역방향 탐색을 사용하는 하이브리드 분석 기법에 관한 연구

장성수*, 최영현*, 임헌정*, 정태명**
*성균관대학교 전자전기컴퓨터공학부
**성균관대학교 정보통신공학부
e-mail: *{ssjang, yhchoi, hjlim99}@imtl.skku.ac.kr, **tmchung@ece.skku.ac.kr

A Study of Advanced Hybrid Execution Using Reverse Traversal

Seongsoo Jang*, Young-Hyun Choi*, Hun-Jung Lim* and Tai-Myoung Chung**
*Dept. of Electrical and Computer Engineering, Sungkyunkwan Univ.
**School of Information Communication Engineering, Sungkyunkwan Univ.

요 약

소프트웨어 분석 기법이 발전하며 다양한 종류의 악성 코드를 점검할 수 있게 되면서 이를 회피하기 위한 기술들이 등장하였다. 실행 시 스스로 코드를 변경하는 등의 진화된 악성 코드들로부터 시스템을 보호하기 위해 프로그램에 존재하는 실행되지 않는 경로에 대해서도 검사를 할 수 있는 기법을 제시한다. 제안하는 기법은 프로그램을 읽어 CFG를 생성하고, 각 종료 지점에서부터 이를 역방향으로 순회하여 모든 실행 경로를 얻는다. 여기서 발생하는 오버헤드는 멀티코어 프로세서를 활용하는 다중 작업으로 완화시킬 수 있다.

1. 서론

컴퓨터 및 인터넷의 발달과 함께 해킹, 악성 코드 등으로 인한 보안사고 또한 증가하고 있다. 특히 최근 몇 년간 우리나라에서도 수차례 이슈가 되었던 분산서비스거부공격(Distributed Denial of Service, DDoS)은 기업 및 개인 사용자들에게 막대한 피해를 안겨주었다. 그 영향으로 소프트웨어의 보안 취약점을 점검하고, 악성 코드를 검출하기 위한 소프트웨어 분석 및 테스트 기법에 관한 연구가 재조명을 받게 되었다.

한편, 소프트웨어 분석 기법이 발전하며 다양한 악성 코드를 점검할 수 있게 되자 이를 우회하기 위한 기술들이 등장하며 봇넷들도 진화를 하기 시작했다.[6] 코드 난독화를 통해 분석을 어렵게 하거나 런타임 중에 스스로 코드를 변경하는 기법 등은 악성 코드를 포함하는 부분을 숨김으로써 기존의 분석 도구에 의해 적발되지 않게 해준다. 따라서 본 논문에서는 실행되지 않는 경로에 대한 분석 또한 지원할 수 있는 기법을 제안하여 잠재된 위험성을 검사함으로써 소프트웨어 분석기술을 우회하는 악성코드에 대한 대비책을 마련하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서 관련 연구로써 concrete execution와 symbolic execution, 실행되지 않는 경로 등에 대해 간략히 살펴보고, 역방향 탐색을 사용하는 향상된 기호 실행 기법을 3장에서 제안한다. 마지막으로 4장에서 결론과 함께 향후 연구 방향에 대해 기술한다.

