기능 및 개발자 명세서

Cold Start and Hot Passion

eBPF 제로카피를 이용한 서버리스 콜드스타트 문제 완화

F4 | 김준엽, 김선희, 오진우, 이예은

Index

1. 기획

1.1 개요

1.1.1 서비스명

Cold Start and Hot Passion

1.1.2 목표

서버리스 환경의 Cold Start 문제를 eBPF가 얼마나 개선하는가를 확인한다.

1.1.3 메인 구현 기능

- 서버리스 환경 구축
- 함수에서 system call (sysexcec()) 될 때마다 로그 찍어주는 함수 구현
- eBPF 매핑 함수 구현 (⇒ eBPF 매핑 후 위의 출력 함수 활용해 성능 비교)

1.1.4 Role & Responsibility

멤버	직책	임무
김선희	Kernel Engineer	eBPF, kernel 핸들링
김준엽	DevOps Engineer	CI/CD 테스트 구축
오진우	Infra Engineer	인프라 세팅
이예은	Performance Engineer	유저 공간 실험 로직

1.2 SWOT 분석

1.2.1 Strengths

- eBPF 기반이라 커널 수준의 높은 가시성 확보 가능
- MiniStack 기반이라 저비용 로컬 재현 가능
- 서버리스 Cold Start라는 명확한 문제 정의
- 기존 발표 자료에서 목표·기능·파이프라인이 이미 구조화되어 있음

1.2.2 Weaknesses

- MiniStack은 실제 AWS Lambda와 완전히 동일한 운영 환경은 아님
- eBPF는 러닝커브가 높아 초기 진입 장벽이 있음
- 초기 버전은 로컬 테스트 환경 중심이라 시장 범위가 좁음
- 실제 가치 증명은 정량적 성능 개선 데이터가 나와야 강해짐

1.2.3 Opportunities

- cloud native / Kubernetes adoption이 매우 높음
- 서버리스는 자동 확장과 운영 단순화 때문에 꾸준히 도입 가치가 있음
- 운영자들이 실제로 cold start와 시스템 레벨 메트릭을 관측하고 있음
- Cilium/Hubble, LoxiLB처럼 eBPF의 성공 사례가 있어 “eBPF 기반 인프라 도구” 자체에 대한 시장 신뢰가 존재함

1.2.4 Threats

- AWS CloudWatch Lambda Insights 같은 클라우드 제공자 기본 도구와 경쟁해야 함
- Cilium/Hubble 등 강력한 eBPF 생태계 프로젝트와 비교될 수 있음
- 서버리스 플랫폼 자체의 개선으로 Cold Start 문제가 점차 완화될 가능성
- 사용자 입장에서 eBPF 기반 툴은 설치/권한/운영 부담이 있을 수 있음

1.3 기능 명세서

1.3.1 서버리스 환경 구축

- 활용 오픈소스: MiniStack

Classification	Domain Function	Detailed Function	Method	API Endpoint	User API	System Call	Behavior
서버리스 환경 구축	인프라 시뮬레이션	MiniStack 기반 Lambda 컨테이너 실행 환경 구축	-	http://localhost:(port)	aws_local_lambda_create()	-	로컬 환경에서 AWS Lambda와 동일한 생명주기(Init-Invoke-Shutdown)를 구현하여 테스트 준비
	Cold Start 유도	Lambda 함수 강제 재시작 및 첫 호출 시나리오 생성	POST	/env/functions/{name}/invocations	aws_local_lambda_invoke	-	컨테이너가 없는 상태에서 함수를 호출하여 'Cold Start'가 발생하는 시점을 식별함

1.3.2 시스템 콜 로그 출력

Classification	**Domain Function**	**Detailed Function**	**Method**	**API Endpoint**	**User API**	**System Call**	**Behavior**
시스템 콜 로그 출력	모니터링/ 프로세스 추적	함수 실행 시 sys_execve() 호출 감지 및 로그 기록	-	-	print_log()	sys_execve()	Lambda 함수 인스턴스가 생성되며 새로운 프로세스가 실행될 때마다 커널 로그에 기록
	모니터링/ 실행 시간 분석	유저 수준에서의 함수 응답 시간 기록	-	-	get_time_of_exec()	-	시스템 콜 발생 시점과 함수 완료 시점 사이의 간격을 측정하여 Cold Start 지연 시간을 계산함

1.3.3 eBPF 매핑 적용

- 활용 오픈소스: eBPF

Classification	Domain Function	Detailed Function	Method	API Endpoint	User API	System Call	Behavior
eBPF 매핑	고성능 데이터 추출	bpf2go를 이용한 eBPF 프로그램 컴파일 및 로드	GET	bpf2go/tracepoint	-	eBPF Hooking	C로 작성된 eBPF 코드를 Go로 바인딩하여 커널 이벤트에 직접 후킹(Hooking)함
	정밀 측정/타임스탬프 로깅	bpf_ktime_get_ns() 기반 나노초 단위 실행 시간 추출	GET	kprobe/tracepoint	bpf_ktime_get_ns()	sys_execve() (Hooked)	시스템 콜 호출 시점에 커널 내부에서 직접 나노초 단위의 시간을 측정하여 오차를 최소화함

1.4 SBOM

1.4.1 Primary

Component	Type	Scope	Purpose / Role	Supplier / Source	Evidence
Cold Start and Hot Passion	Application / Project	Primary	프로젝트 전체 서비스/실험 대상	Project team	서비스명과 목표가 명시됨.

