정 보 보 호 학 회 지
제28권 제2호, 2018. 4

소프트웨어 취약점 자동 수정 기술 소개

오 학 주*

요 약

최근들어 소프트웨어의 오류 및 취약점을 자동으로 수정하는 기술이 주목받고 있다. 소프트웨어의 오류를 수정하는 작업은 소프트웨어 개발 단계에서 매우 큰 비용이 소요된다. 본 글에서는 이를 자동화하는 기술인 소프트웨어 오류 자동 수정 기술에 대해서 소개하고 연구 동향을 살펴본다.

Ⅰ. 서 론

소프트웨어의 오류를 수정하는 작업은 실수하기 쉬우며 큰 비용과 노력을 필요로 하기 때문에 이를 자동화하는 기술이 있다면 매우 유용할 것이다. 소프트웨어의 오류나 취약점을 수정하는 작업은 현재 전적으로 사람에 의존하고 있다. 하지만 오류를 올바르게 고치고 유지보수 하는 것은 매우 까다롭고 오랜 시간이 걸리는 작업이다.〔1,2〕 최근들어 활발하게 연구 되고 있는 프로그램 자동 수정(Automatic Program Repair) 기술은 이러한 디버깅 과정을 자동화함으로써 소프트웨어 개발 비용을 획기적으로 낮추고 소프트웨어의 안전성을 높일 수 있는 차세대 기술로 주목받고 있다.

본 글에서는 C/C++ 언어로 작성된 프로그램에서 자주 발생하는 메모리 관리 오류(memory-leak, double-free, use-after-free)를 자동으로 수정하는 기술을 소개한다. 또한 지금까지 학계에 발표된 소프트웨어 오류 자동 수정 기법들 중 대표적인 기법 세가지를 선정하여 작동원리를 소개한다.

Ⅱ. 메모리 관리 오류 자동 수정

이 장에서는 메모리 관리 오류를 자동으로 수정하는 기법을 소개한다.

2.1. 메모리 관리 오류

C/C++와 같이 메모리 관리를 자동으로 해주지 않는 프로그래밍 언어에서 가장 자주 발생하는 오류가 메모리 관리 오류이다. 메모리 관리 오류는 크게 세가지로 분류할 수 있는데, 할당한 메모리를 해제하지 않거나 너무 늦게 해제하는 오류(memory leak), 동일한 메모리를 두 번이상 해제하는 경우(double-free), 할당한 메모리를 너무 빨리 해제하여 해제된 메로리를 사용하게 되는 오류(use-after-free)이다. 이 세가지 오류 모두 C/C++로 작성된 프로그램에서 가장 자주 발생하는 오류들이며, 올바르게 패치가 이루어지지 않은 경우 심각한 문제를 일으킬 수 있다.

2.2. 오류 수정의 어려움

그림 1의 프로그램을 보자. 왼쪽 프로그램에서는 할당된 메모리를 두 번 해제하는 오류(double-free)가 발생한다. 첫 번째 줄에서 메모리가(o1) 할당되고 두 번째 줄의 조건이 false라고 해 보자. 이 때 7번째 줄이 실행되는데 포인터 q와 p가 모두 같은 메모리 주소를 가리키게 된다. 따라서 9, 10번째 줄에서 같은 메모리를 두 번 해제하게 되어 double-free오류가 발생한다. double-free는 소프트웨어 보안 취약점의 주요 원인 중 하나로, 올바르게 고쳐지지 않으면 심각한 보안 문제로 이어질 수 있다.

이 논문은 2018년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2017-0-00184, 자기학습형 사이버 면역 기술 개발)
* 고려대학교 컴퓨터학과 (hakjoo_oh@korea.ac.kr)

```
1  p = malloc(1); // o1
2  if (...) {
3    q = malloc(1); // o2
4
5  }
6  else
7    q = p;
8  ... = *q; // use q
9  free(q);
10 free(p);
```

(a) Buggy code

```
1  p = malloc(1);
2  if (...) {
3    q = malloc(1);
4    free(p); // +
5  }
6  else
7    q = p;
8  ... = *q;
9  free(q);
10 // -
```

