情報保護學會論文誌
第 23 卷 第 2 號, 2013. 4

소프트웨어 취약성 평가를 위한 길이기반 파일 퍼징 테스트 슈트 축약 알고리즘

이 재 서,[1†] 김 종 명,[1] 김 수 용,[1] 윤 영 태,[1] 김 용 민,[2‡] 노 봉 남[2]
[1]ETRI 부설연구소, [2]전남대학교

A Length-based File Fuzzing Test Suite Reduction Algorithm for Evaluation of Software Vulnerability

Jaeseo Lee,[1†] Jong-Myong Kim,[1] SuYong Kim,[1] Young-Tae Yun,[1]
Yong-Min Kim,[2‡] Bong-Nam Noh[2]
[1]The Attached Institute of ETRI, [2]Chonnam National University

요 약

최근 소프트웨어의 취약점을 찾기 위해 퍼징과 같은 자동화된 테스팅 방법을 이용한 많은 연구가 진행되고 있다. 퍼징은 소프트웨어의 입력을 특정 규칙에 따라 자동으로 변형시켜 소프트웨어의 오작동 여부를 탐지하고 그 결과로부터 취약점을 발견하는 것이다. 이 때 소프트웨어에 입력되는 입력 값, 즉 테스트 케이스에 따라 취약점을 발견할 수 있는 확률이 달라지기 때문에 취약점 발견 확률을 높이기 위해서는 테스트 케이스의 집합인 테스트 슈트 축약 문제를 해결하여야 한다. 이에 본 논문에서는 파일과 같은 대용량 테스트 케이스를 대상으로 효과적으로 테스트 슈트 축약 문제를 해결할 수 있는 방법을 제안하고자 한다. 이를 위해 기존 연구에서 주로 사용되었던 커버리지와 중복도 이외에 새로운 척도인 테스트 케이스의 길이를 제시하고, 본 척도에 적합한 축약 알고리즘을 설계하였다. 실험을 통해 본 논문에서 제안한 알고리즘이 기존 연구의 알고리즘보다 높은 크기와 길이 축약율을 나타냄을 보임으로써 제안하는 알고리즘의 효율성을 증명할 수 있었다.

ABSTRACT

Recently, automated software testing methods such as fuzzing have been researched to find software vulnerabilities. The purpose of fuzzing is to disclose software vulnerabilities by providing a software with malformed data. In order to increase the probability of vulnerability discovery by fuzzing, we must solve the test suite reduction problem because the probability depends on the test case quality. In this paper, we propose a new method to solve the test suite reduction problem which is suitable for the long test case such as file. First, we suggested the length of test case as a measure in addition to old measures such as coverage and redundancy. Next we designed a test suite reduction algorithm using the new measure. In the experimental results, the proposed algorithm showed better performance in the size and length reduction ratio of the test suite than previous studies. Finally, results from an empirical study suggested the viability of our proposed measure and algorithm for file fuzzing.

Keywords: Test Suite Reduction, File Fuzzing, Software Vulnerability

접수일(2013년 3월 6일), 게재확정일(2013년 3월 18일)
† 주저자, mirr1004@ensec.re.kr
‡ 교신저자, ymkim@jnu.ac.kr

I. 서 론

최근 소프트웨어 테스팅 기술의 하나인 퍼징을 이용하여 취약점을 찾으려는 많은 연구[1-4]가 진행되고 있으며, 실제로 취약점[5-7]이 다수 발견되고 있다. 퍼징은 자동화된 방법으로 빠르게 취약점을 찾을 수 있는 장점이 있다[8]. 이러한 이유로 마이크로소프트사와 어도비사 같은 소프트웨어 회사들은 퍼징으로 자사 소프트웨어의 취약점을 찾아 패치하고 있다[1,2]. 퍼징은 소프트웨어의 입력 값을 특정 규칙에 따라 자동으로 변형시켜 소프트웨어의 오작동 여부를 탐지하고 그 결과로부터 취약점을 발견한다. 이 때 소프트웨어에 입력되는 값, 즉 테스트 케이스에 따라 취약점을 발견할 수 있는 확률이 달라지며 취약점 발견 확률을 높이기 위해서는 테스트 슈트 축약 문제를 해결해야 한다. 이에 본 논문에서는 테스트 슈트 축약 문제에서 파일과 같은 대용량 테스트 케이스를 효과적으로 축약할 수 있는 방법을 연구한다.

테스트 슈트 축약 문제는 테스트 슈트에서 불필요한 테스트 케이스를 제거하는 방법으로 테스팅에 적합한 최적의 부분 집합을 찾는 것이다. 이는 축약된 일부 테스트 케이스만을 테스팅하는 것으로 테스트 슈트의 전체를 테스팅하는 것과 유사한 효과를 낼 수 있기 때문에 소프트웨어 테스팅 분야에서 활발히 연구되어 왔다[9]. 기존의 테스트 슈트 축약에 관한 연구는 퍼징과 같이 하나의 테스트 케이스에 대해서 반복적으로 테스팅하는 방법에 대해서는 고려하지 않고 있다. 이는 하나의 테스트 케이스에 대해서 단 한번 실행하여 오동작 여부를 검증하는 방식의 테스팅에서는 테스트 케이스의 길이(length)는 중요한 변수가 되지 않기 때문이다. 왜냐하면, 길이가 짧은 테스트 케이스를 실행할 때 소요되는 시간과 길이가 긴 테스트 케이스를 실행할 때 소요되는 시간이 거의 차이가 나지 않기 때문이다. 반면 퍼징과 같이 하나의 테스트 케이스에 대해서 조금씩 변형시키면서 반복적으로 실행하며 오동작 여부를 검증하는 방식의 테스팅인 경우 테스트 케이스의 길이에 비례하여 퍼징에 소요되는 시간이 늘어나므로 테스트 케이스의 길이는 매우 중요한 변수가 된다. 따라서 파일 퍼징에 적합한 테스트 슈트 축약이 되기 위해서는 테스트 케이스의 길이가 고려될 필요가 있다.

본 논문에서는 기존 연구에서 주로 사용 되었던 커버리지와 중복도 이외에 이전 연구[10]에서 새롭게 제시한 척도인 테스트 케이스의 길이를 소개하고, 본 척도에 적합한 축약 알고리즘을 제안한다. 제안하는 축약 알고리즘의 효율성을 증명하기 위해 기존 연구인 HGS[11], GRE[12] 그리고 BOG[13] 알고리즘과 크기 및 길이 축약율 비교 실험을 진행하고 퍼징 시뮬레이션을 통해 제안하는 알고리즘에 의해 축약된 테스트 케이스 집합이 퍼징에 가장 효율적임을 보인다.

