정 보 보 호 학 회 지
제 32 권 제4호, 2022. 8

CMA 인식을 통한 메모리 안전성 강화 연구

홍 준 화*, 박 찬 민*, 정 성 윤*, 민 지 운*, 유 동 연*, 권 용 휘**, 전 유 석*

요 약

C/C++에는 다수의 메모리 취약점이 존재하며 ASan은 낮은 오버헤드와 높은 탐지율로 이러한 메모리 취약점을 탐지하기 위해 광범위하게 사용되고 있다. 그러나 상용 프로그램 중 다수는 메모리를 효율적으로 사용하기 위해 Custom Memory Allocator(CMA)를 구현하여 사용하며, ASan은 이러한 CMA로부터 파생된 버그를 대부분 탐지하지 못한다. 이를 극복하기 위해 본 연구에서는 LLVM IR 코드를 RNN 신경망에 학습하여 CMA를 탐지하고, ASan이 CMA를 식별할 수 있도록 수정하여 CMA로부터 파생된 메모리 취약점을 탐지할 수 있는 도구인 CMASan을 제안한다. ASan과 CMASan의 성능 및 CMA 관련 취약점의 탐지 결과를 비교·분석하여 CMASan이 낮은 실행시간 및 적은 메모리 오버헤드로 ASan이 탐지하지 못하는 메모리 취약점을 탐지할 수 있음을 확인하였다.

Ⅰ. 서 론

C/C++는 빠른 성능과 높은 자유도로 운영체제와 펌웨어 등 시스템 소프트웨어에 가장 광범위하게 사용되고 있는 언어이다. 그러나 C/C++는 완벽한 보안성을 제공하지 못하고 버퍼오버플로우, Use After Free(UAF) 등 메모리 취약점이 존재한다. Address Sanitizer(ASan)[1]은 비교적 낮은 오버헤드와 높은 탐지율로 이러한 메모리 취약점을 탐지하는 도구로 가장 많이 사용되고 있는 도구 중 하나이다.

다양한 응용 프로그램에서는 메모리를 효율적으로 사용하기 위해 Custom Memory Allocator(CMA)를 사용하는데, Istvan 등[2]에 따르면 Sanitizer가 CMA로부터 생성된 오브젝트(예; Firefox의 경우 60% 이상의 오브젝트가 CMA로부터 할당) 관련 메모리 안전 위반 취약점을 탐지하지 못하고 있다.

Xiangkun 등[3]은 이러한 CMA로부터 파생된 버그를 잡아내기 위한 도구인 HOTracer를 제안하였다. HOTracer는 CMA를 휴리스틱 기반의 행동 패턴을 도출하고 이를 기반으로 CMA를 탐지하여 할당자에서 파생된 포인터와 접근 범위를 추적하여 힙 버퍼오버플로우를 찾아낸다. 그러나 이러한 휴리스틱 기반 탐지 방법을 사용하면 CMA의 구현 방식에 따라 탐지하지 못하는 할당자가 존재할 수 있다. 예를 들어 HOTracer에서 사용하는 두 번째 단계 규칙 중 ‘CMA는 처음 할당이 일어나거나 내부 메모리 풀(Pool)이 고갈되었을 때 시스템으로부터 메모리를 할당하기 위해 표준 할당자 인터페이스를 사용한다.’의 경우 내부 할당자에서 메모리를 CMA를 사용해 할당하는 경우 CMA를 탐지할 수 없다. 또한, 동적 분석 도구이기 때문에 동적 분석 도구가 가지는 코드 커버리지 문제를 가지고 있으며 ASan과 같은 기존의 취약점 탐지도구와의 연결도 고려하고 있지 않다. Jianqiang 등[4]은 NLP와 symbolic execution 기술을 결합하여 메모리 할당자로부터 파생된 취약점을 탐지하는 NLP-EYE를 제안하였으나 특성이 충분히 제거되지 않은 소스코드를 데이터셋으로 활용하여 21%의 매우 낮은 수준의 정확도를 보였다. Franck 등[5]은 특정 규칙을 기반으로 바이너리로부터 libc 할당자와 CMA 할당자를 모두 탐지하여 0.4% 미만의 낮은 오류율을 보였으나 CMA에 대한 성능 평가를 LD_PRELOAD를 활용하여 기존 libc 할당자를 단순 대체하는 방식으로 실험을 설계하였기 때문에 libc 할당자와 유사한 인터페이스를 가진 CMA에 대해서만 평가가 진행되었다는 한계점이 존재한다.

