with dynamic instrumentation.” In Proceedings of PLDI, (2005).
- [11] D. Bruening, T. Garnett, and S. Amarasinghe. “An infrastructure for adaptive dynamic optimization.” In Proceedings of CGO’ 03, (2003).
- [12] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. “Parallelizing security checks on commodity hardware.” In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, (2008).
- [13] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. “Taintpipe: pipelined symbolic taint analysis.” In Proceedings of the 24th USENIX Security Symposium (Security), (2015).
- [14] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks.” In

- Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, (2006).
- [15] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. “BitBlaze: A new approach to computer security via binary analysis.” In Proceedings of the 4th International Conference on Information Systems Security, ICISS, (2008).
- [16] “Triton” [Online]. Available:
<https://github.com/JonathanSalwan/Triton>
- [17] L. M. de Moura and N. Bjørner. “Z3: An efficient SMT solver.” In Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems, (2008).
- [18] “md5sums” [Online]. Available:
<http://www.pc-tools.net/files/win32/freeware/md5sums-1.2.zip>
- [19] “gzip” [Online]. Available:
<http://gnuwin32.sourceforge.net/downloads/gzip-bin-zip.php>
- [20] “bsdtar” [Online]. Available:
<http://downloads.sourceforge.net/gnuwin32/libarchive-2.4.12-1-bin.zip>
- [21] “bzip2” [Online]. Available:
<http://gnuwin32.sourceforge.net/downloads/bzip2-bin-zip.php>

감사의 글

이 글을 통해 부족한 저의 석사과정을 잘 마무리될 수 있게 도움을 주신 모든 분들께 감사의 인사를 드리고자 합니다.

학교에서 아직 무엇을 해야 할지 모르는 저를 연구실에 받아 주시고 보안이라는 연구 방향을 제시해 주신 이경현 교수님께 가장 먼저 감사를 드립니다. 교수님을 만나지 못했더라면 지금의 저는 없었을 것 같습니다. 특히 연구에 대한 열정을 교수님으로부터 많이 배웠습니다. 감사합니다.

바쁘신 와중에도 불구하고 심사를 맡아 주시고 지도와 가르침을 주신 김창수 교수님과 신상욱 교수님도 감사드립니다.

학교 연구실 일에 도움을 드리지도 못했는데 항상 체계 도움을 주신 연구실 분들께도 감사드립니다.

직장을 다니면서도 졸업을 할 수 있게 배려를 많이 해주신 연구소 보안팀 분들께도 감사드립니다.

항상 제 옆을 지켜주는 아내와 귀여운 공주 박설 사랑해.