CREST-BV: 임베디드 소프트웨어를 위한 Bitwise 연산을 지원하는 Concolic 테스팅 기법

(CREST-BV: An Improved Concolic Testing Technique Supporting Bitwise Operations for Embedded Software)

김 윤 호 † 김 문 주 †† 장 윤 규 †††
(Yunho Kim) (Moonzoo Kim) (Yoonkyu Jang)

요 약 기존 소프트웨어 테스팅 기법은 개발자가 수동으로 테스트 케이스를 작성해야 하는 비효율성으로 인해 임베디드 소프트웨어의 안정성 확보에 어려움이 있다. Concolic 테스팅 기법은 자동으로 소프트웨어 테스트 케이스를 생성함으로써 기존 소프트웨어 테스팅 기법의 문제를 해결했지만, 임베디드 소프트웨어 테스트에 필수적인 bitwise 연산을 지원하지 않는 문제가 있었다. 본 논문에서는 임베디드 소프트웨어를 위해 bitwise 연산을 지원하는 Concolic 테스팅 개선 방법을 연구하고 오픈 소스 임베디드 소프트웨어 Busybox에 적용하여 기존 연구와 분기 커버리지 및 실행 속도를 비교하였다. Busybox의 10개 유틸리티에 적용한 결과 기존 연구 결과 대비 평균 33%의 분기 커버리지 향상이 있었다. 또한, CREST-BV의 성능 저하 원인을 분석하였다.

키워드 : Concolic 테스팅, 자동 테스트 케이스 생성, 임베디드 소프트웨어, Bitwise 연산, CREST-BV

Abstract Conventional testing methods often fail to assure reliability of embedded software due to inefficiency and ineffectiveness of manual test case generation. Concolic testing can address this by automatically generating test cases but has a limitation that does not support bitwise operations critical to embedded software. To address this limitation, we have developed an improved concolic testing technique with bitwise operations support for embedded software. We applied the improved technique to 10 utilities of open-source embedded software Busybox and achieved 33% more branch coverage on average. In addition, we analyzed the performance bottleneck of the improved technique.

Key words : Concolic testing, Automated test case generation, Embedded software, Bitwise operations, CREST-BV

· 본 연구는 교육과학기술부/한국연구재단 우수연구센터 육성사업[2012-0000473], 지식경제부/한국산업기술평가관리원 IT R&D 프로그램[10041752, 초소형·고신뢰 OS와 고성능 멀티코어 OS를 동시 실행하는 듀얼 운영체제 원천 기술 개발], 삼성전자의 연구비 지원으로 수행되었습니다.
· 이 논문은 2012 한국컴퓨터종합학술대회에서 'CREST-BV: 임베디드 소프트웨어를 위한 Bitwise 연산을 지원하는 Concolic 테스팅 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : KAIST 전산학과
kimyunho@kaist.ac.kr
†† 종신회원 : KAIST 전산학과 교수
moonzoo@cs.kaist.ac.kr
(Corresponding author임)
††† 비 회 원 : 삼성전자 소프트웨어센터 수석연구원
yoonkyu.jang@samsung.com

논문접수 : 2012년 7월 19일
심사완료 : 2012년 10월 25일

정보과학회논문지: 소프트웨어 및 응용 제40권 제2호(2013.2)

1. 서 론

임베디드 소프트웨어에 오류가 발생할 경우 인명 피해[1,2], 경제적 피해[3]가 발생하는 등의 사회/경제적인 문제가 발생할 수 있다. 하지만, 기존의 소프트웨어 테스팅 기법은 테스트 케이스를 개발자가 직접 작성하기 때문에 테스트 케이스를 작성하는데 많은 시간과 노력이 들고, 개발자가 생각하지 못하는 경우를 테스트하지 못하는 등 테스트의 효과가 떨어진다. 따라서 테스트를 통해 예외적인 상황에서 발생하는 오류를 효과적으로 찾지 못해 임베디드 소프트웨어의 안정성 확보가 어렵다.

Concolic(CONCrete + symbOLIC) 테스팅 기법(혹은 동적 기호 실행 기법(dynamic symbolic execution))[4,5]은 동적 실행 기법과 기호 실행 기법을 결합하여 가능한 모든 실행 경로를 테스트하는 소프트웨어 테스트 케이스 집합을 자동으로 생성하는 기법이다. 테스트 케이스

를 자동으로 생성하기 때문에 테스트에 필요한 개발자의 노력을 최소화 할 수 있고, 특히 모든 실행 가능한 경로를 테스트 할 수 있게 테스트 케이스를 생성하기 때문에 예외적인 상황에서 발생할 수 있는 숨겨진 버그를 발견하는데 효과적이다[6,7]. 하지만 사례 연구[6] 결과 현재 Concolic 테스팅 기법이 bitwise 연산(&, |, ^, << 등)을 지원하지 못하는 한계가 있어 bitwise 연산이 중요한 임베디드 소프트웨어에 적용하기 위한 개선이 필요하다.

본 논문에서는 임베디드 소프트웨어 테스트에 필요한 bitwise 연산을 지원하는 Concolic 테스팅 기법의 개선 방법을 연구하고 Concolic 테스팅 도구 CREST-BV를 개발하였다. CREST-BV의 bitwise 연산 지원의 효과를 보이기 위해 오픈 소스 임베디드 소프트웨어 Busybox[8]의 10개 유틸리티에 대해 CREST-BV를 적용하였으며, 기존 Concolic 테스팅 도구 대비 평균 33%의 분기 커버리지 향상이 있었다. 또한, bitwise 연산을 지원하기 위해 발생한 성능 저하 문제를 분석하여 CREST-BV의 개선 방안을 논의하였다.

2. 관련 연구

처음 Concolic 테스팅 기법이 제안된 후 다양한 프로그램 언어를 대상으로 Concolic 테스팅 관련 연구가 진행되었다. CREST[9], CUTE[10], DART[11]는 C 프로그램 언어를 대상으로 개발된 Concolic 테스팅 도구이다. 이 도구들은 경로 제약 조건식을 추출하기 위해 instrumentation 기반 기법을 사용하였다. 그 외에도 LLVM 가상 머신 언어를 사용한 KLEE[12], C# 언어를 지원하기 위해 .NET 가상 머신을 사용한 PeX[13], Java 언어를 지원하는 Symbolic PathFinder[14] 등의 Concolic 테스팅 도구가 개발되어 활용되고 있으며 컴파일된 x86 바이너리 프로그램을 테스트하기 위한 도구들(BitBlaze[15], SAGE[16], S2E[17])도 개발되어 보안 취약점을 점검하고 프로그램 역공학을 수행하는데 유용하게 활용되고 있다.