이러한 기존 연구의 한계점을 개선하기 위하여 본 연구에서는 CMA를 탐지하고, CMA로부터 파생된 힙 버

* 울산과학기술원 (학부생, qbit@unist.ac.kr, 학부생, vow0723@unist.ac.kr, 학부생, jsy01311@unist.ac.kr, 학부생, min1905@unist.ac.kr, 학부생, dy3199@unist.ac.kr, 조교수, ysjeon@unist.ac.kr)
** 버지니아 주립대 (조교수, yongkwon@virginia.edu)

퍼오버플로우를 탐지할 수 있는 CMASan(Custom Memory Allocator Sanitizer)을 제안한다. 정적 시간에 효과적으로 CMA를 탐지하기 위해 코드의 특성이 일부 제거된 LLVM Intermediate Representation(IR)을 RNN 신경망을 통해 학습하였다. 이러한 모델을 통해 식별된, CMA를 ASan이 식별할 수 있도록 수정하여 CMA로부터 파생된 힙 버퍼오버플로우가 탐지가 가능하도록 하였다

CMASan의 CMA 관련 메모리안전 위반 취약점 탐지 능력을 평가하기 위해 상용 프로그램에 사용되는 분류별 CMA 취약점을 생성하고, 선별된 테스트 프로그램에 ASan과 CMASan을 각각 적용하고 비교하여 CMASan이 보다 효과적으로 힙 버퍼오버플로우를 탐지하는 것을 증명하였다. 또한, CMASan의 실용성을 확인하기 위해 SPEC CPU2017 벤치마크를 적용한 결과 ASan 대비 CMASan가 평균적으로 실행시간 1.07x 오버헤드 및 메모리 사용량 1.05x의 낮은 오버헤드로 동작하는 것을 확인하였다.

Ⅱ. 배경지식

2.1. ASan(Address Sanitizer)

ASan은 메모리 취약점을 탐지하는 도구로, UAF, 힙 버퍼오버플로우(Heap buffer overflow), 스택 버퍼오버플로우(Stack buffer overflow) 등 8가지 메모리 안전 위반 취약점을 섀도우 메모리 기반으로 탐지한다. 섀도우 메모리는 사용자에게 할당되는 힙 영역의 범위를 특정하고, 해당 범위로부터 특정 오프셋만큼 떨어진 곳에 대응되는 정보 저장소를 생성하는 저장 방식이다. ASan은 힙 영역의 각 8byte를 섀도우 메모리 1byte와 대응시켜 레드존(redzone) 상태를 저장할 수 있는 공간을 생성한다. 런타임에서 사용자에게 할당된 영역은 비오염(unpoison) 상태로, 할당되지 않은 영역은 오염(poison) 상태로 저장하여 오염된 영역에 사용자가 접근할 경우 버퍼오버플로우로 판단한다. ASan은 malloc, free를 비롯한 표준 라이브러리에서 제공하는 할당자를 자체 함수로 교체하여 메모리 할당 및 해제에 따라 섀도우 메모리를 업데이트 한다.

2.2. Custom Memory Allocaotor (CMA)

x264, blender 등 이미지/영상 처리 프로그램과 같이 메모리를 많이 사용하는 프로그램은 표준 라이브러리에서 제공하는 malloc 계열과 메모리 할당자를 사용하는 대신 특수 목적을 위한 CMA를 구현하여 사용한다. 이러한 CMA는 메모리 풀 관리, 메모리 재사용, 보안성 향상, address alignment 등의 장점을 가진다. 특정 CMA의 경우 ASan이 인식하는 할당된 영역의 크기보다 작은 크기의 영역을 사용자에게 할당하기 때문에 ASan에서 해당 CMA에서 파생되는 힙 버퍼오버플로우를 탐지할 수 없는 경우가 존재한다.

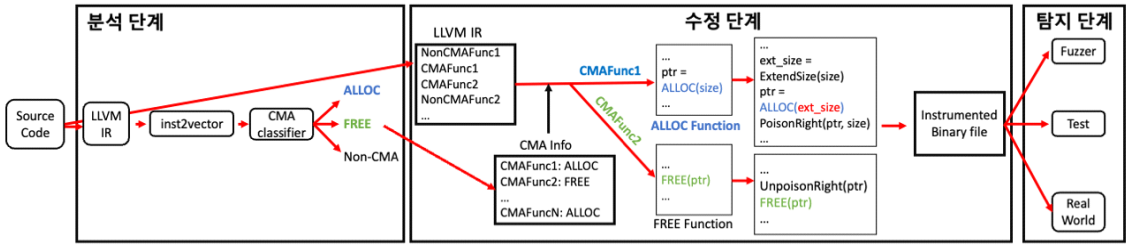
Ⅲ. 디자인

3.1. 개요

본 연구에서 제안하는 Custom Memory Allocator Sanitizer(CMASan) 도구는 그림 1.과 같이 ‘분석 단계(analysis process)’, ‘수정 단계(Modification process)’, ‘탐지 단계(Detection process)’로 구성되어 있다.

