論文紹介 “KASLR in the age of MicroVMs”

2022/05/27

立命館大学 情報理工学部
穐山 空道 (s-akym@fc.ritsumei.ac.jp)

紹介論文

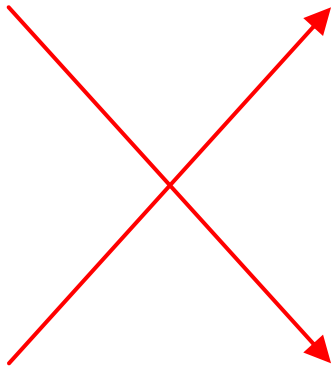
- 著者
  - Benjamin Holmes (Vassar College), Jason Waterman (Vassar College), Dan Williams (Virginia Tech)
- タイトル
  - KASLR in the age of MicroVMs
- 出展
  - Seventeenth European Conference on Computer Systems (EuroSys), 2022
- 概要
  - カーネルのアドレスランダム化を Micro VM でも使えるようにする

前提知識 1：Address Space Layout Randomization (ASLR)

- プログラムの配置される仮想アドレスをランダムにする
  - コード再利用攻撃（例：ROP）に利用されるコード片の発見を困難にする
  - バイナリを関数ごとなどに分割し、分割された単位ごとにランダム配置
    - gcc では -ffunction-sections オプションで各関数を別々のセクションにしてくれる

```
00001064 <_start>:
   1064:    31 ed          xor   %ebp,%ebp
   1066:    49 89 d1       mov   %rdx,%r9
   …
   …

00001140 <main>:
   1140:    55             push  %rbp
   1142:    48 89 e5       mov   %rsp,%rbp
   …
   …
```



```
00234568 <main>:
234568:    55             push  %rbp
234570:    48 89 e5       mov   %rsp,%rbp
   …
   …

00521FE00 <_start>:
52FE00:    31 ed          xor   %ebp,%ebp
52FE02:    49 89 d1       mov   %rdx,%r9
   …
   …
```

前提知識 2：Kernel Address Space Layout Randomization (KASLR)

- ASLR を Linux カーネルにも適用
  - 元々は全体のベースをランダム化、2020 年に関数ごとのランダム化が登場
- 具体的な手順
  - カーネルの ELF ファイルをパースして各関数（セクション）を取り出す
  - 各関数の開始位置にランダムなオフセットを足し配置
  - ランダム化した関数位置に基づき動くよう調整
    - 他の関数を呼び出すコード
    - シンボルテーブル（カーネルモジュールを動的ロードするため）
    - 例外テーブル（ユーザメモリ領域にアクセスしうるカーネル内の命令アドレスの一覧）

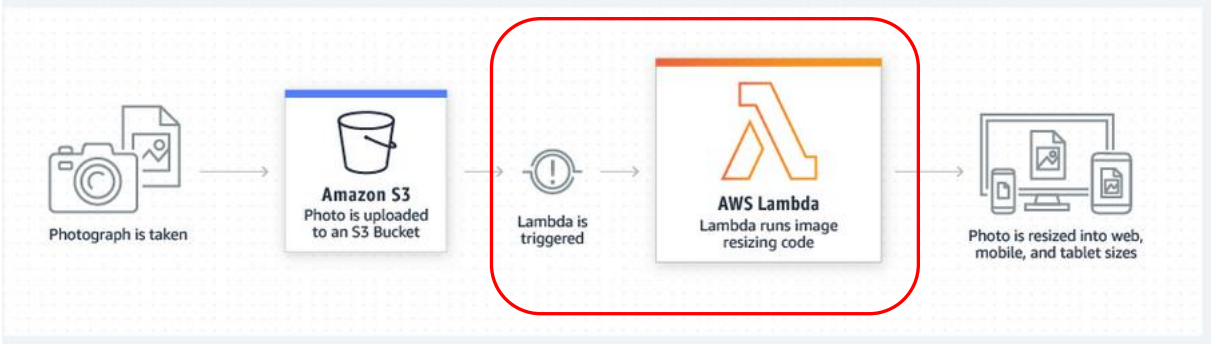
参考：
https://github.com/torvalds/linux/blob/master/Documentation/x86/exception-tables.rst
Chapter 9 “System Call Handler and Service Routines”, Understanding the Linux Kernel