Concolic 테스팅 기법은 임베디드 소프트웨어를 테스트 하기 위한 방법으로도 사용되고 있다. [18]에서는 Concolic 테스팅 기법을 사용해서 Flash 메모리 드라이버를 테스트 하였으며 모델 체킹 기법과의 장, 단점 비교 분석을 하였다. [6]에서는 모바일 폰 플랫폼 소프트웨어에 Concolic 테스팅 기법을 적용하여 Concolic 테스팅 기법을 실제 소프트웨어 프로젝트에 적용할 때 발생하는 어려움과 현재 개발된 Concolic 테스팅 도구의 한계에 대해 분석하였다. [7]에서는 임베디드 플랫폼에서 사용되는 소프트웨어 라이브러리 libexif를 테스트하고 임베디드 소프트웨어에 적합한 Concolic 테스팅 기법을 비교 분석하였다.

Concolic 테스팅 도구 중에 Bitwise 연산을 지원하고 임베디드 소프트웨어를 대상으로 적용된 도구는 KLEE가 있다. [12]에서 KLEE를 Busybox에 적용하여 효과적으로 버그를 발견하였다. 그러나 [7]의 비교 연구 결과 CREST-BV가 KLEE에 비해 10~28배 더 테스트 케이스 생성 속도가 빨랐으며, 실험 당시 최신 버전의 Busybox 코드가 LLVM bytecode로 컴파일하는 데 오류가 발생하는 등의 문제가 있어, CREST-BV가 KLEE에 비해 임베디드 소프트웨어 테스트하는데 적합하다.

3. CREST-BV: Bitwise 연산을 지원하는 Concolic 테스팅 도구 확장

3.1 Concolic 테스팅 기법

Concolic 테스팅 기법은 주어진 테스트 케이스를 사용해서 테스트 대상 프로그램을 수행하고, 실행된 프로그램 경로를 따라 경로 제약 조건을 생성한다. 테스트 대상 프로그램이 종료되면 경로를 따라 생성한 경로 제약 조건식을 활용해서, 기존에 실행되지 않은 경로를 실행하기 위한 테스트 케이스를 생성한다. 이 과정은 모든 실행 가능한 경로가 실행되거나, 사용자가 지정한 종료 조건을 만족할 때까지 반복된다.

그림 1의 경우를 예로 살펴보자. 그림 1의 왼쪽 예제 프로그램은 세 정수 값을 입력으로 받아 가장 큰 값을 돌려주는 함수이고, 오른쪽 그래프는 해당 예제 프로그램의 전체 실행 경로를 나타내는 그래프이다. 초기 입력 값이 x=1, y=1, z=1로 주어졌을 때 3번째 줄의 if 조건문과 4번째 줄의 if 조건문을 만족하게 되고 테스트 대상 프로그램이 종료될 때, 경로 조건 (x>=y)∧(x>=z)가 생성된다. 기존에 실행되지 않은 경로를 실행하기 위해 마지막 분기 조건 x>=z을 부정해서 새로운 경로 조건 (x>=y)∧(x<z)을 생성하고 constraint solver를 사용해서 이 새로운 경로 조건을 풀어내서 새로운 경로 조건을 만족하는 테스트 케이스 x=1, y=1, z=2를 생성한다. 이렇게 얻은 새로운 테스트 케이스를 사용해서 테스트 대상 프로그램을 다시 실행하고 이 과정을 모든 실행 경로가 실행되거나 사용자 지정 종료 조건(시간 제한, 생성할 테스트 케이스 수 제한 등)을 달성할 때까지 반복하게 된다.

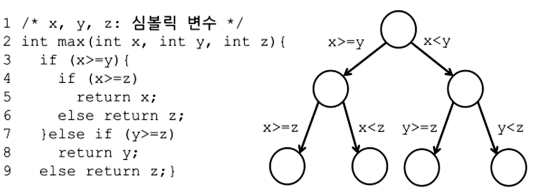

그림 1 3개의 분기문을 갖는 예제 프로그램 및 실행 경로

3.2 CREST-BV 구조

CREST-BV는 C 프로그램을 대상으로 하며, 기존 오픈 소스 instrumentation 기반 Concolic 테스팅 도구 CREST를 확장해서 구현되었다. 기존의 CREST는 bit-wise 연산을 지원하지 않아 bitwise 연산이 중요한 임베디드 소프트웨어에 적용했을 때 높은 분기 커버리지를 달성하지 못했고 버그 발견 능력도 떨어졌다. 이와 같은 문제를 해결하기 위해 CREST-BV는 CREST를 확장하여 bitwise 연산을 지원하도록 개선하였다.

CREST와 CREST-BV는 크게 instrumentation을 수행하는 front-end, 기호 실행을 수행하는 middle-end, 경로 제약 조건식을 풀고 테스트 케이스를 생성하여 테스트 대상 프로그램을 실행하는 back-end로 구성되어 있다(그림 2). Front-end는 C로 작성된 테스트 대상 프로그램의 소스코드를 입력으로 받아서 기호 실행을 수행하기 위한 탐침 함수(probe function)를 삽입(instrumentation)한다. 먼저 주어진 소스코드의 부차효과(side-effect)를 제거하고 원래 입력과 의미가 같은 정규화된 C 소스 모드를 생성한 후 분기문과 대입문에 어떤 분기가 실행되고, 변수 값이 어떻게 바뀌는지를 기록하기 위한 탐침 함수를 삽입한다. 테스트 대상 프로그램이 실제 수행되는 동안 탐침 함수들이 같이 실행되면서 기호 실행을 수행한다.