(b) Fixed code

〔그림 1〕 예제 프로그램

이 문제를 올바르게 해결하는 것은 생각보다 까다롭다. 예를 들어, double-free오류를 제거하기 위해서 두 개의 free중에서 어느 하나를 제거한다고 해 보자. 이 경우 double-free는 사라지지만 원래 프로그램에는 없었던 메모리 누수(memory leak)문제가 발생한다. 이와 같이 소프트웨어의 오류를 해결하려다가 실수하여 오히려 다른 오류를 유발시키는 일이 실제 개발환경에서 자주 일어난다. 한 예로, 리눅스 커널에서 소프트웨어 오류가 개발자에 의해서 패치되는 과정을 살펴보면 잘못된 패치로 인해서 원래 문제가 여러 단계를 거치면서 총 수개월에서 수년의 시간이 소요되는 경우가 종종 발견된다.

위 그림에서 오른쪽은 문제를 올바르게 해결한 경우이다. 이 경우 올바르게 해결하는 방법은 10번째 줄에 있던 free(p)를 4번째 줄로 옮기는 것이다. 이렇게 패치된 코드는 원래 문제인 double-free를 제거하면서 메모리 누수와 같은 다른 문제를 새롭게 유발시키지 않는다. 소프트웨어 자동 수정 기술이란 이와같이 오류가 있는 프로그램(왼쪽)을 입력으로 받아서, 오류가 수정된 프로그램(오른쪽)으로 변환시키는 기술을 의미한다.

2.3. 정적 분석을 이용한 오류 자동 수정

2.3.1. 정적 분석

정적 분석 기술을 이용하여 메모리 관리 오류를 자동으로 수정하는 방법을 소개한다. 크게 두 단계를 거치게 된다: 1) 프로그램을 분석하여 가능한 모든 패치 후보들을 생성하는 단계와 2) 이러한 패치 후보 중에서 최종 패치를 선별하는 단계이다.

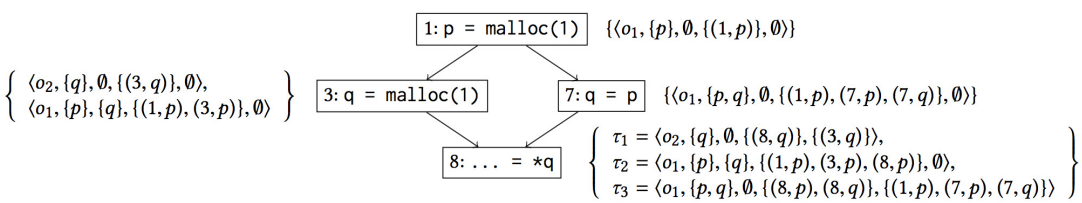
먼저 주어진 프로그램을 분석하여 할당된 메모리 객체들이 가지는 상태들을 각 프로그램 지점별로 계산한다. 메모리 객체 상태는 다섯가지 요소로 정의한다:

<alloc-site, must, mustNot, patch, patchNot>

여기서 alloc-site는 객체가 할당된 프로그램 지점이다. 예를 들어, 예제 프로그램에서 첫 번째 라인에서 할당된 메모리들을 모두 o1으로 나타내며, 세 번째 라인에서 할당된 메모리들은 모두 o2로 나타내기로 한다.

두 번째와 세 번째 요소(must, mustNot)은 메모리 객체를 가리키거나 가리키지 않는 포인터들에 대한 정보이다. must는 해당 객체를 프로그램 실행중에 반드시 가리키는 포인터의 집합이다. 예를 들어, 첫 번째 라인을 분석하면 메모리 o1을 항상 가리키는 포인터인 p가 must 집합에 들어가게 된다. mustNot은 반대로 실행중에 항상 가리키지 않는게 보장되는 포인터들이다. 예를 들어, 세 번째 라인을 분석하면 포인터 q는 더 이상 o1을 가리킬 수 없음을 알게 된다.

마지막 두 요소(patch, patchNot)은 해당 메모리를 안전하게 해제하거나 안전하지 않은 패치들의 집합이다. 예를 들어, 첫 번째 라인에서 포인터 p를 통해서 메모리 o1을 안전하게 해제할 수 있다. 하지만 8번째 라인을 지난 후에는 메모리 o1이 사용될 수 있으므로 첫 번째 이러한 패치가 안전하지 않음을 알 수 있다 (use-after-free).