クラウドの利用モデルの変化

- 従来の利用モデル
  - 普通のサーバと同じようなワークロードを実行
    - 例：データ分析を自前の GPU サーバではなくクラウド上で実行
  - 実行が数時間、数日、数年にわたり続く
- 近年のトレンド
  - 単一の関数を実行する
    - いわゆる “Server-less computing”、AWS Lambda が有名
  - 数秒や数ミリ秒で実行が終わる

→ 起動の所要時間短縮が重要

アップロードされた画像のリサイズだけを
AWS lambda で実行（一瞬で終わる）



引用元：https://aws.amazon.com/jp/lambda/

Micro VM

- 軽量で必要最低限の機能のみを持つ VMM
  - 単一の関数を実行するような利用モデルを想定
  - コンテナよりも強固な分離 かつ コンテナ並みの軽量さを目指す
    - コンテナの懸念：セキュリティ（syscall が直接呼べる）、互換性（ホストと同一カーネル）

- 例：Amazon Firecracker [1]
  - Amazon lambda の実装で使われる
  - 様々な工夫で軽量さを実現
    - デバイスはネットワーク、ブロックデバイス、シリアルポートのみ（USB はなし！）
    - ゲストの Linux の起動プロセスを簡略化（次で説明）

[1] Alexandru Agache *et al.*, “Firecracker: Lightweight Virtualization for Serverless Applications”, *NSDI’20*

Firecracker での Micro VM の起動（1/3）

- 通常の Linux kernel の起動方式
  - 圧縮したカーネルと起動用プログラム（bootstrap loader）がセット

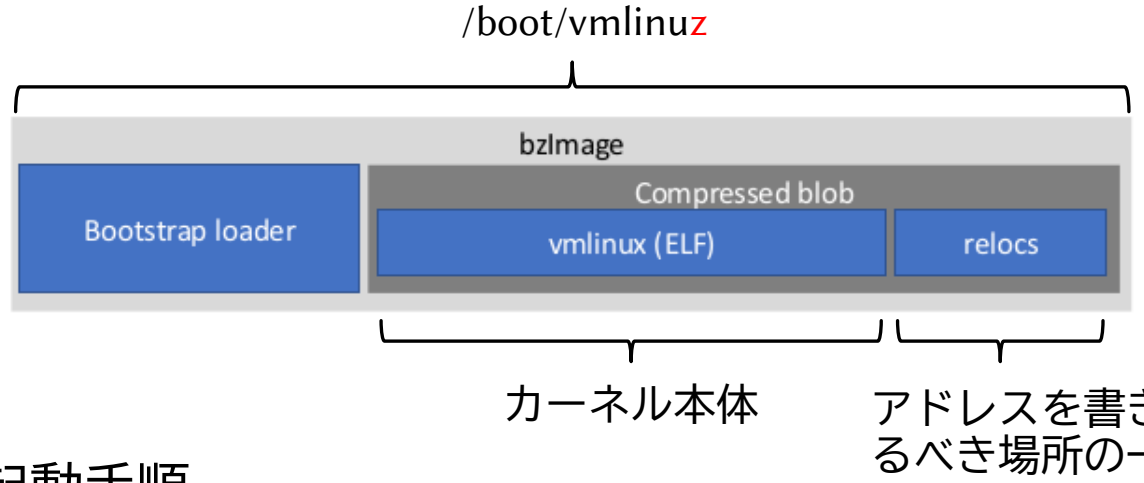


Fig. 2, “KASLR in the age of MicroVMs”

  - 起動手順
    - VMM は bootstrap loader に実行を移す
    - Bootstrap loader はカーネルを解凍し適切な場所に配置
    - 配置されたカーネルに実行を移す

Firecracker での Micro VM の起動（2/3）

- MicroVM の起動方式：Direct Kernel Boot
  - 非圧縮カーネルをそのまま起動し、bootstrap loader を使わない

- 通常方式に比べ起動が高速… なこともある
  - 条件：カーネルイメージがメモリにキャッシュされ disk から読まない
    - 次ページに具体的な実験結果
  - Serverless computing では標準のカーネルをほぼみんな使うので条件が成立
    - 私見：コンテナは互換性が問題と言っていたのにやや矛盾？（みんな同じカーネルを使ならそのカーネルの上でコンテナを立ち上げれば OK…？）