3.2. CMA 패턴화

상용 C/C++ 프로그램 사용되는 CMA 패턴을 파악하기 위해 SPEC CPU2006, SPEC CPU2017,



〔그림 1〕 CMASan 전체 구성도

Phoronix C/C++ Test Suite에 속해 있는 100여개의 상용 프로그램을 분석하였다. CMA는 크게 Pool, Freelist, 좌측 패딩(Left-padding), 우측 패딩(Right-padding), Wrapper의 다섯 가지 패턴으로 분류할 수 있다. Pool은 메모리 할당에 드는 시간을 감소시키고 효율적으로 메모리를 재활용하기 위해 libc 할당자, mmap 등으로 미리 Pool을 할당해두고 해당 버퍼를 사용자가 할당을 요청할 때마다 분할하여 사용하는 방식이다. ASan은 Pool을 하나의 청크(chunk)로 판단하기 때문에 Pool 내부에서 발생하는 할당에 대한 힙 오버플로우를 탐지할 수 없다. Pool에서 분할된 청크를 해제할 경우 단편화(fragmentation)가 발생하게 되는데, 이러한 단편화 문제를 해결하기 위해서 Pool 패턴과 Freelist 패턴을 동시 적용하여 CMA를 구현하는 경우도 다수 존재한다. Freelist는 처음 청크가 할당된 뒤 사용자가 해제를 요청하였을 때 해당 청크를 리스트에 추가하여 이후 할당에 재활용하는 방식이며 이를 위해 최초적합(first-fit), 최적적합(best-fit), 최악적합(worst-fit) 알고리즘을 사용할 수 있다. 그러나 최초적합과 최악적합을 사용할 경우 내부 단편화가 발생하여 ASan이 탐지하지 못하는 영역이 발생한다. 패딩(Padding)은 특수한 목적을 위해 사용자가 요청한 크기보다 큰 크기의 청크를 사용하는 방식이다. 이때, 유저가 사용하는 영역의 왼쪽에 패딩이 존재할 경우 좌측 패딩, 오른쪽에 패딩이 존재할 경우 우측 패딩으로 분류하였다. 좌측 패딩은 메타데이터(metadata)를 저장하기 위한 공간으로 주로 사용되며, 우측 패딩은 캐너리(canary)를 삽입하여 자체적인 메모리 방어 체계를 구현하기 위해 주로 사용된다. Alignment를 위해 패딩을 사용할 경우, 내부 할당자로부터 반환되는 주소 값에 따라 좌측 패딩과 우측 패딩의 크기가 동적으로 변하게 된다. 패딩은 패딩 영역을 포함한 크기를 표준 할당자로부터 할당받아 유저에게 패딩을 제외한 영역을 리턴하기 때문에 패딩 영역으로의 오버플로우를 ASan이 탐지할 수 없다. Wrapper는 유저로부터 전달받은 크기와 함께 내부 할당자를 호출하여 주소값을 변경 없이 반환하는 방식이다. 예외처리와 할당 상태 추적을 위해 주로 사용되며, ASan이 인식하는 청크 크기와 같은 크기의 청크를 유저에게 제공하므로 힙 오버플로우가 ASan에 의해 탐지될 수 있다.

본 연구에서는 ASan이 힙 버퍼오버플로우를 탐지할 수 없는 Pool, Freelist, 좌측 패딩, 우측 패딩 패턴 중 좌측 패딩을 제외한 세 가지 패턴을 대상으로 버그 탐지 도구를 구현하였다. 메타데이터가 주로 저장된 좌측 패딩으로의 접근은 오버플로우가 아닌, 정상 접근으로 판단되어야 하므로 버그 탐지 대상에서 제외하였다.

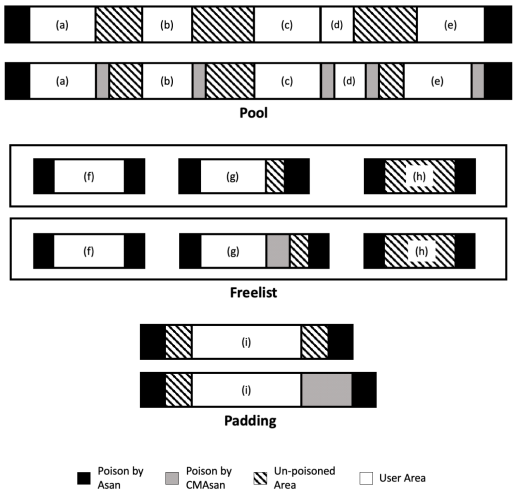
3.3. 분석 단계

CMASan 도구는 프로그램 내의 CMA를 탐색할 수 있는 분류기(classifier)를 제공한다. 분류기에는 Pool 및 FreeList 패턴이 학습되어 있으며, 프로그램의 LLVM IR 코드를 제공하면 ALLOC, FREE, Non-CMA의 세 가지 카테고리로 함수를 분류한다. 청크를 할당하는 함수는 ALLOC, 해제하는 함수는 FREE로 분류되며, 이외의 함수는 Non-CMA로 분류된다. 사용자는 분류기를 활용하여 프로그램을 분석한 뒤, 탐지된 CMA 정보를 수정 단계 및 분석 단계에서 활용한다.