1.4.2 Third Party

Component	Type	Scope	Purpose / Role	Supplier / Source	Evidence
MiniStack	Open-source software	Third-party	로컬에서 AWS Lambda 생명주기(Init-Invoke-Shutdown)를 시뮬레이션하고 Cold Start 테스트 환경을 구축	`github.com/Nahuel990/ministack`	활용 오픈소스와 Lambda 로컬 생성·배포·테스트 용도가 명시됨.
AWS Lambda emulation layer (via MiniStack)	Service capability	Third-party	Lambda 컨테이너 실행 환경 구축, 함수 호출, Cold Start 유도	Provided through MiniStack	MiniStack 기반 Lambda 실행 환경 구축, invoke 시나리오가 명시됨.
eBPF	Kernel technology / open-source ecosystem	Third-party	커널 이벤트 직접 후킹, 정밀 계측	eBPF ecosystem	활용 오픈소스로 명시되고, 성능 비교용 매핑 적용 대상임.
Cilium eBPF Go library	Open-source library	Third-party	eBPF 프로그램 로드/바인딩에 사용되는 라이브러리 계열	`github.com/cilium/ebpf`	활용 오픈소스 URL로 명시됨.
`bpf2go`	Build tool / code generation tool	Third-party	eBPF 프로그램 컴파일 및 Go 바인딩 생성	Included in Cilium eBPF toolchain	문서에서 빌드 도구로 명시됨.

1.4.3 Runtime Dependency

Component	Type	Scope	Purpose / Role	Supplier / Source	Evidence
`bpf_ktime_get_ns()`	Kernel helper API	Runtime dependency	나노초 단위 타임스탬프 획득	Linux eBPF helper	타임스탬프 출력 및 정밀 측정 수단으로 명시됨.
`kprobe` / `tracepoint`	Kernel tracing hook mechanism	Runtime dependency	eBPF 후킹 지점	Linux kernel tracing subsystem	API/Endpoint 열에 명시됨.
Linux system call `sys_execve()`	OS / kernel interface	Runtime dependency	프로세스 생성 시점 감지, Cold Start 관련 이벤트 관찰 지점	Linux kernel	로그 출력/후킹 대상 시스템 콜로 명시됨.

1.4.4 local runtime

Component	Type	Scope	Purpose / Role	Supplier / Source	Evidence
Local HTTP endpoint `http://localhost:(port)`	Interface / endpoint	Local runtime	MiniStack 기반 로컬 호출 인터페이스	Local environment	서버리스 환경 구축 항목에 명시됨.
Invocation endpoint `POST /env/functions/{name}/invocations`	Interface / endpoint	Local runtime	Cold Start 유도용 함수 호출 엔드포인트	Local environment / MiniStack integration	Cold Start 유도 항목에 명시됨.

활용 오픈소스

1. **eBPF**
   https://github.com/cilium/ebpf
   - 타임 스탬프 출력: bpf_ktime_get_ns()
   - 빌드: bpf2go
2. MiniStack

   https://github.com/Nahuel990/ministack

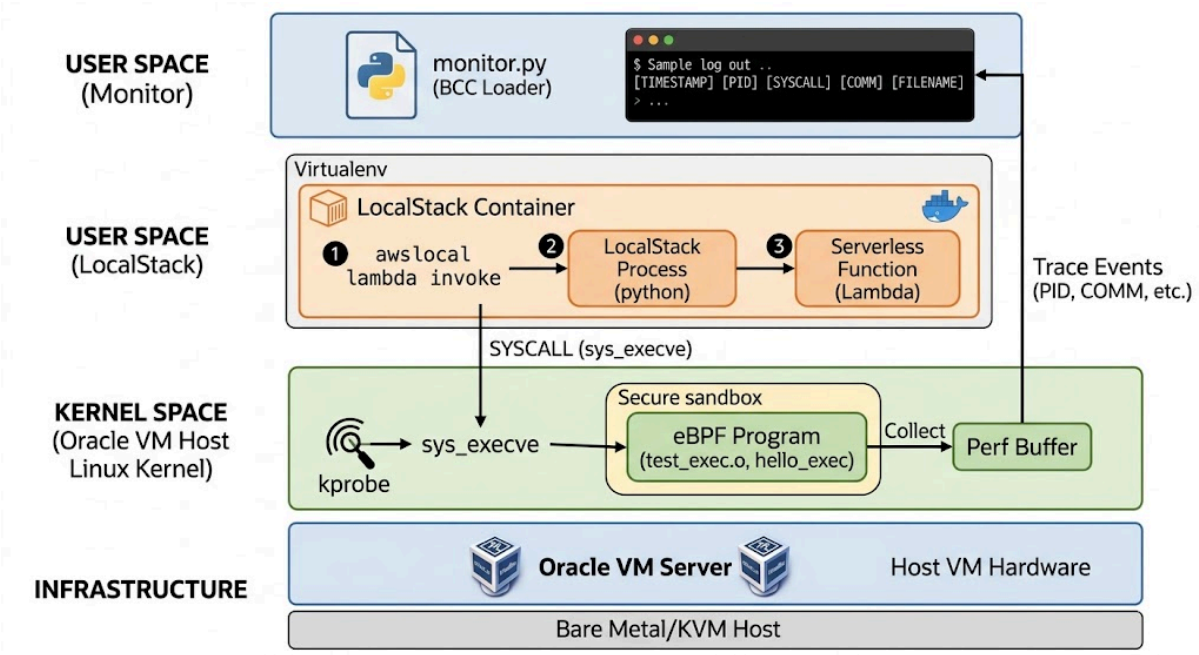
   - Lambda를 로컬에서 생성·배포·테스트할 수 있음.

