UniBPF: Safe and Verifiable Unikernels Extensions

Kai-Chun Hsieh
Advisor: Masanori Misono
Chair of Computer Systems
https://dse.in.tum.de/

16.05.2023 – 15.11.2023

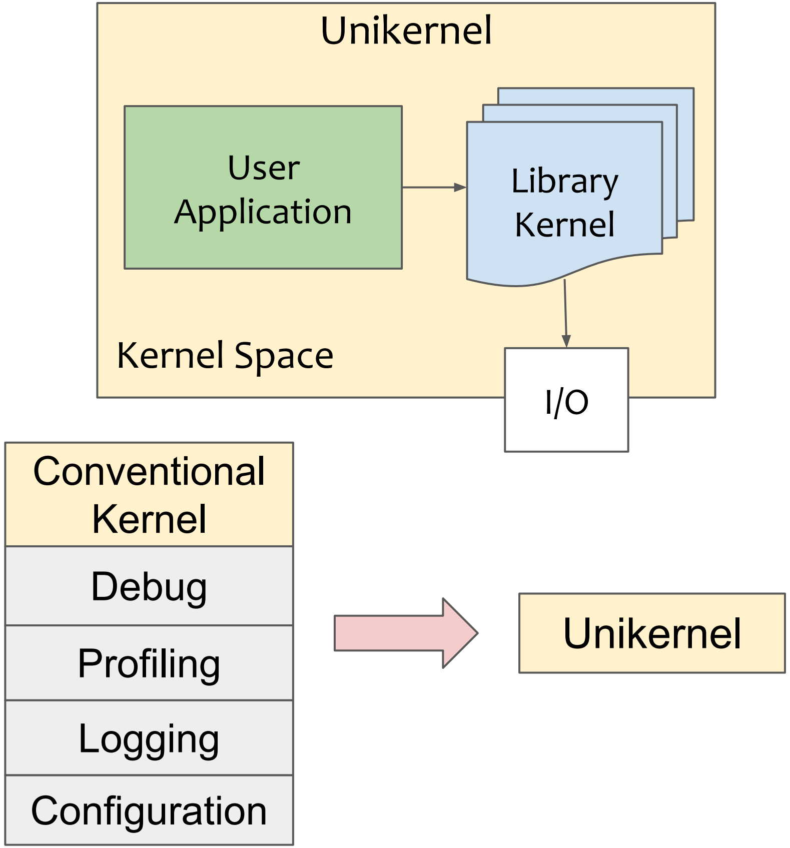
Motivation

Unikernels

- Kernel as a library
- Eliminate unneeded components.
- Optimize system procedures, e.g., system calls
- Compact, efficient, secure

But…

- Lack of **debuggability**
- Lack of **observability**
- Lack of **runtime-extensibility**



State-of-the-art

Extensible Unikernels with **BPF:**

- e**BPF Runtime** + kernel tracing with **interpreters**. But…

❌ **Lack of verifier:**

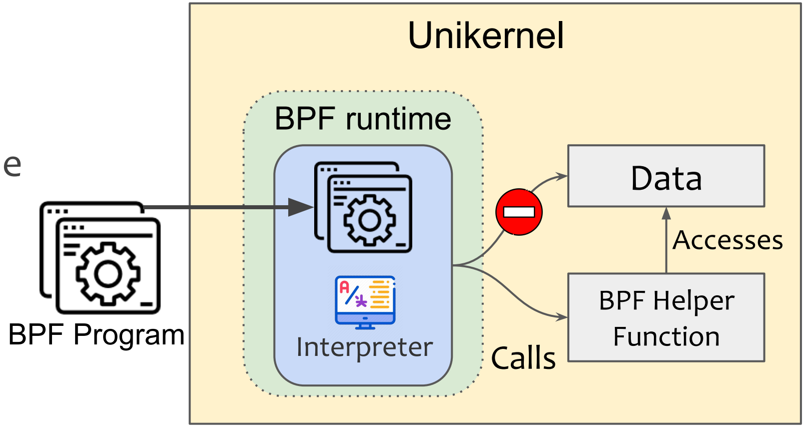
- Use an interpreter to provide sandboxed runtime

❌ **Insufficient security guarantee:**

- Cannot resist runtime errors

❌ **Inefficient runtime:**

- Our work: ≤ 600% **slowdown** in instruction level v.s. JiT compiled

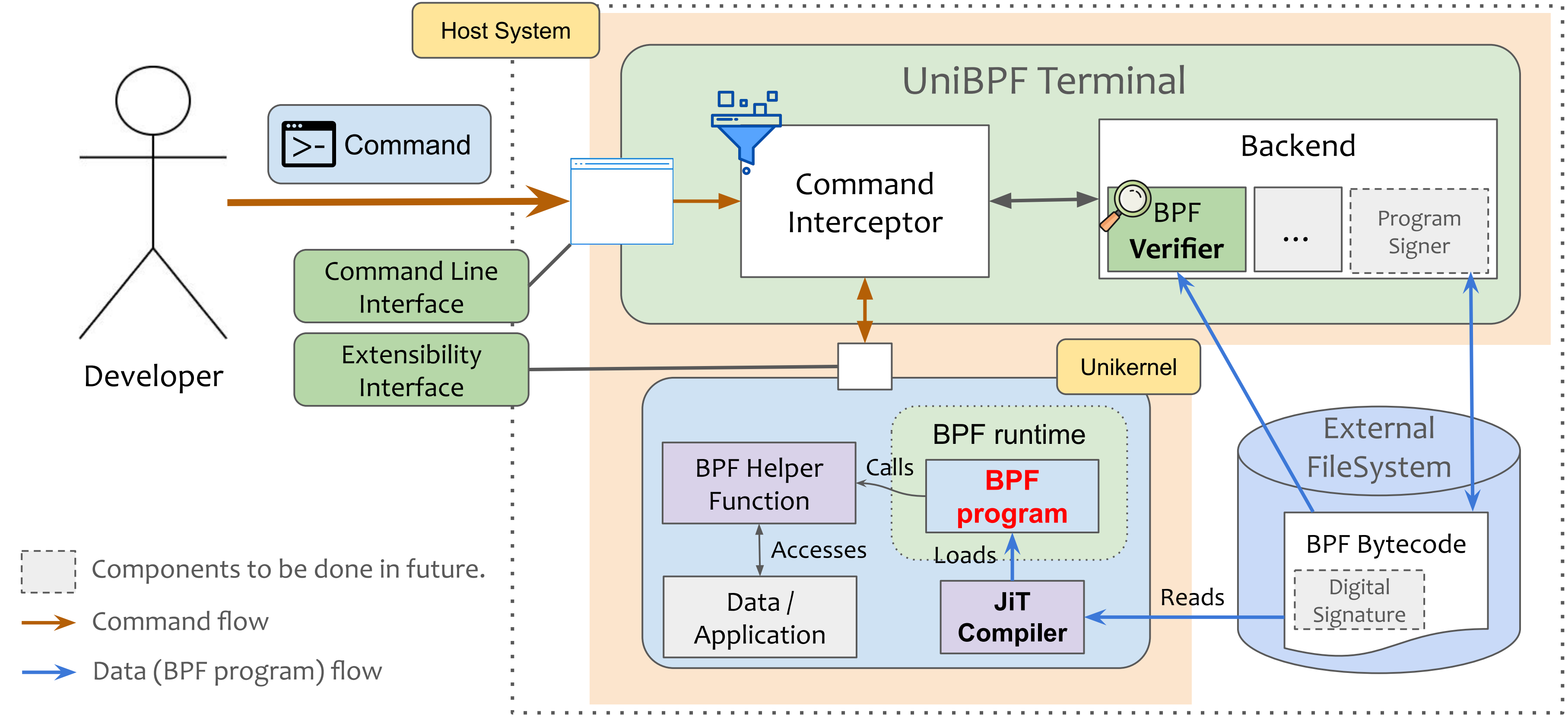


Research Question

How can we have a safe and verifiable extension for Unikernels?

