(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2013-0018642
(43) 공개일자 2013년02월25일

(51) 국제특허분류(Int. Cl.)
G06F 21/12 (2013.01) *G06F 21/52* (2013.01)
G06F 9/455 (2006.01)
(21) 출원번호 10-2012-7015162
(22) 출원일자(국제) 2010년11월12일
심사청구일자 없음
(85) 번역문제출일자 2012년06월12일
(86) 국제출원번호 PCT/CA2010/001761
(87) 국제공개번호 WO 2011/057393
국제공개일자 2011년05월19일
(30) 우선권주장
61/260,887 2009년11월13일 미국(US)

(71) 출원인
어데토 캐나다 코포레이션
캐나다 케이2케이 3쥐5 온타리오 오타와 스위트 300 2500 솔란트 로드
(72) 발명자
구, 유안 시앙
캐나다 케이2티 1지5 온타리오 카나타 39 인스밀 크레센트
아담스, 가니
캐나다 케이2에스 2에이치6 온타리오 스티츠빌 304 업컨츄리 드라이브
롱, 잭
캐나다 케이2더블유 0에이5 온타리오 카나타 179 윈댄스 크레센트
(74) 대리인
박영우

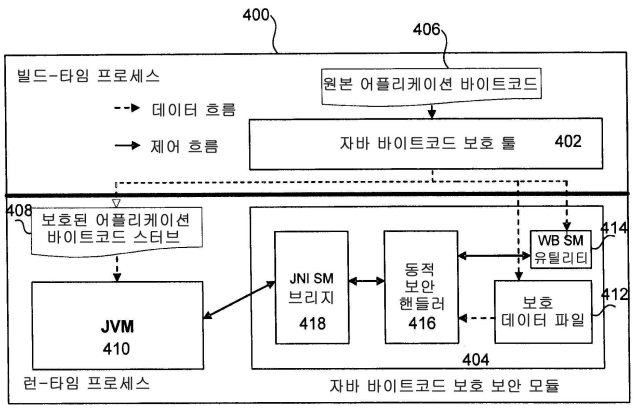
전체 청구항 수 : 총 12 항

(54) 발명의 명칭 **적대적 실행 환경 내에서 정적 및 동적 공격들에 대하여 자바 바이트코드 코드를 보호하는 시스템 및 방법**

(57) 요 약

실행 시간 동안에 자바 플랫폼의 약점을 언급할 수 있고 자바 바이트코드를 보호할 수 있는 보안 모듈을 제공하는 방법과 시스템이 개시된다. 상기 보안 모듈은 예를 들어 C/C++로 구현될 수 있다. 상기 보안 모듈이 C/C++로 구현될 수 있기 때문에, 이에 의하여 C/C++ 소프트웨어 코드를 보호하는 보안 기술을 사용할 수 있다.

대 표 도 - 도4



특허청구의 범위

청구항 1

자바 바이트코드의 탬퍼-저항(tamper-resistance)을 증가시키는 장치로서,

빌드-타임(bulid-time) 동안에 자바 바이트코드를 보안시키는 보호 툴;

상기 보호 툴로부터 상기 보안된 자바 바이트코드를 수신하여 런-타임 동안에 상기 보안된 자바 바이트코드를 론칭(launching)하는 보안 모듈; 및

상기 보안 툴과 상기 보호 모듈과 집적되는 하나 이상의 보호 메커니즘들을 구비하고, 상기 하나 이상의 보호 메커니즘들은 상기 자바 바이트코드에 대한 정적 및 동적 공격들에 대항하여 동작하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 2

제1항에 있어서, 상기 보안된 자바 바이트코드는 보호된 자바 어플리케이션 바이트코드 스터브, 보호된 어플리케이션 페이로드 및 암호화된 클래스 바이트코드 프레임을 포함하고, 상기 자바 어플리케이션 바이트코드 스터브, 상기 보호된 어플리케이션 페이로드 및 상기 암호화된 클래스 바이트코드 프레임은 상기 빌드-타임 동안에 상기 보호 툴에 의하여 형성되는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 3

제2항에 있어서, 상기 보안 모듈은 자바 가상 머신에 대한 기능적 확장으로서 독립적으로 분포되어 보호된 자바 어플리케이션들의 신뢰성의 루트를 제공하고, 상기 보안된 자바 바이트코드는 사용자의 필요에 따라 개별적으로 분포되는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 4

제3항에 있어서, 상기 보호 툴은 상기 보호된 어플리케이션 페이로드가 상기 보호된 자바 어플리케이션 바이트코드 스터브를 통하여 론칭되도록 명령하는 메커니즘을 포함하는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 5

제3항에 있어서, 상기 보안 모듈은 보호된 바이트코드 클래스 로더를 포함하고, 상기 보호 툴은 상기 보호된 어플리케이션 페이로드가 상기 보호된 바이트코드 클래스 로더를 이용하여 상기 암호화된 클래스 바이트코드 프레임을 통하여 론칭되도록 명령하는 메커니즘을 포함하는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 6

제1항에 있어서, 상기 자바 바이트코드의 탬퍼-저항을 증가시키는 장치는 C, C++ 및 자바중 하나 이상을 포함하는 프로그래밍 언어로 구현된 프로그램밍 엔진으로 형성되는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 7

제1항에 있어서, 상기 자바 바이트코드의 탬퍼-저항을 증가시키는 장치는 자바 가상 머신과 인터페이싱할 수 있는 프로그래밍 언어로 구현된 프로그램밍 엔진으로 형성되는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 8

제1항에 있어서, 상기 하나 이상의 보호 메커니즘들은 구성 옵션에 따라 선택가능한 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 9

제8항에 있어서, 상기 하나 이상의 보호 메커니즘들은 상기 보호 툴 내부에 형성된 정적 보안 핸들러들과 상기 보안 모듈에서 형성되는 동적 보안 핸들러들을 포함하는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 10

제9항에 있어서, 상기 정적 보안 핸들러들은 사용자등으로부터의 암호키들을 포함하는 암호 정보를 수신하여 화이트박스 암호화 키 데이터, 화이트박스 복호화 키 데이터 및 화이트박스 보안 모듈 유틸리티를 생성하는 화이트박스 정적 보안 핸들러를 포함하고, 상기 화이트박스 암호화 키 데이터는 상기 정적 보안 핸들러들 중 하나 이상의 다른 정적 보안 핸들러들에서 사용되고, 상기 화이트박스 복호화 데이터 및 상기 화이트박스 보안 모듈 유틸리티는 상기 보안 모듈의 동적 런-타임 보호 동안에 상기 하나 이상의 동적 보안 핸들러들에 의하여 사용되는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 11

제10항에 있어서, 상기 정적 보안 핸들러들은 보호 마킹 정보에 응답하여 상기 보안된 자바 바이트코드에 해시 코드 보호를 적용하는 바이트코드 무결성 검증 정적 보안 핸들러를 포함하고, 상기 동적 보안 핸들러들은 런-타임에 상기 해시 코드 보호를 검증하는 바이트코드 무결성 검증 동적 보안 핸들러를 포함하고, 상기 보안 모듈은 검증 실패에 대한 탬퍼링 대책을 작동시키는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

청구항 12

제10항에 있어서, 상기 정적 보안 핸들러들은 빌드-타임 동안에 보호된 자바 어플리케이션 바이트코드 스터브, 보호된 어플리케이션 페이로드 및 암호화된 클래스 바이트코드 프레임을 형성하는 보안 로딩 바이트코드 정적 보안 핸들러를 포함하고,

상기 동적 보안 핸들러들은 상기 보안된 자바 어플리케이션 바이트코드에 상응하는 상기 암호화된 클래스 바이트코드 프레임을 메모리 버퍼에 로딩하고, 암호화된 클래스에 상응하는 상기 화이트박스 복호화 키 데이터를 통하여 상기 암호화된 클래스 바이트코드 내에 포함된 상기 암호화된 클래스 각각을 복호화하고, 보안 모듈 클래스 로더를 통하여 복호화된 클래스 바이트코드를 어플리케이션 작업 공간으로 로딩하여 상기 어플리케이션 작업 공간 내에서 상기 자바 어플리케이션 바이트코드를 실행시키는 것을 특징으로 하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치.

명 세 서

기 술 분 야

[0001] 본 발명은 일반적으로 컴퓨터 소프트웨어에 관한 것으로, 보다 상세하게는 적대적 실행 환경 내에서 정적 및 동적 공격들에 대하여 컴퓨터 소프트웨어가 저항성을 가지도록 하는 방법 및 시스템에 관한 것이다.

배 경 기 술

[0002] 컴퓨터 프로그래밍 산업 내에서, 자바 프로그램밍 언어는 모든 메이저 산업 부분에 사용되고 있고, 매우 넓은 범위의 장치들, 컴퓨터들 및 네트워크들에 사용되고 있다. 자바 어플리케이션들은 자바 프로그래밍 언어로 작성되어, 호스트 OS(operating system)와 호스트 CPU 명령어 셋 구조(instruction set architecture) 상에 배치되는 자바 가상 머신(JAVA virtual machine; JVM) 상에서 실행되는 기계-독립적인(machine-independent) 바이트코드이다. 자바 기술의 다양성, 효율성, 플랫폼 이동성 및 보안성은 네트워크 컴퓨팅에 이상적이다. 자바 프로그래밍 언어는 랩탑에서부터 데이터센터, 게임 콘솔에서 슈퍼컴퓨터 및 휴대 전화로부터 인터넷에 이르기까지 모든 곳에서 발견된다. 이동성, 확장성, 범용성 및 신뢰성은 자바의 주요한 강점들이다. 하지만, 이러한 동시다발성은 해커들 및 관련된 컴퓨터 공격들에 대하여 상당한 기회를 제공한다.

[0003] 자바 환경에 대한 신뢰할수 없는 어플리케이션으로부터 공격과 권한 없는 액세스를 보호하기 위하여, 자바 기술은 바이러스나 파괴 소프트웨어가 불법적으로 다운로드되거나 설치될 수 있는 실행 환경을 보호하는 자바 샌드-박스 보안 모델을 포함한다. 이러한 적대적 공격들을 방어하는 것은 전화통신 시스템, 교통 시스템, 방어 시스

템, 산업 자동화 시스템 및 전력 관리 시스템등과 같은 고도의 보호가 요구되는 환경과 시스템을 정상적으로 구동되는 중대한 어플리케이션을 설계하는데 필수적이다. 매년 이러한 중대한 시스템들이 자바 프로그램밍 언어를 이용하여 설계되고 구현된다.

[0004] 이와 같이 소비자 전자 산업은 발전된 기술과 생산품, 미디어 디지털화에 대한 빠른 요구 및 이머징 마켓들로부터의 증가하는 저축할 수 있는 수입들과 더불어 소비자 전자산업의 가격은 연속적으로 하락하는 전래없는 시대에 접어들었다. 이러한 소비자 전자 산업들은 그 기능을 주로 소프트웨어 어플리케이션에 의존한다. 이동성, 확장성 범용성 및 단순성과 같은 자바 프로그래밍 언어의 장점들은 소비자 전자 산업 생산품들의 개발기간과 개발비용을 감소시켜서 새로운 소비자 생산품들에 점점 더 많은 자비-기초의 플랫폼과 어플리케이션이 운용되도록 한다.

[0005] 거의 모든 소비자 일렉트로닉스 장치들은 신뢰할 수 없는 환경에서 기능될 것이 요구된다. 신뢰할 수 없는 환경에서, 소비자 일렉트로닉스 장치 내부의 소프트웨어는 이익이 되는 이유(예를 들어 필요로 하는 서비스를 획득)로부터 원하지 않는 이유(예를 들어, 상기 장치를 해킹하는 것)에 이르기까지의 서로 다른 목적을 위하여 액세스된다. 그 결과로 점점 더 많은 컴퓨터 어플리케이션들이 현재 가장 적대적인 환경에서 실행된다. 예들들어, PMP나 스마트 폰과 같은 휴대용 장치, 셋톱 박스, 미디어 플레이어 또는 개인용 컴퓨터와 같은 홈 네트워킹 및 웹을 기반으로 하는 환경들은 공격자들은 많은 양의 시간과 리소스들을 사용하는 영역이다. 그러므로 공격 소프트웨어에 대항하여 적법한 소프트웨어를 보호하는 것은 매우 중요한 이유가 된다. 더욱이, 고성능 하드웨어와 복잡한 공격 툴들은 침입자들에게 많은 새로운 이점들을 제공한다.

[0006] 소프트웨어 배포자들은 그들의 소프트웨어가 공격에 대하여 둔감하고 저항성을 가지도록 확실히 해야한다. 하지만 주어진 플랫폼과 소프트웨어는 시간, 리소스들, 도구들 및 공격자의 처리에 의존하는 웹에 전문가인 공격자에게 너무 잘 알려져 있다. 이러한 적대적 공격 환경은 종종 "화이트박스" 환경이라고 칭해지고, "화이트 박스" 환경에서는 모든 컨텐츠가 확실히 보이고 직접 액세스와 탬퍼링(tampering, 부당변경)에 달려있다. 이것은 컨텐츠가 보호되고 공격으로부터 보호받는 신뢰할 수 있고 보호받는 환경인 "블랙박스" 환경과 반대되는 개념이다. 화이트 박스 환경이 우세한 적대적 환경에서는 소프트웨어 시스템에 대한 직접적이고 자동화된 공격을 방어하거나 멈추는 것은 보안에 있어서 가장 중요한 사항이 되고 있다. 더욱이, 화이트 박스 공격에 대한 강한 방어는 적절하고 보안된 디바이스 기능을 위하여 담보되어야 한다. 자바 프로그래밍 언어는 이러한 보안 문제와 도전들에 적합하도록 설계되지 않았다. 이러한 견지에서, 소정의 자바 강점들은 C 나 C++로 프로그램하는 것과 비교하였을 때 실제적으로 보안상의 약점이 있다.

[0007] C/C++ 언어를 로(raw) 바이너리 데이터 상에서 동작하고 x86 또는 파워 PC와 같은 타겟 하드웨어에 특화된 로우-레벨 명령 세트로 컴파일하는 C/C++ 컴파일러와는 달리, 자바 컴파일러는 자바 소스 코드를 클래스 상에서 운용되고 실행 동안에 자바 가상 머신이 해석할 수 있는 원시 타입의 하이-레벨 포터블 바이트코드로 컴파일한다. 플랫폼 의존성은 자바 가상 머신 내부로 압축되고 자바 어플리케이션과는 분리된다.

[0008] 또한 자바 컴파일러는 C/C++ 컴파일러에서 공통적이며 일반적으로 발견되는 컴파일-타임 최적화를 수행하지 않는다. 그 대신에, 자바는 실행 프로파일을 성능 향상에 고려하면서 런-타임에서 모든 최적화를 실행하는 JIT(jsut-in-time) 컴파일에 의존한다. 주요한 C/C++ 코드 최적화는 컴파일 타임에 수행된다. 예를 들어, 인라인 대체는 주어진 기능이 바이너리 이미지 주위에 분산되는 결과를 초래한다; 컴파일 타임에 익스프레션을 평가하는 것과 함께 프리프로세서를 사용하는 것은 소스 코드에서 정의된 컨텐츠들의 흔적을 남기지 않는다. 일반적으로, 복잡하게 최적화된 코드는 리버스 엔지니어에게 더욱 어렵다.

[0009] 또한 자바 프로그램 의존성은 클래스들이 로딩되는 런-타임에 결정된다. 따라서 클래스의 명칭, 그것의 방법들의 명칭과 필드들이 클래스 파일 내에 존재하여야 하고, 또한 모든 임포티드 클래스들의 명칭들과, 호출된 방법들 및 액세스된 필들도 또한 클래스 파일 내에 존재하여야 한다. 다른 한편으로는 C/C++ 프로그램들은 정적으로 링크된다. 그러므로 클래스들의 명칭, 멤버들 및 변수들은 동적 라이브러리로부터 제공된 명칭들을 제외하고는 컴파일되고 링크된 프로그램 내에 존재할 필요가 없다.

[0010] 결과적으로, 자바 어플리케이션은 일 세트의 자바 아키브(Java Archive; JAR) 파일로서 전해진다. 상기 JAR 포맷은 복수의 파일들이 하나의 아키브 파일로 번들화될 수 있고, 상기 복수의 파일들은 본질적으로 비-암호화된 아키브들이고, 이로부터 개별적 클래스들을 추출하기기 쉽다. 비교하면, C/C++ 어플리케이션은 몇 개의 동적 라이브러리와 링크될 수 있는 수행 가능한 모놀리딕으로 제공되기 때문에, C/C++ 어플리케이션에서는 프로그램 정보와 개별 코드를 확인하기가 쉽지 않다.

[0011] 따라서 자바 바이트코드를 자바 소스로 디컴파일하는 것은 C/C++을 디셈블하는 것보다 더욱 간단하고 쉬우며, 따라서 전적으로 자동화될 수 있다. 클래스 하이어아키, 진술들, 클래스들의 명칭들, 방법 및 필드들과 같은 프로그램 정보는 바이트코드로부터 추출될 수 있다. 비록 이용 가능한 프리웨어와 상업적 자바 숨김 툴들이 존재하지만, 이것들 중 어느 것도 바이트코드의 실행을 직접 공격하는 것을 방지하는 방어를 제공하지는 않는다. 그 결과로 자바 리버스 엔지리어닝은 일상적인 프랙티스가 되었다.

[0012] 더욱이, 자바 가상 머신은 자바 어플리케이션에 대하여 개방 런타임 환경을 제공한다. 자바 가상 머신을 보호하는 내재된 보안은 거의 없고 따라서 자바 가상 머신을 로버스트(robust)하게 만들어야 한다. 자바 가상 머신에 부가하거나 자바 가상 머신을 이용하여 어택을 론칭하는 것은 비교적 부차적인 문제이다. 그러므로, 자바 어플리케이션 코드에 적용되는 보호들의 강도와 관계없이, 자바 가상 머신의 연약함 때문에 해커들은 자바 가상 머신을 가장 약한 링크로 항상 사용하여 화이트박스 공격을 구현할 수 있다. 좀더 신뢰할 수 있고 둔감한 자바 가상 머신이 자바 어플리케이션을 보호하고 화이트 박스 공격들을 방어할 수 있을지라도, 이러한 접근 방법은 현재의 자바 보안 모델과 관련된 중요한 산업적 서포트 및 적용에 상당한 변화를 초래할 것이다. 그러므로 화이트박스 환경 내에서 어플리케이션을 보호할 수 있는 산업적 표준 자바 가상 머신 내의 신뢰가능하고 둔감한 컴포넌트를 구비하는 것이 바람직하다.

