

UniBPF: Safe and Verifiable Unikernels Extensions

Kai-Chun Hsieh

Advisor: Masanori Misono

Chair of Computer Systems

<https://dse.in.tum.de/>

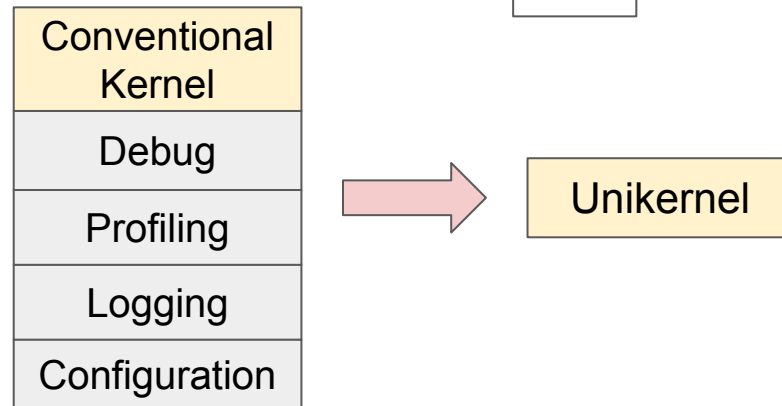
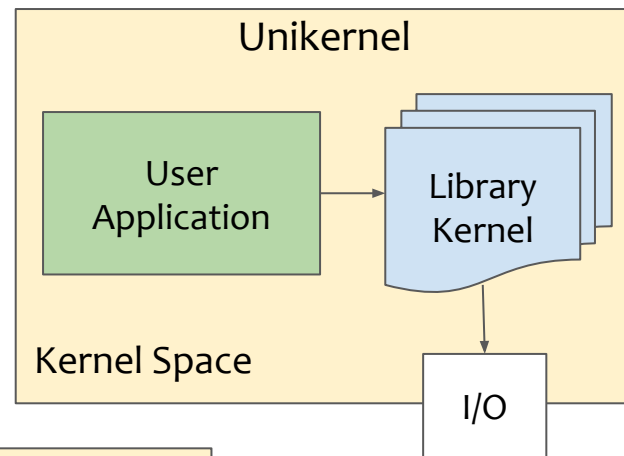


Unikernels

- 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**.



Extensible Unikernels with **BPF**:

- **eBPF Runtime** + kernel tracing with **interpreters**. But...



Lack of verifier:

- Use an interpreter to provide sandboxed runtime.



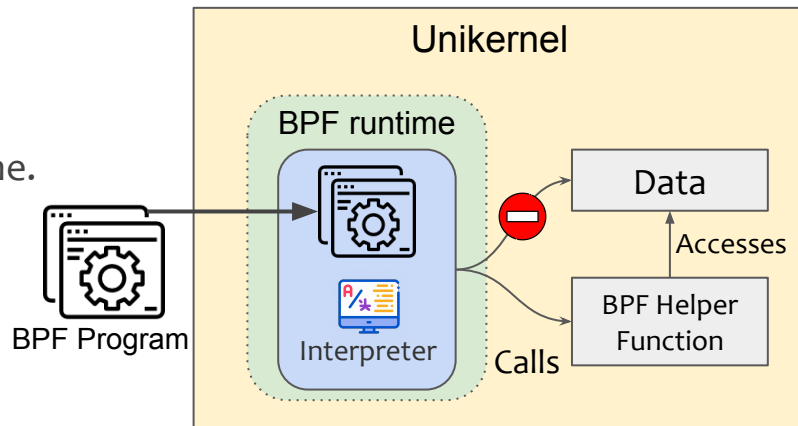
Insufficient security guarantee:

- Cannot resist runtime errors.