- Design Goal
  - **Safety:** Ensure safety of executing extension binaries
  - **Sustainable Design:** Easy to use, easy to maintain
  - **Performance:** Acceptable overhead and improve BPF runtime efficiency

System Overview



Background: extended Berkeley Packet Filter (eBPF)

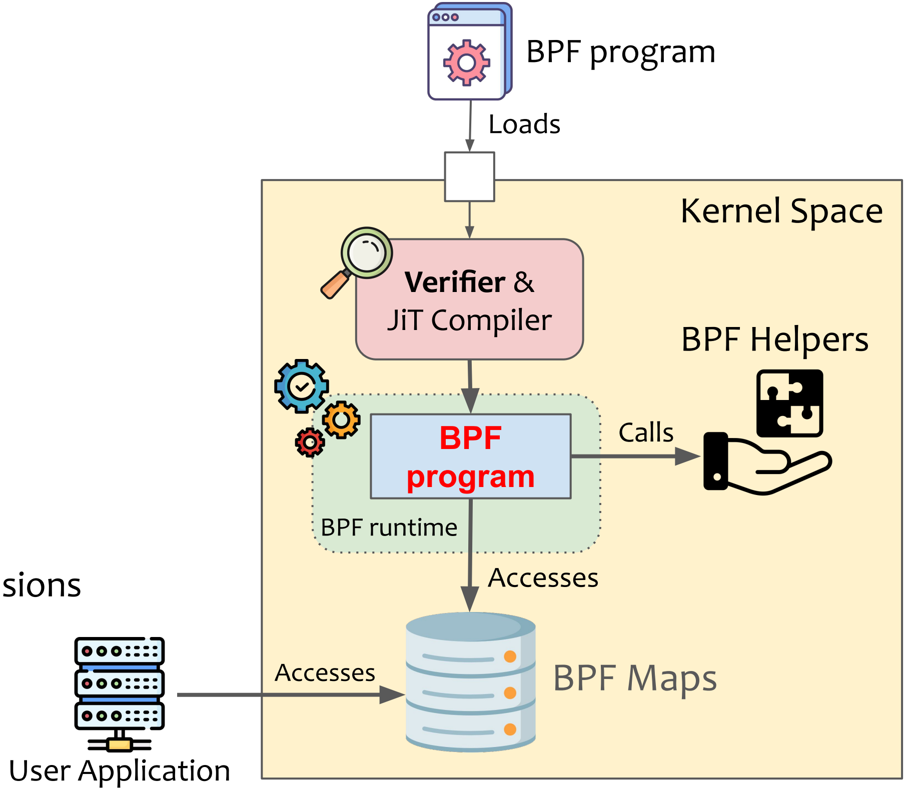
Lightweight in-kernel language VM

Sandbox property can be ensured by:

- Using **interpreters** (weaker)
- Using **verifiers** to verify in advance (stronger)
  - Detects potential sandbox escalation
  - Forbid undefined behaviors

Useful features:

- Maps (kv-store)
- Helper functions
- Program Types: Runtime context & helper permissions



Outline

- ~~Motivation & Background~~
- Design Challenges
- Evaluation
- Further Ideas

Design Challenges

1. Impact of Verification Processes on Unikernel Applications’ Runtime

2. Feasibility of Integrating Verifier into Unikernel Application

3. Usability and Maintainability: Configuring Shared Verifier for Different Unikernels

① Verification can block Unikernel applications

❌ Lack of **multi-processing** support:

- Application is the only process

❌ Lack of comprehensive schedulers:

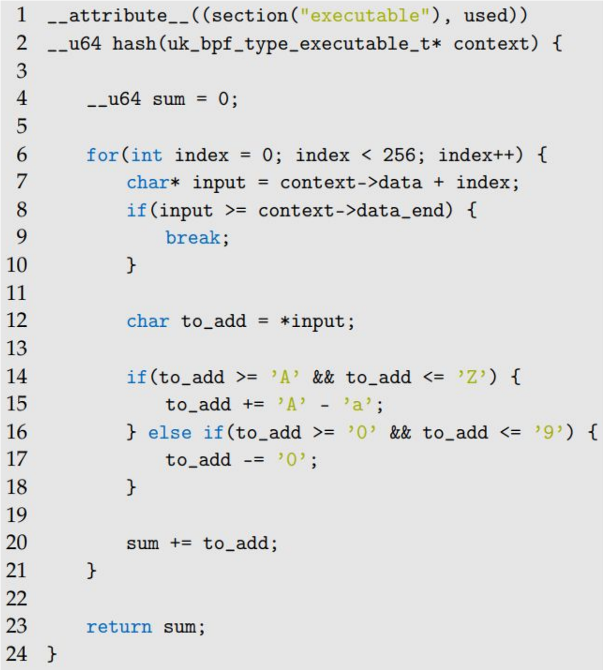
- CPU resource is released by voluntary *“yields”*

🐌 **Verification is time-consuming!**

- Our example BPF program: **12.05 ms** to verify 26 instructions.
- Lower-Bound: **8.82 ms**

⌛ With common approaches:

- Clients may experience huge **latencies**



```
__attribute__((section("executable"), used))
__u64 hash(uk_bpf_type_executable_t* context) {

    __u64 sum = 0;

    for(int index = 0; index < 256; index++) {
        char* input = context->data + index;
        if(input >= context->data_end) {
            break;
        }

        char to_add = *input;

        if(to_add >= ’A’ && to_add <= ’Z’) {
            to_add += ’A’ - ’a’;
        } else if(to_add >= ’0’ && to_add <= ’9’) {
            to_add -= ’0’;
        }

        sum += to_add;
    }

    return sum;
}
```

💡 Put BPF verifiers as processes on the host system where schedulers are more flexible

② BPF Verifiers Are Too Complicated to Integrate

- Common BPF verifiers are **complicated:**
  - PREVAIL (PLDI'19): 27,000 Lines of code
  - KLINT (NSDI'22): 13,000 Lines of code

- Common BPF verifiers need **complicated runtime:**
  - PREVAIL: C++ runtime library
  - KLINT: Python Interpreter
  - Linux BPF verifier: GPL License, Depends on Linux

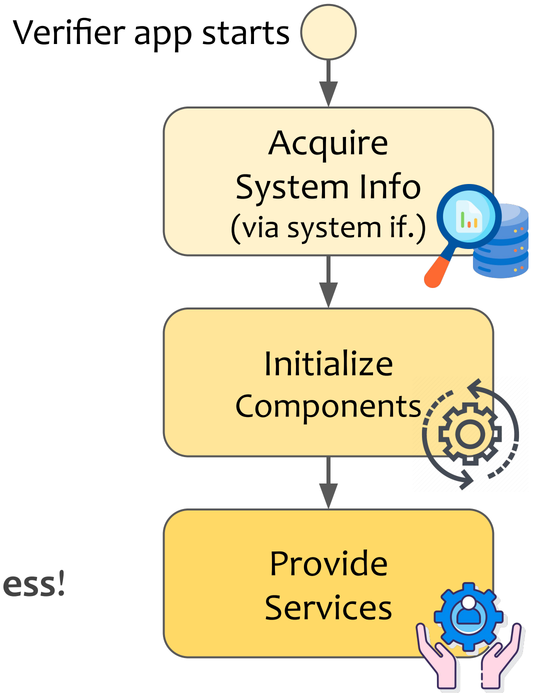
💡 Put BPF Verifiers on the host system utilizing the host system's runtime environment