의존성 트리

```
$ toy-project@1.0.0
├── ministack@latest
│   └── aws-lambda-simulation (로컬 배포 및 테스트)
│       └── cold-start-trigger (성능 측정 대상)
├── kernel-monitor-logic (시스템 콜 분석)
│   └── sys_execve-logger (프로세스 실행 추적)
│       └── timestamp-collector (기존 방식 로그 출력)
└── ebpf-optimization-suite (eBPF 매핑 적용)
    └── cilium-ebpf-library (오픈소스 프레임워크)
        └── bpf2go-compiler (Go 바인딩 및 빌드 도구)
```

2. 구현

2.1 서버리스 환경 구축



2.1.1 구현 요약

1. Ubuntu 위에서 MiniStack (: AWS 서비스를 로컬 컨테이너 안에서 에뮬레이션하는 서비스)을 이용해 AWS Lambda 같은 서버리스 환경을 로컬에서 mock
2. 구축한 lambda mocking 서버리스 환경에서, 임의의 테스트 파일을 실행시켜 system call 모킹

2.1.2 코드 구성

- 인프라: docker-compose.yaml, ministackenv/, bash.sh
- Target: index.py
- Ebpf 모니터링 엔진: v1/

```
$ ebpf-server-implement@1.0.0
├── docker-compose.yaml (Oracle VM 위에서 S3, Lambda 등의 AWS 모의 환경을 컨테이너로 띄우는 역할)
│
├── index.py (test code. 이 코드가 실행될 때 발생하는 system call을 ebpf가 hooking)
│
├── bash.sh (test code. index.py를 압축하고 lambda를 배포하는 등 테스트 루틴 반복)
│
├── ministackenv/ (ministack의 설정 파일, awslocal 실행을 위한 가상 환경이 담긴 폴더)
│
└── v1/
    ├── kprobe.c (커널 코드. 리눅스 커널에 이식되어 프로세스 정보 수집)
    │
    └── main.go (유저 코드. 커널 코드를 컴파일 및 로드하고, 수집한 데이터를 가공 및 출력)
```

3. 검증

3.1 서버 테스트 원리 및 결과

3.1.1 테스트 목적

1) 서버리스 환경 (Lamba mock)이 제대로 돌아가는지

2) 시스템 콜 호출이 제대로 hooking되는지 모니터링

우선 서버리스 환경이 제대로 돌아가는지, 그리고 시스템 콜 호출이 제대로 hooking되어 모니터링하는 과정이 원활한지 검증하기 위해서 간단한 실험을 진행하였습니다.

먼저 팀원간 원활한 협업을 위해 Oracle VM을 사용, 공용 Ubuntu 인스턴스를 구축하여 팀원 모두가 동시에 작업할 수 있는 환경을 조성하였습니다.

매우 간단한 테스트용 Python 함수를 만들었고, 이 파일을 압축해서 Lambda에 올릴 수 있는 형태로 준비했습니다. 즉, 로컬에서 만든 함수 코드를 서버리스 환경에서 배포 가능한 형태로 바꾸는 과정입니다.

이후 `awslocal` 명령어를 사용해 MiniStack 안에 Lambda 함수를 생성했고, Lambda를 직접 호출해서 테스트용 함수의 실행 결과를 확인했습니다. 이를 통해 실제로 AWS Lambda를 사용하지 않고도, 저희가 구축할 프로그램이 실제 Lambda 환경에서 어떻게 동작하는지 로컬 Ubuntu 환경에서 시험해볼 수 있습니다.

이를 통해 Ubuntu에서도 서버리스 환경을 비슷하게 구성이 가능하다는 것을 증명하였고, 시스템 콜 호출이 제대로 hooking되는지 모니터링이 가능함을 성공적으로 검증하였습니다.

3.1.2 테스트 결과

1) 서버리스 환경이 성공적으로 구동되었습니다.

```
Status: Downloaded newer image for localstack/localstack:latest
Creating ubuntu_localstack_1 ... done
```

2) 시스템 콜 (sys_execve())이 ebpf에 의해 hooking되었고, 이 정보를 user가 모니터링하도록 하는 출력 함수가 정상 실행되었습니다.

```
TIMESTAMP       PID         SYSCALL      COMM            FILENAME
03:23:33.038    16293       sys_execve   python3
03:23:33.241    16294       sys_execve   python3.12
03:23:38.551    16295       sys_execve   python3.12
03:23:38.551    16295       sys_execve   python3.12
03:23:38.551    16295       sys_execve   python3.12
03:23:38.551    16295       sys_execve   python3.12
03:23:38.551    16295       sys_execve   python3.12
03:23:38.638    16310       sys_execve   python3.12
03:23:38.638    16310       sys_execve   python3.12
03:23:38.638    16310       sys_execve   python3.12
03:23:38.638    16310       sys_execve   python3.12
03:23:38.638    16310       sys_execve   python3.12
```

[해결해야 하는 과제**]**

- 제한된 프로그래밍 환경: eBPF는 커널 안전성을 위해 복잡한 로직 구현이 어렵고 코드 크기에 제한이 있습니다. 일반 개발자가 짠 Python이나 Java 코드를 어떻게 eBPF가 이해할 수 있는 중간 단계로 변환하느냐가 승부처입니다.
- 보안 검증: 커널에서 직접 실행되므로 보안 루프홀(Loophole)이 없음을 증명해야 합니다.

---

4. CI/CD

4.1 개요

본 프로젝트의 CI/CD 파이프라인은 eBPF 기반 커널 이벤트 계측과 MiniStack 기반 서버리스 시뮬레이션 환경을 대상으로 한다.
CI 단계에서는 의존성 및 SBOM 정합성, eBPF 빌드 가능성, 단위 기능 검증, MiniStack 설정 유효성을 점검한다.
CD 단계에서는 `main` 머지 또는 `v*` 태그 생성 시점을 기준으로 최종 테스트 및 빌드를 재검증하고, SBOM을 생성·첨부하며, GitHub Release를 생성하여 공개 배포를 수행한다.

