(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2024-0128620
(43) 공개일자 2024년08월26일

(51) 국제특허분류(Int. Cl.)
G06F 21/52 (2013.01) *G06F 21/55* (2013.01)
(52) CPC특허분류
G06F 21/52 (2013.01)
G06F 21/55 (2013.01)
(21) 출원번호 10-2024-0023215
(22) 출원일자 2024년02월19일
심사청구일자 2024년02월19일
(30) 우선권주장
1020230021188 2023년02월17일 대한민국(KR)

(71) 출원인
고려대학교 산학협력단
서울특별시 성북구 안암로 145, 고려대학교 (안암동5가)
(72) 발명자
진홍주
서울특별시 성북구 고려대로27길 34-8, B동 505호
이동훈
서울특별시 종로구 사직로8길 34, 1404호(내수동, 경희궁의아침3단지아파트)
(뒷면에 계속)
(74) 대리인
김등용

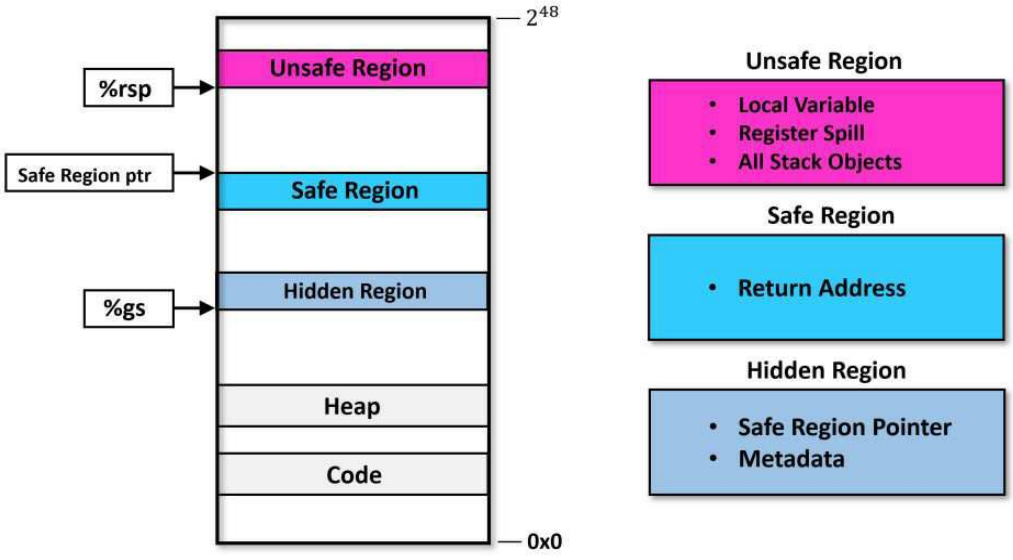
전체 청구항 수 : 총 9 항

(54) 발명의 명칭 **이중 스택 격리 기술을 이용한 정보 유출 방어 방법**

(57) 요 약

이중 스택 격리 기술을 이용한 정보 유출 방어 방법이 개시된다. 상기 정보 유출 방어 방법은 타겟 프로그램을 실행하는 컴퓨팅 장치에 의해 수행되고, 상기 타겟 프로그램에 대응하는 프로세스의 가상 메모리에 안전 영역(safe region)을 할당하는 단계, 상기 가상 메모리에 미리 정해진 크기를 갖는 적어도 하나의 청크(chunk)를 할당하는 단계, 및 상기 적어도 하나의 청크를 재할당하는 단계를 포함한다.

대 표 도 - 도1



(72) 발명자

이지원

서울특별시 성북구 고려대로1길 83, 404호

김선권

대전광역시 유성구 가정로 43, 107동 903호

이 발명을 지원한 국가연구개발사업

과제고유번호	1711160981
과제번호	2021R1A2C2014428
부처명	과학기술정보통신부
과제관리(전문)기관명	한국연구재단
연구사업명	개인기초연구(과기정통부)
연구과제명	PHY-level에서 획득 가능한 부채널 정보를 이용한 기기 보안서비스 기술 연구
과제수행기관명	고려대학교
연구기간	2021.03.01 ~ 2024.02.29

명 세 서

청구범위

청구항 1

타겟 프로그램을 실행하는 컴퓨팅 장치에 의해 수행되는, 정보 유출 방어 방법에 있어서,

상기 타겟 프로그램에 대응하는 프로세스의 가상 메모리에 안전 영역(safe region)을 할당하는 단계;

상기 가상 메모리에 미리 정해진 크기를 갖는 적어도 하나의 청크(chunk)를 할당하는 단계; 및

상기 적어도 하나의 청크를 재할당하는 단계를 포함하는 정보 유출 방어 방법.

청구항 2

제1항에 있어서,

상기 안전 영역을 할당하는 단계는, 상기 타겟 프로그램이 상기 가상 메모리에 로드될 때, 임의의 주소에 할당하는,

정보 유출 방어 방법.

청구항 3

제1항에 있어서,

상기 안전 영역을 할당하는 단계는, 상기 가상 메모리에 숨겨진 영역(hidden region)을 더 할당하고,

상기 안전 영역에 대한 포인터 값을 포함하는 안전 영역 포인터 객체(safe region pointer object)는 상기 숨겨진 영역에 저장되는,

정보 유출 방어 방법.

청구항 4

제3항에 있어서,

상기 숨겨진 영역의 주소는 %gs 레지스터에 의해 식별되고,

상기 %gs 레지스터는 archi_prctl() 시스템 호출(system call)에 의해 활성화되는,

정보 유출 방어 방법.

청구항 5

제1항에 있어서,

상기 안전 영역에는 상기 프로세스의 반환 주소(return address)가 저장되는,

정보 유출 방어 방법.

청구항 6

제1항에 있어서,

상기 안전 영역은, 상기 가상 메모리 내에서, 공유 라이브러리(shared libraries) 아래 주소 공간에 할당되는,

정보 유출 방어 방법.

청구항 7

제1항에 있어서,

상기 적어도 하나의 청크는 상기 가상 메모리 내에서 가장 큰 홀(hole) 내에 할당되는,

정보 유출 방어 방법.

청구항 8

제1항에 있어서,

상기 적어도 하나의 청크는 제1 청크와 제2 청크를 포함하고,

상기 제1 청크는, 상기 가상 메모리 내에서, 공유 라이브러리 아래 주소 공간 중 상반부에 할당되고,

상기 제2 청크는, 상기 공유 라이브러리 아래 주소 공간 중 하반부에 할당되는,

정보 유출 방어 방법.

청구항 9

제1항에 있어서,

상기 적어도 하나의 청크를 재할당하는 단계는, 상기 타겟 프로그램이 실행되는 동안, 상기 적어도 하나의 청크를 주기적으로 재할당하는,

정보 유출 방어 방법.

발명의 설명

기 술 분 야

[0001] 본 발명은 시스템 보안에 관한 것으로, 특히 프로세스의 가상 메모리 보안에 관한 것이다.

배 경 기 술

[0002] 메모리 손상(memory corruption) 취약점은 다량의 데이터를 처리하는 서버 프로그램부터 임베디드 시스템을 기반으로 하는 사물 인터넷(IoT) 제품까지 다양한 환경에서 발생한다. 가상 메모리 구조에서 스택 메모리 영역에서 발생하는 메모리 손상(e.g., 버퍼 오버플로, 버퍼 오버라이트)은 시스템 공격자에게 유용한 도구로써 사용된다. 시스템 공격자는 메모리 손상을 악용하여 프로그램의 제어 흐름을 결정하는 반환 주소(return address)를 변조하여 프로그램의 제어 흐름을 탈취하는 제어 흐름 하이제킹(control-flow hijacking) 공격을 수행한다.

[0003] 현대 운영 체제(Operating System, OS)에서 기본적으로 사용하는 ASLR(Address Space Layout Randomization) 기술은 가상 메모리의 메모리 세그먼트(e.g., stack, heap, bss)를 프로그램을 로드할 때마다 임의의 주소에 배치한다. 시스템 공격자는 버퍼 오버플로(buffer overflow) 취약점을 악용해 반환 주소를 덮어쓰며, 이 과정에서 공격자가 원하는 주소값으로 변조하는 공격이 빈번하게 발생한다.

[0004] SafeStack은 메모리 격리(memory isolation)와 정보 은닉(information hiding)을 결합하여 스택 메모리의 반환 주소를 보호하는 기법이다. SafeStack은 안전한 객체(safe object)와 안전하지 않은 객체(unsafe object)를 분류하고, 각각 2개의 스택(SafeStack, UnSafeStack)에 저장하여 메모리를 격리한다. 또한, 표준 시스템 방어 메커니즘인 ASLR을 이용하여 가상 주소 공간의 큰 엔트로피에 SafeStack의 위치를 숨긴다. ASLR이 제공하는 정보 은닉 속성을 활용하여 프로그램이 메모리에 로드될 때마다, 스택(stack), 힙(heap) 등의 메모리 세그먼트 주소를 임의로 배치하여 효과적으로 반환 주소에 대한 변조를 방지할 수 있다.

[0005] 레지스터 유출(register spill)은 피호출자-저장(callee-saved) 레지스터를 사용할 때 레지스터 자원의 한계에 의해 발생한다. 레지스터 유출이 발생하면 기존 레지스터에 저장된 값을 스택 메모리에 옮겨 저장한 뒤 필요한 값을 해당 레지스터에 할당한다. 레지스터 유출의 보존 값은 스택 메모리에 저장되기 때문에 가상 메모리를 읽기 및 쓰기 가능한 공격자의 표적이 된다.

[0006] SafeStack 기법은 피호출자-저장 레지스터인 %rsp 레지스터를 SafeStack의 관리 레지스터로 활용한다. 따라서, %rsp 레지스터를 대상으로 레지스터 유출이 발생하면, SafeStack의 주소를 Un-SafeStack에 노출 시킬 수 있다.