③ Customizability Impedes Building a Unified Solution

Our Goal: Maintain customizability for BPF runtime

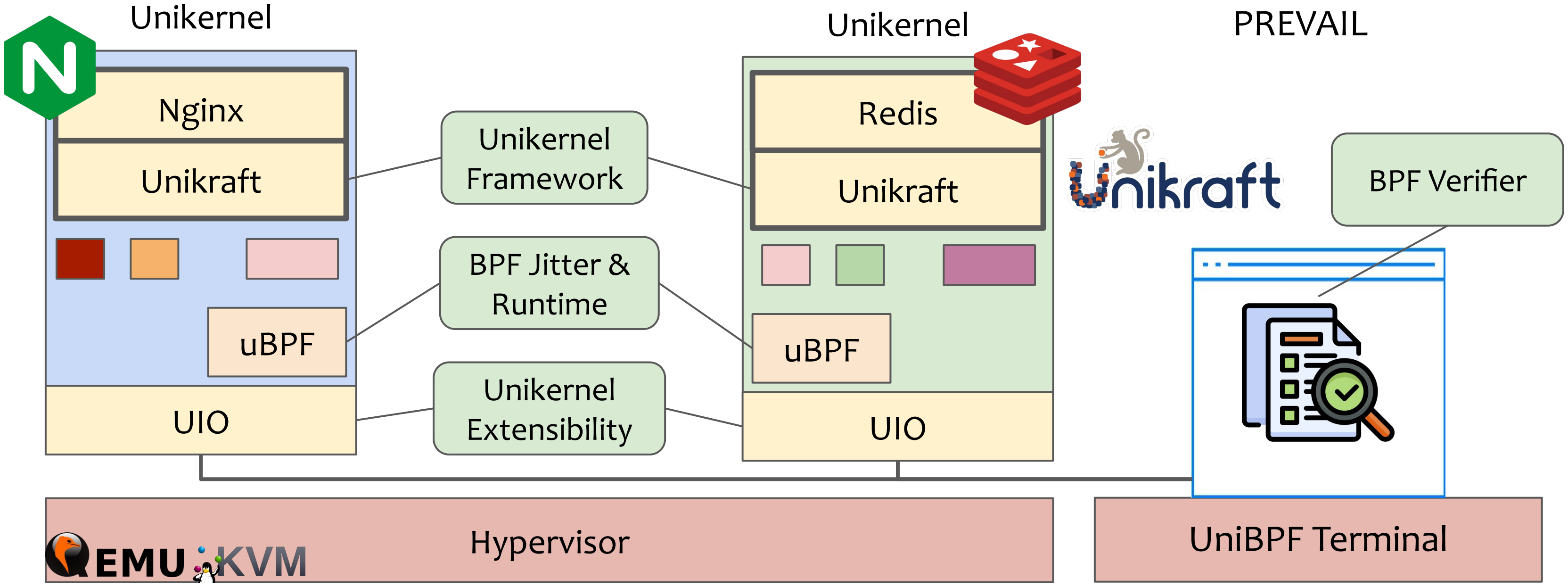
- BPF Helper functions & program types
- Keep compactness
- Increase our system’s usability

- But, without a standard framework:
  - Each Unikernel needs one BPF verifier: **Unmaintable**!
  - Waived support for customizable parts: Our work is **Meaningless**!



💡 We provide libraries that allow developers to easily export their BPF runtime specifications

Implementation



Outline

- ~~Motivation & Background~~
- ~~Design Challenges~~
- Evaluation
- Further Ideas

Evaluation - Safety

Evaluation Program	Result - Interpreter	Result - JiT Compiled	Result - UniBPF
OOB*	Terminated	Exploited	**Denied**
OOB* with Nullptr	Terminated	System crashed	**Denied**
Infinity Loop	System freezes	System freezes	**Denied**
Division by Zero	Error Ignored	Error Ignored	Partially **Denied**
Instruction Type Safety	Error Ignored	Error Ignored	**Denied**
Program Type Safety	Error Ignored	Error Ignored	**Denied**
Helper Function Type Safety	Error Ignored	Error Ignored	**Denied**

🟨: Memory Safety

🟩: Termination

🟦: Runtime Errors

🟪: Type Safety

UniBPF provide a safer BPF runtime extension for Unikernel

*) OOB: Out of bound memory access

Evaluation - Verification and JiT Overhead

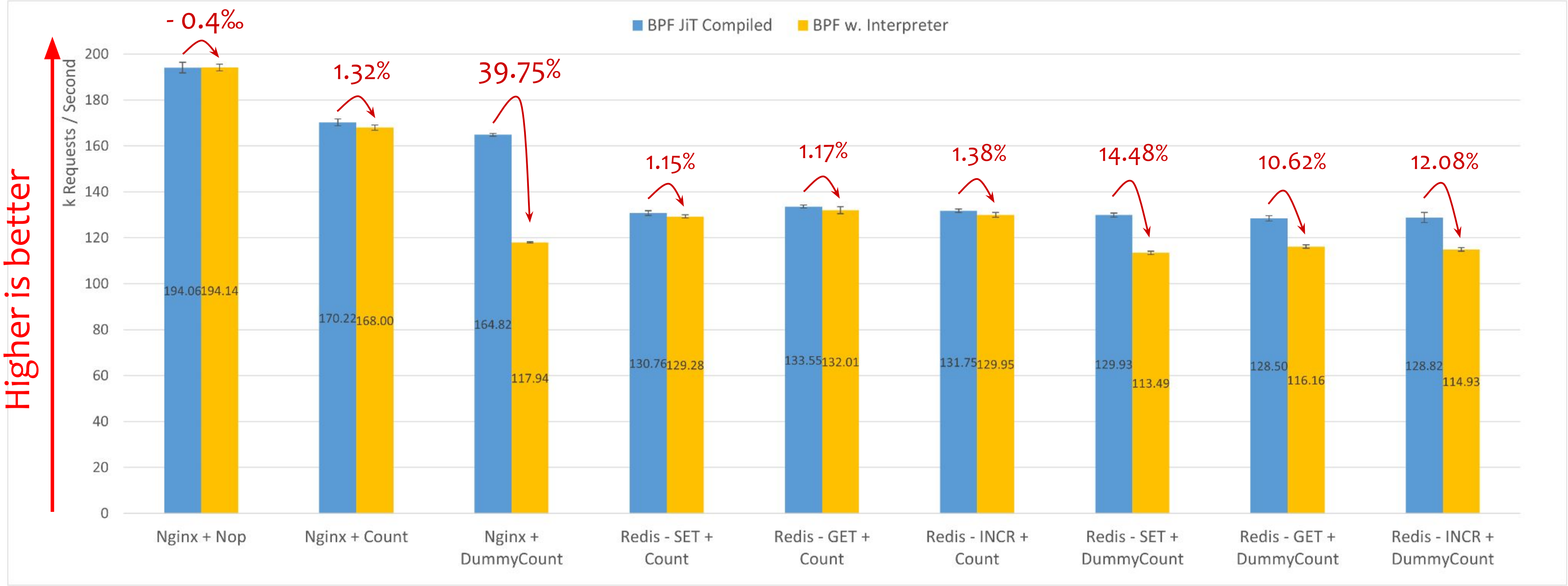
	Instructions	Verification Time Overhead*	Verification Memory Overhead*	JiT Time Overhead
Nop	2	8.82 ms	3328 kb	9.74 ms
Hash	26	12.05 ms (7.43 instr./ ms)	4096 kb	9.79 ms
Adds	1002	43.60 ms (28.75 instr./ms)	5056 kb	9.85 ms

🟥: The lower bound overhead of the entire system.

* : Overhead made to the host system.

The JiT compilation overhead and the corresponding verification overhead are negligible

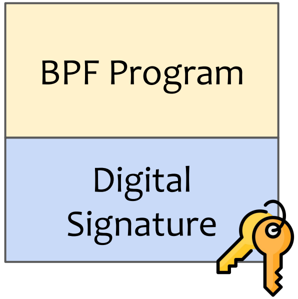
Evaluation - BPF Kernel Tracing **Nginx** and **Redis**



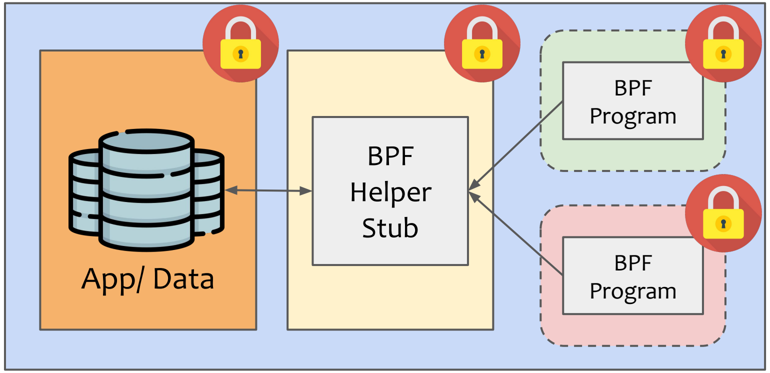
The improvement in jitted BPF runtime is more significant as the program size increases

Further Ideas

- Ensure verification integrity with **digital signature**



- More robust BPF runtime isolation:
  - Intel **MPK**
  - BPF helper function stub



- Support verification with BPF maps
- BPF program as configurations
- Secure verification process from malicious cloud provider: **Confidential VM**