발명의 내용

해결하려는 과제

[0013] 따라서 본 발명의 목적은 기존 자바 플랫폼 구성의 주요한 약점을 경감시키는 것이다.

과제의 해결 수단

[0014] 본 발명의 실시예는 실행 시간 동안에 자바 플랫폼의 약점을 언급할 수 있고 자바 바이트코드를 보호할 수 있는 보안 모듈을 제공한다. 상기 보안 모듈은 예를 들어 C/C++로 구현될 수 있다. 본 발명의 실시예에 따른 보안 모듈이 C/C++로 구현될수 있기 때문에, 이에 의하여 C/C++ 소프트웨어 코드를 보호하는 보안 기술을 사용할 수 있다. 본 발명의 목적에 따른 적절한 보안 기술은 캐나다 온타리오 오타와 소재의 "Cloakware" 사로부터 제공된다. 상기 보안 기술들은 공동 소유의 미국등록특허 제7506177호, 미국등록특허 제7464269호, 미국등록특허 제 7397916호, 미국등록특허 제7395433호, 미국등록특허 제7350085호, 미국등록특허 제7325141호, 미국등록특허 제 6779114호 및 미국등록특허 제6594761호에 개시되어 있고 그 내용들은 본 출원서에서 참조로서 포함된다.

[0015] 상기 참조된 특허 문헌들 및 "Cloakware"의 관계된 제품들에 개시된 현존하는 소프트웨어 보안 기술은 적대적(언트러스티드) 실행 환경에서 합법적인 어플리케이션과 그 어플리케이션의 기능과 지적 자산을 보호하기 위하여 사용되어 상기 어플리케이션들에 대한 화이트박스 공격을 방어한다. 이러한 존재하는 소프트웨어 보안 기술은 C/C++로 구현된 어플리케이션들과 네이티브 컴파일드 코드를 보호하고 소프트웨어 및 보안을 전통적인 어플리케이션 빌딩 프로세스를 향상시킴으로서 분리할 수 없게 함으로써 보호하는 실재적인 소스 코드와 바이너리 보호 툴들을 포함한다.

[0016] 제1 실시예에서, 본 발명은 빌드-타임(bulid-time) 동안에 자바 바이트코드를 보안시키는 보호 툴; 상기 보호 툴로부터 상기 보안된 자바 바이트코드를 수신하여 런-타임 동안에 상기 보안된 자바 바이트코드를 론칭(launching)하는 보안 모듈; 및 상기 보안 툴과 상기 보호 모듈과 집적되는 하나 이상의 보호 메커니즘들을 구비하고, 상기 하나 이상의 보호 메커니즘들은 상기 자바 바이트코드에 대한 정적 및 동적 공격들에 대항하여 동작하는 자바 바이트코드의 탬퍼-저항을 증가시키는 장치를 제공한다.

[0017] 실시예에 따라서, 상기 보안된 자바 바이트코드는 보호된 자바 어플리케이션 바이트코드 스터브, 보호된 어플리케이션 페이로드 및 암호화된 클래스 바이트코드 프레임을 포함하고, 상기 자바 어플리케이션 바이트코드 스터브, 상기 보호된 어플리케이션 페이로드 및 상기 암호화된 클래스 바이트코드 프레임은 상기 빌드-타임 동안에 상기 보호 툴에 의하여 형성된다.

[0018] 실시예에 따라서, 상기 보안 모듈은 자바 가상 머신에 대한 기능적 확장으로서 독립적으로 분포되어 보호된 자바 어플리케이션들의 신뢰성의 루트를 제공하고, 상기 보안된 자바 바이트코드는 사용자의 필요에 따라 개별적으로 분포된다.

[0019] 실시예에 따라서, 상기 보호 툴은 상기 보호된 어플리케이션 페이로드가 상기 보호된 자바 어플리케이션 바이트

코드 스터브를 통하여 론칭되도록 명령하는 메커니즘을 포함한다.

[0020] 실시예에 따라서, 상기 보안 모듈은 보호된 바이트코드 클래스 로더를 포함하고, 상기 보호 툴은 상기 보호된 어플리케이션 페이로드가 상기 보호된 바이트코드 클래스 로더를 이용하여 상기 암호화된 클래스 바이트코드 프레임을 통하여 론칭되도록 명령하는 메커니즘을 포함한다.

[0021] 실시예에 따라서, 상기 장치는 C, C++ 및 자바중 하나 이상을 포함하는 프로그래밍 언어로 구현된 프로그램밍 엔진으로 형성된다.

[0022] 실시예에 따라서, 상기 장치는 자바 가상 머신과 인터페이싱할 수 있는 프로그래밍 언어로 구현된 프로그램밍 엔진으로 형성된다.

[0023] 실시예에 따라서, 상기 하나 이상의 보호 메커니즘들은 구성 옵션에 따라 선택가능하다. 상기 보호 메커니즘들은 상기 보호 툴 내부에 형성된 정적 보안 핸들러들과 상기 보안 모듈에서 형성되는 동적 보안 핸들러들을 포함한다.

[0024] 상기 정적 보안 핸들러들은 사용자등으로부터의 암호키들을 포함하는 암호 정보를 수신하여 화이트박스 암호화 키 데이터, 화이트박스 복호화 키 데이터 및 화이트박스 보안 모듈 유틸리티를 생성하는 화이트박스 정적 보안 핸들러를 포함하고, 상기 화이트박스 암호화 키 데이터는 상기 정적 보안 핸들러들 중 하나 이상의 다른 정적 보안 핸들러들에서 사용되고, 상기 화이트박스 복호화 데이터 및 상기 화이트박스 보안 모듈 유틸리티는 상기 보안 모듈의 동적 런-타임 보호 동안에 상기 하나 이상의 동적 보안 핸들러들에 의하여 사용된다.

[0025] 상기 정적 보안 핸들러들은 보호 마킹 정보에 응답하여 상기 보안된 자바 바이트코드에 해시 코드 보호를 적용하는 바이트코드 무결성 검증 정적 보안 핸들러를 포함하고, 상기 동적 보안 핸들러들은 런-타임에 상기 해시 코드 보호를 검증하는 바이트코드 무결성 검증 동적 보안 핸들러를 포함하고, 상기 보안 모듈은 검증 실패에 대한 탬퍼링 대책을 작동시킨다.

[0026] 상기 정적 보안 핸들러들은 빌드-타임 동안에 보호된 자바 어플리케이션 바이트코드 스터브, 보호된 어플리케이션 페이로드 및 암호화된 클래스 바이트코드 프레임을 형성하는 보안 로딩 바이트코드 정적 보안 핸들러를 포함하고,

[0027] 상기 동적 보안 핸들러들은 상기 보안된 자바 어플리케이션 바이트코드에 상응하는 상기 암호화된 클래스 바이트코드 프레임을 메모리 버퍼에 로딩하고, 암호화된 클래스에 상응하는 상기 화이트박스 복호화 키 데이터를 통하여 상기 암호화된 클래스 바이트코드 내에 포함된 상기 암호화된 클래스 각각을 복호화하고, 보안 모듈 클래스 로더를 통하여 복호화된 클래스 바이트코드를 어플리케이션 작업 공간으로 로딩하여 상기 어플리케이션 작업 공간 내에서 상기 자바 어플리케이션 바이트코드를 실행시킨다.

[0028] 본 발명의 다른 측면 및 특징들은 하기의 상세한 설명 및 첨부되는 도면을 통하여 본 발명의 실시분야에서 통상의 지식을 가진 자들에게 더욱 분명해질 것이다.

도면의 간단한 설명

[0029] 본 발명의 실시예들은 예로서 설명된 첨부되는 도면을 참조하여 보다 상세히 설명된다.

도 1은 네이티브 코드를 구비하는 자바 어플리케이션과 브리징하는 종래의 JNI를 간략히 나타내는 도면이다.

도 2는 종래의 자바 어플리케이션 바이트코드에 대한 정적 공격의 메커니즘을 나타낸다.

도 3은 종래의 자바 어플리케이션 바이트코드에 대한 동적 공격의 메커니즘을 나타낸다.

도 4는 본 발명의 실시예에 따른 자바 바이트코드 보호 시스템의 개략도를 나타낸다.

도 5는 본 발명의 실시예에 따라 빌드-타임에 자바 어플리케이션 바이트코드를 보호하기 위한 도 4에 도시된 상부부분의 빌드-타임 프로세스를 나타낸다.

도 6은 본 발명의 실시예에 따라 런-타임에 자바 어플리케이션 바이트코드를 보호하기 위한 도 4에 도시된 하부부분의 런-타임 프로세스를 나타낸다.

도 7은 본 발명의 실시예에 따라 스타트-업과런-타임에 안티-디버그 능력을 나타낸다.

도 8은 본 발명의 실시예에 따라 외부 화이트박스 암호 라이브러리를 나타낸다.

도 9는 본 발명의 실시예에 따라 내부 화이트박스 암호 설비를 나타낸다.

도 10은 본 발명의 실시예에 따라 바이트코드 보호 툴의 프리-프로세스를 나타낸다.

도 11은 본 발명의 실시예에 따라 BIV 정적 보안 핸들러의 작업 흐름을 나타낸다.

도 12는 본 발명의 실시예에 따라 BIV 동적 보안 핸들러의 작업 흐름을 나타낸다.

도 13은 본 발명의 실시예에 따라 SLB 정적 보안 핸들러의 작업 흐름을 나타낸다.

도 14는 본 발명의 실시예에 따라 SLB 동적 보안 핸들러의 작업 흐름을 나타낸다.

도 15는 본 발명의 실시예에 따라 부트스트랩 인터페이스와 은폐된 자바 얼플리케이션의 보안 로딩을 나타낸다.

도 16은 본 발명의 실시예에 따라 DBD 정적 보안 핸들러의 작업 흐름을 나타낸다.

도 17은 본 발명의 실시예에 따라 DBD 흐름도를 나타낸다.

도 18은 본 발명의 실시예에 따라 DBD 동적 보안 핸들러의 작업 흐름을 나타낸다.

발명을 실시하기 위한 구체적인 내용

[0030] 도 1에 도시된 바와 같이, 자바 플랫폼(110)은 자바 가상 머신(104), 자바 어플리케이션(106) 및 자바 가상 머신 내에서 바이트코드로 로딩되는 라이브러리(108)를 포함하는 자바 세계와 어플리케이션이나 공유 라이브러리가 C/C++/어셈블러와 같은 다른 언어로 작성되고 호스트 CPU ISA로 컴파일된 네이티브 코드 세계(110) 사이의 양방향 상호작용을 연결하는데 편리함을 제공하는 자바 네이티브 인터페이스(Java Native Interface; JNI, 102)를 추가적으로 포함한다. 자바 프로그래밍 언어와 C/C++/어셈블러 코드에서 JNI API를 사용하여, C/C++/어셈블러 바이너리 코드는 자바로부터 호출될 수 있고, 자바 바이트코드를 호출할 수 있다. 여기에는 두 가지 상호작용이 있다: 자바 어플리케이션이 네이티브 방법을 호출하는 다운-콜(하향 호출)과 네이티브 방법이 데이터를 액세스하거나 JNI 환경을 통하여 주어진 자바 어플리케이션의 방법을 호출하는 업-콜(상향-호출)이 있다.

[0031] 런-타임에, 본 발명의 보안 모듈은 JNI를 통하여 자바 가상 머신 내에서 동시 실행되어 상기 주어진 자바 어플리케이션은 보안 모듈 내에서 보안 동작을 작동시킬 수 있고, 상기 보안 모듈에서는 이러한 보안 동작이 자바 어플리케이션이나 자바 가상 머신 내에 로딩된 다른 자바 라이브러리 코드를 액세스하여 보호 동작을 수행할 수 있다.

[0032] 본 발명의 접근 방법은 JNI 메커니즘을 통하여 자바 가상 머신 내에서 전적으로 보호되고 신뢰할 수 있는 보안 모듈을 도입함으로써 현존하는 자바 가상 머신에 보안을 효율적으로 부가하는 것이다. 런-타임에 상기 보안 모듈은 상기 자바 가상 머신 내에서 신뢰의 원천과 "트램폴린 및 엔진"과 같은 보호의 원천으로서 동작하여 자바 바이트코드에 대한 다양한 보호를 시작시키고 수행한다. 이러한 방식으로, 본 발명은 현존하는 자바 플랫폼에 아무 커다란 변화를 필요로 하지 않는다. 차라리, 현존하는 및 새로이 채택되는 장치 및 시스템들은 이 솔루션으로부터 즉시 혜택을 볼 수 있다. 다시 말하면, 본 발명은 현존하는 자바 인프라스트럭처에 보안 확장으로 다루어져 현재의 자바 어플리케이션들이 직면하는 보안 문제를 해결책을 제시할 수 있다. 따라서 본 발명은 런-타임 동안에 바이트코드를 액세스하고 상기 자바 어플리케이션에 대한 정적 및 동적 공격에 응답하여 자바 바이트코드에 대한 일 세트의 보호 방법을 수행할 수 있는 JNI의 능력을 향상시킬 수 있는 자바 바이트코드 보호 보안 모듈을 제공한다.

[0033] 본 발명은 자바 바이트코드 보호 시스템 내에서 고도로 신뢰할 수 있는 보호 툴과 보안 모듈을 제공한다. 본 발명은 자바 가상 머신과 자바 보안만에 기초하여 자바 어플리케이션 보호에 의존하지 않는다. 차라리, 본 발명은 JNI를 통하여 자바 가상 머신과 함께 동작할 수 있는 트러스티드-존(trusted-zone)인 자바 바이트코드 보안 모듈을 도입하여 런-타임 동안에 자바 바이트코드 보호를 시작하고 수행하고 관리한다. 상기 보안 모듈의 신뢰성은 프로그래밍 언어와 상기 보호 툴과 상기 보안 모듈이 작성되는 C/C++ 코드에 알려진 효과적은 보안 보호를 인가함으로써 담보할 수 있다. 이러한 신뢰할 수 있는 보안 모듈에서는, 상기 신뢰성은 아래에서 기술되는 보안 모듈에 의하여 제공되는 소정의 보호에 의하여 상기 보안 모듈로부터 상기 자바 어플리케이션 및 자바 가상 머신까지 확대될 수 있다.

[0034] 도 2 및 도 3을 참조하면, 자바 바이트코드에 대한 전형적인 정적 및 동적 공격들이 도시되어 있다. 일반적으로, 임의의 주어진 자바 어플리케이션은 자바 소스 형태(202)로 개발되어 자바 컴파일러(206)에 의하여 자바 바이트코드(204)로 컴파일된다. 여기서 자바 바이트코드(204)는 배포에 앞서 아키버 유틸리티(201)를사용

하여 아키브 파일(예를 들어, JAR 파일; 208)에 저장된다. 이러한 분포는 CD나 다운로드가능한 파일의 형태를 가질 수 있다.

[0035] 정적 공격자(212)는 일반적으로 자바 컴파일러와 같은 리버스 엔지니어링 툴을 사용하여 상기 분포된 미디어로부터 소유자가 있는 데이터나 소프트웨어 알로리즘과 같은 중요한 지적 자산(214)을 추출한다. 그렇게 하는 동안에 상기 공격자는 상기 코드를 불법적으로 변경하거나 근원적인 코드를 손상시킨다. 상기 주어진 자바 어플리케이션의 분포 동안에 자바 바이트코드에 대한 이러한 정적 공격을 방어하기 위하여, 본 발명에서는 어플리케이션 바이트코드에 일정한 레벨의 보호를 인가한다. 이 보호는 상기 분포전에 제공되어 상기 정적 공격이 매우 어려운 일이 되도록 한다. 본 발명의 방법에 따른 효과적인 보호를 제공한 후에, 상기 어플리케이션 바이트코드에 내장된 상기 지적 자산은 쉽게 리버스 엔지니어링될 수 없고, 상기 보호된 바이트코드를 부당하게 변경하는 것은 매우 비실용적이된다. 더욱이, 본 발명의 정적 보호는 부당하게 변경된 바이트코드는 합법적인 자바 가상 머신에 의하여 로딩될 수 없고 운용될 수 없기 때문에 이점을 갖는다.

[0036] 어플리케이션에 대한 정적 공격과 비교할 때, 동적 공격자(302)는 자바 가상 머신이 상기 자바 어플리케이션을 로딩하고 구동하는 동안에 동적 공격 툴을 사용하여 자바 바이트코드를 공격할 수 있다. 동적 공격 툴 및 방법을 사용하여, 공격자는 자바 가상 머신(304)과 어플리케이션 바이트코드(306)를 액세스하고, 바이트코드(308)를 관찰하고 수정하여 그들의 공격 목적에 따라 원래 의도된 행동과 중요한 값들을 직접적으로 이해 및/또는 변경할 수 있다. 더욱이, 상기 공격자는 원본 바이트코드를 해제하는 것을 포함하여 상기 바이트코드로부터 중요한 지적 자산(310)과 비밀들을 획득할 수 있다. 런-타임 동안에 자바 코드에 대한 이러한 동적 공격을 방어하기 위하여, 본 발명에서는 배포 전에 상기 어플리케이션 바이트코드에 대한 보호를 형성하여 주입한다. 더욱이, 본 발명은 런-타임 동안에 이러한 보호를 주입하여 어떠한 동적 공격들도 실행가능하지 않게 한다. 본 발명은 동적 공격을 방어할 수 있을 뿐만 아니라, 상기 보호된 어플리케이션이 동적 공격을 감지할 수 있게 하고, 잠재적인 공격자들에 대하여 이러한 공격이 시간과 리소스면에서 매우 비싼 노력이 되도록 만들고 또한 동적 공격을 완화시키는 능력을 추가한다.

[0037] 도 4는 본 발명의 일 실시예에 따른 바이트코드 보호 시스템(400)과 관련된 방법을 개략적으로 나타낸다.

[0038] 여기서, 자바 바이트코드 보호 시스템(400)은 빌드-타임 보호 툴(402)과 런-타임 보안 모듈(404)의 두 가지 파트를 포함한다. 상수한 바와 같이, 상기 바이트코드 보호 시스템(400)과 관련된 방법은 캐나다 온타리오의 오타와에 위치한 Cloakware Inc.로부터 입수할 수 있는 기술을 사용하여 C/C++로 구현될 수 있다.

