ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 43, No. 6, pp. 692-699, 2016. 6
http://dx.doi.org/10.5626/JOK.2016.43.6.692

효과적인 내장형 소프트웨어의 정수 확장 (Integer Promotion) 버그 검출 기법

(Effective Integer Promotion Bug Detection Technique for Embedded Software)

김 윤 호 [†] (Yunho Kim)
김 태 진 [††] (Taejin Kim)
김 문 주 [†††] (Moonzoo Kim)
이 호 정 [††††] (Ho-jung Lee)
장 훈 [†††††] (Hoon Jang)
박 민 규 [††††] (Mingyu Park)

요 약 세탁기, 냉장고 등의 가전제품에 탑재되는 8-bit MCU용 C 컴파일러는 소프트웨어 실행 속도를 높이기 위해 표준 C 언어 규칙을 따르지 않고 컴파일을 수행할 수 있다. 개발자가 일반 C 컴파일러와 8-bit MCU용 C 컴파일러의 차이를 정확하게 이해하지 못할 경우 표준 C 언어 환경에서는 발생하지 않으나 8-bit MCU를 사용하는 내장형 시스템에서는 발생하는 버그를 야기할 수 있으며 이런 버그는 표준 C 언어 환경을 가정하는 버그 검출 도구로는 찾기 어렵다. 본 논문에서는 표준 C 정수 확장 규칙을 따르지 않는 8-bit MCU용 컴파일러를 사용할 때 발생하는 정수 확장 버그를 소개하고 정수 확장 버그를 탐지하기 위한 다섯 종류의 버그 패턴을 제안한다. 정수 확장 버그 패턴 검출 도구를 개발하여 LG전자 세탁기 소프트웨어를 분석한 결과 컴파일러 옵션을 잘못 선택한 경우 발생하는 27개의 정수 확장 버그를 발견하였다.

키워드: 소프트웨어 테스팅, 내장형 소프트웨어, 정수 확장 버그, 동적 기호 실행

Abstract C compilers for 8-bit MCUs used in washing machines and refrigerators often do not follow the C standard to improve runtime performance. Developers who are unaware of the difference between C compilers following the C standard and the C compilers for 8-bit MCU can cause bugs that do not appear in the standard C environment but appear in the embedded systems using 8-bit MCUs. It is difficult for bug detectors that assume the standard C environment to detect such bugs. In this paper, we introduce integer promotion bugs caused by the different integer promotion rules of the C compilers for 8-bit MCU from the C standard and propose 5 bug patterns where the integer promotion

· 본 연구는 한국연구재단 한-아프리카 협력기반조성사업(NRF-2014K1A3A1A09063167), 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터육성지원사업(IITP-2016-H8501161012000100 2), LG전자 H&A 제어연구소의 지원으로 수행되었음
· 이 논문은 제42회 동계학술발표회에서 '임베디드 소프트웨어에서 발생하는 정수 확장(Integer Promotion) 버그 검출을 위한 정적 분석 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : KAIST 전산학부(KAIST)
kimyunho@kaist.ac.kr
(Corresponding author임)
†† 학생회원 : KAIST 전산학부
taejin@kaist.ac.kr
††† 종신회원 : KAIST 전산학부 교수
moonzoo@cs.kaist.ac.kr
†††† 비 회 원 : LG전자 제어연구소 주임연구원
sophiahj.lee@lge.com
mingyu88.park@lge.com
††††† 비 회 원 : 현대자동차 샤시기술센터 연구원
hoonjang@hyundai.com

논문접수 : 2016년 2월 19일
(Received 19 February 2016)
논문수정 : 2016년 4월 7일
(Revised 7 April 2016)
심사완료 : 2016년 4월 11일
(Accepted 11 April 2016)

정보과학회논문지 제43권 제6호(2016. 6)

bugs occur. We have developed an integer promotion bug detection tool and applied it to the washing machine control software developed by the LG electronics. The integer promotion bug detection tool successfully detected 27 integer promotion bugs in the washing machine control software.

Keywords: software testing, embedded software, integer promotion bug, dynamic symbolic execution

1. 서 론

우리는 내장형 시스템을 사용하는 제품을 일상생활의 곳곳에서 마주하고 있다. 이러한 내장형 시스템의 세계 시장은 약 200조원에 육박하고 매년 9% 이상의 성장률을 보인다[1]. 이러한 급격한 성장에 따라, 내장형 소프트웨어의 품질을 향상시키는 것 또한 중요해지고 있다.