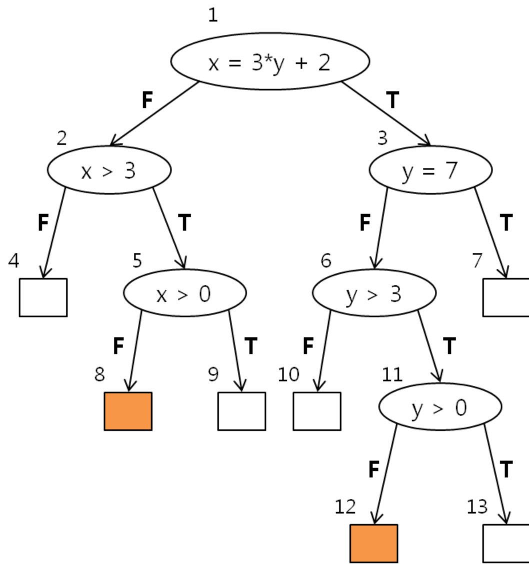
2. 관련연구

2.1 Concrete execution vs. symbolic execution

소프트웨어의 보안취약점 검사를 위한 테스트 방법 중 가장 간단한 것으로는 테스트 케이스들을 생성하여 이를 이용해 프로그램을 직접 실행해 보는 concrete execution 기법이 있다. 이 방법은 소스 코드가 제공되지 않더라도 비교적 간단하게 테스트를 수행할 수 있고, 실제 입력 값을 사용하여 프로그램을 실행하기 때문에 긍정 오류와 부정 오류의 발생이 적다는 장점이 있다. 그러나 모든 실행 경로에 대한 테스트 케이스의 생성이 어렵다는 단점이 있다. Symbolic execution[1]은 실제 값을 입력하는 대신, 입력 변수에 기호 값을 할당하여 그 값을 프로그램 내부로 전달하며 분석하는 방법이다. 기호 값을 논리적, 수학적 수식으로 표현하는 요약 도메인을 검증함으로써 프로그램을 실제로 실행하지 않고 가상으로 실행하는 효과를 얻을 수 있다. Symbolic execution은 프로그램의 모든 실행 경로에 대해 분석을 할 수 있고, 다양한 툴을 통해 자동으로 검사를 할 수 있는 장점이 있다. 그러나 내부 함수 호출, 루프, 스위치 문 등으로 인해 실행 경로가 기하급수적으로 증가하는 path explosion 문제가 있어 구현이 힘들고 오버헤드가 크다. 소프트웨어 분석에 관한 최근의 연구[2][3][5]는 concrete execution과 symbolic execution을 혼합하여 두 기법의 장점을 살리고 단점을 보완하는 하이브리드 분석 기법의 연구에 중점을 두고 있다.

2.2 실행되지 않는 경로

Concrete execution과 symbolic execution을 혼합한 하이브리드 분석 기법은 프로그램 전체를 커버하기 위해 symbolic execution으로 control flow graph(CFG)를 작성한 후, 실행 오버헤드를 줄이기 위해 테스트 케이스를 생성하여 concrete execution을 실행한다. 그러나 분기점에 따라 서로 상충하는 조건을 갖는 경우, control flow에 따라 실행되지 않는 경로가 발생할 가능성이 존재한다. (그림 1)의 {1, 2, 5, 8} 경로와 {1, 3, 6, 11, 12} 경로가 실행되지 않는 경로의 예이다. {1, 2, 5, 8} 경로를 살펴보면, 2번 노드의 분기에서 'x가 3보다 크다'는 조건을 만족한다는 가정 하에 5번 노드로 이동을 한다. 그렇기 때문에 5번 노드의 분기에서 'x가 0보다 크다'는 조건은 항상 참이 되어 8번 노드에는 닿을 수 없다. {1, 3, 6, 11, 12} 경로 역시 6번 노드와 11번 노드의 분기점에서의 조건이 상충하기 때문에 12번 노드로 더 이상 진행할 수 없다. 기존의 분석 도구는 이러한 실행되지 않는 경로에 대해서는 테스트를 수행하지 않는 맹점을 갖고 있다. 따라서 최근 대두되는 런타임 시 스스로 코드를 수정하는 등의 새로운 악성코드 기법에 대해 취약점을 가지므로 실행되지 않는 경로에 대한 테스트 또한 필요하다.



(그림 1) 실행되지 않는 경로

3. 역방향 탐색을 사용하는 하이브리드 분석

본 논문에서 제안하는 역방향 탐색을 사용하는 하이브리드 분석 기법은 다음과 같은 순서로 동작한다. 우선, 프로그램을 스캔하며 분기에 따라 블록 단위로 나누고, 프로그램의 종료 지점들을 체크한다. 그 후 control flow에 따라 각 블록을 연결하여 CFG를 작성한다. 프로그램 각 종료 지점들로부터 CFG를 역으로 순회하며 실행 경로를 확보한다. 각 실행 경로에서 CFG의 정방향으로 테스트 케이스를 생성하며 분석을 수행한다.

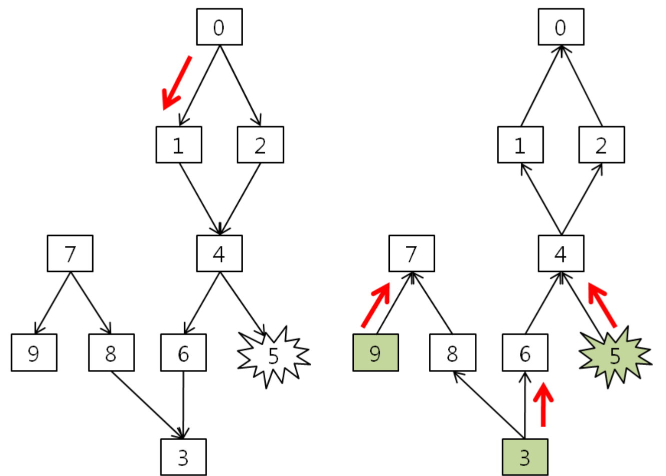
```
     main() {
L0:      if (x > y)
L1:          z = f(y);
         else
L2:          z = f(x);
L3:      return;
     }

     int f(a) {
L4:      if (a > 0)
L5:          abort;
         else
L6:          return -a;
     }

     void g(a) {
L7:      if (a > 0)
L8:          goto L3;
         else
L9:          print 2 * a;
     }
```