Middle-end는 기호 실행을 담당하는 라이브러리 함수들로 구성되어 있으며 front-end에서 생성된 instrumentation 된 소스 코드와 함께 컴파일 되어 테스트 대상 프로그램의 바이너리를 생성한다. CREST-BV 기호 실행 라이브러리 함수는 탐침 함수가 실행될 때 같이 실행되면서 주어진 심볼릭 변수와 실행 경로에 대한 기호 실행을 수행한다. 대입문이 수행되면 심볼릭 변수가 갱신되어야 하는지 확인하고 필요한 경우 심볼릭 변수 값을 갱신하며, 분기문이 수행되면 해당 시점에서의 심볼릭 변수 값과 실행된 분기문의 평가 결과를 기반으로 심볼릭 분기 조건 c_i를 기록한다. 테스트 대상 프로그램이 끝나면 분기문이 수행될 때 기록한 심볼릭 분기 조건을 실행 순서대로 논리곱으로 연결하여 심볼릭 경로 제약 조건식 ϕ: $c_1 \wedge c_2 \wedge \cdots \wedge c_n$을 생성한다.

Back-end는 middle-end 에서 생성한 심볼릭 경로 제약 조건식을 사용해 새로운 테스트 케이스를 생성하고 생성된 테스트 케이스로 테스트 대상 프로그램을 실행시키는 역할을 한다. 기호 실행 결과 생성되는 심볼릭 경로 제약 조건식 ϕ에서 하나의 분기 조건문 c_j (1 <= j <= n)를 선택해서 부정한 새로운 경로 제약 조건 ψ: $c_1 \wedge \cdots \wedge \neg c_j$를 생성하고 ψ를 Yices[19]나 Z3[20] 등의 SMT solver[21]를 사용해서 만족하는 해가 있는지 풀어낸다. 만족하는 해가 있으면 구한 해를 새로운 테스트 케이스로 사용하고 없으면 새로운 분기 조건문을 골라서 부정한다.

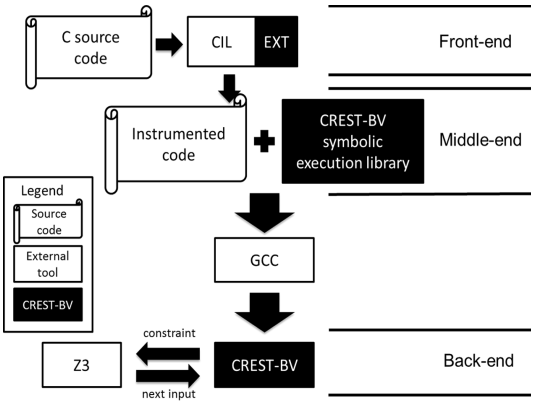

그림 2 CREST-BV 구조

3.3 CREST-BV 확장

CREST-BV는 bitwise 연산을 지원하도록 기존 CREST의 front-end, middle-end, back-end를 확장하였다. 먼저 front-end를 확장하여 bitwise 연산자가 실행될 때 같이 실행하기 위한 probe 함수를 개발하고 CIL (C Intermediate Language)[22]의 instrumentation 엔진을 확장해서 bitwise 연산자에 대한 probe를 삽입할 수 있도록 하였다. 또한 기존의 probe 함수를 확장해서 변수의 값뿐만 아니라 해당 변수 값의 유효 크기 및 타입 정보를 같이 middle-end 로 넘기도록 하였다. 기존에는 변수 값이 모두 자연수 타입에 해당하기 때문에 별도의 타입 정보 없이 심볼릭 경로 제약 조건식을 생성하고 풀 수 있었지만, CREST-BV는 bit-vector를 사용해 연산을 수행하므로 현재 계산된 변수 값의 유효 크기가 어느 정도인지를 정확하게 구해서 해당 크기만큼의 bit 값만 사용해야 한다. 변수 값의 유효 크기는 차후 back-end 에서 경로 제약 조건식을 정확하게 풀기 위해 다시 활용된다.

두 번째로 middle-end를 확장하여 기호 실행 라이브러리 함수들이 bit 수준 연산을 지원할 수 있도록 확장하였다. 기존 middle-end는 표현식을 1차원 배열 형태로 나타냈기 때문에 비 선형 표현식을 다룰 수 없었다. 예를 들어 ax + by + c(a, b, c: 정수 상수, x, y: 심볼릭 변수)를 표현하기 위해서는 크기가 3인 1차원 배열을 a를 생성하고 a[0]에는 상수를, a[1]에는 첫 번째 심볼릭 변수의 계수를, a[2]에는 두 번째 심볼릭 변수의 계수를 넣는 방법으로 구현되어 있었다. 이와 같은 구현 방법은 분기 조건문이 정수 선형 표현식일 경우에는 분기 조건문의 사칙 연산을 빠르게 처리할 수 있고 메모리 사용량이 낮지만, 비 선형 분기 조건문을 전혀 표현할 수 없

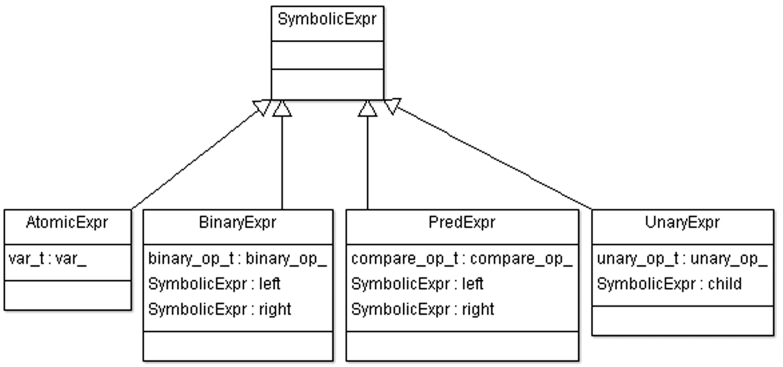

그림 3 경로 조건식 표현을 위한 클래스 계층 구조

는 문제가 있다. 따라서 일반적인 수식 나무 형태로 middle-end의 표현식 구현을 확장하였다. Middle-end의 표현식 구현은 그림 3과 같은 클래스 계층 구조로 되어 있다. 각각의 표현식은 SymbolicExpr 클래스를 상속받은 4개의 서브 클래스 중 하나로 표현된다. 4개의 서브 클래스는 각각 하나의 변수를 표현하는 AtomicExpr 클래스, +, -, /, &, << 와 같은 이진 연산자를 표현하는 BinaryExpr 클래스, <, >, == 등의 비교를 표현하는 PredExpr 클래스, !, ~ 등의 단항 연산자를 표현하는 UnaryExpr 클래스로 구성되어 있다.[1)]