이러한 내장형 시스템 시장에서 8-bit MCU (Micro Controller Unit)의 시장 규모는 전체 MCU 시장의 약 40%를 차지할 정도로 거대하다[2]. 이러한 8-bit MCU에 특화된 소프트웨어에서만 발생하는 버그는 특정 시스템 내에서만 발생하므로 개발자들이 인지하기 어려움에도 불구하고, 이 버그를 검출하는 방법에 대한 연구는 부족한 실정이다.

8-bit MCU에 특화된 소프트웨어에서 발생하는 버그 중에는 대표적으로 정수 확장 방식이 표준 C와 달라 발생하는 정수 확장(integer promotion) 버그가 있다. 정수 확장이란 C 프로그램에서 피연산자 자료형의 크기가 int형보다 작은 경우 피연산자의 자료형을 암묵적으로 int형으로 변환하는 것이다. C 표준에 따라 정수 확장을 수행하는 표준 C 컴파일러와는 달리 일부 8-bit MCU 컴파일러는 8-bit MCU 에서 실행되는 소프트웨어의 성능 향상을 위해 C 표준을 따르지 않으며 정수 확장을 수행하지 않는다. 개발자들은 일반적으로 사용하는 컴파일러가 표준 C 규칙을 따라서 컴파일을 수행할 것으로 생각하기 때문에 정수 확장을 수행하지 않는 8-bit MCU 컴파일러를 사용했을 때 예상하지 못한 정수 넘침 등의 문제가 발생할 수 있다. LG전자의 한 개발부서 내부에서 자체 조사한 결과, 8-bit MCU와 관련된 개발자들 중 약 66%가 정수 확장 규칙 자체에 대해서 잘 모르거나 정수 확장 규칙을 알더라도 8-bit MCU 컴파일러에서 정수 확장 규칙이 적용되지 않을 수 있다는 점을 모른다고 답하였다.

본 논문에서는 8-bit MCU 컴파일러를 사용할 때 발생할 수 있는 정수 확장 버그를 소개하고 정수 확장 버그를 효과적으로 탐지하기 위한 검출 기법을 제안한다. 정수 확장 버그가 발생할 수 있는 다섯 종류의 정수 확장 버그 패턴을 C 프로그램 코드 구조 조건(syntactic condition)과 피연산자의 값 조건(semantic condition)을 사용하여 정의 하였다. 정수 확장 버그 패턴의 문법 구문 조건을 만족하는 C 프로그램 코드는 잠재적으로 정수 확장 버그를 발생시킬 수 있으며 프로그램 실행 시 문법 구문을 만족하는 코드에서 피연산자의 값 조건을 만족할 때 정수 확장 버그가 발생한다. 그리고 정수 확장 버그를 실제로 검출하기 위해 정적 분석과 동적 기호 실행 기법을 사용하여 코드 구조 조건과 피연산자의 값 조건을 검사할 수 있는 방법을 제안하였다.

제안하는 정수 확장 버그 탐지 기법이 효과적임을 보이기 위해 Clang/LLVM [3]을 이용하여 정수 확장 버그 패턴 검출기를 개발하고 LG전자의 세탁기 제어 소프트웨어(약 41KLOC규모 C코드)에 적용하였다. 그 결과 컴파일러 옵션을 잘못 선택한 경우 발생하는 27개의 정수 확장 버그를 발견할 수 있었다.

본 논문의 기여점(contribution)은 다음과 같다.

- 8-bit MCU 컴파일러의 정수 확장 규칙이 C표준과 다를 때 발생하는 정수 확장 버그를 효과적으로 탐지하기 위한 다섯 종류의 정수 확장 버그 패턴을 제안하였다.
- 정수 확장 버그 검출기의 거짓 경보를 줄이기 위해 동적 기호 실행 기반의 자동화된 유닛 테스팅 기법을 활용한 거짓 경보 제거 기법을 제안하였다.
- LG전자의 세탁기 제어 소프트웨어 사례 연구를 통해 제안하는 기법이 실제 내장형 소프트웨어의 정수 확장 버그를 효과적으로 탐지하고 거짓 경보를 제거할 수 있음을 확인하였다.

논문의 구성은 다음과 같다. 2장에서는 관련 연구를 소개하고 3장에서는 정수 확장 버그 패턴을 설명한다. 4장에서는 정적 분석 기법을 사용해서 정수 확장 버그 패턴을 탐지하기 위한 방법을 설명하고 5장에서는 동적 기호 실행 기법을 활용하여 정적 분석의 거짓 경보를 제거하는 방법을 설명한다. 6장에서는 사례 연구를 통해 제안하는 기법의 효과를 살펴본다. 마지막으로 7장에서 본 연구의 결론을 제시하며 논문을 마무리한다.