(그림 2) 예제 코드

(그림 2)에 주어진 예제 코드는 main()함수와 int형 값을 반환하는 f()함수, 리턴 값을 반환하지 않는 g()함수로 구성되어 있다. (그림 2)의 소스 코드를 스캔하면 각 라인을 하나의 블록으로 묶을 수 있고, (그림 3)과 같은 CFG를 작성할 수 있다. 여기서 함수 g()는 호출되지 않음을 알 수 있다. 기존의 symbolic execution에서의 실행 경로 탐색은 깊이우선탐색(depth-first search, DFS) 기반으로 이루어진다.[4] 이에 따라 (그림 3)의 (a)와 같이 주어진 각 노드들을 순회하면 {0, 1, 4, 6, 3}, {0, 1, 4, 5}, {0, 2, 4, 6, 3}, {0, 2, 4, 5}의 4가지 실행 경로를 얻을 수 있으나 control flow가 닿지 않는 {7, 9}, {7, 8, 3}의 경로는 얻을 수가 없다. 그러나 (그림 3)의 (b)에서 보듯, 소스 코드를 스캔하며 프로그램의 종료 지점 L3, L5, L9에 대한 정보를 기록해 놓고, 각 종료 지점으로부터 CFG를 역순으로 탐색하면 {3, 6, 4, 2, 0}, {3, 6, 4, 1, 0}, {3, 8, 7}, {5, 4, 2, 0}, {5, 4, 1, 0}, {9, 7}의 총 6가지 실행 경로를 얻을 수 있다. 이제 각각의 실행 경로에 대해 테스트 케이스를 생성하고 정방향으로 concrete execution을 실시하면, 실행되지 않는 경로에 대해서도 분석을 할 수 있다.



(그림 3) CFG의 탐색

그러나 역방향 탐색을 사용하는 하이브리드 분석 기법에서는 추가로 발생하는 오버헤드에 대해서도 고려를 해야 한다. 프로그램의 각 종료 지점에서부터 CFG를 역방향으로 탐색하기 때문에 기존의 실행되지 않는 경로까지 분석을 하는 만큼의 오버헤드가 추가로 들게 되고, 이는 전체 분석 속도를 저하시킬 가능성이 있다. 그러나 이는 프로그램을 스캔하여 종료 지점에 대한 정보를 얻은 후, fork() 함수를 사용하여 각 종료 지점으로부터의 실행 경로 탐색을 다중 작업으로 처리할 수 있게 한다. 오늘날, 멀티코어 프로세서의 사용이 점점 증가하는 추세에 있기 때문에 오버헤드가 많이 드는 각각의 실행 경로 탐색 작업을 병렬 처리하여 실시한다면 오버헤드를 어느 정도 완화시킬 수 있을 것이라 예상된다.

4. 결론 및 향후 연구

본 논문에서 본 논문에서 제안하는 기법은 분석하고자 하는 프로그램을 읽어 들여 CFG를 작성한 후, 프로그램의 각 종료 지점에서부터 CFG를 역으로 순회함으로써 실행되지 않는 경로를 포함하는 모든 실행 경로를 확보할 수 있다. 이 후, 각 실행 경로에 대해 concrete execution을 수행함으로써 잠재된 위험성도 발견할 수 있어 실행 중 스스로 코드를 수정하는 새로운 악성코드 기법에 대항할 수 있다. 다만 추가로 발생하는 오버헤드가 염려되는데, 이는 멀티코어 프로세서의 사용을 통해 완화시킬 수 있을 것으로 기대된다.

향후 연구 방향으로는 CFG의 각 노드를 본 논문에서 제안하는 기법을 실제로 구현해 봄으로써 실제 구현 시 발생할 수 있는 문제점에 대해 분석하도록 하겠다. 또한 CFG의 역방향 순회가 가능하도록 각 노드의 간선을 반대로 연결하는 변환 알고리즘의 오버헤드를 최소화 시키는 방법에 대해서도 계속하여 연구할 예정이다.

ACKNOWLEDGMENT

본 논문은 중소기업청에서 지원하는 2010년도 산학연공동기술개발사업(No. 00044301)의 연구수행으로 인한 결과물임을 밝힙니다.

참고문헌

[1] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, "Compilers - Principles, Techniques, and Tools," 2nd Edition, Pearson Education, Inc., 2007

[2] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in Automated Software Engineering, 2008. IEEE/ACM Int'l Conf., pp. 443-446, Sep. 2008

[3] K. Sen, D. Marinov and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in ESEC/FSE '05, Sep. 2005

[4] S. Bardin and P. Herrmann, "Pruning the Search Space in Path-based Test Generation," in Proceedings of ICST 2009, pp. 240-249, Apr. 2009

[5] V. Chipounov, *et al.* "Selective symbolic execution," in Workshop on Hot Topics in Dependable Systems, Jun. 2009

[6] 원유재, “봇넷 대응기술 동향,” 한국정보보호진흥원, Aug. 2008