Journal of The Korea Society of Computer and Information
Vol. 18, No. 7, July 2013
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2013.18.7.111

점진적 실행을 통한 소프트웨어의 구조 그래프 생성

이 혜 련*, 신 승 훈**, 최 경 희*, 정 기 현***, 박 승 규**

Constructing Software Structure Graph through Progressive Execution

Hye-ryun Lee*, Seung-hun Shin**, Kyung-hee Choi*, Gi-hyun Jung***, Seung-kyu Park**

요 약

소프트웨어의 취약성을 검증하기 위하여 소프트웨어의 구조를 유추하여 유추된 구조를 활용하여 테스트하는 방법이 주목받고 있다. 이와 같은 방법을 사용하기 위해서 효과적인 소프트웨어의 구조 유추 방법이 요구된다. 많이 사용되는 DFG(Data Flow Graph), CFG(Control Flow Graph) 이나 CFA(Control Flow Automata)와 같은 그래프나 트리 방식은 소프트웨어 모델을 구조적으로 표현하지 못하는 단점을 가진다. 본 논문에서는 이러한 단점을 극복할 수 있는 방법을 제시한다. 제시된 방법은 바이너리 코드에 다양한 입력 데이터들을 부여하여 입력 데이터별 CFG를 생성하고, 생성된 CFG들이 구조적으로 표현될 수 있도록 계층적 제어 흐름 그래프(Hierarchical Control Flow Graph, HCFG)를 작성한다. 또한 제안하는 HCFG을 생성하는데 요구되는 그래프의 구성요소와 점진적 그래프 생성 알고리듬도 제시한다. 제안한 방법론을 공개된 SMTP(Simple Mail Transfer Protocol) 서버 프로그램에 적용시켜 소프트웨어의 모델을 작성하는 실험을 수행하고, 생성된 모델과 실제 소프트웨어 구조를 비교 분석한다.

▸ Keywords : 소프트웨어 구조, 소프트웨어 그래프, HCFG, 계층적 제어 흐름 그래프

Abstract

To verify software vulnerability, the method of conjecturing software structure and then testing the software based on the conjectured structure has been highlighted. To utilize the method, an efficient way to conjecture software structure is required. The popular graph and tree methods such as DFG(Data Flow Graph), CFG(Control Flow Graph) and CFA(Control Flow Automata) have a serious drawback. That is, they cannot express software in a hierarchical fashion. In this paper,

•제1저자 : 이혜련 •교신저자 : 정기현
•투고일 : 2013. 2. 12, 심사일 : 2013. 3. 5, 게재확정일 : 2013. 5. 9.
* 아주대학교 컴퓨터공학과(Dept. of Computer Science, Ajou University)
** 아주대학교 소프트웨어융합학과(Dept. of Software Convergence Technology, Ajou University)
*** 아주대학교 전자공학과((Dept. of Electronic Science, Ajou University)

we propose a method to overcome the drawback. The proposed method applies various input data to a binary code, generate CFG's based on the code output and construct a HCFG (Hierarchical Control Flow Graph) to express the generated CFG's in a hierarchical structure. The components required for HCFG and progressive algorithm to construct HCFG are also proposed. The proposed method is verified through constructing the software architecture of an open SMTP(Simple Mail Transfer Protocol) server program. The structure generated by the proposed method and the real program structure are compared and analyzed.

▸ Keywords : Software Structure, Software Graph, HCFG, Hierarchical Control Flow Graph

I. 서 론

소프트웨어에 내재된 오류는 전체 시스템의 정상적인 동작을 방해할 수 있는 취약성을 유발 시킬 수 있으며, 이 취약성은 해커들에 의한 악의적인 침투 수단으로 사용될 수 있다[1,2]. 따라서 소프트웨어의 취약성은 신뢰성에 깊은 영향을 미칠 수 있으므로, 이와 관련한 관심이 증가하고 있는 추세이다. 이에 따라 취약성 문제를 해결하기 위한 다양한 연구들이 활발하게 진행되고 있는 가운데, 소프트웨어의 구조 분석을 이용한 모델 기반 소프트웨어 테스트방법론이 주목 받고 있다. 이와 같은 방법을 사용하기 위해서는 먼저 소프트웨어의 정확한 모델 생성이 수행되어야 한다. 소프트웨어 모델 생성 방법은 크게 요구사항 기반 모델링[3], 소스코드 기반 모델링[4] 그리고 바이너리 코드 기반 모델링 방법[5]으로 나눌 수 있다.

우선 요구사항 기반 모델링에 사용되는 도구는 대표적으로 UML(Unified Modeling Language)과 시뮬링크(Simulink) 등이 있다. UML은 하나의 소프트웨어에 대하여 다양한 유형의 모델을 작성할 수 있으나, 한 소프트웨어의 구조를 다수의 다른 다이어그램으로 나누어 표현함으로, 요구사항의 변경에 유연한 대처가 어렵다는 단점을 가지며[6], 시뮬링크는 자동차 및 중장비와 같은 임베디드 시스템에 적재되어 있는 소프트웨어의 모델링에 주로 사용되는 언어이다[7]. 이와 같은 도구들을 이용하여 생성된 모델은 요구사항을 기반으로 작성된 모델이므로, 작성된 모델을 바탕으로 실제 소프트웨어의 구조를 정확하게 파악하는데 한계가 있다. 따라서 소프트웨어의 구조 유추를 통한 소프트웨어 취약성 검증에 적용하기 어렵다.

한편, 소스 코드 기반 모델링 방법은 소스 코드가 공개된 경우 유효하게 사용될 수 있으나, 대체로 소스 코드는 공개되지 않으므로, 공개된 소프트웨어를 대상으로만 모델을 작성할 수 있을 뿐만 아니라, 설사 소스 코드가 공개되어 있다고 하더라도 소프트웨어의 복잡도가 높은 경우, 소스 코드를 분석하는데 많은 시간이 소요되거나, 심지어 불가능한 경우도 발생된다[1,8,9].

따라서 소스코드가 없는 상태에서 소프트웨어의 모델을 추정하기 위한 다양한 연구가 수행되고 있으며, 대표적인 방법이 바이너리 코드를 바탕으로 모델을 생성하는 방법이다. 이 때 다수가 DFG, CFG 혹은 CFA을 기반으로 하는 그래프나트리 방식을 사용하고 있다[10-12]. 이 접근 방식은 소프트웨어의 규모가 일정 수준 이상 커지거나, 복잡도가 심한 경우, 그래프 크기가 폭발적으로 증가하게 되므로 생성된 모델이 소프트웨어의 전체 구조를 효과적으로 수용하지 못하게 되는 한계를 가진다.

본 논문에서는 이러한 문제를 해결하기 위해 소프트웨어의 바이너리 코드를 이용하여 소프트웨어의 구조를 추정하되, 추정된 모델을 HCFG로 표현하는 방법을 제안한다. 제안된 방법은 하나의 소프트웨어가 시작 모듈을 기준으로 한 모듈들 간의 상호관계, 즉, 실행 경로가 시작 모듈을 중심으로 한 계층적 구조를 가진다는 점을 이용한다. 또한, 다양한 형태의 입력 데이터를 이용해 바이너리 코드를 실행하여 소프트웨어를 구성하고 있는 분기가 최대한 커버되도록, 즉, 모델에 소프트웨어의 구조가 충실히 반영될 수 있도록 한다. 그리고 실행 회수의 증가에 따라 CFG가 점진적으로 확장되도록, 그리고 이 때 확장되는 CFG에 포함되어야 하는 구성 요소 수의 폭발적 증가를 방지하고, 구조화되어 표현될 수 있도록 CFG를 HCFG로 계층화하여 표현한다.