마지막으로 back-end를 확장하여 bitwise 연산을 지원하는 경로 제약 조건식을 풀 수 있도록 개선하였다. 기존 CREST는 Yices SMT solver의 선형 정수 표현식 논리를 사용하여 경로 제약 조건식을 풀었지만 CREST-BV는 bit-vector 논리를 사용해서 경로 제약 조건식을 푼다. Bit-vector 논리는 숫자를 일련의 bit으로 나타내는 논리로 CPU가 수를 다루는 방식과 동일한 방법 및 연산자를 지원하기 때문에 CPU의 계산을 정확하게 따를 수 있다. 특히 기존 선형 정수 표현식에서는 표현할 수 없었던 정수 오버플로(overflow)와 언더플로(underflow) 등의 의미와 bitwise 연산도 정확하게 표현할 수 있어 프로그램의 의미를 정확하게 표현할 수 있는 장점이 있다. 하지만 SMT solver 의 bit-vector 논리를 사용해서 경로 제약 조건식을 풀기 위해서는 C 프로그래밍 언어의 암묵적인 형 변환 문제를 해결해야 한다.

C 언어의 이진 연산자는 두 피 연산자의 타입이 서로 다른 크기의 정수 타입 경우 암묵적인 형 변환을 수행한다. 예를 들어 char 타입의 변수 a와 int 타입의 변수 b를 더할 경우 서로 크기가 다르지만 암묵적 형 변환을 사용하여 계산을 수행하는 것이다. 하지만 SMT solver의 bit-vector 논리는 암묵적 형 변환을 지원하지 않고 +,-와 같은 이진 사칙 연산자의 피 연산자의 크기가 항상 같아야 하므로 문제가 발생한다. 이와 같이 타입의 크기가 맞지 않는 문제가 발생할 경우 CREST-BV의 back-end에서 C 프로그램 언어 표준에서 정의한 연산자 승급 규칙을 따라 피 연산자의 크기를 동일하게 만들어 준 후 경로 제약 조건을 풀었다. CREST는 선형 정수 표현식의 경로 제약 조건식을 풀기 위해서 Yices SMT solver를 사용하였다. 하지만, Yices는 라이브러리로 사용할 때 bit-vector 논리의 /,% 연산을 지원하지 않는 문제가 있어 CREST-BV는 Z3 SMT solver를 사용하였다.

4. CREST-BV 적용 사례 연구

4.1 실험 환경

Busybox는 임베디드 환경에서 리눅스/유닉스의 커맨드 라인 유틸리티들을 제공하는 소프트웨어다. 본 논문에서는 Busybox 1.17.0 버전의 유틸리티 가운데 10개 유틸리티(cp, cut, expr, grep, ls, mv, od, printf, tr, vi)를 대상으로 CREST-BV와 기존 CREST를 적용하여 분기 커버리지와 실행 시간을 비교하였다. 모든 실험은 Intel Core2Duo E8600@3.3GHz CPU, 8GB 메모리를 장착한 하드웨어와 Debian 6.0.4 32bit OS 환경에서 수행되었다. 분기 커버리지 효과 및 실행 성능 비교를 위해서 CREST-BV(Z3 2.19 사용)와 선형 정수 표현식만 지원하는 CREST(Yices 1.0.29 사용)를 사용해서 결과를 비교하였다. 경로 탐색 방법으로는 깊이 우선 탐색 방법을 사용하였으며, 최고 10,000개의 테스트 케이스를 생성할 수 있도록 제한하였다.

4.2 실험 결과

표 1은 CREST-BV와 CREST를 적용한 실험 결과를 나타낸다. 첫 열과 두 번째 열은 각각 테스트 대상

1) &&, ||와 같은 논리곱, 논리합 연산자는 front-end 에서 부차효과를 제거하는 과정에서 사라지고 동일한 의미를 갖는 nested if 구문으로 대체된다.

표 1 Busybox 10개 유틸리티에 CREST-BV와 CREST를 적용한 결과

Target	# of Brs.	Cov.(%)		Time(s)	
		BV	LIA	BV	LIA
cp	22	**50.0**	36.4	1.4	0.1
cut	102	**45.1**	2.9	30.0	0.2
expr	152	67.1	**67.8**	252.7	34.8
grep	168	**76.8**	41.7	1,245.3	0.1
ls	240	**69.2**	35.8	232.3	0.1
mv	46	17.4	17.4	0.2	0.2
od	72	**94.4**	79.2	178.6	21.9
printf	148	**72.3**	67.6	302.9	44.3
tr	130	**46.2**	35.4	495.6	27.7
vi	1512	46.4	**55.4**	576.2	1,722.4
Avg.	259	**58.5**	44.0	331.5	185.2

프로그램의 이름과 총 분기문의 수를 나타내며 세 번째, 네 번째 열은 각각 CREST-BV(열 이름 BV)와 CREST(열 이름 LIA)를 사용했을 때의 분기 커버리지를 나타낸다. 다섯 번째, 여섯 번째 열은 총 수행에 걸린 시간을 초 단위로 나타낸 것이다.

CREST의 평균 분기 커버리지가 44.0%인데 반해 CREST-BV는 분기 커버리지 58.5%를 달성하였다. 이는 기존 결과 대비 약 33%((58.5-44.0)/44.0*100) 더 향상된 결과이다. 총 10개 유틸리티 가운데 7개 유틸리티에서(expr, mv, vi 제외) CREST-BV가 더 높은 분기 커버리지를 달성할 수 있었다. 특히 cut의 경우 기존 CREST 로는 2.9%의 분기문 밖에 커버할 수 없었으나, CREST-BV를 사용해서 42.2%의 분기문을 더 커버할 수 있었다. vi, expr의 경우 기존 bitwise 연산 지원으로 인해 Concolic 테스팅으로 실행 가능한 경로가 더 늘어나 10,000개의 테스트 케이스로 충분히 가능한 실행 경로를 테스트 할 수 없었기 때문에 분기 커버리지가 더 떨어지는 결과가 발생하였다.