선행기술문헌

비특허문헌

[0007] (비특허문헌 0001) PaXTeam: PaX address space layout randomization (ASLR) (2003), http://pax.grsecurity.net/docs/aslr.txt

(비특허문헌 0002) Burow, N., Zhang, X., Payer, M.: SoK: Shining light on shadow stacks. In: Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland). pp. 985-999. San Francisco, CA (May 2019)

(비특허문헌 0003) Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: {Code-Pointer} Integrity. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 147-163. Broomfield, CO (Oct 2014)

(비특허문헌 0004) LLVM: SafeStack (2015), https://clang.llvm.org/docs/SafeStack.html

(비특허문헌 0005) Oikonomopoulos, A., Athanasopoulos, E., Bos, H., Giuffrida, C.: Poking Holes in Information Hiding. In: Proceedings of the 25th USENIX Security Symposium (Security). pp. 121-138. Austin, TX (Aug 2016)

(비특허문헌 0006) Wang, Z., Wu, C., Li, J., Lai, Y., Zhang, X., Hsu, W.C., Cheng, Y.: Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). pp. 143-156. Xi'an, China (Apr 2017)

(비특허문헌 0007) Corbet, J.: x86 NX support (2004), https://lwn.net/%20Articles/87814/

(비특허문헌 0008) Microsoft: Control flow guard (2016), http://msdn.microsoft.com/en-us/library/Dn919635.aspx

발명의 내용

해결하려는 과제

[0008] 본 발명이 이루고자 하는 기술적인 과제는 이중 스택 결리 기술을 이용한 정보 유출 방어 방법을 제공하는 것이다.

과제의 해결 수단

[0009] 본 발명의 일 실시예에 따른 정보 유출 방어 방법은 타겟 프로그램을 실행하는 컴퓨팅 장치에 의해 수행되고, 상기 타겟 프로그램에 대응하는 프로세스의 가상 메모리에 안전 영역(safe region)을 할당하는 단계, 상기 가상 메모리에 미리 정해진 크기를 갖는 적어도 하나의 청크(chunk)를 할당하는 단계, 및 상기 적어도 하나의 청크를 재할당하는 단계를 포함한다.

[0010] 또한, 상기 안전 영역을 할당하는 단계는, 상기 타겟 프로그램이 상기 가상 메모리에 로드될 때, 임의의 주소에 할당할 수 있다.

[0011] 또한, 상기 안전 영역을 할당하는 단계는, 상기 가상 메모리에 숨겨진 영역(hidden region)을 더 할당하고, 상기 안전 영역에 대한 포인터 값을 포함하는 안전 영역 포인터 객체(safe region pointer object)는 상기 숨겨진 영역에 저장될 수 있다.

[0012] 또한, 상기 숨겨진 영역의 주소는 %gs 레지스터에 의해 식별되고, 상기 %gs 레지스터는 archi_prctl() 시스템 호출(system call)에 의해 활성화될 수 있다.

[0013] 또한, 상기 안전 영역에는 상기 프로세스의 반환 주소(return address)가 저장될 수 있다.

[0014] 또한, 상기 안전 영역은, 상기 가상 메모리 내에서, 공유 라이브러리(shared libraries) 아래 주소 공간에 할당될 수 있다.

[0015] 또한, 상기 적어도 하나의 청크는 상기 가상 메모리 내에서 가장 큰 홀(hole) 내에 할당될 수 있다.

[0016] 또한, 상기 적어도 하나의 청크는 제1 청크와 제2 청크를 포함하고, 상기 제1 청크는, 상기 가상 메모리 내에서, 공유 라이브러리 아래 주소 공간 중 상반부에 할당되고, 상기 제2 청크는, 상기 공유 라이브러리 아래 주소 공간 중 하반부에 할당될 수 있다.

[0017] 또한, 상기 적어도 하나의 청크를 재할당하는 단계는, 상기 타겟 프로그램이 실행되는 동안, 상기 적어도 하나의 청크를 주기적으로 재할당할 수 있다.

발명의 효과

[0018] 본 발명의 실시예들에 의할 경우, 스택 격리 기법을 통해, 반환 주소를 변조하고자 하는 공격자로부터 반환 주소를 안전하게 숨길 수 있다.

[0019] 본 발명은 반환 주소가 저장되는 안전 스택 영역을 관리하는 포인터를 이중 스택 구조로 관리하여 레지스터 유출로부터 안전한 스택 격리를 구현할 수 있다.

도면의 간단한 설명

[0020] 본 발명의 상세한 설명에서 인용되는 도면을 보다 충분히 이해하기 위하여 각 도면의 상세한 설명이 제공된다.

도 1은 제안하는 스택 격리 기술을 사용할 때의 가상 메모리 공간(virtual memory space)의 개요를 도시한다.

도 2는 가상 메모리 주소 공간의 레이아웃을 도시한다.

도 3은 제안 기법(Satellite)을 컴파일 프로세스(compilation process) 중에 소프트웨어에 적용하는 방법을 설명하기 위한 도면이다.

도 4는 SPEC CPU2017 벤치마크 응용프로그램에 제안 기법(Satellite), Stack Canary, SafeStack, 및 Shadow Stack 기술을 적용하여 얻은 성능 오버헤드 측정 결과를 도시한다.

도 5는 SPEC CPU2006 벤치마크 응용프로그램에 제안 기법(Satellite), SafeStack, RERANZ, 및 Shadow Stack with Safehidden 기술을 적용하여 얻은 성능 오버헤드 측정 결과를 도시한다.

발명을 실시하기 위한 구체적인 내용

[0021] 본 명세서에 개시되어 있는 본 발명의 개념에 따른 실시예들에 대해서 특정한 구조적 또는 기능적 설명들은 단지 본 발명의 개념에 따른 실시예들을 설명하기 위한 목적으로 예시된 것으로서, 본 발명의 개념에 따른 실시예들은 다양한 형태들로 실시될 수 있으며 본 명세서에 설명된 실시예들에 한정되지 않는다.

[0022] 본 발명의 개념에 따른 실시예들은 다양한 변경들을 가할 수 있고 여러 가지 형태들을 가질 수 있으므로 실시예들을 도면에 예시하고 본 명세서에서 상세하게 설명하고자 한다. 그러나, 이는 본 발명의 개념에 따른 실시예들을 특정한 개시 형태들에 대해 한정하려는 것이 아니며, 본 발명의 사상 및 기술 범위에 포함되는 모든 변경, 균등물, 또는 대체물을 포함한다.

[0023] 제1 또는 제2 등의 용어는 다양한 구성 요소들을 설명하는데 사용될 수 있지만, 상기 구성 요소들은 상기 용어들에 의해 한정되어서는 안 된다. 상기 용어들은 하나의 구성 요소를 다른 구성 요소로부터 구별하는 목적으로만, 예컨대 본 발명의 개념에 따른 권리 범위로부터 벗어나지 않은 채, 제1 구성 요소는 제2 구성 요소로 명명될 수 있고 유사하게 제2 구성 요소는 제1 구성 요소로도 명명될 수 있다.

[0024] 어떤 구성 요소가 다른 구성 요소에 "연결되어" 있다거나 "접속되어" 있다고 언급된 때에는, 그 다른 구성 요소에 직접적으로 연결되어 있거나 또는 접속되어 있을 수도 있지만, 중간에 다른 구성 요소가 존재할 수도 있다고 이해되어야 할 것이다. 반면에, 어떤 구성 요소가 다른 구성 요소에 "직접 연결되어" 있다거나 "직접 접속되어" 있다고 언급된 때에는 중간에 다른 구성 요소가 존재하지 않는 것으로 이해되어야 할 것이다. 구성 요소들 간의 관계를 설명하는 다른 표현들, 즉 "~사이에"와 "바로 ~사이에" 또는 "~에 이웃하는"과 "~에 직접 이웃하는" 등도 마찬가지로 해석되어야 한다.

[0025] 본 명세서에서 사용한 용어는 단지 특정한 실시예를 설명하기 위해 사용된 것으로서, 본 발명을 한정하려는 의도가 아니다. 단수의 표현은 문맥상 명백하게 다르게 뜻하지 않는 한, 복수의 표현을 포함한다. 본 명세서에서, "포함하다" 또는 "가지다" 등의 용어는 본 명세서에 기재된 특징, 숫자, 단계, 동작, 구성 요소, 부분품 또는

이들을 조합한 것이 존재함을 지정하려는 것이지, 하나 또는 그 이상의 다른 특징들이나 숫자, 단계, 동작, 구성 요소, 부분품 또는 이들을 조합한 것들의 존재 또는 부가 가능성을 미리 배제하지 않는 것으로 이해되어야 한다.

[0026] 다르게 정의되지 않는 한, 기술적이거나 과학적인 용어를 포함해서 여기서 사용되는 모든 용어들은 본 발명이 속하는 기술 분야에서 통상의 지식을 가진 자에 의해 일반적으로 이해되는 것과 동일한 의미를 가진다. 일반적으로 사용되는 사전에 정의되어 있는 것과 같은 용어들은 관련 기술의 문맥상 가지는 의미와 일치하는 의미를 갖는 것으로 해석되어야 하며, 본 명세서에서 명백하게 정의하지 않는 한, 이상적이거나 과도하게 형식적인 의미로 해석되지 않는다.

[0027] 이하, 본 명세서에 첨부된 도면들을 참조하여 본 발명의 실시예들을 상세히 설명한다. 그러나, 특허출원의 범위가 이러한 실시예들에 의해 제한되거나 한정되는 것은 아니다. 각 도면에 제시된 동일한 참조 부호는 동일한 부재를 나타낸다.

[0028] 스택 메모리(stack memory)는 소프트웨어 실행에서 중요한 구성 요소이다. 스택이 손상되면, 데이터 유출(data breaches), 시스템 충돌(system crashes), 및 서비스 손실(loss of service) 등과 같은 부정적인 결과로 이어질 수 있다. 또한, 스택에는 반환 주소(return address)와 같은 "제어(control)" 데이터가 포함되어 있기 때문에, 프로세스의 제어 흐름(control flow)이 ROP(Return-Oriented Programming) 및 임의 메모리 쓰기(arbitrary memory writes)에 의해 탈취될 수 있다. 따라서, 스택을 보호하기 위해 다양한 완화 기술이 제안되었다.

