Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.27, NO.6, Dec. 2017 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2017.27.6.1331

바이너리 분석도구 효율성 평가를 위한 Instrumentation 성능 측정기법*

이 민 수,† 이 제 현, 김 호 빈, 류 찬 호‡
한국과학기술원 사이버보안연구센터

Instrumentation Performance Measurement Technique for Evaluating Efficiency of Binary Analysis Tools*

Minsu Lee,† Jehyun Lee, Hobin Kim, Chanho Ryu‡
Cyber Security Research Center,
Korea Advanced Institute of Science and Technology

요 약

바이너리 instrumentation 기법은 소스코드가 공개되어 있지 않은 프로그램을 모니터링하거나 디버깅을 통해 오류를 진단, 또는 메모리 정보획득 등을 위해 개발, 발전되어왔다. 그러나 instrumentation 기법을 사용한 바이너리 분석기법에 관한 연구들은 주로 그 활용방법과 누락없는 정확한 분석에 집중하고 있으며, 실용적 측면에서 중요한 성능지표인 효율성에 대한 연구는 거의 이뤄지지 않고 있다. 특히 분석도구나 알고리즘의 분석시간을 상호 비교할 수 있는 지표와 방법론이 정립되어있지 않았다. 이 연구는 바이너리 instrumentation 기법의 오버헤드를 측정하여 그 효율성을 비교평가할 수 있는 단위기능과 측정방법론을 제안한다. 또한 제안한 방법을 DynamoRIO와 Pin에 적용하여 성능차를 도출하였다. 분석도구의 효율성 비교결과는 사용목적에 따른 분석도구의 선택기준이 되며, 측정방법론은 기존도구와 앞으로 새롭게 개발될 분석도구들에 대해서도 그 효율성을 검증하는 방법으로 사용될 수 있다.

ABSTRACT

Binary instrumentation has been developed for monitoring and debugging executables without their source codes. Previous efforts on the binary instrumentation are mainly focused on its capability and accuracy, but not on efficiency for practical application. In particular, criteria and measurement methodologies for evaluating and comparing the efficiency of binary investigation tools and algorithms do not estimated yet. In this paper, we propose the instrumentation primitives which are a unit functionality and measurement methodology. Through the empirical experiments by adopting the proposed methodology on DynamoRIO and Pin, we show the feasibility of the proposal.

Keywords: Binary instrumentation, overhead measurement, instrumentation primitives

I. 서 론

Instrumentation 기법은 실행중인 프로그램의 제어흐름을 변경하거나 내부값을 조회, 변경하기 위해 추가적 코드를 삽입하는 기술들을 포괄적으로 일컫는 것으로서 소프트웨어공학 분야에서 프로그램의

Received(05. 04. 2017), Modified(10. 19. 2017), Accepted(10. 20. 2017)
* 본 연구는 미래창조과학부 글로벌사이버보안기술연구 (과제 고유번호: 1711030958) 사업에서 지원하였습니다.
† 주저자, minsulee@kaist.ac.kr
‡ 교신저자, choryu@kaist.ac.kr(Corresponding author)

버그탐지와 최적화의 목적으로 사용되어왔다. 제어흐름의 분석, 관리와 버그탐지는 과거에 비해 프로그램의 코드크기가 커지고 동적제어흐름을 가지거나 더 다양하고 복잡한 기능들을 포함하게 되면서 그 난이도와 복잡성이 증가추세에 있고[1,2,3], 예상치 못한 복합적 원인의 버그가 잠재되어있을 가능성이 증가하면서 그 기술적 중요성이 주목 받고 있다.

Instrumentation 기법을 이용한 디버깅 기술들은 소프트웨어공학적 목적 뿐 아니라 정보보안의 목적으로도 활용되고 있다. 프로그램에 존재하는 버그는 외부에서 공격 가능한 취약점으로 악용(exploitation) 될 가능성이 있어 버그탐지 기법은 프로그램의 보안성 검증과 취약점 탐지를 위한 중요한 기법 중 하나로 이용되고 있다. 대표적으로 오염분석(taint analysis) 기법은 감시하고자 하는 변수와 코드의 위치에 instrumentation 코드를 삽입하여 자료와 제어의 흐름을 감시하여 악성행위나 버그, 취약점을 탐지한다[4].

이와 같이 프로그램의 취약점탐지와 안정성 검증 분야에서 instrumentation 기법이 널리 사용되고 연구되면서 그 정확성과 기능은 지속적으로 발전되어 왔지만, 프로그램의 규모와 복잡성의 증가추세에 대응하기 위한 효율성에 대한 연구는 상대적으로 주목받지 못하였다. 특히 취약점탐지를 위한 기술이나 도구에서는 그 탐지속도의 신속성이 잠재적 피해를 감소시키는데 직접적 영향을 미치기 때문에 완성도에 중점을 둔 활용분야에 비해 더 중요하게 고려되어야 한다. 기존의 instrumentation 기법에 대한 성능분석은 기법의 개발자가 제안하는 기법의 일반적 성능진보성을 보일 목적으로 일부 기능의 성능비교결과를 제시하는데 그쳐있을 뿐이다[7],

효율적인 instrumentation 알고리즘과 분석도구를 개발하고 목적에 맞는 최적의 분석도구를 선별하기 위해서는 그 성능을 정량적으로 측정하고 비교할 수 있는 평가척도와 방법론의 정의가 선행되어야 한다. 이 연구에서 우리는 instrumentation을 통해 획득하는 정보를 기준으로 하여 원시척도(instrumentation primitive)를 정의하고, 이 척도들의 계측성능들의 측정방법론을 제안한다.

우리는 원시척도를 획득하는 기능을 성능측정의 단위기능으로 삼고, 분석대상 바이너리의 수행시간과 분석도구의 instrumentation으로 인해 증가한 수행시간의 편차를 원시척도 별로 측정한다. 원시척도의 구체적인 정의와 취약점분석과의 연관관계는 4장에서 자세히 다룬다.

우리는 제안한 성능측정기법의 타당성과 실용성을 검증하기 위하여 instrumentation을 사용하는 바이너리 분석도구에 측정코드를 삽입하고 ***SPEC CPU2006***[5]에 포함된 바이너리 집합 중 일부를 분석대상으로 하여 그 원시척도를 계측하여 계측에 소모되는 처리시간을 측정하였다. 실험은 CPU2006의 CINT 12개 바이너리를 대상으로 본 논문에서 6개의 원시척도 중 네거티브 4개를 동일한 환경, 동일한 입력 바이너리를 대상으로 계측하였으며, *DynamoRIO*[6]와 *Pin*[7]이 원시척도를 계측할 때 소모되는 시간을 산출하여 비교하였다. 대상 분석도구의 소스코드는 모두 각각의 웹페이지에 공개되어 있으며, 이 논문의 5장에 명시된 측정설계를 참조하여 누구나 이 기법을 재현하고 사용할 수 있도록 하였다. 그 외 측정과정에서 구체적으로 명시되지 않은 사항은 성능측정용 바이너리 집합을 이용하는 일반적 방법론에 따르며, 본 실험에서 사용한 바이너리 집합에 한정되지 않는다.

제안한 원시척도 정의와 성능측정기법은 단위기능별 성능을 분리하여 측정함으로써 어떤 분석도구나 instrumentation 알고리즘이 가진 전체기능들의 평균성능이 특정기능의 성능을 대표하지 못하는 문제점을 해결하였으며, 최적의 알고리즘과 분석도구를 그 사용목적에 따라 비교평가 할 수 있게 한다. 나아가 분석도구와 알고리즘의 연구개발단계에서 그 진보성을 검증하는 비교척도 중 하나로써도 그 활용가치가 있다.

이 연구의 기여점은 다음과 같다.

- 원시척도의 정의: instrumentation을 이용한 취약점분석의 단위기능의 정의
- 원시척도의 계측시간 측정방법 제안: 구현 실례를 포함하여 instrumentation 도구의 비교 가능한 성능측정 방법론 제안
- 실측정 비교결과의 제시: instrumentation 도구에 제안한 방법론을 적용한 비교측정 결과의 제시

II. 배경지식

최근 제안, 사용되고 있는 instrumentation 기법들은 크게 정적바이너리 재기록(static binary rewriting) 기법[8,9,10]과 DBI(Dynamic

Binary Instrumentation)으로 불리는 동적기법[6,7,12]으로 분류해 볼 수 있다.