4.2 전체 파이프라인 요약표

구분	단계명	목적	실행 시점	주요 산출물
CI	의존성 및 SBOM 정적 분석	오픈소스 의존성 정합성 및 보안 취약점 점검	push, pull_request	검증 로그
CI	eBPF 컴파일 및 바인딩 테스트	eBPF C 코드와 Go 바인딩 생성 가능 여부 확인	선행 CI 통과 후	생성된 Go 바인딩 파일
CI	단위 테스트	내부 모듈 기능 정확성 검증	eBPF 빌드 테스트 후	테스트 로그
CI	MiniStack 설정 검증	서버리스 시뮬레이션 환경의 설정 유효성 확인	단위 테스트 후	설정 검증 로그
CD	최종 테스트/빌드 재검증	배포 직전 코드 상태 최종 확인	`main` 머지 또는 `v*` 태그	릴리즈 빌드 결과
CD	SBOM 생성 및 첨부	배포 산출물의 구성요소 명세 공개	CD 검증 후	SPDX/CycloneDX SBOM
CD	GitHub Release 생성	오픈소스 릴리즈 공개	태그 기준 또는 수동 배포	Release note, 첨부 파일

4.3 CI 단계 상세 표

단계 번호	Job 이름	검증 대상	수행 내용	성공 기준
1	`sbom-and-security-scan`	Third-party 라이브러리, 보안 취약점	Go 모듈 다운로드, SBOM 명세와 의존성 정합성 확인, SCA 수행	지정 라이브러리 버전 확인 및 취약점 스캔 통과
2	`ebpf-build-test`	eBPF 코드 및 Go 바인딩	LLVM/Clang 설치, go `generate ./...` 수행, 생성 파일 존재 확인	bpf2go 변환 성공, 생성 파일 확인
3	`unit-tests`	내부 모듈 기능	monitor/logging 관련 단위 테스트 수행	테스트 실패 없음
4	`Ministack-config-check`	MiniStack 환경 정의	`docker-compose.yml` 존재 확인, MiniStack 이미지 사용 여부 점검	MiniStack 설정 유효

4.4 CD 단계 상세 표

단계 번호	Job 이름	목적	수행 내용	배포 결과
1	`release-validation`	최종 배포 가능성 검증	전체 테스트 재실행, 릴리즈 빌드 수행	배포 가능한 최신 상태 확보
2	`generate-sbom`	공급망 명세 공개	프로젝트 기준 SBOM 생성	SBOM 파일 생성
3	`create-release`	공개 릴리즈	GitHub Release 생성, SBOM 첨부	버전별 공개 릴리즈

4.5 트리거 정책 표

이벤트	동작	의미
push to main	CI 실행 + CD 후보 상태 진입	메인 브랜치 반영 시 통합 검증
push to develop	CI 실행	개발 브랜치 품질 점검
pull_request to main	CI 실행	병합 전 사전 검증
push tag v*.*.*	CD 실행	공식 버전 릴리즈
workflow_dispatch	선택적 CD 실행 가능	수동 릴리즈 또는 긴급 배포

4.6 SBOM 관점 구성요소 정리표

업로드한 SBOM 설계서 내용을 기준으로 재구성하면 아래와 같습니다.

구성요소	유형	구분	역할
Cold Start and Hot Passion	Application / Project	First-party	전체 서비스 및 실험 대상
MiniStack	Open-source software	Third-party	로컬 Lambda 생명주기 시뮬레이션
AWS Lambda emulation layer	Runtime capability	Third-party	Init-Invoke-Shutdown 실행 환경 제공
aws_local_lambda_create()	Project wrapper/API	First-party	로컬 Lambda 생성
aws_local_lambda_invoke()	Project wrapper/API	First-party	함수 invoke 및 Cold Start 유도
Serverless function test workload	Application component	First-party	Cold Start 측정 대상 함수
print_log()	Software component	First-party	sys_execve 호출 로그 기록

`get_time_of_exec()`	Software component	First-party	응답 시간 측정
`sys_execve()`	OS / kernel interface	Runtime dependency	프로세스 생성 시점 감지
eBPF	Kernel technology	Third-party technology	커널 이벤트 후킹 및 정밀 계측
Cilium eBPF Go library	Open-source library	Third-party	eBPF 로드 및 바인딩
`bpf2go`	Build/codegen tool	Third-party tooling	eBPF C 코드를 Go 바인딩으로 생성
eBPF C source	Software component	First-party	커널 이벤트 후킹 로직
Go binding / loader	Software component	First-party	사용자 공간 연동
`bpf_ktime_get_ns()`	Kernel helper API	Runtime dependency	나노초 단위 타임스탬프 획득
`kprobe` / `tracepoint`	Kernel tracing hook	Runtime dependency	후킹 지점 제공
`http://localhost:(port)`	Local endpoint	Local runtime	MiniStack 호출 인터페이스
`POST /env/functions/{name}/invocations`	Invocation endpoint	Local runtime	함수 호출 엔드포인트

4.7 통합 CI/CD 설계 의도 표

설계 관점	반영 방식	기대 효과
재현성	CI에서 eBPF 빌드와 MiniStack 설정을 반복 검증	개발자 환경 차이 감소
안정성	CD 직전 전체 테스트/빌드 재검증	불완전 릴리즈 방지
공급망 투명성	SBOM 생성 및 릴리즈 첨부	사용 의존성 추적 가능
공개 배포 관리	GitHub Release 기반 버전 관리	오픈소스 배포 이력 명확화
유지보수성	CI와 CD 역할 분리	검증과 배포 책임 구분

4.8 CI/CD 설명

4.8.1 CI

본 프로젝트의 CI 파이프라인은 eBPF 기반 계측 로직과 MiniStack 기반 서버리스 테스트 환경의 신뢰성을 확보하기 위해 설계되었다. 먼저 오픈소스 의존성과 SBOM 명세 간 정합성을 점검하고, 알려진 보안 취약점을 탐지한다. 이후 bpf2go를 통해 eBPF C 코드를 Go 바인딩으로 생성할 수 있는지 검증하고, 내부 모듈에 대한 단위 테스트를 수행한다. 마지막으로 MiniStack 설정 파일을 확인하여 Lambda 시뮬레이션 환경이 정상적으로 정의되어 있는지 검사한다.