[0029] ASLR(Address Space Layout Randomization)은 대표적인 스택 보호 기술로, 메모리 세그먼트를 무작위 가상 메모리 주소에 배치하여 공격자가 대상 데이터나 함수의 메모리 주소를 예측하기 어렵게 만든다. 그러나, 다른 완화 기술들과 마찬가지로, ASLR은 완벽하지 않으며, 구현의 제한을 이용하여 우회될 수 있다.

[0030] ASLR의 약점을 극복하기 위해, 두 가지 연구 방향이 있다: (1) 임의 메모리 주소에서 메모리 세그먼트를 세분화하거나 런타임에서 메모리 세그먼트를 재할당하는(re-allocations) 세밀한(fine-grained) ASLR 기술; 그리고 (2) 메모리 세그먼트에 저장된 객체를 분류하고 분리하여 메모리 손상 취약점(memory corruption vulnerabilities)으로부터 민감한 데이터를 보호하는 스택 격리 기술(stack isolation technique).

[0031] 스택 격리는, 스택 버퍼 오버플로우와 같은 스택 기반의 취약점(stack-based vulnerabilities)으로부터 소프트웨어 시스템을 보호하는 것을 목표로 한다. 이는, 프로그램 스택을 구분된 영역으로 분리하여 무단 접근(unauthorized access)이나 수정(modification)을 방지하여, 성공적인 공격의 위험을 줄인다. 본 발명에서는, 격리된 영역을, 각 격리 기술에 의해 보호되는 특정 유형의 데이터만을 포함하는 "안전 영역(safe region)"으로 정의한다. 예를 들어, SafeStack은 안전 영역에 반환 주소, 레지스터 스필(register spills), 및 변수(variables)를 저장 및/또는 관리한다. 반면에, AG-Stack은 안전 영역에 리턴 주소만 저장한다.

[0032] 스택 격리는 ASLR을 활용할 수도 있다. ASLR을 스택 격리 기술과 함께 사용하면, 안전 영역을 가상 메모리 공간에서 임의의 주소로 로드할 수 있다. 예를 들어, 반환 주소와 같은 민감한 데이터는 거대한 메모리 공간의 임의의 메모리 주소에 저장될 수 있으므로, 안전 영역을 찾기 위해 무차별 대입 공격을 수행하는 공격자는 2^{28} 엔트로피를 극복하여야 한다.

[0033] 스택 격리 기술들 중에서, Shadow Stack은 다양한 컴파일러(예: LLVM, Debian gcc 등)에 대한 보안 확장으로 적용되었다. Shadow Stack은 스택 메모리에 저장된 반환 주소와 같은 제어 데이터를 보호하기 위해 스택의 그림자 복사본(shadow copy)을 생성한다. 초기의 Shadow Stack(naive Shadow Stack)은 스택에 저장된 반환 주소의 무결성(integrity)을 확인하기 위해 Shadow Stack에 저장된 반환 주소와 비교 연산을 수행하였다. Shadow Stack의 비교 연산은 스택의 그림자 복사본에 저장된 반환 주소를 로드할 때마다 무결성 검사를 수행해야 하기 때문에 높은 성능 오버헤드를 가지게 된다.

[0034] 반면에, SafeStack은 CPI의 일부로써, 반환 주소를 보호하기 위한 스택 격리 기술이다. SafeStack은 반환 주소, 레지스터 스필(register spills), 및 로컬 변수를 안전 영역에 저장한다. SafeStack은 세그먼트 레지스터를 사용하여 안전 영역에 접근하고 프로그램을 계기로 이를 사용하는 다른 명령을 생성한다.

[0035] 할당 오라클(Allocation Oracle)은 ASLR 기반 정보 은닉(ASLR-based information hiding)의 엔트로피를 감소시켜, 무차별 대입 공격의 효율성을 향상시키는 공격 기술이다. 이 기술은, 가상 메모리의 모든 가능한 빈 공간의 크기를 결정하는 것이 남은 세그먼트의 레이아웃을 추론하는 것을 단순화한다는 관찰에 의존한다. 예를 들어, 가장 큰 홀의 크기(즉, 가상 메모리의 빈 공간)가 20바이트라면, 24바이트의 할당 요청이 수락되지 않으며 특정

주소에 대한 할당도 지정할 수 없다. 공격자는 할당 오라클을 서버 프로그램에서 찾아 원하는 양의 메모리를 자유롭게 할당할 수 있게 되며, 사용 가능한 홀의 크기를 찾기 위해 할당 시도를 반복한다. 다양한 메모리 할당 시도와 크기를 사용하는 이진 탐색 알고리즘(binary search algorithms)은 효율적으로 모든 홀을 식별할 수 있게 한다. 프로세스 메모리의 홀을 식별할 수 있는 공격자는 각 메모리 세그먼트의 할당 특성을 사용하여 전체 메모리 레이아웃을 획득할 수 있다. 최악의 경우, 가상 메모리의 전체 레이아웃을 평균 12분의 스캔 시간 내에 식별할 수 있으며, 공격자가 목표로 하는 SafeStack의 위치를 서버 프로그램의 메모리 공간에서 식별할 수 있다.

[0036] ASLR 기반 메모리 보호 기술은, 안전 영역의 위치를 노출시키는 정보 공개로 인해 손상될 수 있다. 할당 오라클은, 프로세스 내 메모리 내에서 할당되지 않은 모든 영역(홀)을 식별하는 것을 목표로, 메모리 검색 공격(memory scanning attack)을 수행하기 위해 반복적인 할당 요청을 가능케 하는 프리머티브(privitives)를 활용한다. 앞서 언급한 기술은, 안전 영역을 찾기 위한 무차별 대입 공격에 필요한 2^{28}의 엔트로피를 크게 감소시킨다. 안전 영역의 위치가 노출되면, 공격자는 임의 메모리 쓰기 취약점(arbitrary memory write vulnerabilities)과 같은 메모리 손상 취약점을 활용하여 안전 영역 내의 데이터를 조작할 수 있다. 따라서, 정보 공개로 인해 안전 영역의 위치가 노출되는 것을 방지하는 보호 기술의 사용이 중요하다.

[0037] 안전 영역의 위치를 밝히는 정보 공개 공격(information disclosure attacks)은, 로딩 시간(loading time) 동안에 메모리 세그먼트 시작 주소(memory segment starting addresses)가 결정되는 ASLR 기반 메모리 보호 기법의 특성을 활용한다. 공격자는, 가상 메모리 공간 내에서 로딩 시간에 결정된 메모리 주소를 효율적으로 획득하기 위해, 런타임 중에 메모리 스캐닝 공격을 수행한다. 다음으로, 로딩 시간 동안 설정된 가상 메모리 공간 레이아웃을 분석하여 안전 영역을 식별할 수 있다. 공격자가 런타임 중에 정보 유출 공격에 대한 힌트를 수집하지 못하면, 안전 영역의 위치를 알아낼 수 없다. 정보 공개 공격을 완화하기 위해, 런타임 동안 안전 영역의 위치를 변경하는 재랜덤화 기법(re-randomization techniques)이 제안되었다. 메모리 재랜덤화 기법은 정보 공개 공격자의 논리를 교란하여 안전 영역을 효과적으로 보호한다. 그럼에도 불구하고, 런타임시 메모리 세그먼트를 재배치하는 지속적인 작업은 복잡한 프로그램에 상당한 성능 오버헤드를 부과한다. 본 발명에서는, 프로세스 메모리 내에서 할당된 영역과 할당되지 않은 영역(홀)의 효율적인 검색을 방해하여 안전 영역 위치의 노출을 방지하는 Satellite라는 솔루션을 제안한다. 제안 기법(Satellite)은 런타임 메모리 재랜덤화 기법에 비해 정보 공개 공격을 보다 효율적으로 저지할 수 있다.

[0038] 본 발명에서 고려하는 위협 모델(Threat Model)은, 프로그램의 제어 흐름을 가로채기 위해 프로세스의 스택 메모리를 읽고 쓰려는 공격자를 가정한다. 공격자는 궁극적으로 대상 프로그램의 메모리 손상 취약점을 악용하여 제어 데이터(예: 반환 주소)를 손상시키는 것을 목표로 한다. 또한, 이러한 취약점을 악용하여 공격자는 정보은닉 속성에 의해 숨겨진 반환 주소의 위치를 알아내기 위해 할당 오라클 공격과 같은 정보 공개 공격을 수행할 수 있다고 가정한다. 따라서, 공격자는 대상 프로그램의 스택에 저장된 반환 주소를 조작하여 피해자 프로세스의 제어 흐름을 탈취할 수 있다. 반면에, 하드웨어와 운영 체제는 안정적이므로 레지스터 상태 정보와 같은 시스템 데이터를 악용하여 공격자에게 노출될 수 없다고 가정한다. 또한, 시스템이 코드 주입 공격을 실행할 수 없는 DEP, W⊕X와 같은 표준 완화 기술(standard mitigation techniques)을 사용함을 고려한다.

[0039] 요컨대, 위협 모델은, 공격자가 스택을 부수어 악용할 수 있는 취약한 프로그램이 있고, 공격자는 할당 오라클 공격과 같은 정보 공개 공격을 수행할 수 있지만, 나머지 프로그램에는 알려진 취약점이 없다고 가정한다.

[0040] 본 발명에서는, "Satellite"로 명명된 새로운 스택 격리 기법을 제안한다. 제안하는 스택 격리 기법은 반환 주소를 저장하는 안전 영역(safe region)에 대한 정보 노출 공격(information disclosure attacks)에 강력한 보호를 제공하도록 설계되었다. 구체적으로, AMD64 아키텍처를 위한 제안 기법(Satellite)를 설계하고 구현하였다.