실행 시간의 경우 8개 유틸리티(mv, vi 제외)에서 CREST-BV가 더 오래 걸렸다. 이는 기존 CREST에 비해 CREST-BV에서 수행하는 기호 실행 방법이 더 복잡해지고, bit-vector 논리의 경로 조건식은 선형 정수 표현식으로 표현된 경로 조건식보다 더 풀기가 어렵기 때문이다. CREST-BV와 CREST의 성능 비교는 5장에서 자세하게 분석하였다.

5. CREST-BV 성능 분석

본 장에서는 CREST-BV와 CREST의 실행 시간을 기호 실행 측면과 경로 제약 조건을 푸는 측면에서 상세히 비교 분석하였다. CREST-BV는 CREST에 비해 약 33% 더 향상된 분기 커버리지를 달성할 수 있었으나 실행 시간이 약 1.8배 더 걸렸다. 따라서, CREST-BV의 실용성을 높이기 위해선 실행 시간이 증가한 원인을 분석하고 속도 문제를 해결해야 한다. 정확한 성능 분석을 위해서 CREST-BV와 CREST가 모두 10,000개의 테스트 케이스를 생성할 수 있었던 expr, od, printf, tr, vi 의 5개 유틸리티를 대상으로 비교, 분석하였다.

5.1 실행 시간 비율 분석

그림 4는 CREST-BV와 CREST의 실행 시간 비율을 비교한 그래프이다. CREST-BV 및 CREST의 실행 시간은 크게 세 부분으로 나눌 수 있다. 테스트 대상 프로그램이 실행될 때 기호 실행을 수행하는 시간(맨 밑의 Concolic exec. 계열), 테스트 대상 프로그램을 실행하기 위해 운영체제 커널을 수행하는 시간(가운데 System exec. 계열), 그리고 SMT solver를 사용해서 경로 실행 조건식을 풀고 새로운 테스트 케이스를 생성하는 시간(맨 위 SMT solving 계열)이다.

기존 CREST 의 주요 성능 병목 구간은 경로 제약 조건을 풀기 위한 SMT solving 시간이었다. [18]에서 분

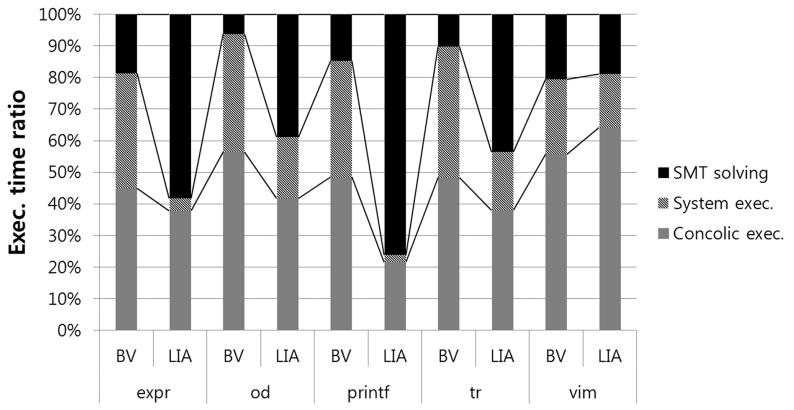

그림 4 CREST-BV와 CREST의 실행 시간 비율

석한 바와 같이 하나의 테스트 케이스를 생성하기 위해 약 10배에 달하는 SMT formula 를 풀어야 하기 때문에 SMT solving에 걸리는 시간이 전체 시간의 50% 이상 차지하였기 때문이다. 그림 4의 LIA 계열을 살펴보면 Busybox에 CREST를 적용했을 때 SMT solver 가 차지하는 시간이 전체의 약 40%(od)에서 약 76%(printf)에 달해 SMT solving 이 주요 성능 병목 구간임을 알 수 있다.

반면 CREST-BV를 적용했을 때 실행 시간 비율은 CREST를 적용했을 때와 크게 달랐다. 가장 큰 차이는 운영체제 커널이 실행되는 시간의 비율이 증가한 것이다. CREST를 적용했을 때 운영체제 커널이 실행되는 시간의 비율은 3%(printf)에서 20%(od) 정도였으나 CREST-BV를 적용했을 때는 34%(expr)에서 40%(tr)로 증가하였다. 특히 printf의 경우 3%에서 35%로 가장 많은 증가를 보였다.

실행 시간 비율 분석 결과 CREST-BV의 성능 병목 구간은 기호 실행 수행 시간과 운영체제 커널 실행 시간으로 볼 수 있으며 CREST에 비해 SMT solving 시간의 비율은 감소하였다.

5.2 기호 실행 시간 분석

3.2절에서 설명한 바와 같이 CREST-BV의 기호 실행은 CREST에 비해 복잡해졌다. CREST에서는 경로 제약 조건식이 선형 정수 표현식이기 때문에 1차원 배열을 사용해서 경로 제약 조건식을 표현할 수 있었다. 하지만 CREST-BV의 경로 제약 조건식은 비선형 정수 표현식이기 때문에 더 이상 간단한 배열의 형태로 나타낼 수 없다. 따라서 일반화된 수식 나무 형태로 경로 제약 조건식을 표현하였다. 수식 나무를 다루는 것은 배열을 다루는 것에 비해 새로운 메모리 할당 등의 복잡한 연산을 사용하므로 기호 실행 시간이 길어지게 된다.

표 2는 CREST-BV와 CREST의 평균 기호 실행 경로 길이와 평균 기호 실행 시간을 보여주고 있다. 먼저 평균 기호 실행 경로 길이를 살펴보면 CREST-BV보다 오히려 CREST가 더 긴 것을 알 수 있다. 일반적으로 동일한 경로에 대한 기호 실행을 수행할 경우 CREST-BV가 CREST보다 더 긴 기호 실행 경로를 생성한다. 하지만 동일한 유틸리티에 대해 동일한 개수만큼 테스트 케이스를 생성하더라도, 기호 실행 방법의 차이로 인해 CREST-BV와 CREST가 탐색하는 실행 공간이 서로 달라지기 때문에 CREST-BV의 기호 실행 경로 길이가 CREST보다 짧아질 수 있다.