4.8.2 CD

본 프로젝트의 CD 파이프라인은 `main` 머지 또는 `v*` 태그 생성을 기준으로 동작한다. 배포 단계에서는 우선 최종 테스트와 빌드를 재검증하여 현재 코드가 릴리즈 가능한 상태인지 확인한다. 이후 프로젝트 구성요소를 문서화한 SBOM을 생성하고, 이를 GitHub Release에 첨부하여 공급망 투명성을 확보한다. 최종적으로 GitHub Release를 생성함으로써 오픈소스 릴리즈 버전을 체계적으로 공개한다.

4.9 보안 요구사항

번호	보안 요구사항	적용 근거 **(OWASP** 항목**)**	프로젝트 반영 대상
SR-01	CI/CD 파이프라인에는 승인된 브랜치와 태그만 릴리즈를 수행할 수 있도록 배포 흐름 통제를 적용해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-1: Insufficient Flow Control Mechanisms**. 추가 승인이나 리뷰 없이 악성 코드·아티팩트가 파이프라인을 따라 배포되는 위험을 다룬다. (OWASP)	`main` 머지, `v*` 태그 기반 Release
SR-02	의존성, 빌드 도구, 생성 코드에 대해 소프트웨어 공급망 검증을 수행해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-3: Dependency Chain Abuse**. 외부 패키지, 라이브러리, 플러그인, 빌드 체인 악용 위험에 대응한다. (OWASP)	`cilium/ebpf`, `bpf2go`, MiniStack, Go module
SR-03	배포 전 모든 아티팩트에 대해 SBOM 생성 및 보관을 수행해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-9: Improper Artifact Integrity Validation**와 연관된다. 아티팩트의 출처성과 구성요소 추적 가능성을 확보해야 한다. (OWASP)	Release 첨부 SBOM
SR-04	CI/CD 실행 환경에서 사용하는 토큰, 패키지 레지스트리 자격증명, GitHub 권한은 최소 권한 원칙으로 제한해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-2: Inadequate Identity and Access Management**, **CICD-SEC-6: Insufficient Credential Hygiene**. 권한 과다 및 자격증명 관리 미흡 위험에 대응한다. (OWASP)	GitHub Actions token, release 권한
SR-05	MiniStack, Docker Compose, 로컬 엔드포인트 구성은 기본값 그대로 사용하지 않고 보안 설정을 명시적으로 검증해야 한다.	**OWASP Top 10 2021 – A05: Security Misconfiguration**. 클라우드 권한 오설정, 보안 하드닝 미흡, 기본 설정 사용 위험을 포함한다. (OWASP)	`docker-compose.yml`, localhost endpoint

SR-06	로컬 테스트 환경이라도 불필요한 서비스·포트·컨테이너 기능을 비활성화해야 한다.	**OWASP Top 10 2021 – A05: Security Misconfiguration**. 불필요한 포트, 서비스, 기능 활성화를 대표적 오설정 사례로 본다. (OWASP)	MiniStack, Docker runtime
SR-07	eBPF 프로그램은 허용된 hook 지점만 사용하고, 필요 최소 이벤트만 수집해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-5: Insufficient PBAC**와 **A05: Security Misconfiguration** 관점에서 해석할 수 있다. 즉, 파이프라인·런타임 모두 최소 권한/최소 범위 원칙이 필요하다. (OWASP)	`sys_execve`, kprobe/tracepoint
SR-08	eBPF 로더 및 커널 연동 코드는 입력 검증과 로딩 실패 처리를 구현해야 한다.	**OWASP Top 10 2021 – A05: Security Misconfiguration**. 런타임 설정 오류, 검증 누락, 예외 처리 부재는 보안 결함으로 이어질 수 있다. (OWASP)	eBPF attach
SR-09	시스템 콜 로그, PID, 커맨드명, 파일명 등 관측 데이터는 민감정보 최소화 및 마스킹 정책을 적용해야 한다.	**OWASP Secrets Management Cheat Sheet**는 비밀정보와 접근정보의 중앙 관리, 감사, 노출 방지를 강조한다. 로그·텔레메트리에도 동일한 최소 노출 원칙을 적용해야 한다. (OWASP Cheat Sheet Series)	`print_log()`, trace event 출력
SR-10	보안 이벤트와 CI/CD 실행 로그는 위·변조 방지와 추적 가능성을 보장해야 한다.	**OWASP CI/CD Security Cheat Sheet**와 **OWASP Top 10 CI/CD Security Risks**는 파이프라인 전반의 감사 가능성과 추적 통제를 강조한다. (OWASP Cheat Sheet Series)	GitHub Actions logs, eBPF monitor logs
SR-11	테스트 및 릴리즈 환경은 재현 가능하고 변경 이력이 남는 설정 파일 기반으로 관리해야 한다.	**OWASP Top 10 2021 – A05: Security Misconfiguration**. 수동 설정 편차와 비관리형 환경 구성을 줄이는 것이 핵심이다. (OWASP)	`docker-compose.yml`, workflow 파일