[0041] 제안 기법(Satellite)의 설계 목표는 정보 노출 공격을 수행하는 공격자로부터 안전 영역으로 이끄는 포인터(pointers)에 관련된 정보를 난독화하는(obfuscate) 것이다. 일단 안전 영역으로 이끄는 포인터 값(pointer value)이 노출되면, 공격자들은 이를 변조하여 스택 격리 기술로 보호되는 대상 프로그램(target programs)의 제어 흐름을 하이제킹(hijack)할 수 있다. 스택 포인터 (%rsp)는 C 표준 라이브러리(C standard library)나 C++ 라이브러리에서 예외 처리 구문(exception handling syntax)을 사용할 때 공격자에게 노출될 수 있다. x86 아키텍처의 피호출자 저장 레지스터(callee saved registers, 예: %rbx, %rsp, %rbp, %r12, %r13, %r14, %r15)를 컴파일러 예약 레지스터(compiler-reserved registers)로 사용할 때도, 레지스터 값은 라이브러리 함수(library function)의 호출과 같은 외부 코드를 실행함에 의해 노출될 수 있다. 예컨대, 예외 처리에 사용되는

라이브러리 함수들인 setjmp() 및 longjmp()은, 함수 간 점프를 지원하기 위해 jump buffer라는 객체(object)를 사용한다. 함수 간 점프가 수행되면, 점프 당시의 레지스터 값은 jump buffer에 저장되고, 이 버퍼는 배열 데이터 구조(array data structure)로 스택 메모리(stack memory)에 저장된다. 따라서, 안전성을 위해, 일반 목적 레지스터(general-purpose registers)에 엄격하게 바인딩되지 않도록 안전한 공간(safe spaces)에 안전 영역을 가리키는 포인터 값을 저장하여야 한다.

[0042] 제안 기법(Satellite)은 안전 영역을 가리키는 포인터로 동작하며, %gs 레지스터(세그먼트 레지스터)를 사용한다. 이 레지스터는 x86-32 아키텍처에서 세그먼트화 목적(segmentation purpose)으로 사용될 수 있지만, 64비트 아키텍처에서는 세그먼트화(segmentation)가 일반적으로 비활성화되거나(disabled) 크게 제한되기 때문에 사용되지 않는다. 이는 64비트 주소 모드에서는 주로 플랫한 64비트 선형 주소 공간(flat 64-bit linear address space)을 초래한다. 프로세스에서 %gs 레지스터의 사용을 가능하게 하기 위해서, arch_prctl() 시스템 호출(system call)을 통한 %gs 레지스터의 활성화가 요구된다. LLVM과 같은 현대적인 컴파일러는, 64비트 선형 주소 공간으로 인해, %gs 레지스터를 사용하지 않는다. 따라서 LLVM 및 gcc 컴파일러는 %gs 레지스터를 사용하는 명령을 생성하지 않는다. SPEC CPU2017 벤치마크 프로그램과 해당 벤치마크에서 사용하는 라이브러리에 대한 포괄적인 검토를, LLVM 및 gcc 컴파일러를 이용하여 컴파일링함으로써, 수행하였다. 결과적으로, 모든 바이너리에서, %gs 레지스터 사용의 어떤 인스턴스도 식별되지 않았다.

[0043] 만약 공격자가 %gs 레지스터를 참조하는 명령을 실행하려고 시도한다면, 공격자들은 코드 삽입 공격(code injection attack)에 의존해야 할 것이다. 그러나, 이는 불가능하다. 본 발명에서 제안하는 시스템은 DEP 및 W⊕?X와 같은 전통적인 완화 전략(mitigation strategies)을 사용하기 때문이다. 따라서, AMD64 아키텍처에서 사용되지 않는 %gs 레지스터를 숨겨진 영역(hidden region)을 가리키기 위해 사용한다면(레지스터를 사용하는 명령이 없고 메모리의 다른 영역에서 숨겨진 영역을 가리키는 주소가 없는 한), 이는 정보 노출 공격(information disclosure attacks)에 대해 안전한 상태를 유지할 수 있다.

[0044] 도 1은 제안하는 스택 격리 기술을 사용할 때의 가상 메모리 공간(virtual memory space)의 개요를 도시한다. 제안 기법(Satellite)에서, 메모리 세그먼트(memory segments)는 안전 영역(safe region), 불안전 영역(unsafe region), 및 숨겨진 영역(hidden region)으로 구성된다.

[0045] **불안전 영역(Unsafe region).** 불안전 영역에는, 로컬 변수(local variables), 레지스터 스필(register spills), 및 배열(array)을 포함하는 스택 객체들(stack objects)이 저장된다. 제안 기법(Satellite)에서는, 불안전 영역을 가리키기 위해, %rsp를 사용한다.

[0046] **안전 영역(Safe region).** 안전 영역은 메모리 손상 취약점(memory corruption vulnerabilities)을 이용하려는 공격자들의 주요 타겟인 반환 주소만을 저장한다. 안전 영역에 반환 주소만을 저장함으로써 안전 영역의 크기를 최소화하고, 이에 따라 엔트로피 손실도 최소화할 수 있다. 이는 안전 영역을 정보 노출 공격에 강건하게(robust) 만든다. 또한, 안전 영역에 대한 포인터 값을 숨기기 위해, 일반 목적 레지스터(general purpose registers)나 세그먼트 레지스터(segment registers)를 사용하는 대신 안전 영역 포인터 객체(safe region pointer object)를 사용하고, 이를 숨겨진 영역에 저장한다.

[0047] **숨겨진 영역(Hidden region).** 숨겨진 영역에는 안전 영역 포인터 객체와, 특히 C 및 C++ 예외 처리 라이브러리 함수(exception-handling library functions)의 구문(context)에서 함수 간 점프를 처리하는 데 필요한 메타데이터가 포함되어 있다. 이는 스택 격리 기술과 관련된 충돌을 해결하기 위한 것이다. 메타데이터는 setjmp() 및 longjmp() jmpbuffer 정보, 현재 스택 프레임 깊이(current stack frame depth), 및 임시 레지스터 값(temporary register values)과 같은 데이터를 나타낸다. 이 메타데이터는 함수 간 점프(inter-function jumps, 예: setjmp(), longjmp(), try(), 및 catch())와 관련된 라이브러리 함수와 호환성을 갖추기 위해 설계된 후크 함수(hook functions)에 의해 활용된다. 제안 기법(Satellite)은 %gs 레지스터를 관리하여 이를 숨겨진 영역을 가리키는 포인터로 사용한다.

[0048] 안전 영역과 숨겨진 영역은 가상 메모리 공간 내의 임의의 주소에 위치할 수 있다. 그러므로, AMD64 아키텍처 CPUs에 의해 제공되는 광범위한 주소 공간 때문에 영역의 주소를 추측하는 것은 매우 비현실적이다. 또한, 올바른 주소를 추측하는 데 실패하면 프로그램 충돌이 불가피하게 발생하며, 이러한 충돌이 빈번하게 발생하면 여러 모니터링 또는 디버깅 메커니즘을 통해 쉽게 감지될 수 있다. 결과적으로, 많은 눈에 띄는 프로그램 충돌과 관련된 중대한 리스크로 인해 추측 시도의 실현 가능성과 성공률이 크게 감소한다.

[0049] 안전 영역은, 프로그램이 가상 메모리에 로드될 때, 초기에 임의의 주소에 할당된다. 그러나, 프로그램이 로드

된 후 실행을 시작하면 안전 영역은 고정된 주소에 위치하게 된다. 따라서, 할당 오라클 공격과 같은 공격에 의해 안전 영역의 위치가 노출될 수 있다. 구체적으로, 공격자는 가상 메모리 내의 갭(gaps)이나 홀(holes)을 식별하기 위해 다수의 할당 쿼리(allocation queries)를 전송할 수 있다. 모든 갭이나 홀이 식별되면, 공격자는 안전 영역의 주소와 같은 의미 있는 정보를 가지고 있는 메모리 세그먼트를 찾을 수 있다.

[0050] 제안 기법(Satellite)은 할당 오라클을 활용한 무차별 대입 공격에 대한 런타임 보호장치로 기능할 수 있다. 할당 오라클 공격에서 중요한 요소인 프로세스 메모리(process memory)의 모든 갭(gaps)에 대한 식별을 방지함으로써, 제안 기법(Satellite)은 스택 격리의 보안을 효과적으로 강화할 수 있다. 제안 기법(Satellite)은, 독립적으로 동작하도록 특별히 설계되어 있으며, 정보를 숨기기 위한 다양한 ASLR 기반 보호 기법에 적용될 수 있다.

[0051] 제안 기법(Satellite)은, 런타임 동안에 프로세스의 가상 메모리 공간에 "satellite chunk"로 불리는 메모리의 작은 청크(chunks)를 할당한다. 청크는 미리 정해진 크기를 가질 수 있으며, 적어도 하나의 청크가 할당될 수 있다. 이러한 작은 청크들은 계속해서 무작위 주소로 재할당된다. 효율적인 런타임 보호를 제공하기 위해, 제안 기법(Satellite)은 대상 프로그램의 스레드(thread)의 컨텍스트(context) 내에서 작동한다. 이는, 어플리케이션 런타임(application runtime) 동안 보호 모듈(protection module)을 실행하기 위해, 메인 스레드(main thread)와 함께 Satellite 청크 스레드(satellite chunk thread)를 생성하는 것을 포함한다. 제안 기법(Satellite)으로 보호된 프로세스가 메모리 할당(allocation) 및 해제(deallocation) 작업을 빈번히 수행하는 경우, 상당한 성능 부하가 발생할 수 있다. 런타임에서 홀의 크기(hole size)를 동적으로 변경하는 개념은, 사용 가능한 모든 홀을 발견하는 것을 방지함으로써, 할당 오라클 공격을 방지할 수 있다. 지속적인 할당 오라클 공격이 존재할 때라도, 공격자는 프로그램이 종료될 때까지 무한한 수의 할당 쿼리를 계속해서 보낼 것이다.

[0052] 일반적인 AMD64 프로세서에서 실행되는 시스템은 기본 가상 주소 공간(default virtual address space)에 세 개의 할당되지 않은 영역(홀)을 가진다. 첫 번째 홀(A)은 스택 영역(stack area) 위에 위치하며 높은 주소(high address)를 차지한다. 두 번째 홀(B)은 스택과 힙 영역(heap area) 사이에 위치한다. 마지막으로, 세 번째 홀(C)은 힙(heap)과 공유 라이브러리(shared libraries) 아래 주소 공간에 존재한다. 도 2의 가상 메모리 주소 공간의 레이아웃에 도시된 바와 같이, 홀의 크기는 C > B > A의 순서를 따른다. 예시적으로, 홀 C의 크기는 131,068 GiB, 홀 B는 1,028 GiB이며, 홀 A는 4 GiB이다(다만, ASLR의 구현에 따라 각 실행(run)마다 홀의 크기가 변할 수 있다. 그러나, 본 발명의 목적상 최대 홀 크기를 가정한다). mmap() 시스템 호출(system call)을 통해 설정된 안전 영역은 세 개의 홀 중 가장 큰 홀인 홀 C 내의 임의의 주소로 할당된다. 할당 오라클은 가장 큰 홀이 mmap 공간 내에서 할당된 안전 영역의 위쪽이나 아래쪽에 위치할 것으로 예측하며 공격을 수행한다. 할당 오라클의 초기 예측은, 제안 기법(Satellite)으로 보호된 프로세스의 가상 메모리에 satellite 청크가 런타임 중 할당되면서 효과적으로 홀(C)가 분리되어 무효화될 수 있다.