마지막으로, 본 논문에서 제안하는 방법의 가용성을 검증

하기 위하여 공개된 SMTP 서버 프로그램의 모델을 생성하는 실험을 수행하고, 실험 결과 생성된 모델과 실제 소프트웨어의 구조를 비교 분석하여, 제안된 방법을 통해 소프트웨어의 전체적인 구조를 파악할 수 있음을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 소프트웨어 모델 생성 관련 연구들을 분석하여 서술하며, 3장에서는 HCFG를 생성하는 방법을 기술하며, 4장에서는 실험을 통하여 소프트웨어 모델을 생성할 수 있는지를 보인다. 마지막으로 5장에서는 결론을 맺는다.

II. 관련 연구

1. 소스코드 기반 모델링 방법

데이터 흐름을 분석해 DFG를 생성하고, 이를 통해 모델을 생성하는 방법은 소프트웨어 내의 데이터들 간 상호관계에 초점을 맞추고 있으며, 이들은 데이터의 흐름 파악에 요구되는 변수의 수명 문제를 주요 이슈로 다룬다[13]. 이 방법은 데이터의 흐름이 파악되면 변수간의 연관관계를 확인할 수 있기 때문에 악의적인 데이터 변경의 탐지가 용이해지므로 보안 측면에서 중요성을 가진다. 하지만 소프트웨어에 사용된 변수의 양이 증가할수록, DFG가 기하급수적으로 커지는 현상이 발생된다. 또한, 포인터 변수가 복잡하게 사용된 경우, 데이터 간의 관계를 명확히 파악하기 어렵거나 불가능 하므로 모델의 생성이 어렵다는 한계점을 갖는다.

CFG를 이용하여 소프트웨어의 구조를 유추해 모델을 생성하는 방법은 소프트웨어의 제어 흐름을 파악하고, 이를 바탕으로 모델을 생성한다[14]. CFG를 기초로 모델을 생성하는 방법은 다음과 같이 크게 3가지로 구분된다.

- 라벨 전환 그래프(Labeled Transition Graph, LTG)

LTG는 통상적인 CFG 생성 규칙에 따라 생성되는 그래프로, 분기를 포함하지 않은 각각의 소스코드 블록에 고유한 라벨을 부여하고, 소프트웨어의 제어 흐름을 라벨 간 전환 그래프로 표현한다. 따라서 LTG를 이용해 작성된 모델은 라벨이 부여된 소스코드 블록 집합과 LTG를 포함하는데, LTG를 구성하는 각 노드 간 에지에는 해당 분기를 유발하는 변수의 이름과 값의 범위가 표현된다[15].

- 경계/내부 그래프(Boundary/interior graph)

경계/내부 그래프는 CFG 기반의 경로 테스팅(path testing)에 주로 사용된다[16]. 경계/내부 그래프 작성 시 하나의 노드는 참/거짓을 나타내는 최대 2개의 에지를 포함할 수 있으며, 특히 내부 그래프 작성을 위해서는 표현될 반복문의 수행 횟수가 사전에 정의되어야 한다.

- 제어 흐름 오토마타(Control-Flow Automata, CFA)와 추상적 도달성 트리(Abstract Reachability Trees, ART)

CFA는 소프트웨어를 구성하는 각 함수 단위로 하나의 방향성 제어 흐름(directed control flow) 오토마타를 생성하며, 노드는 프로그램의 제어 포인트들을 사용하고, 에지에는 두 제어 포인트 간에서 사용된 명령어이거나 해당 에지를 유발하는 조건을 표현한다. 한편, 각 함수에 대한 CFA를 취합하여 전체 소프트웨어의 구조를 표현한 트리가 ART이다[12]. ART는 트리 구조를 가지므로, 에지는 하나의 제어 포인트 m이 분기를 생성 하는 경우, 제어 포인트m과 연결되는 두 제어 포인트 n, k로의 에지에 각각 m→n과m→k로제어흐름을유발하는조건을표현해제어포인트사이의관계를나타내어 반복구문(iteration)를 별도로 표현하지 않기 때문에, 트리에 포함되어야 하는 노드 수가 발산하지 않는다는 특징이 있다[12].

소스 코드를 이용해 CFG 작성 시, 버퍼 오버런, 메모리 누수, 부동소수점, 오버플로우와 같은 소프트웨어 오류를 효과적으로 찾아낼 수 있는 장점을 갖지만, 규모가 큰 소프트웨어의 경우에는 모델 내에 표현해야 하는 정보가 방대해져 확장성이 떨어지는 단점을 갖는다. 즉, 소프트웨어의 복잡도가 높을수록, 그래프의 크기가 폭발적으로 증가하게 되므로 생성된 모델이 소프트웨어의 전체 구조를 효과적으로 수용하지 못하게 되는 한계를 가진다. 또한, 소프트웨어에 오류가 있는 경우, CFG 생성을 중단하고 해당 단계까지의 모델만을 제공하기 때문에 소프트웨어의 전체 구조 유추를 위한 모델링에는 사용하기 어렵다.

2. 바이너리 코드 기반 방법

바이너리 코드 기반으로 CFG을 작성하는 경우, 소스 코드가 없는 경우에도 모델을 생성할 수 있기 때문에 소프트웨어의 구조 유추를 위한 모델 생성에 효과적이지만, 실행 파일의 분석에 긴 시간이 요구된다. 따라서 분석할 소프트웨어의 크기가 커지면 분석시간 또한 길어져, 소스 코드를 이용해 모델을 생성하는 것보다 상대적으로 긴 시간이 걸리는 단점이 있다[5,17]. CFG를 사용해 모델을 생성하는 방법은 다음과

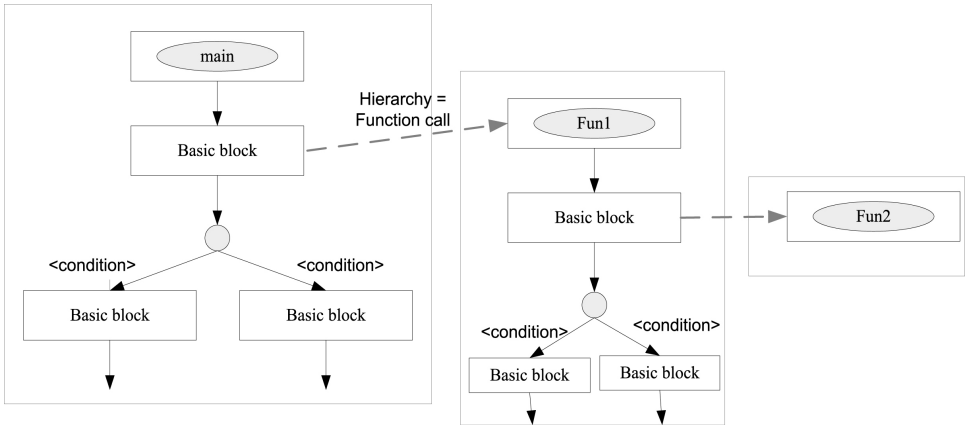


그림1. HCFG의 예
Fig 1. example of HCFG

같이 크게 3가지로 구분된다.