Firecracker での Micro VM の起動（3/3）

- 通常方式と Direct Kernel Boot の起動時間比較
  - -bzImage が通常方式（圧縮あり）　-vmlinux が direct kernel boot（圧縮なし）
  - lupine、aws、ubuntu はカーネルコンフィグの違い

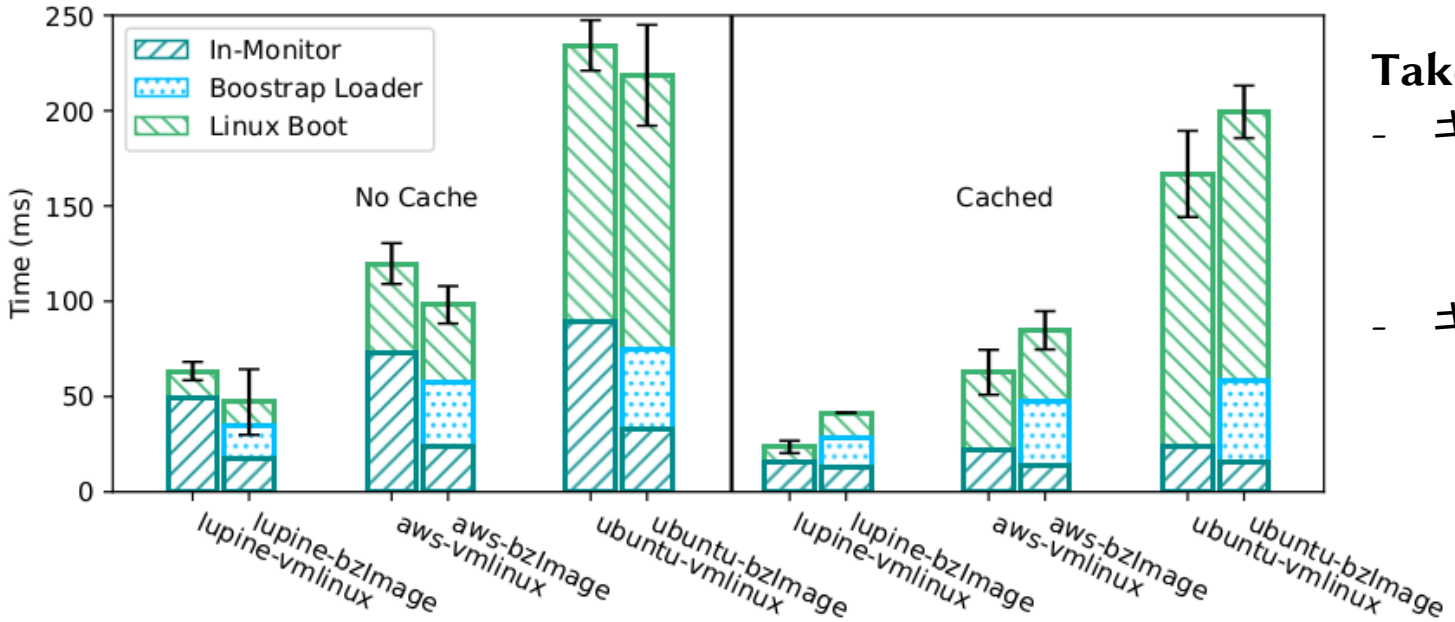
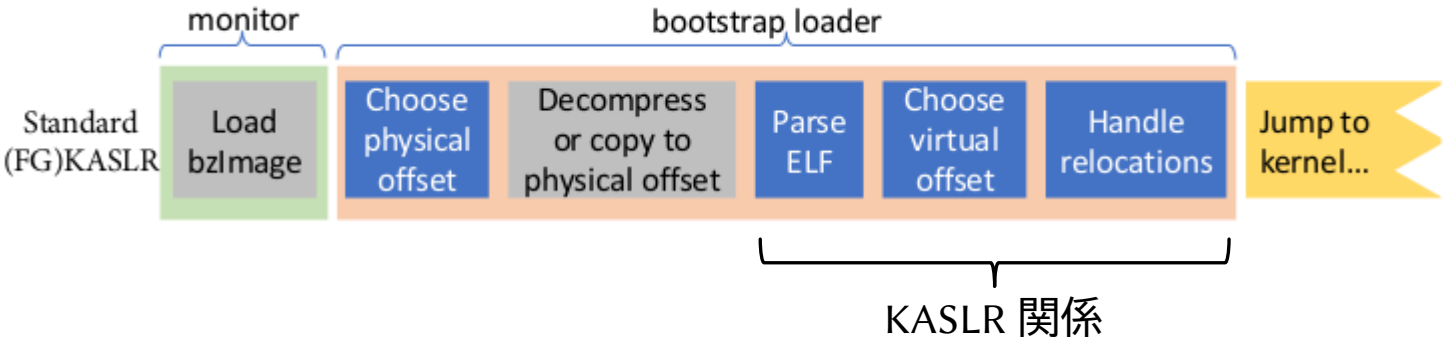