[0053] 제안 기법(Satellite)은, 가장 큰 홀 C 내에, satellite 청크로 명명된 두 개의 작은 메모리 세그먼트를 할당한다(도 2 참조). 다만, 실시예에 따라, satellite 청크는 단수 개로 구현되거나 2 이상의 복수 개로 구현될 수도 있다. 이러한 satellite 청크는 주기적으로(예: 매 초) 또는 비주기적으로 무작위 주소로 재배치될 수 있다. 일 실시예에 따르면, 할당 오라클 공격에 효과적으로 대응하기 위해, 두 satellite 청크는 주기적으로(또는 비주기적으로) 홀 C의 상반부와 하반부(upper and lower halves)의 영역에 무작위로 배치된다. 이러한 설계는, mmap 공간 내에서 안전 영역이 어디에 할당되든지와 무관하게, satellite 청크가 실제로 가장 큰 홀(actual largest hole)을 분할하여 가짜 가장 큰 홀(decoy largest hole)을 생성하도록 한다. 할당 오라클은 메모리 할당 요청을 단계적으로 축소하여 실제로 가장 큰 홀의 최대 크기를 성공적으로 할당할 때까지 진행된다. 그 결과, 가짜 가장 큰 홀이 할당된다. 할당 오라클은 이 과정을 두 번째로 큰 홀에 대해 반복 수행하려고 하지만, 제안 기법(Satellite)은 가장 큰 홀과 관련된 모든 할당이 사실 가짜 할당으로 처리되도록 보장한다. 이후, 할당 오라클은 다음 실제로 가장 큰 홀을 찾으려고 하지만, 제안 기법(Satellite)은 가장 큰 홀과 관련된 모든 할당이 가짜 할당으로 처리되도록 보장한다. 지속적인 할당 오라클 공격 동안, 공격자는 가짜, 조각난 가장 큰 홀을 다수 할당한다. 제안 기법(Satellite)은 프로세스가 종료될 때까지 가상 메모리 공간의 레이아웃을 그대로 유지함으로써 공격자가 안전 영역의 위치를 식별하는 것을 방해한다.

[0054] 도 3은 제안 기법(Satellite)을 컴파일 프로세스(compilation process) 중에 소프트웨어에 적용하는 방법을 설명하기 위한 도면이다. 통합된 프로그램(integrated program)에 대한 정적 분석(static analysis) 및 코드 재작성(code rewrite)을 수행하기 위해, llvm-link 도구(tool)를 사용하여 모든 비트코드 파일(bitcode files)을 병합하고(merge), LLVM IR(Intermediate Representation, .ll file)을 생성한다. 이 과정을 통해 모든 심볼 정보(symbol information)를 얻을 수 있다. 생성된 LLVM IR의 정적 분석을 통해, 제안 기법(Satellite)을 적용하

는 데 필요한 심볼 목록(list of symbols)을 작성한다. IR Code Rewriter는 코드를 수정하여 프로세스 메모리 세그먼트를 추가하고 제안 기법(Satellite)에 의해 제공되는 함수를 실행(invocation)할 수 있도록 한다. .ll 파일이 .s 파일로 변환되면, Instruction Rewriter를 사용하여 어셈블리 코드를 적용하고(adapt), 스택 격리 정책(stack isolation policy)을 준수하여 기존 스택 메모리 객체를 보존한다. 또한, Exception Reconstructor는 스택 격리 기술의 구현으로 인해 발생할 수 있는, setjmp()/longjmp() 및 try()/catch()의 테일 콜(tail calls)과 같은 부작용(side effects)을 최적화하기 위해 어셈블리 코드를 재구성한다. 마지막으로, 제안 기법(Satellite 라이브러리)를 대상 소프트웨어에 링크한다.

[0055] 스택 격리 기술은, 가상 메모리 공간 내에서 무작위 위치(randomized position)를 할당하기 위해, 프로그램 시작(program startup) 중에 메모리 초기화(memory initialization)를 필요로 한다. 스택 격리 스킴이 어셈블리 코드 수준(assembly code level)에서 구현된 경우, 안전 영역이 메모리에 로드되지 않은 상태에서 반환 주소를 푸시(push)하거나 팝(pop)하려는 시도는 프로그램 충돌(program crash)을 초래한다. 대상 프로그램의 프로세스 생성 단계(process creation phase) 동안, 메모리 레이아웃을 초기화하기 위해, attribute((section(".preinit_array"), used)) 속성을 가진 초기화 함수(initialization function)를 정의한다. 스택 격리를 구현하는 동안 다음과 같은 시스템 호출(system call)을 사용하여 가상 메모리에 메모리 세그먼트를 할당한다: (1) mmap() 시스템 호출은 메모리 매핑 공간(memory mapping space)의 임의의 위치에 숨겨진 영역을 할당하는 데 사용된다. (2) arch_prctl() 시스템 호출은 %gs 세그먼트 레지스터를 할당된 숨겨진 영역에 매핑하는 데 사용된다. 프로세스 생성 시 생성되는 안전 영역 및 안전 영역 포인터는, 안전한 스택 격리를 제공하기 위해, 초기화 중에 안전하게 처리된다.

[0056] 제안 기법은, LLVM 툴체인(LLVM toolchain)의 프론트-엔드(front-end)인 Clang에 의해 소스 코드(soure code)가 IR(intermediate representation) 코드로 변환된 이후에 적용될 수 있다. 제안하는 프로토타입은 추가적인 하드웨어 또는 커널 수정이 필요하지 않으며, 기존 라이브러리(legacy libraries)의 재컴파일(recompilation)도 필요로 하지 않는다. 컴파일 프로세스(compilation process) 중에 제안 기법(Satellite)은 IR 코드와 어셈블리 코드를 다시 쓰는 방식으로 대상 프로그램에 구현된다. 제안 기법(Satellite)의 런타임 메모리 청크 모듈은 링킹 단계(linking phase) 중에 대상 프로그램과 연결되어 실행 중에 동작한다. 런타임 메모리 청크가 생성됨에 따라, 가상 메모리 공간에서 무작위 기본 주소로 설정된다. 또한, 런타임 메모리 청크는 대상 프로그램의 스레드 내에서 작동하며, Satellite 기법(Satellitetechnique)의 초기화는 메인 스레드에서 실행되는 main() 함수 내에서 필요로 한다. 제안 기법에서는, Satellites_init()을 매개변수로 사용하여 워커 스레드(worker thread)를 생성하여 Satellite 함수(function)를 실행하도록 pthread_create()를 사용하였다. 이 워커 스레드는 프로세스가 종료될 때까지 Satellite 함수(function)를 수행한다.

[0057] 제안 기법(Satellite)의 런타임 메모리 청크는 프로그램의 원본 소스 코드(original source code)와 독립적인 별도의 스레드(distinct thread)로 동작한다. 이 기술은 대상 프로그램의 main() 함수에서 pthread_create()를 호출함으로써 생성된 스레드를 유지하며, 프로그램 호환성에 중대한 방해를 일으키지 않는다. 메인 스레드에 의해 생성된 스레드에 의해 실행되는 Satellite 함수(function)는, 프로세스가 종료될 때까지 무한 루프(while(1))를 통해 반복한다. Satellite 스레드는 unmmap() 및 mmap() 함수를 사용하여 기존(existing) Satellite을 해제하고(deallocate) 임의의 영역에 다시 할당한다. 이후에는, usleep()을 이용하여 Satellite의 할당 주기를 조절할 수 있다. 따라서, Satellite의 양(quantity)과 빈도(frequency)는 보안 요구사항 및 시스템 자원에 기반하여 조절될 수 있다. 본 발명에서는, Satellite가 가장 간단한 형태로 가정되어, 두 개의 런타임 메모리 청크와 1초의 주기로 구성된다. 즉, 생성되는 메모리 청크의 개수 적어도 하나일 수 있으며 실시예에 따라 가변될 수 있다. 또한, 재할당의 주기 또한 실시예에 따라 가변할 수 있다.

[0058] 제안 기술에서는, throw() 및 catch() 동작(operations)을 관리하기 위한 동기화 함수(function)가 어셈블리 코드 수준에 삽입되며, 레지스터 값과 관련된 문제를 발생시킬 수 있다. 예를 들어, 이는 catch() 직후에 호출되는 동기화 함수 내에서 %rax, %rcx 및 %rdx와 같은 호출자 저장 레지스터(caller-saved registers)의 손상을 일으킬 수 있다. 이 문제를 완화하기 위해, 동기화 함수 내에서 호출자 저장 레지스터의 값은 숨겨진 영역이라 불리는 기술을 사용하여 보존된다. 저장된 값은 동기화 함수(synchronization function)의 완료시에 복원된다.

[0059] 이하에서는, 제안 기법(Satellite)의 보안 특성(security properties)을 조사하여 제안 기법의 효과를 설명한다. 더 나아가서, 계산 집약적인 벤치마크(computationally intensive benchmarks) 및 실제 응용 프로그램(real-world applications)에 제안 기법을 적용하여, 스택 보호 기술 및 런타임 재랜덤화 기술(runtime re-randomization techniques)과의 비교를 통해 그 효과를 시연한다.

[0060] Intel i9-13900K @ 5.80GHz CPU (64GB RAM 및 32 코어)에서 제안 기법(Satellite)을 구현하고 평가하였다. 또한, Ubuntu 22.04 LTS 운영 체제에서 5.15.0 커널 버전으로 빌드된 LLVM 버전 16.0.0 컴파일러를 사용하였다. 종합적인 평가를 위해, SPEC CPU2006, SPEC CPU2017, Nginx 웹 서버, 및 ProFTPD FTP 서버를 포함하는 다양한 벤치마크에 제안 기법(Satellite)을 적용하였다.

