DOI: 10.3745/KIPSTA.2011.18A.2.061

PinMemcheck: 이동통신 기기 개발을 위한 Pin 기반의 메모리 오류 검출 道具

조 경 진† · 김 선 욱††

요 약

메모리 관련 오류 검출은 소프트웨어 개발 시 신뢰성 향상을 위해서 수행하여야 할 가장 중요한 작업중의 하나이다. 그러나 메모리 오류 검출을 위한 긴 디버깅 시간은 이동 통신 기기 개발 과정에 있어 큰 문제가 되었다. 대부분의 메모리 오류 검출 도구는 정적 분석 기법을 사용하나, 큰 용량의 동작 메모리로 인하여 이동 통신 기기 개발에는 사용되지 못하는 경우가 많다. 때문에 이동통신 기기 업체는 고품질의 기기를 빠른 시간 내에 개발하는 것이 매우 어려웠다. 이 논문에서 소개될 이동통신 기기 개발을 위한 Pin 기반의 메모리 오류 검출 도구인 PinMemcheck은 Pin의 이진 가공 기법과 간단한 데이터 구조를 적용하여 기준 설정 대비 약 1.5배의 실행 시간 부하 내에서 필수 오류들을 모두 검출해 내었다.

키워드 : 메모리 오류 검출 도구, 메모리 누출

PinMemcheck: Pin-Based Memory Leakage Detection Tool for Mobile Device Development

Kyongjin Jo† · Seon Wook Kim††

ABSTRACT

Memory error debugging is one of the most critical processes in improving software quality. However, due to the extensive time consumed to debug, the enhancement often leads to a huge bottle neck in the development process of mobile devices. Most of the existing memory error detection tools are based on static error detection; however, the tools cannot be used in mobile devices due to their use of large working memory. Therefore, it is challenging for mobile device vendors to deliver high quality mobile devices to the market in time. In this paper, we introduce "PinMemcheck", a pin-based memory error detection tool, which detects all potential memory errors within 1.5x execution time overhead compared with that of a baseline configuration by applying the Pin's binary instrumentation process and a simple data structure.

Keywords : Memory Error Detection Tool, Memory Leakage

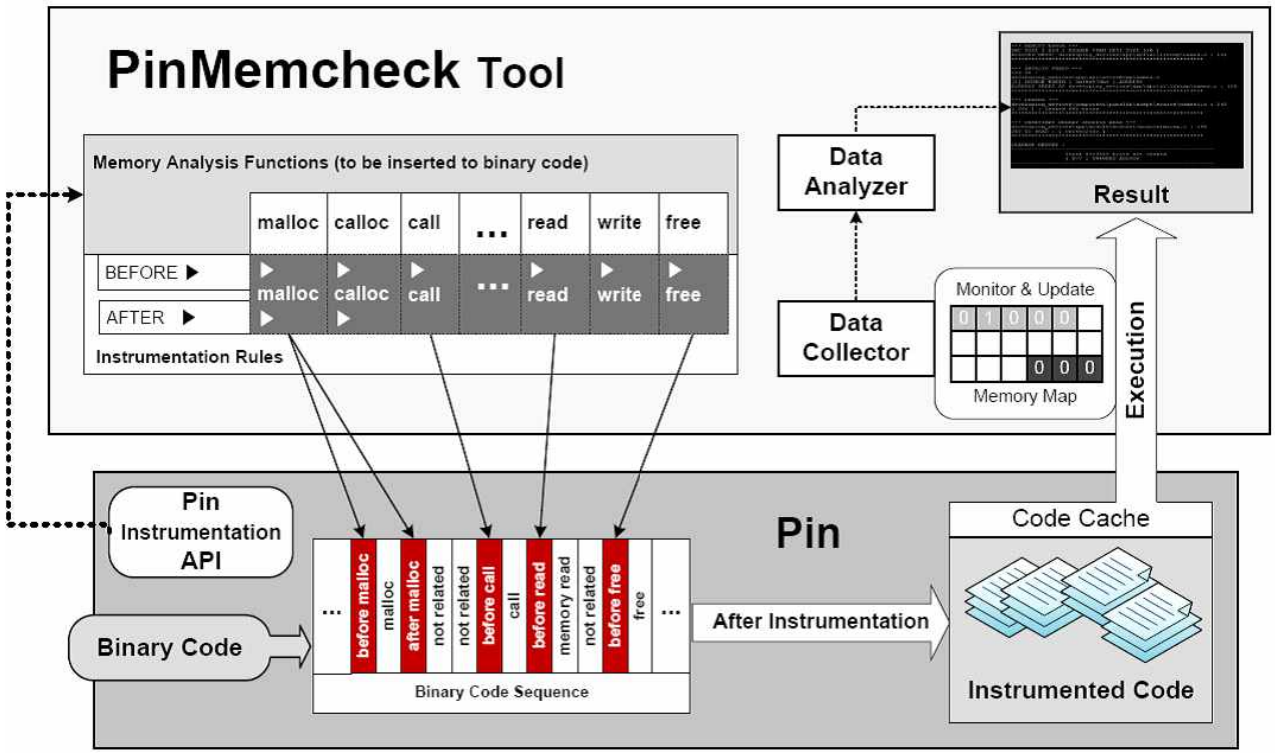
1. 서 론

전력, 가용 면적, 메모리 용량과 같은 한정된 자원을 얼마나 효율적으로 활용하느냐는 이동통신 기기 개발에 있어서 가장 큰 숙제 중 하나이다. 전력의 부족은 기기의 계산 능력과 사용시간을 저하시키며, 기기 가용 면적의 감소는 탑재할 수 있는 기능의 숫자를 제한한다.