본 논문의 2장에서는 테스트 슈트 축약 문제를 소개하고 이와 관련된 선행 연구들을 설명한다. 3장에서는 새로운 척도인 테스트 케이스의 길이를 제시하고, 제안하는 척도에 적합한 축약 알고리즘을 제안한다. 4장에서 실험을 통해 제안하는 알고리즘의 우수성을 증명한다. 마지막으로 5장에서 결론 및 향후 연구에 대해 기술한다.

II. 관련 연구

본 장에서는 테스트 슈트 축약 문제를 소개하고 이와 관련된 선행 연구들을 설명한다.

2.1 테스트 슈트 축약 문제

테스트 슈트 축약(reduction)은 테스트 슈트 선정(selection), 테스트 슈트 우선순위 결정(prioritization)과 함께 소프트웨어 테스팅에 소요되는 비용은 최소화하면서 그 성과는 최대화하기 위한 하나의 방법이다[9]. 테스트 슈트 축약 문제에 대한 정의는 Harrold, Gupta, 그리고 Soffa에 의해 1993년 처음 정형화 되었다[11]. 이후 2002년 Rothermel[14]에 의해 보다 정형화된 정의가 이루어졌으며 이는 [그림 1]과 같다.

2.2 선행 연구

테스트 슈트 축약 문제는 NP-complete이다[9,11]. 따라서 부분 최적 해를 찾기 위해 휴리스틱 그리디(heuristic greedy) 알고리즘, 유전적

입력 조건: T, 테스트 케이스 모음(suite) $\{t_1,\ldots,t_m\}$
R, 실험 요구(requirement) 집합 $\{r_1,\ldots,r_n\}$,
$T_1,\ldots,T_n$, 실험 요구 r_i를 만족하는 테스트 케이스의 집합
문제: 테스트 케이스 집합 T로부터 모든 실험 요구를 만족하는 대표 집합 T'찾기

[그림 1] 테스트 슈트 축약 문제에 대한 정의

(genetic) 알고리즘, 그리고 정수 선형 계획(integer linear programming) 알고리즘 기반으로 활발히 연구 되고 있다. Zhong은 앞의 세 가지 기반 알고리즘별 대표적인 연구[11,12,16,17,18]를 대상으로 축약율과 성능에 대해 실험했다. 실험에서 유전적 알고리즘은 축약율과 성능 모두에서 가장 효율적이지 못했고 휴리스틱 그리디 알고리즘에 기반한 방법은 정수 선형 계획 알고리즘에 기반한 방법과 비교하여 개수 축약율은 비슷한 결과를 보인 반면 성능에 있어서는 가장 우수한 결과를 보였다[15]. 테스트 슈트 축약에 관한 연구의 대부분이 휴리스틱 그리디 알고리즘에 기반하고 있다는 점과 Zhong의 실험 결과를 바탕으로 본 논문에서는 휴리스틱 그리디 알고리즘을 기반으로 연구를 진행한다.

2.2.1 기존 연구

본 절에서는 기존 연구 중에서 제안하는 알고리즘의 비교 실험에 대해서 설명한다. 비교 실험 대상 알고리즘에 대한 선정 기준은 문제에 대한 접근 방법을 달리한 알고리즘, 즉 새로운 척도가 제시된 알고리즘을 대상으로 결정하였으며, 이에 해당하는 기존 연구는 [표 1]과 같이 HGS, GRE 그리고 BOG이다.

첫 번째, HGS는 테스트 슈트 축약에 있어 필수성과 커버리지 증가도를 척도로 이용하였다. 여기에서 필수성은 테스트 케이스가 반드시 선정되어야만 하는 정도를 의미한다. 그리고 커버리지 증가도는 어떤 테스트 케이스에 의해서, 만족되지 않은 테스팅 요구(requirement)들 중에서 추가적으로 만족되는 정도를 의미한다. HGS는 반드시 선정되어야 하는 필수 테스트 케이스를 우선적으로 모두 선정한다. 그 다음부터는 선정될 확률이 1/2인 테스트 케이스들 중에서 가장 높은 커버리지 증가도를 보이는 테스트 케이스를 선정하다. 이 과정에서 확률이 1/2인 테스트 케이스가 더 이상 존재하지 않으면 그 다음으로 선정될 확률이 1/3인 테스트 케이스를 대상으로 선정을 진행한다. 이와 같은 방법으로 모든 테스팅 요구가 만족될 때까지 선정 과정을 반복하여 진행한다[11].

두 번째, GRE는 필수성과 커버리지 증가도 이외에 추가적으로 요구 중복여부를 척도로 이용하였다. 요구 중복여부는 어떤 테스트 케이스가 만족하는 테스팅 요구가 다른 테스트 케이스가 만족하는 테스팅 요구와의 포함 관계를 의미한다. 어떤 테스트 케이스가 다른 테스트 케이스와 테스팅 요구가 완전히 중복되면

[표 1] 기존 연구의 테스트 슈트 축약 척도 분류

알고리즘	척도1	척도2	척도3	척도 관계 (우선순위)
HGS	필수성	커버리지 증가도	--	척도1〉척도2
GRE	필수성	커버리지 증가도	요구 중복여부	척도1〉 (척도2〉척도3)
BOG	커버리지 증가도	요구 중복도	--	척도1＝척도2

해당 테스트 케이스를 제거한다. GRE는 현재 선정되지 않은 테스트 케이스들 중에서 필수적인 테스트 케이스를 모두 선정한다. 그리고 나머지 테스트 케이스들을 커버리지 증가도를 기준으로 내림차순 정렬을 하고 차례대로 테스팅 요구 중복여부를 확인하며 중복되는 테스트 케이스를 제거한다. 더 이상 테스팅 요구가 중복되는 테스트 케이스가 없으면 다시 필수적인 테스트 케이스를 모두 선정한다. 이와 같은 방법으로 모든 테스팅 요구가 만족될 때까지 선정 과정을 반복하며 진행한다[12].

