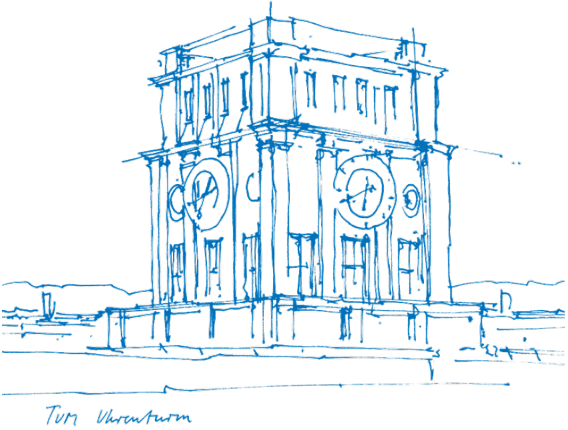
Distributed Systems and Operating Systems
Department of Computer Science
Technical University of Munich

Serverless computing in unikernels using eBPF code injection

Author: Milen Vitanov
Supervisor: Prof. Dr. Pramod Bhatotia
Advisors: Ilya Meignan–Masson
Masanori Misono



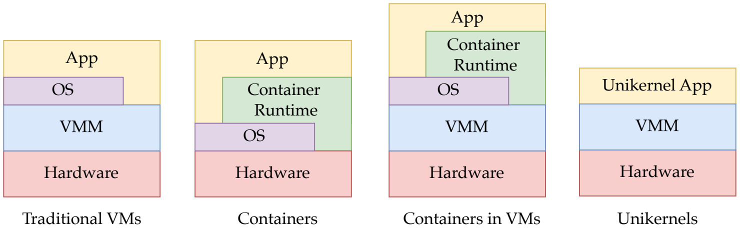
01.06.2024 - 01.12.2024

Outline

1. Background & Motivation
2. Design
3. Implementation Details
4. Evaluation
5. Conclusion & Future Work

Background

Virtualization Technologies



Serverless

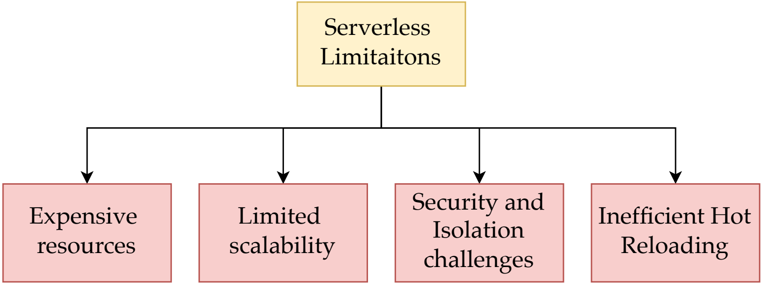
- Event-Driven Execution
- No Server Management, Pay-Per-Use Pricing
- Automatic Scaling

- Data Processing: **MapReduce**

eBPF in Unikernels

- Dynamic Functionality
- Performance Monitoring
- Enhanced Security

Motivation
Research Gap



- General-purpose OS

- Context Switches
- Initialization Time

- Lack of isolation
- Shared resources

- Secure code reloading

Motivation
Problem

How can we build a performant serverless data processing framework which allocates only resources that it needs?

Existing issues

- slow initialization time
- a lot of resources allocated for the OS
- limited isolation of distributed workers

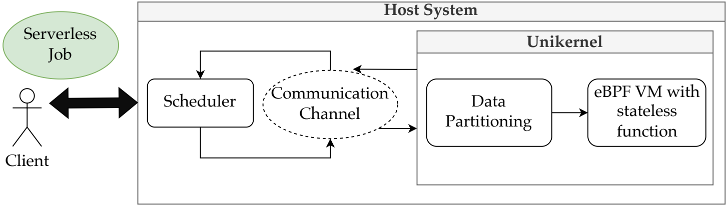
Motivation
Overview

Proposal

Design a new serverless data processing method using unikernels as workers, equipped with eBPF dynamic code injection to allow for any serverless use case.

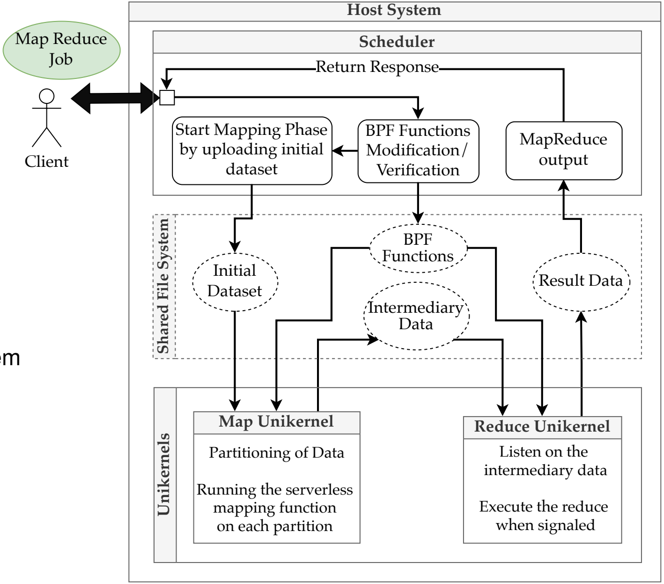
Our framework should:

- use resources efficiently
- be performant
- offer high security & isolation
- be able to inject code at runtime
- be flexible and usable.