정적기법들은 바이너리를 디스어셈블(disassembly)하여 계측하고자 하는 구간에 instrumentation 코드를 삽입 또는 정상코드를 instrumentation 코드로 치환 후 실행하는 방식이다. 이 방법은 실행 중에 외부의 간섭 없이 바이너리 내에 이미 기록되어있는 코드가 수행되면서 계측된다는 특징이 있다.

동적기법들은 바이너리의 실행 중 메모리상에서 계측하고자 하는 위치에 동적으로 생성된 instrumentation 코드를 삽입하거나, 계측하고자 하는 코드영역을 instrumentation 코드가 있는 영역으로 복사하고 제어흐름만을 이동시켜 계측하는 방법들을 사용하고, Java와 같이 목적코드를 실행시점에 기계어 코드로 컴파일하는 언어들에서는 JIT(Just-In-Time) 컴파일 시점에 instrumentation 명령을 삽입하는 방법도 사용되고 있다.

이에 더하여 최근에는 하드웨어 자체, 특히 CPU에 실행중인 바이너리에 대한 계측 기능들이 탑재되면서 DBI 코드를 분석대상 바이너리 코드의 실행전후에 삽입하지 않고 CPU에서 제공하는 기능을 이용하여 소프트웨어적 방법보다 더 빠르고 다른 프로세스로부터의 간섭위험이 적은 기법도 등장하였다.

2.1 소프트웨어 기반 Instrumentation 기법

정적 바이너리 재기록방법은 일반적으로 디스어셈블 과정을 거쳐서 특정 명령을 동일한 길이의 `Trampoline' 명령으로 대치하여 instrumentation 코드로 제어흐름을 변경하거나 계측 명령을 직접 삽입하는 방식을 사용한다. 때문에 이 방식은 고정된 길이의 명령집합을 사용하는 체계에서 더 쉽게 사용될 수 있다. 그러나 대다수의 컴퓨터 시스템에서 사용하고 있는 Intel x86, x64 혹은 ARM 32-bit, 64-bit과 같은 가변길이의 명령집합을 사용하는 컴퓨터에서는 trampoline 명령을 넣기가 까다롭다. 그 원인으로는 대치하고자 하는 원본명령이 가변길이인 경우 한 명령 단위의 길이와 값의 용도가 문맥에 따라 서로 달라 본래 동작의 손실 없이 대치하기 어려우며, trampoline 명령이 사용가능한 메모리 주소와 레지스터 등을 코드분석만으로 추측해야하기 때문에 바이너리의 명령들과 제어흐름이 다양해지고 복잡해질수록 그 정확성을 보장하기 어렵다.

DBI는 실행 도중 계측하고자 하는 위치의 제어흐름만을 우회시키거나 JIT 컴파일 시점에 instrumentation 코드를 삽입하는 방식을 사용한다. 때문에 바이너리 파일 자체에 수정을 가하지 않고 메모리상에서 변화되며, 바이너리의 문맥을 분석하는 작업을 수행하지 않아도 되므로 크기가 크고 명령집합이 복잡한 바이너리, 난독화가 적용되어 정적분석이 어려운 바이너리에 대해 분석시간과 정확성 면에서 장점을 가진다. DBI의 잘 알려진 단점으로는 계측을 위한 코드의 추가로 인한 수행시간의 증가와 계측 프로세스와 분석 대상프로세스 간의 잦은 문맥전환(context switching)을 들 수 있다. 이로 인해 대상 바이너리의 난독화정도와 복잡도에 따라 정적, 동적기법의 성능에 크게 차이가 발생할 수 있다. 대표적 정적, DBI 기법에 대한 관련 연구와 도구들은 Table 1에 정리된 바와 같다.

Table 1. Categories on binary instrumentation methodologies

Category	Tools and Related Work
Static Binary Rewriting	Pebil[8], Vulcan[9], DynInst[10], PSI[11]
Dynamic Binary Instrumentation	DynamoRIO[6], Pin[7], Valgrind[12], Detours[13], Dtrace[14], Strata[15]

2.2 하드웨어 기반 Instrumentation 기법

하드웨어 기반의 instrumentation 기법은 보안명령이나 모듈이 내장된 CPU에 의해서 수행된다. 이는 CPU에서 지원하는 디버깅 기능을 활용하는 것으로 LBR(Last Branch Record), BTS(Branch Trace Store), AET(Architecture Event Trace), PT(Processor Tracing) 등을 CPU에 내장된 명령들을 사용하여 분기추적(Branch tracing)을 수행할 수 있다[16,17].

III. 관련 연구

Instrumentation은 소프트웨어공학과 정보보안 분야에 걸쳐 연구되어왔으며 단위기능들의 활용방법이나 용어정의 또한 정착되어있으나, 이 단위기능들을

Table 2. Definition of Instrumentation Primitives

Instrumentation Primitives	Subdivision	Feature Summary
Count	Instruction	Process instruction counting
	Basic block	Process basic block counting
Coverage	Basic block	Unique maximum basic block execution counting
	Branch	Maximum branch traverse counting
	Path	Unqiue maximum execution path traverse counting
Full tracing		Taint Analysis, Symbolic execution.

효율적으로 계정하는 방법이나 그 성능측정문제를 고려한 연구는 소수 존재한다.

바이너리 instrumentation의 효율성을 증진시키기 위한 연구들은 단위기능의 알고리즘을 개선하여 그 속도를 높이고자 했다[8,18]. 이러한 연구들에서는 실제 분석도구들에 그 방법을 적용하여 성능개선을 달성하고 비교치를 제시했으나, 각각의 연구는 개선하고자 하는 단일기능의 성능변화에 집중하였기 때문에 사용하고자 하는 기능별로 최적의 도구가 무엇이며, 도구 간 편차를 비교평가 하는데 그 목적을 두지는 않았다.

단위기능을 수행하는데 소모되는 시간의 측정과 그 단축을 통한 instrumentation 효율성의 증가는 주로 하드웨어를 이용한 instrumentation 방법을 제시한 연구들[19,20,21]에서 진행되어왔다. 하드웨어를 통한 instrumentation은 구현된 모든 단위기능에 대해서 소프트웨어 방식과 비교하여 명확히 가시적인 성능차이를 보였다. 이러한 연구들은 하드웨어를 이용한 기법의 유리함은 명확히 제시하였으나 하드웨어 간의 비교와 하드웨어에서 계측하지 못하는 기능이 조합된 분석 작업의 최적성능 평가 등의 영역에서 추가적 연구필요성이 있다.

IV. Instrumentation 성능 측정을 통한 취약점 탐지기법 효율성 평가

4.1 Instrumentation Primitives 정의

Instrumentation 기법과 구현의 효율성을 평가하기 위해서는 각 기법이 소모하는 시간을 측정할 수 있는 지표가 필요하다. 이 지표는 측정환경과 다른 바이너리, 다른 컴퓨팅 성능을 가진 구동환경에서도 그 상대비교치가 유의미한 참조 값으로 사용될 수 있어야 한다. 이 연구에서는 instrumentation 기법과 바이너리 분석도구들의 효율성을 정확히 분석하기 위해 분석도구를 통해 얻고자 하는 정보 중 기본이 되는 값들을 ***원시척도***(instrumentation primitives)로 정의하고, 이 원시척도를 측정하는 기능을 단위기능으로 정의한다. 단위기능은 바이너리를 대상으로 한 instrumentation을 통하여 계측할 수 있는 가장 단순한 수치부터, 바이너리의 제어흐름을 따라 방문 가능한 모든 명령과 주소들을 추적(tracing)하는 단위의 작업에 걸쳐 정의된다. 단, 레지스터나 주소값의 조회와 같이 분석 과정 중에 단일 기계어명령으로 조회 가능한 수치는 알고리즘이나 분석도구 간에 차이점이 발생할 요소가 매우 적으므로 단위기능이기는 하나 비교를 위한 척도로 적합하지 않다. 원시척도는 Table 2에서 보는 바와 같이 3개의 단위 기능에 대해 6개이다.