2. 관련 연구

정수 확장 버그는 정수형 변수의 크기에 따라 연산 결과를 다 저장하지 못해 발생하는 오류라는 점에서 일종의 정수 넘침 버그로 볼 수 있다. 정수 넘침 버그를 탐지하기 위한 기존의 방법은 주로 컴파일 타임에 instrumentation 을 통해 정수 넘침을 검출하기 위한 탐지 함수를 삽입하거나[4,5] 런타임에 바이너리 코드에 대한 동적 instrumentation 을 사용해 탐지 함수를 삽입하는[6,7] 방법을 택해왔다. RICH[4]는 컴파일 타임에 탐지 함수를 삽입하여 정수 넘침과 큰 자료형에서 작은

자료형으로 변환할 때 발생할 수 있는 값 정확도 하락을 탐지하였다. IOC[5]는 C 프로그램의 언어 규약에 정의되어 있는 않은 동작(undefined behavior)으로 인한 정수 넘침을 탐지하기 위한 도구이다. 컴파일 타임에 탐지 함수를 삽입하여 프로그램이 실행되면서 정수 넘침이 발생하는지 검사한다. BRICK[6]은 Valgrind[8] 도구를 사용하여 컴파일된 바이너리 프로그램의 정수 넘침을 탐지하였다. SmartFuzz[7]도 BRICK과 동일하게 Valgrind 도구를 사용하여 바이너리 프로그램을 분석하는 도구이다. SmartFuzz는 단순히 프로그램 실행 과정에서 정수 넘침을 검사할 뿐 아니라 동적 기호 실행 기법을 활용하여 자동으로 테스트 케이스를 생성하며 커버리지를 높여 나간다.

본 장에서 소개한 정수 넘침 검출 기법들은 컴파일러가 C 표준을 따른다고 가정하고 있기 때문에 컴파일러가 C 표준을 의도적으로 따르지 않는 경우에 발생하는 정수 확장 버그를 탐지하지 못한다. 본 논문은 일반적인 정수 넘침 버그가 아닌 8-bit MCU 등의 특화된 대상 플랫폼의 C 컴파일러를 사용할 때 발생하는 정수 확장 버그를 탐지하며 기존의 기법이 찾지 못하는 새로운 자료형의 버그를 검출한다.

3. 정수 확장 버그 패턴

이 장에서는 8-bit MCU에서 발생할 수 있는 정수 확장 버그를 예를 들어 설명하고 정수 확장 버그가 발생할 수 있는 다섯 종류의 버그 패턴을 정의한다.

3.1 정수 확장 버그 예제

정수 확장 버그는 동일한 C 프로그램을 C 정수 확장 규칙을 따르는 컴파일러와 8-bit MCU 컴파일러와 같이 표준 C의 정수 확장 규칙을 따르지 않는 컴파일러로 컴파일했을 때 서로 다른 실행 결과를 보이는 것을 의미한다. 표준 C를 따르는 컴파일러는 C 프로그램의 피연산자가 문자형(8-bit character)인 경우, 정수형으로 자료형을 확장(promotion) 한 뒤 연산을 수행하도록 컴파일한다. 반면 8-bit MCU컴파일러는 C 프로그램이 정수 확장 없이 피연산자의 자료형 그대로 연산을 수행하도록 컴파일한다.

예를 들어, 그림 1의 함수 f()의 4번째 줄에서 result 값은 표준 C를 사용하는 소프트웨어와 8-bit MCU의 소프트웨어에서 다르게 나타난다. 표준 C 소프트웨어는 부호없는 문자형인 op1, op2의 값을 덧셈하기 전에 op1, op2를 정수형으로 자료형 확장하여 계산하므로 op1 + op2가 정수형으로 계산되고 result에 400이 들어간다. 반면, 8-bit MCU의 소프트웨어는 op1, op2의 자료형인 부호없는 문자형 그대로 덧셈 연산을 하고, op1 + op2가 부호없는 문자형의 최댓값인 255를 초과하면서 넘침이

```
1  void f(){
2    unsigned char op1 = 200;
3    unsigned char op2 = 200;
4    int result = op1 + op2; }
5  void g(){
6    signed char op1 = -1;
7    unsigned char op2 = 1;
8      if (op1<op2){ /* C SW */ }
9    else { /* 8-bit MCU SW */ } }
```

그림 1 정수 확장 버그의 코드 예시
Fig. 1 An integer promotion bug example

발생한다. 따라서 result에는 144가 들어간다.

또한, 그림 1의 함수 g()의 7번째 줄 조건문의 참/거짓 여부도 정수 확장 여부에 따라 달라진다. C 표준을 따르는 컴파일러는 op1, op2를 모두 signed int 자료형으로 변환하는 정수 확장을 수행하고 조건을 비교하기 때문에 참인 분기문을 수행하지만 8-bit MCU 컴파일러는 정수 확장 없이 op1을 unsigned char 자료형으로 변환하여 비교하기 때문에 op1의 값이 255가 되어 거짓인 분기문을 수행하게 된다.