메모리 문제는 이보다 더 심각하다. 적은 메모리를 가진 시스템은 메모리 누출로 인하여 대용량 메모리 시스템에 비해 시스템의 오 동작을 일으킬 확률이 더 높다. 빠른 시간 내에 시장에 출시해야만 하는 이동통신 기기의 특성으로 인하여 개발 업체는 짧은 개발 시간 내에 모든 메모리 누출을 확인하기가 어렵고 결국 소프트웨어의 품질이 매우 나빠지게 된다. 메모리 누출은 초기화 되지 않은 변수 사용, 유효하지 않은 메모리 영역 접근 등과 같은 메모리 동작 오류로 인해 발생한다. 따라서, 이를 방지하기 위한 메모리 관리[1][2][3]는 개발 업체가 무엇보다 신경 써야 할 부분이다.

다양한 메모리 오류 검출 도구들이 언급한 메모리 관리 작업을 위해 사용되고 있으나 이동통신 기기 개발, 특히 피쳐 폰 (feature phone)개발 과정에서 사용할 만한 메모리 오류 검출 도구는 제한적이다. 그 이유는 크게 2가지로 나뉜다. 첫째, 대부분 윈도우 운영 체제 하에서 이루어지는[10]

† 준 회 원 : 고려대학교 전기전자전파공학과 박사과정
†† 종신회원 : 고려대학교 전기전자전파공학과 교수
논문접수 : 2010년 12월 24일
수 정 일 : 1차 2011년 3월 29일
심사완료 : 2011년 3월 29일



(그림 1) PinMemcheck 전체 구조

피쳐 폰 개발 과정의 특성 상 윈도우에서 동작 가능한 메모리 오류 검출 도구를 사용해야 하나 가능한 도구를 찾기 어렵다[7][8].

둘째, 각 피쳐 폰 개발사는 직접 메모리 관리 함수를 작성하여 사용하는데 상용으로 제공되는 대표적인 윈도우 기반 메모리 오류 검출 도구[9]는 수정이 불가능 하여 개발사가 직접 작성한 메모리 관리 함수를 사용할 수 없기 때문이다.

이러한 문제를 해결하기 위하여 본 논문에서는 Pin[11] 이진 가공 엔진을 활용하여 개발된 동적 메모리 오류 검출 도구 *PinMemcheck*을 소개한다. PinMemcheck는 Pin을 통해 메모리 오류 검출을 위해 우리가 직접 작성한 메모리 오류 검출 라이브러리를 이진 실행 파일에 삽입하여 해당 실행 파일의 실행 과정에서 동적 메모리 오류를 검출해 낼 수 있도록 한다.

PinMemcheck는 다음과 같은 부분에서 산업적, 학술적 가치를 가진다. 첫째, PinMemcheck는 피쳐 폰 개발 과정에서 사용되는 개발사가 직접 작성한 메모리 관리 함수와 Visual Studio Emulator를 지원한다. 개발사는 메모리 관리 함수의 이름 및 메모리 관리 동작을 명시하는 과정을 통해 다양한 형태의 메모리 관리 함수를 PinMemcheck을 통해 감시할 수 있다. 둘째, Pin 을 사용함으로써 PinMemcheck는 모든 메모리 오류를 검출하는 과정에서 이진 코드 가공의 부하를 크게 줄였다. 실제 개발 과정에 있는 피쳐 폰 플랫폼 및 탑재된 응용 프로그램을 대상으로 실험한 결과 PinMemcheck는 Pin 이진 가공 코드만 실행하는 기본 실행 시간에 비하여 약 50% 가량의 실행 시간만이 증가되는 것을 확인하였다. 이는 Linux 기반의 오픈 소스 메모리 오류 검출 도구[7][8]에 비해 약 1/3의 부하에 불과하다.

본 논문의 구성은 다음과 같다. 2장에서는 관련된 연구들에 대해 간략히 소개하고, 3장의 내용을 통해 PinMemcheck의 구조 및 기능을 설명한다. 4장은 성능 및 부하에 대한 분석 결과를 제시하고 5장의 결론으로 마무리 짓도록 한다.

2. 관련 연구

메모리 오류는 정적 오류와 동적 오류로 나뉜다. 정적 오류는 원본 코드 분석을 통해 빠르고 쉽게 검출해 낼 수 있다. Splint[4]와 같은 도구가 정적 오류 분석 및 검출에 사용되고 있다. 그러나 많은 오류들이 실행 환경에 따라 다른 메모리 동작을[6] 나타내므로 정적 오류 검출 도구로는 검출에 한계가 있어 일반적으로 동적 메모리 오류 검출 도구가 널리 사용된다.

동적 메모리 오류 검출 도구는 원본 코드 개발 과정에서 가공하는 방식과 실행 시간에 이진 코드를 가공하는 방식으로 나뉜다. 전자의 방식 (예, Insure++[5])은 개발자의 업무량을 증가시키는데다 원본 코드 없이는 오류 검출이 불가능하다는 단점을 가진다. 이에 반해 이진 코드 가공 방식은 원본 코드 없이 동적 메모리 오류를 검출할 수 있어 널리 활용되고 있다. 일반적으로 DBI(Dynamic Binary Instrumentation, 동적 이진 가공)라 불리는 대표적인 동적 이진 가공 엔진 및 도구는 다음과 같다.