단위기능이 원시척도를 일회 구하는데 소모되는 평균시간이 성능의 비교단위가 된다. 척도는 하나의 정수나 실수를 산출하는 척도와 계측과정에서 가변길이의 자료구조형태의 산출 값을 출력하는 척도가 있다. Table 2에 정의된 원시척도 중 Count는 단일값을 산출하며, Coverage는 Tracing을 통해 집합값을 산출한다. Full Tracing은 Coverage들을 모두 계측하지만 한 번의 작업으로 여러 개의 Coverage를 측정하는 작업이므로 다른 단위기능의 조합으로 정의하지 않고 독립적 단위기능으로 정의한다.

특정 수치를 산출하기 위해서 수행하는 작업과 작업과정 중에 부산되는 값을 기록 또는 확인하기 위해 수행하는 작업이 동일한 경우 이는 한 개의 단위기능으로 정의한다. 예를 들어 Branch coverage의 계측은 계측이 종료된 시점까지 방문한 분기의 비율을 0과 1 사이의 실수 값으로 산출하는 작업임과 동시

에 방문한 분기들의 출발지 주소와 목적지 주소를 기록하거나 조회하는 작업이다.

Full tracing은 취약점의 존재여부를 판단할 수 있는 단일한 정량적 수치를 산출하지 않고 여러 개의 단위기능을 동시에 계측하는 척도다. Full tracing은 계측을 위한 계산 작업과 이는 세분화 가능한 단위기능을 개별로 측정하여 조합하지 않고 단일척도로서 정의한다.

Count는 실행중인 프로세스의 가장 최소 단위인 명령(instruction)과 기본블록(basic block)의 수를 세는 것으로, 특정조건을 만족하는 명령이나 기본블록을 세는 것을 포함한다. 바이너리가 실행되는 동안 계측 중인 정수 값만이 메모리 상에 유지되고 다른 명령이나 값을 변경하지 않는 작업을 말한다. 명령의 수를 기록하는 '**Instruction count**'와 한 프로세스의 제어흐름을 이루는 단위인 기본블록의 수를 세는 '**Basic block count**'가 Count에 속한다.

Coverage는 특정 입력, 특정 환경에서 바이너리가 실행되었을 때 바이너리가 가진 코드 중 얼마나 많은 영역을 방문하였는지를 나타내는 척도로, 방문한 기본블록의 수, 방문한 분기(Branch)의 수, 또는 방문한 경로(Path)의 수를 기준으로 하여 측정된다.

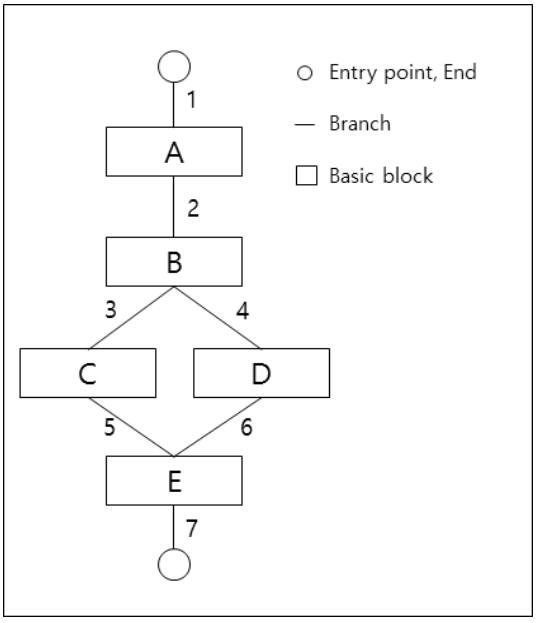

Fig. 1. Example of blocks, branches and paths

기준이 되는 요소에 따라 각각 '**Basic block coverage**', '**Branch coverage**', 그리고 '**Path coverage**'라 한다. Coverage를 측정하기 위해서는 실행 중에 방문한 기본블록, 분기, 경로 들을 그래프의 형태로 기록하며, 이를 CFG(Control Flow Graph)라 한다. Coverage는 전체 기본블록과 분기의 수를 알고 있다면 실제적으로는 방문한 블록들 중 고유한 기본블록의 수를 세는 과정과 같다. 이 과정에서 고유한 색인(ID) 값을 계산하고 기록, 조회하는 연산이 요구된다. 분석대상 프로그램이 적은 수의 기본블록이나 분기를 계속 순회하는지, 다수의 고유한 기본블록과 분기를 거치면서 수행되는지에 따라 count와 그 계정 성능이 크게 달라질 수 있다.

- **Basic block coverage**는 CFG에서 edge를 제외한 기본블록만을 고려하여 기록한다. 기본블록의 첫 시작 주소는 기본블록의 고유 ID로 사용되어, 실행된 기본블록의 고유 ID를 기록하여 coverage를 측정한다.

- **Branch coverage**는 기본블록 coverage와 다르게 기본블록이 아닌 edge를 기준으로 기록한다. 분기는 현재 기본블록의 고유 ID와 현재 기본블록에 도달하기 직전 어떤 기본블록에서 왔는지 주소를 기록하여 coverage를 측정한다.

- **Path coverage는** basic block coverage와 branch coverage를 합친 개념으로 프로세스가 실행될 때 시작부터 끝까지 어떤 고유한 path를 통해 프로세스가 실행되었는지 기록한다. 고유한 path를 기록하기 위해서는 고유한 path의 해시값을 이용하는 방법이 사용된다. Figure. 1의 예를 참조하면, A-B-C-E의 순으로 프로세스 제어흐름이 진행되었다고 가정했을 때, *hash(hash(hash(hash(A) + B) + C) + E)*와 같이 path ABCE의 고유한 해시값을 산출한다.

Full tracing은 instrumentation으로 수행할 수 있는 계측 중 가장 복잡한 작업이라고 볼 수 있다. 대표적으로 오염분석과 바이너리 기호실행(Binary symbolic execution)을 예로 들 수 있

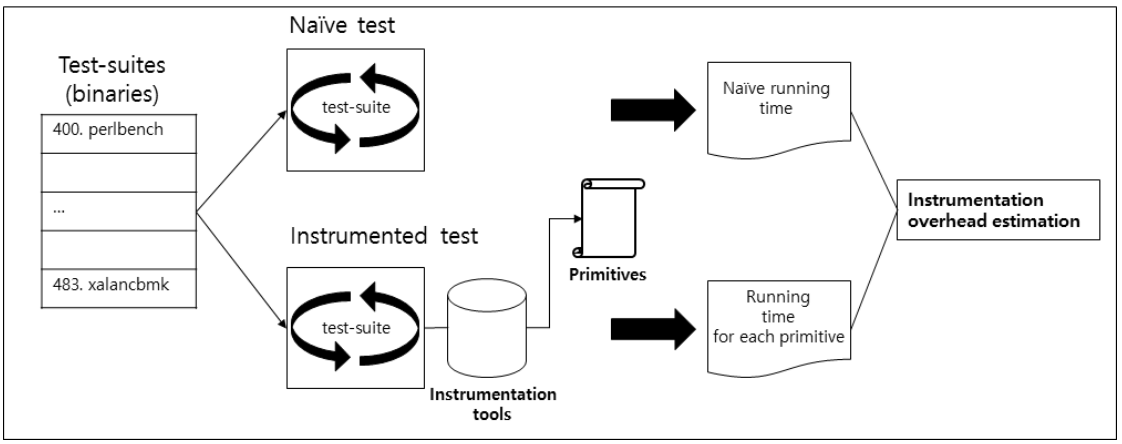

Fig. 2. Overall architecture of instrumentation overhead estimation for primitive measurement

으며, 코드의 CFG에서 자료전파와 변화를 기록하고 추적할 필요가 있는 작업과 특정 기본블록, 분기 또는 코드에 도달하기 위한 입력 값의 조건을 찾는데 사용된다. Full tracing 작업은 가장 큰 단위의 단위기능이며, 분석도구나 알고리즘이 사용하는 자료구조나 Count, Coverage 단위기능의 측정성능에 따라 복합적으로 영향 받는다. Full tracing은 앞선 척도들과 달리 수식적으로 그 방법과 범위가 불분명할 수 있으나, 이 연구에서는 [22]에 정의된 내용을 기준으로 삼는다.