3.2 정수 확장 버그 패턴

표 1은 3.1절에서 설명한 정수 확장 버그가 발생하는 구문을 코드 구조 조건과 피연산자의 값 조건에 따라 다섯 종류 패턴으로 정의한 것이다. 1, 2번 패턴은 단항연산자, 3, 4번 패턴은 이항연산자, 5번 패턴은 삼항연산자이다. 표 1에서 uchar는 unsigned char 자료형을 의미하고, schar는 signed char 자료형을 의미한다. 코드의 구문이 연산자 형태, 피연산자 자료형 그리고 피연산자의 값 조건이 모두 일치하면 해당 구문은 정수 확장 버그가 발생한 것이다.

1번 패턴은 bit-wise not 연산에서 발생하는 정수 확장 버그를 찾는다. 예를 들어 int q = ~p; 와 같은 C 코드가 있고 p가 unsigned char 자료형일 때 C 표준을 따를 경우 p가 unsigned char 에서 int 로 형변환이 되고 bit-wise not 연산이 수행된다. 만약 p의 값이 0이었다면 q의 값은 2진수로 11111111 11111111 11111111 11111111 로 표현된다. 하지만 8-bit MCU 컴파일러는 정수 확장을 수행하지 않기 때문에 unsigned char 에서 int 형변환 없이 바로 bit-wise not 연산을 수행하게 되고 q의 값은 00000000 00000000 00000000 11111111 이 저장되어 서로 다른 실행 결과를 만들게 된다. 2번 패턴은 1번 패턴과 비슷하게 최상위 sign bit가 달라지는 경우이다.

3번과 4번 패턴은 3.1절의 예제에서 설명한 경우이다. 3번은 정수 확장으로 인해 연산 결과가 제대로 표현되지 않는 경우이며 4번은 부호 차이로 인해 연산 결과의 부호가 정확하지 않게 되는 경우이다. 5번 패턴은 3항 연

표 1 정수 확장 버그 패턴
Table 1 Integer promotion bug patterns

No.	Syntactic condition		Semantic condition
	Operator type	Operand type	
1	~p	p is uchar and ~p is stored to an integral type variable	All values of p
2	-p	p is schar or uchar	Either (1) p==-128 where p is schar or (2) p!=0 where p is uchar
3	p+q, p*q, p-q, p<<q	p is schar or uchar, and q is schar or uchar	A result of an operation cannot be represented as schar or unsigned char
4	p/q, p%q, p|q, p^q, p==q, p!=q, p<q, p<=q, p>q, p>=q	Either (1) p is schar and q is uchar or (2) p is uchar and q is schar	Either (1) p is schar and p<0 or (2) q is schar and q<0
5	x ? p : q		

산자에서 서로 다른 부호를 가질 때 연산 결과의 부호값이 달라지는 경우이다. 예를 들어 p, q가 각각 schar, uchar 자료형이고 p의 값이 -1, q의 값이 1인 경우 C 표준에서는 p, q의 값을 int 자료형으로 확장하여 연산을 수행하나 8-bit MCU 컴파일러는 p의 값을 uchar로 간주하기 때문에 255가 되어 음수 값을 양수 값으로 해석하는 정수 확장 버그가 발생할 수 있다.

4. 정수 확장 버그 패턴 검출기 구현

본 장에서는 정적 분석을 사용하여 정수 확장 버그 패턴의 코드 구조 조건과 피연산자 값 조건을 탐지하는 구현 방법에 대해 설명한다.

4.1 코드 구조 조건 검출

코드 구조 매칭을 위해 Clang/LLVM 프론트엔드를 사용하여 대상 프로그램의 abstract syntax tree(AST)를 생성하고 AST를 순회하면서 구문 조건에 맞는 연산자 종류와 피연산자의 자료형을 갖는 프로그램 코드가 있는지 탐색하였다. 정수 확장 버그 패턴의 구문 조건은 정수 확장 버그가 발생할 수 있는 필요 조건이기 때문에 구문 조건을 만족하지 않는 프로그램 코드에서는 정수 확장 버그가 발생하지 않는다.

실제 정수 확장 버그가 발생하기 위해서는 정수 확장 버그 패턴의 구문 조건을 만족하는 프로그램 실행 구문에서 피연산자의 값 조건을 만족해야 한다. 코드 구조 조건만 검출해서 보고할 경우 다수의 거짓 경보가 발생할 수 있기 때문에 거짓 경보를 줄이기 위해 피연산자 값 조건을 만족하는지 확인하기 위한 분석 기법이 필요하다.