〔그림 2〕 패치 후보 생성을 위한 프로그램 분석 결과

이런식으로 프로그램을 순차적으로 분석하여 각 요소를 계산하면 그림 2와 같은 결과를 얻게 된다. 각 메모리 객체를 서로 다른 상태로 구분하여 모두 모으는 정적 분석이며 이 결과에서 프로그램의 마지막 지점에 모인 상태 3개가 최종적으로 패치 생성에 이용된다.

2.3.2. 패치 생성

지금까지 정적 분석을 이용하여 각 프로그램 지점별로 가능한 패치와 불가능한 패치들을 모두 모았다. 그 다음은 정보를 종합하여 최종적으로 올바른 패치를 생성할 수 있다. 정적 분석 결과로부터 다음의 세가지 집합을 정의한다.

Safe = { (1,p), (3,p), (8,p), (8,q) }
UnSafe = { (1,p), (3,q), (7,p), (7,q) }
Cand = { (3,p), (8,p), (8,q) }

집합 Safe는 모든 메모리 객체 상태들에 있는 사용할 수 있는 패치들을 모은 것이다. UnSafe에는 사용할 수 없는 패치들이 모두 담겨 있다. 이 두 정보는 그림 2의 마지막 분석 결과로부터 쉽게 구할 수 있다. 마지막으로 Cand는 Safe와 UnSafe의 차집합을 나타낸 것으로, 사용할 수 없다고 판단된 패치들을 모두 제거함으로써 어느 메모리 객체에 대해서든 안전한 패치, 즉 최종적으로 사용 가능한 패치들의 집합을 나타낸다.

그 다음 Cand에 모인 패치들을 가지고 아래와 같은 행렬을 구성한다.

	τ_1	τ_2	τ_3
(3, p)	0	**1**	0
(8, *p*)	0	1	1
(8, q)	**1**	0	**1**

행렬의 가로줄은 Cand 집합에 모인 패치들이며 세로줄은 그림 2에서 프로그램 마지막에 모인 메모리 객체 상태들을 의미한다. 행렬의 (i,j) 원소는 주어진 패치 i가 해당 메모리 객체 j를 해제할 수 있는지를 의미한다. 즉, 1이면 해제할 수 있음을 의미하고 0이면 없음을 나타낸다. 이 정보는 단지 정적 분석의 결과를 테이블로 표현한 것일뿐 행렬을 나타내기 위해 추가적인 계산이 필요한 것은 아니다.

이제 최종적으로 패치를 찾는 문제는 위 행렬로 표현된 부분집합들에서 전체집합을 모두 포함하며 서로 겹치지 않는 부분집합들의 집합을 찾는 문제(exact cover problem)을 푸는 것이다. 위의 예제에서는 굵게 표시된 두 가로줄 (3,p)와 (8,q)를 선택하면 된다. (3,p)는 두 번째 메모리 객체를 담고 있는 부분집합을 나타내고, (8,q)는 첫 번째와 세 번째 객체로 구성된 부분집합을 나타낸다. 이 두 집합들을 모두 합하면 전체집합이 되며 각 부분집합들의 교집합은 공집합이 된다.

여기서 각 부분집합의 합이 전체집합이 된다는 것은 프로그램에 더 이상 memory leak문제가 없이 모든 할당된 메모리들이 올바르게 해제된다는 것을 의미하고, 서로 교집합이 없다는 것은 하나의 메모리가 두 번 이상 해제 되는 일(double-free)이 없음을 의미한다.

Ⅲ. 테스트 케이스 기반 소프트웨어 오류 수정 기술

지금까지 정적 분석을 이용하여 소프트웨어 오류를 자동으로 수정하는 기법을 설명하였다. 하지만 정적 분석 기반의 오류 수정 기술은 최근들어 등장하고 있고, 전통적으로 소프트웨어 오류 수정 기술은 정적 분석보다는 동적 분석, 즉 프로그램을 주어진 테스트 케이스에 대해서 실행하여 프로그램이 올바르게 수정되었는지를 확인하는 기술 위주로 발달해왔다.

이 장에서는 이러한 테스트 케이스 기반 소프트웨어 오류 수정 기법 중에서 대표적인 기법이라 할 수 있는 몇가지 기술을 소개하고 정적 분석 기반 기술과의 장단점을 살펴본다.