[0039] 상기 자바 바이트코드 보호 툴(402)은 배치되기 전에 자바 바이트코드(406)에 대하여 보안(예를 들어 "은폐물")을 인가하기 위하여 사용된다. 상기 자바 바이트코드 보호 툴(402)은 빌드-타임 동안에 명시되는 보안 설정과 보호 메커니즘을 가능하게 한다. 이 툴은 상기 원본 자바 어플리케이션 바이트코드(406), 보안 사양들 및 화이트박스 암호 키들을 입력으로 하여 상기 자바 바이트코드 보호 보안 모듈(404)과 함께 구동되는 은폐된 자바 바이트코드를 생성한다. 상기 자바 바이트코드 보호 툴(402)은 상기 배치된 보안된 자바 바이트코드에 대한 보안 테크닉을 명시하는 옵션들 뿐만 아니라 보호된 어플리케이션 바이트코드 스터브 또는 보호된 바이트코드 클래스 로더를 통하는 것과 같은 바이트코드가 어떻게 시작될지를 명시하는 옵션들을 포함할 수 있다. 상기 자바 어플리케이션의 상기 은폐된 바이트코드는 목표 자바 가상 머신 환경(410)에 로딩되는 보호된 자바 어플리케이션 바이트코드 스터브(408)와 런-타임 동안에 개별적으로 로딩되고 보안 모듈에 의하여 액세스되는 보호된 데이터 파일(412)과 화이트박스 보안 모듈(white box security module, WB SM) 유틸리티(414)의 두 파트로 배포된다. 본 발명의 실시예에 따른 자바 바이트코드 보호 보안 모듈(404)은 어플리케이션 제공 접근방법에 따라 이 두 파트 또는 독립적으로 배포될 수 있다. 일반적으로 상기 보안 모듈(404)은 한번만 설치할 수 있고 임의의 자바 어플리케이션의 은폐된 바이트코드에 적용할 수 있다는 점에서 포괄적이다.

[0040] 본 발명의 실시예에 따른 바이트코드보호 방법의 다양한 예들이 상기 자바 바이트코드 보호 시스템(400)과 함께 사용되어 활성화될 수 있다. 상기 바이트코드 보호 방법 각각은 바이코드 폼의 자바 어플리케이션에 대한 정적 및 동적 공격들에 대한 해결책을 제시할 수 있다.

[0041] 바이트코드를 보호하는 하나의 방법은 적대적 환경에서 암호화키를 및 다른 암호화 값을 누출시키지 않고 동작을 실행할 수 있는 암호화 알고리즘을 보호하는 유일한 암호화 기술인 화이트박스 암호화를 포함한다. 다시 말하면, 상기 화이트박스 암호화 방법은 직접적 공격에 대하여 실행될 수 있다. 본 발명은 외부 화이트박스 암호화 라이브러리 및 내부 화이트박스 암호 시설을 포함하는 두 종류의 암호화 기술을 포함한다.

[0042] 상기 외부 화이트박스 암호화 라이브러리는 C로 구현되고 숨겨진 암호화 키 및 다른 암호화 정보에 의하여 탬퍼

링 저항 값을 갖도록 보호되어 상기 키를 포함하는 중요한 정보를 유출시키지 않고 보호된 자바 어플리케이션에 의하여 화이트박스 암호화 동작이 이용되고 수행되도록 한다. 상기 내부 화이트박스 암호 시설은 상기 암호화 정보 및 키를 수신하여 본 발명의 실시예에 따른 보호 툴 및 보안 모듈(404)이 서로 다른 형태의 자바 어플리케이션 바이트코드 및 적절한 정보를 암호화하고 복호화하도록 하는 화이트박스 키 데이터와 유틸리티들을 생성하는 빌드-타임 보호 툴(404)이 기능적 구성요소이다.

[0043] 바이트코드를 보호하는 다른 하나의 방법은 바이트코드 무결절 검증(Bytecode Integrity Verification, BIV)을 포함한다. BIV를 통한 보호는 클래스를 로딩하거나 자바 방법들을 구동하는 동안에 자바 클래스나 방법에 대한 정적 및 동적 부당 변경(tampering) 공격을 완화시킨다. 빌드-타임에, 본 발명의 방법은 JAR파일, 클래스 및 원본 어플리케이션 아키브 파일로부터의 방법 바이트코드의 정적 해쉬 값들을 계산한다. 로딩 및 런-타임 동안에, 본 발명의 방법은 자바 가상 머신(410)에 로딩된 방법 바이트코드와 클래스를 어드레싱하여 동적 해쉬 값들을 계산하고, 정적 해쉬 값들에 대한 동적 해쉬 값들을 체크함으로써 상기 무결성 검증을 수행한다.

[0044] 바이트코드를 보호하는 다른 하나의 방법은 도 7을 참조하여 설명될 안티-디버그를 포함한다. 안티-디버그는 도 4에 도시된 동적 보안 핸들러들(416) 중의 하나이다. 안티-디버그 보호는 런-타임 동안에 디버거들을 사용하여 동적 공격들을 방어하고 감지할 수 있다. 안티-디버그는 스타트업과 런-타임 동안에 시스템 환경의 내외적 상태를 모니터링하여 공격을 감지하는 기술들로 구성된다. 일단 안티-디버그 공격들이 감지되면 적절한 대책들이 기동된다.

[0045] 바이트코드를 보호하는 다른 하나의 방법은 보안 로딩 바이트코드(Secure Loading Bytecode, SLB)를 포함한다. 상기 SLB 보호 방법은 자바 가상 머신(410)에 로딩되기 전에 아키브 파일 및 자바 클래스 코드에 대한 리버스 엔지니어링 및 탬퍼링 공격들을 방어하고 감지한다. 빌드-타임에 SLB 보호 방법은 JAR 파일들과 상기 원본 어플리케이션 아키브 파일에서 선택된 클래스 바이트코드를 암호화하고, 어플리케이션 스터브 클래스를 도입한다. 자바 가상 머신(410)이 상기 보호된 어플리케이션을 로딩하면, 상기 자바 가상 머신(410)은 먼저 상기 어플리케이션 스터브 클래스를 로딩한 다음에 상기 보호된 어플리케이션의 로딩을 트리거한다. 하기에서 기술되는 SLB 동적 보안 핸들러(416)는 런-타임 실행동안에 JNI(418)를 통하여 가자 가상 머신(410)과 연결되는 자바 바이트코드 보호 보안 모듈(404)의 기능적 구성요소이다. 상기 SLB 동적 보안 핸들러(416)는 자바 가상 머신(410)의 작업 공간으로 보호된 자바 어플리케이션 바이트코드의 로딩을 관리하고 제어한다.

[0046] 바이트코드를 보호하는 다른 하나의 방법은 동적 바이트코드 복호화(Dynamic Bytecode Decryption; DBD)이다. 상기 DBD 보호 방법은 런-타임 동안에 자바 클래스나 방법에 대한 동적 공격을 방어하고 완화시킨다.

[0047] 바이트코드를 보호하는 다른 세터의 방법은 전달-실행과 부분 실행을 포함한다. 이 보호 방법들은 모두 원본 실행의 일부를 보안 모듈(404)로 이동시키고 실행의 부분만이 자바 가상 머신(410) 내부에서 노출되도록 하여 런-타임 동안에 동적 코드 리프팅 공격을 방어하고 완화시킨다. 예를 들어, 소정의 자바 바이트코드는 상기 보안 모듈(404) 내에서 보호되고 실행될 수 있는 C 코드로 변환(J2C)될 수 있다.

[0048] 바이트코드를 보호하는 다른 하나의 방법은 바이트코드 변환을 포함한다. 이런 종류의 보호는 데이터 흐름 변환과 제어 흐름 변환을 포함하는 기술들에 의하여 달성될 수 있다. 바이트코드 변환은 상기 원본 기능을 유지하면서 원본 바이트코드를 다른 코드 구조로 변환시킬 수 있다. 상기 변환된 바이트코드는 리버스 엔지니링에 더욱 더 어려워지고 탬퍼 저항을 갖게 된다.

[0049] 도 5를 참조하면, 상기 자바 바이트코드 보호 툴(402)은 원본 어플리케이션 바이트코드에 대하여 서로 다른 보호 기술들을 적용한다. 따라서, 상기 자바 바이트코드 보호 툴(402)은 보호된 바이트코드와 관련된 데이터를 생성하고, 런-타임 동안에 상기 작업을 자바 바이트코드 보안 모듈과 이용하여 자바 바이트코드에 대한 지정된 보호 기술들을 구현한다. 상기 자바 바이트코드 보호 툴(402)은 사용자 인터페이스(518)를 통하여 암호화 정보 및 키들(502), 원본 JAR 파일들(504) 및 구성 옵션들(506)의 세 가지 입력을 수신하여 세 가지 종류의 동작을 수행한다.

[0050] 상기 세 가지 종류의 동작 중에서 제1 기본 동작은 화이트박스 키 데이터와 유틸리티의 생성을 포함한다. 암호 정보 및 키들(502)을 이용하여, 화이트박스 정적 핸들러(508)는 서로 다른 정적 핸들러들(이하에서 상세히 설명됨)과 툴 자체에서 사용되는 화이트박스 암호화 키 데이터(510)를 생성한다. 또한, 상기 보호 툴(402)의 빌드-타임 프로세스는 데이터 보호 폴더(514)에 런-타임 데이터(512)의 일부로서 저장되는 화이트박스 복호화 키 데이터를 생성한다. 화이트박스 보안 모듈 유틸리티(516)는 상기 화이트박스 복호화 데이터를 사용하여 런-타임 동안에 상기 동적 보안 핸들러에 의하여 촉발되는 화이트박스 복호화 동작을 수행한다.

[0051] 상기 세 가지 종류의 동작 중에서 제2 기본 동작은 보호 기술들을 적용하는 것을 포함한다. 구성 옵션들에 따라서, 상기 자바 바이트코드 보호 툴(402)은 서로 다른 정적 보안 핸들러들을 적용하여 어플리케이션 바이트코드를 수정하여 원본 형태로부터 보호된 형태로 변환한다. 그렇게 함으로써, 제2 기본 동작에 의하여 보호된 자바 어플리케이션 바이트코드 스터브(408)와 여러 가지 보호 형태의 보호된 어플리케이션 바이트코드와 중요한 런-타임 데이터를 포함하는 관계된 보호 데이터 파일들을 생성한다.

[0052] 상기 세가지 종류의 동작 중에서 제3 기본 동작은 상기 보호된 자바 바이트코드의 배치가능한 폼을 패키징하는 것을 포함한다. 상기 과정의 끝에서, 상기 자바 바이트코드 보호 툴(402)은 모든 출력 파일들을 구조화하고 패킹하여 상기 자바 어플리케이션 바이트코드 스터브(408)가 상기 자바 가상 머신(410)에 의하여 로딩되도록 한다. 이 자바 어플리케이션 바이트코드 스터브(408)는 상기 엄폐된 자바 어플리케이션을 시작시키는 엔트리 포인트이고, 여러 가지 형태일 수 있다. 이러한 여러 가지 형태는 외부 프로그램에 의하여 시작될 수 있는 클래스 파일, 또 다른 자바 클래스에 의하여 시작될 수 있는 클래스 파일 또는 자바 클래스 로더를 포함할 수 있다. 상기 자바 바이트코드 보호 툴(402)은 모든 출력 파일들을 적절히 구조화하고 패킹하여 상기 화이트박스 보안 모듈 유틸리티(516)가 상기 자바 바이트코드 보안 모듈의 기능적 구성요소들에 의하여 작동될 수 있도록 한다. 또한 상기 자바 바이트코드 보호 툴(402)은 모든 출력 파일들을 적절히 구조화하고 패킹하여 모든 보호 데이터 파일들이 상기 자바 바이트코드 보안 모듈의 소정의 기능적 구성요소들에 의하여 액세스될 수 있도록 한다.

[0053] 도 5는 상기한 자바 어플리케이션 바이트코드를 보호하는 빌드-타임 프로세스를 개략적으로 나타낸다. 도 5를 참조하여, 주요한 기능적 컴포넌트들 및 데이터 파일들이 이하에서 상술된다.

[0054] 상기 자바 바이트코드 보호 툴(402)은 사용자와 인터페이싱하여 사용자 커맨드와 주요한 입력들을 받아들이는 사용자 인터페이스(518)를 포함한다. 상기 사용자 커맨드와 주요한 입력들은 암호 알고리즘 선택 및 원본 키 자료들을 포함하는 암호 정보 및 키들(502); 보호되어야 할 보호되지 않은 바이트코드를 포함하는 원본 어플리케이션 바이트코드 아키브 파일(504); 및 보호여부를 결정해야할 사용자가 명시할 수 있는 특정한 자바 클래스와 방법들과 같은 무엇을 그리고 어떻게 상기 어플리케이션 바이트코드를 보호할지에 대한 상기 자바 바이트코드 보호 툴을 수행하는 사용자 옵션을 포함하는 구성 옵션(402)을 포함할 수 있다.

[0055] 상기 자바 바이트코드 보호 툴(402)은 또한 보호 매니저(520)를 포함한다. 상기 보호 매니저(520)는 상기 구성 옵션(506)을 해석하고, 이에 의존적인 순서로 서로 다른 보호 기술들을 편성하여 서로 다른 보호 기술들이 서로 맞물리게 하여 상기 보호 기술들 각각보다 더욱 강력한 보호를 하기 위하여 제공된다. 또한 상기 보호 매니저(520)는 상기 자바 바이트코드 보호 툴(402)의 다른 기능 컴포넌트들에 의하여 공통으로 이용되는 유틸리티들을 포함한다.

[0056] 상기 자바 바이트코드 보호 툴(402)은 또한 정적 보안 핸들러들(522)을 포함한다. 상기 정적 보안 핸들러들(522) 각각은 상기 보호 매니저(520)에 의하여 동작되어 각각의 미리결정된 보호 기술을 수행한다. 도 5의 실시예에서는, 화이트박스-정적 핸들러(502), BIV 정적 핸들러(524), AD 정적 핸들러(526), SLB 정적 핸들러(528), DBD 정적 핸들러(530), 전송-실행 정적 핸들러(523), 부분 실행 정적 핸들러(534) 및 코드 변환 툴(536)이 도시되어 있다. 상기 각 정적 핸들러는 이하에서 상술된다. 상기 보호 매니저(520) 및 정적 보안 핸들러들(522)은 함께 동작하여 상기 보호 툴과 함께 새로운 보안 핸들러들을 쉽게 포함하여 보안 능력과 새로운 보호를 부가하고 확장하는 플러그-인 메커니즘을 제공하도록 설계된다.

[0057] 상기 자바 바이트코드 보호 툴(402)은 또한 상기 화이트박스 정적 보안 핸들러(508)에 의하여 생성된 화이트박스 암호화 키 데이터(510)를 포함한다. 상기 화이트박스 암호화 키 데이터(510)는 상기 보호 매니저(520)와 상기 정적 보안 핸들러들(522)에 의하여 이용되어 소정의 형태의 바이트코드와 보호 데이터를 암호화한다.

[0058] 상기 자바 바이트코드 보호 툴(402)은 또한 화이트박스 정적 보안 핸들러(508)에 의하여 생성된 화이트박스 보안 모듈 유틸리티(516)를 포함한다. 상기 화이트박스 보안 모듈 유틸리티(516)는 상기 보안 모듈(404) 내에서 상기 동적 보안 핸들러들에 의하여 사용된다.

[0059] 상기 자바 바이트코드 보호 툴(402)은 또한 보호된 자바 어플리케이션 바이트코드 스터브(408)를 포함한다. 상기 스터브(408)는 상기 자바 가상 머신(410)에 대한 상기 보호된 자바 어플리케이션의 부트-스트랩만을 포함하여 먼저 로딩하고 다음에 보호 데이터 파일들로부터 실제의 보호된 바이트코드를 로딩하는 보안 바이트코드 로더 기능을 트리거한다.

[0060] 상기 자바 바이트코드 보호 툴(402)은 또한 상기 툴에 의하여 생성된 보호된 J2C 라이브러리(538)를 포함한다. 상기 보호된 J2C 라이브러리(538)는 자바 바이트코드로부터 변환된 C로 구현된 여러 가지 보호된 코드를 포함한

다. 이 라이브러리는 상기 자바 바이트코드 보안 모듈에 의하여 동적으로 링크되고 동작된다.

[0061] 상기 자바 바이트코드 보호 툴(402)은 또한 보호된 바이트코드 데이터(540)를 포함한다. 상기 보호된 바이트코드 데이터(540)는 상기 툴에 의하여 생성된 보호 데이터 파일들 중의 한 종류로서 다양한 보호된 바이트코드를 포함한다.

[0062] 상기 자바 바이트코드 보호 툴(402)은 또한 런-타임 데이터(512)를 포함한다. 상기 런-타임 데이터(512)는 화이트박스 복호화 키 데이터, 무결성 검증 정적 해쉬 값들, 보호된 클래스 및 방법 정보 및 테이블과 같은 보안과 관련된 다양한 종류의 정보를 포함한다.

[0063] 상기 자바 바이트코드 보호 툴(402)은 다운로드 가능하다. 따라서 상기 보호된 자바 어플리케이션 바이트코드 스터브(408), 보호된 J2C 라이브러리(538), 보호된 바이트 코드 데이터(540) 및 런-타임 데이터(512)를 포함하는 이 툴로부터의 모든 출력은 런-타임 동안에 다운로드 가능하다.

[0064] 도 6은 도 4에 도시된 자바 바이트코드 보호 보안 모듈(404)의 관점에서 본 발명의 일 실시예에 따른 자바 어플리케이션 바이트코드를 보호하는 런-타임 프로세스를 개략적으로 나타낸다.

[0065] 상술한 바와 같이, 상기 자바 바이트코드 보호 보안 모듈(404)은 C 프로그래밍 언어로 구현되어 캐나다 온타리오의 오타와에 위치한 Cloakware Inc.로부터 제공되는 탬퍼 저항 기술들에 의하여 보호되어 둔감하고 부당 변경에 저항할 수 있다. 상기 자바 바이트코드 보호 툴(402) 및 보안 모듈에 근저의 프로그래밍 엔진들은 다른 프로그래밍 언어로 개발된 엔진들일 수 있다. 실제로, 본 발명에 밑에 있는 상기 보안 모듈(404)은 프로그래밍 언어가 상기 자바 가상 머신과 인터페이싱할 수 있기만 하면, 다른 프로그래밍 언어로 개발될 수 있다.

[0066] 런-타임을 개시하자마자, 상기 자바 가상 머신(410)은 임의의 정상적인 자바 어플리케이션을 로딩하는 것과 마찬가지로 상기 보호된 자바 어플리케이션 바이트코드 스터브(408)를 로딩한다. 이에 의하여 상기 자바 어플리케이션 바이트코드 스터브(408)가 트리거되어 JNI(602)를 통하여 상기 보안 모듈(404)과 인터페이싱하여 신뢰할수 있고 보호된 자바 어플리케이션 바이트코드를 부트스트랩한다. 상기 런-타임 동안에, 상기 보안 모듈(404)은 데이터 흐름을 관리하고 제어하여 상기 자바 바이트코드와 그 실행을 보안하고 보호하여 상기 바이트코드 및 상기 실행한 대한 동적 공격들을 방어한다.