4.2 정적 범위 분석 기법을 사용한 피연산자 값 조건 검출

정수 확장 버그 패턴을 이용해 경보 검출 시 피연산자가 변수인 경우 피연산자의 값 조건을 알 수 없어 해당하는 모든 구문을 검출하고, 따라서 많은 거짓 경보가 발생하는 문제가 있다. 이 문제를 해결하기 위해 정적 범위 분석(static range analysis)[9] 기법을 적용한다.

정적 범위 분석 기법은 프로그램 코드의 데이터 흐름을 이용해 변수들이 각 코드의 위치에서 가질 수 있는 값의 범위를 정적으로 분석한 것이다. 본 논문에서 사용한 정적 범위 분석 기법은 건전성(soundness)이 증명되어 있기 때문에 정적 분석을 통해 나온 변수의 값 범위는 실제 프로그램 수행 시 해당 변수가 가질 수 있는 값의 범위보다 항상 크다. 정적 범위 분석 기법을 사용하면 컴파일 타임에 변수 값의 범위를 알 수 있으므로 정수 확장 버그 패턴의 구조 조건을 만족하는 프로그램 구문 이 피연산자 값 조건을 만족하는지 여부를 알 수 있다. 만약 정적 범위 분석 기법이 알려준 변수 값 범위가 정수 확장 버그 패턴의 피연산자 값 조건을 절대 만족할 수 없다면 해당 프로그램 구문은 정수 확장 버그가 발생하지 않기 때문에 경보에서 제거한다. 또한, 함수 간(inter procedural) 범위 분석 기법을 사용하기 때문에, 각 함수에서 사용하는 매개변수 값의 범위도 알 수 있다.

그림 2의 코드에 정적 범위 분석 기법을 적용하면 x와 y 변수가 사용되는 위치에서 그 값의 범위를 알 수 있다. 4번째 줄의 조건문에 의해 x 값의 범위는 [0, 9]가 되고, y 값의 범위는 [0, 19]임을 알 수 있다. 또한 x, y 값이 foo 함수의 매개 변수로 사용되면서, 7번째 줄의 x + y의 연산 결과 값 범위는 [0, 28]이 된다. 이 연산 결과 값은 255를 초과하지 않기 때문에 해당 버그

```
1 int main(){
2   unsigned char x = rand();
3   unsigned char y = rand();
4    if(x<10 && y<20) { foo(x, y);} }
5 void foo(unsigned char x,
6            unsigned char y){
7      int a = x + y; }
```

그림 2 정적 범위 분석 기법 적용을 위한 코드 예시
Fig. 2 An example code for static range analysis

패턴은 정수 확장 버그가 아니라고 판단하고, 이 거짓 경보를 검출하지 않는다.

5. 정수 확장 버그 패턴 검출기 거짓 경보 제거 기법

본 장에서는 동적 기호 실행 기반 자동화된 유닛 테스트 기법을 적용하여 정수 확장 버그 패턴의 피연산자 값 조건을 검출하고 거짓 경보를 줄이는 기법에 대해 설명한다.

5.1 동적 기호 실행 기법

동적 기호 실행 기법은 동적 분석과 정적 분석 기법을 결합하여 테스트 케이스를 자동으로 생성하는 기법이다. 테스트를 생성할 대상 프로그램과 초기 테스트 케이스가 주어지면 주어진 테스트 케이스를 실행하고 실행된 프로그램 경로를 분석하여 분기문에서 어떤 조건이 수행되었는지 나타내는 경로 제약 조건식을 생성한다. 테스트 실행이 종료되면 생성된 경로 제약 조건식을 분석하여 이전에 실행되지 않은 새로운 프로그램 경로를 실행할 수 있는 테스트 케이스를 생성한다. 모든 프로그램 실행 경로를 테스트하거나 사용자가 지정한 종료 조건을 만족하면 동적 기호 실행 기법이 종료된다.

그림 3의 예제 프로그램을 사용해 동적 기호 실행 기법의 실행 과정을 살펴보자. 그림 3 왼쪽의 예제 프로그램은 세 정수 x, y, z를 입력으로 받아 가장 큰 값을 돌려주는 함수이다. 그림 3의 오른쪽 그림은 프로그램 실행 경로를 나타내는 실행 경로 나무이다.