3.4. 수정 단계

수정 단계는 소스코드가 LLVM IR 코드로 변환되었을 때 기존 IR 코드를 수정하고 일부 코드를 추가적으로 instrumentation하는 방식으로 진행된다. 구체적으로, CMASan은 사용자가 요청한 청크의 크기를 레드존 크기만큼 추가적으로 증가하여 요청하고, 확장된 청크의 오른쪽을 오버플로우 취약점 탐지를 위해 오염(poison)시킨다. 이를 위해, 분석 단계에서 전달받은 CMA 정보를 기반으로 ALLOC 함수와 FREE 함수에 instrumentation을 진행한다. ALLOC 함수 호출 코드 앞에 청크 크기 인자를 레드존만큼 증가시키는 런타임 함수가 삽입되며, 호출 코드의 뒤에는 증가된 영역을 오염시키는 런타임 함수가 삽입된다. FREE 함수의 호출 코드 앞에는 ALLOC 함수 호출코드의 뒤에서 오염되었던 영역을 해제하는 런타임 함수가 삽입된다. 이때, 오염된 영역의 위치를 판단하기 위해 청크의 사이즈 정보가 필요하므로, ALLOC 함수의 후방에서 오염시키는 과정에서 추가적인 메타데이터 테이블에 정보를 기록한다.

Pool 할당자의 경우 그림 2.의 청크 (a), (b), (c), (d), (e)와 같은 경우가 존재한다. (a)의 좌측 오버플로우와 (e)의 우측 오버플로우는 기존 ASan에 의해 탐지될 수 있지만 (a)의 우측 및 (e)의 좌측 오버플로우는 탐지될 수 없다. 또한 (b), (c), (d)의 경우 양 말단에 오염된 영



〔그림 2〕 Pool, Freelist, 패딩 할당자의 청크 할당에 대한 ASan(상) 및 CMASan(하) 적용 예

역이 존재하지 않기 때문에 오버플로우가 탐지될 수 없다. 그러나 CMASan의 수정 단계 이후에는 모든 청크의 우측에 오염된 영역이 삽입되므로 Pool의 모든 청크의 우측 오버플로우가 탐지될 수 있으며, (a), (d)의 경우 좌측 오버플로우가 탐지될 수 있다. 일반적으로, Pool 패턴은 Pool 청크의 전방으로부터 청크를 연속적으로 할당하여 (d) 패턴이 가장 많이 발생한다.

Freelist 할당자의 경우 청크 (f), (g), (h)의 경우가 존재하며, (f)의 경우 최적적합, (g)의 경우 최초적합 혹은 최악적합으로 할당된 청크이며, (h)는 해제된 상태의 청크이다. (f)의 오버플로우는 기존 ASan에 의해 탐지될 수 있으며, (g), (h)의 좌측 오버플로우 역시 탐지될 수 있다. 그러나 (g)의 우측 말단에 오염된 영역이 존재하지 않으므로 우측 오버플로우가 탐지될 수 없다. 반면, CMASan 수정 단계 이후에는 (g)의 우측 말단에 오염된 영역이 삽입되므로 Freelist의 모든 청크의 좌측 및 우측 오버플로우가 탐지될 수 있다.

패딩 할당자의 경우 청크 (i)와 같이 좌측 패딩 및 우측 패딩이 존재하는 경우와 단방향으로 패딩이 존재하는 경우가 발생할 수 있다. 양 말단에만 오염된 영역이 존재하므로 ASan에 의해 오버플로우가 탐지될 수 없지만, CMASan의 수정 단계 이후 우측 오버플로우 탐지가 가능하다. 좌측 패딩의 경우 메타데이터가 저장되는 영역으로 사용되는 경우가 다수이므로 메타데이터에 대한 접근과 좌측 오버플로우를 구분할 수 없어 CMASan의 탐지 대상에서 제외하였다.

3.5. 탐지 단계

힙 오버플로우의 실제 탐지는 프로그램이 실행되는 단계인 탐지 단계에서 이루어진다. 유저가 오염된 영역인 레드존에 엑세스할 경우 힙 오버플로우로 보고한다. 이때, 해당 영역이 추가적인 CMASan의 메타데이터 테이블에 표시되어 있다면 CMA 관련 힙 버퍼오버플로우로 보고한다. 분석 단계는 퍼저(fuzzer) 혹은 테스트 프로그램과 결합하여 버그를 탐색하는 방식으로도 진행할 수 있다.