SR-12	MiniStack 기반 mock 환경과 실제 운영 환경의 차이를 문서화하고, 운영 환경 전환 시 추가 검증 절차를 두어야 한다.	**OWASP Cloud-Native Application Security Top 10**은 클라우드 네이티브 환경의 안전한 도입을 돕기 위한 프로젝트다. 이 프로젝트 맥락에서는 mock 환경과 실제 클라우드 간 보안 차이 관리가 필요하다. 다만 현재 공개 페이지에는 세부 항목 목록이 바로 노출되지 않아, 이 요구사항은 프로젝트 취지 수준에서 연결하는 것이 안전하다. (OWASP Nest)	MiniStack ↔ 실제 AWS 이전 시
SR-13	공개 오픈소스 배포 시 Release 산출물은 무결성 검증 정보와 함께 제공해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-9: Improper Artifact Integrity Validation**. 릴리즈 아티팩트의 무결성과 출처성 검증이 핵심이다. (OWASP)	GitHub Release, SBOM 첨부
SR-14	컨테이너/시뮬레이션 환경에서 사용하는 이미지와 패키지는 취약점 스캔을 수행해야 한다.	**OWASP Top 10 2021 – A05: Security Misconfiguration** 및 **CICD-SEC-8: Ungoverned Usage of Third-Party Services**와 연결된다. 신뢰되지 않거나 관리되지 않는 외부 서비스·이미지 사용 위험을 줄여야 한다. (OWASP)	Ministack image, 빌드 환경
SR-15	CI/CD 파이프라인 실패 시 릴리즈가 자동 진행되지 않도록 Fail-closed 정책을 적용해야 한다.	**OWASP Top 10 CI/CD Security Risks – CICD-SEC-1: Insufficient Flow Control Mechanisms**. 검증 실패 후에도 배포가 진행되는 흐름은 통제 실패에 해당한다. (OWASP)	test/build/SBOM 실패 시 Release 중단

5. 일정관리
(WBS)

WBS ID	상위 항목	작업명	세부 내용	주요 산출물	상태	담당자
1	서버리스 환경 및 테스트 베드 구축	프로젝트 실험용 로컬 서버리스 환경 및 테스트 인프라 구축	전체 실험 기반 환경 구성	서버리스 테스트 환경, 인프라 설정 문서	**-**	**-**
1.1	인프라 시뮬레이션 환경 조성	**Oracle VM** 기반 **Ubuntu + Ministack** 환경 준비	**AWS Lambda** 모의 실행 환경 구성	공용 개발 환경, 인프라 구성안	**-**	**-**
1.1.1	인프라 시뮬레이션 환경 조성	**Oracle VM** 기반 공용 **Ubuntu** 인스턴스 환경 구축	공용 **Ubuntu** 인스턴스 생성, 계정·접속·기본 개발환경 세팅	**Ubuntu** 인스턴스, 접속 가이드	완료	오진우
1.1.2	인프라 시뮬레이션 환경 조성	**Docker Compose** 기반 **Ministack** 컨테이너 오케스트레이션 설계	서비스 연결 방식과 컨테이너 구조 설계	**docker-compose** 파일, 컨테이너 구성 문서	완료	김선희
1.1.3	인프라 시뮬레이션 환경 조성	**S3** 및 **Lambda** 모의 서비스 활성화 및 검증	실험용 **S3, Lambda** 기능 활성화 및 정상 동작 확인	서비스 검증 결과, 초기 테스트 로그	완료	김선희
1.2	람다 생명주기 및 호출 인터페이스 구현	함수 생성·배포·호출 및 **cold start** 재현 흐름 구성	**Lambda lifecycle** 및 **invoke** 체계 구현	함수 호출 체계, 테스트 시나리오	**-**	**-**

1.2.1	람다 생명주기 및 호출 인터페이스 구현	서버리스 생명주기**(Init–Invoke –Shutdown)** 시뮬레이션 로직 구현	함수 초기화·실행·종료 흐름을 로컬에서 재현	생명주기 시뮬레이션 로직	완료	김선희
1.2.2	람다 생명주기 및 호출 인터페이스 구현	**awslocal** 기반 **Lambda** 생성 및 배포 자동화 스크립트 작성	패키징, 생성, 배포, **invoke** 자동화	배포 자동화 스크립트	진행중	김선희
1.2.3	람다 생명주기 및 호출 인터페이스 구현	**Cold Start** 유도를 위한 강제 재시작 및 첫 호출 시나리오 설계	컨테이너 재시작**/**첫 호출 상황을 활용한 실험 시나리오 설계	**cold start** 테스트 시나리오	예정	김선희
2	시스템 콜 로깅 및 기준 성능 측정**(Baseline)**	**eBPF** 적용 전 기준 성능과 **syscall** 패턴 측정 체계 구축	**baseline** 구축 및 비교 기준 확보	**baseline** 측정 결과, 로그 데이터	**-**	**-**
2.1	프로세스 실행 추적 로직 개발	**Lambda** 실행 시 주요 시스템 콜 및 프로세스 흐름 추적	프로세스 실행 경로 추적	프로세스 추적 모듈	**-**	**-**
2.1.1	프로세스 실행 추적 로직 개발	**sys_execve()** 호출 감지 및 커널 로그 기록 함수 구현	**sys_execve** 발생 시 감지 후 커널 로그 기록	**print_log** 함수, **syscall** 로그	완료	김선희
2.1.2	프로세스 실행 추적 로직 개발	람다 인스턴스 생성 시 시스템콜 패턴 분석 및 데이터화	반복적 **syscall** 흐름 정리 및 데이터화	**syscall** 패턴 분석 결과, 정리 데이터	완료	김선희
2.2	유저 수준 응답 시간 분석기 구축	사용자 관점 함수 완료 시간 및 응답 반환 시간 측정	응답 시간 분석기 구현	응답 시간 분석기	**-**	**-**