Valgrind[7][8] 유명한 오픈 소스 이진 가공 엔진으로써, 메모리 오류 검출 (memcheck)도구, 캐시 메모리 동작 분석 (cachegrind)도구 외에 다양한 기능을 가진 도구를 부가적으로 제공한다. Valgrind는 x86, AMD64, PPC32, PPC64 / 리눅스 조합 및 x86, AMD64/MacOS X 조합을 지원하며 C/C++ 로 작성된 코드에 한하여 동작한다.

DynamoRIO Valgrind와 같은 오픈 소스 가공 엔진으로 Hewlett-Packard 연구소와 MIT의 합작 하에 개발되고 있다. DynamoRIO는 Valgrind 에 상응하는 기능을 제공하며 윈도우즈와 리눅스 환경에서C/C++/Java/C# 언어로 작성된 코드를 가공할 수 있다.

Rational Purify[9] IBM에서 제공하는 동적 메모리 오류 검출 도구로 윈도우즈, 리눅스, 솔라리스, AIX 와 같이 다양한 운영체제 및 UltraSPARC[15], Intel64[16], IA-32[17] and IBM POWER5/6[18][19] 아키텍쳐를 지원한다. 또한 Valgrind와 달리C/C++외에Java/VS.NET으로 작성된 코드의 메모리 오류도 분석 가능하다.

Pin Alpha 시스템에서 유래한ATOM[20]을 Intel[17]이 인수하여 개발하고 있는 동적 이진 가공 엔진이다. Pin은 비공개 엔진과 수십여 가지의 공개 도구 및 API 형태로 배포된다. Pin은 x86, Itanium[21], IA-32, Xscale[22], ARM[23] 아키텍쳐와 윈도우즈 및 리눅스 조합을 지원하며 C/C++로 작성된 이진 코드 가공이 가능하다. 이진 가공 단계에서 인라이닝 및 명령어 스케쥴링 등의 최적화 기법을 적용하여[8] Valgrind나 DynamoRIO에 비해 월등한 성능을 보인다. 실험 결과에 따르면 Pin의 성능 (이진 코드 가공 및 실행 시간)은 Valgrind에 비해 약 6배, DynamoRIO에 비해 약 2배 정도 빠른 것으로 나타났다[13][14].

3. PinMemcheck 의 구조

3.1 구조 개략

PinMemcheck는 (그림 1)과 같이 Pin[11] 을 통하여 이진 코드를 가공하는 부분(그림의 아랫부분)과 메모리 동작 정보를 수집하고 분석하는 부분(그림의 윗부분)으로 나뉜다.

이진 코드 가공 파트는 메모리 동작에 영향을 주는 동작 및 함수인 malloc, calloc, 함수 호출, 메모리 읽기, 메모리 쓰기 등을 찾아 PinMemcheck내에서 정의한 메모리 동작 분석 함수를 삽입한다. 각각의 메모리 관련 동작에 따라 다른 가공 규정을 적용하여 코드를 가공한다. 가공 규정은 메모리 동작 이전에 삽입, 메모리 동작 이후에 삽입, 메모리 동작 전후로 삽입의 3가지로 구분된다. 각각의 규정은 (그림 1)에서 흑/백의 삼각형으로 표현되어있는데, 메모리 동작 위의 삼각형은 메모리 동작 이전에 삽입 하는 규정을 뜻하며 아래의 삼각형은 이후에 삽입하는 규정을 뜻한다. 가공 규정이 각기 다른 이유는 가공 규정마다 수집할 수 있는 정보가 다르기 때문이다. 일반적으로 메모리 동작 이전 삽입의 경우, 함수 인자를 수집할 수 있으며 (예, malloc 에서 메모리 할당 크기) 메모리 동작 이후 삽입의 경우 함수의 반환 값 (malloc 에서 할당된 메모리의 포인터)을 수집할 수 있다.

위에 설명한 방법대로 코드 가공 과정이 마무리 되면, 가공된 이진 코드는 코드 캐시 (code cache)라 불리는 코드 저장 영역에 저장되어 실행된다. 이 과정에서 삽입되었던 메모리 동작 분석 함수는 가상 메모리 맵 (3.3 장에서 설명)을 갱신하며 메모리 동작을 기록하고 수집한다. 이렇게 수집된 메모리 동작 정보는 데이터 분석기를 통해 분석되고 그 결과를 사용자에게 제공하여 메모리 오류를 확인토록 한다.

3.2 이진 가공 세부 내용

PinMemcheck는 전체 코드 가공과 선택적 코드 가공을 선택하여 이진 코드를 가공할 수 있다. (그림 2)에서 의사 코드를 통해 두 코드 가공 방법을 나타내었다. Image0는 전체 코드 가공 기법의 최상위 함수이며 Image1은 선택적 코드 가공 기법의 최상위 함수이다.

```
0: /* pseudo code for a fine grained instrumentation */
1: IMG Image0(IMG img, void *v)
2: {
3:    While(end of instruction stream) /* call stack */
4:      INS_InsertCall(ins, BEFORE,  GetCallStack );
5: }
6:
7: /* pseudo code for a selective instrumentation */
8: IMG Image1(IMG img, void *v){   /* malloc */
9:      RTN rtn = RTN_FindByName(img, "malloc")
10:     RTN_InsertCall(rtn, BOTH, malloc_analysis,…);
11: }
12:
13: int main(int argc, char** argv){
14:     IMG_AddInstrumentFunction(Image0, 0);
15:     IMG_AddInstrumentFunction(Image1, 0);
16: }
```

(그림 2) PinMemcheck 이진 가공에 쓰이는 의사 코드

전체 코드 가공 기법은 프로그램 실행 영역의 모든 명령어를 가공하는 방법이다. 예를 들어, 함수 호출 정보를 확인하기 위해서는 프로그램의 모든 명령어에 대하여 함수 호출/반환을 확인할 수 있는 분석 함수를 삽입하여야 하므로 전체 코드 가공 기법이 쓰인다.