Ⅳ. 구 현

AI 모델이 프로그래머 코드 스타일에 영향을 받지 않고 CMA의 알고리즘 특징을 학습할 수 있도록 불필요한 코드 특징 등이 일부 제거된 IR 코드를 RNN 신경망에 학습시키고자 하였다. Arrow, leveldb, nginx, scipy 등 8개 프로그램에 포함된 Freelist 및 Pool 할당자를 대상으로 데이터셋을 구축하였으며, 패딩의 경우 알고리즘이 ASan으로 탐지가 가능한 Wrapper와 유사한 로직을 가지고 있으므로 거짓 양성을 방지하기 위해 데이터셋에서 제외하였다. 하지만 사용자가 분석 단계에서 직접 패딩 할당자를 CMA 정보에 추가하여 패딩 할당자로부터 파생된 버그를 탐지할 수 있다. IR 코드를 기반으로 각 명령을 임베딩(embedding)하여 RNN 신경망에 학습시켰다. 모델의 결과를 유저가 수정할 수 있는 YAML 직렬화 양식으로 출력하고, LLVM에서 YAML의 정보를 기반으로 CMA를 식별하도록 구현하여 사용자가 모델의 결과를 참고하여 수정 단계 및 분석 단계를 수행할 수 있도록 하였다.

LLVM 11.0.0 버전을 기반으로 CMASan 도구를 구현하였다. 런타임 라이브러리에 ALLOC 및 FREE 함수의 전후방에 삽입된 instrumentation을 처리할 런타임 함수를 구현하였다. 구체적으로, ALLOC 함수의 크기 인자를 증가시키는 extendSize 및 ALLOC 함수에서 리턴되는 청크의 우측을 오염시키는 PoisonRight 런타임 함수를 구현하여 ALLOC의 전후방에 삽입하였으며, 메타데이터 테이블을 참조하여 FREE 함수의 포인터 인자로부터 청크 크기만큼 떨어진 레드존을 비오염 상태로 변경하는 UnPoisonRight 런타임 함수를 구현하여 FREE 함수의 전방에 삽입하였다. Instrumentation은 LLVM의 ASan 패스가 적용되기 이전에 적용하도록 구

현하였다. 또한, ASan의 report 로직을 수정하여 힙 오버플로우를 탐지할 경우 테이블을 확인하여 CMA로부터 파생된 버그인지 판단할 수 있도록 구현하였다. 이를 위해 2단계 테이블(two level 표)에 청크의 크기를 기록하였다.

Ⅴ. 실 험

ASan과 CMASan의 비교 분석을 통해 CMASan의 성능 및 기능을 검증하고자 하였다.

5.1. 성능 평가 환경

SPEC CPU2017 벤치마크 프로그램 중 CMA가 포함된 502.gcc_r, 523.xalancbmk_r, 511.povray_r, 526.blender_r, 538.imagick_r 벤치마크 대상으로 ASan 대비 CMASan을 적용하였을 때의 실행시간을 측정하였다. 또한, 동일한 프로그램을 대상으로 메모리 사용량 측정 툴인 perf mem을 활용하여 메모리 사용량을 측정하였다.

CMA 분류기의 정확도 측정은 SPEC CPU2017에 포함된 벤치마크 중 Pool 및 Freelist 할당자를 포함하고 있는 502.gcc_r, 523.xalancbmk_r, 538.imgaick_r 벤치마크를 대상으로 진행하였다.

5.2. 기능 평가 환경

비교 분석을 위해 상용 프로그램 중 Crypto++, nginx, leveldb에 포함된 padding, pool, freelist 패턴의 할당자를 사용하였다.

Crypto++의 Aligned 할당자는 청크의 16-byte alignment를 위해 사용되는 패딩 패턴에 해당된다. 주소값에 따라 최소 1byte, 최대 16byte의 좌측 패딩과 최대 15byte, 최소 0byte의 우측 패딩이 삽입된다. 좌측 패딩 영역 중 청크와 인접한 1byte가 청크의 크기를 저장하기 위해 사용된다. 즉, Aligned 할당자의 내부 할당자에서 정렬된 주소가 반환될 경우 16byte의 좌측 패딩이, 정렬되지 않은 주소가 반환될 경우 정렬을 위한 오프셋(offset) 크기의 좌측 패딩과 16-오프셋 크기의 우측 패딩이 추가된다. 본 실험에서는 좌측 패딩과 우측 패딩에 대한 실험을 모두 진행하기 위해 내부 할당자가 정렬되지 않은 주소를 반환하는 시나리오에 대해 실험

```
#include "../aligned/aligned.h"

#ifdef TEST_PADDING_LEFT
static const int OFFSET = -1;
#elif TEST_PADDING_RIGHT
static const int OFFSET = 10;
#endif

int main() {
    char* p1 =
    (char*) AlignedAllocate(sizeof(char)*10);
    *(p1+OFFSET) = 123;
    AlignedDeallocate(p1);
}
```

〔그림 3〕 Crypto++의 aligned 할당자를 활용한 Padding 패턴 실험 코드

을 진행하였다. 내부 할당자가 반환하는 주소의 alignment를 컨트롤하기 위해 내부 할당자를 항상 8-byte alignment된 주소를 리턴하는 libc malloc으로 교체하고 오프셋을 조절할 수 있도록 구현하여 좌우 패딩에 대한 오버플로우 탐지 여부를 실험하였다.