2.2.1	유저 수준 응답 시간 분석기 구축	유저 수준 함수 완료 시점 기록 기능 구현	함수 실행 완료 시점 기록	**get_time_of_exec** 함수	완료	김선희
2.2.2	유저 수준 응답 시간 분석기 구축	시스템콜 발생 시점과 함수 완료 시점 간 지연 계산 로직 설계	**syscall** 기준 시각과 응답 시각 비교	지연 시간 계산 로직, **baseline** 비교 지표	완료	김선희
3	**eBPF** 기반 고성능 매핑 및 정밀 측정 구현	커널 후킹 및 정밀 타임스탬프 계측 기반 측정 체계 구현	성능 비교용 **eBPF** 측정 구조 구현	**eBPF** 측정 엔진	**-**	**-**
3.1	**eBPF** 커널 프로그램 개발	**kprobe/tracepoint** 기반 후킹 및 커널 시간 계측 구현	커널 레벨 계측 기능 개발	**eBPF** 커널 프로그램	**-**	**-**
3.1.1	**eBPF** 커널 프로그램 개발	**kprobe/tracepoint** 기반 **sys_execve()** 직접 후킹 로직 작성	**sys_execve** 이벤트 직접 후킹 및 프로세스 정보 수집	**kprobe.c**	완료	김선희
3.1.2	**eBPF** 커널 프로그램 개발	**bpf_ktime_get_ns** 기반 나노초 단위 타임스탬프 추출 구현	**syscall** 발생 시점 정밀 측정	정밀 타임스탬프 추출 로직	완료	김선희
3.2	**Go** 기반 데이터 처리 및 바인딩	**eBPF** 프로그램 **Go** 연동 및 커널**-**유저 공간 데이터 전달 구현	**Go** 바인딩 및 로깅 체계 구축	**Go** 연동 모듈, 로깅 시스템	**-**	**-**
3.2.1	**Go** 기반 데이터 처리 및 바인딩	**bpf2go**를 이용한 **eBPF** 프로그램 컴파일 및 **Go** 로드 기능 구현	**eBPF** 코드의 **Go** 빌드**/**로드 구조 구현	**bpf2go** 빌드 파이프라인**, Go** 로더	예정	이예은
3.2.2	**Go** 기반 데이터 처리 및 바인딩	**Perf Buffer** 또는 **Ring Buffer** 기반 커널**-**유저 공간 데이터 매핑 구축	이벤트 전달용 버퍼 구조 구현	**Perf/Ring Buffer** 매핑 코드	진행 중	김선희, 이예은

3.2.3	**Go** 기반 데이터 처리 및 바인딩	추출된 나노초 데이터를 가공하는 정밀 로깅 시스템 완성	이벤트를 시간순으로 정리해 로그 형태로 가공	정밀 로깅 시스템, 이벤트 로그	진행중	김선희, 이예은
4	통합 검증 및 성능 분석	전체 **PoC** 통합 실행 및 **baseline** 대비 **eBPF** 효과 정량 분석	통합 검증과 성능 비교 수행	성능 분석 결과, **PoC** 보고서	-	-
4.1	**PoC** 기능 통합 및 테스트	서버리스, **syscall** 로깅, **eBPF** 매핑 기능 통합 검증	하나의 흐름으로 연계 동작 확인	통합 테스트 결과	-	-
4.1.1	**PoC** 기능 통합 및 테스트	서버리스 환경 구동 및 시스템콜 후킹 정상 작동 여부 검증	**Lambda mock** 환경에서 함수 실행과 **syscall** 후킹 확인	통합 동작 검증 로그	진행중	김선희, 이예은
4.1.2	**PoC** 기능 통합 및 테스트	다양한 언어 기반 테스트 함수의 콜드 스타트 데이터 수집	**Python** 등 다양한 함수 유형 비교 데이터 수집	실험 데이터셋, 테스트 결과 로그	진행중	김선희, 이예은
4.2	정량적 성능 개선 데이터 도출	**eBPF** 적용 전후 차이 측정 및 수치화	정량 비교 수행	정량 비교표, 분석 자료	-	-
4.2.1	정량적 성능 개선 데이터 도출	기존 방식 대비 **eBPF** 매핑 적용 시 측정 정밀도 및 오차 범위 비교	**baseline**과 **eBPF** 방식의 정밀도 비교	정밀도 비교 결과	진행중	김선희, 이예은
4.2.2	정량적 성능 개선 데이터 도출	수집 데이터 기반 콜드 스타트 병목 지점 식별 및 분석 보고서 작성	병목 구간 식별 후 보고서 정리	병목 분석 보고서	진행중	김선희, 이예은
5	**DevOps** 및 형상 관리	프로젝트 자동화, **SBOM** 관리, 보안 검증, 재현 가능한 운영 체계 구축	운영·보안·자동화 체계 정비	운영 문서, 자동화 체계	-	-

5.1	자동화 파이프라인 및 **CI/CD** 구축	빌드·테스트·배포 자동화	반복 실험 가능한 파이프라인 구성	**CI/CD** 파이프라인	**-**	**-**
5.1.1	자동화 파이프라인 및 **CI/CD** 구축	프로젝트 전체 기능 대상 **CI/CD** 테스트 환경 구축 및 운영	자동 테스트 가능한 환경 구성 및 운영	**CI/CD** 환경**,** 테스트 워크플로우	예정	김준엽**,** 오진우
5.1.2	자동화 파이프라인 및 **CI/CD** 구축	**bpf2go** 컴파일 및 배포 루틴 포함 자동화 워크플로우 구성	**eBPF** 컴파일부터 함수 배포까지 자동화	자동화 스크립트**,** 배포 워크플로우	예정	김준엽**,** 오진우
5.2	**SBOM** 및 취약점 관리	프로젝트 의존성과 보안 리스크 문서화 및 검증	**SBOM** 및 보안 검토 수행	**SBOM** 문서**,** 취약점 검토 결과	**-**	**-**
5.2.1	**SBOM** 및 취약점 관리	**Primary** 및 **Third-party** 라이브러리 **SBOM** 명세 최신화 및 관리	내부**/**외부 의존성 정리 및 최신 상태 유지	최신 **SBOM**	완료	전원
5.2.2	**SBOM** 및 취약점 관리	**eBPF** 커널 실행에 따른 보안 루프홀 검증 및 안전성 증명	권한**, verifier,** 커널 안전성 이슈 점검	보안 검토 문서**,** 안전성 검증 자료	예정	전원