초기 입력 값이 x=0, y=0, z=0 으로 주어졌다면 3, 4번

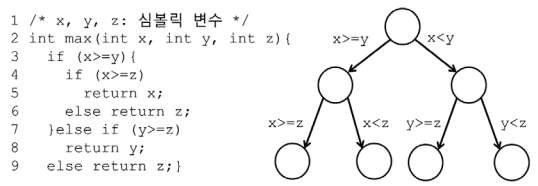


그림 3 3개의 분기를 갖는 예제 프로그램 및 실행 경로 ([10]에서 발췌)

Fig. 3 An example program that has 3 branches and its execution paths(taken from [10])

째 줄의 if 조건문이 만족하게 되고 예제 프로그램이 종료될 때 경로 제약 조건식 (x>=y)∧(x>=z)가 생성된다.

새로운 실행 경로를 실행하는 테스트 케이스를 생성하기 위해 마지막 경로 조건 x>=z 를 부정해서 새 경로 조건 (x>=y)∧(x<z) 를 생성하고 제약 조건 해결기(constraint solver)를 사용해서 경로 조건을 만족하는 테스트 케이스 x=0, y=0, z=1 을 생성한다. 이 과정을 동적 기호 실행이 종료될 때까지 반복하게 된다.

5.2 자동화된 유닛 테스팅 기법을 활용한 피연산자 값 조건 검출 기법

동적 기호 실행 기반 자동화된 유닛 테스팅 기법[11]은 대상 프로그램의 유닛 테스트 드라이버/스텁 함수를 자동으로 생성하고 해당 유닛 테스트의 입력 값을 동적 기호 실행 기법을 사용하여 자동으로 생성하는 기법이다. 대상 함수 f()의 모든 파라메터와 f()가 사용하는 모든 전역 변수를 동적 기호 실행의 심볼릭 입력 값으로 설정하고 f()가 호출하는 모든 함수를 심볼릭 값을 리턴하는 심볼릭 스텁 함수로 대체한 후 생성된 테스트 드라이버/스텁 함수에 동적 기호 실행 기법을 적용해 테스트 입력 값을 생성한다. 대상 함수 f()의 파라메터와 전역 변수의 자료형에 따라 표 2와 같이 심볼릭 입력 값을 설정한다. Primitive 자료형의 경우 해당 자료형에 맞는 심볼릭 입력 값으로 설정하고 배열의 경우 원소의 자료형에 맞는 심볼릭 입력 값을 모든 원소에 설정한다. 구조체의 경우 모든 구조체 필드에 대해 해당 필드의 자료형에 맞는 심볼릭 입력 값을 설정한다. 포인터의 경우 해당 포인터가 가리키는 자료형 크기만큼 메모리를 할당하고 가리키는 자료형에 따라 심볼릭 입력을 설정한다. 단 포인터의 경우 구조체 포인터 등으로 인해 무한히 심볼릭 입력을 설정할 수 있기 때문에 최대 1번만 포인터 참조를 수행하여 입력 값을 생성한다.

유닛 테스트 수행 과정에서 피연산자 값 조건이 만족하는지 확인하기 위해 구문 조건을 만족하는 프로그램 코드에 피연산자 값 조건을 만족하는지 검사하기 위한 단언문(assert())을 삽입하였다. 피연산자 값 조건을 만족하는 경우 단언문 위배 발생하여 정수 확장 버그 경보를 발생시킨다. 만약 구문 조건을 만족하는 프로그램 코드에 대해 단언문 위배가 발생하지 않는 경우 피연산

표 2 동적 기호 실행 입력 설정

Table 2 Symbolic input setting

Input type	Symbolic input setting
Primitive type	Specify a symbolic input to an input variable according to the input variable type
Array type	Specify symbolic inputs to all elements according to the element's type
Struct type	Specify symbolic inputs to all fields of the struct type according to each field's type
Pointer type	Allocates memory space whose size is equal to pointee type and specify a symbolic input according to the pointee type. If the pointee type is a struct or pointer type, the dereference chain is followed only once

자 값 조건을 만족할 수 없는 것으로 간주하여 경보를 발생시키지 않는다.

6. 사례 연구

사례 연구에서 사용한 대상은 LG전자에서 8-bit MCU를 사용하는 세탁기 프로그램이다. 대상 프로그램은 약 41KLOC의 크기에 약 1.4K 개의 함수를 가진 C 프로그램이다. 실험은 Intel Core i7-2600K 3.4GHz 프로세서와 32GB RAM이 장착되어 있는 Debian 8.2 64 bit OS가 설치된 워크 스테이션에서 수행하였다. Clang을 사용하여 유닛 테스트 드라이버 자동 생성 도구를 구현하였고 CREST-BV[10]를 사용하여 동적 기호 실행을 수행하여 테스트 케이스를 생성하였다. 동적 기호 실행에서 각 대상 함수 별 최대 실행 시간은 5분으로 제한하였으며 DFS 탐색 기법을 사용하였다.