Fig. 4, “KASLR in the age of MicroVMs”

Takeaways:
- キャッシュなしでは通常方式が高速
  - ディスクからカーネルイメージを読み出すのに時間がかかっている
- キャッシュありでは direct kernel boot が高速
  - Bootstrap loader にかかる時間がなくなっている分速い

Direct kernel boot の課題と対策

- 課題：KASLR が実行されない
  - KASLR は bootstrap loader で実行される
  - 通常方式の起動時の動作の breakdown



Part of Fig. 7, “KASLR in the age of MicroVMs”

- そこで VMM 内で直接 KASLR を実行する

VMM 内での KASRL：脅威モデル

- ポイント：従来型の IaaS 的なモデルとは異なる
  - 従来：攻撃者が好きな VM を起動 → ゲストの特権は攻撃者にあり
  - 今回：攻撃者は Micro VM 上の関数を起動 → ゲストの特権は信頼できる管理者にあり

- 想定
  - Code Injection は不可能
    - ページの書き込みと実行権限の排他設定による
  - ユーザコードの特権モードでの実行は不可能
    - Supervisor Mode Execution Protection による
  - OS の脆弱性を突き Control flow を変えることはできうる ← この脅威を KASLR で下げる

提案システムの設計

- ランダム化を VMM 内で行う
  - （この節はあまり大したことは書いていない）
  - 上：通常の KASLR　下：提案する In-monitor KASLR

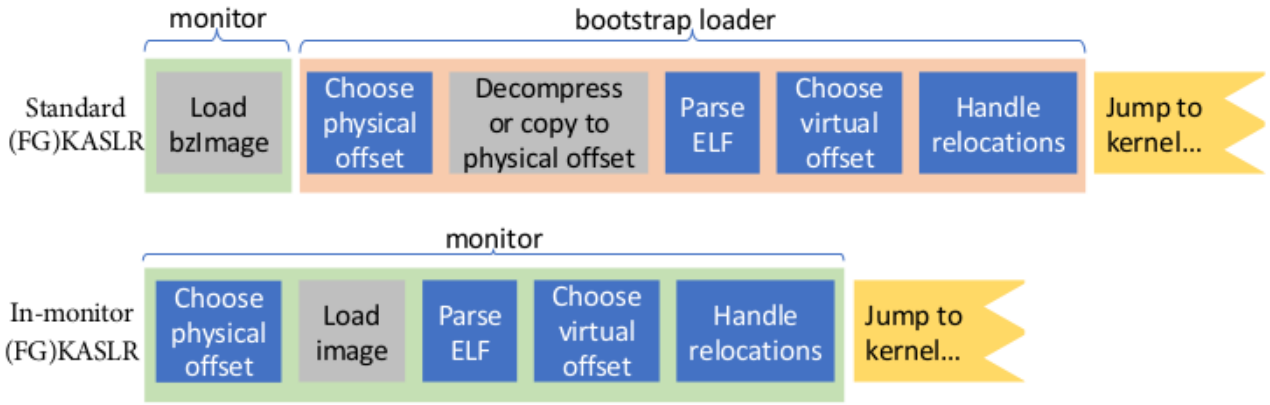


Fig. 7, “KASLR in the age of MicroVMs”

- 通常の起動方式に比べイメージのコピーが一回減っている
  - 通常：bzImage を VMM がコピーし、bootstrap loader が再度 blob から取り出す

/proc/kallsyms の調整

- カーネル内の全シンボル位置を表示する proc ファイル
  - トレースツールやプロファイラが使う

```
0000000000000000 A fixed_percpu_data
0000000000000000 A __per_cpu_start
0000000000001000 A cpu_debug_store
0000000000002000 A irq_stack_backing_store
0000000000006000 A cpu_tss_rw
000000000000b000 A gdt_page
000000000000c000 A exception_stacks
0000000000010000 A entry_stack_storage
0000000000011000 A espfix_waddr
```