Inefficient runtime:

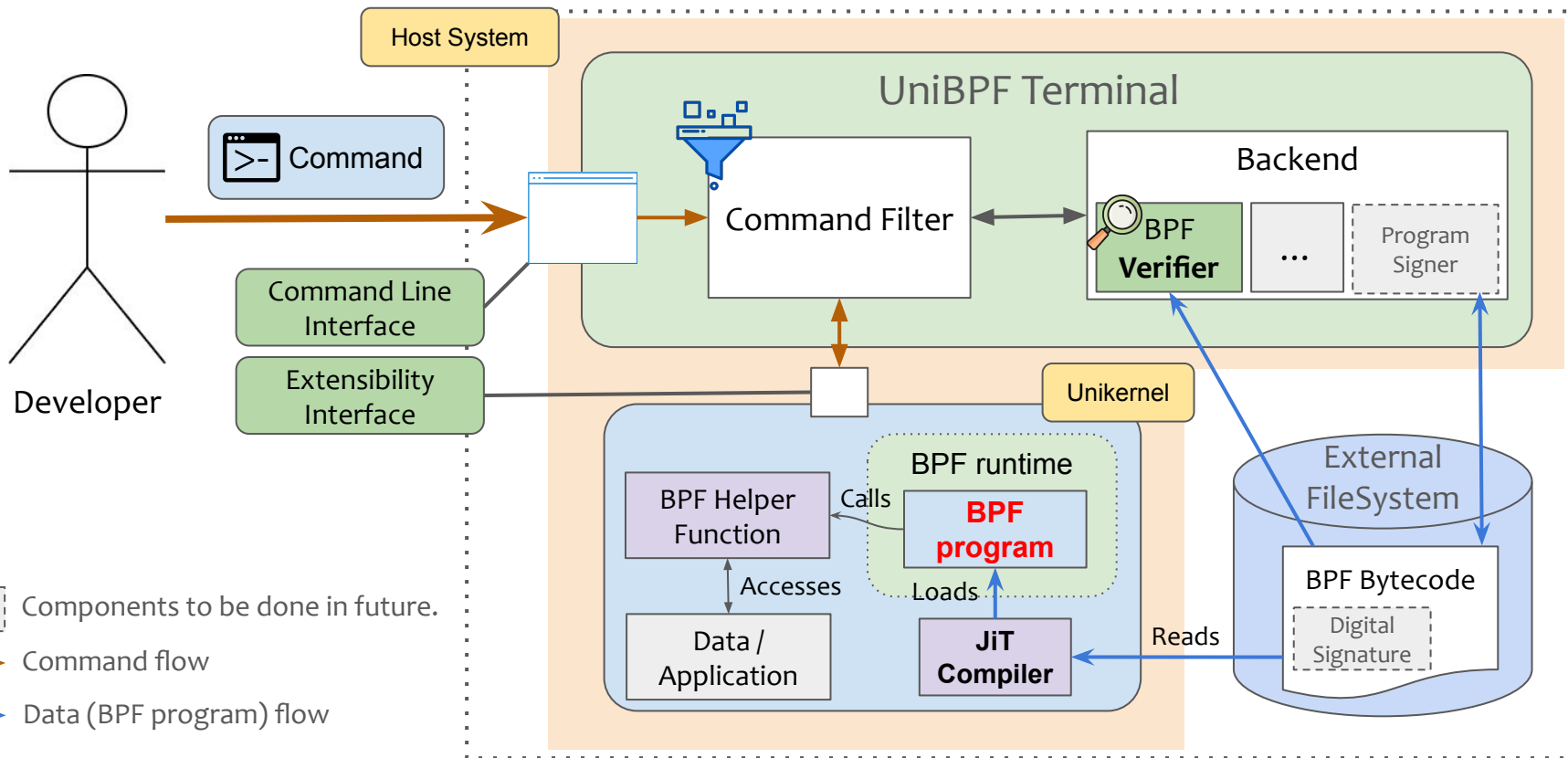
- Our work: $\leq 600\%$ **slowdown** in instruction level v.s. JIT compiled



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

- Design Goal
 - **Security:** Improving the security of the Unikernel's BPF runtime system.
 - **Sustainable Design:** Easy to use, easy to maintain.
 - **Performance:** Acceptable overhead and improve BPF runtime efficiency.

System Overview



Background: extended Berkeley Packet Filter (eBPF)