nginx의 ngx_pool 할당자는 Pool에서 처음 할당한 큰 버퍼(buffer)의 앞에서부터 청크를 쪼개어 할당하는 Pool 패턴에 해당된다. Pool의 잔여 크기보다 큰 할당이 요청될 경우 일반적인 할당 로직을 따르게 되므로 해당 로직을 실험에서 제거하였다. 그림2.의 청크 중 (b)와 (d)를 대상으로 하여 좌우 오버플로우의 탐지 여부를 실험하였다. 그림4.와 같이 세 개의 청크를 연속적으로 할당한 후 가운데 청크인 p3을 대상으로 오버플로우를 발생시켜 (d)의 조건을 재현하였다. 또한 p3에 오버플로우가 존재하지 않을 경우 p3을 해제한 후 p4를 대상으로 좌측 오버플로우를 발생시켜 (b) 조건을 재현하였다.

leveldb의 LowLevel 할당자는 Freelist를 유지하면서 블록을 할당할 때 최적적합 방식으로 Freelist로부터 청크를 할당하는 Freelist 패턴에 해당되며, Freelist에 청크가 존재하지 않을 때 mmap을 사용하여 페이지를 할당하여 전방부터 할당하므로 Pool 패턴에도 해당된다. 또한 청크의 크기 정보를 청크의 전방에 저장하므로 Left padding 패턴 또한 해당된다. 기능 평가에서는 Freelist 패턴에 초점을 맞춰 실험을 진행하였다. 그림2.의 청크 (g)를 타겟으로 하여 우측 오버플로우의 탐지 여부를 실험하였다. 좌측 오버플로우의 경우 Crypto++의 Aligned 할당자에 관한 실험과 중복되므로 우측 오

```
#include "../ngx_pool/ngx_pool.h"
#define POOL_SIZE 4096

#ifdef TEST_POOL_LEFT
static const int OFFSET = -1;
#elif TEST_POOL_RIGHT
static const int OFFSET = 10;
#elif TEST_POOL_B_LEFT
static const int OFFSET = 3;
#endif

int main(){
    ngx_pool_t* pool =
    ngx_create_pool(POOL_SIZE);

    char* p2 = (char*)
    ngx_palloc_small(pool, sizeof(char)*10);
    char* p3 = (char*)
    ngx_palloc_small(pool, sizeof(char)*10);
    char* p4 = (char*)
    ngx_palloc_small(pool, sizeof(char)*10);

    *(p3+OFFSET) = 123;
    ngx_pfree(pool, p3);

    *(p4-1) = 123;
    ngx_pfree(pool, p2);
    ngx_pfree(pool, p4);

    ngx_destroy_pool(pool);
}
```

〔그림 4〕 nginx의 ngx_pool 할당자를 활용한 Pool 패턴 실험 코드

```
#include "../low_level/low_level.h"

static const int OFFSET = 7;

int main(){
    LowLevelAlloc a;
    LowLevelAlloc::Arena* arena = ac.DefaultArena();

    char *p5 =
    (char*) a.AllocWithArena(sizeof(char)*10, arena);
    char *p6 =
    (char*) a.AllocWithArena(sizeof(char)*9, arena);
    char *p7 =
    (char*) a.AllocWithArena(sizeof(char)*8, arena);

    a.Free(p6); // p2 is in freelist

    // first-fit is freed p2
    char *a =
    (char*) a.AllocWithArena(sizeof(char)*7, arena);
    *(p+OFFSET) = 123;

    a.Free(p);
    allocator.Free(p5);
    allocator.Free(p7);
}
```

〔그림 5〕 leveldb의 LowLevel 할당자를 활용한 Freelist 패턴 실험 코드

버플로우만을 대상으로 하였다. 그림5.와 같이 9byte 크기의 청크 p6을 해제한 뒤 7byte 크기의 청크 p8을 할당하여 그림2.의 (g) 조건을 재현하였다.

Ⅵ. 결 과

6.1. 기능 평가 결과

표 1과 같이 ASan이 6개의 취약점을 모두 탐지하지 못하는 것을 확인할 수 있었다. 반면, CMASan은 Pool 할당자로부터 할당된 청크 p4의 좌측 오버플로우와 Padding 할당자로부터 할당된 청크 p1의 좌측 오버플로우를 제외하고 6개의 취약점 중 4개의 취약점을 탐지하여 높은 탐지율을 보였다.

패딩 할당자로부터 할당된 p1 청크에 대한 좌우 오버플로우 실험 결과, ASan은 패딩 영역에 레드존이 존재하지 않아 p1의 좌측과 우측에 대한 버퍼오버플로우를 모두 탐지하지 못하였으나, CMASan은 우측 오버플로우를 성공적으로 탐지한 것을 확인할 수 있다. [그림 6]의 메모리 맵에서 청크의 오른쪽에 16byte의 레드존이 할당되어 우측 오버플로우가 탐지된 것을 확인할 수 있다.