4.2 Instrumentation 성능 측정 방법

Instrumentation 도구의 성능 측정은 Figure. 2에 도시된 설계를 통해 이뤄진다. 성능 측정은 성능을 측정하고자 하는 instrumentation 도구를 선정한 후, 선정한 도구의 소스코드를 직접 수정하거나 제공하는 API를 이용하여 원시척도를 측정하는 기능을 개별 단위기능으로 구현한다.

Instrumentation에 의해서 발생하는 시간소모는 테스트 대상 바이너리의 naive 실행과 바이너리 분석도구를 사용하여 instrumentation된 상태에서의 실행 간의 실행시간 편차를 구함으로써 산출한다. 이 편차의 측정은 각 원시척도를 구하는 작업에 대해 개별 시간을 구하여 단위기능별 성능을 측정한다. 측정용 입력 바이너리의 특성에 따라 원시척도별 계측시간이 달라질 수 있다는 점을 고려하여 입력 바이너리 집합(test suites)은 명령어 조합과 제어흐름 구조의 다양성을 가지도록 서로 다른 용도, 서로 다른 기능들을 가진 바이너리의 집합으로 선정한다.

원시척도들을 측정하는데 있어 제안한 측정방법은 바이너리 분석도구들이 실행흐름이나 입력 값을 제어하지 않는 순수한 instrumentation 기능만을 수행한다고 가정한다. 소프트웨어공학적 관점에서 바이너리의 테스팅을 목적으로 하는 경우 테스팅 도구는 바이너리의 입력집합의 생성을 통해 방문할 기본블록과 선택할 분기를 선별한다. 이 때 최대한 많은 기본블록과 분기를 방문하도록 입력집합을 생성하거나, instrumentation을 통해 제어흐름을 강제적으로 통제한다. 이러한 경우 같은 바이너리를 실행하는 경우에도 그 실행입력집합이나 제어흐름의 변경 전략에 따라 instrumentation 자체에 소요되는 시간과 별개로 전체 바이너리의 수행이 종료되는 시간에 차이가 발생할 가능성이 있다. 이러한 분석도구의 수행 전략의 차이에서 오는 성능차는 대상 바이너리마다 가변적이기 때문에 성능측정의 대상에서 제외하고, 동일한 기본블록, 동일한 분기를 방문할 때 소모되는 시간만을 측정한다. 이 측정결과는 두 개의 서로 다른 분석도구가 동일한 개수의 기본블록을 방문하거나 동일한 길이의 경로를 방문하면서 그 고유정보를 기록하는데 소모되는 시간을 의미한다.

바이너리 분석도구의 성능측정은 참조 값이 되는 Naive 실행을 기준으로 하여 편차 값을 도출하므로 n개의 분석도구의 성능을 구할 때, 참조 값을 포함하여 $n+1$개의 측정값 집합이 필요하다. 테스트는 입력 바이너리 집합에 속해있는 바이너리들 별로 개별 측정한 평균값을 특정 도구의 특정 원시척도의 측정시간이 되고, naive 수행의 측정시간과의 편차가

instrumentation에 의한 오버헤드 값이 된다.

Instrumentation 오버헤드를 측정할 때, 바이너리의 실행 횟수는 측정하고자 하는 원시척도의 수를 p, 입력 바이너리 집합의 크기를 b, 비교대상 분석도구의 수를 n, 측정실험의 반복횟수를 k회라고 할 때, $pbk(n+1)$회가 되고, Naive 실행과의 편차를 구하여 대상 분석도구 마다 p개, 총 pn개의 결과치를 도출한다.

V. 실증실험 및 결과

우리는 제안한 성능측정 방법과 단위기능의 유의미성과 타당성을 검증하기 위하여 실제로 바이너리 분석에 사용되는 instrumentation 도구들을 대상으로 실증실험을 수행하였다. 본 실험의 목적은 다음 두 가지 이다. 첫째로 방법론을 대상도구에 실제적으로 구현하고 적용하는 과정에서 발생가능한 문제점을 확인하고 이를 해결하여 실용가능성을 보인다. 둘째로 단위기능의 계측에 대해 도구간의 성능편차가 존재함을 입증하고, 제어된 입력 바이너리를 대상으로 한 실험결과를 보임으로써 그 일반성을 보인다.

제안한 성능측정 방법과 측정대상 단위기능들이 유의미한 비교지표가 되기 위해서는 instrumentation 도구 간에 가시적인 성능의 편차가 존재해야한다. 만약 단위기능을 계측하는데 소모되는 시간이 대부분의 분석도구 간에 매우 유사하거나, 또는 평균적으로 좋은 성능을 가진 분석도구가 모든 단위기능에 대해 더 빠른 처리성능을 보인다면 단위기능 단위로 분석도구의 성능을 평가하는 것은 비효율적인 측정방법이라 할 수 있다.

제안한 측정방법의 적용은 instrumentation 도구에서 제공하는 instrumentation API를 이용하여 원시척도를 계측하는 코드를 대상 바이너리의 실행코드에 삽입하는 프로그램을 제작하고, 이를 수행시간계측기능을 가진 프로그램을 대상으로 하여 수행한다. 수행시간계측기능을 가진 프로그램은 instrumentation에 의해 소모된 시간을 포함하여 자신의 수행시간을 측정한다. 메모리를 분석하여 코드를 삽입하고 문맥교환을 수행하는 instrumentation 작업 자체와 원시척도 측정에 소모되는 시간을 산출하기 위해 instrumentation 하지 않은 상태에서의 수행시간을 측정하여 비교기준으로 한다. 이를 통해 각 instrumentation 도구가 특정 원시척도를 얼마나 빠르게 측정할 수 있는지 평가한다.

실증실험은 Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz quad-core, 16GB RAM, Linux Kernel 3.19.0-59-generic 운영체제에서 수행하였다.

5.1 평가대상 선정

실증실험 instrumentation 도구는 DBI 도구인 Pin과 DynamoRIO를 대상으로 하였다. 원시척도 계측을 구현하기 위하여 사용자가 정의한 코드를 instrumentation 하는 기능을 제공하면서 관련 연구들에서 사용사례가 충분히 존재하는 도구들을 후보로 하였다. 본 연구는 계측척도에 따라 발생하는 성능편차를 측정, 비교하는 것이 목적이므로, 계측 이외의 요소에서 발생하는 성능편차를 가장 적게 가지는 것으로 알려진 DynamoRIO와 Pin를 비교대상으로 선정하였다[7]. 실제 적용환경에서는 분석하고자 하는 바이너리의 종류나 구동환경에 따라 다양한 분석도구가 사용될 수 있으며, 분석방법론은 단위기능 계측을 수행하는 기능범위를 명확히 판단할 수 있는 모든 도구와 알고리즘에 대해 적용 가능하다.

5.2 성능측정 입력집합

평가대상 분석도구의 단위기능 성능을 측정하기 위해 공통된 입력 바이너리 집합을 사용한다. 이 연구에서는 SPEC CPU2006의 CINT2006 입력 바이너리 집합을 사용하여 DynamoRIO와 Pin의 단위기능 측정성능을 도출, 비교한다.

CINT2006은 12개의 벤치마크 프로그램들을 통해서 수행시간을 측정함으로써 CPU의 정수 처리 성능을 측정하게 된다. SPEC은 각 12개 프로그램의 기초 수행시간을 측정해서 기록한다. 그 후 테스트한 프로그램들의 기초 수행시간과 비교 대상의 수행시간을 비교하여 변화율을 계산한다. 예를 들어 SPECint2006을 이용하여 CPU가 400.perlbench를 2,000초 만에 수행 되고 성능비교대상 컴퓨터에서 측정한 시간이 9,770초일 때, 변화율은 4.885(488.5%)가 된다. 이런 과정을 통해 400.perlbench부터 483.xalancbmk까지의 바이너리를 대상으로 변화율을 측정한다.