3.1. GenProg〔3〕

GenProg은 실제 소프트웨어를 자동으로 수정하는데 처음 성공한 기법으로 Genetic Programming에 기반하고 있다. GenProg은 오류를 포함하고 있는 대상 프로그램과 프로그램의 명세를 나타내는 테스트 케이스를 입력으로 받는다. 이 때 오류가 수정된 프로그램은 테스트 케이스를 모두 통과해야 한다.

이러한 입력이 주어졌을 때 GenProg은 다음의 과정을 거쳐서 프로그램의 오류를 수정한다.

(1) 오류를 수정할 소스코드상의 위치를 찾는다. 이 과정을 위해 GenProg은 간단한 휴리스틱을 이용한다. 주어진 테스트 케이스중에서 오류를 유발하는 입력으로

프로그램을 실행했을 때 가장 많이 실행되는 프로그램 지점을 오류를 수정할 대상으로 추측한다.

(2) 수정할 위치가 정해지면 해당 영역의 코드를 삭제하거나 다른 곳에 존재하는 코드를 가져와서 수정/교체하는 연산을 오류가 수정될때까지 반복한다. 프로그램을 수정할 수 있는 방법이 매우 많기 때문에 GenProg은 제한적이지만 효과적인 몇몇 연산자에만 집중한다. 여기서 GenProg이 가정하고 있는 것은 대부분의 오류들이 같은 프로그램의 다른 영역의 코드를 가져와서 쉽게 수정할 수 있다는 것이다 (Redundancy Assumption). 즉, 어느 한곳에서 프로그래머가 실수로 오류를 만들었어도 다른 비슷한 상황에서는 올바른 코들 작성했을 것이라는 가정이다. 예를 들어, C 프로그램에서 개발자가 어느 지점에서는 Null 값 체크를 빠뜨렸어도 다른 곳에서는 실수하지 않을 수 있다.

GenProg은 위와 같은 단순한 기법으로도 실제 C 프로그램의 오류들을 자동으로 수정할 수 있음을 보였다. 총 6만여 라인의 코드에서 전체 오류의 50%이상을 자동으로 수정할 수 있었고, 오류 수정에 필요한 시간은 수분 가량으로 실제 현장에서 충분히 활용가능한 수준임을 보였다.

3.2. SemFix〔4〕

GenProg은 기술적으로 우수한 기법이라기 보다는 실제 오류를 자동으로 수정가능하다는 것을 처음으로 보였다는 점에서 큰 의미를 가진다. 실제로 GenProg이 발표된 이후에 많은 연구자들이 소프트웨어 자동 수정 기술에 관심을 가지게 되었고, 현재 매년 많은 오류 수정기법들이 발표되고 있다.

SemFix 기법의 동기는 GenProg의 단점을 해결하는 것이다. GenProg의 기본적인 가정인 “Redundnacy Assumption”은 패치를 위한 코드가 프로그램에 항상 존재한다는 것이다. 이러한 가정 위에서 GenProg은 프로그램에 존재하는 코드조각들의 조합을 통해서 패치생성을 시도한다. 이러한 가정은 많은 경우 말이되지만, 근본적으로 프로그램에 패치가 되는 코드조각이 존재하지 않는다면 오류를 수정할 수 없다는 제한이 있다. 즉, GenProg과 같이 프로그램의 특정 겉모습(Syntax)에 의존하는 기법은 패치를 자동으로 합성할 수 없는 경우가 반드시 존재한다는 근본적인 단점을 가진다.

SemFix는 이 문제를 프로그램 합성 기법을 적용하여 해결한다. 하지만 단순하게 모든 가능한 프로그램들을 나열하는 합성 기법(Enumerative Synthesis)은 효과적으로 패치를 생성하기 어렵 다. 일반적으로 프로그램에서 사용하는 식과 명령문들은 임의의 것이 될 수 있고 그 탐색 공간은 무한하기 때문이다. 따라서 SemFix는 나열 기반 프로그램 합성이 아닌 제약식 기반 프로그램 합성(Constraint-based Synthesis)기법을 활용한다. 합성할 프로그램이 만족해야 하는 제약은 심볼릭 실행으로부터 생성되고 해당 제약 조건을 만족시키는 프로그램은 SMT Solver를 활용한 프로그램 합성 기법으로 얻어진다. 구체적으로 다음의 세가지 기법을 사용한다:

(1) 오류 지점 예측 (Fault Localization): 소스코드의 어느 부분을 수정할지를 결정한다. 이를 위해 SemFix에서는 Tarantula라고 불리는 통계적 오류 지점 예측 기법을 이용한다. 이 기법은 오류를 일으키는 테스트 케이스들의 실행을 통해서 각 프로그램 지점별로 오류가 존재할 확률을 계산한 후 그 크기순서대로 프로그램 지점을 정렬해준다. SemFix는 이 순서대로 프로그램 지점을 차례대로 수정을 시도해본다.

(2) 제약식 생성 (Constraints Generation): 수정할 프로그램 지점이 정해지면, 해당 지점의 코드가 만족해야 하는 제약조건들을 생성한다. 심볼릭 실행을 통해서 주어진 각 테스트 케이스마다 하나씩 해당 프로그램 지점에서 해당 테스트 케이스를 통과하기 위해 필요한 제약 조건을 생성해낸다.

(3) 프로그램 합성 (Program Synthesis): 제약 조건이 만들어지면, 이를 만족시키는 프로그램 조각을 합성하는 단계이다. 이 때, SMT Solver를 통해 반복문이 없는 프로그램을 합성해 내는 프로그램 합성 기법을 이용한다.

3.3. Prophet〔5〕

GenProg과 SemFix기법은 전적으로 프로그램 상태 공간을 탐색하여 패치를 생성하는 기법인데 비해, Prophet은 소스코드 저장소에 있는 기존에 개발자가 수정했던 패치 데이터로부터 확률 모델을 학습하여 성공 확률이 높은 패치를 더욱 효율적으로 만들어낸다. 주요 특징은 다음과 같다.

- 확률 모델(Probabilistic model)의 사용: Prophet은 오픈소스 저장소에서 개발자들의 패 치 정보들로부터 올바른 패치에 대한 확률모델을 배운다. 자동 패치 생성기법은 보통 많은 수의 패치후보들을 생성한 후 테스트 케이스를 통과하는 패치들을 찾아내는 방식으로 동작하는데(generate-and-validate), Prophet은 확률모델을 이용해서 이들 중에서 성공확률이 높은 패치를 효과적으로 찾아낸다.
- 주변 코드와의 상호작용 고려: 패치의 성공여부는 패치에 의해 추가된 코드가 원래코드와 올바르게 상호작용하는가에 달려있다. Prophet이 사용하는 확률모델은 이러한 패치와 주변 코드의 상호작용을 표현하는 특성(feature)을 충분히 활용하고 있다.
- 범용 특성 사용: Prophet은 올바른 코드들은 일반적으로 공통된 성질을 가짐을 가정하고 있다. Prophet은 이러한 공통적인 프로그램들의 성질을 표현하는 범용 특성을 정의하고 사용한다. 범용 특성의 사용은 주어진 데이터에서 확률모델을 배운후 새로운 프로그램에 효과적으로 적용하기 위해 필수적이다.

구체적인 작동방식은 다음과 같다.

- 오류 지점 예측(Defect Localization): Prophet은 먼저 테스트 케이스들을 실행해봄으로써 소스 코드 상의 오류 지점을 예측한다. 테스트 케이스를 실행하면 오류가 일어날때 가장 빈번하게 실행되는 프로그램 영역을 알게 되고, 오류가 일어나지 않을때는 거의 실행되지 않는 곳을 알아낸다. Prophet은 이 지점에 오류의 원인이 담겨있다고 가정한다.
- 탐색 공간 생성(Search Space Generation): 오류 지점을 찾으면 그 지점을 다양한 형태로 수정함으로써 탐색 공간을 생성한다. 여기서 탐색 공간이란 패치 후보들의 집합으로써 이 집합을 생성하기 위해서 미리 정해놓은 템플릿을 사용한다. Prophet은 모두 기본적으로 이러한 탐색공간을 생성해 놓고 하나씩 대상 패치들을 테스트케이스를 통해 검증해봄으로써 올바른 패치를 찾아나간다.
- 패치 우선순위화(Rank Candidate Patches): Prophet은 탐색 공간을 탐색하는 방법이 가장 큰 특징이다. 기존 기법들은 탐색 공간을 하나씩 차례대로 고려하는 반면, Prophet은 학습한 확률모델을 이용하여 탐색공간내의 패치 후보들을 정렬하고 그 순서대로 시도한다. 확률모델은 사람이 작성한 패치와의 유사성에 대한 확률을 계산하므로 정렬상 위에 속한 패치가 실제 패치일 가능성이 높게 되며 올바른 패치를 더 빨리 찾아낼 수 있다.
- 패치 검증 단계(Validate Candiate Patches): 마지막으로 선택된 패치가 모든 테스트 케이스를 통과하는지를 체크한다. 한 예롤 PHP프로그램의 오류를 수정할때 모두 6957개의 테스트 케이스를 모두 통과하는지 확인하였다.