Pool 할당자로부터 할당된 p3 청크의 좌우 오버플로우 실험결과, ASan은 Pool 내부에 레드존이 존재하지 않아 오버플로우를 탐지하지 못하였고, CMASan은 p2로부터 할당된 레드존을 활용하여 좌측 오버플로우를, p3로부터 할당된 레드존을 활용하여 우측 오버플로우를 탐지한 것을 확인할 수 있다. 반면 p4의 좌측 오버플로우의 경우 전방에 청크가 존재하지 않기 때문에 CMASan 역시 좌측 오버플로우를 탐지할 수 없음을 확인할 수 있다. 그림7.의 메모리 맵에서 p2, p3, p4의 우측에 각각 16byte의 레드존이 삽입된 것을 확인할 수 있다.

Freelist 할당자로부터 할당된 p8 청크의 우측 오버플로우 실험 결과, 최초적합으로 할당된 청크가 내부 할

```
Shadow bytes around the buggy address:
  0x0c0c7fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0c7fff8000: fa fa fa fa fd fd fd fd fd fd fd fd fa fa fa fa
=>0x0c0c7fff8010: 00 00 00 00 00 00 00 06 fa fa fa fa 00 00 00[02]
  0x0c0c7fff8020: fa fa 03 fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0c7fff8060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

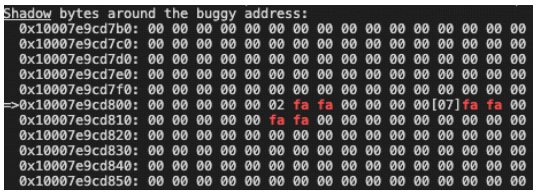
〔그림 6〕 Padding 할당자인 Aligned 할당자로 할당된 p1에 대한 우측 오버플로우 실험 결과

〔표 1〕 ASan 및 CMASan의 기능 평가 실험 결과 비교

할당자	패턴	청크 포인터	청크 타입	취약점	ASan	CMASan
ngx_pool	Pool	p3	(d)	좌측 오버플로우	X	O
ngx_pool	Pool	p3	(d)	우측 오버플로우	X	O
ngx_pool	Pool	p4	(b)	좌측 오버플로우	X	X
Aligned	Padding	p1	(i)	좌측 오버플로우	X	X
Aligned	Padding	p1	(i)	우측 오버플로우	X	O
LowLevel	Freelist	p8	(g)	우측 오버플로우	X	O

당자에 의해 할당된 크기보다 작아, ASan이 우측 오버플로우를 탐지하지 못하는 것을 확인할 수 있다. 반면 CMASan은 할당 시 레드존을 추가하여 우측 오버플로우를 탐지한 것을 확인할 수 있다. 그림8.의 메모리 맵에서 p5, p8, p7의 우측에 각각 16byte의 레드존이 삽입된 것을 확인할 수 있다.

〔그림 7〕 Pool 할당자인 ngx_pool 할당자로 할당된 p3에 대한 좌측 오버플로우 실험 결과



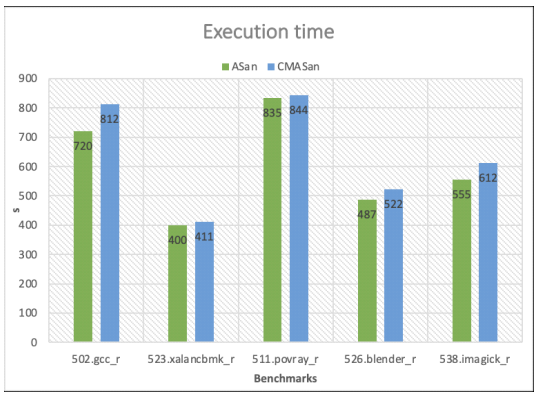
〔그림 8〕 Freelist 할당자인 LowLevel 할당자로 할당된 p8에 대한 우측 오버플로우 실험 결과

6.2. 성능 평가 결과

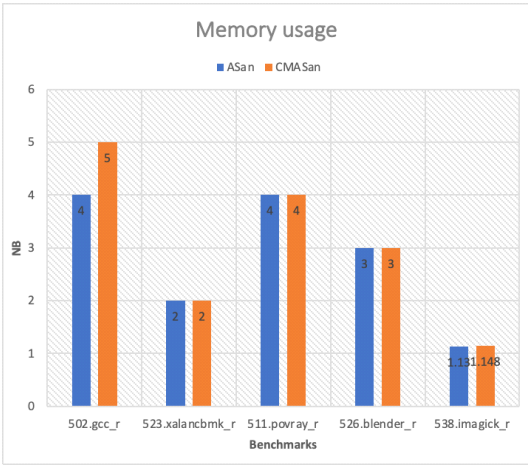
SPEC CPU2017 벤치마크 프로그램으로 실행시간을 비교한 결과 평균적으로 실행시간 오버헤드는 1.07x로 측정되었다. 또한 perf mem을 활용하여 메모리 사용량을 비교한 결과 평균적으로 메모리 오버헤드는 1.05x로 나타났다. 2단계 테이블 초기화로 인하여 실행시간이 증가하였으며, 2단계 테이블 및 레드존 영역 추가할당으로 인해 메모리 사용량이 일부 증가함을 확인할 수 있다. 해당 실험결과를 통해, CMASan이 낮은 실행시간 및 메모리 오버헤드를 가지는 실용적인 도구임을 확인할 수 있었다. 또한, 502.gcc_r, 523.xalancbmk_r, 538.imgaick_r 벤치마크에 포함된 모든 함수를 분류 및 평가한 결과 Non-CMA 함수를 대상으로 90%, CMA 함수를 대상으로 46%의 정확도를 보였다. 이는, NLP-EYE의 CMA 분류 정확도인 21%보다 2배 이상 향상된 수치이다.

〔표 2〕 SPEC CPU2017 벤치마크 실행시간 및 메모리 사용량 오버헤드 측정 결과

Benchmark	overhead (execution time)	overhead (memory usage)
502.gcc_r	1.13x	1.25x