[0061] 할당 오라클은 런타임 프로세스 메모리의 모든 세그먼트와 홀 크기를 찾아내는 강력한 런타임 공격 기술이다. 노출된 홀 크기를 사용하여, 공격자는 전체 메모리 레이아웃을 효율적으로 획득할 수 있으며, 정보 숨김 속성(information-hiding property)에 의해 획득되는 엔트로피는 크게 감소한다. 할당 오라클은 서버 프로그램의 취약점을 이용하여 메모리 할당(allocation) 및 해제(deallocation)를 제어하는 오라클을 찾은 다음 메모리 스캔 공격(memory-scanning attack)을 실행한다. 그 후 공격자는 할당 크기를 제어하여, 가장 큰 홀을 찾기 위하여, 큰 크기에서 작은 크기의 홀에 대한 조사를 시도한다. 할당 시도(allocation attempts)는 이진 검색 알고리즘을 사용하여 효율적인 시간 복잡도로 홀을 채우게(fill) 된다. 모든 홀이 채워지면, 공격자는 모든 홀 크기를 얻고, 이 정보를 사용하여 전체 메모리 구조와 안전 영역의 위치를 알아낸다. 위와 같은 공격에 대응하기 위해, 제안 기법(Satellite)은 정기적인 간격(regular intervals)으로(또는 비정기적인 간격으로) 임의의 위치에 할당된 메모리 영역을 사용하여 홀의 크기를 빈번하게 변경한다.

[0062] 실험에서는, 제안 기법(Satellite)에 의해 보호된 Nginx 서버에서 "안전 영역"을 찾기 위해 할당 오라클 접근법을 사용하였다. 제안 기법(Satellite)을 적용하지 않은 경우, 공격자는 71번의 할당 시도만으로 안전 영역을 찾을 수 있었다. 할당 오라클의 이진 검색은, 할당되지 않은 메모리 영역(홀)의 크기가 변하지 않는 것을 가정하고, 가장 큰 홀부터 가장 작은 홀까지 순서대로 식별한다. 그러나, 제안 기법(Satellite)에서의 주기적인(또는 비주기적인) 홀 크기 변경은 공격 알고리즘을 성공적으로 무력화시킨다. 할당 시도가 계속되면, 공격자는 할당 시간에 잘못된 가장 큰 홀을 식별하게 되며, 결과적으로 무수히 많은 가짜 홀을 식별하게 된다.

[0063] SPEC CPU2017 벤치마크 슈트(benchma가 suite)와 함께 네 가지 보호 기법(Satellite, Stack Canary, SafeStack, Shadow Stack)을 사용하여 성능 오버헤드를 측정하였다. SPEC CPU2017에서, 각 벤치마크 프로그램을 100번 실행하고 평균을 계산하였다.

[0064] 실제 응용 프로그램 평가(Real-world applications Evaluation). 제안 기법의 실제 적용 가능성을 검증하기 위해, 인기 있는 웹 서버인 NginX의 1.16.1 버전을 사용하였다. 실험에서는, 오픈 소스 웹 서버 프로그램인 NginX에 제안 기법(Satellite)을 구현하고, 기본 서버 프로그램과의 결과를 비교하였다. 10,000개의 요청을 NginX 웹 서버에 보내어 각 요청이 완료되는 데 걸리는 시간을 측정하는 실험을 진행하였다. 특히, 배치 사이즈 10으로(즉, 10개씩 묶어서) 1,000번 반복하여 10,000개의 요청을 처리하는 데 걸리는 시간을 측정하였다. 결과적으로, 평균 오버헤드는 0.46%로, 실제 웹 서버에 대한 제안 기법의 효과적인 보호 능력을 나타낸다. 반면에, 이전에 제안된 스택 격리 접근 방식인 SafeStack은 평균 0.18%의 성능 오버헤드를 부과하였고, 재랜덤화 접근법(re-randomization approach)인 Safehidden은 Shadow Stack에 적용될 때 평균 5.35%의 성능 오버헤드를 부과하였다.

[0065] 더불어, 보안과 기능(functionality)에 중점을 둔 FTP 서버로 유명한 ProFTPD에도 제안하는 프로토타입을 적용하였다. 제안 기법(Satellite)은 최대 100명의 동시 사용자를 지원하는 ProFTPD 버전 1.3.5에 구현되었다. 실험에서는, 배포된 FTP 서버에 1킬로바이트 전송을 500회 반복하였다. 이로 인해, 제안 기법(Satellite)은 ProFTPD FTP 서버에서 평균 성능 오버헤드가 0.33%로 나타나며, 실제 제품에서도 제안 기술의 효율성을 보여준다.

[0066] 스택 보호 기술에 대한 평가(Evaluation with Stack Protection Techniques) 스택 격리의 효과를 평가하기 위해 SPEC CPU2017 벤치마크 슈트를 사용하여 제안 기법을 대표적인 스택 보호 방법과 비교하였다. 비교 대상에는 Stack Canary, SafeStack 및 Shadow Stack과 같은 대표적인 스택 보호 방법이 포함되었다. Stack Canary 기술 및 SafeStack 기술은 LLVM 컴파일러 프레임워크에 통합된 버전을 사용하여 구현되었다(with the use of *-fstack-protector* and *-fsanitize=safe-stack* flags). Shadow Stack은 Burow 등에 의해 제안된 유형 중 간결한 Shadow Stack 체계를 채용하였다. Shadow Stack 기술은 평균 런타임 오버헤드가 6.29%이며, 이에 비해 Satellite 기술은 낮은 평균 성능 오버헤드가 0.23%를 유지한다. 또한, 제안 기술은 성능 오버헤드가 거의 없다고 알려진 Stack Canary 기술 및 실제 컴파일러의 보안 확장으로 제공되는 SafeStack과 유사한 성능 오버헤드를 가지고 있어 효율적이다. 특히, 2023 Pwn2Own 대회에서는 NETGEAR의 RAX30 라우터에 적용된 Stack Canary가 다섯 가지 취약점으로 손쉽게 우회되었다. 그러나, 제안 기법(Satellite)은 반환 주소가 스택 격리를 통해 보호된 영역 내에서 안전하게 저장되고 관리되기 때문에 Stack Canary의 보안에 의존하지 않는다.

[0067] 재랜덤화 기술에 대한 평가(Evaluation with Re-randomization Technique). 더 나아가, 다른 보호 기술과 비교

하여 정보 노출 공격에 대한 효과를 평가하기 위해 재랜덤화 기술(ref)과 제안 기술을 비교 분석한다. 이를 위해, Satellite, SafeStack, RERANZ, 및 Safe-hidden을 구비한 Shadow Stack의 SPEC CPU2006 벤치마크에서의 성능 오버헤드를 측정하고 비교한다(도 5 참조). 제안 기법(Satellite)은 평균 성능 오버헤드가 0.18%로, 이는 SafeStack의 성능 측정 결과인 0.90%와 근접한다. RERANZ는 평균 성능 오버헤드가 5.93%이며, Shadow Stack에 따라 구현된 Safehidden은 평균 성능 오버헤드가 12.49%를 기록한다. 5.93%에서 12.49%까지의 추가 성능 오버헤드는 풍부한 자원을 갖춘 시스템에서는 허용 가능할 수 있다. 그러나, 현실 제품과 관련된 성능에 민감한 상황(performance-sensitive situations)에서는 제안 기법이 상당한 이점을 가진다.

[0068] 메모리 오버헤드(Memory Overhead). 제안 기법(Satellite)은 메모리 매핑 공간 내에서 추가 메모리(extra memory)를 할당하고 활용하며, 동시에 IR 코드와 어셈블리 코드를 수정하여 추가적인 메모리 오버헤드를 일으킨다. 본 발명에서는, 제안 기법(Satellite)의 메모리 오버헤드를 측정하기 위해 RSS(Resident Set Size)를 사용하며, 이는 측정 중에 프로세스에 의해 실제로 사용된 물리적 메모리 양을 나타낸다. 메모리 오버헤드를 정량화하기 위해, SPEC CPU2006 벤치마크 프로그램 각각을 실행하고 0.0001초 간격으로 측정된 RSS 값의 평균을 계산하였다. 제안 기법(Satellite)의 평균 메모리 오버헤드는 0.56%로, SafeStack의 0.21%와 비교 가능하며, 프로세스가 사용하는 총 메모리 양에 비해 미미하므로 사실상 무시할 수 있다. 더 나아가, 재랜덤화 기술에 대한 비교에 있어서, RERANZ는 평균 메모리 오버헤드가 130.97%이지만, 제안 기술은 메모리 오버헤드 측면에서 매우 효율적임을 입증하였다(표 1 참조). 표 1은 SPEC CPU2006 벤치마크 응용프로그램에 Satellite, SafeStack, 및 RERANZ 기술을 적용하여 얻은 메모리 오버헤드 측정 결과를 나타낸다.