Conclusion

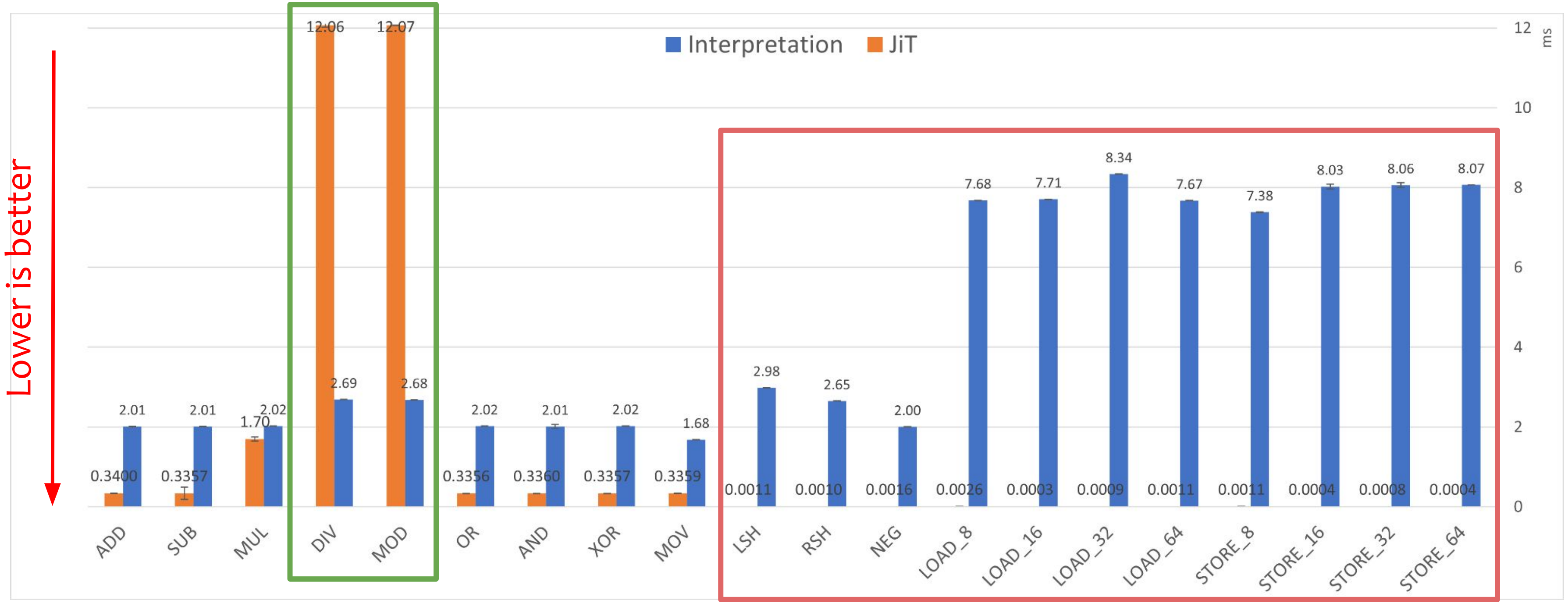
- UniBPF provides **safer** BPF runtime
  - Resist runtime errors interpreters cannot
  - Protect jitted runtime from malicious codes
- Only brings **negligible overhead**
- Enables **more efficient runtime** through JiT compilation
  - Instruction level: Up to 600%
  - Kernel-Tracing:
    - Nginx: 40% ~
    - Redis: 14.48% ~

Try it out!

https://github.com/TUM-DSE/ushell/

Backup

Evaluation - Instruction Level Performance



JiT-Compiled BPF Runtime is **up to 600%** faster and may trigger **hardware level optimization**

Deeper Explanations

Research Gap and Our Assumption TODO

❌ Interpreted mode is **slow:**

- BPF native mode **insecure** unless bytecodes verified

❌ The interpreter’s security guarantee is week

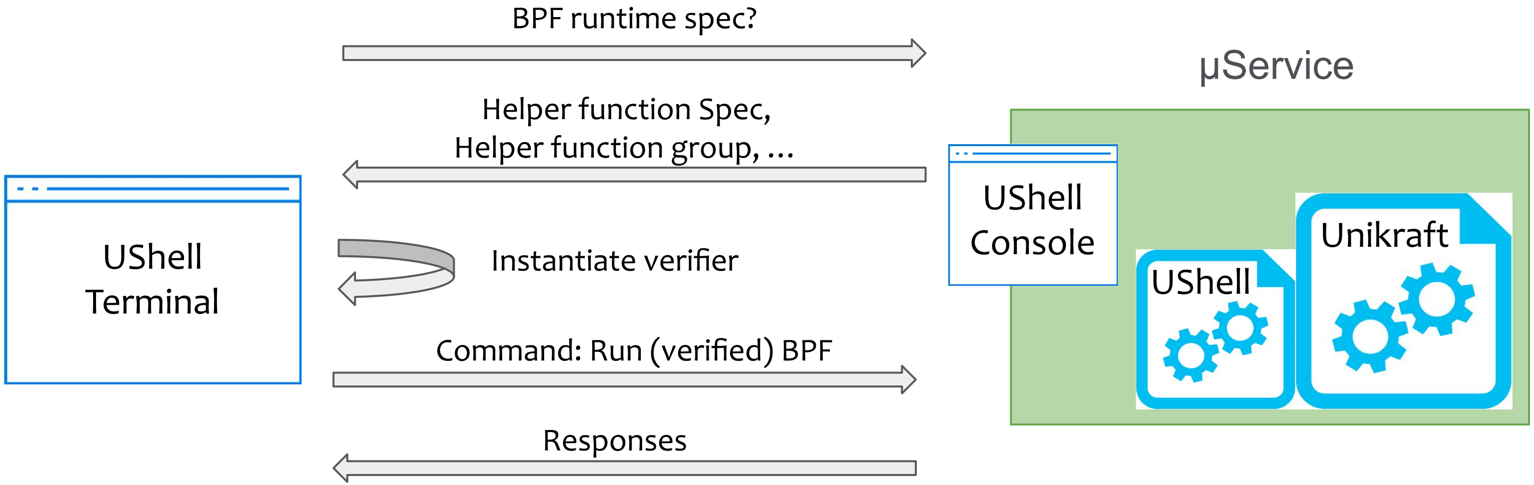
- BPF helper function invocation unchecked
- Some correctness of program unchecked, e.g., termination
- Helper function permissions are unchecked

❌ Runtime isolation is weak

- **Software** address-space boundary check on the fly: **slow**
- **Verifiers can not be perfect (e.g., CVE-2021-33624)**

How can we integrate BPF verifier with unikernel to improvement and security?

Implementation: Workflows

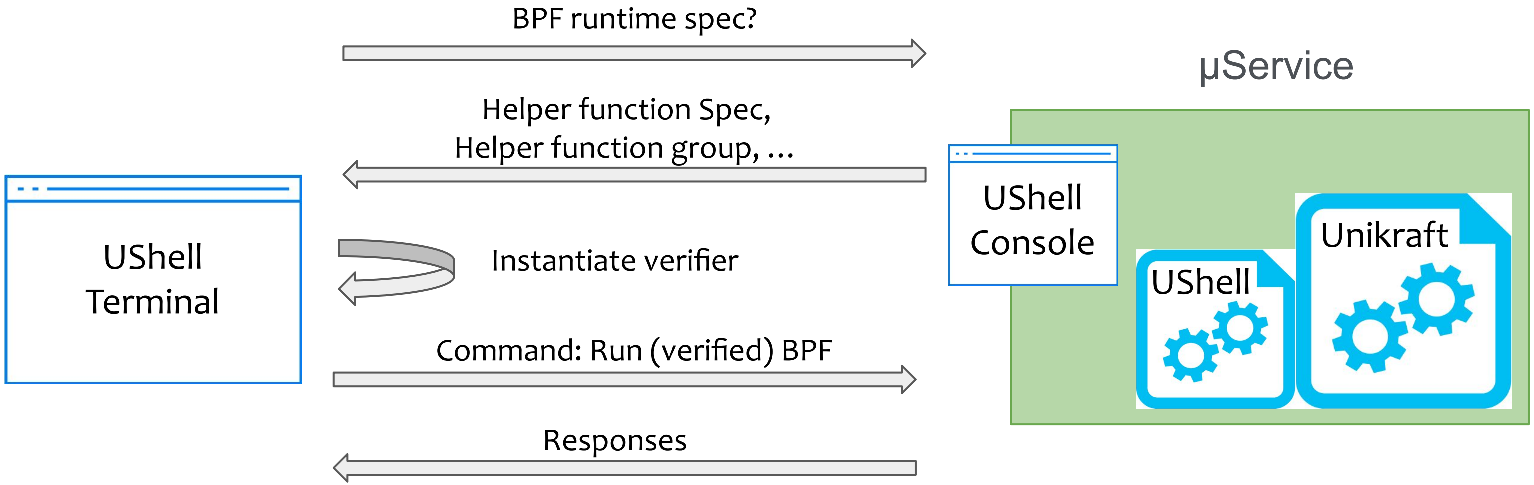


UShell Terminal grabs verification info in runtime and build customized verifier in real time for different µServices.