3.4. 정적 분석 기반 vs. 동적 분석 기반

지금까지 살펴본바와 같이 소프트웨어 자동 수정 기술은 크게 정적 분석 기반과 동적 분석 기반으로 구분할 수 있다. 이는 수정된 프로그램이 올바르게 수정되었는지를 판단하는 방법을 기준으로 구분한 것으로 다른 기준으로 자동 수정 기법을 나눌수도 있다.

이와 같은 기준으로 구분할 때 각 기법이 가지는 장단점은 소프트웨어 오류 탐지를 위한 동적/정적 분석 기술이 가지는 장단점과 유사하다. 먼저 정적 분석을 기반으로 하면 분석의 안전성(Soundness)가 보장되는 경우, 패치를 생성하였을 때 그 패치의 올바름(Safety)를 완전히 보장할 수 있다. 예를 들어, 2장에서 설명한 오류 자동 수정 기법은 패치가 만들어지는 경우 그 패치는 항상 올바르게 문제를 해결한다는 사실이 엄밀히 보장된다. 반면에 각 오류의 종류마다 검증할 수 있는 정적 분석이 달라져야 하므로 일반적으로 오류를 수정하는 기술을 만들기가 어렵다.

테스트 케이스를 가지고 프로그램 오류 수정 여부를 판단하는 경우의 장점은 오류의 종류에 대해 보다 일반적으로 패치할 수 있다는 것이다. 하지만 동적 분석이 가지는 단점을 고스란히 가지게 되는데, 테스트 케이스를 모두 통과하더라도 그 패치가 올바른지를 보장할 수 없고, 제대로 고쳐졌는지를 다시 확인하는 과정이 필요하게 된다. 또한 테스트 케이스로 오류의 존재 여부를 확인하기 어려운 경우(e.g. memory leak)에도 적용이 어렵다는 단점이 있다.

Ⅳ. 결 론

이 글에서는 소프트웨어 오류 자동 수정 기법에 대해

서 간략히 소개하였다. 메모리 오류를 자동으로 수정하는 정적 분석 기반의 새로운 기법을 소개하였고, 기존에 발표된 테스트 케이스 기반의 기법들을 소개하였다. 소프트웨어 오류 자동 수정 기술은 여전히 초기 연구단계에 있으며 향후 더 많은 기법과 연구결과가 기대된다.

참 고 문 헌

[1] T. Britton et al. Reversible Debugging Software.
[2] B. Liblit et al. Bug isolation via remote program sampling. In PLDI. 2003.
[3] Weimer et al. Automatically Finding Patches using Genetic Programming. In ICSE. 2009.
[4] Nguyen et al. SemFix: program repair via semantics analysis. In ICSE. 2013.
[5] Fan Long and Martin Rinard. Automatic Patch Geneartion by Learning Correct Code. In POPL. 2016

〈저 자 소 개〉

오 학 주 (Hakjoo Oh)
2015년 3월~현재 : 고려대학교 컴퓨터학과 조교수
2012년 3월~2015년 2월 : 서울대학교 컴퓨터공학과, 박사 후 연구원
2012년 2월 : 서울대학교 컴퓨터공학과 박사
2007년 2월 : 서울대학교 컴퓨터공학 석사
2005년 2월 : 한국과학기술원 전산학 학사
관심분야 : 프로그램 분석, 수정, 합성