따라서 (그림 2)의 4번째 줄에 표시된 바와 같이, 함수 호출 정보 확인을 위해서는 Image0안에 있는 모든 명령어의 앞 (BEFORE)에 GetCallStack이라 명시된 함수 호출 정보 수집 함수를 삽입하게 된다.

반면 선택적 코드 가공 기법은 프로그램을 구성하고 있는 요소들인 섹션, 트레이스, 함수, 베이직 블록내의 명령어만 가공하는 방식이다. 일반적으로 앞서 설명된 프로그램 구성 요소의 시작이나 끝에 메모리 분석 함수를 삽입하여 필요한 정보를 수집한다. 이는 3.1장에서 언급했던 malloc함수를 가공하는 경우에 사용되는데 (그림 2)의 9번째 줄에 표시된 바와 같이 malloc 앞뒤 (BOTH)로 malloc_analysis 분석 함수를 삽입하여 함수 인자와 반환 값을 얻어오게 된다.

이진 코드 가공 기법을 적용하기 위해서는 Pin 에서 제공한 API 사용 방법에 따라 14~15번째 줄에 표시된 것처럼 메인 함수 내에 IMG_AddInstrumentFunction을 호출하여 이진 코드 가공 함수인 Image0와 Image1을 각각 등록해야 한다.

3.3 PinMemcheck의 데이터 구조

메모리 오류 검출을 위한 메모리 관련 동작들 (읽기, 쓰기, 할당, 해제) 을 관찰하기 위하여 PinMemcheck는 각각의 메모리 주소에 연결되는 데이터 구조를 사용한다. Memory status map (MSM)이라 불리는 이 데이터 구조는 (그림 3)에 나온 바와 같이 메모리 주소를 키로, 메모리 정보를 멤버로 하는 해시 맵 (hash map)형태로 제작 되었다. Memory status map의 구조와 역할은 다음과 같다.

첫째, 해당 메모리 주소를 시작으로 할당된 메모리 영역의 끝을 나타내는 end_address를 제공하여 할당 크기를 알 수 있도록 하였다.

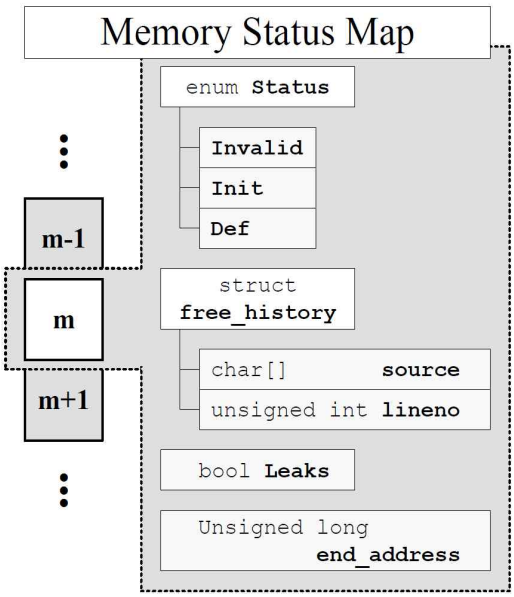
둘째, 메모리 해제에 관한 기록 (free_history)을 제공하여 메모리 해제 오류가 발생할 경우 디버깅을 쉽게 할 수 있게 하였다. 직전에 메모리 해제 동작이 일어난 소스 파일 이름과 줄 번호가 제공된다.

셋째, Leaks라 불리는 플래그를 두어 해당 메모리 주소가 누출 되었는지 여부를 프로그램 종료 시에 판단 할 수 있도록 하였다.

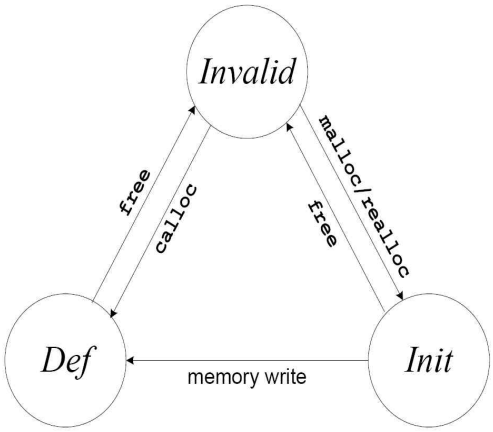
마지막으로 Status라는 멤버를 사용하여 가장 기본적인 메모리 상태를 체크할 수 있도록 하였다. 이에 사용되는 메모리 상태는 'Invalid', 'Init', 'Def' 3가지로 구성된다. 각각의 메모리 상태는 (그림 4)의 상태 전이도에 나온 바와 같이 메모리 동작에 따라 변화한다.