Table 3. Binaries for the benchmark test suite from CINT2006

Binary	Language	Category	Description
400.perlbench	C	Programming language	SpamAssassin, MHonArc, specdiff execution using Perl v5.8.7.
401.bzip2	C	Compression	Julian Seward's bzip2 v 1.0.3
403.gcc	C	C compiler	gcc v3.2
429.mcf	C	Combinational optimization	Transit scheduling using network simplex algorithm.
445.gobmk	C	AI: go	AI go game.
456.hmmer	C	Search gene sequence	Analysis of protein sequence using Hidden Markov models.
458.sjeng	C	AI: chess	AI chess game.
462.libquantum	C	Physics quantum computing	Using Shor's polynomial-time factorization algorithm, Quantum Computing simulation
464.h264ref	C	Video compression	H.264/AVC video compression
471.omnetpp	C^{++}	Discrete event simulation	Campus Ethernet modeling simulation using $OMNet^{++}$
473,astar	C^{++}	Path-finding algorithms	Directions algorithm for 2D maps
483.xalancbmk	C^{++}	XML processing	Xalan-C^{++} based XML document parser

5.2.1 Pin의 성능측정방법

Pin은 Intel에서 제작한 DBI 도구로서 리눅스 혹은 윈도우 환경에서 어플리케이션을 분석하는 데 사용된다. 동적으로 instrumentation이 동작하여 바이너리 파일을 실행 중에 분석한다. 그러므로 instrumentation 코드를 삽입하여 재컴파일하거나 소스코드가 없이도 바이너리 파일을 분석할 수 있다. Pin API를 이용해 유저가 원하는 새로운 도구를 만드는 것도 가능하다. 우리는 이 PinAPI를 이용하여 단위기능을 수행하고 소모시간을 측정하는 도구를 제작하여 이로부터 Pin의 단위기능 수행성능을 측정한다. Instrumentation 성능 측정 실험을 위하여 Pin 3.0 버전을 사용하였다.

Count: count 단위기능은 PinAPI를 사용하여 구현하였다. 구현과정에서 추가적으로 고려해야할 사항은 이 도구는 멀티 쓰레드를 사용해서 동작하는 프로그램에서도 그 계측이 가능해야 한다는 점이다. Count를 계측하기 위해서는 프로세스 실행 타임에 개별 명령문(instruction)이나 기본블록이 실행 될 때마다 기록치를 증가 시키는 간단한 방식을 사용한다. instruction count와 basic block count의 구현은 각각 Figure . 3과 Figure . 4와 같다.

- **Instruction count의 구현**: Pin은 VOID Trace라는 사용자정의 함수를 호출하여 instrumentation을 실행한다. Trace 함수는 기본블록 단위로 반복문을 수행하여 바이너리의 첫 기본블록부터 마지막 기본블록 까지 순회한다. 각 기본블록을 방문할 때마다 *BBL_InsertCall* 함수를 이용하여 명령문의 숫자를 세는 *docount*를 실행한다. *BBL_InsertCall* 함수를 호출할 때 *docount* 함수로 *BBL_NumIns(bbl)* 함수의 반환값인 명령문의 수를 넘겨준다. 그 후, *docount*는 전달 받은 기본블록 내의 명령문의 수를 *_count* 변수에 누적한다. 앞서 언급한 것과 같이 SPEC2006에 있는 테스트 프로그램 중에는 다중 쓰레딩 환경으로 구현된 프로그램도 있기 때문에 *docount* 함수가 *_count* 값을 각 쓰레드에서 개별적으로 계산해 주어야한다. 이를 위해 *get_tls* 함수를 이용하여 각 쓰레드의 *tls* 안에서 *_count*가 더해지도록 쓰레드의 ID를 얻어 각기 명령문의 수를 계산한다.

- **Basic block count의 구현**: basic black count의 경우 instruction count를 구하는 것과 크게 다르지 않다. 전체적인 작동 방식은 같지만, Trace 함수에서 *docount* 함수로 인자를 넘겨주지 않고 기본블록이 실행될 때마다 *_count* 변수를 증가시키는 방법으로 기본블록의 수를 계측한다.

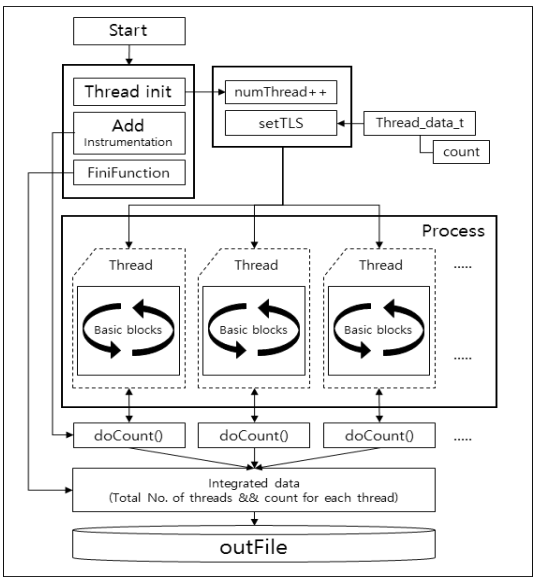

Fig. 3. Instrumentation design for **count primitives** measurement

Coverage의 계측은 바이너리가 실행되는 과정에서 계측하고자 하는 대상을 방문할 때마다 고유하게 인식할 수 있는 ID를 부여하고 중복되지 않은 ID만을 기록한 뒤 프로세스가 종료될 때 그 수를 산정하는 방식으로 구현한다.

- **Basic block coverage**는 프로세스가 실행되는 동안 방문한 고유한 기본블록 수를 구한다. 이를 위해서는 기본블록의 고유 ID를 생성하고 중복을 조회한 뒤 저장한다. 이 실험에서는 기본블록의 처음 시작 명령의 메모리 주소를 고유 ID로 사용하지만, 고유한 값을 부여하는 어떤 함수도 사용될 수 있다. Figure . 4에 도시된 구조와 각 쓰레드의 매 기본블록마다 사용자 지정함수 *getaddr()* 가 호출되어 기본블록의 시작주소를 가져오는데, 이는 Pin의 API 함수 *bblset.insert()*를 사용하여 얻는다. 이렇게 수집된 ID는 각 쓰레드별로 할당된 쓰레드정보 구조체 *tls*에 수집되었다가 모든 쓰레드가 종료되고 프로세스가 종료되기 직전에 중복을 허용하지 않는 구조체에 삽입하여 그 원소수를 측정함으로써 전체 기본블록 수 대비 방문한 기본블록의 비율을 산출한다.

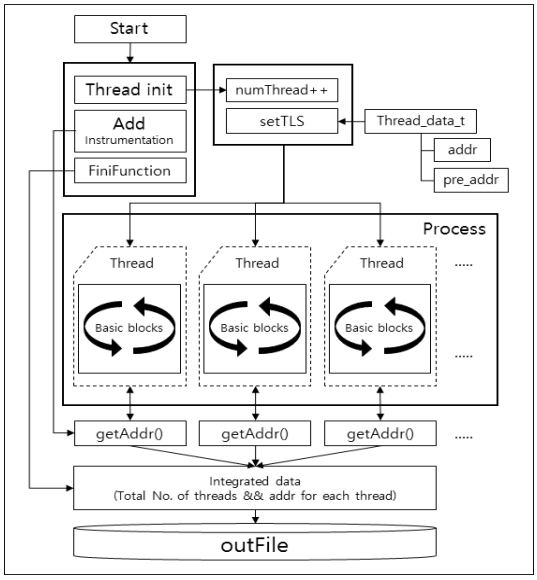

Fig. 4. Instrumentation design for **block and branch coverage** measurement

- **Branch coverage**는 basic block coverage를 구하는 과정에 더하여 기본블록의 고유 ID를 기록할 때 현재 방문중인 기본블록에 도달하기 직전 기본블록의 ID를 *pre_addr* 변수에 기록하는 부분이 추가된다. 전체적인 과정은 사용자 정의함수인 *getaddr()* 함수를 호출하여 쓰레드 정보 구조체인 *tls*에 현재 실행중인 기본블록의 ID *addr*와 이전에 저장해둔 기본블록의 ID *pre_addr*를 순서쌍으로 저장한다. 마지막으로 프로세스가 종료될 때 모든 쓰레드의 *tls*가 가지고 있는 순서쌍의 중복을 허용하지 않은 자료구조에 취합하여 실행중 방문한 고유한 분기의 수를 측정한다.
- **Path coverage**를 구하기 위해서는 기본블록이나 분기와 마찬가지로 path마다 고유한 ID를 부여해야 한다. Figure . 5는 path