- 난독화 코드 기반의 CFG 작성

크리스토도레스쿠 등[18]은 악성 패턴을 감지하기 위해 실행 파일을 이용해 정적 분석을 수행하는 도구(static analyzer for executables, SAFE)를 개발하였다. SAFE는 바이너리 코드를 로더(Executable Loader)에서 IDA pro와 CodeSurfer를 사용하여 난독화된 코드(obfuscated code)을 작성한 다음, 이 코드를 이용해 CFG을 생성한다.

- 바이너리 코드 기반의 CFA 작성

Jakstab(Javatoolkitforstaticanalysisofbinaries)은 명령(instructions)과 IL 문장(Intermediate Language statements)를 이용해 바이너리 코드로부터 CFA를 생성한다[19]. 실행 순서는 실행 파일로부터 얻은 바이너리 코드에 대하여 본 연구에서 정의한 상대적 PC 값을 오프셋 한 다음, 바이너리 코드를 명령어들로 디코드 한다. 디코드 한 명령어들을 IL로 번역하여 CFA을 생성한다.

- 니모닉 데이터 생성을 통한 CFG 작성

흐름 그래프 분석기(Flow Graph Analyzer)[5]는 소프트웨어의 구조를 추정할 수 있도록 바이너리 코드를 이용하여 자동으로 CFG를 생성한다. 실행 순서는 PE(Portable Executable) 형식의 바이너리 파일에서 도구(PE_VIEWER)를 이용하여 텍스트 섹션을 찾고 이를 출력한다. 그 후 디스어셈블러(PE-Disassembler)를 이용하여 PE_VEIWER로부터 텍스트 섹션 정보를 확보하고, 이를 이용해 바이너리 형태로 되어있는 코드를 니모닉(mnemonic) 형태의 어셈블리 언어로 변환한 후, 니모닉 데이터를 사용해 CFG을 생성한다.

III. HCFG(Hierarchical Control Flow Graph)

기존 연구들을 보면 소프트웨어의 구조를 유추하여 모델을 생성하는 도구들은 소프트웨어의 구조를 분석하여 그래프나 트리로 모델을 작성함으로써, 소프트웨어의 오류를 찾는데 기여하였다. 하지만, 많은 모듈들로 구성된 복잡한 소프트웨어에 대한 그래프를 생성하거나, 소프트웨어의 내부에서 사용되는 변수들이 많아 이를 적용시켜 그래프 생성 시 그래프가 기하급수적으로 커지게 되어 완성된 모델이 소프트웨어의 전체 구조를 효과적으로 수용하지 못하게 되는 한계를 가진다.

따라서 본 논문에서는 대부분의 소프트웨어들이 모듈 간 계층적 구조를 가진다는 점에 착안하여 소프트웨어의 논리적 구조를 파악하기 위해 CFG에 계층구조(hierarchical) 개념을 추가한 HCFG로 모델을 표현하는 방법을 제안한다. 또한 제안하는 HCFG를 생성하는데 요구되는 그래프의 구성요소와 그래프 생성 알고리듬을 제시한다.

1. 계층적 제어 흐름 그래프 생성(HCFG)

HCFG는 이미 1995년 프리츠(D.G Fritz)가 개별적 이벤트 시뮬레이션을 위한 계층적 모델링 패러다임 연구논문에 사용했다[20]. 이 논문에서 HCFG는 메시지 패싱에 의하여 통신하는 각각의 독립적인 모듈들의 상호작용 및 동작을 모델

링하기 위하여 사용되었으나, 본 논문에서는 소프트웨어의 논리 구조 모델링에 HCFG 개념을 차용하고, 이를 통해 모델 내에 포함되어야 하는 실행 경로가 폭발적으로 증가하는 현상을 억제해 관리 가능한 수준이 되도록 한다.

본 논문에서의 HCFG 표현 방법은 기존의 CFG 혹은 플로차트와 유사하지만, 소프트웨어의 실행 과정에서 동적으로 작성된다는 점과 실행 경로의 표현을 위해 모듈 개념을 사용하고, 이 과정에서 소프트웨어의 구조를 계층적으로 표현할 수 있도록 한 것이 특징이다.

따라서 기본적으로 하나의 분기 문장 다음 명령어부터 다음 분기 문장 직전까지의 순차적인 명령어 집합에 대하여 기본 블록으로 정의하여, 하나의 함수 호출에 의해 연달아 발생될 수 있는 복잡한 흐름이 기본 블록에서는 하나의 명령행과 동일한 수준으로 표현되므로, 하나의 기본 블록은 다수의 하위 모듈들의 HCFG를 내포하는 형태가 되므로 소프트웨어의 구조를 계층화하여 표현할 수 있다. 또한, 분기에 의해 선택되는 기본 블록은 분기 블록과 연결하여 표현하되, 이 둘을 연결하는 에지에 해당 분기를 유발한 입력 데이터를 분기 유발 조건으로 표현한다.

그림 1은 본 연구에서 제안하는 HCFG 예제로 main 모듈과 2개의 사용자 정의 모듈을 가지는 간단한 소프트웨어의 구조를 HCFG로 표현하였다.

HCFG 작성 흐름을 보면, 먼저 main 모듈에서 기본 블록을 생성하고, 분기 발생 전까지의 명령문에 대한 기본 블록을 생성한다. 기본 블록 생성 시, 블록 안에 모듈을 호출하는 명령문이 존재하는 경우 해당 모듈이 실행되므로 main 모듈의 HCFG 생성을 중단하고, 호출된 모듈에 대한 HCFG를 작성하기 시작한다(Fun1 모듈). Fun1 모듈에서도 코드의 흐름에 따라, main 모듈에서의 동일한 방식으로 분기 발생 전까지 기본 블록을 생성한다. 이 때, 기본 블록 생성 시, 블록 안에 모듈을 호출하는 명령문이 존재하는 경우, Fun1 모듈의 HCFG 생성을 중단하고, 호출된 모듈에 대한 HCFG를 작성하기 시작한다(Fun2 모듈). Fun2 모듈에서도 앞에서 진행한 것과 동일하게 HCFG를 작성하고, 모듈의 종료 시점까지 블록들을 모두 작성하였다면, Fun1 모듈의 HCFG 생성을 중단한 시점으로 돌아간다. 다시 Fun1 모듈의 HCFG 생성을 시작하는데, 분기가 발생한다면, 분기 블록을 생성하고, 분기 조건에 따라 기본 블록들을 여러 개 생성하고, 기본 블록을 가리키는 에지에 조건을 작성한다. 그 다음 코드의 흐름에 따라 모듈의 종료 시점까지 블록들을 작성한 다음, main 블록의 HCFG 생성을 중단한 시점으로 돌아간다. main 모듈로 돌아와서 동일하게 모듈의 종료 시점까지 블록을 작성하여 HCFG 작성을 끝낸다.