- *Invalid* : 할당되지 않은 메모리 주소 (할당 없이는 어떠한 접근도 불가능)
- *Init* : 할당 되었으나 어떠한 데이터도 기록되지 않은 메모리 주소 (NULL로 채워져 있어 쓰기만 가능)
- *Def* : 할당 및 데이터 기록까지 완료된 메모리 주소 (읽기 쓰기 모두 가능)

메모리 변수가 포인터로 선언되면 Invalid 상태가 되고, 그렇지 않은 경우는 Init 상태가 된다. Invalid 상태에서 메모리 할당 동작인 malloc/realloc가 발생하면 상태는 쓰기가 가능한 *Init* 상태로 바뀐다. 만약 할당 동작이 calloc일 경우 할당과 동시에 읽기 가능한 데이터 (모두 0으로)가 채워지므로 읽기/쓰기가 모두 가능한 *Def* 상태가 된다. 또한 *Init* 상태에서 해당 메모리에 대한 쓰기가 완료되어 읽기 동작이 가능해 지면 마찬가지로 *Def* 상태가 된다. 메모리 해제 동작인 free 가 일어나면 *Def*/*Init* 상태에서 모두 *Invalid* 상태로 변화하여 접근이 불가능해 진다. 간단히 요약하면 *Invalid* 상태에서는 메모리 할당 동작, *Init* 상태에서는 메모리 쓰기와 해제 동작만이 가능하며 *Def* 상태에서만 모든 메모리 접근이 허용된다.



(그림 3) PinMemcheck 데이터 구조



(그림 4) 메모리 맵의 상태 전이도

3.4 PinMemcheck에 사용된 오류 검출 기법

PinMemcheck에서 메모리 오류가 검출되는 과정을 설명하기 위해 (그림 5)에 필수적으로 검출되어야 하는 메모리 오류인 할당되지 않은 메모리 접근, 정의되지 않은 메모리 값 사용, 잘못된 메모리 해제, 유효하지 않은 memcpy, 메모리 누출 총 5 가지의 오류가 포함된 예제 코드를 표시하였다.

또한, (그림 6)에서 (그림 5)의 코드에 의해 PinMemcheck의 데이터 구조 변화를 확인할 수 있다. 데이터 구조 내 각각의 멤버는 앞글자를 따 S(Status), F(freeing history), L(Leaks), E(end_address)로 표시하였으며, 데이터 구조가 변화한 줄번호 (회색 박스, Line xx)와 메모리 오류를 발생시키는 줄번호 (검은 박스, Line xx)를 각각 표시하였다.

먼저, 모든 MSM의 초기 상태는 (그림 6)에서 initial로 표시된 부분과 같이 S(Invalid), F(NULL), L(FALSE), E(0)이다. 3번째 줄에서 정의된 변수 cp는 Init 상태로 변하고

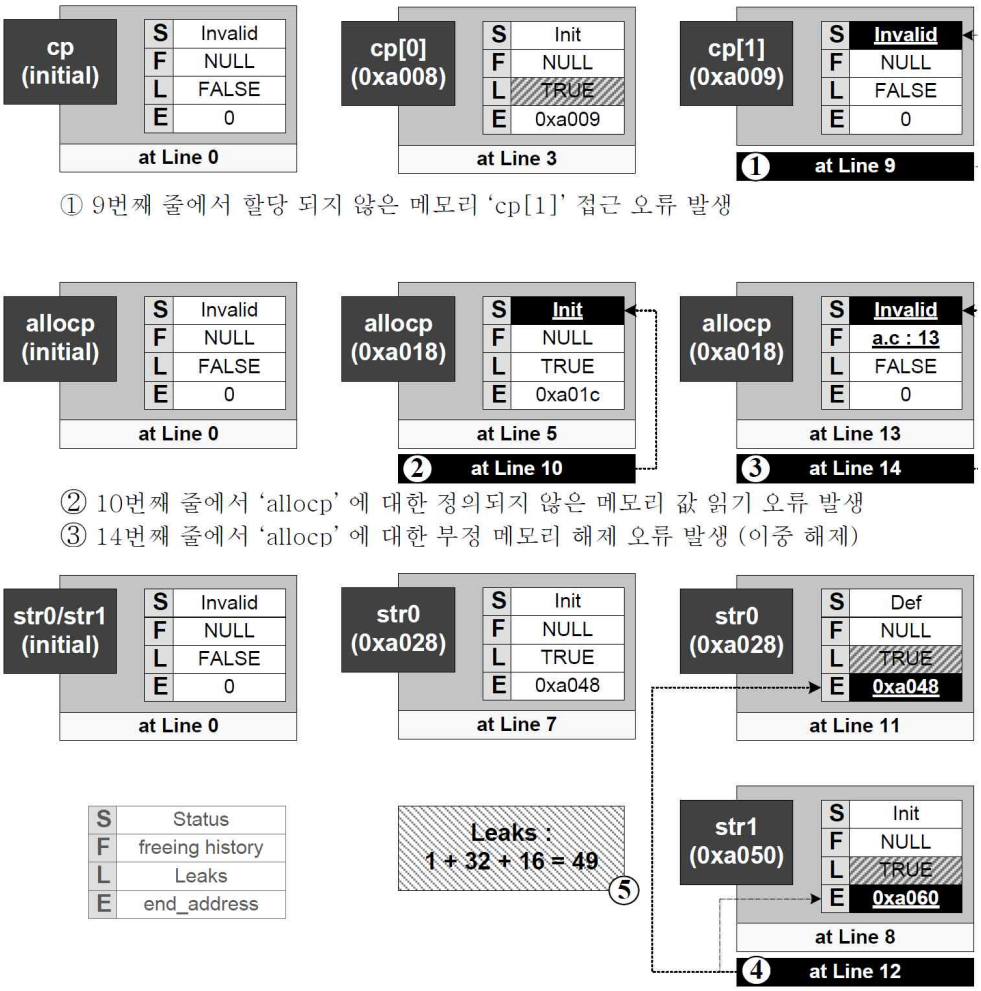
```
        /* source file : a.c */
0:      /* sample code of memory errors */
1:      int main(int argc, char** argv){
2:        char c;
3:        char *cp = (char*)malloc(1);
4:        int tmp;
5:        int *allocp = (int*)malloc(sizeof(int));
6:        char buf[32] = "here is an example";
7:        char* str0 = (char*)malloc(32);
8:        char* str1 = (char*)malloc(16);
9:        c = cp[1];          // 할당되지 않은 메모리 사용
10:       tmp = allocp;       // 정의되지 않은 값 사용
11:       strcpy(str0, buf);
12:       memcpy(str1, str0, 32); //잠재적 memcpy 오류
13:       free(allocp);
14:       free(allocp);  // 이중 해제
15:       return 0;      // cp[0], str0, str1 메모리 누출
16:     }
```