coverage의 측정구조의 도식을 나타낸다. Path의 ID을 구하기 위해 path 상에 있는 기본 블록들의 순열에 해시값을 사용한다. path의 해시값은 3.1절의 설계에서 제시된 것과 같은 누적해시 방법을 사용하여 직전에 방문한 기본블록에서 구했던 해시값에 현재 방문한 기본블록의 ID를 더하여 해시값을 갱신함으로써 path가 종료되었을 때 path의 최종 해시값을 가지도록 하는 방식으로 계산한다. 고정된 크기의 정수값을 증가시키거나 매우 단순한 해시를 사용하여 계측코드의 in-lining이 가능한 Figure . 4의 구조와 달리, 전역 기억장소에 대한 저장, 갱신을 수행하고 해시충돌을 최소화 하는 해시함수를 사용하여야만 하는 Figure . 5의 path coverage의 측정은 계측과정에서의 외부함수 호출과 점진적으로 증가하는 저장 공간 사용을 동반한다. Instrumentation 과정이 효율적이더라도 계측코드의 수행 후 다시 원래 제어흐름으로 돌아오는 과정이 비효율적이거나, 계측코드에 대한 별도의 최적화를 수행하지 않는 경우, 계측코드에서 사용한 저장 공간 영역이 빈번히 해제, 재할당 되거나, 복사되는 구조를 가진 경우 계측코드가 복잡해 질수록 도구의 성능은 저하되게 된다. 해시함수는 조합 가능한 path의 수를 고려하여 해시충돌이 발생하지 않는 어떤 해시함수도 사용가능하고, 이 논문의 실증실험에서는 Jenkins hash[23]를 이용해서 해시값을 구하였다.

5.2.2 DynamoRIO의 성능측정방법

DynamoRIO는 바이너리 최적화 시스템으로 시작된 instrumentation 도구로서 현재는 보안, 디버깅 등등의 용도로 더 많이 사용된다. 처음 시작은 Hewlett-Packard Dynamo 최적화 시스템과 the Runtime Introspection 그리고 MIT의 최적화(RIO) 연구그룹이 합쳐지면서 DynamoRIO로 발전되었다. 그 후에는 프로그램 분석 및 이해, 프로파일링, instrumentation, 최적화, translation, 등등의 다양한 분석을 지원하는 프로젝트로 확장되었다. DynamoRIO도 Pin과 마찬가지로 자체 API를 이용해 DynamoRIO 사용자가 도구를 제작할 수 있다. 본 실험을 위하여 DynamoRIO 6.1.1 버전을 사용하였다.

DynamoRIO의 성능측정은 Pin과 마찬가지로 DynamoRIO API를 이용하여 측정도구를 구현하였다. Pin과 상이한 점은 Pin이 모든 쓰레드의 instrumentation 과정이 종료되면 *Fini* 함수에서 쓰레드 별 데이터를 처리하는데 반해, DynamoRIO는 각각의 쓰레드가 종료되면 그 즉시 쓰레드의 *exit* 함수를 호출하여 쓰레드 정보를 우선 처리하고, 모든 쓰레드의 작업이 종료되었을 때 *event_exit()* 함수를 호출하여 instrumentation을 종료한다.

Count: DynamoRIO는 기본블록의 실행마다 사용자가 API를 이용하여 등록한 callback 함수가 실행되는 방식을 가지고 있다. 따라서 프로세스 시작 지점에서 count에 사용할 함수를 등록하여 그 내부에서 instruction 또는 기본블록의 수를 저장하고 있는 변수를 증가시키는 방식으로 측정한다.

- **Instruction count**는 기본블록 방문 후에 호출된 callback 함수 내부에서 방문한 기본블록이 가지고 있는 instruction의 수를 측정하여 이를 다시 변수에 누적시키는 방식으로 측정한다. 이 값들은 쓰레드 마다 개별적으로 관리되므로 쓰레드가 종료될 때 이를 파일로 출력하고,

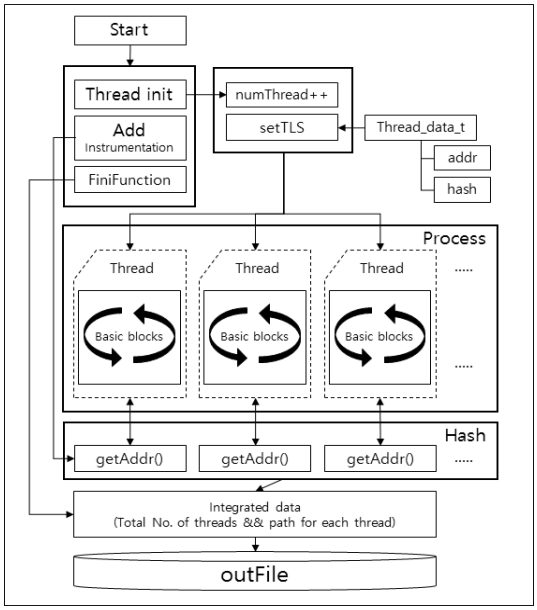

Fig. 5. Instrumentation design for **path coverage** primitive measurement

프로세스가 종료될 때 누적된 파일들을 읽어 합산한다.

- **Basic block count**는 DynamoRIO에서 기본블록이 방문되었을 때 마다 사용자가 쓰레드 시작 시에 API 통해 지정한 callback 함수가 호출되는 구조를 이용한다. Callback 함수 내부에 *count* 변수를 두어 호출 횟수를 증가시켜 방문한 기본블록의 수를 산정한다. DynamoRIO에서는 개별 쓰레드가 종료될 때 자신이 가진 정보를 모두 초기화한 후 개별종료 되므로, 쓰레드가 종료될 때 호출되는 callback 함수에서 쓰레드별 수치를 외부 파일에 기록한 후 프로세스가 종료될 때 기록들을 취합하는 방식으로 산정한다.

Coverage의 도출은 기본블록의 주소를 ID로 하여 방문한 기본블록의 고유한 수를 세거나, 직전에 방문한 기본블록의 ID와 순서쌍을 구성하여 고유한 분기의 수를 계산한다는 점에서는 Pin과 동일하나, 기본블록을 능동적으로 순회하는 것이 아닌 방문 시 callback 함수가 호출된다는 점이 다르다. Callback 함수 내부에서 기본블록의 시작주소를 API를 이용하여 획득할 필요 없이 함수의 입력 인자 값으로 시작주소가 전달된다는 점에서도 차이가 있다.

- **Basic block coverage**는 DynamoRIO의 *dr_insert_clean_call()* API함수를 이용하여 기본블록이 방문될 때 마다 호출될 callback 함수를 등록하고, 등록한 함수가 호출될 때 인자로 전달되는 *void *tag* 값을 방문한 기본블록의 ID로 하여 쓰레드마다 저장한다. tag 변수는 방문한 기본블록의 시작주소를 전달한다.

- **Branch coverage**는 callback 함수의 등록을 제외하고 5.2.1절의 Pin의 branch coverage의 계측과 동일하다. 쓰레드마다 직전에 방문한 기본블록의 ID를 저장하는 변수를 전역으로 선언한 후, 새로운 기본블록을 방문 했을 때 직전에 방문한 기본블록의 ID와 현재 기본블록의 ID 순서쌍을 분기의 ID로 사용한다. 이후 현재 ID를 직전방문 기본블록의 ID로 갱신한다.

- **Path coverage**는 직전 기본블록의 ID가 아닌 직전 기본블록까지의 path의 ID, 즉 해시 값을 저장하는 변수를 사용하고, 현재 기본블록의 ID를 더하여 해시값을 재계산 하여 전역변수에 갱신한다.

5.3 Primitive 도출 성능 비교

Instruction count를 도출하기 위한 Pin과 DynamoRIO의 소모시간 증가율은 Figure. 6와 같이 나타났다. Pin을 통해서 instruction count를 측정했을 때, instrumentation 하지 않은 상태의 naive run보다 최대 약 1000%, 평균 570% 정도의 소요시간 증가를 보였다. DynamoRIO를 이용한 instrumentation에서는 instruction count를 측정하는데 평균 3300%의 소요시간 증가율을 보였다. 이를 정리하면 instruction count를 측정하는데 DynamoRIO가 Pin 보다 평균적으로 평균 6배 정도 더 많은 시간을 소모했다. 가장 편차가 적은 429.mcf(차량 스케줄링, network simplex 알고리즘 실행)에서 조차 4배 가량의 속도 차이를 보였다. 동일한 작업을 수행하는데 발생하는 이러한 큰 시간차가 발생하는 원인은 다음 세 가지 요소에서 분석할 수 있다.