手元のマシン（Ubuntu 20.04 LTS, Linux 5.11.0-46-generic）の例

- 表示するアドレスの調整を起動時にせず、はじめて読まれたときまで遅延
  - この調整が起動時間の 22 % を占めた
  - MicroVM では使われないことが多いと思われる
    - “may never been examined” ← 具体的な証拠はなし

実装

- Firecracker v0.26 を改変
  - AWS のプロダクションで利用、起動時間が短い
  - 元の Rust 実装に 1000 行程度のコード追加で実現
  - 基本的に元の bootstrap loader の C コードを参考にしている

- VMM の起動時に relocs 情報を入力
  - カーネルのコンパイル時に生成される

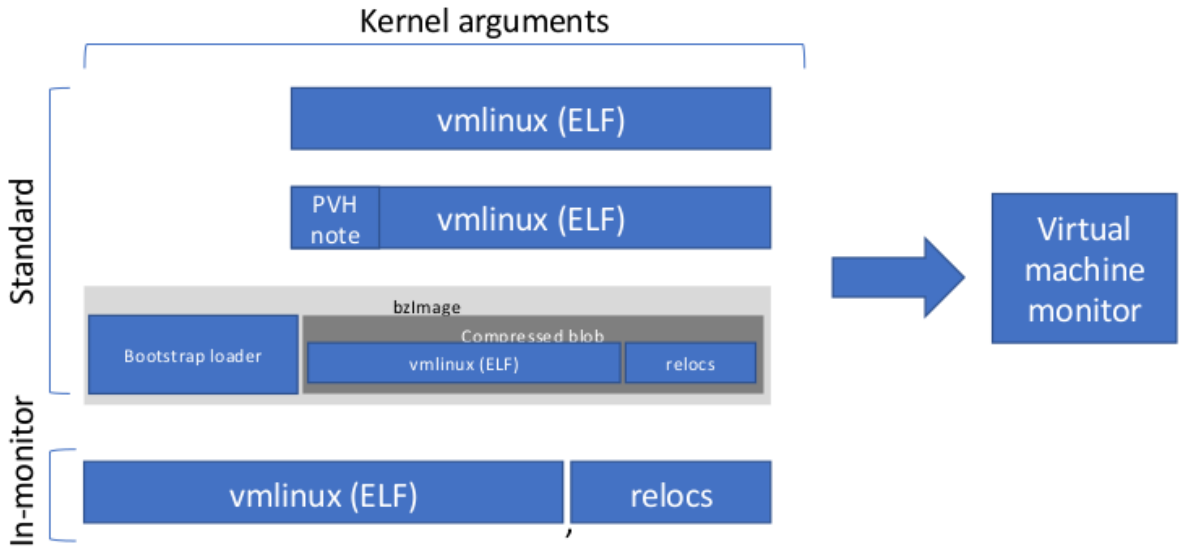


Fig. 8, “KASLR in the age of MicroVMs”

実験の評価軸

- メインの評価軸：起動時間
  - KASLR を実現しかつ高速起動したい
- 具体的な Question
  - In-monitor KASLR により目標の起動時間を達成できるか？（“achieve its boot targets”）
  - ゲストに割り当てるメモリ量による起動時間への影響は？
  - In-monitor KASLR による起動時間以外のオーバーヘッドは？

実験環境

- ホストマシン
  - Core i7-4790, 8 GB DDR3 1600, SATA SSD, Ubuntu 18.04 (Linux 4.15.0-101-generic)
- テストするゲストカーネル

kernel	vmlinux size	bzImage size (Compression None, LZ4)	relocs size	config
lupine-nokaslr	20M	22M, 4.1M	N/A	Lupine
lupine-kaslr	20M	22M, 4.2M	95K	Lupine + KASLR
lupine-fgkaslr	22M	24M, 4.7M	304K	Lupine + FGKASLR
aws-nokaslr	39M	41M, 7.9M	N/A	Firecracker
aws-kaslr	39M	41M, 8.3M	348K	Firecracker + KASLR
aws-fgkaslr	42M	45M, 9.8M	1.1M	Firecracker + FGKASLR
ubuntu-nokaslr	45M	47M, 13M	N/A	Ubuntu 18.04.5
ubuntu-kaslr	45M	48M, 14M	1M	Ubuntu + KASLR
ubuntu-fgkaslr	50M	54M, 17M	2.3M	Ubuntu + FGKASLR