표 2 기호 실행 경로 길이 및 평균 실행 시간

Targets	Sym. path len.		Avg. sym. exec. time(s)	
	BV	LIA	BV	LIA
expr	73.8	74.3	1.8	0.2
od	26.9	38.1	4.6	0.2
printf	72.1	84.9	1.7	0.2
tr	86.9	90.8	1.7	0.1
vi	1,200.3	5,566.3	0.3	0.2
Avg.	292.0	1170.9	2.0	0.2

한 연산에 대한 평균 기호 실행 시간은 CREST-BV가 CREST에 비해 약 8.5~23배 더 길다. 예를 들어 경로 제약 조건식 ax + b(a, b는 정수 상수, x는 심볼릭 변수)와 cy + d(c, d는 정수 상수, y는 심볼릭 변수)를 더하는 기호 실행을 수행하는 경우를 살펴보자. 기존 CREST에서는 2개의 서로 다른 심볼릭 변수의 계수와 상수를 저장할 크기 3의 배열에 x의 계수의 합 a, y의 계수의 합 c, 상수의 합 b+d를 계산해서 그 결과를 기존의 배열에 저장하면 되기 때문에 3번의 사칙연산만 수행하면 된다. 반면 CREST-BV에서는 먼저 이진 연산자를 저장하기 위한 수식 나무의 노드를 새로 생성하고 해당 노드에 두 피 연산자를 대입하기 때문에 추가적인 메모리 할당에 많은 시간이 소요된다. 특히 경로 제약 조건식과 상수를 더할 경우 CREST는 1번의 사칙연산만 수행하면 되지만, CREST-BV는 상수를 나타내는 수식 나무의 노드를 새로 생성하고 해당 연산자의 수식 나무 노드를 생성해야 하기 때문에 2번의 메모리 할당이 필요하다. 따라서 CREST-BV의 한 연산에 대한 평균 기호 실행 시간이 CREST보다 길어지게 된다.

5.3 운영체제 커널 실행 시간 분석

Concolic 테스팅을 수행하는 동안 운영체제 커널은 테스트 대상 프로그램을 실행하고 종료하기 위한 프로세스 생성/종료 및 스케쥴링, 메모리 관리 등을 수행한다. 특히 Concolic 테스팅은 잦은 테스트 대상 프로그램 실행으로 인해 프로세스 생성/종료로 인한 운영체제 커널 실행 시간이 무시 못 할 수준이다. 따라서 본 논문에서는 Oprofile[23]을 사용해서 리눅스 커널을 프로파일링하였다.

표 3은 CREST-BV와 CREST를 실행했을 때 오래 실행된 커널 함수 상위 5개를 나타낸다. 실행 시간 점유 비율은 각 테스트 대상 유틸리티를 프로파일링한 결과의 평균으로 계산했으며 각 유틸리티 별 커널 함수 실행 시간 분포 비율은 크게 다르지 않았다. 상위 5개의 커널 함수 모두 메모리 관리에 관련된 함수임을 확인할 수 있다.

CREST-BV와 CREST가 차이를 보이는 부분은 CREST-BV에서 주로 실행된 copy_user_highpage 함수이다. 해당 함수는 사용자 프로세스가 사용하고 있는 메모리 페이지를 복사하는 함수로써 리눅스 프로세스가 새로 생성될 때 메모리 페이지의 Copy-on-Write기능

표 3 실행 시간 기준 상위 5위 커널 함수

Rank	BV		LIA	
	Func. name	% in system exec	Func. name	% in system exec
1	copy_user_highpage	37.9	page_fault	28.1
2	native_flush_tlb_single	11.4	native_flush_tlb_single	22.7
3	page_fault	8.3	Kunmap_atomic	8.2
4	copy_page_ranget	5.8	unmap_vmas	7.6
5	unmap_vmas	3.9	read_hpet	7.1
	Sum	67.2	Sum	73.8

을 구현하는데 주로 사용된다. 즉, CREST-BV가 생성한 자식 프로세스는 CREST가 생성한 자식 프로세스에 비해 Copy-on-Write를 통한 메모리 페이지 복사가 더 많이 발생하는 것이다.

CREST-BV의 잦은 페이지 복사의 이유는 CREST-BV의 메모리 사용량이 CREST보다 높다는 것으로 추정된다. CREST-BV가 CREST보다 더 많은 페이지에 데이터를 쓰게 되고 더 많은 Copy-on-Write가 발상하는 것으로 추정할 수 있다. 더 상세한 분석을 통해 메모리 페이지 복사의 원인을 구체적으로 밝히려고 노력했으나 사용자 프로세스와 커널 프로세스 사이의 메모리 맵핑 및 커널 연산의 콜 스택을 상세하게 모니터링 하기 현실적으로 어렵고 본 연구의 범위를 벗어나기 때문에 구체적인 원인 규명은 후속 연구로 남겨 놓는다.

5.4 경로 제약 조건 풀이 시간 분석

SMT solver를 사용해서 경로 제약 조건식을 푸는 것은 오랜 시간이 걸리는 작업이기 때문에 최대한 불필요한 경로 제약 조건식을 제거하기 위한 기법이 개발되었다. CREST-BV와 CREST는 경로 제약 조건식을 최적화하기 위해 문법적 모순 체크와 부정된 분기 조건 슬라이싱(slicing) 기법을 사용하였으며, 4.2절의 실험 결과는 두 기법을 모두 적용하고 난 후의 결과이다. 본 절에서는 CREST-BV와 CREST의 경로 제약 조건 풀이 시간을 분석하고 경로 제약 조건식의 길이를 줄이기 위한 기법의 효과를 분석한다.

문법적 모순 체크 기법은 현재 경로 제약 조건식을 SMT solver 를 사용해서 풀기 전에 분기 조건 가운데 서로 문법적으로 모순이 되는 경우가 있는지를 체크해서 제거하는 기법이다. 예를 들어 현재 풀어야 할 경로 제약 조건 ψ: $c_1 \wedge \cdots \wedge \neg c_n$이라고 할 때 c_1과 c_n이 문법적으로 서로 같은지 체크하는 것이다. c_1과 c_n이 서로 같다면 c_1과 $\neg c_n$이 동시에 만족될 수 없기 때문에 주어진 경로 제약 조건 ψ가 unsatisfiable 함을 SMT solver를 사용하지 않고 판별할 수 있다.