첫 번째는 분석도구 자체가 가진 오버헤드이다. 분석도구에 따라 분석대상 바이너리를 특정한 환경위에서 수행하거나 코드를 분석하여 instrumentation에 필요한 정보를 획득한 후에 수행하는데, 이 때 소모되는 시간이 전체 수행시간에 영향을 미친다. 그러나 기존연구에 의하면 DynamoRIO와 Pin의 성능 비교에 있어서 이 원인에 의한 차이는 거의 발생하지

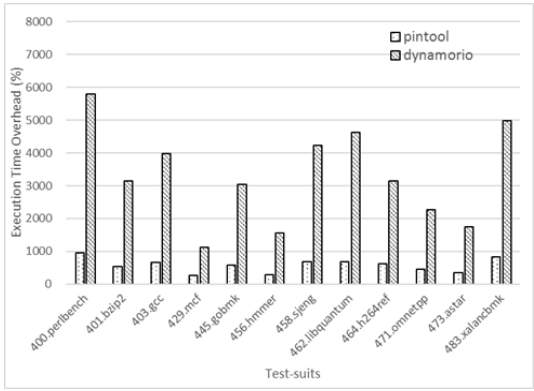

Fig. 6. Instruction count

않는 것으로 보고되어있다[7]. 바이너리의 구동 이외의 계측을 전혀 수행하지 않은 상태이므로 이 오버헤드는 모든 계측척도에 동일하게 적용된다.

두 번째 원인은 계측코드의 실행방식에서 있다. 바이너리분석도구들은 JIT 방식의 사용여부, code cache의 최적화, inlining 최적화 기법 등에서 성능차이를 가지는데, Pin과 DynamoRIO는 JIT 방식과 inlining 최적화에서는 유사한 방식을 사용하고 있다. 이 방식에 의한 차이는 실행코드와 계측코드가 단순할수록 그 차이가 적고, 최적화의 여지가 많은 복잡한 코드에서 그 차이가 크게 발생한다.

세 번째 원인은 실행과정에서 수행중인 프로세스의 상태와 계측한 값을 저장하는 방식의 차이가 원인이 된다. DynamoRIO는 Pin에 비하여 수행중인 프로세스의 상태를 더 많이 저장하는 것으로 알려져 있다. 이 차이는 Instruction count와 같이 그 계측이 잦은 척도를 측정하는데 있어 정보의 보존에서 얻어지는 이익보다 오버헤드로 인한 손실이 더 커지는 결과를 만든다. 이는 더 큰 단위로 계측되는 척도에서는 상대적으로 그 오버헤드의 영향이 감소할 것으로 예측가능한데 basic block count의 측정에서 실제로 그러한 양상이 나타난다.

Instruction count와 Figure. 7의 basic block count의 측정시간 차이에 있어서는 Pin과 DynamoRIO가 서로 다른 양상을 보였다. 소요시간의 증가율 자체는 여전히 DynamoRIO가 더 높은 것으로 나타났으나, DynamoRIO는 basic block count보다 instruction count에서 더 높은 증가율을 보인데 반해, Pin은 instruction count가 basic block count 보다 오히려 약간 더 낮은 오버헤드를 보였다. 일반적으로, instruction count가 basic block count가 보다 구조적으로 더 많은 작업이 수행되지만 Pin의 경우 최적화를 통해 instruction 레벨이 아닌 Trace 레벨에서 instruction count이 가능하기 때문에 이와 같은 차이가 발생한 것으로 보인다.

Figure. 8과 9는 basic block과 branch coverage를 측정한 결과이다. 앞서 본 결과와 같이 일반적으로 대부분에 프로그램에서 Pin이 DynamoRio보다 뛰어난 성능을 보여주는 것으로 나타났다. 하지만 464.h264ref의 경우는 반대의 결과를 보여주는 것을 확인할 수 있다. Pin과 DynamoRio 모두 JIT을 통한 basic block cache를 이용하여 최적화를 수행하고 있지만 Pin의 경우 inline 최적화 주로 DynamoRio의 경우는 Trace cache를 추가로 두어 JIT 레벨에서 최적화를 수행[25]하는 등 최적화 방식의 차이가 존재한다. 따라서 최적화 방식과 464.h264ref 바이너리 특성과 맞물려 실험 결과의 차이가 발생하는 것으로

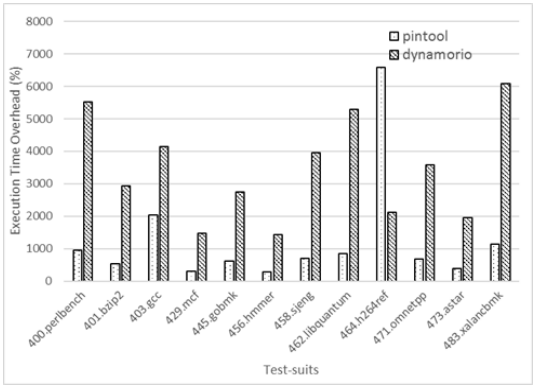

Fig. 8. Basic block coverage

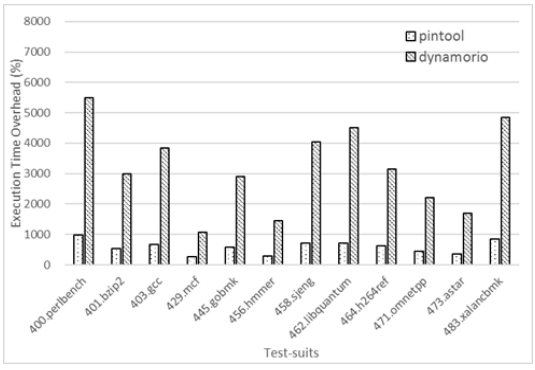

Fig. 7. Basic block count

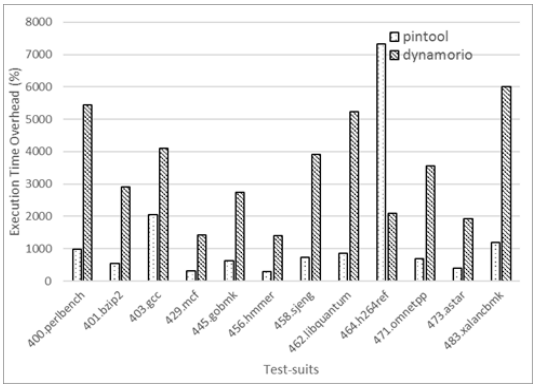

Fig. 9. Branch coverage

추정해 볼 수 있다. 또한, 이러한 결과는 2017 CGO Workshop 프로그램의 자료[24]에서도 464.h264ref 바이너리에 대해 동일하게 발생함을 확인하였다.

위 결과들을 종합하면 다음과 같은 결론을 도출할 수 있다. 정의한 원시척도들은 공통적으로 계측코드의 최적화가 쉽고 계측 값이 지속적으로 변화하며 계측코드 내에서 수행하는 행위가 단순하여, 전체 성능이 계측코드의 성능에 크게 좌우되는 일반적 사용례와 달리 바이너리의 수행과정에서 발생하는 오버헤드에 전체 효율성이 좌우된다. 따라서 464.h264ref와 같이 대상 바이너리가 특정 분석도구에 적합한 경우를 제외하고는 일반적으로 instrumentation 과정에서의 부하를 최소화한 도구가 더 나은 효율성을 보인다.