Further Ideas

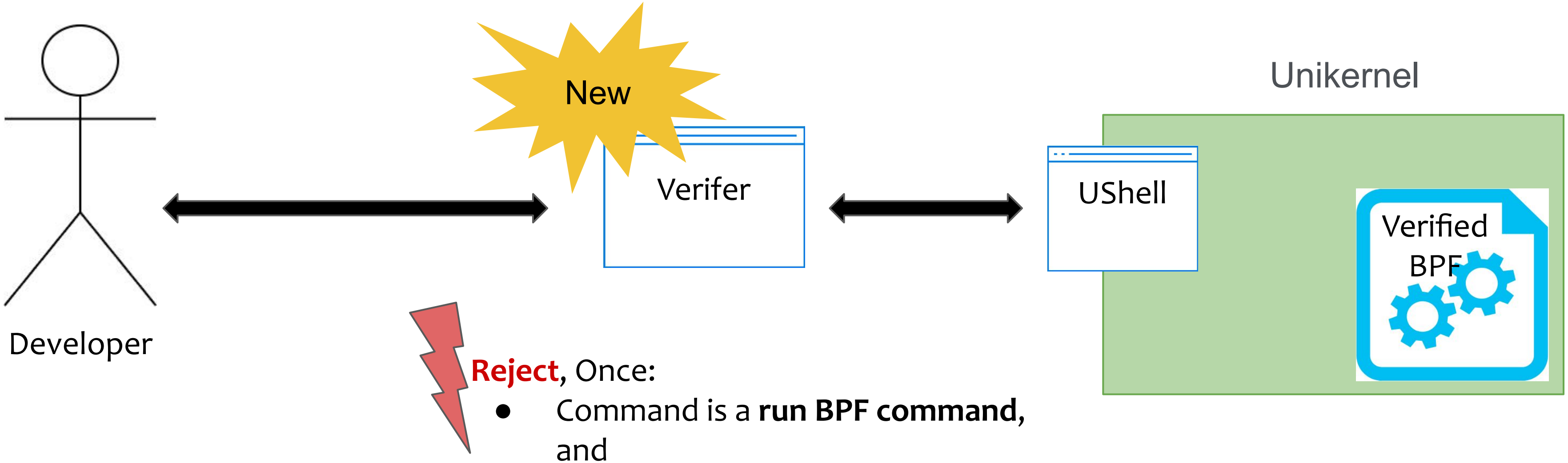
- **Static** verifiers cannot find out every security hazards:
  - e.g., Access to memory with an offset acquired from “pkt” memory area provided by system.
- **Static** verifiers can make also mistakes: false positive.
- Solution:
  - Containerize BPF runtime:
    - ■ MPK: Most feasible within the time limit of this project.
    - ■ With processes: Much more complicate, left for future discussions.

Implementation: Workflows



UShell Terminal grabs verification info in runtime and build customized verifier in real time for different µServices.

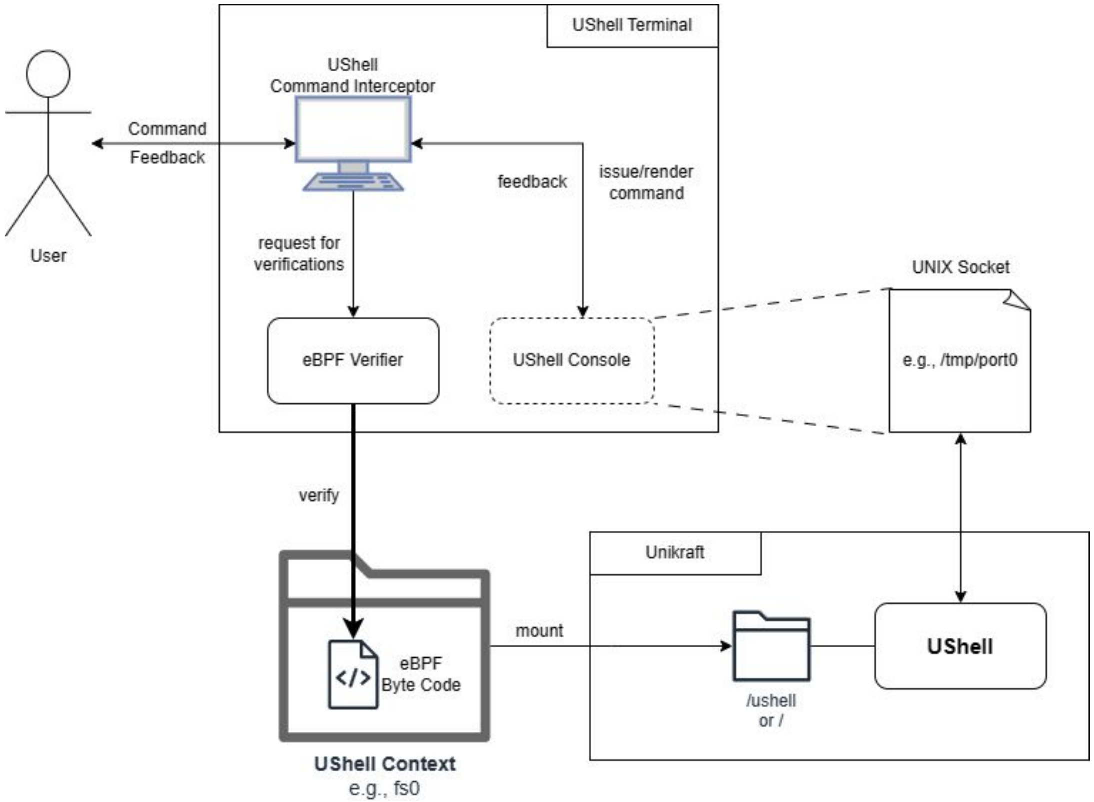
System Overview



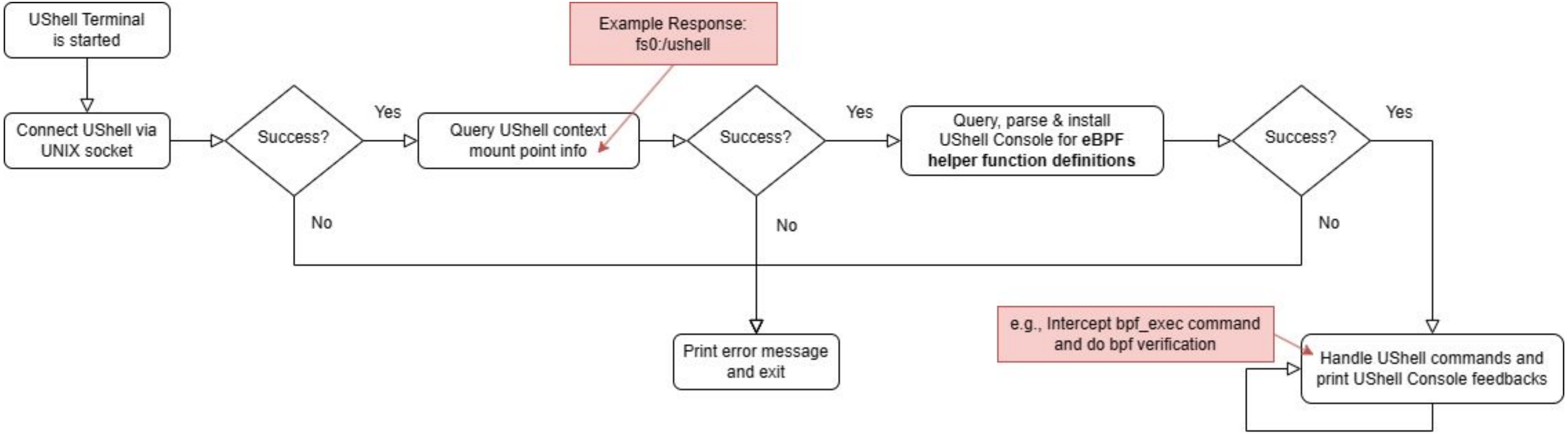
Reject, Once:

- Command is a **run BPF command,** and
- Given BPF bytecode is **not safe**

UShell-Terminal: Command Interception

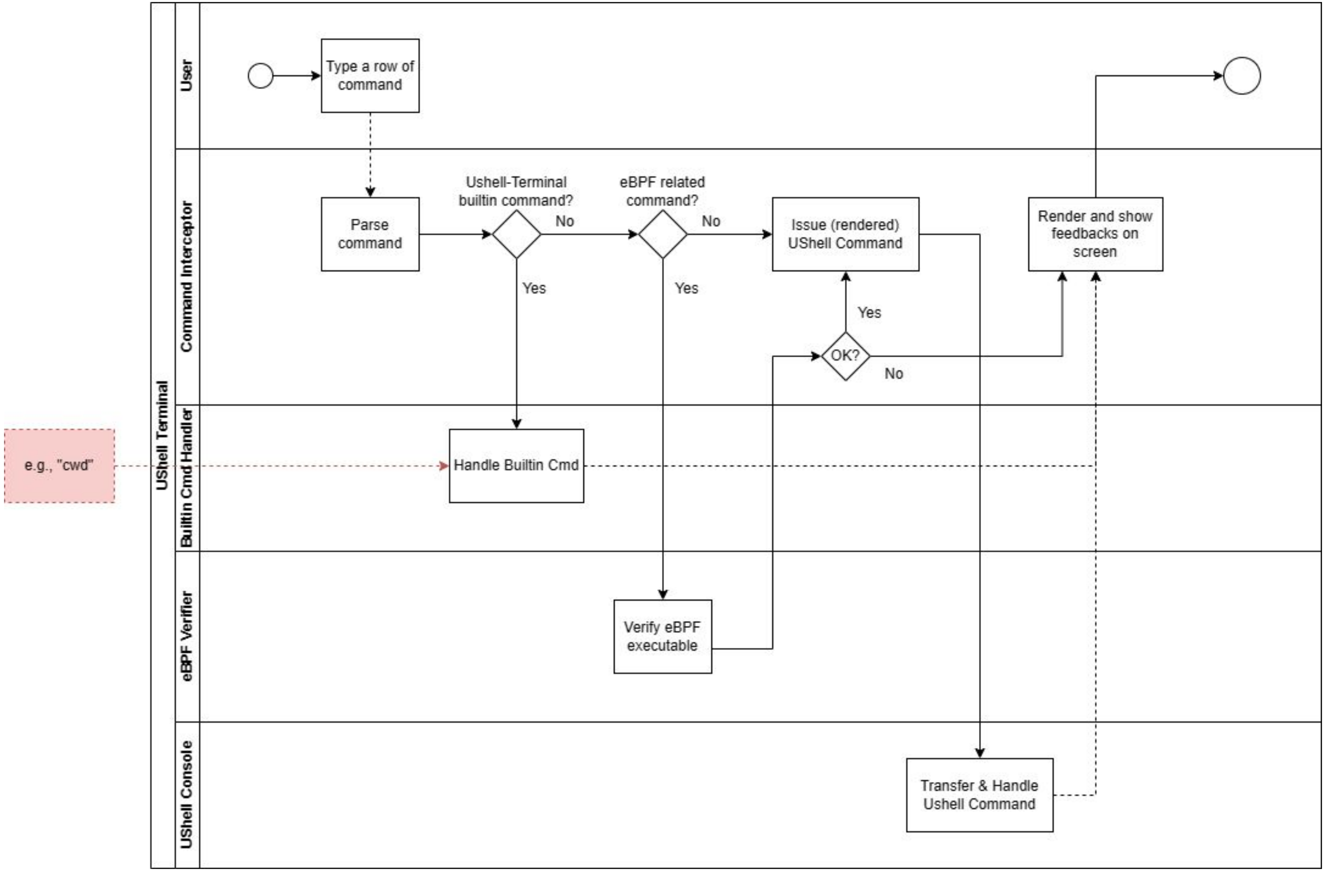


UShell Terminal: Implementation Overview


UShell Terminal is started
Connect UShell via UNIX socket
Success?
Yes
No
Query UShell context mount point info
Example Response: fs0:/ushell
Success?
Yes
No
Query, parse & install UShell Console for **eBPF helper function definitions**
Success?
Yes
No
Print error message and exit
e.g., Intercept bpf_exec command and do bpf verification
Handle UShell commands and print UShell Console feedbacks

UShell Terminal grabs verification info in runtime and build customized verifier in real time for different target µServices.

UShell-Terminal: Work-Flow



Backup

1

Impact of Verification Processes on Unikernel Applications’ Runtime.

2

Feasibility of Integrating Verifier into Unikernel Application

3

Usability and Maintainability: Configuring Shared Verifier for Different Unikernels

Hardware Assisted Secure BPF Runtime

Problem:

- Verifier cannot be perfect **(e.g., CVE-2021-33624)**
- Once runtime compromised, system compromised

Solution:

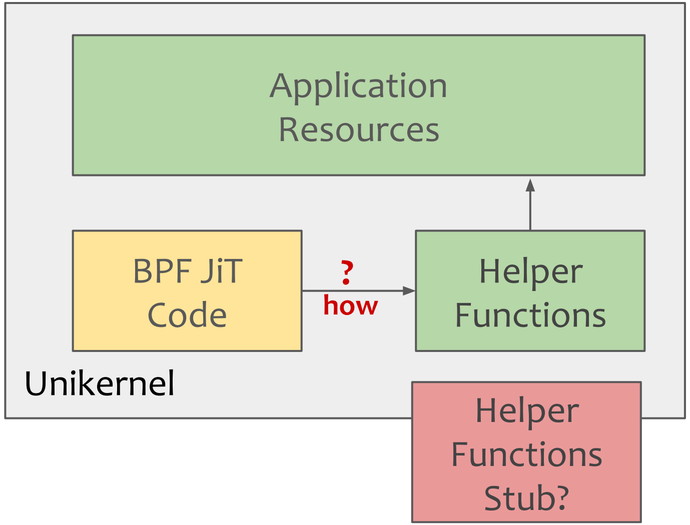
- Hardware assisted BPF runtime isolation
- Hardware provide solid security measurements

Available solutions in hand:

- **MPK:**
  - Split memory into domains
  - Lightweight

Undecided design issues:

- How can runtime access helper functions in other memory domain?