세 번째, BOG는 커버리지 증가도와 요구 중복도를 척도로 이용하였다. 요구 중복도는 앞에서 설명한 요구 중복여부와 유사하지만, 참 또는 거짓으로 표현되는 요구 중복여부와는 달리 요구의 중복되는 정도를 의미한다. BOG는 현재 선정되지 않은 테스트 케이스들을 대상으로 가장 높은 커버리지 증가도를 가지는 테스트 케이스들로 최대 집합을 만든다. 그 다음으로 가장 낮은 테스팅 요구 중복도를 가지는 테스트 케이스들로 최소 집합을 만든다. 이렇게 만들어진 최대 집합과 최소 집합에 포함된 모든 테스트 케이스들을 대상으로 각각 커버리지 증가도와 요구 중복도의 거리(distance)를 계산하여 거리 값이 가장 큰 테스트 케이스를 선정한다. 이와 같은 방법으로 모든 테스팅 요구가 만족될 때까지 선정 과정을 반복하며 진행한다[13].

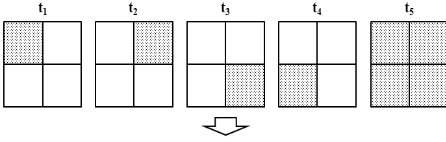
2.2.2 기존 연구의 한계

[그림 2]는 4개의 함수로 구성된 매우 간단한 프로그램 P를 대상으로 각각의 함수 f_i를 테스팅 요구 r_i로 정의하고, 다섯 개의 테스트 케이스를 원소로 갖는 집합 T를 입력(input)으로 기존 방법을 통해 축약한 결과를 나타낸다. 기존의 축약 알고리즘들은 모두 동일하게 테스트 케이스 t_5 하나만을 원소로 갖는 집합 T′로 축약하였다. 본 결과는 단순히 축약된 집합 T′의

Program P: A simple program which consists of 4-functions

f_1	f_2
f_3	f_4

Function coverage of each test case



Input: Test Suite T := {t_1, t_2, t_3, t_4, t_5}
R := {r_1, r_2, r_3, r_4} = {f_1, f_2, f_3, f_4}
T_1 := {t_1, t_5}, T_2 := {t_2, t_5}, T_3 := {t_3, t_5}, T_4 := {t_4, t_5}
t_1's length := 1, t_2's len. := 1, t_3's len. := 1, t_4's len. := 1, t_5's len. := 10

Output: // Reduced set T'
by HGS: T' := {t_5}
by GRE: T' := {t_5}
by BOG: T' := {t_5}

〔그림 2〕 4개의 함수로 구성된 간단한 프로그램을 대상으로 테스트 슈트 T를 기존 알고리즘으로 축약한 결과

크기 축약율만을 고려한다면, 모두 최적해를 찾은 것이라 할 수 있다. 그러나 테스트 케이스의 길이를 고려한다면 이는 최적해라고 할 수 없다.

예를 들어, 집합 T에서 축약될 수 있는 다른 테스트 케이스 집합 T″:={t_1,t_2,t_3,t_4}가 있을 때 집합 T′의 모든 원소 테스트 케이스의 길이를 계산하면 테스트 케이스 t_5의 길이인 10이 된다. 반면에 집합 T″의 모든 원소 테스트 케이스의 길이는 4가 된다. 이 경우 집합 T′와 집합 T″의 원소를 대상으로 파일 퍼징을 수행한다면 T′을 퍼징하는데 소요되는 시간이 T″를 퍼징하였을 때 소요되는 시간보다 약 2.5(10/4)배가 더 오래 걸리게 된다.

이처럼 기존의 축약 알고리즘은 파일 퍼징을 위한 테스트 슈트 축약에 있어 테스트 케이스의 길이를 고려하고 있지 않다는 한계가 있다. 이에 본 논문에서는 파일과 같이 테스트 케이스의 길이가 긴 테스트 케이스를 대상으로 축약된 집합의 크기와 길이 모두에 있어서 효율적인 새로운 축약 알고리즘을 제안한다.

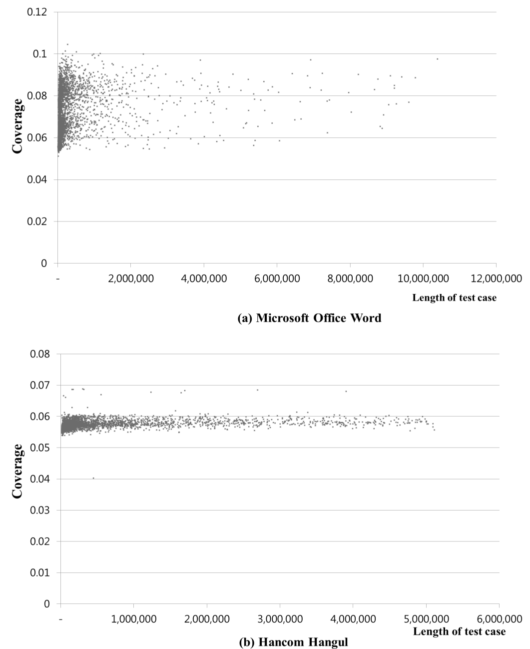
III. 제안 연구

본 장에서는 제안하는 척도와 해당 척도를 기반으로 하는 테스트 슈트 축약 알고리즘을 설명한다. 이에 앞써 파일의 크기(size)와 테스트 슈트의 크기(cardinality)간에 용어 혼란을 피하기 위해 테스트 케이스의 크기(size) 대신에 테스트 케이스의 길이(length)를 용어로 이용한다.

3.1 제안 척도: 테스트 케이스의 길이

인터넷에서 무작위로 수집한 테스트 케이스의 길이와 커버리지(coverage) 측정을 통해 길이와 커버리지간의 관계를 분석하였으며 측정 결과는 〔그림 3〕과 같다. 〔그림 3〕은 마이크로소프트사 워드와 한글과 컴퓨터사 한글의 테스트 케이스 길이(x-축)와 해당 테스트 케이스의 함수 커버리지(y-축)의 관계를 나타낸 것으로 커버리지는 테스트 케이스의 길이와 관계가 없음을 의미한다. 반면에 테스트 케이스의 길이는 퍼징에 소요되는 시간을 결정하는 데에 있어서는 밀접한 관련이 있다. 예를 들어 bit-flipping 퍼징 기법[19]은 테스트 케이스의 시작 비트에서부터 마지막 비트에 이르기까지 차례대로 반복하며 퍼징을 수행한다. byte-transposing 퍼징 기법[20]은 시작 바이트에서부터 마지막 바이트에 이르기까지 인접 바이트와 값을 바꾸어 가며 반복하여 퍼징을 수행한다. 따라서 테스트 케이스의 길이가 길면 길수록 퍼징에 소요되는 시간도 비례하여 증가한다.