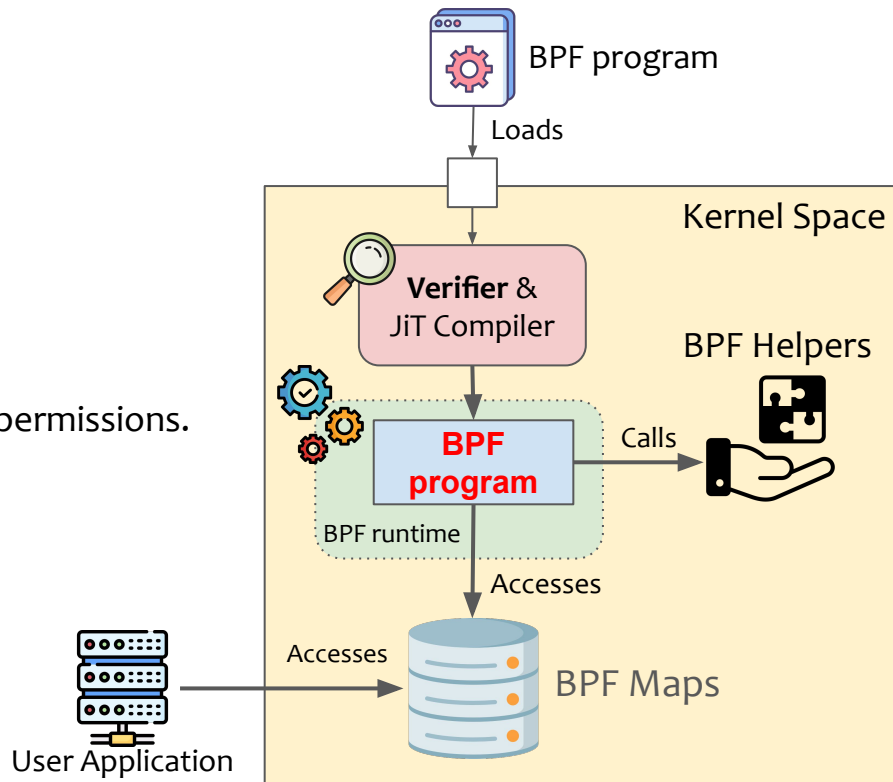
Lightweight in-kernel language VM.

BPF Runtime is **not safe** unless:

- **Isolated** – i.e., sandboxed.
- However, strong isolation makes it unusable
 - Maps (kv-store).
 - Helper functions.
 - Program Types: Runtime context & helper permissions.

Verifiable:

- Verifier ensures program validity:
 - Detects sandbox escalation.
 - Forbid undefined behaviors.

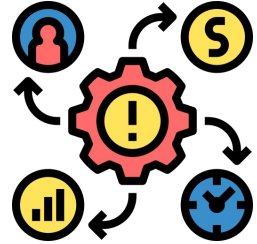


Outline



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

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**.

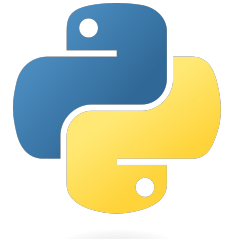
```
1 __attribute__((section("executable"), used))
2 __u64 hash(uk_bpf_type_executable_t* context) {
3
4     __u64 sum = 0;
5
6     for(int index = 0; index < 256; index++) {
7         char* input = context->data + index;
8         if(input >= context->data_end) {
9             break;
10        }
11
12        char to_add = *input;
13
14        if(to_add >= 'A' && to_add <= 'Z') {
15            to_add += 'A' - 'a';
16        } else if(to_add >= '0' && to_add <= '9') {
17            to_add -= '0';
18        }
19
20        sum += to_add;
21    }
22
23    return sum;
24 }
```



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.



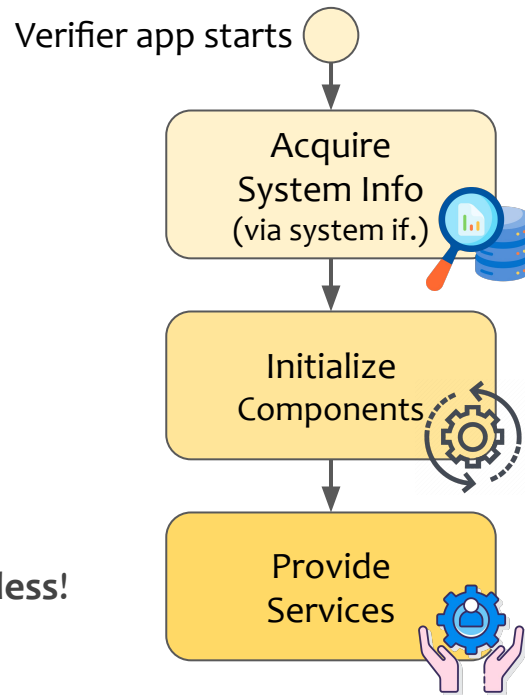
Put BPF Verifiers on the host system utilizing the host system's runtime environment