State of the Art: Unikernel + BPF

Why BPF:

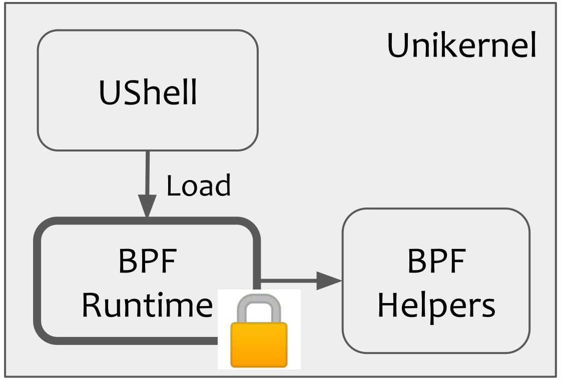
- UShell can run **arbitrary binary**, but, it may be **dangerous**
- BPF programs are designed to be **verified and sandboxed**
- Even not verified, interpreter can **check them on the fly**

State of the Art:

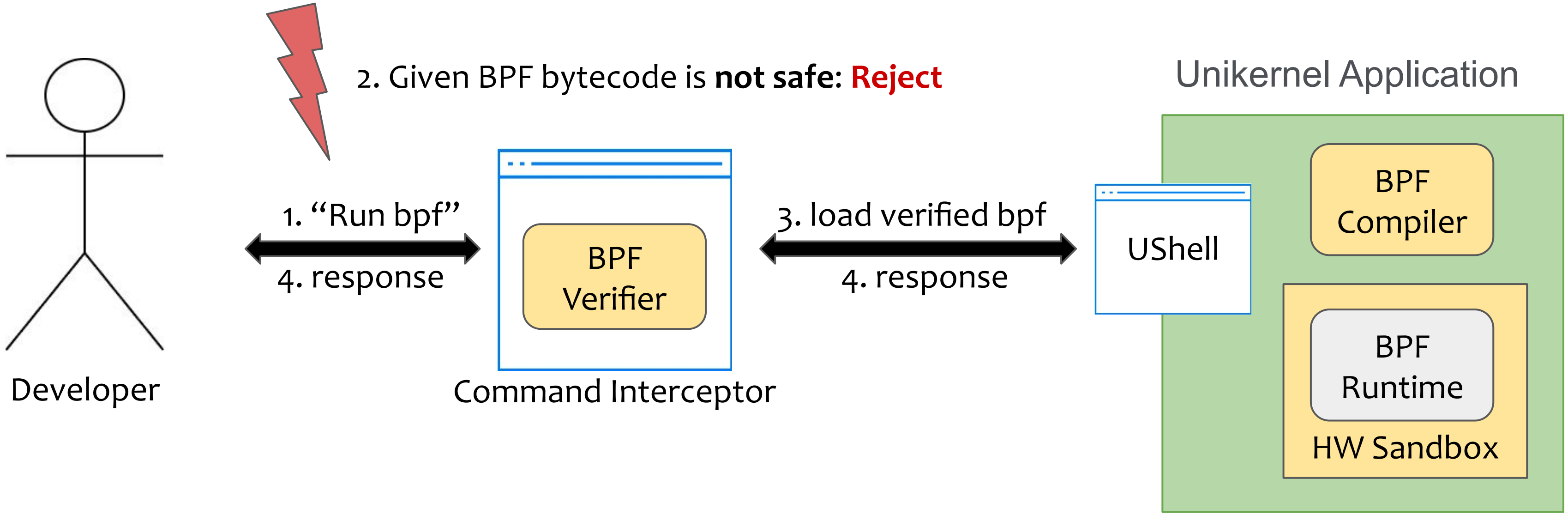
- Unikernel + UShell + BPF interpreter

Capabilities:

- Run BPF bytecodes
- Isolation
- BPF helper functions
- Example use case: Kernel tracer



System Overview



2. Given BPF bytecode is **not safe**: **Reject**

Unikernel Application

Developer

1. “Run bpf”