[0067] 도 6을 참조하여, 주요한 기능적 컴포넌트들 및 데이터 파일들이 이하에서 상술된다. 상기 보안 모듈(404)로부터의/로의 데이터 및 플로우 제어는 자바 어플리케이션 바이트코드 작업 공간(604)을 경유한다. 상기 작업 공간(604)은 상기 자바 가상 머신(410) 내에서 상기 자바 어플리케이션에 대한 가상의 작업 공간이다. 상기 어플리케이션을 로딩하고 실행하는 런-타임의 다른 상태들에서는, 상기 자바 가상 머신(410) 내에 존재하는 실제의 어플리케이션 바이트코드는 다르게 관리된다. 상기 작업 공간의 각 상태는 합법적이고 전적으로-기능적인 어플리케이션 바이트코드를 포함하지만 완전한 어플리케이션 바이트코드를 포함하지는 않는다. 선택적으로, 이러한 바이트코드들의 일 부분은 예를 들어 자바 및 C 실행 옵션에 있는 전송 실행으로 활성화하는 것과 같이 빌드-타임 구성 설정들에 의존하는 보호 형태들에 항상 유지될 수 있다. 바이트코드들의 상기 부분이 실행되어야하는 경우에는, 상기 보안 모듈(404)은 상기 작업 공간 내에서 바이트코드들의 부분을 상기 자바 가상 머신(410)에 적시에 로딩하고 복원하고 실행 후에는 상기 바이트코드들의 부분을 제거한다. 또한 소정의 원본 방법 바이트코드는 자바 가상 머신(410)에서는 직접 볼 수 없는 C 기능들로 해석되고 상기 보안 모듈(404)에 의하여서만 동작할 수 있다. 이러한 접근 방법에서는, 공격자는 런-타임 동안의 주어진 임의의 순간에 상기 원본 어플리케이션 바이트코드의 조각들만을 볼 수 있기 때문에 전체 어플리케이션 바이트코드를 리버스 엔지니어링하기가 대단히 어렵게 된다.

[0068] 상기 보안 모듈(404)은 도 6에서 JNI SM 브리지(418)로 언급되는 브리지 메커니즘을 포함한다. 상기 JNI SM 브리지(418)는 자바 가상 머신(410)과 상기 보안 모듈(404) 사이에서 JNI(602)를 통하여 연결과 상호-기능을 가능하게 하는 상호작용 컴포턴트이다. JNI SM 브리지(418)의 서브-컴포넌트들은 상기 자바 가상 머신(410)과 네이티브 코드 사이에서 상호작용의 유일한 메커니즘을 제공하는 JNI(602)를 포함한다. 또한 상기 서브-컴포넌트들은 다운-콜 스터브(612)와 업-콜 스터브(608)을 포함한다. 이러한 스터브들은 네이티브 프로그래밍 코드에서 상기 보안 모듈(4040을 통하여 상기 자바 가상 머신(410)의 자바 어플리케이션 바이트코드 작업 공간(604)으로부터 상기 동적 보안 핸들러들(611)로의 다운-콜들의 방향을 재설정하고, 상기 보안 모듈(404)로부터 상기 자바 가상 머신(410)으로서 업-콜들의 방향을 재설정한다. 제3 스터브 컴포넌트는 보안 모듈 매니저(610)이다. 상기 보안 모듈 매니저(610)는 상기 보안 모듈(404)에 대한 컨트롤러이고 코디네이터이다. 상기 보안 모듈 매니저(610)는 상기 자바 어플리케이션 바이트코드 뿐만 아니라 상기 보안 모듈(404) 자체에 대한 여러 가지 지정된

보호를 관리하고 유지한다. 상기 보안 모듈 매니저(404)는 또한 상기 보안 모듈(404)의 다른 기능 컴포넌트들에 의하여 공통적으로 사용되는 유틸리티들을 포함한다.

[0069] 상기 보안 모듈(404)은 또한 복수의 동적 보안 핸들러들(611)을 포함한다. 상기 동적 보안 핸들러들(611) 각각은 동작되어 각각의 고유한 보호 기술을 수행한다. 도시된 바와 같이, 본 발명의 실시예에 따르면 동적 보안 핸들러들(611)은 화이트박스 동적 보안 핸들러(614), BIV 동적 보안 핸들러(616), AD 동적 보안 핸들러(618), SLB 동적 보안 핸들러(620), DBD 동적 보안 핸들러(622), 전송-실행 동적 보안 핸들러(624), 부분 실행 동적 보안 핸들러(626) 및 코드 변환 툴(628)을 포함한다. 상기 각 동적 보안 핸들러들(611)은 이하에서 상술된다.

[0070] 상기 빌드-타임 자바 바이트코드 보호 툴(402)과 함께 동작하여 상기 보안 모듈 매니저(610)와 동적 보안 핸들러들(611)은 새로운 보안 핸들러들을 쉽게 포함하여 보안 능력과 새로운 보호를 부가하고 확장하는 플러그-인 메커니즘을 제공하도록 설계된다.

[0071] 도 7에는 본 발명의 일 실시예에 따른 외부 안티-디버깅 모니터링의 일 실시예가 도시된다. 여기서, 자바 플랫폼 디버그 구조(Java Platform Debug Architecture; JPDA)가 상기 자바 어플리케이션을 디버깅하는 것을 용이하게 한다. 본 발명의 방법은 상기 JPDA에 기초하여 디버그 활성화와 후속적인 디버깅 활동들에 중점을 둔다. 도시된 바와 같이 자바 가상 머신 프로세스를 구동하면서 정적 및 동적으로 상기 디버깅 활동들을 캡처링하는 기화를 최대화하기 위하여 멀티레이어 방어 전략이 사용된다. 도 7에 도시된 AD 방법에는 세가지 에이전트들이 도시되어 있는데, 이 세가지 에이전트들은 수행될 정상적인 또는 법적인 디버깅 활동을 허용하도록 구성될 수 있다. 상기 세가지 에이전트들은 커널 모니터 에이전트(Kernal Monitor Agent, KMA; 702), 디버거 부착 모니터 에이전트(Debugger Attachment Monitor Agent, DAMA; 710) 및 디버깅 절차 모니터 에이전트(Debugging Procedure Monitor Agent, DPMA; 718)를 포함한다.

[0072] KMA(702)가 커널 공간(701)을 액세스하는 것에 관하여, 자바 가상 머신 프로세스가 디버깅 기능이 수행되기 전에 메모리 공간에 디버깅 라이브러리(705)를 로딩하는 것이 필요하다. 상기 KMA(702)는 자바 어플리케이션이 시작되면 유발된다. 상기 KMA(702)는 커널로부터 자신의 프로세스 맵(703)을 주기적으로 체크하여 JPDA와 관련된 라이브러리가 메모리 공간에 로딩되어야 하는지 여부를 결정한다. 이러한 적절한 관련된 액션은 이러한 라이브러리가 발견되면 취해진다.

[0073] 상기 DAMA(710)에 관하여는, 이 에이전트는 방어의 제2 선으로서 역할을 수행한다. 상기 DAMA(710)는 자바 가상 머신이 시작되면(700), 자바 가상 머신 툴 인터페이스(Java Virtual Machine Interface; JVMTI) 능력과 함께 용이해지고 로딩된다. 재호출 기능이 제공되어 런-타임 동안에 생성된 모든 쓰레드에 대하여 지속적으로 쓰레드-시작 스크린들을 모니터링한다. 상기 자바 어플리케이션 내의 부착된 JDPA 디버거의 활동들은 디버깅을 수행하는데 필수적이라고 판단되는 소정의 쓰레드들을 로딩할때마다 캡쳐될 수 있다. 이러한 측면에서, AMA는 쓰레드 시작 리스너를 활성화하고(707), 새로운 쓰레드 시작(709)을 감지하고 JPDA 관련된 쓰레드를 감지한다(711).

[0074] 상기 DPMA(710)에 관하여는, 이 에이전트는 방어의 제3 선으로서 역할을 수행한다. 상기 DPMA(710)는 JVMTI 환경 하에서 동작한다. 브레이크 포인트 라인을 가격하는 것과 같은 디버깅 절차를 모니터링하는 콜백 기능은 이러한 액션이 취해질때마다 트리거된다. 상기 쓰레드와 브레이킹 포인트의 위치와 같은 세부적인 메시지들이 수집된다. 이러한 측면에서, DPMA는 라인 브레이크 리스너를 활성화시키고(713), 디버깅 활동들을 감지하고(715), 임의의 쓰레드 및 방법 정보를 보고한다(717). 상기 KMA, DAMA 및 DPMA는 액션을 트리거할 수 있고, 자바 가상 머신(723)을 비활성화시킬 수 있다.

[0075] 상기에서 언급된 정적 및 동적 보안 핸들러들이 좀더 상세히 설명될 것이다. 상기 화이트박스 보안 핸들러는 도 8에 도시된 외부 화이트박스 암호 라이브러리와 도 9에 도시된 내부 화이트박스 암호 시설을 포함한다.

[0076] 상기 화이트박스 동적 핸들러(614)에 의하여 제공되는 도 8의 외부 화이트박스 암호 라이브러리는 JNI보안 모듈 인터페이스(804)를 통하여 화이트박스 암호화화 및 복호화 기능에 대하여 상가 자바 어플리케이션에 의하여 사용되는 라이브러리를 제공한다. 상기 화이트박스 정적 핸들러는 사용자로부터 암호 정보와 원본 키들(502)을 수신하고 필요에 따라 분포되고 등록되는 화이트박스 키 데이터(803)를 생성한다. 이 화이트박스 키 데이터(803)는 상기 암호 라이브러리가 보안 암호 작업을 위하여 사용할 수 있다.

[0077] 상기 내부 화이트박스 암호 설비는 도 9에 도시된 바와 같이 화이트박스 정적 핸들러(508)와 복수의 정적 및 동적 컴포넌트들을 포함한다. 상기 화이트박스 정적 핸들러(508)는 사용자로부터 암호 정보와 원본 키들(502)을 수신하고 화이트박스 암호화 키 데이터(904)를 생성한다. 이 암호화 키 데이터(904)는 서로 다른 보호 기술들의 일부로서 서로 다른 형태의 어플리케이션 바이트코드들에 따라 다른 정적 보안 핸들러들(906)이 암호화 동작에

사용할 수 있다. 상기 화이트박스 정적 핸들러(508)는 상기 보안 모듈(914)이 동적 보호를 수행하는 동안 동적 보안 핸들러들(611)이 복호화 동작을 수행하는 데 사용하는 화이트박스 복호화 키 데이터(908)를 생성하고 화이트박스 보안 모듈 유틸리티(630)를 제공한다.

[0078] 상기 자바 바이트코드 보호 툴(402)은 또한 도 10에 도시된 프리-프로세싱 방법을 포함한다. 상기 프리-프로세싱 방법(1001)은 상기 원본 자바 어플리케이션 바이트코드 아키브 파일(1005)을 수신하여 이를 원본 어플리케이션 바이트코드의 내부 표현(internal representation; IR)으로 해석한다. 그 다음에 특정 클래스들과 방법들이 사용자 옵션(1003)에 따라서 보호와 보호 방법에 따라 표시된다. 보호 마크 정보(1004)는 이에 의하여 생성된다. IR 형태의 원본 어플리케이션 바이트코드(1000)와 보호 마크 정보(1004)가 원하는 보호를 위하여 정적 보안 핸들러들(522)에 의하여 사용된다.

[0079] 상기 자바 보호 툴(402)의 각 정적 보안 핸들러 내에는, BIV가 제공된다. 도 11은 BIV 정적 보안 핸들러(524)의 작업 흐름을 도시한다. 도 12는 BIV 동적 보안 핸들러(616)의 작업 흐름을 도시한다. 여기서 BIV는 런-타임 동안에 자바 바이트코드에 대한 동적 무결성 검증 능력을 도입하여 고유한 탬퍼 저항 보호를 제공한다. 일반적으로, 빌드-타임에 BIV 데이터(1202)가 생성되고 보호되며, BIV 액션들이 자바 바이트코드로 변환되는 BIV 보호를 필요로하는 클래스들과 방법들을 표시하기 위하여 툴이 사용된다. 런-타임에, 동적 보안 해쉬 값들이 각각의 바이트코드에 대하여 적시에 계산되는 BIV 보호된 클래스들과 방법들을 위하여 자바 바이트코드 보호 보안 모듈(404)을 통하여 BIV 액션들이 트리거된다. 정적 및 동적 해쉬 값들은 보안된 형태로 표현되어 성공 및/또는 실패 콜-백 기능과 함께 탬퍼 저항 게이트 키퍼(Tamper Resistance Gate Keeper, TRGK; 1216)에 제공된다. 상기 TRGK(1216)은 외부적으로 상기 정적 및 동적 해쉬 값들을 비교하지 않고도 BIV 체크가 성공인지 실패인지를 판단한다. 이러한 판단은 특정하게 설계된 수학적 계산의 형태의 적절한 알고리즘을 통하여 이루어질 수 있다. 만일 상기 정적 및 동적 해쉬 값들이 서로 동일하다면, 이는 일반적으로 BIV 체크가 성공하였고 성공 콜-백 함수가 호출될 수 있다. 만일 상기 정적 및 동적 해쉬 값들이 서로 동일하지 않다면, 이는 특정 클래스나 방법에 대한 탬퍼링(부당 변경)이 감지되었고, BIV 체크가 실패하였음을 나타낸다. 따라서 실패 콜-백 함수가 호출될 수 있다. 이러한 콜-백 함수들은 감지된 탬퍼링 공격들에 대한 사용자-지정의 대응책들이다.

[0080] 본 발명에서는 자바 바이트코드의 동적 보안 해쉬 값들(1214)을 계산하는 과정은 계산이 수행가능한 것들에 할당된 메모리로부터 직접 네이티브 코드를 골라내는 정상적인 네이티브 바이너리 코드에 대한 계산과는 다르다. 일반적으로, 어플리케이션 코드는 자바 런-타임에 메모리로부터 직접적으로 코드 세그먼트를 획득할 수 없다. 그 대신에, 어플리케이션 코드는 자바 가상 머신(410) 메커니즘을 통하여 클래스 또는 방법 바이트코드를 획득한다. 본 발명에서는, 상기 보안 모듈이 자바 가상 머신(410)에 대한 업-콜을 사용하여 바이트코드를 검색하고 상기 보안 동적 해쉬 값들(1214)을 계산하고 기-등록된 해쉬 값에 대하여 상기 검색된 바이트코드의 무결정 검증 체크를 수행하여 이 능력과 상기 JNI 인터페이스에 대하여 영향을 미친다.

[0081] 도 11을 참조하면, BIV 정적 보안 핸들러(524)는 바이트코드 표시를 포함한다. 상기 BIV 정적 보안 핸들러(524)의 주요한 기능중의 하나는 어플리케이션 바이트코드에 진입하고 보호 마크 정보(1105)를 사용하여 클래스들과 방법들을 체크하여 어떤 클래스와 방법이 BIV 보호를 필요로 하는지를 판단하는 것이다. 클래스 또는 방법이 상기 BIV 보호를 필요로 하면, 상기 특정 클래스 또는 방법 바이트코드(1106, 1107)에 보안 해쉬 계산을 적용하여 특정한 해쉬 값을 계산한다. 일반적으로, 당 분야에 잘 알려진 보안 해쉬 계산 알고리즘이 사용된다. 유기적이고 구조적인 방법으로 상기 결과적인 해쉬 값들이 BIV 데이터(1108)로서 저장되어 런-타임 동안에 효율적으로 사용될 수 있도록 한다.

[0082] 상기 BIV 정적 보안 핸들러의 BIV 데이터(1108)는 클래스 및 데이터 정적 해쉬 값 및 화이트박스 BIV 복호화 키 데이터와 같은 다른 정보를 포함하는 데이터이다. 이러한 데이터는 런-타임에 상기 동적 BIV 보안 핸들러에 의하여 사용된다. 좀 더 효율적으로 사용되기 위하여, BIV 데이터는 보호되어야 하는 클래스와 방법 및 해당하는 정적 해쉬 값에 상응하는 정보로 구조화된다.

[0083] 상기 BIV 정적 보안 핸들러(524)는 BIV 데이터(1108)를 변화하고 암호화하는 역할도 수행한다. BIV 데이터는 네트워크를 통하여 전송되거나 다운로드될 수 있다. 따라서 본 발명에서는 BIV 데이터(1108)를 패킹하는 일부로서 BIV 데이터를 변환하고 암호화할 수 있다. 패킹 타임 동안, BIV 정적 보안 핸들러는 BIV 데이터에 대한 이중 보호를 수행하여 민감한 BIV 데이터에 대한 정적 공격들을 방지한다. 우선, 상기 BIV 정적 보안 핸들러(524)는 상기 정적 해쉬 값들에 대한 데이터 변환을 수행하여 이 값들이 런-타임에 상기 동적 BIV 보안 핸들러(616)에 의하여 변화된 형태로 동작하도록 한다. 이에 의하여 실제의 플레인 값들이 절대로 노출되지 않도록 한다. 다음으로, 상기 BIV 정적 보안 핸들러(524)는 상기 변환된 값들을 암호화하여 이 값들이 동적으로 사용되기 전에 상기

변환된 값들에 발생할 수 있는 임의의 탬퍼링도 방지하도록 한다.

[0084] 상기 BIV 데이터(1108)는 상기 동적 보안 핸들러들(611)에 의하여 런-타임에 사용되는 일종의 런-타임 데이터이다. 런-타임 데이터는 사용자 옵션에 따라 유기화되어 단일 파일 또는 다중 파일들에 저장될 수 있다. 다중 런-타임 데이터는 데이터정보가 더욱 파인-그레인(fine-grain)으로 업데이트되고 다운로드되는 이점을 갖을 수 있다. 예들 들어, 상기 BIV 데이터(1108)는 보호되어야 하는 자바 클래스 각각에 대하여 구조화될 수 있다. 이러한 방식으로, BIV보호는 각 클래스 가반으로 좀더 쉽게 수행될 수 있다.