[0069] [표 1]

Benchmarks	Baseline RSS (KB)	SafeStack RSS (KB)	SafeStack Overhead (%)	Satellite RSS (KB)	Satellite Overhead (%)	RERANZ RSS (KB)	RERANZ Overhead (%)
400.perlbench	311,859	313,465	**0.51 %**	312,110	**0.08 %**	380,289	**21.94 %**
401.bzip2	692,778	693,365	**0.08 %**	694,708	**0.28 %**	728,808	**5.20 %**
403.gcc	246,235	245,015	**-0.50 %**	246,915	**0.28 %**	366,695	**48.92 %**
429.mcf	1,664,801	1,663,285	**-0.09 %**	1,666,201	**0.08 %**	1,700,061	**2.12 %**
433.milc	650,795	651,172	**0.06 %**	652,726	**0.30 %**	694,125	**6.66 %**
444.namd	48,351	48,499	**0.31 %**	48,500	**0.31 %**	106,941	**121.18 %**
450.soplex	246,608	247,683	**0.44 %**	245,830	**-0.32 %**	311,778	**26.43 %**
453.povray	7,450	7,649	**2.67 %**	7,853	**5.40 %**	81,310	**991.41 %**
456.hmmer	11,201	11,232	**0.27 %**	11,437	**2.10 %**	58,641	**423.52 %**
458.sjeng	175,898	176,348	**0.26 %**	175,836	**-0.03 %**	213,308	**21.27 %**
462.libquantum	65,857	65,862	**0.01 %**	65,862	**0.01 %**	108,107	**64.15 %**
464.h264ref	56,446	56,116	**-0.58 %**	56,450	**0.01 %**	105,876	**87.57 %**
470.lbm	410,485	410,388	**-0.02 %**	410,480	**-0.01 %**	452,135	**10.15 %**
471.omnetpp	169,246	169,514	**0.16 %**	169,561	**0.19 %**	239,686	**41.62 %**
473.astar	127,445	126,568	**-0.69 %**	126,420	**-0.80 %**	182,295	**43.04 %**
482.sphinx3	42,276	42,267	**-0.02 %**	42,452	**0.42 %**	87,326	**106.56 %**
		Average	**0.21 %**	**Average**	**0.56 %**	**Average**	**130.97 %**

[0070]

[0071] 알고리즘 1(Algorithm 1)은 대상 프로세스(target process)의 가상 메모리에서 가장 큰 홀을 식별하는 데 사용되는 할당 오라클 알고리즘이다. 알고리즘 1의 DEDUCE 함수는 이진 검색 알고리즘을 사용하여 가장 큰 홀의 크기를 결정하기 위해 재귀적으로 호출된다. DEDUCE 함수는 검색 대상 크기의 하한과 상한으로서의 low 및 high 매개변수를 활용한다. 이러한 경계를 사용하여 반복적으로 검색 범위를 좁혀가며 알고리즘은 원하는 값을 찾는다.

[0072] [알고리즘 1]

Algorithm 1: Base EAP-only algorithm

```
Input: A range defined by the lower bound low and the upper bound high
Output: The value being searched for within the range
Function DEDUCE (low, high):
    if low = high then
        return low;
    end
    if high - low = 1 then
        res ← TEMP-ALLOC(high);
        if SUCCESS(res) then
            return high;
        end
        else
            return low;
        end
    end
    midpoint ← ⌊(high + low)/2⌋ ;
    res ← TEMP-ALLOC(midpoint);
    if SUCCESS(res) then
        return DEDUCE (midpoint, high);
    end
    else
        return DEDUCE (low, midpoint − 1);
    end
```

[0073]

[0074] 알고리즘 2(Algorithm 2)는 제안 기법(Satellite)을 사용하여 가장 큰 홀을 찾지 못하도록 설계되었다. TEMP-ALLOC()을 사용하여 메모리 할당 요청을 수행할 때, 런타임 메모리 청크의 존재로 인해 홀 크기 정보가 잘못 계산될 수 있다. 결과적으로, DEDUCE 함수가 반환하는 가장 큰 홀 크기는 무의미하게 작은 값이 되어 의미가 없어진다. 할당 오라클은 DEDUCE 함수가 잘못 반환한 가장 큰 홀 크기를 영구적인 메모리 할당의 인자로 사용한다. DEDUCE 함수는 모든 홀의 크기를 결정할 때까지 반복적으로 호출된다. 제안 기법(Satellite)의 존재로 인해 할당 오라클은 계속해서 잘못된 크기로 메모리를 할당하고 채우기 때문에 효과적으로 모든 홀을 메모리로 채우지 못한다.

[0075] [알고리즘 2]

Algorithm 2: Algorithm where Satellite is applied to Algorithm 1 to find the wrong hole

```
Input: A range defined by the lower bound low and the upper bound high
Output: The value being searched for within the range
Function DEDUCE (low, high):
    if low = high then
        return low;
    end
    if high - low = 1 then
        res ← TEMP-ALLOC(min(high, available_space));
        if SUCCESS(res) then
            return high;
        end
        else
            return low;
        end
    end
    midpoint ← ⌊(high + low)/2⌋ ;
    res ← TEMP-ALLOC(midpoint);
    if SUCCESS(res) then
        return DEDUCE (midpoint, min(high, available_space));
    end
    else
        return DEDUCE (low, midpoint − 1);
    end
```

[0076]

[0077] 이상에서 설명된 장치는 하드웨어 구성 요소, 소프트웨어 구성 요소, 및/또는 하드웨어 구성 요소 및 소프트웨어 구성 요소의 집합으로 구현될 수 있다. 예를 들어, 실시예들에서 설명된 장치 및 구성 요소는, 예를 들어, 프로세서, 콘트롤러, ALU(Arithmetic Logic Unit), 디지털 신호 프로세서(Digital Signal Processor), 마이크로컴퓨터, FPA(Field Programmable array), PLU(Programmable Logic Unit), 마이크로프로세서, 또는 명령(instruction)을 실행하고 응답할 수 있는 다른 어떠한 장치와 같이, 하나 이상의 범용 컴퓨터 또는 특수 목적 컴퓨터를 이용하여 구현될 수 있다. 처리 장치는 운영 체제(Operation System, OS) 및 상기 운영 체제 상에서 수행되는 하나 이상의 소프트웨어 애플리케이션을 수행할 수 있다. 또한, 처리 장치는 소프트웨어의 실행에 응답하여, 데이터를 접근, 저장, 조작, 처리 및 생성할 수도 있다. 이해의 편의를 위하여, 처리 장치는 하나가 사용되는 것으로 설명된 경우도 있지만, 해당 기술 분야에서 통상의 지식을 가진 자는, 처리 장치가 복수 개의 처리 요소(Processing Element) 및/또는 복수 유형의 처리 요소를 포함할 수 있음을 알 수 있다. 예를 들어, 처리 장치는 복수 개의 프로세서 또는 하나의 프로세서 및 하나의 콘트롤러를 포함할 수 있다. 또한, 병렬 프로세서(Parallel Processor)와 같은, 다른 처리 구성(Processing Configuration)도 가능하다.

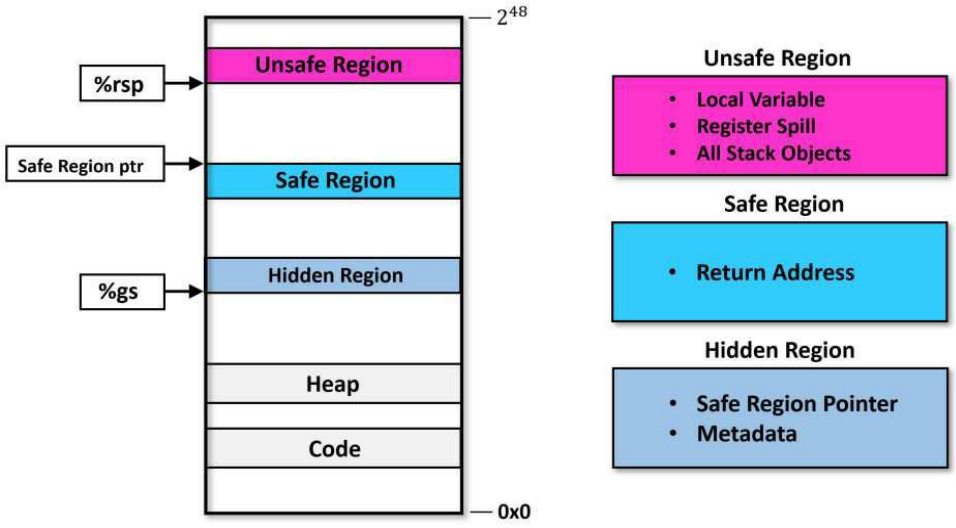
[0078] 소프트웨어는 컴퓨터 프로그램(Computer Program), 코드(Code), 명령(Instruction), 또는 이들 중 하나 이상의 조합을 포함할 수 있으며, 원하는 대로 동작하도록 처리 장치를 구성하거나 독립적으로 또는 결합적으로(Collectively) 처리 장치를 명령할 수 있다. 소프트웨어 및/또는 데이터는, 처리 장치에 의하여 해석되거나 처리 장치에 명령 또는 데이터를 제공하기 위하여, 어떤 유형의 기계, 구성 요소(Component), 물리적 장치, 가상 장치(Virtual Equipment), 컴퓨터 저장 매체 또는 장치, 또는 전송되는 신호 파(Signal Wave)에 영구적으로, 또는 일시적으로 구체화(Embody)될 수 있다. 소프트웨어는 네트워크로 연결된 컴퓨터 시스템 상에 분산되어서, 분산된 방법으로 저장되거나 실행될 수도 있다. 소프트웨어 및 데이터는 하나 이상의 컴퓨터 판독 가능 기록 매체에 저장될 수 있다.

[0079] 실시예에 따른 방법은 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어 컴퓨터 판독 가능 매체에 기록될 수 있다. 상기 컴퓨터 판독 가능 매체는 프로그램 명령, 데이터 파일, 데이터 구조 등을 단독으로 또는 조합하여 포함할 수 있다. 상기 매체에 기록되는 프로그램 명령은 실시예를 위하여 특별히 설계되고 구성된 것들이거나 컴퓨터 소프트웨어 당업자에게 공지되어 사용 가능한 것일 수도 있다. 컴퓨터 판독 가능 기록 매체의 예에는 하드 디스크, 플로피 디스크 및 자기 테이프와 같은 자기 매체(Magnetic Media), CD-ROM, DVD와 같은 광기록 매체(Optical Media), 플롭티컬 디스크(Floptical Disk)와 같은 자기-광 매체(Magneto-optical Media), 롬(ROM), 램(RAM), 플래시 메모리 등과 같은 프로그램 명령을 저장하고 수행하도록 특별히 구성된 하드웨어 장치가 포함된다. 프로그램 명령의 예에는 컴파일러에 의해 만들어지는 것과 같은 기계어 코드뿐만 아니라 인터프리터 등을 사용해서 컴퓨터에 의해서 실행될 수 있는 고급 언어 코드를 포함한다. 상기된 하드웨어 장치는 실시예의 동작을 수행하기 위해 하나 이상의 소프트웨어 모듈로서 작동하도록 구성될 수 있으며, 그 역도 마찬가지이다.