4. response

BPF Verifier

Command Interceptor

3. load verified bpf

4. response

UShell

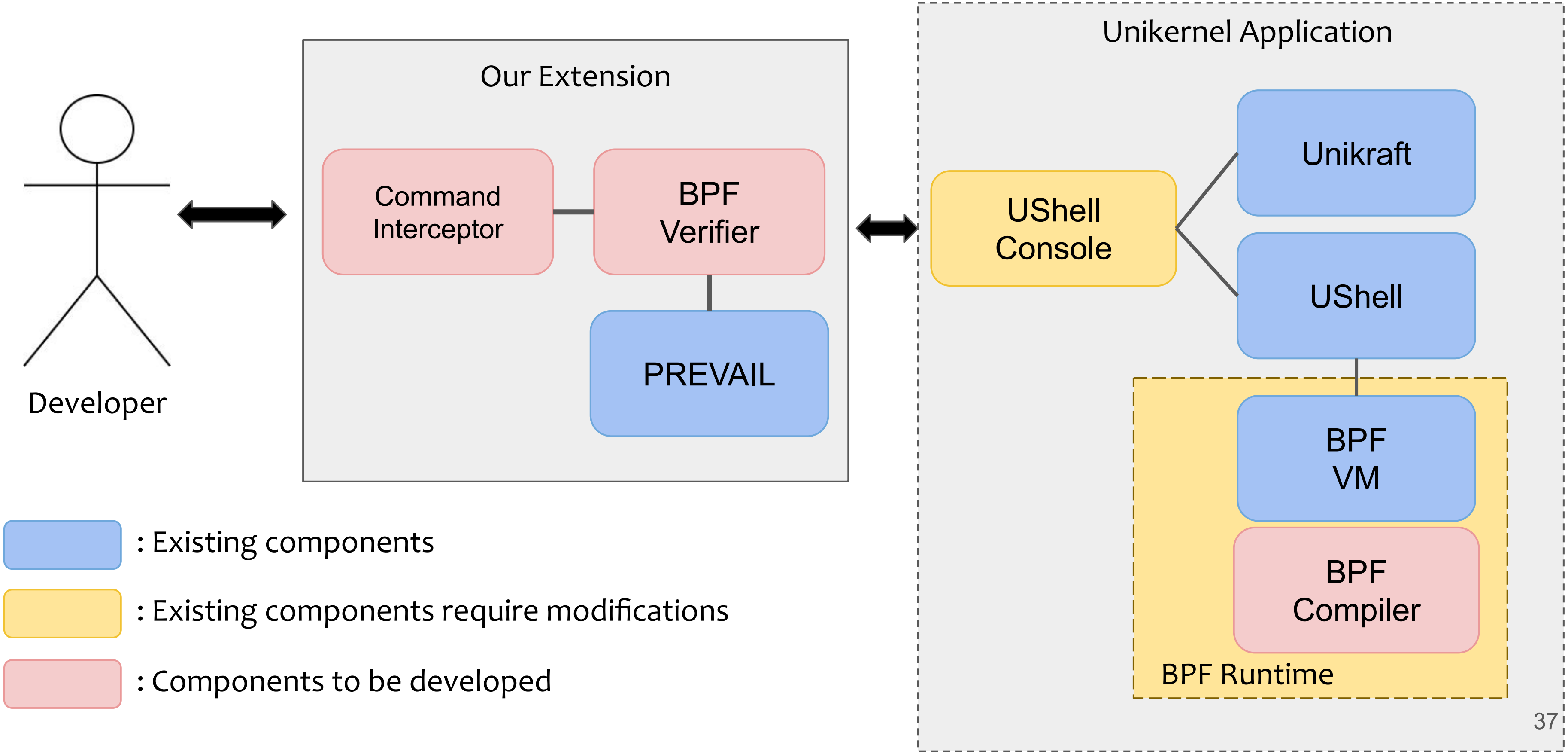
BPF Compiler

BPF Runtime

HW Sandbox

🟨 : components to be integrated

System Components to be done



🟦 : Existing components

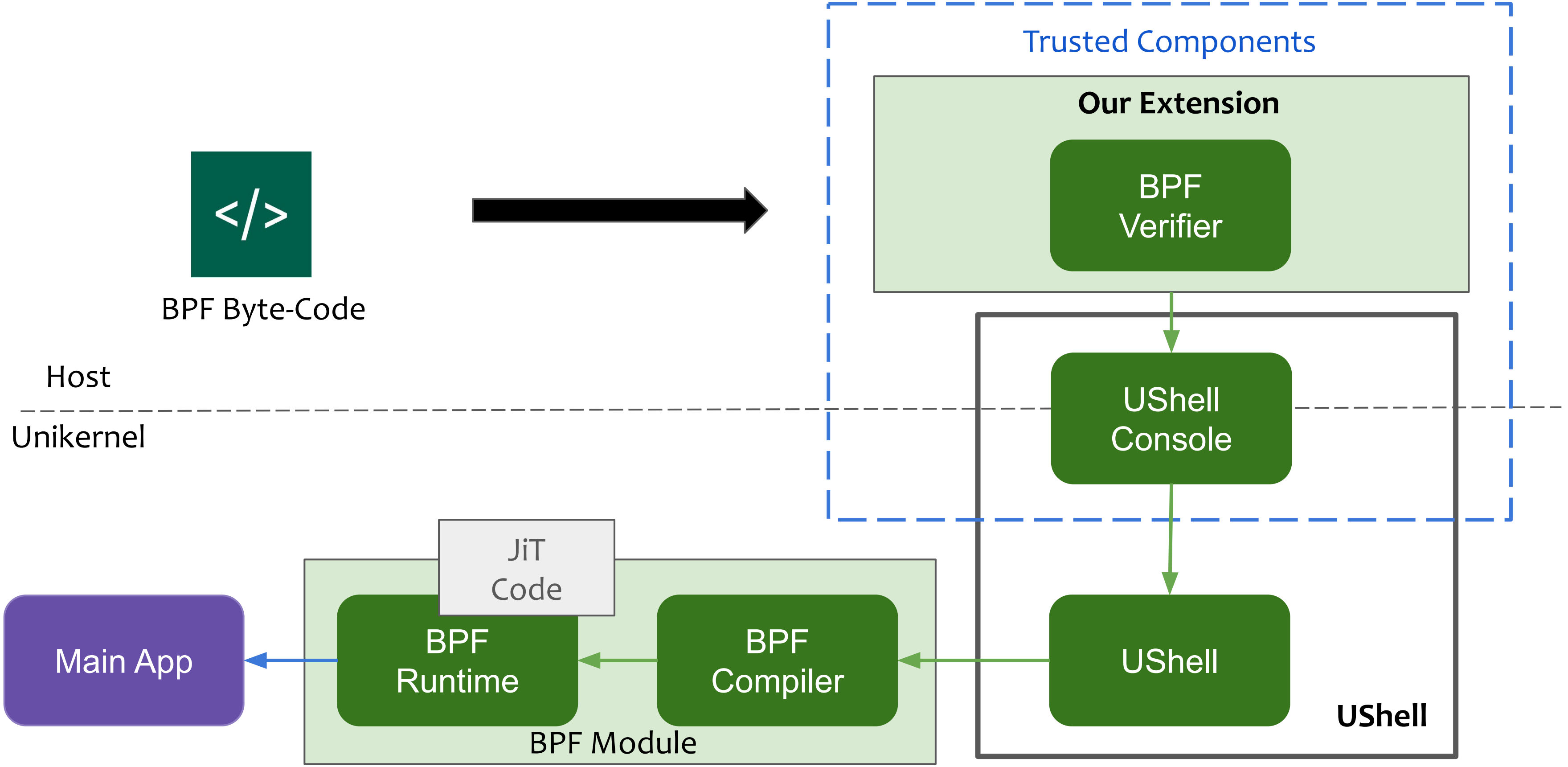
🟨 : Existing components require modifications

🟥 : Components to be developed

Design Goals

- Safe BPF language runtime:
  - BPF bytecodes are verified
- Efficient BPF language runtime:
  - Run BPF program under JiT compiled mode
- Secure BPF language runtime:
  - Stronger isolation promise

Our Solution



Outline

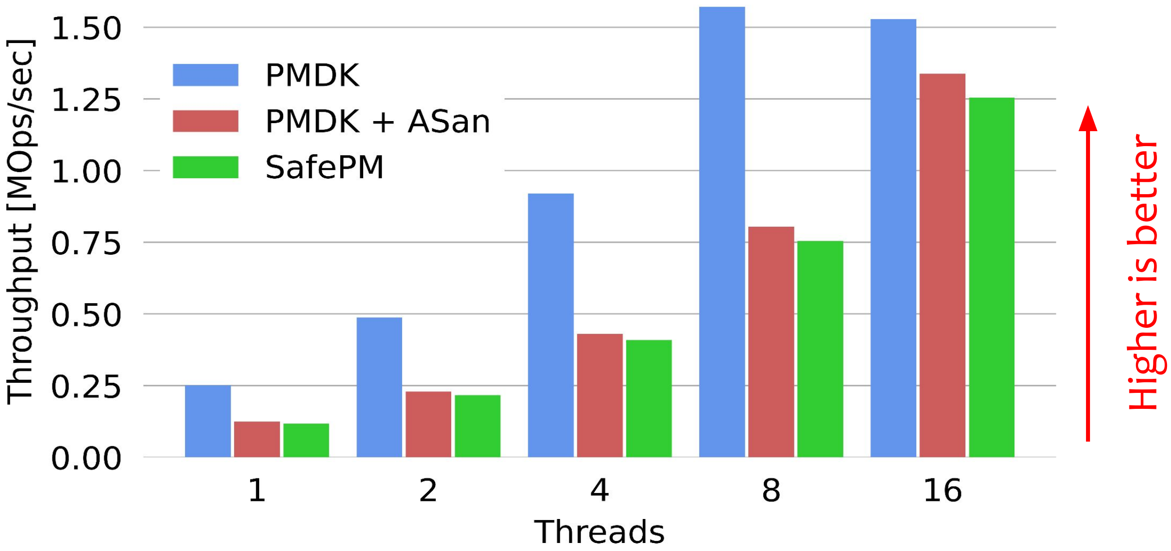
- ~~Motivation & Background~~
- ~~Design~~
- Evaluation
- Further Ideas

Schedule

	May	June	July	Aug	Sep	Oct	Nov	Dec
Planning	paper reading							
Implement		Verifier	System Integrate	Runtime Isolation				
Evaluation			Verifier & Security Promises		Security Promises			
Writing					🟨	🟨	MSc Thesis	Paper

Performance overhead

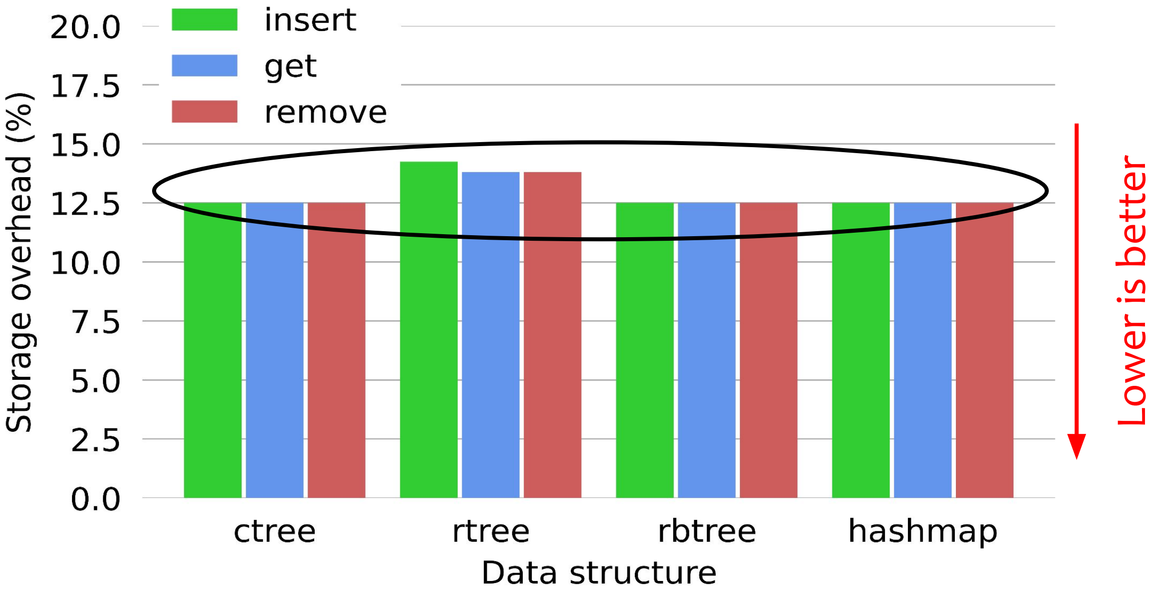
Persistent KV-store benchmark, **10M** ops, **50%** reads / **50%** writes



SafePM incurs similar performance overheads with ASan

Space overhead

Persistent indices, insert/get/remove workloads, relative to PMDK



SafePM increases the required PM space by 12.5% due to the PSM

Efficiency

RIPE benchmark, **1334** memory safety exploits

Variant	**Exploitable memory safety bugs**
DRAM	**320**
DRAM + ASan	**28**
PM + ASan	**131**
PM + SafePM	**28**

SafePM provides equivalent memory safety effectiveness for PM with ASan

Summary

Current memory safety approaches are not designed for PM applications

- PM programming model
- data/metadata durability & crash consistency
- recovery paths

SafePM:

- comprehensive spatial and temporal memory safety
- no source code modifications
- crash consistency & high coverage

Try it out!

https://github.com/TUM-DSE/safepm