[0085] 상기 BIV 정적 보안 핸들러(524)는 고유의 BIV 트리거링을 제공한다. 외부 BIV API와 외부 BIV 트리거를 통하여 런-타임에 BIV를 두가지 접근으로 트리거한다. 본 발명의 지정된 시스템의 일부인 제1 접근 방법으로서, 사용자가 BIV 체크를 수행할 명확한 아이디어를 가지는 자바 코드 내의 적절한 장소들에서 사용할 일 세트의 외부 BIV API들이 제공된다. 사용자는 어떤 자바 클래스나 방법이 BIV 체크를 필요로 하는지를 나타낼 수 있다. 사용자는 콜-백 함수를 사용하여 완화 액션을 전적으로 제어할 수 있다. 다른 접근 방법은 사용자가 외부 API들을 호출하는 트리거링에 대한 대안이다. 그 대신에, BIN 트리거들은 상기 자바 보호 보안 모듈의 일부 함수들 내에 미리 설치될 수 있다. 자바 어플리케이션이 이러한 함수들을 호출할 때마다, 상기 내부 BIV 액션들은 미리 정렬된 형식으로 트리거될 수 있다. 일부 완화 액션들이 미리 정의되고 보안 모듈에 의하여 내부적으로 취해질 수 있다. 하지만, 사용자는 상기 완화 액션에 대하여 부분적인 제어권을 갖는다. 이는 사용자에게 상기 보안 모듈이 취하는 완화 액션을 미리 설정하여 상기 완화 액션이 상기 설정에 따라 동작하게 함으로써 가능하다. 일반적으로, 사용자는 상기 외부 BIV API를 사용할지 어디서 사용할지에 대한 전적인 제어권을 갖고 빌드-타임에 상기 내부 BIV 트리거들을 사용할지 여부에 대한 간접적인 제어권을 갖는다. 사용자는 상기 보안 모듈에 의하여 은닉되고 제어되는 상기 내부 BIV을 어디서 트러거할 지에 대한 제어권은 전혀 갖지 않는다.

[0086] 도 12를 참조하면, 상기 BIV 동적 보안 핸들러(616)는 BIV 초기화를 포함한다. BIV 초기화는 보안 정적 BIV 데이터(1202)를 로딩하고 화이트박스 복호화 키 데이터(1203)를 사용하여 복호화하고 복호화된 데이터를 보안된 형태로 메모리에 로딩하기 위하여 제공된다. BIV 초기화는 두 가지 방법으로 구현될 수 있다. 보안 모듈 초기화의 부분으로서 또는 동적 BIV 동안의 요구에 의해서이다. 제1 방법에 관하여는, BIV 초기화는 보호된 자바 어플리케이션을 로딩할 때 SM 초기화의 부분으로서 수행될 수 있다. 두 번째 방법에 관하여는, 동적 BIV 동안에 요구에 의하여 필요로하는 것을 로딩하여 수행될 수 있다. 이는 BIV가 클래스에 대하여 필요로하는 경우에 수행될 수 있다. BIV 데이터 파일을 클래스 레벨로 조직될 수 있다. 특정 클래스에 대하여 BIV 데이터는 이 클래스만에 대하여만 로딩되고 복호화될 수 있다. 상기 제2 방법은 상기 클래스 바이트코드가 변화하는 경우에 사용자에게 BIV 데이터에 필요로 하는 작은 변화를 조율하는데 좀 더 유연성을 제공할 수 있다.

[0087] 상기 BIV 동적 보안 핸들러는 또한 동적 BIV를 수행한다(1210). 상기에서 논의된 바와 같이, 클래스 또는 방법에 대한 동적 BIV는 상기 보호된 자바 어플리케이션으로부터 미리-배열된 내부 BIV 트리거를 포함하는 상기 보안 모듈에 대한 외부 BIV API 호출이나 다른 함수 호출에 의하여 개시될 수 있다. 동적 BIV를 수행하는 것은 다음의 주요한 액션들을 적어도 포함한다: 최신의 바이트코드를 획득하는 것; 동적 보안 해쉬 값을 계산하는 것; 및 탬퍼 저항 게이트 키퍼(Tamper Resistance Gate Keeper(TPGK); 1216)을 제공하는 것을 포함한다.

[0088] 최신의 바이트코드를 획득하는 것은 업-콜을 통하여 발생한다. 상기 보안 모듈 내에서 클래스 또는 방법에 대한 보안 동적 해쉬 값을 계산하기 위하여 JNI를 통하여 자바 가상 머신(410)에의 업-콜을 통하여 클래스 또는 방법에 대한 최신 바이트코드를 획득하여야 한다. 상기 자바 가상 머신(410)에 로딩된 상기 클래스 또는 방법을 실행하는 동안에 상기 동일한 바이트코드는 바이너리 형태로 번역되거나 컴파일되어야 한다. 상기 바이트코드에 대한 탬퍼링 공격이 전혀 없다면, 상기 바이트코드는 상기 정적 보안 해쉬 값이 계산된 바이트코드와 동일하여야 한다.

[0089] TRGK(1216)을 제공하는 것은 두 입력을 수반한다. 상기 TRGK(1216)는 특정 클래스나 방법의 바이트코드의 무결성이 손상되었는지 여부를 검증하기 위하여 상기 특정 클래스나 방법에 대하여 정적 및 동적 보안 해쉬 값들(SSHV(1212), DSHV(1214))를 모두 사용한다. 상기 바이트코드에 대하여 탬퍼링이 발생하면, 상기 바이트코드에 대한 DSHV(1214)는 SSHV(1212)와 동일할 수 없다. 상기 TRGK(1216)는 바이트코드에 대한 임의의 탬퍼링을 감지할 수 있다. 상기 BIV 검증이 통과되면, TRGK(1216)는 성공 콜-백 함수를 호출하거나 원본 BIV 트리거로 회귀한다. 상기 BIV 검증이 통과되지 않으면, TRGK(1216)는 사용자의 완화 액션으로서 실패 콜-백 함수를 호출한다.

[0090] 상기 BIV 동적 보안 핸들러(616)는 BIV 종료의 형태로 종료 단계를 포함한다. 상기 보안 모듈의 일부로서 상기 BIV 종료는 상기 BIV 동적 보안 핸들러에 의하여 사용된 메모리 공간과 다른 정보를 삭제하는 것을 포함한다.

[0091] 도 13을 참조하면, SLB 정적 보안 핸들러(528)가 도시된다. 상기 SLB 정적 보안 핸들러(528)는 상기 원본 자바 어플리케이션 바이트코드(1301)의 내부 표현, 화이트박스 암호화(1302), 복호화 키 데이터(1304) 및 보호 마킹 정보(1306)를 수신한다.

[0092] SLB 정적 보안 핸들러(528)의 중요한 출력은 어플리케이션 스터브(1308)이다. 상기 어플리케이션 스터브(1308)는 런-타임 동안에 상기 보안 모듈을 통하여 로딩 프로세스를 시작시키는 부트스트래핑 클래스를 포함한다. 상기 어플리케이션 스터브(1308)은 상기 자바 가상 머신(410)에 의하여 로딩된다. 상기 어플리케이션 스터브(1308)는 독립적으로 시작되거나 다른 자바 어플리케이션을 통하여 시작되는 어플리케이션을 활성화하는데 필요한 각 외부 공개 API를 포함한다. 상기 어플리케이션 스터브(1308)는 상기 보안 모듈에 대한 다운-콜 함수들을 호출하는 방법들을 포함한다. 이 방법들은 상기 자바 어플리케이션을 복호하하고 실행을 위하여 자바 가상 머신에 로딩한다.

[0093] 상기 어플리케이션 스터브를 준비하기 위하여, 상기 SLB 정적 보안 핸들러(528)는 어플리케이션 바이트코드 작업 프레임(1310)과 암호화된 어플리케이션 바이트코드 작업 프레임(1312)을 포함한다. 상기 어플리케이션 바이트코드 작업 프레임(1310)은 상기 원본 어플리케이션 바이트코드(1301)와는 다르다. 일반적으로, 상기 어플리케이션 바이트코드 작업 프레임(1310) 내의 클래스는 보호를 필요로 하지 않기 때문에 원본 어플리케이션 바이트코드와 동일하다. 만일 클래스가 보안적으로 로딩되어야 하는 경우에, 클래스 스터브가 상기 원본 바이트코드를 대체하고, 따라서 상기 클래스 바이트코드가 원본 바이트코드가 된다. 상기 암호화된 프레임(1312)은 빌드-타임 동안에 정적 보안 핸들러(528)를 통하여 어플리케이션 화이트박스 암호화 키 데이터(1302)를 사용하여 어플리케이션 바이트코드 작업 프레임(1310)을 암호화하여 획득할 수 있고, 상기 암호화된 프레임(1312)은 런-타임 동안에 동적 보안 핸들러(620)를 통하여 어플리케이션 화이트박스 복호화 키 데이터(1304)를 이용하여 복호화된다.

[0094] 어플리케이션 스터브(1308)에 추가하여, 어플리케이션 페이로드(1314)도 생성된다. 어플리케이션 페이로드(1314)는 암호화된 어플리케이션 작업 프레임(1312)과 어플리케이션 화이트박스 복호화 키 데이터(1316)를 포함한다. 보호된 어플리케이션 페이로드의 어플리케이션 화이트박스 복호화 키 데이터(1316)는 화이트박스 정적 보안 핸들러(508)에 의하여 생성되어 화이트박스 복호화 키 데이터(1302)의 일부로서 상기 SLB 정적 보안 핸들러(528)에 전달된 키 데이터이다. 런-타임에 어플리케이션 화이트박스 복호화 키 데이터(1316)는 암호화된 어플리케이션 바이트코드 작업 프레임(1312)을 복호화 하는데 사용된다.

[0095] 도 13에 도시된 바와 같이, 근본적인 코드는 클래스 바이트코드(1318), 클래스 스터브(1320) 또는 암호화된 클래스 바이트코드(1322)로서 형성될 수 있다. 클래스 바이트코드(1318)는 원본 바이트코드이다. 클래스 스터브(1320)는 런-타임 동안에 보안 모듈을 통하여 신뢰받는 클래스 로딩 프로세스를 동작시키는 부트스트래핑 방법을 포함하여 필요한 경우 암호화된 클래스 바이트코드(1322)를 로딩한다. 패키징 타임 동안에, 클래스 바이트코드(1318)는 분석된다. 마크된 방법들은 보안 모듈에 대한 다운-콜 방법들을 호출하는 방법으로 대체되고, 보안 모듈은 패키징 타임에 지정된 보안 핸들러 방법을 통하여 원본 바이트코드 기능을 호출한다. 암호화된 클래스 바이트코드(1322)는 빌드-타임 동안에 정적 보안 핸들러(528)를 통하여 클래스 화이트박스 암호화 키 데이터(1302)를 사용하여 클래스 바이트코드(1318)를 암호화하여 획득할 수 있고, 런-타임 동안에 동적 보안 핸들러(620)를 통하여 클래스 화이트박스 복호화 키 데이터를 사용하여 복호화될 수 있다.

[0096] 암호화된 클래스 바이트코드 프레임(1324)은 SLB 정적 보안 핸들러(528)에 의하여 생성될 수 있다. 암호화된 클래스 바이트코드 프레임(1324)은 하나 이상의 클래스들에 대한 암호화된 클래스 바이트코드와 클래스 화이트박스 복호화 키들을 포함한다. 사용자는 암호화된 클래스 바이트코드 내에 얼마나 많은 클래스들이 포함될지를 제어하는 옵션을 가진다. 사용자는 런-타임 동안에 클래스들이 개별적으로 또는 함께 로딩하는 것에 대한 옵션을 갖는다. 클래스 화이트박스 복호화 키 데이터는 화이트박스 정적 보안 핸들러(508)에 의하여 생성되어 화이트박스 복호화 키 데이터(1304)의 일부로서 SLB 정적 보안 핸들러(528)에 전달된다. 런-타임에 클래스 화이트박스 복호화 키 데이터(1304)는 암호화된 클래스 바이트코드(1322)를 복호화하는데 사용된다. 사용자는 하나 또는 다중의 클래스 화이트박스 암호화 및 복호화 키들을 생성할지 여부에 대한 옵션을 갖는다.

[0097] 도 14를 참조하면, SLB 동적 보안 핸들러(620)의 작업 흐름이 도시되너 있다. SLB 동적 보안 핸들러(620)는 바이트코드 실행 동안에 JNI를 통하여 자바 가상 머신(410)과 연결되는 보안 모듈의 기능적 컴포넌트이다. SLB 동적 보안 핸들러(620)는 보호된 자바 어플리케이션 바이트코드가 자바 가상 머신(410) 내의 작어 공간으로의 로딩을 관리하고 제어한다. 이 능력에 의하여, 원본 자바 어플리케이션 바이트코드가 보호되고 보호된 형태로 배포되어 어플리케이션 바이트코드가 자바 가상 머신(410)에 로딩되기 전에 발생가능한 어플리케이션 바이트코드에 대한 임의의 정적 공격을 방어할 수 있다. SLB 동적 보안 핸들러(620)는 보안된 어플리케이션 로딩과 보안된

클래스 로딩의 두 가지 주요한 기능적 컴포넌트들을 포함한다.

[0098] 보안 어플리케이션 로딩은 클래스 경로에 위치하고 자바 가상 머신(410)에 의하여 정상적으로 로딩되는 보호된 어플리케이션 스터브(1404)를 포함한다. 로딩 후에 메인 부트스트래핑 방법이 실행되고, 어플리케이션 부트스트래핑 방법(1403)이 JNI SM 브리지(418)를 통한 다운-콜 API를 통하여 호출된다. 이에 의하여 다음의 SLB 동적 보안 핸들러(620)의 어플리케이션 로딩 액션이 트리거된다. 먼저, 보호된 어플리케이션 페이로드(1408)가 보호 데이터 폴더로부터 로딩된다. 이는 페이로드(1408)로부터 메모리 버퍼로의 암호화된 어플리케이션 바이트코드 작업 프레임(1410)의 로딩과 어플리케이션 화이트박스 복호화 키 데이터(1304)를 사용하여 메모리 내에서 암호화된 어플리케이션 바이트코드 작업 프레임(1410)의 복호화를 포함한다. 다음에, 복호화된 어플리케이션 바이트코드 작업 프레임(1412)이 진입되어 특정 SM 클래스 로더(1414)를 사용하여 상기 작업 프레임으로부터 어플리케이션 작업 공간으로 클래스 바이트코드와 클래스 스터브들 각각이 로딩된다. 상기 SM 클래스 로더(1414)는 보안 모듈을 사용하여 암호화된 바이트코드를 로딩하고, 바이트코드를 복호화하고 바이트코드를 자바 가상 머신(410)에 로딩한다. SM 클래스 로더(1414)에 BIV 보호를 추가하기 위하여 추가적인 보안 체크들이 병합될 수 있고, 로딩 및 런 타임 동안에 클래스 로더 하이어아키와 무결성에 대한 체크가 수행될 수 있다. 결국, 작업 공간 내에서 메인 어플리케이션의 메인 방법에 대한 수행이 통과된다.

[0099] 보안 클래스 로딩은 도 15에 도시된 바와 같이 클래스 부트스트래핑 방법에 대한 트리거링을 수반한다. 일반적으로, 암호화된 클래스 바이트코드 프레임은 미리 설치되거나 보호된 어플리케이션을 실행하기 이전에 디바이스에 다운로드될 수있거나, 수행 도중에 디바이스에 다운로드될 수 있다. 이는 어플리케이션의 기능적 본질에 좌우된다. 보호된 어플리케이션의 수행 도중에 클래스 스터브를 구비하는 클래스가 필요한 경우에는, 클래스 부트스트래핑 방법이 트리거되고 JNI SM 브리지를 통하여 다음의 단계들(1500)이 실행되어 암호화된 바이트코드 클래스로부터 필요한 클래스를 로딩한다. 첫 번째로, 해당하는 암호화된 클래스 바이트코드 프레임이 메모리 버퍼에 로딩된다. 다음으로, 바로 그 프레임 메모리에 포함된 암호화된 클래스들 각각이 특정한 클래스 화이트박스 복호화 키 데이터 각각을 사용하여 복호화된다. 복호화된 클래스 바이트코드는 다음에 SM 클래스 로더를 사용하여 어플리케이션 작업 공간으로 로딩된다. 그 이후로, 상기 작업 공간 내에서 어플리케이션의 실행이 계속된다.

[0100] 모든 코드가 먼저 로딩되어야 하는 네이티브 어플리케이션을 구동하는 것과는 달리, 자바 가상 머신은 온-더-플라이 방식으로 새로운 클래스를 로딩하는 것을 허용한다. 이에 의하여 필요한 경우에만 클래스를 로딩함으로써 어플리케이션을 동적으로 확장할 수 있다. 더욱이, 자바의 이러한 특성은 코드 리프팅 공격에 대하여 SLB 보안 클래스 로딩을 사용할 좋은 기회를 제공한다. 더욱이, 본 발명은 보호된 클래스가 SLB 방식으로 안전하게 로딩되고 실행된 후에, 클래스 스터브로 복원함으로써 보호 상태에서 클래스를 유지할 수 있는 옵션을 제공한다. 이러한 방식으로, 자바 이미지 내에서, 실행 순간에만 클래스의 원본 바이트코드를 이용할 수 있고, 다른 때에는 원본 바이트코드는 보호된 형태로 유지된다.

[0101] 일반적으로, DBD는 보안된 방법이 자바 프로그램을 구동하는 것에 의하여 시작될 때만 보호된 방법 바이트코드를 기술하는 것을 수반한다. 이에 의하여 어플리케이션의 암호화되지 않은 모든 바이트코드는 한번에 메모리에 절대로 존재할 수 없다.

[0102] 도 16을 참조하면, DBD 정적 보안 핸들러(530)의 빌드-타임 작업 흐름이 도시된다. 빌드-타임 동안에, 보호되지 않은 클래스 바이트코드 파일(1602) 각각은 내부 버퍼에 로딩되고, 보호 마킹 정보(1306)를 사용하여 DBD에 의하여 보호되어야 할 클래스에 대한 새로운 클래스 바이트코드 작업 프레임이 구축된다. 여기서 마크된 방법들은 런-타임에 DBD 동적 보안 핸들러(530)의 동작을 트리거할 다운 콜 방법들을 호출할 방법 스터브(1604)로 대체된다. 보호되어야 할 각 자바 방법에 대하여, 방법 화이트박스 복호화 키들을 이용하고 보호된 바이트코드 데이터의 일부로서 배포될 화이트박스 복호화 키 데이터(1608)와 함께 패키지되는 암호화된 방법 바이트코드 프레임(1606)에 암호화된 방법을 저장함으로써 각 자바 방법에 대한 바이트코드가 암호화된다. 상기 원본 바이트코드 클래스는 배포를 위하여 보호된 클래스 바이트코드 작업 프레임(1610)으로 대체된다.