표 3은 정수 확장 버그 검출기를 세탁기 소프트웨어에 적용한 결과이다. 정수 확장 버그 패턴을 소프트웨어에 적용하였을 때 1.55초의 수행 시간으로 100개의 경보가 검출되는 것을 확인하였다. 여기에 정적 범위 분석 기법을 추가 적용하여 24.58초의 수행 시간으로 미탐지 버그 없이 거짓 경보 73개 중 15개 (20.5%)의 거짓 경보를 제거하였다. 동적 기호 실행 기반 자동화된 유닛 테스팅 기법이 얼마나 거짓 경보를 제거하고 미탐지 버그를 야기하는지 확인하기 위해 LG전자 개발팀과 함께 경보 및 대상 프로그램을 분석하였다. 분석 결과 남은 85개의 경보 중 27개의 경보는 컴파일러 옵션에 따라 실제 발생할 수 있는 정수 확장 버그임을 확인하였다.

85개 경보가 발생한 61개 함수를 대상으로 동적 기호 실행 기반 자동화된 유닛 테스팅을 적용한 결과 632초 (2개 함수에서 시간 초과 발생)의 실행 시간이 소요되었으며 85개 경보 중 35개 경보가 제거되어 50개 경보만 보고되었다. 그 중 실제 버그는 23개로 4개 버그(14.8%)가 탐지되지 못했으며 거짓 경보는 27개로 정적 범위 분석 기법만 적용했을 때 발생한 58개 거짓 경보 중에서 53.4%가 제거되었다. 거짓 경보 비율도 정적 범위 분석 기법만 적용했을 때 68.2%에서 54.0%로 상대적으로 20.8% 더 감소되어 효과적으로 거짓 경보를 제거할 수 있음을 확인하였다. 비록 4개 버그를 탐지하지 못했으나 거짓 경보를 효과적으로 줄임으로써 개발 시간이 부족한 개발 현장에서 실용적으로 사용할 수 있는 기법임을 확인할 수 있었다.

표 4는 자동화된 유닛 테스트를 적용한 동적 분석 결과 발생한 경보와 개발자가 직접 분석한 경보 분석 결과를 비교한 것이다. 먼저 전체 27개의 정수 확장 버그 (4번째 열 6번째 줄) 중에서 동적 분석을 통해 단언문 위배를 발견한 버그는 총 23개(4번째 열 3번째 줄)이다. 동적 분석 결과 탐지하지 못한 버그 4개 가운데 2개는 단언문을 실행하지 못했고(4번째 열 5번째 줄) 2개는 단언문은 실행하였으나 단언문 위배가 발생하지 않았다 (4번째 열 4번째 줄). 4개 버그 공통적으로 단언문을 실행하지 못하거나 단언문 위배가 발생하는 입력 값을 생성하지 못한 이유는 CREST-BV가 심볼릭 배열 인덱스 구문을 지원하지 않기 때문이다. 발생한 거짓 경보 58(5번째 열 6번째 줄)개중에서 31개는 동적 분석 결과 제

표 3 정수 확장 버그 검출기를 통해 세탁기 소프트웨어에서 검출된 경보 분석
Table 3 The number of integer promotion bug alarms on LG washing machine control software

	Syntactic condition check	Semantic condition check	
		Static range analysis	Static range analysis + automated unit testing
True alarms	27	27 (-0%)	23 (-14.8%)
False alarms	73	58 (-20.5%)	27 (-63.0%)
False alarm ratio	73.0%	68.2%	54.0%
Total alarms	100	85 (-15.0%)	50 (-50.0%)

표 4 동적 분석 결과 발생한 경보와 개발자의 경보 분석 결과 비교
Table 4 Comparison of the number of alarms reported by dynamic analysis and developers' manual analysis

			Manual analysis result (ground truth)		
			True alarms	False alarms	Total alarms
Dynamic analysis result	Assert() is violated		23	27	50
	Assert() is not violated (False negative)	Assert() is executed	2	30	32
		Assert() is not executed	2	1	3
	Total		27	58	85

거되었고 27개의 거짓 경보가 발생하였다. 제거된 31개 거짓 경보를 분석한 결과 30개의 거짓 경보는 단언문을 실행했으나 단언문 위배가 실제 발생하지 않았고(5번째 열 4번째 줄), 나머지 1개의 거짓 경보는 단언문을 실행하지 못했다(5번째 열 5번째 줄).