이러한 테스트 케이스의 길이와 커버리지 및 퍼징 소요시간과의 관계는 퍼징에 최적화된 테스트 슈트 축약에 있어 테스트 케이스의 길이가 고려되어야 하는



〔그림 3〕 (a)MS 워드 프로그램의 입력 파일인 DOC 파일과 (b)한글 프로그램의 입력 파일인 HWP 파일을 대상으로 길이별 커버리지를 측정한 결과

```
input     t1,t2,…,tm: all test cases which present in the test suite
          r1,r2,…,rn: all requirements
          lv[m]: array representing length of each test case
          cm[m,n]: coverage matrix representing requirement coverage of each test case,
                   1 for covered and 0 for uncovered
output    RS: a reduced suite of test cases from the test pool
declare   nextTest: one of test cases
          sumRows: array[1..n] of integer, representing the covering if requirements, initially 0
          essentialList, unessentialList, maxList: list of ti's
          value: a value of integer
          Card(): returns the cardinality of a set
          Min(): returns the minimum of a set of numbers
          Coverage(): returns the coverage of the test case

algorithm ReduceTestSuite
begin
STEP 1: initialization
          RS := { t1,t2,…,tm }
          foreach ri do compute sumRows[i], sum of the elements in the ith column of the cm
STEP 2:
          loop
            foreach ti ∈ RS do
              foreach rj ∈ requirements where cm[ti,rj] = 1 do
                value := sumRows[j] - the j-th column element in the i-th row of the cm
                if value == 0 then essentialList := {ti}
              endfor
            endfor
            unessentialList = RS - essentialList
            Construct maxList consisting of test cases from unessentialList for which lv[i] is the
            maximum
            if Card(maxList) = 1 then
              nextTest := the test case in maxList
            else
              nextTest := Min(Coverage(each test case in maxList))
STEP 3:     RS := RS - {nextTest}
            foreach rj ∈ requirements where cm[nextTest,rj] = 1 do
              sumRows[j] := sumRows[j] - 1
            endfor
            Clear all elements of essentialList
          until Card(unessentialList) = 0
          return RS
```

〔그림 4〕 제안하는 GF3 알고리즘 의사코드

당위성으로 해석이 가능하다. 이로써 테스트 케이스의 길이는 최소이면서도 커버리지는 높은 테스트 케이스를 선별하고 이렇게 선별된 테스트 케이스들을 대상으로 퍼징한다면, 퍼징에 소요되는 시간은 최소화하면서 빠르게 커버리지를 높일 수 있는 효율적인 퍼징이 될 것이다.

이러한 이유로 이전 연구[10]에서 기존 연구에서 주로 사용 되었던 커버리지와 중복도 이외에 테스트 케이스의 길이를 테스트 슈트 축약의 척도로 제안하였다. 이전 연구는 커버리지, 중복도 그리고 테스트 케이스의 길이를 기반으로 9개의 비교 척도를 정의하고, 이를 기준으로 테스트 케이스를 우선 선정하였을 때 선정되는 테스트 케이스에 의해 증가되는 커버리지를 비교 실험한 것이다. 실험을 통해 제안한 테스트 케이스의 길이가 고려되었을 때 기존 척도만을 이용할 때 보다 빠르게 커버리지가 증가됨을 알 수 있었다.

3.2 제안 테스트 슈트 축약 알고리즘

제안하는 테스트 슈트 축약 알고리즘의 이름은 GF3(Greedy for File Fuzzing)이며 의사 코드는 〔그림 4〕와 같다. GF3 알고리즘의 입력(input)은 테스트 케이스들의 길이로 구성되는 *lv*(length vector)배열과 BOG 알고리즘에서의 테스트 케이스와 테스팅 요구(requirement)간의 관계를 나타내는 행렬인 *cm*(coverage matrix)이다. *cm*행렬은 테스트 케이스 집합의 크기인 *m*행과 테스팅 요구 집합의 크기인 *n*열로 이루어진다. 본 논문에서는 커버리지 척도(metric)로 함수 커버리지(function coverage)를 이용하기 때문에 테스팅 요구는 함수(function)를 의미한다. *cm*행렬의 원소는 0과 1의 값으로 구성되며, *i*-번째 테스트 케이스를 입력으로 대상 소프트웨어를 실행하였을 때 소프트웨어의 *j*-번째 함수가 호출되어 실행되면 *cm*〔*i,j*〕의 값은 1로 채워지며 그렇지 않은 경우 0으로 채워진다.

3.2.1 알고리즘 설명

GF3 알고리즘은 3단계로 이루어지며 6개의 변수와 3개의 함수로 구성된다. 1단계에서는 먼저 모든 테스트 케이스를 테스트 케이스 집합인 *RS*에 넣는다. 그 다음으로 테스팅 요구별로 만족하는 테스트 케이스의 크기를 의미하는 *sumRows* 배열을 구성한다. 예를 들어, *i*-번째 테스팅 요구를 테스트 케이스 a, b 두 개가 만족한다면 *sumRows*〔*i*〕의 값은 2가 된다.

2단계에서는 더 이상 불필요한 테스트 케이스가 없을 때까지 반복하며 불필요한 테스트 케이스를 선정한다. 먼저 *RS* 집합에 들어 있는 모든 테스트 케이스를 대상으로 필수적으로 선택되어야 하는지를 확인하고 반드시 선택되어야 하는 필수 테스트 케이스인 경우 *essentialList* 집합에 넣는다. 다음으로 *RS* 집합에서 *essentialList* 집합에 포함되지 않은 테스트 케이스들인 즉, 현재 단계에서 필수적이지 않은 테스트 케이스들을 *unessentialList* 집합에 넣는다. 그리고 *unessentialList* 집합에 포함되어 있는 테스트 케이스 중에서 테스트 케이스의 길이가 가장 큰 테스트 케이스들을 *maxList* 배열에 넣는다. 마지막으로 *maxList* 집합에 포함되어 있는 테스트 케이스가 오직 한 개뿐이면 해당 테스트 케이스를 제거할 *nextTest*로 선정하고 두 개 이상이면 해당 테스트 케이스들 중에서 커버리지가 가장 낮은 테스트 케이스를 *nextTest*로 선정한다.

3단계에서는 2단계에서 선정된 *nextTest* 테스트 케이스를 *RS* 집합에서 제거한다. 다음으로 제거된 *nextTest* 테스트 케이스를 제외하고 *sumRows* 배열을 재계산하고 다시 2단계를 진행한다.

3.2.2 함수 설명

Card() 함수는 입력된 집합에서 원소의 크기를 계산하여 반환한다. *Min*() 함수는 입력된 집합의 원소들 중에서 가장 작은 값을 찾아 반환한다. 마지막 *Coverage*()는 입력된 테스트 케이스에 의해 실행되는 함수의 개수인 커버리지를 반환한다.