2. HCFG의 구성요소

본 논문에서 제안하는 HCFG는 다음의 5 가지 요소로 구성된다.

1) 기본 블록(basic block): 내부에 분기가 존재하지 않는 명령어 집합.
2) 정션 블록(junction block): 분기 조건을 포함하는 명령문. 조건의 판단 결과에 따라 실행될 기본 블록과 연결.
3) 분기 조건: 정션 블록과 연결된 기본 블록들의 관계. 분기 조건의 식별에 해당 분기를 유발한 입력 데이터를 사용.
4) 모듈 호출: 기본 블록 내에 포함된 모듈의 호출. 모듈 호출 시 새로운 모듈의 탐지로 판단하고 이를 계층 구조로 표현
5) 입력 데이터: 소프트웨어의 실행을 위해 부여되는 데이터. 대부분의 입력 데이터는 시스템 콜을 이용해 소프트웨어에 부여되며, 본 논문의 HCFG에서는 이를 분기 조건의 명시에 사용.

3. 그래프 작성 알고리듬

본 논문에서 제안하는 HCFG 생성 알고리듬은 표 1에 제시한 것과 같다. 제시한 알고리듬은 입력 데이터를 구조 추정 대상 소프트웨어에 부여하여 실행시키는 작업을 반복해 점진적으로 HCFG가 확장되도록 하여 궁극적으로는 실제 소프트웨어의 구조와 유사한 그래프를 얻을 수 있도록 한다. 이 과정에서 다양한 형태의 입력 데이터를 사용하여 가능한 모든 제어 흐름을 탐지 가능하도록 한다.

제안하는 알고리듬의 상세 동작 방식은 다음과 같다. 우선 임의의 입력 데이터를 선택하여 이를 모델 생성 대상 소프트웨어에 부여하여 실행시킨다. 이 작업은 대상 소프트웨어의 구조에 따라 일반적으로 다수의 커버되지 않은 분기 블록을 포함한 HCFG를 생성하게 된다. 이후, 생성된 HCFG에 있는 커버되지 않은 분기 블록을 선택하고, 이 분기 블록을 커버할 수 있는 확률이 높은 입력 데이터 리스트를 과거 입력 데이터를 기반으로 선별하여 입력 데이터 리스트를 구성한다. 입력 데이터 리스트가 구성되면 이 리스트에 포함되어 있는 입력 데이터를 이용하여 소프트웨어를 실행시키고 그 과정에서 발견되는 기본 블록과 분기 블록을 HCFG에 추가하여 점진적으로 HCFG를 대상 소프트웨어 구조와 유사한 형태로 발전시켜 가도록 한다.

이와 같은 작업은 지정된 회수까지 반복적으로 수행하거나, 반복 수행 과정에서 분기 블록이 모두 커버되었다면

표 1. HCFG 생성 알고리듬
Table 1. Generation Algorithm of HCFG

```
Algorithm HCFGAlgo {
generate initial test input randomly and apply it to a program;

while (1) {
        choose a junction; // several strategies are possible
        if (no more uncovered junction or iteration >= Max_Iteration)
         break;
        list = GeneratNewTestInputs(Jct) ;
       // list contributes to increasing the branch coverage at JCT
        for (each input in the list) {
         Run the software with an input;
         Extend HCFG;
       }
     }
 }
End HCFGAlgo
```

```
Algorithm   GeneratNewTestInputs(Junction Jct ) :
 // generating new test   inputs
 {
    Jct = junction with uncovered branch;
    Lprev = conjunction of path constraints from start of ASN to Jct
    LJct = disjunction of branch conditions of Jct
    Generate test inputs such that L (= Lprev && ! LJct ) is satisfied;
 }
End GeneratNewTestInputs
```

HCFG 구성이 완료된 것으로 판단하고 HCFG 생성 작업을 중단한다. 입력 데이터 리스트의 구성은 HCFG 내의 커버되지 않은 분기 블록(Jct)을 대상으로 수행되는데, 이 때 해당 분기 블록까지 도달하는데 사용된 입력 데이터(L_{prev})와 분기 블록의 가지는 이전 분기 조건을 위배할 것으로 추정되는 입력($!L_{Jct}$)을 이용해 생성한다. 한편 분기 블록의 커버 여부 판단은 모델 생성 대상 소프트웨어가 바이너리 코드라고 할지라도 이를 지원하는 도구[21] 등을 사용하는 경우, 분기의 커버 여부뿐만 아니라 분기의 존재 및 시스템 콜을 포함한 함수 호출의 수행 등을 파악할 수 있으므로 충분히 구현 가능하다.

4. HCFG 작성 예제

제안하는 HCFG 작성 알고리듬을 표2에 제시된 예제 프로그램에 적용했을 때 생성되는 HCFG는 아래와 같은 형태로 확장이 이루어진다.

프로그램이 시작되면 우선 기본 블록 하나를 생성하고 분기가 이루어지기 전까지의 명령문 집합을 이 기본 블록에 포함시킨다. 통상적인 분기와 함수 호출을 위한 분기는 메모리에 대한 작업 등이 서로 다르기 때문에 구분이 가능하다. 따라서 식별 결과에 따라 3번 라인의 명령문 수행에 수반되는 명령문 집합은 계층적인 HCFG 작성을 위해 별도의 HCFG로 작성해야 하며, 해당 기본 블록에서는 추가된 HCFG로의 연결만 표현한다. 함수는 시스템 콜의 수반 여부 혹은 입력 데이터 수용 등을 위한 프로그램의 블록 상태 진입과 같은 정보를 토대로 함수 형태별로 구분하여 HCFG 생성이 가능하나, 본 논문에서는 단순히 라이브러리 함수와 사용자 정의 함수만으로 나누어 표현한다.

표 2. 소스 코드 예제
Table 2. example of source code

```
1   int   main(void)
2   {
3       scanf("%d", &x);
4       while ( x > 0 )
5       {
6           if ( x > 10000 )
7           {
8               printf("Too big x   \n");
9               return 0;
10          }
11          y = y + sum( x );
12          log( x, y );
13          scanf("%d", &x);
14      }
15      return 0;
16  }

17  int   sum( x )
18  {
          . . .
19          k = k + sum( x-1 );
20  }
```

3번 라인의 명령문 실행 시 입력 데이터를 요구하며, 이는 첫 번째 입력이므로 임의의 데이터를 부여한다. 첫 번째 입력 데이터로 0이 사용됐다고 가정하면, main의 HCFG에는 분기와 사용한 입력 데이터가 표현되고, 적용된 입력 데이터에 의해 프로그램이 종료되었으므로 그림 2와 같이 HCFG을 작성하고, 프로그램을 다시 실행시킨다.