[0103] 도 18은 DBD 동적 보안 핸들러(622)의 실행 시간 작업 흐름을 나타낸다. 자바 가상 머신 상에서 보호된 자바 어플리케이션을 실행하는 도중에 암호화된 DBD 자바 방법이 호출되면, 방법 스터브가 제일 먼저 실행되고, 다운-콜과 방법 부트스트래핑(1802)이 DBD 동적 보안 핸들러(622) 내에서 호출된다. 화이트박스 방법 복호화 키 데이터를 이용하여 암호화된 방법 바이트코드 프레임(1804)로부터 암호화된 방법을 식별하고 복호화하고 자바 가상 머신에 대한 실제 바이트코드로 복원한다. 자바 가상 머신에 대한 클래스 바이트코드를 복원하는 것을 구현하는 것은 자바 가상 머신 명칭 공간에서의 명칭 충돌을 방지하기 위하여 클래스를 재명명하면서 자바 가상 머신에 대한 클래스의 카피를 복원하는 것을 수반한다. 도 17에는 부분적으로 복호화된 클래스가 새로운 클래스 명칭으

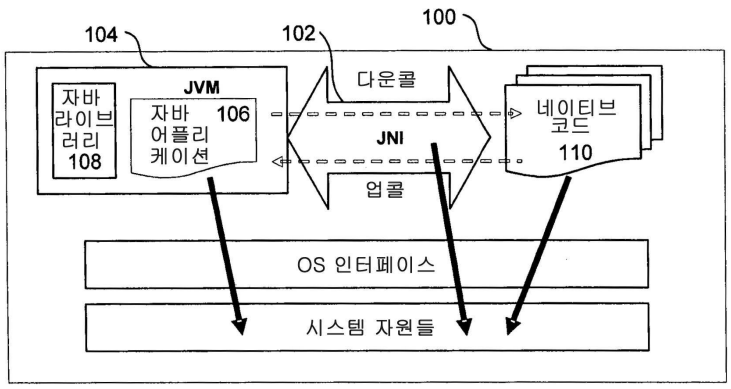
로 자바 가상 머신에 로딩된 예가 도시되어 있다.

[0104] 도 18에서는 필요에 따라, 일단 원본 바이트코드가 자바 가상 머신에 복원되면, DBD 동적 보안 핸들러(622)는 클래스 상태를 실제 바이트코드 인스턴스에 카피할 수 있다. 이 옵션은 빌드-타임시에 결정된다. DBD 동적 보안 핸들러(622)는 자바 가상 머신(410)에서 암호화되지 않은 방법을 호출한다. 일단 방법 호출이 완료되면, DBD 동적 보안 핸들러(622)는 상기 암호화되지 않는 클래스 인스턴스로부터의 실제 상태를 암호화된 상태로 복원하고, 다운-콜 방법으로 제어가 복귀된다. 도 17은 암호화되지 않은 방법을 호출하기 전에 방법 호출과 상태 카피 동작(1700)의 일예를 나타낸다. 일단 암호화되지 않은 방법이 실행을 완료하면, DBD 동적 보안 핸들러(622)에 의하여 보호된 방법 스터브를 구비하는 클래스 인스턴스로 상태가 카피된다. 보안 핸들러가 자바 가상 머신(410)으로부터 암호화되지 않은 클래스와 인스턴스를 제거하는 동안에 제어는 보호된 방법으로 복귀된다.

[0105] 상기에서는 본 발명의 실시예들을 참조하여 설명하였지만, 해당 기술분야에서 통상의 지식을 가진 자는 하기의 특허청구범위에 기재된 본 발명의 사상 및 영역으로부터 벗어나지 않는 범위 내에서 본 발명을 다양하게 수정 및 변경시킬 수 있음을 이해할 것이다.

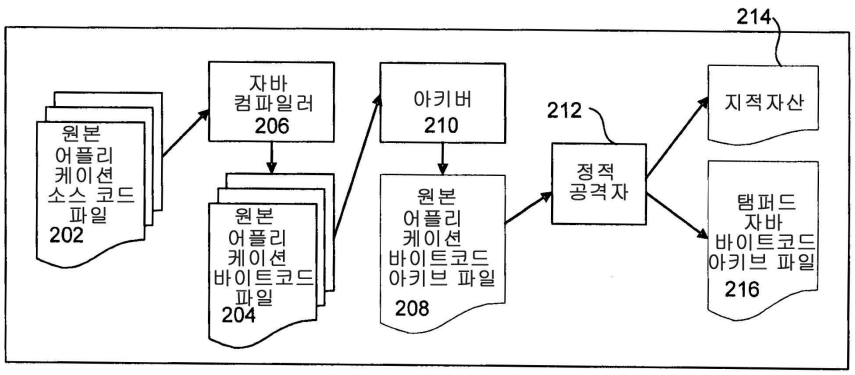
도면

도면1



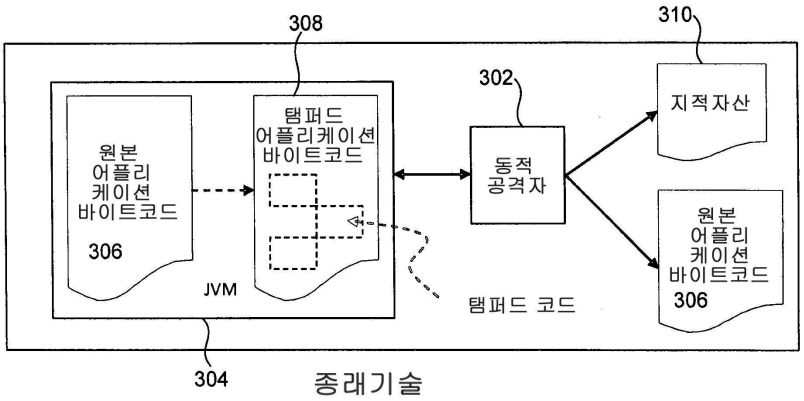
종래기술

도면2



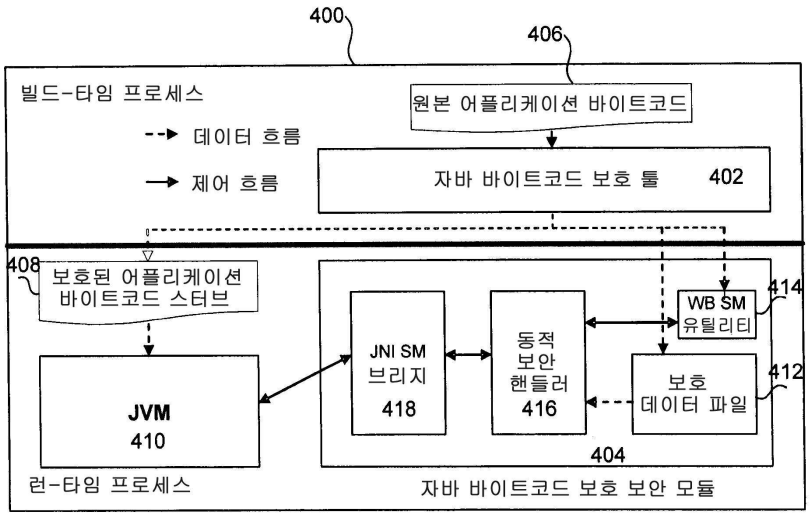
종래기술

도면3

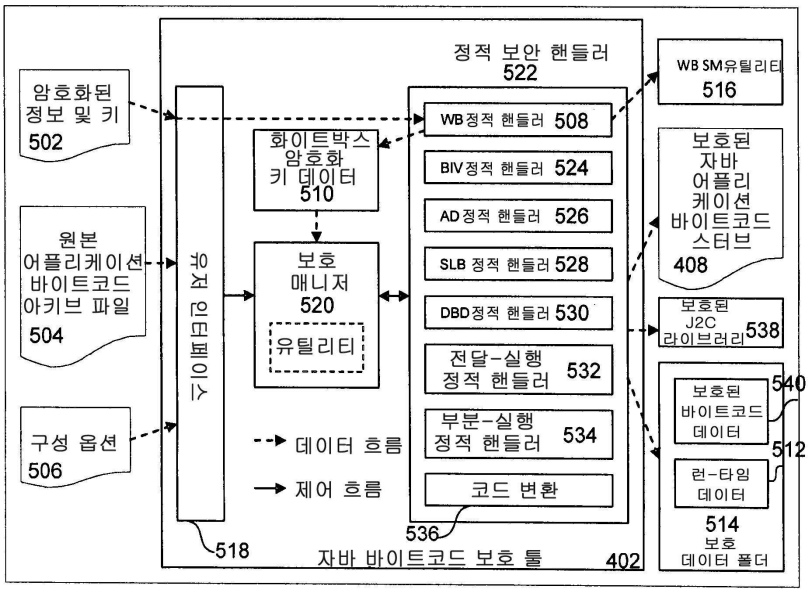

308
310
302
원본
어플리
케이션
바이트코드
306
템퍼드
어플리케이션
바이트코드
JVM
동적
공격자
지적자산
원본
어플리
케이션
바이트코드
306
템퍼드 코드
304

종래기술

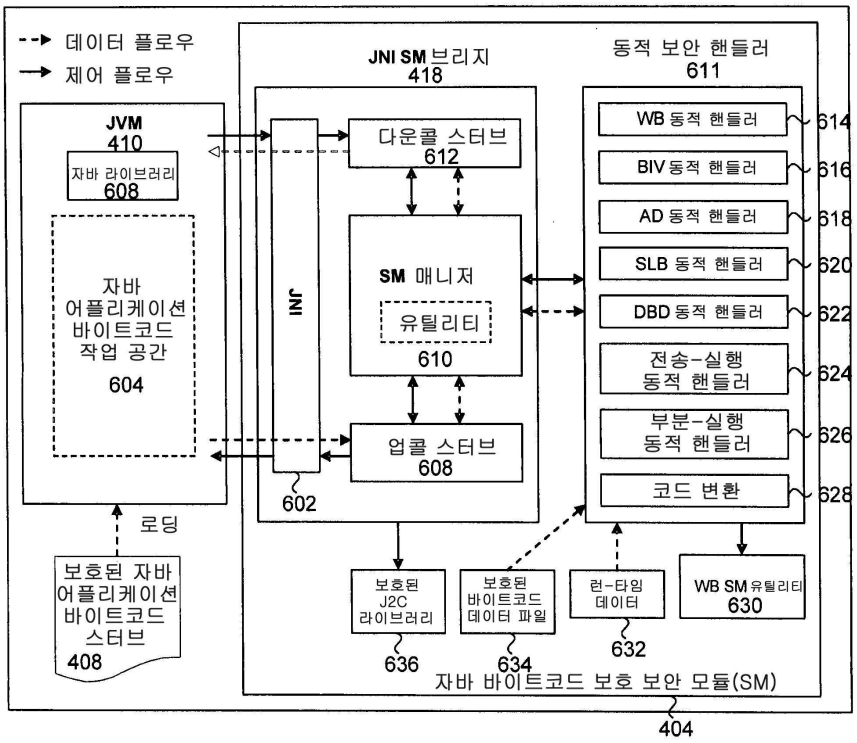
도면4


400
406
빌드-타임 프로세스
데이터 흐름
제어 흐름
원본 어플리케이션 바이트코드
자바 바이트코드 보호 툴
402
408
보호된 어플리케이션
바이트코드 스터브
JVM
410
JNI SM
브리지
418
동적
보안
핸들러
416
WB SM
유틸리티
414
보호
데이터 파일
412
404
런-타임 프로세스
자바 바이트코드 보호 보안 모듈

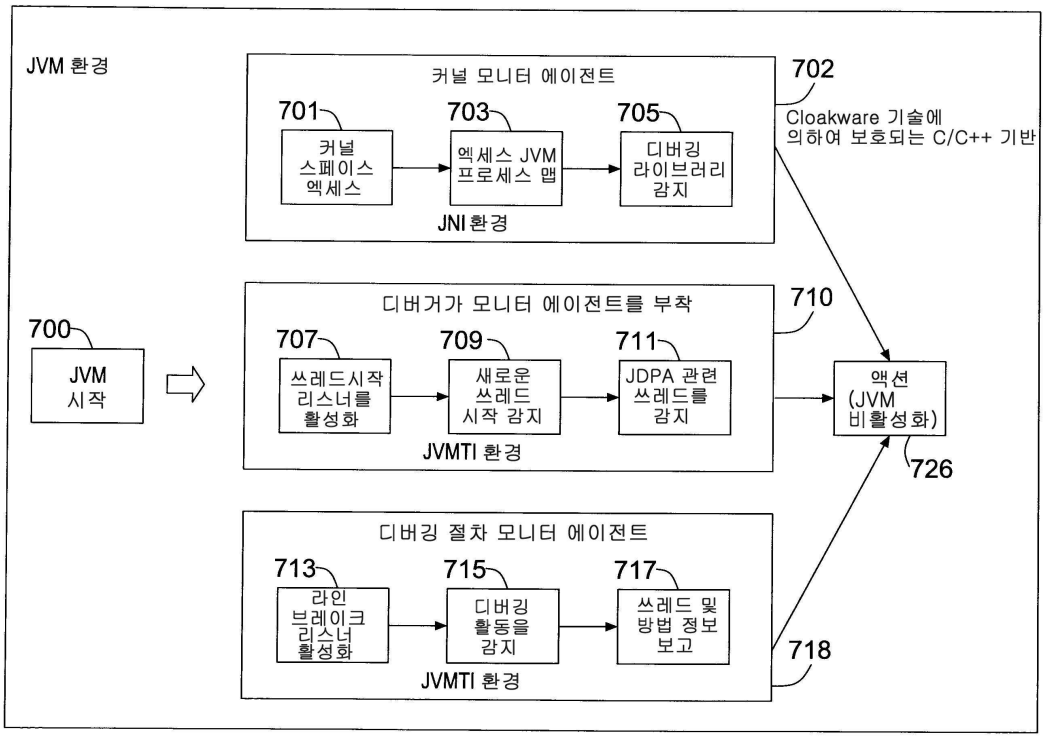
도면5


정적 보안 핸들러
522
암호화된 정보 및 키
502
원본 어플리케이션 바이트코드 아키브 파일
504
구성 옵션
506
유저 인터페이스
518
화이트박스 암호화 키 데이터
510
보호 매니저
520
유틸리티
WB정적 핸들러 508
BIV정적 핸들러 524
AD정적 핸들러 526
SLB 정적 핸들러 528
DBD정적 핸들러 530
전달-실행 정적 핸들러 532
부분-실행 정적 핸들러 534
코드 변환
536
데이터 흐름
제어 흐름
자바 바이트코드 보호 툴
402
WB SM유틸리티 516
보호된 자바 어플리케이션 바이트코드 스터브
408
보호된 J2C 라이브러리 538
보호된 바이트코드 데이터 540
런-타임 데이터 512
514
보호 데이터 풀더

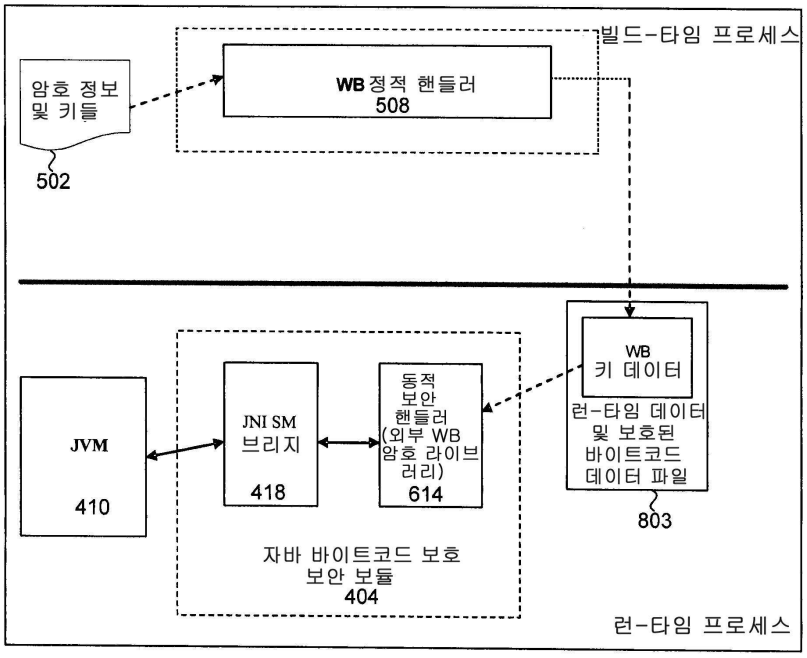
도면6


데이터 플로우
제어 플로우
JVM
410
자바 라이브러리
608
자바 어플리케이션 바이트코드 작업 공간
604
로딩
보호된 자바 어플리케이션 바이트코드 스터브
408
JNI SM 브리지
418
JNI
602
다운콜 스터브
612
SM 매니저
유틸리티
610
업콜 스터브
608
동적 보안 핸들러
611
WB 동적 핸들러 614
BIV 동적 핸들러 616
AD 동적 핸들러 618
SLB 동적 핸들러 620
DBD 동적 핸들러 622
전송-실행 동적 핸들러 624
부분-실행 동적 핸들러 626
코드 변환 628
보호된 J2C 라이브러리
636
보호된 바이트코드 데이터 파일
634
런-타임 데이터
632
WB SM 유틸리티
630
자바 바이트코드 보호 보안 모듈(SM)
404

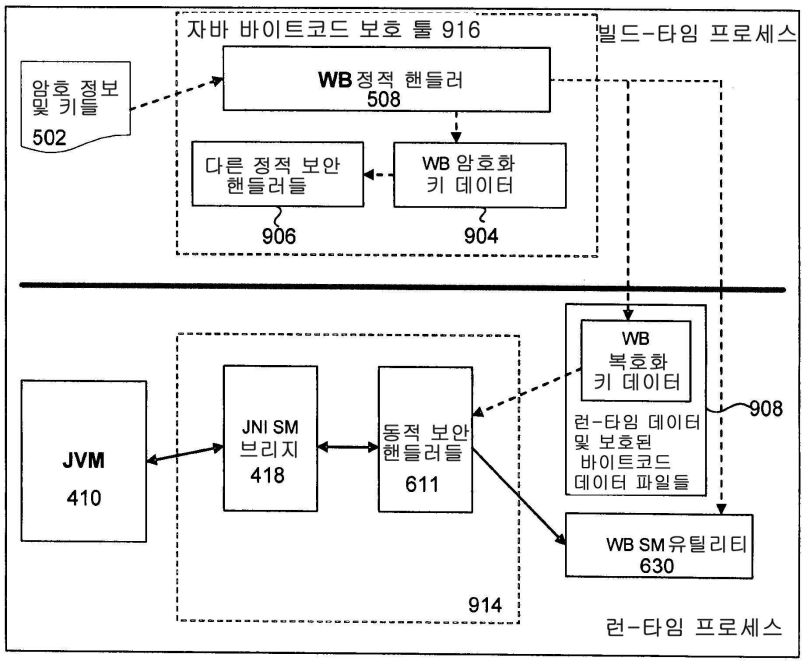
도면7


JVM 환경
커널 모니터 에이전트
701
커널 스페이스 엑세스
703
엑세스 JVM 프로세스 맵
705
디버깅 라이브러리 감지
JNI 환경
702
Cloakware 기술에 의하여 보호되는 C/C++ 기반
700
JVM 시작
디버거가 모니터 에이전트를 부착
707
쓰레드시작 리스너를 활성화
709
새로운 쓰레드 시작 감지
711
JDPA 관련 쓰레드를 감지
JVMTI 환경
710
액션 (JVM 비활성화)
726
디버깅 절차 모니터 에이전트
713
라인 브레이크 리스너 활성화
715
디버깅 활동을 감지
717
쓰레드 및 방법 정보 보고
JVMTI 환경
718