IV. 실험 및 평가

실험에 이용한 소프트웨어는 국내・외에서 가장 널리 사용되는 소프트웨어 중에서 파일을 입력으로 하는 문서 관련 소프트웨어 3종과 RIA(Rich Internet Applications) 관련 소프트웨어 1종이다. 실험에 이용한 각 소프트웨어의 테스트 케이스는 인터넷에서 무작위로 수집하여 구성하였으며 〔표 2〕와 같다. 〔표 2〕에서의 크기는 개수를 의미하며 길이는 수집한 파일들의 크기(size) 합을 의미한다.

4.1 크기 및 길이 축약율 실험

축약율 실험은 원본 테스트 케이스 집합($T_{original}$)에서 각각의 축약 알고리즘을 적용하여 축약된 테스트 케이스 집합($T_{reduced}$)을 만들고, 축약된 테스트 케이스 집합의 크기와 길이를 원본 테스트 케이스 집합의 크기, 길이와 각각 비교하여 얼마만큼 더 많이 축약되었는가를 비교 분석하는 것으로 이루어진다.

〔표 2〕 실험 대상 소프트웨어 4종 및 수집 테스트 케이스

소프트웨어	확장자	버전	수집 테스트 케이스	
			크기(개수)	길이
MS Office Word Viewer SP3	DOC	11.8169.8172	8,883	2.54GB
Hancom Office 2010 Hangul	HWP	8.0.0.466	9,362	3.81GB
Adobe Flash Player	SWF	11.0.1.152	13,220	10.0GB
Adobe Reader X	PDF	10.0.1.0	17,038	10.6GB

〔표 3〕 크기 및 길이 축약율 실험 결과

소프트웨어	단위 구간	$T_{original}$ (MB)	HGS		GRE		BOG		GF3	
			S%	L%	S%	L%	S%	L%	S%	L%
DOC	B1	143	78.13	57.42	76.87	55.29	45.47	29.60	77.73	61.01
	B2	275	86.73	75.26	85.30	71.40	63.33	55.08	86.27	80.75
	B3	432	89.09	77.20	88.40	74.57	70.31	58.77	88.82	81.18
	B4	642	91.53	82.99	90.87	82.22	76.17	71.38	91.17	86.06
	B5	1,415	95.48	90.90	95.14	89.82	87.20	84.45	95.44	93.14
HWP	B1	202	85.53	77.31	84.33	74.65	66.87	65.26	85.47	82.12
	B2	427	92.00	87.15	90.97	84.15	79.73	81.89	91.53	89.55
	B3	635	93.58	89.22	93.11	87.90	84.56	85.64	93.53	91.88
	B4	844	94.43	90.34	93.93	89.29	87.77	87.49	94.32	93.48
	B5	2,058	97.26	95.10	97.06	94.38	93.96	93.24	97.26	95.96
SWF	B1	588	80.93	85.55	79.53	84.80	48.73	69.68	80.73	86.55
	B2	1,210	87.00	88.15	86.13	86.73	61.50	75.16	86.93	89.48
	B3	1,870	90.31	90.87	90.09	90.44	69.84	80.55	90.31	92.19
	B4	2,411	92.13	92.30	91.57	91.81	74.48	83.14	91.97	92.85
	B5	6,269	95.82	95.55	95.64	95.23	85.76	90.06	95.70	96.33
PDF	B1	406	75.07	69.42	73.73	68.72	48.33	42.60	75.20	73.90
	B2	801	83.63	79.26	82.97	78.37	64.83	56.52	83.70	82.14
	B3	1,265	87.51	85.01	86.73	83.95	72.11	67.91	87.38	88.92
	B4	1,650	89.10	87.34	88.12	86.07	76.43	72.73	89.13	90.56
	B5	4,051	94.14	93.36	93.70	92.94	87.00	84.78	94.04	95.13

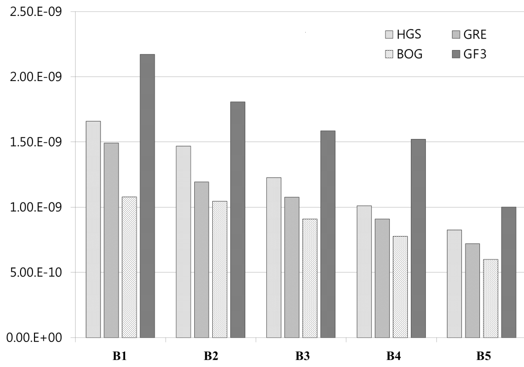
실험을 위해 테스트 케이스 개수를 기준으로 B1 구간부터 B5 구간까지 5개의 단위 구간으로 나누었다. B1 구간은 500개, B2 구간은 1,000개, B3 구간은 1,500개, B4 구간은 2,000개, 마지막 B5 구간은 5,000개이며, 〔표 2〕에서의 수집 테스트 케이스를 이용하여 각 구간별로 3개씩의 테스트 케이스 집합을 임의로 만드는 방법으로 데이터셋을 구성하였다. 또한, 테스트 케이스 각각의 함수 커버리지 측정을 위하여 Pintool〔21〕을 이용하여 call 명령(instruction)을 추적하였으며, 테스트 케이스 마다의 호출 함수 데이터베이스를 구성하였다. 이렇게 구성한 데이터셋과 호출 함수 데이터베이스를 이용하여 HGS, GRE, BOG 그리고 GF3 알고리즘별로 각각 크기 축약율과 길이 축약율을 실험하였으며 그 결과는 〔표 3〕과 같다.

$$S = (1 - \frac{|T_{reduced}|}{|T_{original}|}) \times 100 \quad (1)$$

$$L = (1 - \frac{\sum length(t_i \in T_{reduced})}{\sum length(t_j \in T_{original})}) \times 100 \quad (2)$$

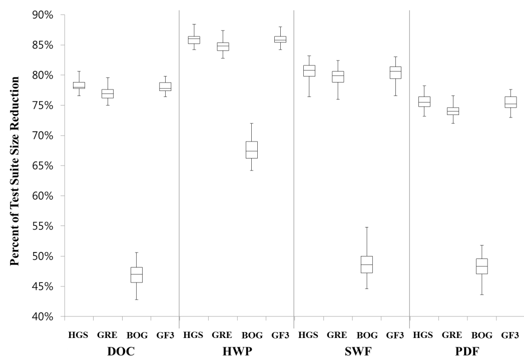
〔표 3〕에서의 기호 $T_{original}$는 각 테스트 케이스 구간에 포함되어 있는 테스트 케이스들의 길이를 모두 합한 값이며 그 단위는 mega-byte이다. 기호 S%는 크기 축약율을 의미하며 그 계산식은 〔식 1〕과 같다. 그리고 기호 L%는 길이 축약율을 의미하며 계산식은 〔식 2〕와 같다. 크기 및 길이 축약율은 각각 세 번에 걸쳐 실험한 결과의 평균이다.