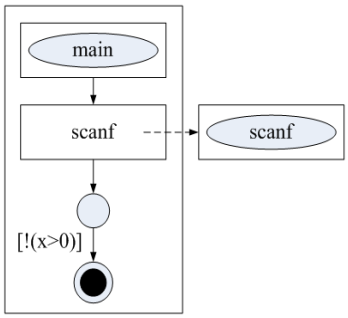


그림 2. HCFG 작성 예 - 1단계
Fig 2. Example of Creating HCFG - Step 1

2번째 실행에서는 아직 커버되지 않는 4번 라인의 분기를 커버 대상 분기로 선택하고, 이 분기를 커버하기 위한 입력 데이터 리스트를 생성한다. 현재 Lprev가 존재하지 않으므로 !LJct을 만족하도록 입력 리스트에 정수 데이터의 최소값(INT_MIN)과 최대값(INT_MAX)이 포함되었다고 가정하면, 이를 이용한 실행에서 INT_MIN의 경우 새로운 실행 경로를 탐색시키는데 실패하나, INT_MAX의 경우 4번 라인의 분기를 커버시키고 6번 라인의 새로운 분기와 8번 라인에서 시작되는 새로운 기본 블록이 그림 3과 같이 생성되도록 한다.

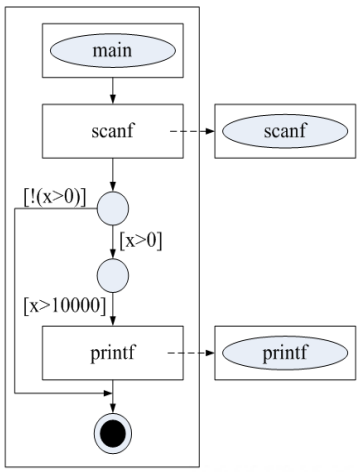


그림 3. HCFG 작성 예 - 2단계
Fig 3. Example of Creating HCFG - Step 2

세 번째 실행에서는 아직 커버되지 않은 6번 라인의 분기를 커버하기 위한 입력 데이터 리스트를 생성하되, 해당 분기까지 프로그램을 실행시키기 위해 4번 라인 분기에 사용된 입력 데이터(Lprev)와 종전 실행에 사용됐던 데이터(!LJct)를 이용해 생성한다. 이 때 가령 기존 입력 데이터를 근거로 양의 정수만 리스트 내에 포함이 되었다면, 이들 가운데에는 리스트 생성 시에는 알지 못하나 6번 라인의 분기를 커버시키는 입력 데이터인 10,000 이하의 정수도 포함될 수 있다. 따라서 이 데이터에 의해 11~13번 라인의 기본 블록을 탐색하게 된다. 또한 상기한 절차를 새로 탐색한 블록에도 적용하면 사용자 정의 함수인 sum을 포함해 다른 두 함수의 HCFG를 추가한 기본 블록과 연계하여 표현할 수 있게 되고, 새로이 요구하는 입력 데이터에 의해 프로그램의 제어가 4번 라인의 분기 블록으로 이동하므로 커버 되지 않은 분기가 없는 것으로 판단하여 그림 4와 같은 HCFG를 생성하고, HCFG의 생성을 종료한다.

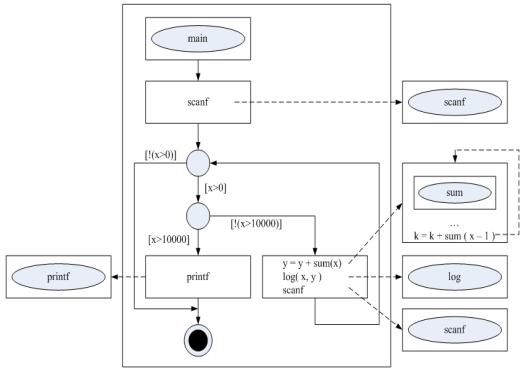


그림 4. 확장 완료된 HCFG
Fig 4. completed HCFG

IV. 실험

이 장에서는 앞서 제안하는 HCFG 생성 알고리듬의 가용성 평가를 위하여 공개된 SMTP 서버 프로그램을 대상으로 HCFG을 작성하고, 작성된 HCFG가 실제 소프트웨어의 구조와 유사한지를 비교 분석하는 실험을 수행하였다.

1. 실험 환경

본 논문에서는 소프트웨어의 바이너리 코드를 이용하여 HCFG을 작성한다. 소프트웨어의 소스 코드가 실제적으로 공개되는 경우가 드물며, 공개되어 있어도 소프트웨어의 복잡도가 높은 경우, 모델 생성이 불가능한 경우도 발생하므로 바

이너리 코드를 이용하여 HCFG을 작성한다. 바이너리 코드 분석에 사용할 수 있는 응용 프로그램들로는 Valgrind, IA-32EL, Boa, Transmeta's CMS, 인텔의 PIN 등이 있는데, 이 가운데 PIN은 인텔에서 개발된 동적 바이너리 도구 삽입 시스템으로, Linux 및 Windows에서 사용자 응용 프로그램을 분석하는데 사용할 수 있다[22]. PIN에서는 JIT(Just in Time) 컴파일러를 이용하며, 작업부하 특성, 프로그램 추적, 캐시 모델링 및 시뮬레이션과 같은 성능 분석 작업과 같은 다양한 사용을 가능하게 하는 사용자가 요구하는 도구 삽입을 작성하기 위한 API를 제공한다. 이 API을 이용하여 PIN에서는 분석 대상의 바이너리 코드에 정보를 삽입하여 사용자가 원하는 형태의 바이너리 코드를 만든다.

이와 같은 바이너리 분석 도구를 사용 가능함에도 불구하고, 본 논문에서는 제안하는 알고리듬이 실제 소프트웨어 구조와 유사한 모델을 생성할 수 있는지 명확하게 확인하기 위하여 소스 코드를 확보 가능한 소프트웨어에 자체 정의한 도구를 삽입하여 컴파일 및 실행한다. 따라서 바이너리 분석 도구를 이용하여 얻을 수 있는 수준의 프로그램 제어 흐름 정보를 획득을 위한 도구를 소스 프로그램 내에 삽입하고, 그 실행 결과물을 바탕으로 HCFG 확장 과정을 확인한다. 이를 위해 다음과 같이 삽입할 도구를 정의한다.

1) 문장(statement) 식별 코드 : 파일 식별 기호 + 라인 번호 : 바이너리 프로그램의 프로그램 카운터(PC)를 대신하여, 각 문장을 고유하게 식별하기 위해 삽입된다.
2) 문장(statement) 종류 코드
   - BLB : basic BLock Begin, 내부에 조건 및 반복에 의한 분기가 존재하지 않는 문장들의 블록을 '기본 블록'이라고 정의할 때, 해당 블록의 시작 위치에 표현.
   - UFN : User FuNction 사용자 정의 함수
   - LFN : Library FuNction, 일반 라이브러리 함수
   - BLF : BLocking Function, 특정 문장에서 일정 시간 동안 일정 시간 이상 PC가 증가하지 않는 경우, 현 프로그램이 블록 상태에 진입.
   - SCL : System CalL, 시스템 콜 유발 함수
   - JCN : JunCtioN, if, while 등과 같이 특정 조건에 따라 선택되는 BLB가 달라지는 경우 삽입
   - RET : RETurn, 함수 호출 후 복귀하는 지점에 사용되는 코드이며, RET은 타 코드와는 달리 별도의 인덱스를 지정하지 않음.

이와 같이 정의된 문장 식별 코드와 문장 종류 코드는 다음과 같은 형태로 소스 파일에 삽입되게 된다.