523.xalancbmk_r	1.03x	1x
511.povray_r	1.01x	1x
526.blender_r	1.07x	1x
538.imagick_r	1.10x	1.02x
Average	1.07x	1.05x



〔그림 9〕 SPEC CPU2017 벤치마크 실행시간 오버헤드 측정 결과



〔그림 10〕 SPEC CPU2017 벤치마크 메모리 사용량 오버헤드 측정 결과

Ⅶ. 결 론

ASan은 C/C++ 프로그램에 존재하는 메모리 취약점을 효율적으로 탐지해낼 수 있지만 표준 할당자만을 대상으로 동작하기 때문에 CMA로부터 파생되는 일부 오버플로우를 탐지할 수 없다. 따라서 본 연구에서는 100여개의 상용 프로그램을 분석하여 CMA를 패턴화하고, ASan이 탐지하지 못하는 취약점을 지닌 Pool, Freelist, 패딩 패턴의 할당자들을 대상으로 힙 버퍼오버플로우를 탐지할 수 있는 CMASan을 구현하였다. 상용 프로그램에 포함된 각 패턴의 할당자를 사용한 기능평가를 시행하여 ASan이 탐지하지 못하는 힙 버퍼오버플로우를 낮은 오버헤드로 CMASan이 탐지할 수 있음을 보였다.

참 고 문 헌

[1] SEREBRYANY, Konstantin, et al, “AddressSanitizer: A Fast Address Sanity Checker”, 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 309-318, 2012.

[2] HALLER, Istvan, et al, “TypeSan: Practical type confusion detection”, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 517-528, 2016.

[3] JIA, Xiangkun, et al, “Towards efficient heap overflow discovery”, 26th USENIX Security Symposium (USENIX Security 17), pp. 989-1006, 2017.

[4] WANG, Jianqiang, et al, “NLP-EYE: Detecting Memory Corruptions via {Semantic-Aware} Memory Operation Function Identification”, 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 309-321, 2019.

[5] DE GOËR, “Franck; GROZ, Roland; MOUNIER, Laurent. Metrics for runtime detection of allocators in binaries”, 10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17), 2017.

〈저 자 소 개〉

홍 준 화 (Junwha Hong)
학생회원
2020년 2월~현재 : 울산과학기술원 (UNIST) 컴퓨터공학과 학사과정
<관심분야> 시스템구조, 시스템보안

박 찬 민 (Chanmin Park)
2021년 2월~현재 : 울산과학기술원 (UNIST) 컴퓨터공학과 학사과정
<관심분야> 소프트웨어 보안

정 성 윤 (Seongyun Jeong)

2020년 2월~현재 : 울산과학기술원 (UNIST) 컴퓨터공학과 학사과정

<관심분야> 소프트웨어 보안

민 지 운 (Jiun Min)

2017년 2월~현재 : 울산과학기술원 (UNIST) 컴퓨터공학과 학사과정

<관심분야> 소프트웨어 보안

유 동 연 (Dongyeon Yu)

2017년 2월~현재 : 울산과학기술원 (UNIST) 컴퓨터공학과 학사과정

<관심분야> 소프트웨어 보안

권 용 휘 (Yonghwi Kwon)

2011년 7월 : 건국대학교 컴퓨터공학과 졸업

2018년 8월 : 퍼듀대학교 컴퓨터공학 박사

2018년 8월~현재 : 버지니아 주립대 (University of Virginia) 조교수

<관심분야> 시스템보안, 소프트웨어, 정보보호

전 유 석 (Yuseok Jeon)

종신회원

2007년 8월 : 인하대학교 컴퓨터공학 졸업

2010년 2월 : 포항공과대학교 정보통신학 석사

2020년 12월 : 퍼듀대학교 컴퓨터공학 박사

2021년 2월~현재 : 울산과학기술원(UNIST) 컴퓨터공학과 조교수

<관심분야> 소프트웨어 보안 및 시스템보안