실험 결과, 〔표 3〕에서와 같이 본 논문에서 제안하는 GF3 알고리즘을 이용하여 축약하였을 때 모든 단위 구간과 대상 소프트웨어에 대해 길이 축약율이 가장 높게 나타났다. 그리고 크기 축약율에서도 모든 실험에서 GRE와 BOG보다 높은 축약율을 보였으며 HGS와는 비슷한 축약율을 보였다. 이는 제안하는 알고리즘인 GF3에 의해 축약된 테스트 케이스 집합이 파일 퍼징에 있어서 가장 유리한 축약 알고리즘이라는 것을 의미한다. 이를 뒷받침하기 위해 소프트웨어 한글을 대상으로 실험한 결과를 각각의 알고리즘별로 축약된 테스트 케이스의 단위 바이트당 커버리지를 계산하였으며 〔그림 5〕와 같다. x-축은 테스트 케이스 단위 구간이며 y-축은 단위 바이트당 커버리지이다. 〔그림 5〕에서 보이는 것처럼 축약된 테스트 케이스의 단위 바이트당 커버리지는 제안하는 GF3 알고리즘을 이용하였을 때 가장 높게 나타났다. 이는 파일 퍼징을 수행하는 과정에서 커버리지가 가장 빠르게 높아진다는 것을 의미하기 때문에 취약점을 발견할 확률도 높아진다고 해석할 수 있다.

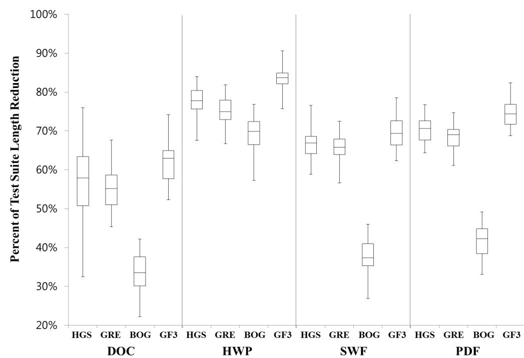


〔그림 5〕 축약된 테스트 케이스의 단위 바이트당 커버리지

〔그림 6, 7〕은 대상 소프트웨어와 알고리즘별 크기 축약율과 길이 축약율에 대한 boxplot 분산 그래프이다. boxplot 그래프는 테스트 케이스 축약에 관한 연구 분야에서 실험 결과를 가시적으로 보이기 위한 방법으로 가장 널리 사용된다. boxplot 그래프는 소프트웨어마다 4개의 박스로 구성된다. 가장 왼쪽 박스부터 차례대로 HGS, GRE, BOG 알고리즘의 결과를 의미하며, 가장 오른쪽 박스가 본 논문에서 제안하는 GF3 알고리즘의 결과에 해당한다.



〔그림 6〕 크기 축약 결과에 대한 boxplot 그래프



〔그림 7〕 길이 축약 결과에 대한 boxplot 그래프

본 실험을 위해 소프트웨어별로 B1 단위 구간 크기의 테스트 케이스 집합을 50개씩 임의로 생성하였으며, 본 데이터셋을 이용하여 각 알고리즘별로 각각 크기 및 길이 축약 실험을 진행하였으며 그 결과는 각각 〔그림 6, 7〕과 같다. 실험 결과, 제안한 알고리즘을 이용하여 축약하였을 때 나타나는 크기 축약율의 분산도는 〔그림 6〕에서와 같이 HGS의 크기 축약율 분산도와 비슷한 수준의 결과를 보였으며 GRE 및 BOG의 크기 축약율 분산도와 비교하여 높은 크기 축약율에서 낮은 분산도를 보이는 것으로 나타났다. 또한, 길이 축약율의 분산도는 〔그림 7〕에서와 같이 다른 모든 알고리즘과 비교하여 높은 길이 축약율에서 낮은 분산도를 보였다. 이 결과는 제안하는 알고리즘이 테스트 케이스의 구성에 관계없이 일반적으로 보다 높은 축약율을 보인다는 것을 의미하며 분산도가 낮다는 것은 축약의 신뢰성이 높다는 것으로 해석 가능하다.

퍼징 시뮬레이션 실험

길이 축약율 실험 결과는 본 논문에서 제안한 GF3 알고리즘을 이용하여 축약한 테스트 케이스 집합을 실제로 퍼징하였을 때 퍼징에 소요되는 시간이 가장 짧다는 것을 의미한다. 이는 한정된 시스템과 시간 자원 하에서 가장 많은 퍼징을 수행할 수 있다는 것을 의미하는 것이다. 본 실험에서는 실제 퍼징을 수행하는 것은 아니며, 한정된 시스템 자원과 시간이 주어졌을 때 얼마나 더 빠르게 퍼징이 완료되어 자원과 시간을 절약할 수 있는지를 비교 분석한다.

$$fuzzable\ length = mp/ct \quad (3)$$

〔식 3〕은 가용할 수 있는 시스템과 시간 자원에서 퍼징 가능한 파일 길이의 계산식이다. m은 머신(machine)의 약어이며, 시스템 자원의 개수를 의미한다. p는 기간(period)의 약어로 시간 자원을 의미한다. t는 횟수(times)의 약어로 1 바이트를 퍼징할 때에 퍼징 기법마다 요구되는 퍼징 횟수를 의미한다. c는 시간 비용(cost)의 약어로 1회 퍼징에 소요되는 시간(단위, sec)을 의미한다. 본 퍼징 시뮬레이션의 실험 조건은 Uhley의 조건과 동일하다. Uhley〔2〕의 조건인 약 2,000개의 시스템(m := 2,000)을 이용하여 3주간(p := 1,814,400sec) bit-flipping 기법(t := 8 times/byte)으로 퍼징하며 1회 퍼징에

약 5초(c := 5)가 소요된다고 가정하였다. 이 때 퍼징 기법별로 퍼징 가능한 테스트 케이스의 길이는 〔식 3〕으로 계산할 수 있으며, bit-flipping 퍼징 기법인 경우 약 88,594KB(kilobyte)가 퍼징 가능한 길이가 된다. 그리고 추가로 byte-increasing 퍼징 기법도 추가로 고려하였으며, 본 퍼징 기법으로 퍼징 가능한 테스트 케이스의 길이는 약 2,779KB가 된다. byte-increasing 퍼징 기법은 각각의 바이트에 대해서 반복적으로 1부터 255의 값을 차례대로 더하는 방법(t := 255)으로 퍼징하는 방법이다.