(그림 5) 메모리 오류가 포함된 예제 코드

현 상태에서 해제되지 않았으므로 Leaks가 TRUE가 된다. 또한 1바이트의 할당이 이루어졌으므로 end_address에 1 바이트가 증가한 주소를 담게 된다. 이 정보를 바탕으로 cp의 메모리 주소는 메모리 쓰기와 해제가 가능하고 누출 가능성이 있으며 해당 위치로부터 1바이트 할당이 이루어진 상태임을 알 수 있다. 9번째 줄에서 cp[1]을 접근하게 되는데 이 주소는 할당이 이루어지지 않아 Invalid 상태이므로 (그림 6). ①에 나온 바와 같이 할당되지 않은 메모리 영역에 대한 접근 오류가 발생한다.

둘째, 5번째 줄에서 정의된 포인터 변수 allocp의 경우 10번째 줄에서 읽기 동작이 일어나는데 읽기 동작이 불가능한 Init 상태이므로 (그림 6). ②의 정의되지 않은 메모리 값 사용에 관한 오류가 발생한다.

셋째, 13번째 라인에서 allocp는 해제되어 상태가 다시 Invalid로 돌아가고 Leaks 값도 FALSE가 되며 end_address도 0으로 초기화 된다. 그리고 메모리 해제 동작에



(그림 6) 메모리 동작에 따른 데이터 구조 변화

따라 freeing history가 갱신된다(a.c의 13번째 줄에서 해제). 따라서, (그림 6). ③에서 나타난 바와 같이 14번째 줄에서 발생한 메모리 해제 동작은 Invalid 상태의 메모리에 대해 이루어졌으므로 오류이며 freeing history의 값에 따라서 메모리 해제가 일어난 직전 위치를 확인할 수 있다.

넷째, 변수 str0와 str1은 7, 8번째 줄에서 선언되고 str0의 경우 11번째 줄에서 정의된다(그림 6의 하단). 12번째 줄에서 발생한 memcpy는 프로그램의 안정성에 영향을 미치지 않을 가능성도 있으나 16 바이트가 할당된 str1의 메모리 영역에 32바이트를 복사하게 되어 잠재적인 위험 요소가 된다. 따라서 (그림 6). ④에 나온 바와 같이 잠재적인 memcpy오류를 발생시킨다.

마지막으로, 프로그램이 종료되는 시점에서 cp[0], str0, str1의 Leaks가 TRUE이므로 누출된 메모리를 모으면(그림에서는 메모리 누출이 일어난 변수의 첫 번째 주소만 표시되었으나 실제 49개의 MSM에서 Leaks가 TRUE인 상태) (그림 6). ⑤에 표시된 바와 같이 49 바이트가 누출되었음을 확인할 수 있다.

4. 성능 평가

4.1 실험 환경

실험환경은 아래 <표 1>과 같다. Core 2 Duo, 2GB 메모리, Windows 7 운영 체제를 가진 시스템에서 Microsoft Visual Studio 2008 에뮬레이터를 이용 LG 전자에서 개발중인 피쳐폰의 플랫폼 및 응용 프로그램에서 메모리 오류를 검출하게 하였다.

〈표 1〉 실험 환경

CPU	Core 2 Duo 2.26GHz
RAM	2GB DDR2
L3 Cache	3 MB
Benchmarks	LG platform
S/W environment	Windows 7 / Visual Studio 2008 / Visual Studio 2008 Emulator

4.2 메모리 오류 검출 결과

(그림 5)의 예제 코드를 PinMemcheck를 통해 실제 검증한 결과를 (그림 7)에 표시 하였다. 3.4장에서 검증한 내용대로 5개의 메모리 오류가 검출되었음을 확인할 수 있다. 각각의 메모리 오류는 오류가 발생한 원인, 발생 위치 (소스파일 및 줄 번호) 및 호출 스택과 더불어 제공된다. 따라서 사용자는 단순한 오류의 원인뿐 아니라 프로그램의 흐름 중 어디에서 오류가 발생했는지에 대한 정보도 얻을 수 있다.