-kaslr: 全体のベースのみランダム化
-fgkasrl: 関数ごとにランダム化
（関数ごとに再配置するので relocs が大きい）

- 起動時間測定方法
  - 時刻を取るポイントに目印となる syscall 呼び出しを入れ、perf でトレース

実験結果（1/3）

- Q: In-monitor KASLR により目標とする起動時間を達成できるか？

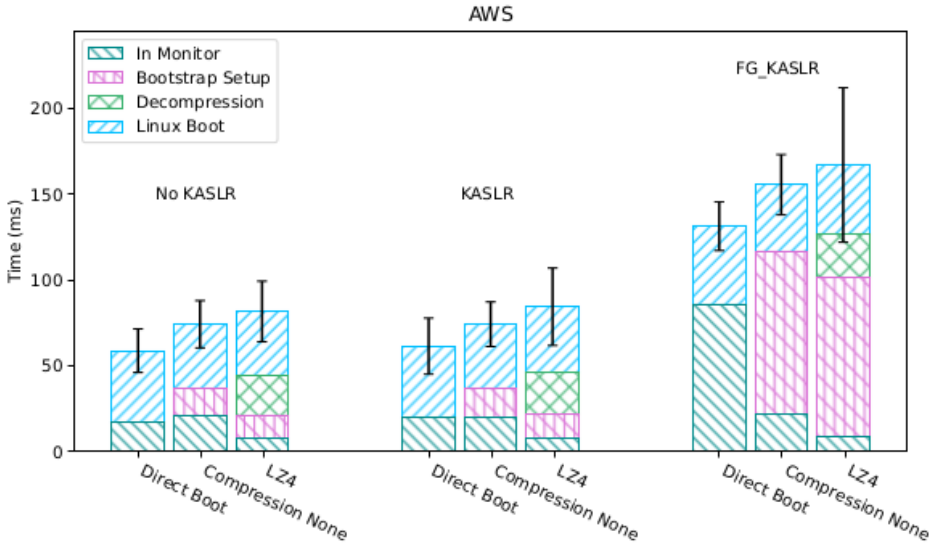


Fig. 9 (b), “KASLR in the age of MicroVMs”

各バーのラベル（本論文はラベルが非常に分かりにくい...）
- **Direct Boot**: 提案手法
- **LZ4**: 通常の圧縮カーネルと bootstrap loader の組み合わせ
- **Compression None**: vmlinuz の中身を非圧縮カーネルイメージに変えたもの（つまり bootstrap loader ありでかつ解凍の手間をなくせる）