퍼징 시뮬레이션은 4.1절 축약율 실험의 결과로 축약된 테스트 케이스 집합을 대상으로 퍼징하였을 때 테스트 케이스 집합의 전체 길이에서 퍼징된 길이의 비율을 계산한 것으로 그 결과는 〔표 4〕와 같다. 〔표 4〕에서의 기호 F1은 bit-flipping 퍼징 기법으로 퍼징하였을 때의 퍼징 진행도이며, 기호 F2는 byte-increasing 퍼징 기법으로 퍼징하였을 때의 퍼징 진행도이다. 시뮬레이션 결과, 제안하는 GF3 알고리즘으로 축약된 테스트 케이스 집합을 대상으로 퍼징하였을 때 다른 알고리즘과 비교하여 가장 높은 퍼징 진행도를 보였다. 이는 한정된 시스템과 시간 자원하에서 가장 빠르게 퍼징을 완료할 수 있다는 것을 의미하며 그 만큼 취약점을 발견할 수 있는 확률을 높일 수 있는 것으로 해석할 수 있다.

V. 결 론

자동화된 테스팅 방법을 통해 취약점 발견 확률을 높이기 위해서는 테스트 케이스 집합인 테스트 슈트 축약 문제를 해결하여야 한다. 이에 본 논문에서는 테스트 케이스 축약 문제에서 파일과 같은 대용량 테스트 케이스를 대상으로 효과적으로 축약이 가능하도록 하기 위하여 새로운 척도인 테스트 케이스의 길이를 제시하였고, 제안하는 척도에 적합한 축약 알고리즘을 제안하였다.

제안한 알고리즘의 효율성을 증명하기 위해 기존의 대표적인 HGS, GRE 그리고 BOG 알고리즘과 비교 실험하였다. 실험 대상으로 국내・외에서 가장 많이 사용되는 문서 관련 소프트웨어 3종과 RIA 관련 소프트웨어 1종을 선정 하였고, 각 소프트웨어별로 실험 데이터셋을 인터넷에서 무작위로 수집한 테스트 케이스를 이용하여 구성하였다. 또한 실험의 객관성을 높이기 5개의 단위 구간을 나누었으며 단위 구간별로 각각 3번씩 실험하였다.

〔표 4〕 퍼징 시뮬레이션 결과

(단위, 퍼징 진행도)

소프트웨어	단위구간	HGS		GRE		BOG		GF3	
		F1	F2	F1	F2	F1	F2	F1	F2
DOC	B1	1.00	0.04	1.00	0.04	0.86	0.03	1.00	0.05
	B2	1.00	0.04	1.00	0.03	0.70	0.02	1.00	0.05
	B3	0.87	0.03	0.78	0.02	0.48	0.02	1.00	0.03
	B4	0.79	0.02	0.76	0.02	0.47	0.01	0.96	0.03
	B5	0.67	0.02	0.60	0.02	0.39	0.01	0.89	0.03
HWP	B1	1.00	0.06	1.00	0.05	1.00	0.04	1.00	0.08
	B2	1.00	0.05	1.00	0.04	1.00	0.04	1.00	0.06
	B3	1.00	0.04	1.00	0.04	0.95	0.03	1.00	0.05
	B4	1.00	0.03	0.96	0.03	0.82	0.03	1.00	0.05
	B5	0.86	0.03	0.75	0.02	0.62	0.02	1.00	0.03
SWF	B1	1.00	0.03	0.97	0.03	0.48	0.02	1.00	0.03
	B2	0.60	0.02	0.54	0.02	0.29	0.01	0.68	0.02
	B3	0.51	0.02	0.48	0.02	0.24	0.01	0.59	0.02
	B4	0.46	0.01	0.44	0.01	0.21	0.01	0.50	0.02
	B5	0.31	0.01	0.29	0.01	0.14	0.00	0.38	0.01
PDF	B1	0.69	0.02	0.68	0.02	0.37	0.01	0.81	0.03
	B2	0.52	0.02	0.50	0.02	0.25	0.01	0.60	0.02
	B3	0.46	0.01	0.43	0.01	0.21	0.01	0.62	0.02
	B4	0.42	0.01	0.38	0.01	0.19	0.01	0.56	0.02
	B5	0.32	0.01	0.30	0.01	0.14	0.00	0.44	0.01
Total		14.486	0.533	13.847	0.483	9.817	0.319	16.032	0.6594

크기 및 길이 축약율, 단위 바이트당 커버리지와 퍼징 진행도를 평가 항목으로 알고리즘별 비교 실험을 진행한 결과, 제안하는 GF3 알고리즘이 길이 축약율, 단위 바이트당 커버리지 그리고 퍼징 진행도에서 기존 연구의 알고리즘보다 우수한 결과를 보임을 확인할 수 있었다. 그리고 크기 축약율에서는 HGS, GRE와 유사한 축약율을 보였으며 BOG보다는 높은 축약율을 보임을 확인할 수 있었다. 이러한 실험 결과를 바탕으로 제안하는 알고리즘이 파일 퍼징에 있어서 기존 연구들과 비교하여 가장 우수한 축약 알고리즘임을 증명할 수 있었다.

향후 연구에서 제안한 테스트 슈트 알고리즘에 대한 결함 탐지율 또는 결함 손실율에 대한 평가를 진행할 예정이다. 테스트 슈트 축약에 있어서 축약율은 알고리즘의 효율성을 평가하기 위해 가장 널리 사용되는 평가 척도이지만 축약율이 높다고 하여 무조건 좋은 알고리즘이라고 평가할 수는 없다. 왜냐하면 축약이 너무 많이 되면 취약점이 포함되어 있을만한 테스트 케이스가 축약으로 테스트 케이스 집합에서 제외될 수 있기 때문이다. 따라서 많은 기존 연구들에서는 축약율에 대한 비교 평가와 함께 결함 탐지율 또는 결함 손실율도 평가 척도로 이용하고 있다. 그러나 결함 탐지율 및 결함 손실율에 대한 평가는 많은 시스템 자원과 시간이 요구되기 때문에 본 논문에서는 그 결과를 모두 포함시키지 못하고 퍼징 시뮬레이션을 통해 예상된 결과만을 보였다. 이는 향후 연구를 통해 보완할 예정이다.