7. 결론 및 향후 연구

본 논문에서는 8-bit MCU 컴파일러를 사용할 때 발생 할 수 있는 정수 확장 버그를 소개하고 정수 확장 버그를 검출하기 위한 다섯 종류의 정수 확장 버그 패턴 및 패턴 검사 기법을 제안하였다. 정수 확장 버그 패턴을 자동으로 검출할 수 있는 검출 도구를 개발하였으며 거짓 경보를 줄여 정확성을 높이기 위해 정적 범위 분석 기법과 동적 기호 실행 기법 기반 자동화된 유닛 테스팅 기법을 적용하였다. 제안하는 기법 및 개발 도구가 실제 정수 확장 버그를 효과적으로 검출할 수 있는지 확인하기 위해 LG전자 세탁기 제어 소프트웨어를 대상으로 사례 연구를 수행하여 27개의 정수 확장 버그를 검출할 수 있었다. LG전자 개발팀에서는 본 연구 결과 심각한 버그를 찾을 수 있었다고 평가하였으며 정수 확장 버그 검출 도구 수행을 개발 프로세스 상에 추가하여 실제 개발 과정에서 본 연구 결과를 활용하고 있다.

향후 연구로는 추가적으로 거짓 경보를 제거할 수 있는 거짓 경보 제거 기법에 대해 연구하고 정수 확장 버그 외에 8-bit MCU 컴파일러를 사용할 때 발생할 수 있는 버그 종류가 더 있는지 LG전자와 같이 연구할 예정이다.

References

[1] C. Elbert and C. Jones, "Embedded software: facts, figures, and future," *IEEE Computer*, Vol. 42, No. 4, pp. 42-52, Apr. 2009.

[2] I. Fredriksen, "Choosing a MCU for your next design; 8 bit or 32 bit?," Atmel Corp., 2014.

[3] Clang/LLVM. [Online]. Available: http://llvm.org.

[4] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song, "RICH: Automatically protecting against integer-based vulnerabilities," *Proc. of the Symposium on Network and Distributed Systems Security*, 2007.

[5] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," *ACM Transactions on Software Engineering and Methodology*, Vol. 25, No. 1, pp. 2:1-2:20, 2015.

[6] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "BRICK: A binary tool for run-time detecting and locating integer-based vulnerability," *Proc. of the International Conference on Availability, Reliability and Security*, pp. 208-215, 2009.

[7] D. Molnar, X. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," *Proc. of the USENIX Security Symposium*, pp. 67-82, 2009.

[8] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Proc. of the Workshop on Runtime Verification*, 2003.

[9] R. Rodrigues, V. Campos, and F. Pereira, "A fast and low-overhead technique to secure programs against integer overflows," *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 1-11, 2013.

[10] Y. Kim, M. Kim, and Y. Jang, "CREST-BV: An improved concolic testing technique supporting bit-wise operations for embedded software," *Journal of KIISE: Software and Applications*, Vol. 40, No. 2, pp. 90-98, 2013, (in Korean)

[11] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim, "Automated unit testing of large industrial embedded software using concolic testing," *Proc. of the IEEE/ACM Automated Software Engineering Experience track*, pp. 519-528, Nov. 2013.

김 윤 호

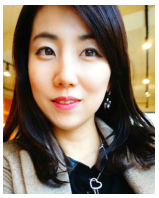
2007년 KAIST 전산학과 학사. 2007년~현재 KAIST 전산학부 석/박사 통합과정 재학. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅, 자동 오류 위치 추정, 정형검증

김 태 진

2014년 중앙대학교 컴퓨터공학부 학사 2014년~현재 KAIST 전산학부 석사과정 재학. 관심분야는 Concolic 테스팅, 소프트웨어 정적 분석

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 전산학부 부교수. 관심분야는 Concolic 테스팅, 자동 오류 위치 추정, Concurrency 테스팅, 변이 테스팅, 정형검증, 내장형 소프트웨어



이 호 정

2010년 경북대학교 전자전기컴퓨터학부 학사. 2011년~2013년 LG전자 H&A 사업본부 세탁기제어1팀(SW). 2014년~현재 LG전자 H&A HA 선행제어연구1팀 주임연구원. 관심분야는 SW Testing, Home-Network

장 훈

2008년 경북대학교 전자전기컴퓨터학부 학사. 2010년 경북대학교 전자전기컴퓨터대학원 석사. 2010년~2015년 LG전자 H&A 사업부 주임연구원. 2015년~현재 현대자동차 샤시기술센터 연구원. 관심분야는 정형검증, 소프트웨어 테스팅

박 민 규

2011년 경북대학교 컴퓨터학부 학사. 2013년 경북대학교 전자전기컴퓨터학부 석사 2015년~현재 LG전자 제어연구소 주임연구원. 관심분야는 컴포넌트 기반 개발 방법, 소스 코드 대상 정형 검증