各グループのラベル
- **KASLR**: 全体のベースのみランダム化
- **FG_KASLR**: 関数ごとにランダム化

Takeaways:
- 全ケースで提案手法が最も速い起動時間を実現
- AWS 以外のカーネルも同様の傾向（論文に図あり）

Firecracker との比較
- KASLR では AWS で 3.7% 増、FGKASLR では AWS で 2.15 倍（論文にも図なし）

実験結果（2/3）

- ゲストに割り当てるメモリ量による起動時間への影響は？

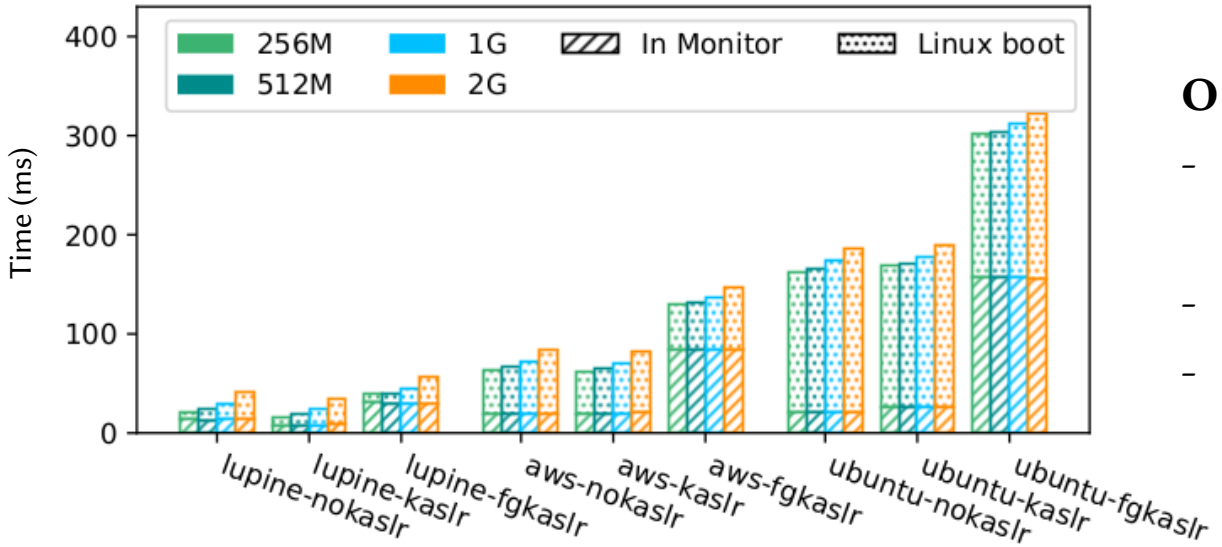


Fig. 10, “KASLR in the age of MicroVMs”

Observation:

- ゲストのメモリ量を変えると Linux boot（Linux に飛んでから init が起動するまで）は伸びる
- 一方 In-monitor でかかる時間は一定
- Linux boot の部分は ASLR あり・なしで変化なし

結論：提案手法を使用した場合の起動時間は、ゲストのメモリ量の影響を受けない

実験結果（3/3）

- In-monitor KASLR による起動時間以外のオーバーヘッドは？
  - KASLR あり・なしで LEBench の性能を比較（baseline: aws-noaslr + Firecracker）

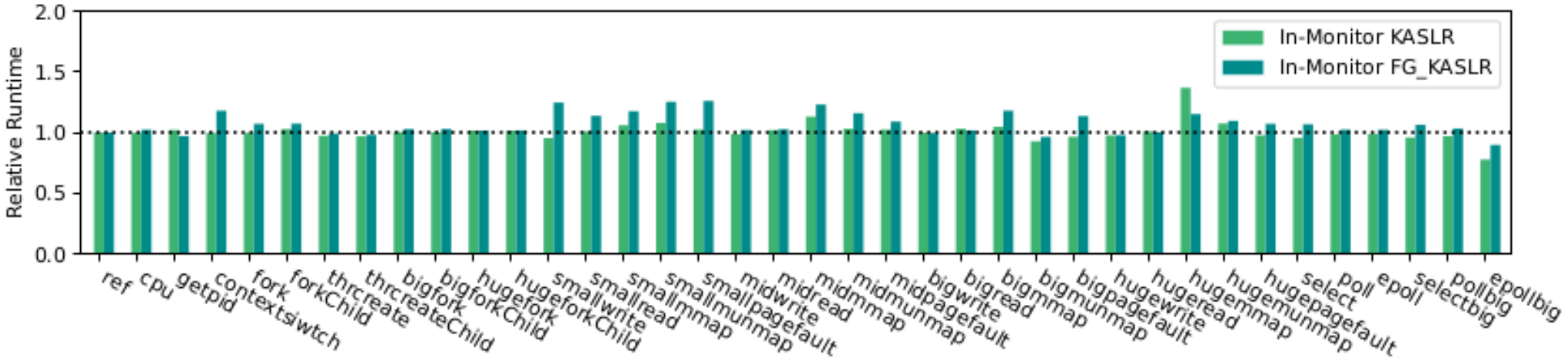


Fig. 11, “KASLR in the age of MicroVMs”

  - 関数単位でランダム化すると平均約 7% の性能低下
    - L1 キャッシュミスが少し増える ← 書いていないけど I-cache の意味か
    - Hot area を単一セクションにする議論が LKML であり ← セキュリティレベル低下？

まとめ

- VMM 内で動作する KASLR を実現した
  - MicroVM 時代にはゲスト起動の高速化が重要
  - そのため KASLR を行う bootstrap loader が使われない
  - KASLR ができかつ bootstrap loader を使う場合より高速な起動を実現

- 私見：手法的にはやっただけに見えるが目の付け所が重要？
  - MicroVM で KASLR が省略されているという重要な指摘をした