4.3 부하 분석 및 토의

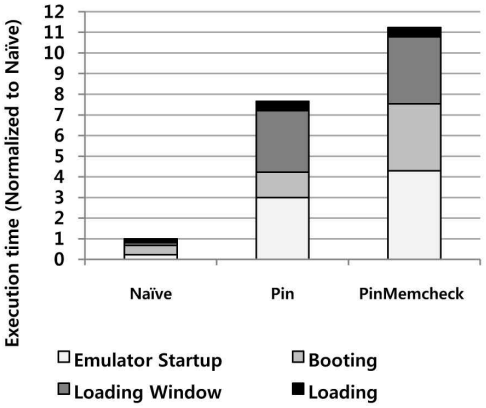
이동통신 기기 개발과정에서 사용되기 위해서는 PinMemcheck의 부하를 측정하여 납득 가능한 수준의 부하인지를 확인할 필요가 있다. 우리는 실제 개발 중인 이동

```
*** UNALLOCATED MEMORY READ ***
c:\Pin_Memcheck/test/a.c : 9 / main()
TRY TO READ : [0x805a009]
*********************************************************

*** UNDEFINED MEMORY READ ***
c:\Pin_Memcheck\test\a.c : 10 / main()
TRY TO READ : [ 0x805a018 ]
*********************************************************

*** MEMCPY ERROR ***
SRC SIZE [ 32 ] BIGGER THAN DEST SIZE [ 16 ]
ALLOCED DEST: c:\Pin_Memcheck\test\a.c : 8 / main()
*********************************************************

*** INVALID FREED ***
c:\Pin_Memcheck\test\a.c : 14
[1] DOUBLE FREED [ 0x805a018 ] ADDRESS
ALREADY FREED AT c:\Pin_Memcheck\test\a.c : 13 / main()
*********************************************************

*** LEAKED ***
c:\Pin_Memcheck\test\a.c : 3 / main()
[1] : leaked 1 byte(s)
c:\Pin_Memcheck\test\a.c : 7 / main()
[1] : Leaked 32 byte(s)
c:\Pin_Memcheck\test\a.c : 8 / main()
[2] : Leaked 16 byte(s)
*********************************************************

LEAKAGE REPORT :
---------------------------------------------------------
                Total 49 bytes are leaked
                [ 3 ] UNFREED ALLOCs
---------------------------------------------------------
```

(그림 7) 메모리 오류 검출 결과

통신 기기 플랫폼과 응용 프로그램에 PinMemcheck를 적용하여 그 부하를 측정하였다. (그림 8)에 나온 Naïve의 경우 원본 프로그램의 실행을 의미하며 Pin은 가공 코드를 포함하지 않은 이진 변환 과정의 성능을 뜻한다. 마지막으로 PinMemcheck는 메모리 분석 함수를 포함한 이진 변환 과정의 성능이다.

그림에서는 전체 실행 시간을 emulator startup (전화기 emulator 초기화 시간), booting (emulator 부팅 시간), loading window (응용 프로그램 로딩 윈도우 호출 시간), loading (응용 프로그램 실행 시간) 의 네 가지로 나누어 세부적인 분석이 가능하게 하였고 각각의 실행시간은 실제 에뮬레이터 실행 시간을 1로 표준화 (Naïve 가 1) 하였다.

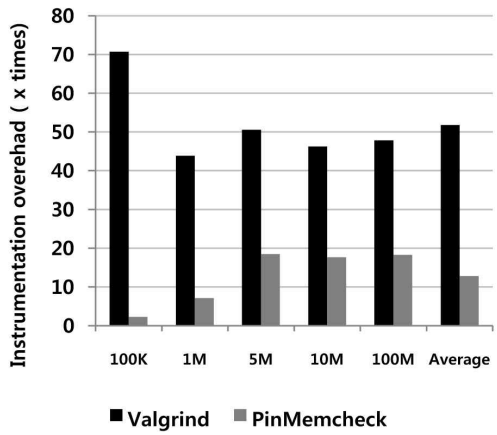


(그림 8) 실행 시간 비교

Pin을 통한 이진 코드 변환의 경우 Pin 이진 가공 엔진 내부에서 가상 머신 등을 거치며 발생하는 부하로 인해 원본 프로그램에 비해 7배 정도 느려짐을 확인할 수 있다. 이 데이터를 기준으로 하였을 때 PinMemcheck는 약 1.5배 정도의 성능 저하를, 원본 프로그램에 비해 11배 정도의 성능 저하를 보인다.

그러나 emulator startup/booting 시간을 제외한 loading window/loading에서의 소요 시간은 Pin 이진 코드 변환과 크게 차이가 없음을 알 수 있다. 따라서 일단 PinMemcheck를 통한 초기화 (emulator booting)가 완료되면, Pin 이진 코드 변환에 비해 큰 체감 성능의 차이를 보이지 않는다. 이는 실제 현장에서 충분히 사용 가능한 부하이며, 만약 초기화 과정에 대한 최적화가 더 이루어 진다면 전체적으로 Pin 이진 변환과 비슷한 성능을 보일 수 있다.

PinMemcheck의 상대적인 성능을 알아 보기 위해 (그림 9)에서 Valgrind memcheck와의 성능 비교를 시도하였다. 두 도구가 각기 다른 플랫폼에서 동작 하기 때문에 절대 적인 실행 시간 비교가 아닌 원본 프로그램 실행 시간 대비 이진 코드 가공을 통한 실행 시간 비율을 서로 비교하는 방식을 채택하였다. 이진 코드 가공 부하는 다음과 같은 방식으로 계산되었다.



(그림 9) Valgrind memcheck 와의 상대적 성능 비교

$$\text{Instrumentation overhead} = \frac{\text{이진가공된코드실행시간}}{\text{원본코드실행시간}}$$

테스트에 사용된 코드는 (그림 5)의 예제 코드를 100K~100M 번 반복 실행하여 부하를 높였다.

결과에 따르면 PinMemcheck의 경우 원본 프로그램 실행에 비해 약 18배 가량 성능이 줄어드는 모습을 보이는데 이는 실제 이동통신 기기 개발 과정의 성능 (그림 8)에 비해 다소 나쁜 수치이다. 이런 수치가 나온 이유는 실제 개발 중인 이동 통신 기기용 플랫폼 에뮬레이터에 비해 이번 테스트에 사용된 (그림 5)의 코드가 다수의 메모리 동작을 빈번하게 반복하기 때문에 더 많은 이진 코드 가공 및 실행이 발생했기 때문으로 추정할 수 있다. 그러나 Valgrind memcheck가 원본 프로그램 대비 약 50배의 성능 감소가 있었음을 감안하면 상대적으로 매우 좋은 성능을 보이고 있다고 할 수 있다.