③ Customizability Impedes Building a Unified Solution



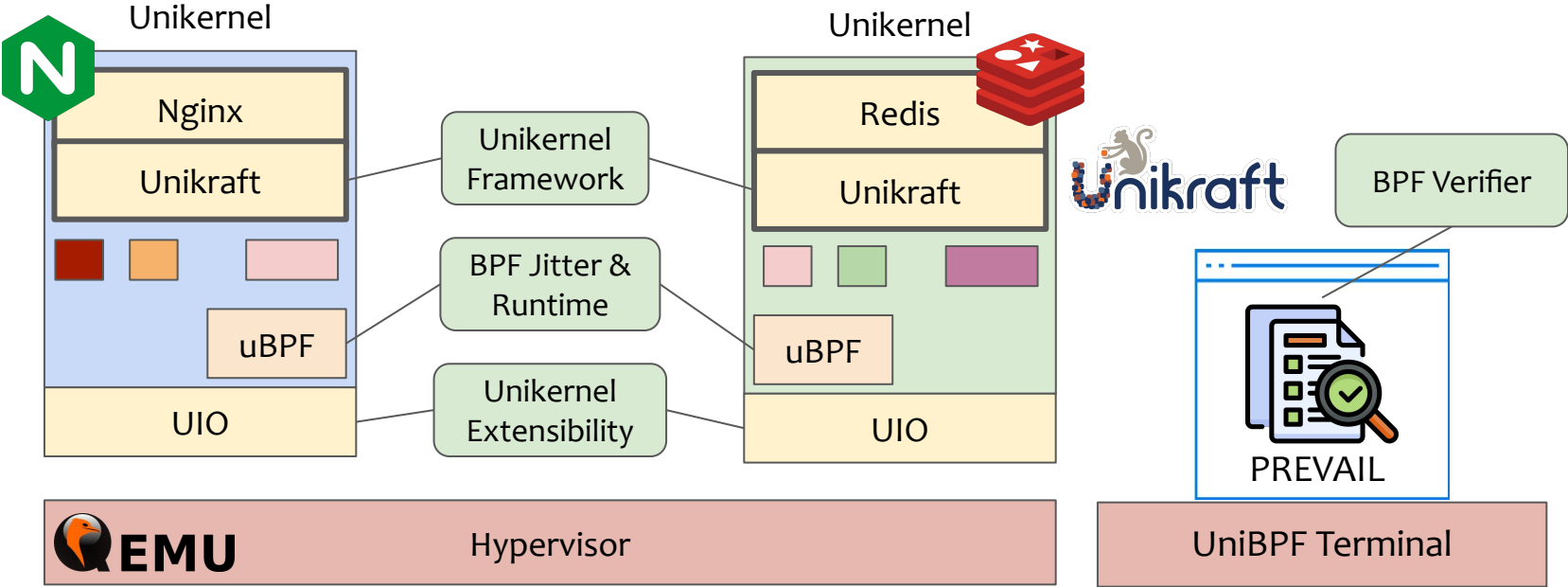
Our Goal: Maintain customizability for BPF runtime

- 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 - Security

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 overall provides more solid security promises