WBS ID	작업명	세부 내용	상태	담당자
1.1.1	**Oracle VM** 기반 공용 **Ubuntu** 인스턴스 환경 구축	공용 **Ubuntu** 인스턴스 및 기본 개발환경 세팅	완료	오진우
1.1.2	**Docker Compose** 기반 **Ministack** 오케스트레이션 설계	컨테이너 실행 구조 및 서비스 연결 설계	완료	김선희
1.1.3	**S3** 및 **Lambda** 모의 서비스 활성화 및 검증	실험용 모의 서비스 동작 검증	완료	김선희

1.2.1	서버리스 생명주기 시뮬레이션 로직 구현	**Init–Invoke–Shutdown** 흐름 재현	완료	김선희
1.2.2	**Lambda** 생성 및 배포 자동화 스크립트 작성	**awslocal** 기반 자동 배포·호출 스크립트	진행 중	김선희
1.2.3	**Cold Start** 유도 시나리오 설계	강제 재시작 및 첫 호출 시나리오 구성	예정	김선희
2.1.1	**sys_execve()** 호출 감지 및 로그 기록	**syscall** 감지 및 커널 로그 기록	완료	김선희
2.1.2	시스템콜 패턴 분석 및 데이터화	반복 **syscall** 흐름 정리	완료	김선희
2.2.1	함수 완료 시점 기록 기능 구현	유저 수준 완료 시점 측정	완료	김선희
2.2.2	지연 시간 계산 로직 설계	**syscall** 시점과 응답 시점 간 간격 계산	완료	김선희
3.1.1	**sys_execve()** 직접 후킹 로직 작성	**kprobe/tracepoint** 기반 후킹	완료	김선희
3.1.2	나노초 단위 타임스탬프 추출 구현	**bpf_ktime_get_ns** 기반 정밀 측정	완료	김선희
3.2.1	**bpf2go** 기반 컴파일 및 **Go** 로드 기능 구현	**eBPF** 코드 **Go** 연동	예정	이예은
3.2.2	**Perf/Ring Buffer** 기반 데이터 매핑 구축	커널-유저 공간 이벤트 전달 구조 구현	진행 중	김선희, 이예은
3.2.3	정밀 로깅 시스템 완성	시간순 이벤트 가공 및 로그화	진행 중	김선희, 이예은
4.1.1	서버리스 환경 구동 및 **syscall** 후킹 검증	통합 동작 확인	진행 중	김선희, 이예은

4.1.2	다양한 테스트 함수의 **cold start** 데이터 수집	비교 실험용 데이터 확보	진행 중	김선희, 이예은
4.2.1	**eBPF** 적용 전후 정밀도 및 오차 비교	**baseline vs eBPF** 비교	진행 중	김선희, 이예은
4.2.2	병목 지점 식별 및 분석 보고서 작성	**cold start** 병목 분석	진행 중	김선희, 이예은
5.1.1	프로젝트 전체 기능 **CI/CD** 테스트 환경 구축	자동 테스트 환경 구축 및 운영	예정	김준엽, 오진우
5.1.2	**bpf2go** 포함 자동화 워크플로우 구성	컴파일~배포 자동화 루틴 구성	예정	김준엽, 오진우
5.2.1	라이브러리 **SBOM** 최신화 및 관리	내부/외부 의존성 정리	완료	전원
5.2.2	**eBPF** 실행 보안 루프홀 검증 및 안전성 증명	**verifier,** 권한, 커널 안전성 점검	예정	전원

6. 개발자 메뉴얼

Description for Dev

항목	상세 내용	관련 코드/파일 위치
Cold Start 및 네트워크 **I/O** 병목 해결		
Zero-Copy 데이터 매핑 (Lambda 핸들러)	/dev/shm의 Arrow 데이터를 mmap으로 직접 조회	src/handler.py
eBPF 커널 트레이싱	execve 시스템 콜 추적 및 링버퍼 기록	ebpf_writer.py, trace_clone.c
로컬 모킹 인프라	MiniStack을 통한 S3 및 Lambda 환경 구축	docker-compose.yml, deploy.sh
성능 대시보드 (또는 관련 UI 스크립트)	Baseline vs Zero-Copy 지연 시간 시각화	streamlit_app.py
인터페이스 구분		
S3 Event Trigger	S3에 JSON 업로드 시 Lambda 호출 (S3/Lambda 연결 설정)	deploy.sh
Lambda Invocation	Dashboard에서 동기 방식으로 Lambda 호출 (boto3 호출부)	streamlit_app.py
eBPF to Lambda IPC	커널 이벤트를 /dev/shm/lambda_ringbuf 에 기록	ebpf_writer.py, ringbuffer.py
컴포넌트/라이브러리		
Docker Engine & Compose	런타임 환경 및 오케스트레이션	docker-compose.yml
MiniStack	AWS 서비스 로컬 모킹	deploy.sh
bpfcc-tools, bcc	커널 데이터 획득 및 트레이싱	ebpf_writer.py
pyarrow, mmap, os	Arrow 포맷 처리 및 메모리 매핑	src/handler.py, ringbuffer.py
GitHub Actions	자동화 빌드 및 배포	.github/workflows/ci-pipeline.yml
시스템 요구사항 및 의존성		
도커 설정	--ipc=host, /dev/shm 마운트 필수	docker-compose.yml (LAMBDA_DOCKER_FLAGS)
실행 권한	eBPF 트레이서 실행 시 sudo 권한 필요	ebpf_writer.py (실행 가이드)
실행 순서	bake_arrow_shm.py 선행 실행 필수	deploy.sh (또는 실행 스크립트)
데이터 생명주기		
Create	Arrow IPC 데이터 생성 및 eBPF 이벤트 기록	bake_arrow_shm.py, ebpf_writer.py
Read	mmap을 이용한 메모리 주소 즉시 읽기	src/handler.py, ringbuffer.py
Update	링버퍼 Tail 포인터 갱신 및 이벤트 실시간 업데이트	ebpf_writer.py
Delete	기존 컨테이너 및 이력 정리, 메모리 매핑 해제	clear_before_start.sh, ringbuffer.py