5. 결 론

우리는 상용 이동통신 기기 개발에 있어 메모리 오류 검출이 가능한 Pin 기반의 동적 메모리 오류 검출 도구인 PinMemcheck을 개발하였다. 실제 피쳐폰에 PinMemcheck을 적용하여 유용성을 검증하였다. 본 연구에서 개발한 도구는 필수적인 메모리 오류 검출 기능을 제공함과 동시에 Pin 이진 코드 변환에 비해 약 1.5배의 성능 저하만을 보였다. 또한 상대적인 성능 비교 테스트를 통해 Valgrind memcheck에 비해 약 2.5배 이상의 성능을 보이는 것을 확인하였다.

참 고 문 헌

[1] 유용덕, 박상현, 최훈. “응용프로그램 특성을 고려한 모바일 플랫폼의 동적 메모리 관리기법”, 정보처리학회논문지A, 제13권 제7호, pp.561-572, 2006.

[2] 정진우, 장승주. “임베디드 시스템에서 가상 메모리 압축 시스템 설계”, 정보처리학회논문지A, 제9권 제4호, pp.405-412, 2002.

[3] 조대완, 오승욱, 김현수. “C언어 기반 프로그램의 소스코드 분석을 이용한 메모리 접근오류 자동검출 기법”, 정보처리학회논문지D, 제14권 제6호, pp.675-688, 2007.

[4] J. Kong, C. C. Zou and H. Zhou, “Improving software security via runtime instruction-level taint checking”, In Proceedings of the 1st workshop on Architectural and system support for improving software dependability, pp.18-24, 2006.

[5] A. Kolawa and A. Hicken, “Insure++: A tool to support total quality software”, www.parasoft.com/insure/papers/tech.htm.

[6] J. Gray, “Why do computers stop and what can be done about it?”, HP Labs Technical Reports / TR-85.7.

[7] N. Nethercote and J. Seward, “Valgrind: A program supervision framework”, In Proceedings of the 3rd Workshop on Runtime Verification, 2003.

[8] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation”, In Proceedings of Programming Language Design and Implementation (PLDI), pp.89-100, 2007.

[9] IBM Rational software, “IBM Rational Purify”, http://www.rational.com.

[10] QUALCOMM Incorporation, “Brew”, http://brew.qualcomm.com/brew/en/developer/getting_started/get_started.html

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S.Wallace, V. J. Reddi and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation”, In Proceedings of Programming Language Design and Implementation (PLDI), pp.191 - 200, 2005.

[12] N. Nethercote, “Dynamic Binary Analysis and Instrumentation”, PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, 2004.
[13] B. M. Cantrill, M. W. Shapiro and A. H. Leventhal, Dynamic instrumentation of production systems, In Proceedings of USENIX Annual Technical Conference (ATEC), 2004.
[14] D. L. Bruening, “Efficient, Transparent, and Comprehensive Runtime Code Manipulation”, PhD thesis, M.I.T. (http://www.cag.lcs.mit.edu/dynamorio/), 2004.
[15] T. Horel and G. Lauterbach, “UltraSPARC-III: Designing Third-Generation 64-Bit Performance”, In Proceedings of the International Symposium on Microarchitecture (MICRO), Vol.19, No.7, pp.73-85, 1999.
[16] Intel, “Intel Extended Memory 64 Technology Software Developer’s Guide”, Vol.1-2, 2004.
[17] Intel, “IA-32 Intel Architecture Software Developer’s Manual”, Vol.1-3, 2003.
[18] R. N. Kalla, B. Sinharoy and J. M. Tendler, “IBM Power5 chip: A dual-core multithreaded processor”, In Proceedings of the International Symposium on Microarchitecture (MICRO) / pp.40-47, 2004.
[19] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz and M. T. Vaden, “IBM POWER6 microarchitecture”, IBM Journal of Research and Development, Vol.51, No.6, pp.639-662, 2007.
[20] A. Srivastava and A. Eustace, “Atom: A system for building customized program analysis tools”, In Proceedings of Programming Language Design and Implementation (PLDI), pp.196-205, 1994.
[21] Intel, “Intel Itanium Architecture Software Developer’s Manual”, Vol.1-4, 2002.
[22] G. Contreras and M. Martonosi, “Power prediction for Intel XScale® processors using performance monitoring unit events”, In Proceedings of the 2005 international symposium on Low power electronics and design, pp.201-226, 2005.
[23] Intel, “Intel PXA27x Processor Family Developer’s Manual”, 2004.

조 경 진

e-mail : realdale@korea.ac.kr
2006년 고려대학교 전기전자전파공학부 (학사)
2008년 고려대학교 전자전기공학과 (공학석사)
2009년~현 재 고려대학교 전기전자전파공학과 박사과정
관심분야 : 컴파일러, 이동통신 기기 플랫폼, 개발 환경 최적화 등

김 선 욱

e-mail : seon@korea.ac.kr
1988년 고려대학교 전자전산공학과(학사)
1990년 Ohio State Univ. 전기공학과 (공학석사)
2001년 미국 퍼듀대학 전기컴퓨터공학과 (공학박사)
2002년~2005년 고려대학교 전기전자전파공학과 조교수
2005년~2009년 고려대학교 전기전자전파공학과 부교수
2009년~현 재 고려대학교 전기전자전파공학과 교수
관심분야 : 컴파일러, 프로세서 및 SoC 디자인, 병렬처리, 성능평가 등