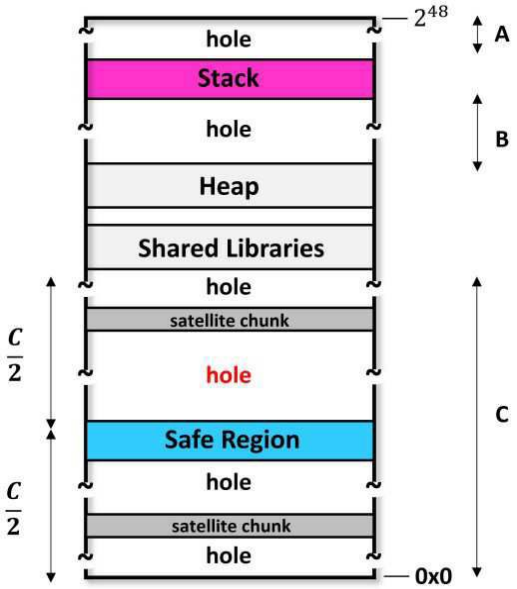
[0080] 본 발명은 도면에 도시된 실시예를 참고로 설명되었으나 이는 예시적인 것에 불과하며, 본 기술 분야의 통상의 지식을 가진 자라면 이로부터 다양한 변형 및 균등한 타 실시예가 가능하다는 점을 이해할 것이다. 예를 들어, 설명된 기술들이 설명된 방법과 다른 순서로 수행되거나, 및/또는 설명된 시스템, 구조, 장치, 회로 등의 구성 요소들이 설명된 방법과 다른 형태로 결합 또는 조합되거나, 다른 구성 요소 또는 균등물에 의하여 대치되거나 치환되더라도 적절한 결과가 달성될 수 있다. 따라서, 본 발명의 진정한 기술적 보호 범위는 첨부된 등록청구범위의 기술적 사상에 의해 정해져야 할 것이다.

도면

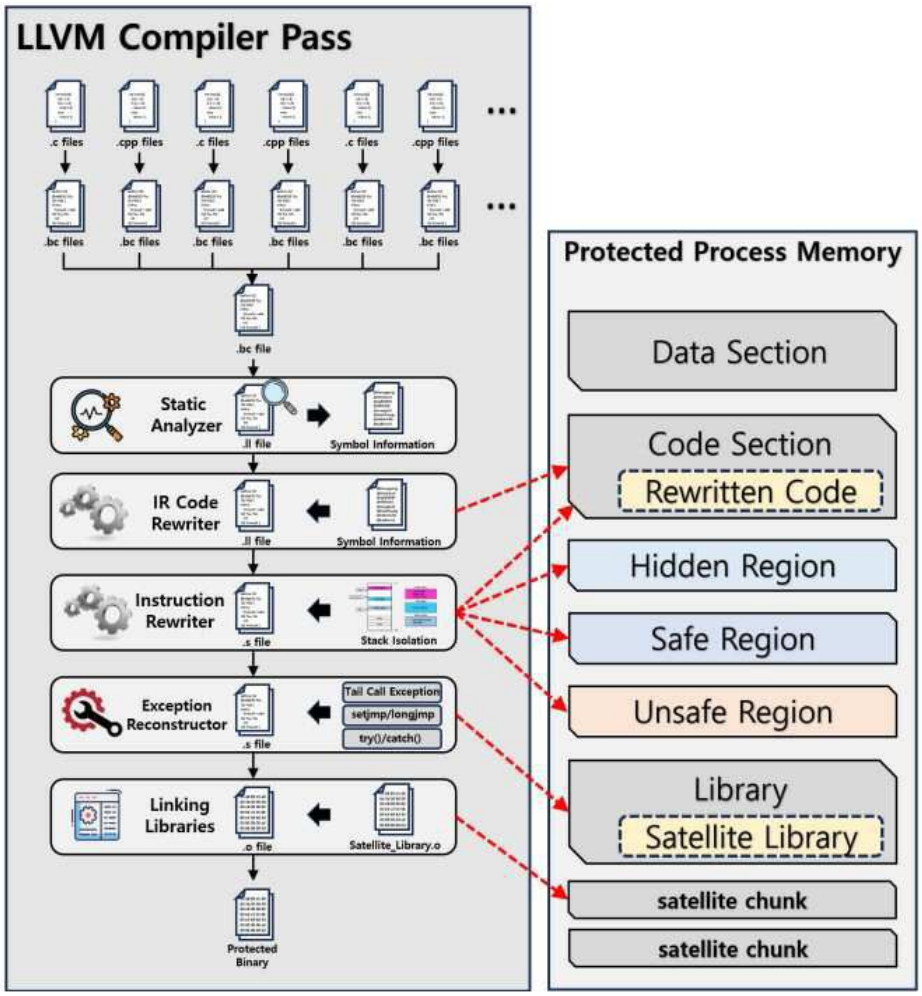
도면1


2^{48}
%rsp
Unsafe Region
Safe Region ptr
Safe Region
%gs
Hidden Region
Heap
Code
0x0
Unsafe Region
• Local Variable
• Register Spill
• All Stack Objects
Safe Region
• Return Address
Hidden Region
• Safe Region Pointer
• Metadata

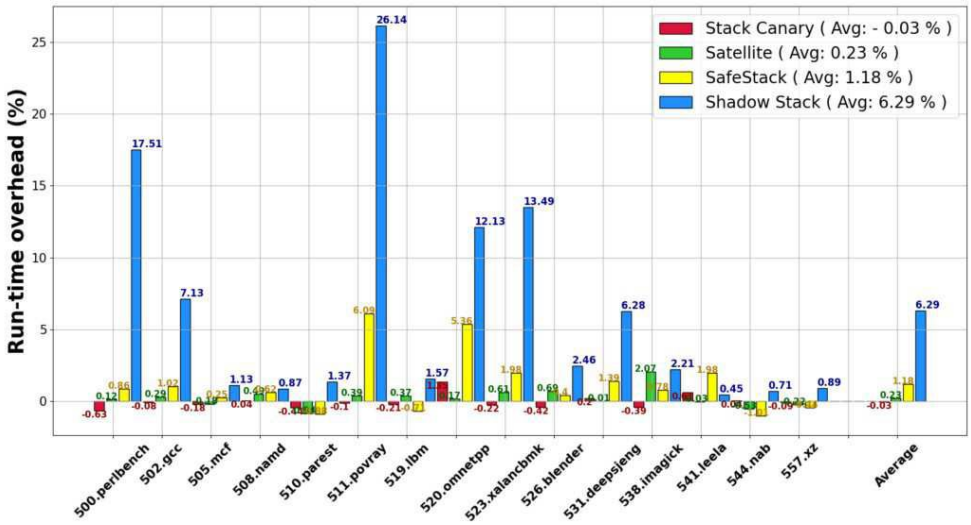
도면2


2^{48}
hole
A
Stack
hole
B
Heap
Shared Libraries
hole
satellite chunk
$\frac{C}{2}$
hole
C
Safe Region
hole
$\frac{C}{2}$
satellite chunk
hole
0x0

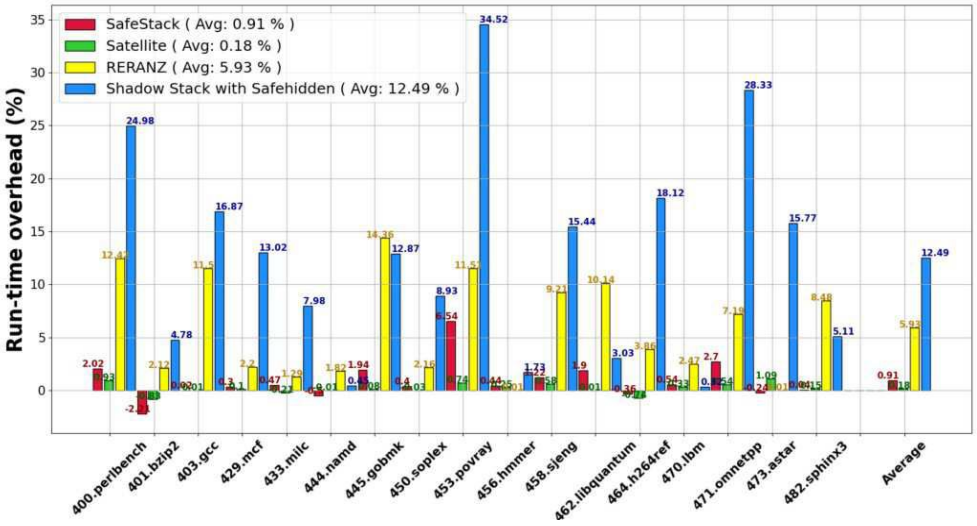
도면3


LLVM Compiler Pass
.c files
.cpp files
.bc files
.bc file
Static Analyzer
.ll file
Symbol Information
IR Code Rewriter
Instruction Rewriter
.s file
Stack Isolation
Exception Reconstructor
Tail Call Exception
setjmp/longjmp
try()/catch()
Linking Libraries
.o file
Satellite_Library.o
Protected Binary
Protected Process Memory
Data Section
Code Section
Rewritten Code
Hidden Region
Safe Region
Unsafe Region
Library
Satellite Library
satellite chunk
satellite chunk

도면4


Stack Canary (Avg: - 0.03 %)
Satellite (Avg: 0.23 %)
SafeStack (Avg: 1.18 %)
Shadow Stack (Avg: 6.29 %)
Run-time overhead (%)
500.perlbench
502.gcc
505.mcf
508.namd
510.parest
511.povray
519.lbm
520.omnetpp
523.xalancbmk
526.blender
531.deepsjeng
538.imagick
541.leela
544.nab
557.xz
Average

도면5


SafeStack (Avg: 0.91 %)
Satellite (Avg: 0.18 %)
RERANZ (Avg: 5.93 %)
Shadow Stack with Safehidden (Avg: 12.49 %)
Run-time overhead (%)
400.perlbench
401.bzip2
403.gcc
429.mcf
433.milc
444.namd
445.gobmk
450.soplex
453.povray
456.hmmer
458.sjeng
462.libquantum
464.h264ref
470.lbm
471.omnetpp
473.astar
482.sphinx3
Average