부정된 분기 조건 슬라이싱 기법은 경로 제약 조건을 전부 다 푸는 게 아니라 새로 바뀐 부분이 영향을 준 부분만 다시 풀고 나머지 부분은 기존의 해를 사용하는 방법이다. 예를 들어 테스트 케이스(x=1, y=1, z=1)을 실행해서 경로 제약 조건식 ϕ: x==z $\wedge$ y > 0 $\wedge$ z==1 을 생성하고 z==1을 부정한 ψ: x==z $\wedge$ y > 0 $\wedge$ z!=1 을 풀어야 하는 경우를 생각해보자. 마지막 조건의 부정으로 인해 변수 z의 값이 바뀔 수 있고 첫 번째 절에서 z와 의존성이 있는 변수 x의 값 역시 바뀔 수 있다. 하지만 y는 x, z와 의존성이 없기 때문에 현재 값을 그대로 사용할 수 있다. 따라서 이와 같은 경우 ϕ': x==z $\wedge$ z!=1 을 풀어서 x, z의 값만 새로 구하고 y는 기존의 값을 사용할 수 있다.

표 4는 문법적 모순 체크 기법이 얼마나 효과적으로 SMT solver 실행 횟수를 줄여줄 수 있는지 보여준다. CREST-BV의 경우 약 9~98% 경로 조건식이 문법적 모순 체크 기법으로 제거되었고 CREST의 경우 약 1~71%의 경로 조건식이 제거되었다. 표 4에서 한 가지 흥미로운 결과는 CREST-BV와 CREST를 적용했을 때 tr 유틸리티가 보여주는 양상이다. CREST-BV를 적용

표 4 문법적 모순 체크 기법 적용 결과

Targets	BV			LIA		
	# of total formulas	# of removed formulas	Reduction ratio(%)	# of total formulas	# of removed formulas	Reduction ratio(%)
expr	107,643	44,110	41.0	105,655	73,870	69.9
od	21,573	6,326	29.3	57,858	41,139	71.1
printf	62,296	5,547	8.9	100,016	19,445	19.4
tr	629,734	618,706	98.2	13,876	200	1.4
vi	293,348	176,668	60.2	359,972	191,544	53.2
Avg.	222,919	170,271	76.4	127,475	65,239	51.2

표 5 부정된 분기 조건 슬라이싱 기법 적용 결과

Targets	BV			LIA		
	Avg. len. of formulas	# of removed conditions	Reduction ratio(%)	Avg. len. of formulas	# of removed conditions	Reduction ratio(%)
expr	72.8	48.9	67.1	69.8	48.6	69.7
od	25.6	21.9	85.3	30.2	21.3	70.6
printf	26.1	7.0	26.6	76.1	58.4	76.7
tr	75.8	39.7	52.4	84.5	62.1	73.4
vi	1,300.9	1,248.5	96.0	5,487.8	5,427.3	98.9
Avg.	300.3	273.2	91.0	1,149.7	1,123.5	97.7

했을 경우 98.2%가 제거되는 반면 CREST를 적용할 경우 1.4%만 제거되는 정 반대의 결과를 보여주고 있다. CREST-BV와 CREST는 기호 실행 방법이 서로 다르기 때문에 서로 다른 실행 공간을 탐색하게 되며 이로 인해서 같은 테스트 대상 프로그램이라 할지라도 완전히 다른 실행 결과를 보여줄 수 있기 때문이다.

표 5는 부정된 분기 조건 슬라이싱 기법이 얼마나 효과적으로 경로 제약 조건식의 길이를 줄여줄 수 있는지를 보여준다. 문법적 모순 체크 기법을 적용하고 남은 경로 제약 조건식을 대상으로 평균 길이를 conjunction으로 연결된 절의 수로 계산하고 얼마나 많은 절(분기 조건문)이 제거되는지 계산하였다. CREST-BV의 경우 26.6~96%의 경로 조건이 제거되며 CREST의 경우 64.8~99.9%의 경로 조건이 제거되었다. 따라서 두 최적화 기법 모두 효과적으로 SMT solving 시간을 줄일 수 있음을 알 수 있다.

5.5 CREST-BV 성능 개선을 위한 제안

5.1~5.4절을 통해 CREST-BV 의 성능 저하 원인을 분석한 결과 느리고 복잡해진 기호 실행기법이 주 원인임을 파악할 수 있었다. 특히 비선형 경로 제약 조건식을 표현하기 위한 SymbolicExpr 클래스와 4개의 서브 클래스들의 복잡한 연산이 성능 저하의 직접적인 요인이었다. 또한 SymbolicExpr 클래스를 사용한 기호 실행 기법은 기존 CREST의 기호 실행 기법보다 더 많은 메모리를 사용하기 때문에 더 많은 메모리 페이지를 운영체제 커널에 요구하게 되고 이를 준비하기 위해 운영체제 커널이 수행되는 시간 또한 같이 증가시키는 문제점을 야기하였다. 따라서 CREST-BV의 성능을 개선하기 위해 경량화된 기호 실행 기법이 필요하다.

임베디드 소프트웨어에서 bitwise 연산이 사용되는 비율이 낮더라도 Concolic 테스팅을 통한 분기 커버리지 달성에 큰 영향을 줄 수 있다. 그림 5는 bitwise연산을 사용하는 예제 프로그램으로 주어진 입력 변수 x의 마지막 최하위 bit가 1이면(4번째 줄) 실제 연산을 수행하는 large_func() 함수를 호출하는(5번째 줄) 프로그램이다. 예제 프로그램에서 bitwise 연산은 4번째 줄에서

```
1 /* x: 심볼릭 변수 */
2 #define OPT_RUN 1
3 void func(int x){
4   if (x & OPT_RUN){
5     large_func();
6   }
7   return;
8 }
```

그림 5 Bitwise 연산을 사용하는 예제 프로그램

만 사용되고 5번째 줄에서 호출되는 large_func() 함수는 bitwise 연산을 사용하지 않는다고 가정하자. CREST를 사용하여 테스트 케이스를 생성할 경우 bitwise 연산을 지원하지 않기 때문에 4번째 줄의 조건을 만족하는 테스트 케이스를 생성하지 못할 수 있으며 large_func() 함수의 분기문을 커버하지 못 해 낮은 분기 커버리지를 달성하게 된다. 이와 같이, bitwise 연산이 전체 임베디드 소프트웨어에서 사용되는 비율이 낮더라도 분기 커버리지 달성에 중요한 영향을 미치기 때문에 Concolic 테스팅의 bitwise 연산 지원 여부가 중요하다. 실험 대상이었던 Busybox 10개 유틸리티의 기호 실행 경로를 살펴봤을 때 bitwise 연산의 비율은 10%가 되지 않았으며, 5.4절에서 설명한 슬라이싱 기법을 적용했을 때, 대부분 사라졌다. 따라서 대부분의 경우 기존 CREST로도 빠르고 효과적으로 분기 커버리지를 높일 수 있으며 꼭 필요한 경우에만 CREST-BV를 사용해 bitwise 연산을 지원하면 된다.