〔파일 식별 코드〕〔라인 번호〕 : 〔문장 종류 코드〕 : 〔각 코드별 인덱스〕

예를 들어 'A043:BLB:034'라는 코드가 부여된 경우, 이는 "파일 'A' 에 존재하는 '43'번째 라인에서 새로운 블록(BLB)이 시작되며, 이는 전체 블록 가운데 id가 '34'번이 부여된 블록이다"라는 의미를 가진다.

다음 그림 5는 실험 대상 소프트웨어의 일부에 도구를 삽입했을 때의 모습이다. 삽입한 도구는 실제 소프트웨어의 실행 결과에 어떠한 영향을 주지 않고, HCFG 생성에 필요한 정보들을 파일로 출력한다.

2. 실험 순서

실험 순서는 실험하는 프로그램과 상관없이 동일하게 총 5단계로 나누어 진행된다.

1) 도구 삽입 : 실험 대상 소프트웨어에 도구를 삽입한다.
2) 입력 데이터 리스트 생성 : 프로그램 실행 및 분기 블록 커버를 위한 입력 데이터 리스트를 생성한다.
3) 바이너리 코드 실행 : 작성된 입력 데이터 리스트를 이용하여 실제 소프트웨어에 적용시켜서 도구가 삽입된 바이너리 코드를 실행시키고, 생성되는 파일을 확보한다.
4) HCFG 생성 : 생성된 결과물을 이용하여 HCFG을 확장해 나간다.
5) 커버리지 분석 : 최종 완성된 HCFG와 소스코드의 구조와 비교함으로써 어느 정도 커버리지 됐는지 분석한다.

입력 데이터 리스트의 생성에는 SMT 해석기를 이용한 방법과 임의의 데이터를 사용하는 방법이 고려될 수 있다. SMT 해석기를 사용하는 경우, 분기 블록을 커버할 것으로 추정되는 입력 데이터를 이전 입력 데이터 간의 연산식을 구성해 생성하므로 복잡한 연산식의 작성과 해당 식을 만족하는 해를 찾는 과정이 필요하다. 입력 데이터를 생성하는 방법은 SMT를 이용하는 방법보다 적은 정보를 이용하게 되므로 분기를 커버하기 위해 더 많은 수의 입력 데이터를 필요로 할 수 있으나, 데이터 생성이 용이하다. 본 논문에서는 알고리즘의 가용성 확인하기 위하여, 임의의 데이터 집합을 생성할 수 있는 알고리즘에 적합하게 생성하여 주는 도구인 Sulley[23]을 이용한다. Sulley는 TippingPoint의 P. Amini와 A. Portnoy가 개발한 파이썬 기반의 네트워크 프로토콜 퍼저 중의 하나로, Sulley는 전송될 데이터의 필드 길이와 데이터 타입 등을 지정하면, 미리 정의된 다양한 형태의 라이브러리를 이용하여 입력 데이터 리스트를 자동으로 생성한다.

```
int CMailSession::ProcessNotImplemented(bool bParam,
FILE *instFd)  {
  int num;
  /*-------------------------------------*/
  fprintf(instFd, "I001:BLB:050\n");
  /*-------------------------------------*/

  /*-------------------------------------*/
  fprintf(instFd, "I002:JCN:027\n");
  /*-------------------------------------*/
  if (bParam) {
  /*-------------------------------------*/
  fprintf(instFd, "I004:BLB:051\n");
  fprintf(instFd, "I004:UFN:001\n");
  /*-------------------------------------*/
  num = SendResponse(504, instFd);
  /*-------------------------------------*/
  fprintf(instFd, "I004:RET\n");
  /*-------------------------------------*/
  }
  else {
  /*-------------------------------------*/
  fprintf(instFd, "I007:BLB:052\n");
  fprintf(instFd, "I007:UFN:001\n");
  /*-------------------------------------*/
  num = SendResponse(502, instFd);
  /*-------------------------------------*/
  fprintf(instFd, "I007:RET\n");
  /*-------------------------------------*/
  }
  return num;
}
```

그림 5. 도구가 삽입된 예제코드
Fig 5. Sample code to inserted tool

3. 실험

3.1 실험 구성

실험 대상 소프트웨어는 소스가 공개된 SMTP 서버 프로그램으로 C++로 작성되어 있으며[24], 그림 6과 같이 모두 11개의 함수와 30개의 분기로 구성되어 있다.

Sulley는 서버 프로그램에 대해 프로토콜이 정의하는 순서로 입력 데이터를 보내기 위해, 그림 7과 같이 세션을 구성했다. 그림 5와 같이 SMTP 서버 프로그램에 도구를 삽입하여 컴파일한 후, 실행시킨 상태에서 Sulley로부터 소켓을 통해 전달되는 입력 데이터들을 받는다. SMTP 서버 프로그램은 입력 데이터들에 대한 그에 상응하는 동작을 수행하며, 삽입된 도구에 의해 결과물을 생성한다.

```
from sulley import *
from requests import smtp_fuzz

def do_smtp_fuzz();
      #Define Session Properties
    #Add   target and agents
    ……..
   sess.add_Target(target)
   sess.connect(s_get("HELO"))
   sess.connect(s_get("HELO"),    s_get("MAIL"))
   sess.connect(s_get("MAIL"), s_get("RCPT"))
   sess.connect(s_get("RCPT"), s_get("DATA"))
   sess.connect(s_get("DATA"), s_get("QUIT"))
   sess.fuzz()
   print  "Test Cases Completed For SMTP"

Do_smtp_fuzz()
```

그림 7. Sulley의 SMTP 세션
Fig 7. Sulley's SMTP session

3.2 실험 결과

Sulley를 이용하여 크기가 1Kb ~ 1.1GB 인 약 12,219개의 입력 데이터들을 만들고, 이 입력 데이터들을 SMTP 서버 프로그램에 적용시키면, 표 3과 같은 실험 결과를 얻게 된다.

또한, 생성된 출력물을 이용하여 HCFG을 작성하면, 그림 9와 같이 나타낼 수 있으며, 그림 9의 mail 메서드의 HCFG는 그림 8의 mail 메서드의 코드 구조와 일치한 것을 볼 수 있다.

```
int CMailSession::ProcessMAIL(char *buf, int len, FILE
*instFd) {
  ……
  ….

  if(m_nStatus!=SMTP_STATUS_HELO)  {
    num = SendResponse(503,   instFd);
    return num;
  }

  memset(address,0,sizeof(address));
  ……

  if(!CMailAddress::AddressValid(address))  {
    num = SendResponse(501,instFd);
    return num;
  }

  return num;
}
```