*) OOB: Out of bound memory access

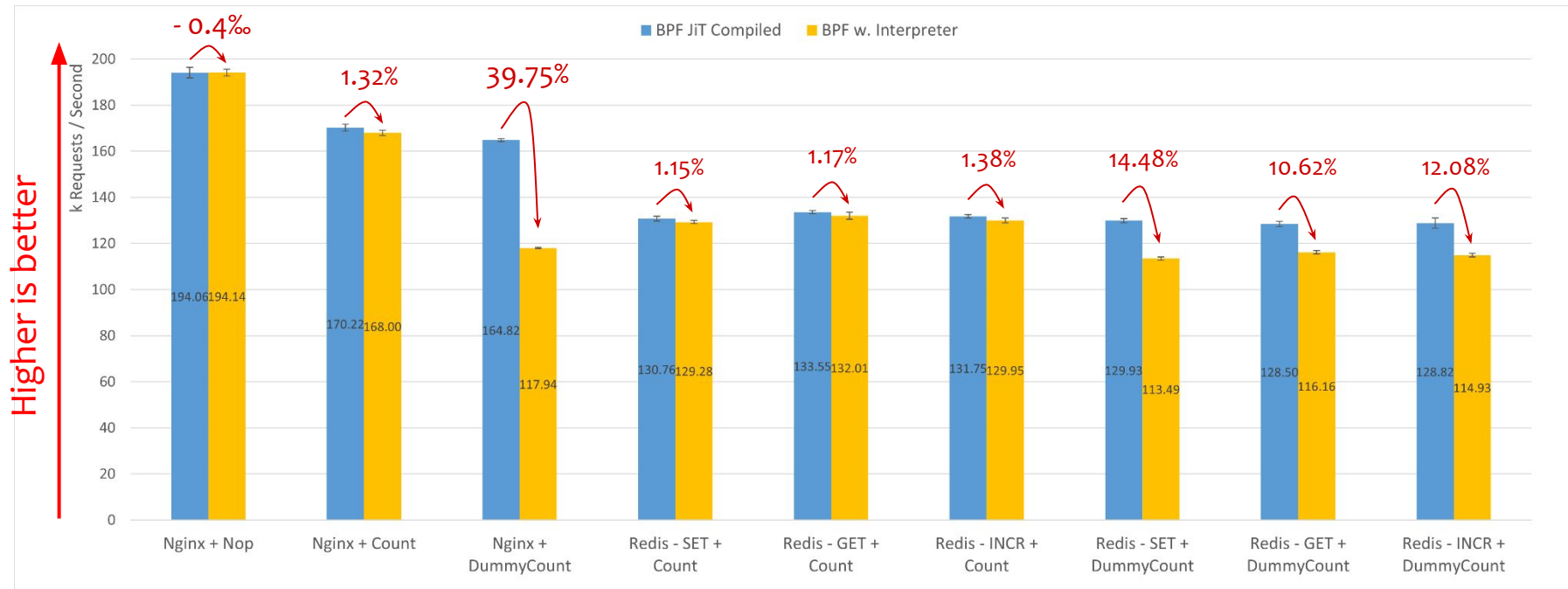
	Instructions	Verification Time Overhead*	Verification Memory Overhead*	JiT Time Overhead
Nop	2	8.82 ms	3328 kb	9.74 ms
Hash	26	12.05 ms (7.43 instr./ms)	4096 kb	9.79 ms
Adds	1002	43.60 ms (28.75 instr./ms)	5056 kb	9.85 ms

 : 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

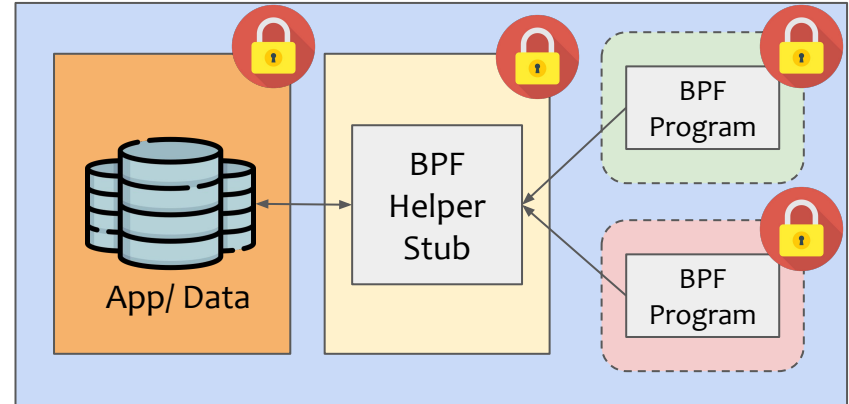
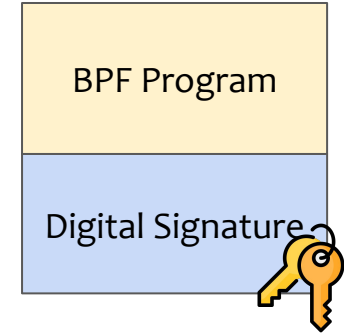
Evaluation - BPF Kernel Tracing Nginx and Redis



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

Further Ideas

- BPF program as configurations.
- Support verification with BPF maps.
- Ensure verification integrity with **digital signature**.
- Secure verification process from malicious cloud provider: **Confidential VM**.
- More robust BPF runtime isolation:
 - Intel **MPK**.
 - BPF helper function stub.



Conclusion

- UniBPF provides **more secure** 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/>