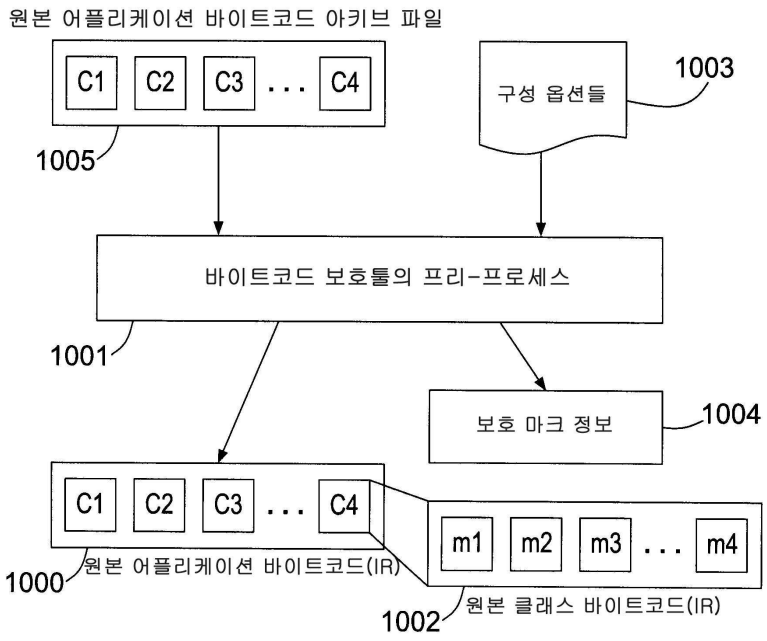
도면8


빌드-타임 프로세스
암호 정보 및 키들
502
WB정적 핸들러
508
WB
키 데이터
런-타임 데이터 및 보호된 바이트코드 데이터 파일
803
JVM
410
JNI SM 브리지
418
동적 보안 핸들러 (외부 WB 암호 라이브러리)
614
자바 바이트코드 보호 보안 보율
404
런-타임 프로세스

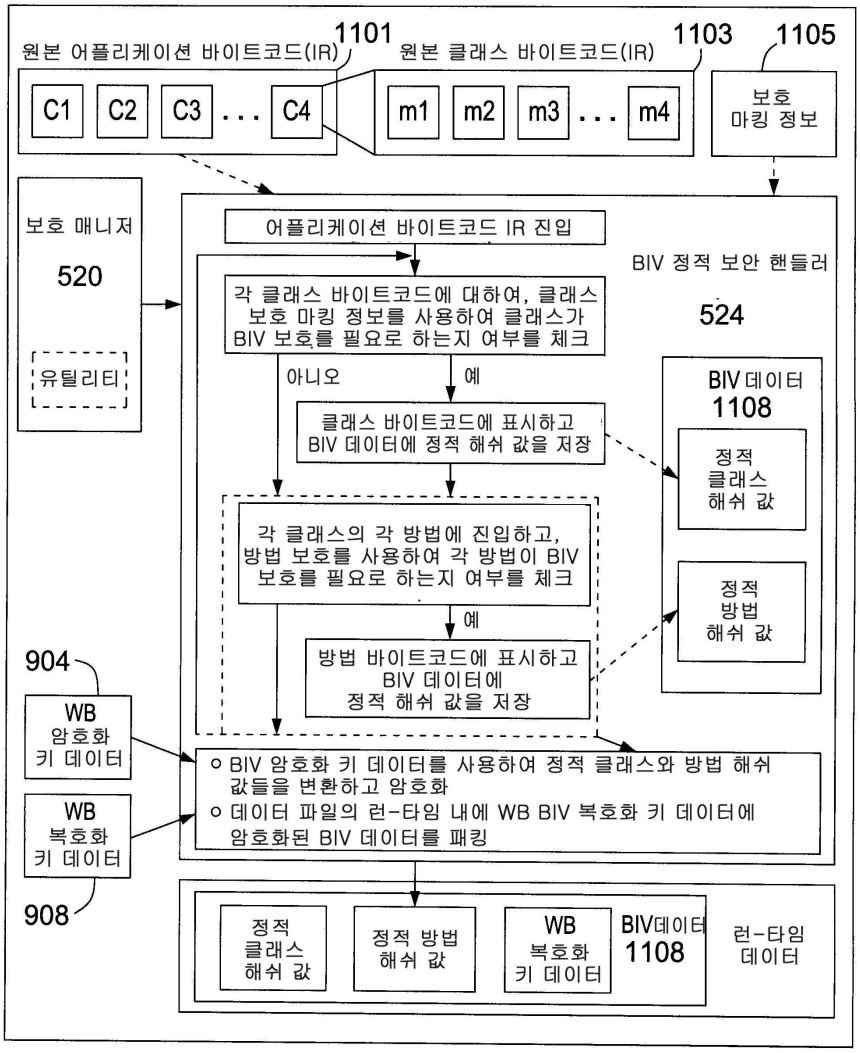
도면9


자바 바이트코드 보호 툴 916
빌드-타임 프로세스
암호 정보 및 키들
502
WB 정적 핸들러
508
다른 정적 보안 핸들러들
906
WB 암호화 키 데이터
904
WB 복호화 키 데이터
908
런-타임 데이터 및 보호된 바이트코드 데이터 파일들
JVM
410
JNI SM 브리지
418
동적 보안 핸들러들
611
WB SM 유틸리티
630
914
런-타임 프로세스

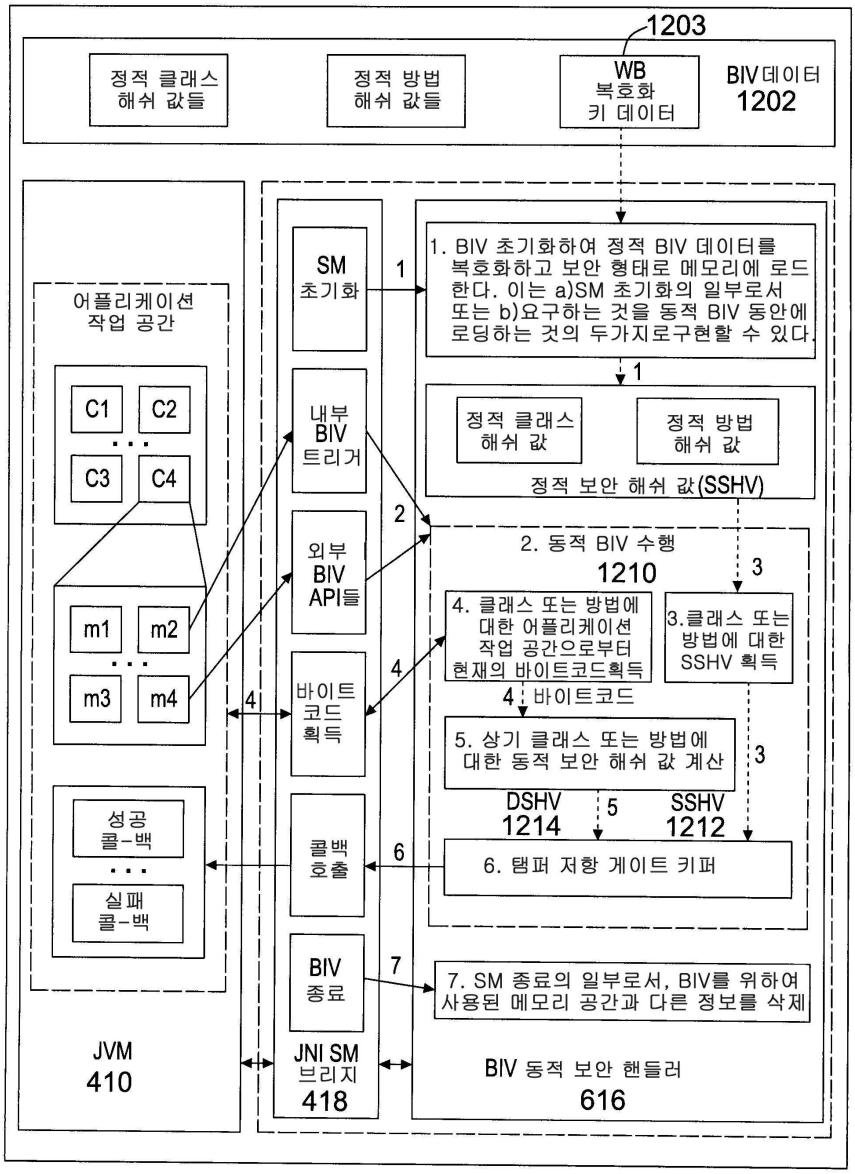
도면10


원본 어플리케이션 바이트코드 아키브 파일
C1
C2
C3
C4
1005
구성 옵션들
1003
바이트코드 보호툴의 프리-프로세스
1001
보호 마크 정보
1004
m1
m2
m3
m4
1000
원본 어플리케이션 바이트코드(IR)
1002
원본 클래스 바이트코드(IR)

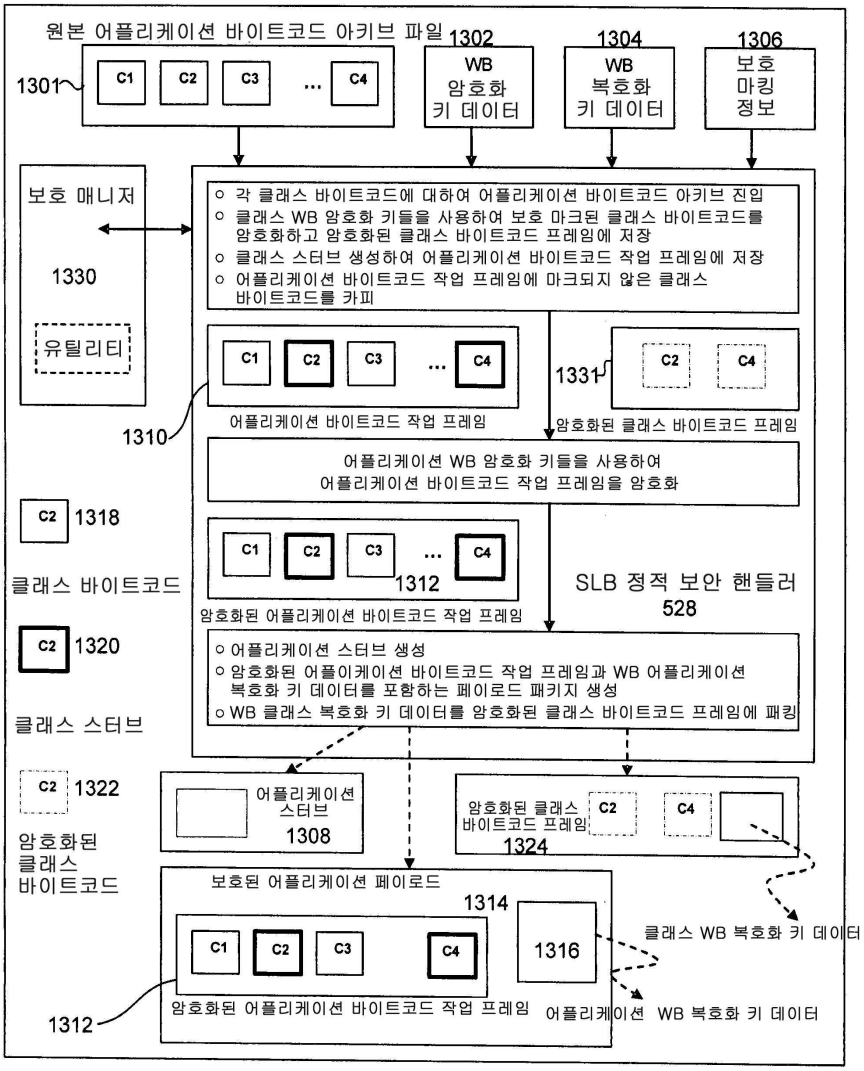
도면11


원본 어플리케이션 바이트코드(IR)
1101
원본 클래스 바이트코드(IR)
1103
1105
C1
C2
C3
C4
m1
m2
m3
m4
보호 마킹 정보
보호 매니저
520
유틸리티
어플리케이션 바이트코드 IR 진입
BIV 정적 보안 핸들러
524
각 클래스 바이트코드에 대하여, 클래스 보호 마킹 정보를 사용하여 클래스가 BIV 보호를 필요로 하는지 여부를 체크
아니오
예
BIV 데이터
1108
클래스 바이트코드에 표시하고 BIV 데이터에 정적 해쉬 값을 저장
정적 클래스 해쉬 값
각 클래스의 각 방법에 진입하고, 방법 보호를 사용하여 각 방법이 BIV 보호를 필요로 하는지 여부를 체크
예
정적 방법 해쉬 값
904
방법 바이트코드에 표시하고 BIV 데이터에 정적 해쉬 값을 저장
WB 암호화 키 데이터
WB 복호화 키 데이터
908
ㅇ BIV 암호화 키 데이터를 사용하여 정적 클래스와 방법 해쉬 값들을 변환하고 암호화
ㅇ 데이터 파일의 런-타임 내에 WB BIV 복호화 키 데이터에 암호화된 BIV 데이터를 패킹
정적 클래스 해쉬 값
정적 방법 해쉬 값
WB 복호화 키 데이터
BIV데이터
1108
런-타임 데이터

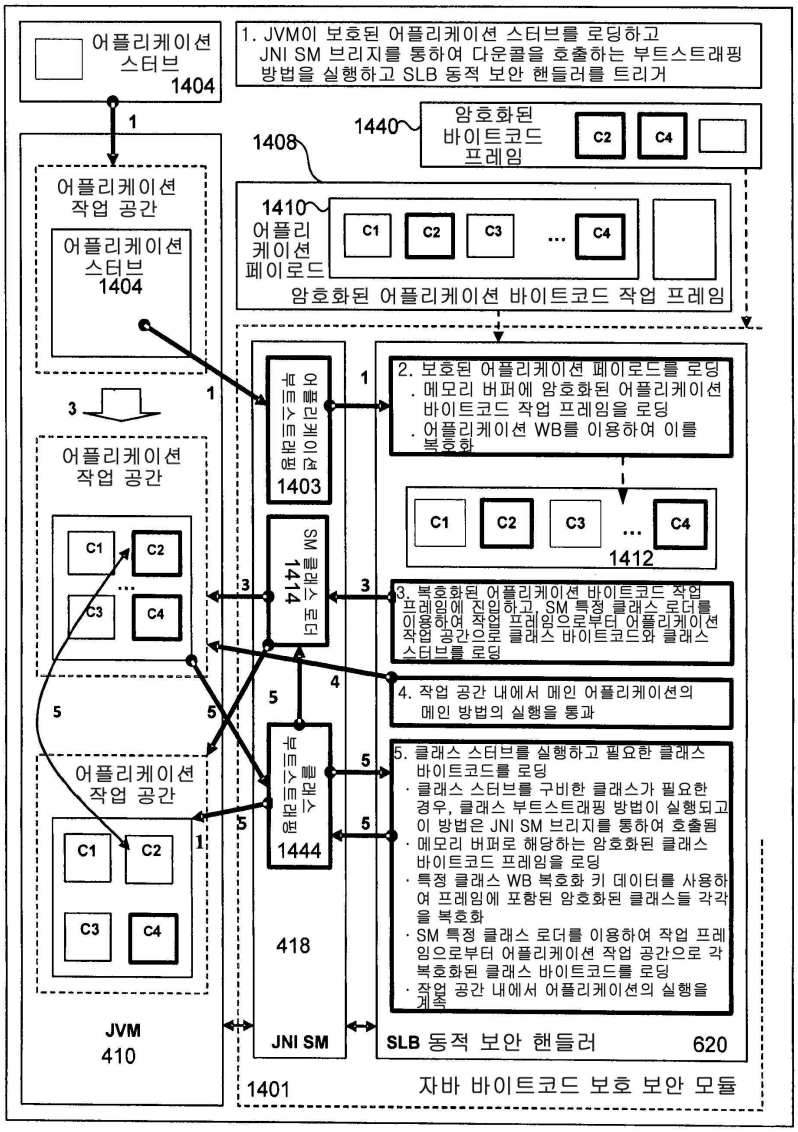
도면12


1203
정적 클래스 해쉬 값들
정적 방법 해쉬 값들
WB 복호화 키 데이터
BIV데이터 1202
어플리케이션 작업 공간
C1
C2
C3
C4
m1
m2
m3
m4
성공 콜-백
실패 콜-백
JVM 410
SM 초기화
내부 BIV 트리거
외부 BIV API들
바이트 코드 획득
콜백 호출
BIV 종료
JNI SM 브리지 418
1. BIV 초기화하여 정적 BIV 데이터를 복호화하고 보안 형태로 메모리에 로드한다. 이는 a)SM 초기화의 일부로서 또는 b)요구하는 것을 동적 BIV 동안에 로딩하는 것의 두가지로구현할 수 있다.
정적 클래스 해쉬 값
정적 방법 해쉬 값
정적 보안 해쉬 값(SSHV)
2. 동적 BIV 수행
1210
4. 클래스 또는 방법에 대한 어플리케이션 작업 공간으로부터 현재의 바이트코드획득
3.클래스 또는 방법에 대한 SSHV 획득
바이트코드
5. 상기 클래스 또는 방법에 대한 동적 보안 해쉬 값 계산
DSHV 1214
SSHV 1212
6. 탬퍼 저항 게이트 키퍼
7. SM 종료의 일부로서, BIV를 위하여 사용된 메모리 공간과 다른 정보를 삭제
BIV 동적 보안 핸들러 616