그림 8. Command-mail 메서드 코드 일부분
Fig 8. the code of Command-mail method

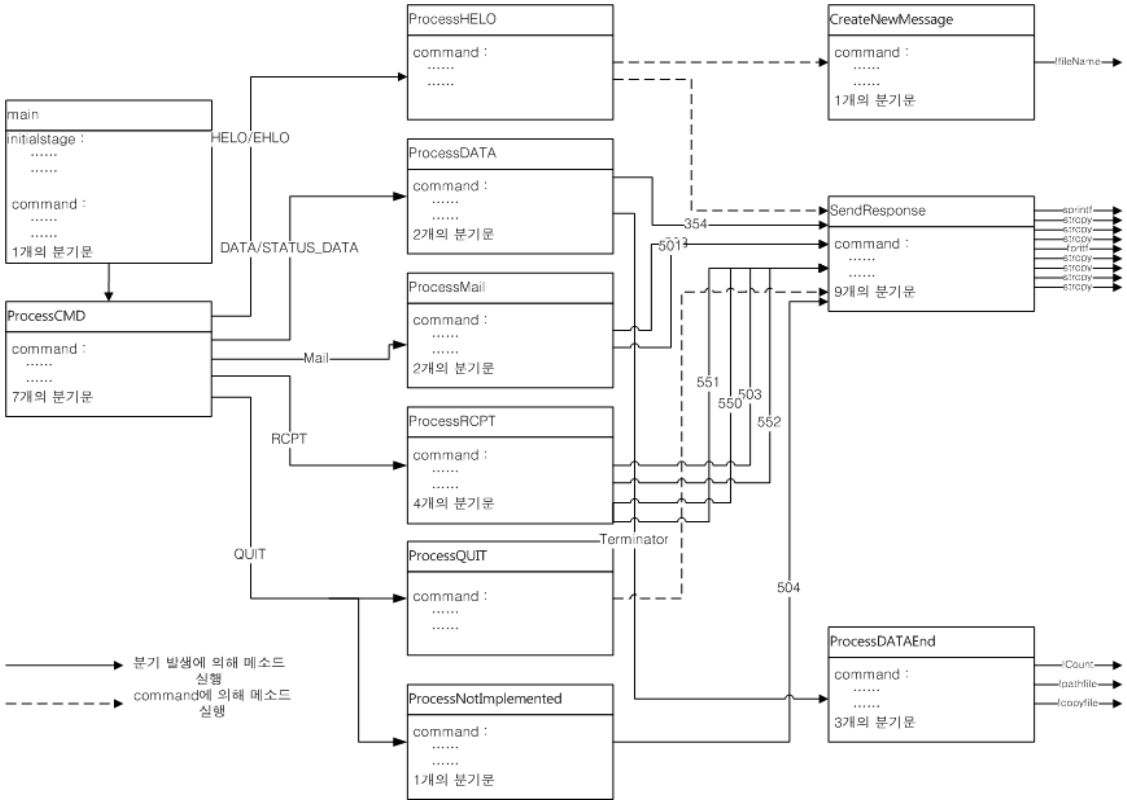


그림 6. SMTP 서버 프로그램 구조
Fig 6. structure of SMTP server program

본 논문에서는 하나의 분기문에서 실행될 수 있는 경로가 2개 이며, 입력 데이터를 통해 분기문을 만족하여 그 다음 문장을 실행하였다면, 하나의 경로를 실행한 것이다. 따라서 입력 데이터들에 따라 하나의 분기문에서 실행될 수 있는 2개의 경로를 모두 실행하면 그 분기에 대하여 모두 커버 하였다라고 정의한다. 따라서 이러한 정의에 의하여 생성한 모델을 분

표 3. 실험결과
Table 3. The experimental results

입력 데이터 수	12,219개		
입력 데이터 크기(개당)	1KB ~ 1.1GB		
전체 분기 수	33개		
분기 커버리지	모듈	커버리지	입력 관련 분기 블록 수
	Main	1(100%)	1
	Message Receiver	9(100%)	9
	Command Parser	7(100%)	7
	Command - Data	2(100%)	2
	Command - HELO		0
	Command - MAIL	2(100%)	2
	Command - RCPT	4(100%)	4
	Command - Quit		0
	Command - NotImplemented	0(0%)	1
	Command - DataEND	2(66.7%)	3
	CreateMessage	1(100%)	1

석하면, SMTP 서버 프로그램을 구성하는 전체 33개의 분기 중 입력 데이터와 관련된 분기는 30개 이며, 이 중 28개의 분기를 커버하여, 93%의 커버리지를 보였다.

3.3 실험 분석

제안한 알고리듬으로 커버하지 못한 모듈의 분기를 살펴보면, 커버 되지 않은 분기블록을 가지고 있는 NotImplemented 메서드는 향후에 기능을 추가하기 위하여 비워둔 상태이며, NotImplemeted 메서드는 ProcessCMD 메서드에 의해서만 호출되는 메서드로 그림 10과 같은 형태로 호출된다. 진행된 실험에서 NotImplemeted 메서드의 커버하지 못한 분기 블록은 if 분기인데, 이 분기의 조건이 되는 값을 ProcessCMD 메서드 내에서 NotImplemented 메서드 호출 시, 완성되지 않는 내용이 있어 항상 'false'로 부여하기 때문에, 해당 분기의 커버는 정상적인 절차를 통해서는 절대로 불가능함을 의미한다. 따라서 표 3에 제시한 실험 결과와 커버하지 못한 소스의 분석 결과를 기반으로 고려 할 때, 제안된 HCFG 생성 알고리듬이 소프트웨어의 구조 파악을 통한 모델 작성에 사용 가능함을 의미한다.

기존 관련연구들과 비교 분석한 결과, 소스 코드에 의한 모델 생성 시 소프트웨어의 전체 구조를 유추하기 어려운 반면에 바이너리 코드를 이용하는 방법을 사용하여 소프트웨어의 전체 구조를 유추 할 수 있었다. 또한 기존 관련 연구들에서 바이너리 코드에 의한 모델 생성 시 CFG로 표현하여 기하급수적으로 그래프가 커지는 현상이 발생하여 소프트웨어 모델을 구조적으로 표현하지 못하였으나, 본 논문에서 HCFG을 이용하여 소프트웨어 모델을 구조적으로 표현할 수 있었다.

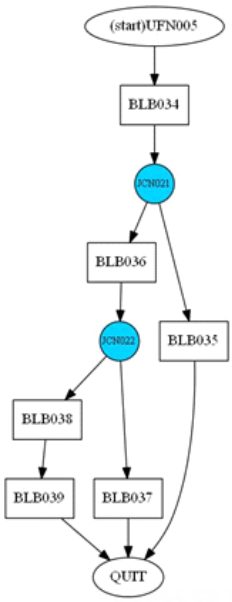


그림9. mail HCFG모델
Fig 9. model of mail HCFG

```
int CMailSession::ProcessCMD(char *buf, int len)
{
  ...
  num = ProcessNotImplemented(false);
  ...
}
```

그림 10. ProcessCMD 메서드 코드 일부분
Fig 10. the code of ProcessCMD method

VI. 결론

본 논문에서는 바이너리 코드를 이용하여 소프트웨어 구조를 유추하기 위한 모델을 작성하는 방법으로 HCFG 생성 알고리듬을 제안하였다. 제안하는 HCFG 생성 알고리듬의 가용성 평가를 위하여 공개된 SMTP 서버 프로그램을 대상으로 HCFG을 작성하고, 작성된 HCFG가 실제 소프트웨어의 구조와 유사한지를 비교 분석하는 실험을 진행하였다.

그 결과, 실험을 통해 제안된 HCFG 생성 알고리듬을 이용할 경우, 프로그램의 구성상 실행 불가능한 분기 블록들을 제외한 나머지 전체 분기 블록을 커버하였다. 이는 제안된 알고리듬에 의해 작성된 HCFG가 실제 소프트웨어의 구조와 유사함을 의미하므로 소프트웨어 취약성 검증을 위한 모델 기반 테스트 케이스 생성 정책에 제안된 HCFG 생성 알고리듬을 적용 가능하다는 것을 의미한다.