VI. 한계점 및 향후연구

본 연구에서는 소프트웨어적 instrumentation에서 측정 가능한 척도들을 정의하고 척도들을 계측하는데 소요되는 오버헤드를 측정하는 방법을 제안하였다. 최근의 분석도구들은 하드웨어에서 지원하는 기능들을 사용하고 있으며 이를 이용하고 있는 분석도구들이나 알고리즘의 성능을 정확하게 측정하고 하드웨어 지원 기능들의 시간적 이익을 정확히 파악하고 이를 극대화하여 활용하기 위해서는 내부기능에 대한 이해가 필요하다. Intel 프로세서의 경우 내장하고 있는 LBR(Last Branch Record), BTS(Branch Trace Store), 그리고 PT(Processor Trace)같은 instrumentation 기능들에 대해 정보가 공개되어 있기 때문에 이를 이용하여 분석도구를 구현하기 용이하나, 정보가 공개되어있지 않은 제조사의 프로세서에 대해서는 단위기능의 성능을 비교하는 방식은 그 한계가 있다. 또한 하드웨어에서 지원하는 instrumentation 기능들을 일정 버전 이상에서의 최신 OS에서만 사용할 수 있다는 제한이 있어, 성능측정 환경이나 개발한 분석도구 환경이 특정 버전의 OS로 제한된다는 점도 고려되어야 한다.

VII. 결 론

Dynamic Binary Instrumentation은 소스코드가 없는 상태에서 프로그램을 분석하기 위해 필수적인 기술로 악성코드 탐지, 취약점 탐지, 버그 탐지 등의 분야에서 폭넓게 활용되고 있지만, 정적분석과 비교하여 그 수행시간의 증가가 주요한 단점으로 제기되어왔다. 그럼에도 불구하고 다양한 DBI 도구들 간의 기능적 분석만 이뤄져 왔을 뿐 구체적 시간소모의 차이는 분석되어있지 않았다. 우리는 이 연구에서 DBI를 이용하여 통상적으로 계측되는 원시척도들을 정의하고 그 단위기능을 수행하는데 소모되는 시간을 측정, 비교하는 방법론을 제시하고, 실제 널리 사용되고 있는 DBI 도구와 신뢰성 있는 실험대상 바이너리를 사용하여 그 실측결과를 비교하였다. 이 방법론은 향후 연구와 분석과정에서 사용하고자 하는 도구, 얻고자 하는 정보에 적용하여 DBI에 의한 수행시간 증가를 최소화 할 수 있는 최적의 도구를 평가하고, 계측함수의 구현을 비교하는데 활용 될 수 있다.

References

[1] Lehman, M.M., "Programs, life cycles, and laws of software evolution," Proceedings of the IEEE, vol. 68, no. 9, pp. 1060-1076, Sep. 1980

[2] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E. and Turski, W.M., "Metrics and laws of software evolution-the nineties view," Software Metrics Symposium, 1997. Proceedings., Fourth International, pp. 20-32, Nov. 1997

[3] Ebert, C. and Jones, C., "Embedded software: Facts, figures, and future," Computer, vol. 42, no. 4, pp. 42-52, Apr. 2009

[4] Newsome, James, and Dawn Song. "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," In Proceedings of the 12th Network and Distributed Systems Security Symposium, Feb. 2005

[5] SPEC CPU 2006. https://www.spec.org/cpu2006/

[6] DynamoRIO. http://dynamorio.org/

[7] Luk, C.K., Cohn, R., Muth, R., Patil, H.,

Klauser, A., Lowney, G., Wallace, S., Reddi, V.J. and Hazelwood, K., "Pin: building customized program analysis tools with dynamic instrumentation," In ACM Sigplan Notices, vol. 40, no. 6, pp. 190-200, June. 2005

〔8〕 Laurenzano, M.A., Tikir, M.M., Carrington, L. and Snavely, A., "Pebil: Efficient static binary instrumentation for linux," In Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, pp. 175-183, Mar. 2010

〔9〕 Srivastava, Amitabh, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. technical report msr-tr-2001-50, microsoft research, 2001

〔10〕 Bernat, A.R. and Miller, B.P., "Anywhere, any-time binary instrumentation," In Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, pp. 9-16. Sep. 2011

〔11〕 Zhang, M., Qiao, R., Hasabnis, N. and Sekar, R., "A platform for secure static binary instrumentation," ACM SIGPLAN Notices, vol 49, no. 7, pp. 129-140. Jan. 2014

〔12〕 Nethercote, N. and Seward, J.,"Valgrind: a framework for heavyweight dynamic binary instrumentation," In ACM Sigplan notices, vol. 42, no. 6, pp. 89-100, June. 2007

〔13〕 Hunt, Galen, and Doug Brubacher. "Detours: Binaryinterception ofwin 3 2 functions," 3rd usenix windows nt symposium, July. 1999

〔14〕 BEAUCHAMP, Tiller: WESTON, David. Dtrace: The reverse engineer's unexpected swiss army knife. Blackhat Europe. 2008

〔15〕 Scott, K., Davidson, J.W. and Skadron, K., Low-overhead software dynamic translation. University of Virginia, Charlottesville, VA, 2001

〔16〕 Intel, Intel Microarchitecture codename Nehalem performance monitoring unit programming guide, https://software.intel.com/en-us/articles/intel-microarchitecture-codename-nehalem-performance-monitoring-unit-programming-guide-1.

〔17〕 Intel developer zone, Processor Tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

〔18〕 Tikir, M.M. and Hollingsworth, J.K., "Efficient instrumentation for code coverage testing," In ACM SIGSOFT Software Engineering Notes, vol. 27, no. 4, pp. 86-96, July. 2002

〔19〕 RUIZ-ALVAREZ, Arkaitz; HAZELWOOD, Kim, "Evaluating the impact of dynamic binary translation systems on hardware cache performance," In: Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on. IEEE, 2008, pp. 131-140, Sep. 2008

〔20〕 Soffa, M.L., Walcott, K.R. and Mars, J., "Exploiting hardware advances for software testing and debugging (nier track)," In Proceedings of the 33rd International Conference on Software Engineering, pp. 888-891, May. 2011

〔21〕 Walcott-Justice, K., Mars, J. and Soffa, M.L., "THeME: a system for testing by hardware monitoring events," In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 12-22, July. 2012

〔22〕 Schwartz, E.J., Avgerinos, T. and Brumley, D., "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," In 2010 IEEE Symposium on Security and Privacy, pp. 317-331, May. 2010

〔23〕 Bob Jenkins, one-at-a-time hash. http://collaboration.cmc.ec.gc.ca/science/rpn/

biblio/ddj/Website/articles/DDJ/1997/9709/9709n/9709n.htm
〔24〕 Derek Bruening, Building Dynamic Tools with DynamoRIO on x86 and ARMv8. http://cgo.org/cgo2017/workshop-program.html
〔25〕 E. Duesterwald and V. Bala, "Software profiling for hot path prediction: less is more," In ACM SIGPLAN Notices, vol. 35, no 11, pp. 202-211, Nov. 2000

〈저자소개〉

이 민 수 (Minsu Lee) 정회원
2014년 2월: 세종대학교 디지털컨텐츠학과 졸업
2016년 2월: University College London, MSc in Information Security
2016년 3월~현재: 한국과학기술원 사이버보안연구센터
〈관심분야〉 정보보호, 바이너리 분석, 프라이버시

이 제 현 (Jehyun Lee) 정회원
2007년 2월: 고려대학교 컴퓨터학과 졸업
2009년 2월: 고려대학교 컴퓨터전파통신공학과 석사
2015년 8월: 고려대학교 컴퓨터전파통신공학과 박사
2016년 4월~2016년 12월: 한국과학기술원 사이버보안연구센터
2017년 1월~현재: Singapore Management University, Secure Mobile Center
〈관심분야〉 네트워크 보안, 악성코드, 모바일 악성코드

김 호 빈 (Hobin Kim) 정회원
2015년 2월: 숭실대학교 컴퓨터학부 졸업
2017년 2월: 한국과학기술원 정보보호대학원 석사 졸업
2017년 2월~현재: 한국과학기술원 사이버보안연구센터 연구원
〈관심분야〉 정보보호, 시스템 보안.

류 찬 호 (Chanho Ryu) 정회원
1986년 2월: 충남대학교 계산통계학과 졸업
1988년 2월: 동국대학교 전산학과 석사 졸업
2000년 2월: 충남대 컴퓨터과학과 박사 졸업
2015년 1월~현재: 한국과학기술원 사이버보안연구센터
〈관심분야〉 소프트웨어보안, 역공학, 취약점분석