제안하는 방법은 다음과 같다. 먼저 CREST를 사용해서 Concolic 테스팅을 수행한다. Concolic 테스팅을 수행하다 어떤 특정 조건에 맞는 분기문을 부정해서 새로운 경로를 테스트하려고 시도했으나 실패했을 때 실패 원인을 데이터 흐름 분석 등으로 살펴보고 bitwise 연산이 포함되어 있을 경우 CREST-BV의 복잡한 기호 실행 기법을 사용해서 해당 경로를 다시 수행한다. 이와 같은 방법을 사용하면 대부분의 경로를 가벼운 기호 실행 기법을 사용하는 CREST를 사용해서 테스트할 수 있고, bitwise 연산이 문제가 되는 특정 경로에 대해서만 CREST-BV를 사용하기 때문에 전체적인 실행 속도

가 빨라지고 bitwise 연산으로 인한 테스트 효과 저하 문제도 해결할 수 있다.

6. 결론 및 향후 연구

본 논문에서는 임베디드 소프트웨어 테스트를 위한 bitwise 연산을 지원하는 Concolic 테스팅 기법을 연구하고 이를 구현한 CREST-BV를 개발하였다. CREST-BV를 오픈 소스 임베디드 소프트웨어 Busybox에 적용한 결과 기존의 CREST보다 약 33.1% 더 향상된 분기 커버리지를 달성할 수 있었다. 또한 기존 CREST 대비 느려진 실행 속도를 향상시키기 위해, CREST-BV의 실행 속도 저하 원인을 분석하고 CREST-BV의 실행 속도를 향상시키기 위한 가벼운 기호 실행 기법을 개발하는 연구를 향후 연구 주제로 제안하였다.

참 고 문 헌

[1] Nancy Leveson. Software: System Safety and Computers, Addison-Wesley, 1995.
[2] FDA Alerts and Notices, http://www.fda.gov/Radiation-EmittingProducts/RadiationSafety/AlertsandNotices/ucm116533.htm
[3] Toyota: Software to blame for Prius brakes problems, http://articles.cnn.com/2010-02-04/world/japan.prius.complaints_1_brake-system-anti-lock-prius-hybrid?_s=PM:WORLD
[4] C. Pasareanu, W. Visser, A Survey of New Trends in Symbolic Execution for Software Testing and Analysis, STTT 11(4), 2009.
[5] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, W. Visser, Symbolic Execution for Software Testing in Practice-Preliminary Assessment, ICSE, 2011.
[6] M. Kim, Y. Kim, Y. Jang, Industrial Application of Concolic Testing on Embedded Software: Case Studies, ICST, 2012.
[7] Y. Kim, M. Kim, Y. Kim, Y. Jang, Industrial Application of Concolic Testing Approach: A Case Study on libexif by Using CREST-BV and KLEE, ICSE, 2012.
[8] Busybox. http://www.busybox.net
[9] CREST. http://code.google.com/p/crest
[10] K. Sen, D. Marinov, G. Agha, CUTE: A Concolic Unit Testing Engine for C, ESEC/FSE, 2005.
[11] P. Godefroid, N. Klarlund, K. Sen, DART: Directed Automated Random Testing, PLDI, 2005.
[12] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs, OSDI, 2008.
[13] N. Tillmann, W. Schulte, Parameterized Unit Tests, ESEC/FSE, 2005.
[14] C. S. Pasareanu, N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode, ASE, 2010.
[15] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, BitBlaze: A New Approach to Computer Security via Binary Analysis, ICISS, 2008.
[16] P. Godefroid, M. Levin, D. Molnar, Automated Whitebox Fuzz Testing, NDSS, 2008.
[17] V. Chipounov, V. Kuznetsov, G. Candea, S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems, ASPLOS, 2011.
[18] M. Kim, Y. Kim, Y. Choi, Concolic Testing of the Multi-sector Read Operation for Flash Storage Platform Software, FACJ 24(2), 2012.
[19] B. Dutertre, L. Moura, A Fast Linear-arithmetic Solver for DPLL(T), CAV, 2006.
[20] L. Moura, N. Bjorner, Z3: An Efficient SMT Solver, TACAS, 2008.
[21] SMT-LIB: The Satisfiability Modulo Theories Library, http://combination.cs.uiowa.edu/smtlib/
[22] G. Necula, S. McPeak, W. Weimer, CIL:Intermediate Language and Tools for Analysis and Transformation of C Programs, CC, 2002.
[23] OProfile. http://oprofile.sourceforge.net

김 윤 호

2007년 KAIST 전산학과 학사. 2007년~현재 KAIST 전산학과 석/박사 통합과정 재학. 관심분야는 Concolic 테스팅, 소프트웨어 정적 분석, 정형검증

김 문 주

1995년 KAIST 전산학과 학사. 2001년 Univ. of Pennsylvania 박사. 2002년~2004년 SECUi.COM 차장. 2004년~2006년 POSTECH 연구원. 2006년~2012년 KAIST 전산학과 조교수. 2012년~현재 KAIST 전산학과 부교수. 관심분야는 Concolic 테스팅, Concurrency 테스팅, 정형검증, 내장형 소프트웨어

장 윤 규

1996년 KAIST 전산학과 학사. 1998년 KAIST 전산학과 석사. 2003년 KAIST 전산학과 박사. 2003년~2008년 삼성전자 SW센터 책임연구원. 2008년~2011년 삼성전자 DMC 연구소 책임연구원. 2011년~현재 삼성전자 SW센터 수석연구원 관심분야는 Web API 표준화, 소프트웨어 테스트, 프로그램 분석, 소프트웨어 재사용