향후, 이 결과를 바탕으로 모델을 효과적으로 생성 할 수 있는 방법에 대하여 연구가 필요하다. 본 논문에서는 HCFG 생성 알고리듬에 대하여 제안할 뿐, 모델의 효과적인 생성을 위한 수단에 대한 고려는 수행하지 않아, 보다 효율적으로 분기 블록을 커버할 수 있도록 하면서 데이터 생성에 장시간을 요구하지 않는 입력 데이터 리스트 생성 방법에 대하여 자세히 연구할 필요가 있다.

참고문헌

[1] P. Godefroid, M.Y. Levin, and D. Molnar. "Automated Whitebox Fuzz Testing," Technical Report MS-TR-2007-58, Microsoft, May 2007.
[2] Jackson, Daniel, and Martin Rinard, "Software

analsis:a roadmap." Processdings of the Conference on the Future of Sftware Engineering, ACM, pp.133-145, May, 2000.

〔3〕 A.Gargantini and C.Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," in Proceedings of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp.146-162, Sep. 1999.

〔4〕 Jungsup Oh Kyunghee Choi Gihyun Jung, "Automatic Test Case Generation Through 1-to-1 Requirement Modeling," KIPS, D, Vol. 17D, No. 1, pp. 41-52, Oct. 2010.

〔5〕 Ki-Tae Kim, Je-Min Kim, Weon-Hee Yoo, "Static Control Flow Anlaysis of Binary Codes," The Korea Contents Association, Vol. 10, No. 5, pp,70-79, May. 2010,

〔6〕 Ajitha Rajan, "Automated requirements-based test case generation," Newsletter of ACM SIGSOFT Software Engineering Notes, Vol.31, No. 6, pp.1-2, Nov. 2006.

〔7〕 The MathWorks, http://www.mathworks.com/

〔8〕 Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Active property checking." Proceedings of the 8th ACM international conference on Embedded software. ACM, pp.207-216, Oct, 2008.

〔9〕 P. Godefroid, A. Kiezun, M. Y. Levin, "Grammar-based Whitebox Fuzzing," in Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, pp,206-215, Jun. 2008.

〔10〕 P S. Anand, P. Godefroid, and N. Tillmann. "Demand-driven compositional symbolic execution," In Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp 367-381, April. 2008.

〔11〕 Zijiang Yang, "Mixed Symbolic Representations for Model Checking Software Programs," in processdings of the Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on, pp17-26, July. 2006.

〔12〕 D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering," Int'l J. Software Tools for Technology Transfer, vol. 9, pp. 505-525, Sep. 2007.

〔13〕 Mark M. Seeger, "Using Control-Flow Techniques in a Security Context A Survey on Common Prototypes and their Common Weakness," in Proceedings of the 2011 International Conference on Network Computing and Information Security, pp.133-137, May. 2011.

〔14〕 IVANCIC, F., YANG, Z., GANAI, M. K., GUPTA, A., AND ASHAR, "Efficient SAT-based bounded model checking for software verification" Theoret. Comput. Sci. 404, 3, pp.256-274. 2008.

〔15〕 Z. Yang, C. Wang und A. Gupta. "Model checking sequential software programs via mixed symbolic analysis," Journal of ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol 14. No. 1, pp.1-26. Jan. 2009.

〔16〕 M. Pezzé and M. Young, Software Testing and Analysis: Process, Principles, and Techniques, Draft Version of 31st, Chap. 14.5, Mar. 2000.

〔17〕 Hangbae Chang, Sang-Soo Yeo, Seong-eon Cho, Heau-Jo Kang, "The Study on Improvement of the Program that Traces the Binary Codes in Execution," Journal of Security Engineering, Vol. 5, No 3, pp209-218, Jun. 2008

〔18〕 M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns", in Proceedings of the 12th USENIX Security Symposium, vol. 12, pp. 169-186, Aug. 2003.

〔19〕 J Kinder, "Static Analysis of x86 Executables," Fachbereich Informatik Technische Universit¨at Darmstadt, Chapter5. Sep. 2010

〔20〕 D. G. Fritz and R. G. Sargent, "An overview of Hierarchical Control Flow Graph Models," in Proceedings of the 1995 Winter Simulation Conference, pp. 1347-1355, Dec. 1995.
〔21〕 C. 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 the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190-200, Jun. 2005.
〔22〕 A Dynamic Binary Instrumentation Tool, http://www.pintool.org/
〔23〕 Hye-ryun Lee, Seung-hun Shin, Kyung-hee Choi, Ki-hyun Chung, Seung-kyu Park, Jun-yong Choi, "Detecting the vulnerability of software with cyclic behavior using Sulley," in Proceedings of the Advanced Information Management and Service (ICIPM), 2011 7th International Conference on, pp,83-88, Dec. 2011.
〔24〕 SMTP Server Download, http://www.codeproject.com/Articles/20604/SMTP-Server

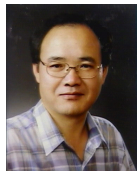
저 자 소 개

이 혜 련

2006: 조선대학교 인터넷소프트웨어공학과 공학사.
2008: 아주대학교 정보통신전문대학원 공학석사.
현 재: 아주대학교 컴퓨터공학과 박사수료
관심분야: 보안테스팅, 소프트웨어공학
Email : cocom12@ajou.ac.kr

신 승 훈

2000: 아주대학교 정보 및 컴퓨터공학부 공학사
2002: 아주대학교 정보통신공학과 공학석사
2011: 아주대학교 정보통신공학과 공학박사
현재: 아주대학교 소프트웨어융합학과 특임교원
관심분야 : 소프트웨어 테스트 자동화, 멀티미디어 서비스 정책 등
Email : sihnsh@ajou.ac.kr



최 경 희

1976: 서울대학교 수학교육과 학사.
1979: 프랑스 그랑데꼴 Enseeiht대학 석사.
1982: 프랑스 Paul Sabatier대학 정보공학부 박사
현 재: 아주대학교 컴퓨터공학과 교수
관심분야: 컴퓨터공학, 운영체제, 실시간시스템, 테스팅 등
Email : khchoi@ajou.ac.kr

정 기 현

1984: 서강대학교 전자공학과 학사.
1988: 미국 Ilinois주립대 EECS(석사)
1990: 미국 Purdue대학 전기전자공학부 박사
현 재: 아주대학교 전자공학과 교수
관심분야: 컴퓨터구조, VLSI 설계, 실시간시스템, 테스팅 등
Email : khchung@ajou.ac.kr

박 승 규

1974: 서울대학교 응용수학과 학사
1976: 한국과학원(KAIST) 전산학과 석사
1982: Institut National Polytechn- ique de Grenoble 전산학과 박사
1976 ~ 1992: KIST, KIET, ETRI 선임/책임연구원
1992 ~ 현재: 아주대학교 소프트웨어융합학과 교수
관심분야 : 임베디드 테스팅, 자가 컴퓨팅/치료 시스템, 차세대 컴퓨터 구조 등
Email : sparky@ajou.ac.kr