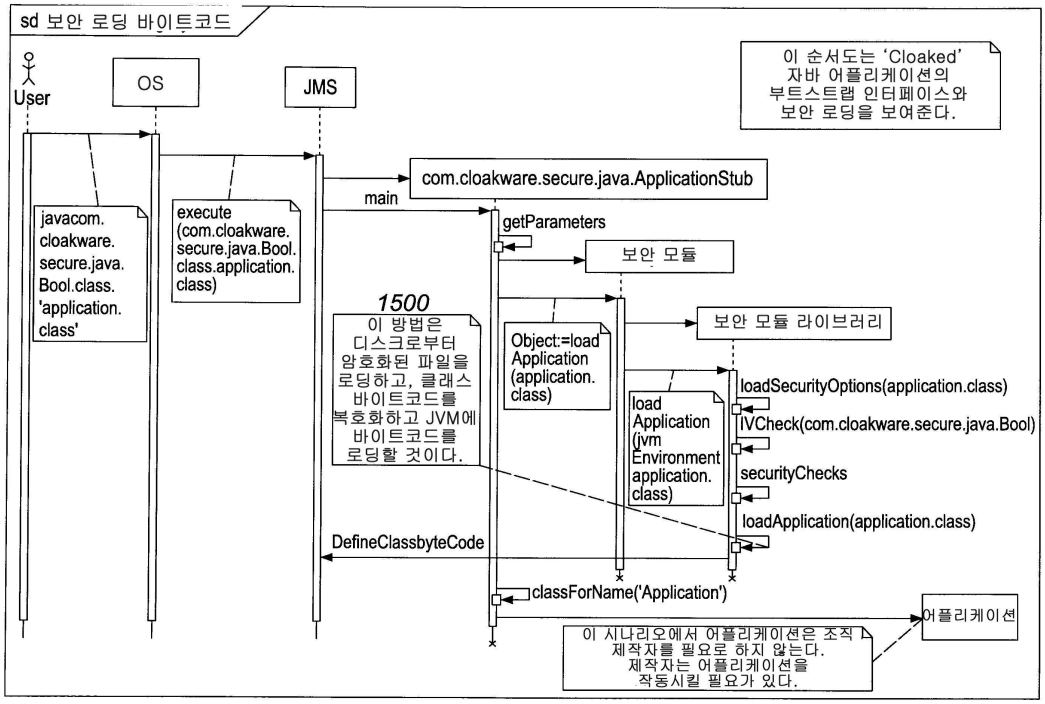
도면13


원본 어플리케이션 바이트코드 아키브 파일
1301
C1
C2
C3
...
C4
1302
WB
암호화
키 데이터
1304
WB
복호화
키 데이터
1306
보호
마킹
정보
보호 매니저
1330
유틸리티
각 클래스 바이트코드에 대하여 어플리케이션 바이트코드 아키브 진입
클래스 WB 암호화 키들을 사용하여 보호 마크된 클래스 바이트코드를 암호화하고 암호화된 클래스 바이트코드 프레임에 저장
클래스 스터브 생성하여 어플리케이션 바이트코드 작업 프레임에 저장
어플리케이션 바이트코드 작업 프레임에 마크되지 않은 클래스 바이트코드를 카피
1310
어플리케이션 바이트코드 작업 프레임
1331
암호화된 클래스 바이트코드 프레임
어플리케이션 WB 암호화 키들을 사용하여
어플리케이션 바이트코드 작업 프레임을 암호화
C2
1318
클래스 바이트코드
1312
암호화된 어플리케이션 바이트코드 작업 프레임
SLB 정적 보안 핸들러
528
C2
1320
클래스 스터브
어플리케이션 스터브 생성
암호화된 어플이케이션 바이트코드 작업 프레임과 WB 어플리케이션 복호화 키 데이터를 포함하는 페이로드 패키지 생성
WB 클래스 복호화 키 데이터를 암호화된 클래스 바이트코드 프레임에 패킹
C2
1322
암호화된
클래스
바이트코드
어플리케이션
스터브
1308
암호화된 클래스
바이트코드 프레임
1324
보호된 어플리케이션 페이로드
1314
1316
클래스 WB 복호화 키 데이터
어플리케이션 WB 복호화 키 데이터
1312
암호화된 어플리케이션 바이트코드 작업 프레임

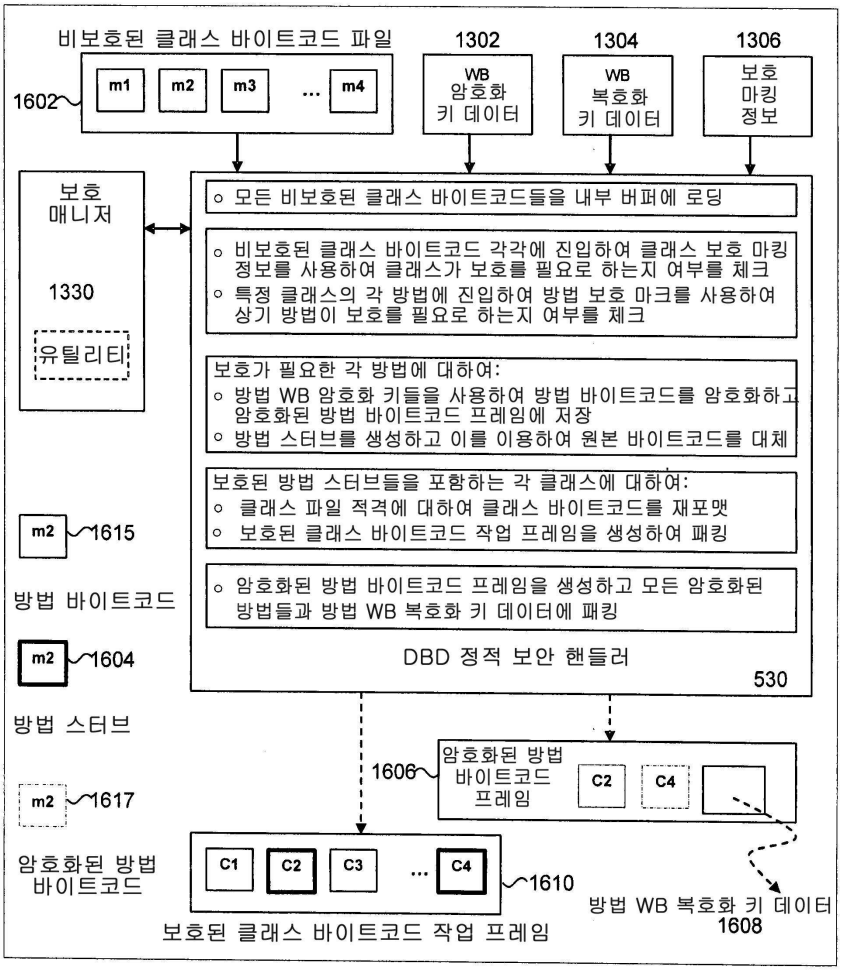
도면14


어플리케이션 스터브 1404
1. JVM이 보호된 어플리케이션 스터브를 로딩하고 JNI SM 브리지를 통하여 다운콜을 호출하는 부트스트래핑 방법을 실행하고 SLB 동적 보안 핸들러를 트리거
1440
암호화된 바이트코드 프레임
C2
C4
1408
1410
어플리케이션 페이로드
C1
C2
C3
...
C4
암호화된 어플리케이션 바이트코드 작업 프레임
어플리케이션 작업 공간
어플리케이션 스터브 1404
어플리케이션 부트스트래퍼 1403
SM 클래스 로더 1414
클래스 부트스트래퍼 1444
2. 보호된 어플리케이션 페이로드를 로딩
. 메모리 버퍼에 암호화된 어플리케이션 바이트코드 작업 프레임을 로딩
. 어플리케이션 WB를 이용하여 이를 복호화
1412
3. 복호화된 어플리케이션 바이트코드 작업 프레임에 진입하고, SM 특정 클래스 로더를 이용하여 작업프레임으로부터 어플리케이션 작업공간으로 클래스 바이트코드화 클래스 스터브를 로딩
4. 작업 공간 내에서 메인 어플리케이션의 메인 방법의 실행을 통과
5. 클래스 스터브를 실행하고 필요한 클래스 바이트코드를 로딩
· 클래스 스터브를 구비한 클래스가 필요한 경우, 클래스 부트스트래핑 방법이 실행되고 이 방법은 JNI SM 브리지를 통하여 호출됨
· 메모리 버퍼로 해당하는 암호화된 클래스 바이트코드 프레임을 로딩
· 특정 클래스 WB 복호화 키 데이터를 사용하여 프레임에 포함된 암호화된 클래스들 각각을 복호화
· SM 특정 클래스 로더를 이용하여 작업 프레임으로부터 어플리케이션 작업 공간으로 각 복호화된 클래스 바이트코드를 로딩
· 작업 공간 내에서 어플리케이션의 실행을 계속
418
JVM 410
JNI SM
SLB 동적 보안 핸들러
620
1401
자바 바이트코드 보호 보안 모듈

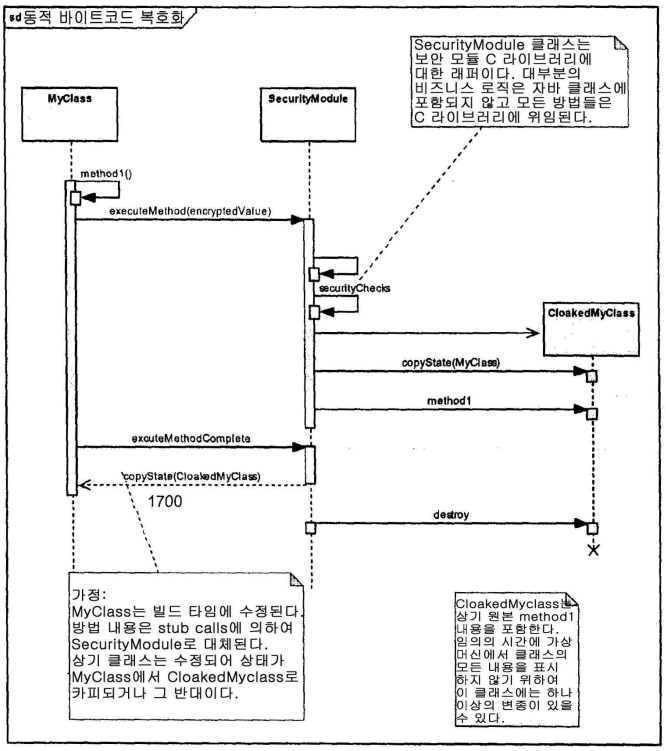
도면15


sd 보안 로딩 바이트코드
User
OS
JMS
이 순서도는 'Cloaked' 자바 어플리케이션의 부트스트랩 인터페이스와 보안 로딩을 보여준다.
com.cloakware.secure.java.ApplicationStub
main
getParameters
보안 모듈
javacom. cloakware. secure.java. Bool.class. 'application. class'
execute (com.cloakware. secure.java.Bool. class.application. class)
1500
이 방법은 디스크로부터 암호화된 파일을 로딩하고, 클래스 바이트코드를 복호화하고 JVM에 바이트코드를 로딩할 것이다.
Object:=load Application (application. class)
보안 모듈 라이브러리
load Application (jvm Environment application. class)
loadSecurityOptions(application.class)
IVCheck(com.cloakware.secure.java.Bool)
securityChecks
loadApplication(application.class)
DefineClassbyteCode
classForName('Application')
어플리케이션
이 시나리오에서 어플리케이션은 조직 제작자를 필요로 하지 않는다. 제작자는 어플리케이션을 작동시킬 필요가 있다.

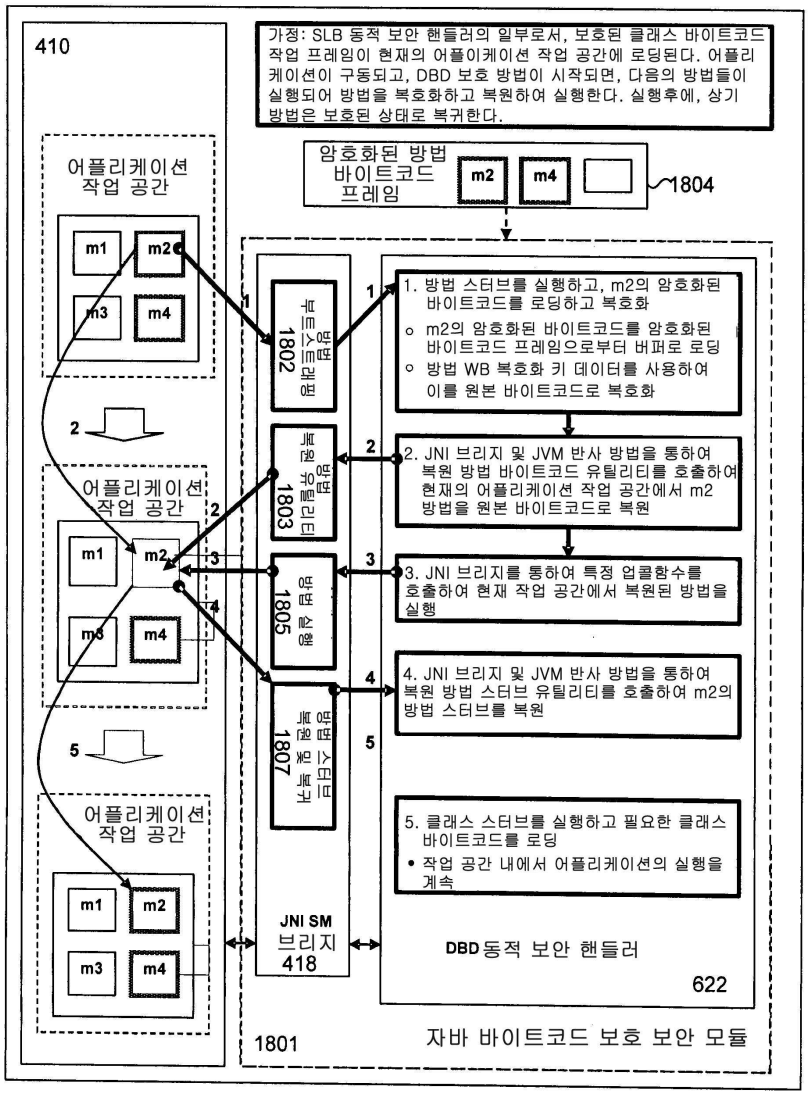
도면16


비보호된 클래스 바이트코드 파일
1602
m1
m2
m3
...
m4
1302
WB 암호화 키 데이터
1304
WB 복호화 키 데이터
1306
보호 마킹 정보
보호 매니저
1330
유틸리티
○ 모든 비보호된 클래스 바이트코드들을 내부 버퍼에 로딩
○ 비보호된 클래스 바이트코드 각각에 진입하여 클래스 보호 마킹 정보를 사용하여 클래스가 보호를 필요로 하는지 여부를 체크
○ 특정 클래스의 각 방법에 진입하여 방법 보호 마크를 사용하여 상기 방법이 보호를 필요로 하는지 여부를 체크
보호가 필요한 각 방법에 대하여:
○ 방법 WB 암호화 키들을 사용하여 방법 바이트코드를 암호화하고 암호화된 방법 바이트코드 프레임에 저장
○ 방법 스터브를 생성하고 이를 이용하여 원본 바이트코드를 대체
보호된 방법 스터브들을 포함하는 각 클래스에 대하여:
○ 클래스 파일 적격에 대하여 클래스 바이트코드를 재포맷
○ 보호된 클래스 바이트코드 작업 프레임을 생성하여 패킹
○ 암호화된 방법 바이트코드 프레임을 생성하고 모든 암호화된 방법들과 방법 WB 복호화 키 데이터에 패킹
DBD 정적 보안 핸들러
530
m2
1615
방법 바이트코드
m2
1604
방법 스터브
m2
1617
암호화된 방법 바이트코드
1606
암호화된 방법 바이트코드 프레임
C2
C4
C1
C2
C3
...
C4
1610
보호된 클래스 바이트코드 작업 프레임
방법 WB 복호화 키 데이터
1608

도면17


sd동적 바이트코드 복호화
SecurityModule 클래스는 보안 모듈 C 라이브러리에 대한 래퍼이다. 대부분의 비즈니스 로직은 자바 클래스에 포함되지 않고 모든 방법들은 C 라이브러리에 위임된다.
MyClass
SecurityModule
method1()
executeMethod(encryptedValue)
securityChecks
CloakedMyClass
copyState(MyClass)
method1
excuteMethodComplete
copyState(CloakedMyClass)
1700
destroy
가정:
MyClass는 빌드 타임에 수정된다.
방법 내용은 stub calls에 의하여 SecurityModule로 대체된다.
상기 클래스는 수정되어 상태가 MyClass에서 CloakedMyclass로 카피되거나 그 반대이다.
CloakedMyclass는 상기 원본 method1 내용을 포함한다. 임의의 시간에 가상 머신에서 클래스의 모든 내용을 표시하지 않기 위하여 이 클래스에는 하나 이상의 변종이 있을 수 있다.

도면18


410
가정: SLB 동적 보안 핸들러의 일부로서, 보호된 클래스 바이트코드 작업 프레임이 현재의 어플이케이션 작업 공간에 로딩된다. 어플리케이션이 구동되고, DBD 보호 방법이 시작되면, 다음의 방법들이 실행되어 방법을 복호화하고 복원하여 실행한다. 실행후에, 상기 방법은 보호된 상태로 복귀한다.
암호화된 방법 바이트코드 프레임
m2
m4
1804
어플리케이션 작업 공간
m1
m2
m3
m4
뉴트스크래퍼 1802
방법 복원 유틸리티 1803
방법 실행 1805
방법 스터브 복원 1807
JNI SM 브리지 418
1. 방법 스터브를 실행하고, m2의 암호화된 바이트코드를 로딩하고 복호화
m2의 암호화된 바이트코드를 암호화된 바이트코드 프레임으로부터 버퍼로 로딩
방법 WB 복호화 키 데이터를 사용하여 이를 원본 바이트코드로 복호화
2. JNI 브리지 및 JVM 반사 방법을 통하여 복원 방법 바이트코드 유틸리티를 호출하여 현재의 어플리케이션 작업 공간에서 m2 방법을 원본 바이트코드로 복원
3. JNI 브리지를 통하여 특정 업콜함수를 호출하여 현재 작업 공간에서 복원된 방법을 실행
4. JNI 브리지 및 JVM 반사 방법을 통하여 복원 방법 스터브 유틸리티를 호출하여 m2의 방법 스터브를 복원
5. 클래스 스터브를 실행하고 필요한 클래스 바이트코드를 로딩
작업 공간 내에서 어플리케이션의 실행을 계속
DBD동적 보안 핸들러
622
1801
자바 바이트코드 보호 보안 모듈