참고문헌

〔1〕 P. Godefroid, M.Y. Levin, D. Molnar, “Automated Whitebox Fuzz Testing,” In Proceedings of Network and Distributed Systems Security, pp. 151-166, Feb. 2008.

〔2〕 P. Uhley, “Advanced Persistent Responses,” CanSecWest, Mar. 2012.

〔3〕 C. Miller, Z.N.J. Peterson, “Analysis of Mutation and Generation-Based Fuzzing,” Independent Security Evaluators, Mar. 2007.

〔4〕 fuzzing.info, http://fuzzing.info/papers.

〔5〕 CVE-2010-3654, “fuzz-my-life-flash-player-zero-day-vulnerability-cve-2010-3654,” MITRE, 2010.

〔6〕 Microsoft Security Bulletin, “Microsoft Security Bulletin MS05-031,” Jun. 2005.

〔7〕 A. Manion, M. Orlando, “Fuzz Testing for Dummies,” Industrial Control Systems Joint Working Group Spring Meeting. May 2011.

〔8〕 WIKIPEDIA, “Fuzz testing,” http://en.wikipedia.org/wiki/Fuzz_testing.

〔9〕 S. Yoo, M. Harman, “Regression testing minimization, selection and prioritization: a survey,” Software Testing, Verification, and Reliability, vol. 22, pp. 67-120, Mar. 2010.

〔10〕 Jaeseo Lee, SuYong Kim, Young-Tae Yun, Kiwook Sohn, Yong-Min Kim, Bong-Nam Noh, “A New Measure for Test-Suite Noble Reduction under File Fuzzing,” International Conference on Smart Convergence Technologies and Applications, pp. 80-82, Aug. 2012.

〔11〕 M. J. Harrold, R. Gupta, M. L. Soffa, “A Methodology for Controlling the Size of a Test Suite,” ACM Transactions on Software Engineering and Methodology, vol. 2, no. 3, pp. 270-285, Jul. 1993.

〔12〕 T. Y. Chen, M. F. Lau, “A new heuristic for test suite reduction,” Information and Software Technology, vol. 40, pp. 347-354, 1998.

〔13〕 S. Parsa, A. Khalilian, “On the Optimization Approach towards Test Suite Minimization,” International Journal of Software Engineering and Its Applications, vol. 4, no. 1, pp. 15-28, Jan. 2010.

〔14〕 G. Rothermel, M. J. Harrold, J. Ronne, C. Hong, “Empirical Studies of Test-Suite Reduction,” Software Testing, Verification, and Reliability, vol. 4, no. 2, pp. 219-249, Feb. 2002.

〔15〕 H. Zhong, L. Zhang, H. Mei, “An experimental study of four typical test suite reduction techniques,” Information and Software Technology, vol. 50, pp. 534-546, 2008.

〔16〕 N. Mansour, K. El-Falkin, "Simulated annealing and genetic algorithms for optimal regression testing," Journal of Software Maintenance, vol. 33, no. 4, pp. 225-237, 1999.

〔17〕 J. Black, E. Melachrinoudis and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," Proceedings of 26th International Conference on Software Engineering, pp. 106-115, May 2004.

〔18〕 Z. Chen, X. Zhang, B. Xu, "A Degraded ILP Approach for Test Suite Reduction," Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering, pp. 494-499, Jul. 2008.

〔19〕 A. Takaken, J. D. Demott, C. Miller, FUZZING - for Software Security Testing and Quality Assurance, Artech House, 978-1-59693-214-2, 2008.

〔20〕 P. Amini, "Fuzzing Frameworks," Blackhat USA, Aug. 2007.

〔21〕 Pintool, "A Dynamic Binary Instrumentation Tool," http://pintool.org.

〈著者紹介〉

사 진

이 재 서 (Jaeseo Lee) 정회원
2004년 2월: 전남대학교 컴퓨터정보학부 학사
2006년 2월: 전남대학교 정보보호협동과정 석사
2006년 3월~현재: 전남대학교 정보보안협동과정 박사과정
2009년 4월~현재: 한국전자통신연구원 부설연구소 연구원
〈관심분야〉 웹 해킹 및 보안, 소프트웨어 취약점

사 진

김 종 명 (Jong-Myoung Kim) 정회원
2007년 2월: 성균관대학교 정보통신공학과 학사
2009년 2월: 성균관대학교 전자전기컴퓨터공학과 석사
2009년 1월~2011년 6월: 한국인터넷진흥원 연구원
2011년 7월~현재: 한국전자통신연구원 부설연구소 연구원
〈관심분야〉 악성코드 분석, 소프트웨어 취약점

사 진

김 수 용 (SuYong Kim) 정회원
2000년 2월: 한국과학기술원 전산학과 학사
2002년 2월: 한국과학기술원 전산학과 석사
2011년 8월: 한국과학기술원 전산학과 박사
2002년~현재: 한국전자통신연구원 부설연구소 선임연구원
〈관심분야〉 소프트웨어 취약점

사 진

윤 영 태 (Young-Tae Yun) 정회원
1995년 2월: 충남대학교 컴퓨터과학과 학사
1995년~1997년: 현대전자 정보시스템 사업본부
1999년 2월: 충남대학교 컴퓨터과학과 석사
2006년 8월: 충남대학교 컴퓨터과학과 박사
1999년~현재: 한국전자통신연구원 부설연구소 책임연구원
〈관심분야〉 소프트웨어 취약점, 네트워크 보안

김 용 민 (Yong-Min Kim) 종신회원
2002년 8월: 전남대학교 전산통계학과 박사
2004년 3월~2006년 2월: 여수대학교 정보기술학부
2009년 3월~2012년 1월: 전남대학교 정보과학연구소장
2006년 3월~현재: 전남대학교 문화콘텐츠학부 부교수
〈관심분야〉 시스템 및 네트워크 보안, 전자상거래 보안 등

노 봉 남 (Bong-Nam Noh) 종신회원
1978년 2월: 전남대학교 수학교육과 학사
1982년 2월: 한국과학기술원 전산학과 석사
1994년 2월: 전북대학교 정보보호 전공 박사
1983년~현재: 전남대학교 전자컴퓨터공학부 교수
2000년~현재: 전남대학교 시스템보안연구센터 센터장
〈관심분야〉 디지털 포렌식, 시스템 보안, 데이터베이스 보안