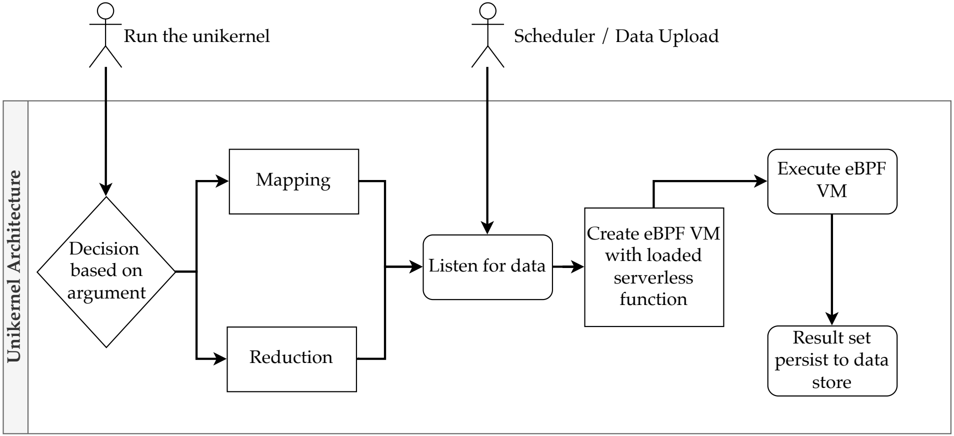
Outline

Design
Components



- Scheduler
- Shared File System
- Unikernels

Design
Unikernels



Design
Challenges

- **Challenge I**: Integrating unikernels and eBPF into the data processing algorithm
  - Work with concrete addresses, simple and small stack memory usage
  - Predefined BPF hooks according to the execution model
  - File System Synchronization Level
- **Challenge II**: Defining the BPF Helper Functions
  - Basic read/write operations
  - Generic communication data structure
  - Generic Helpers API
- **Challenge III**: Complex BPF Verification
  - Decoupled Verification using a robust external verifier

Challenge I
Integrating unikernels and eBPF into the data processing algorithm

- uBPF Integration - porting of an existing library
- Fixed-role structure of the unikernel workers
  - More structured and predictable system
- Mapper Phase Hook
- Synchronization of workers interaction
  - direct and efficient data flow
  - low latency
- Reducer Phase Hook

Challenge II
Defining the BPF Helper Functions

BPF Limitations

- No pointer arithmetics
- Limited stack size
- No floating point numbers
- No dynamic data structures
- No unbounded loops
- Limited/Non-existent Debugging

Data Structure Helpers

- Read/Modify the transformed data
- Traverse the data
- Allocate/deallocate the structure

Data Helpers

- Read/Modify the incoming dataset
- Traverse the dataset

Debugging Helpers

- Allow for printing messages/numbers
- Allow for printing the transformed data

Challenge II

Data Structure Helpers API ⇒ bpf_hashtable_...

#	Helper Name	Description
1	bpf_hashtable_initialize	Initialize new hashtablein host
2	bpf_hashtable_free	Deallocate hashtable
3	bpf_hashtable_get_value_address	Retrieve value address linked to a key
4	bpf_hashtable_get_map_key_value_address	Retrieve value address linked to a key, which is a value in another map
5	bpf_hashtable_get_data_key_value_address	Retrieve value address linked to a key, which is a substring from data at offset
6	bpf_hashtable_insert_value	Insert a value for a key
7	bpf_hashtable_insert_map_key_value	Insert a value for a key, which is a value to another map key
8	bpf_hashtable_insert_data_key_value	Insert a value for a key, which is a substring from data at offset
9	bpf_hashtable_get_size	Retrieve hashtable bucket size
10	bpf_hashtable_get_first_node	Retrieve first node address at bucket index
11	bpf_hashtable_get_next_node	Retrieve next linked node to a given node
12	bpf_hashtable_get_node_value_size	Retrieve the value size of a node
13	bpf_hashtable_get_node_value	Retrieve the value of a node
14	bpf_hashtable_get_node_key	Retrieve the key of a node
15	bpf_hashtable_insert_hashtable	Inserts a map as a value inside another map; key is a string
16	bpf_hashtable_insert_map_key_hashtable	Inserts a map as a value in another map; key is linked val to key is another map
17	bpf_hashtable_insert_data_key_hashtable	Inserts a map as a value in another map; key linked to a key, which is a substring from data at offset
...27	..._node_address, ..._int	simplified functions when working directly with integers or nodes

Challenge II

Data Helpers API ⇒ `bpf_data_...`

#	Helper Name	Description
1	`bpf_data_get_next_index_delimiter`	Retrieve the next index pointing to a matching delimiter in the data chunk
2	`bpf_data_get_next_index_map_word`	Retrieve the next index pointing to a word linked to a map key
3	`bpf_data_update_index_content`	Update the data at index with a new byte content

Challenge II

Debugging Helpers API ⇒ `bpf_print_...`

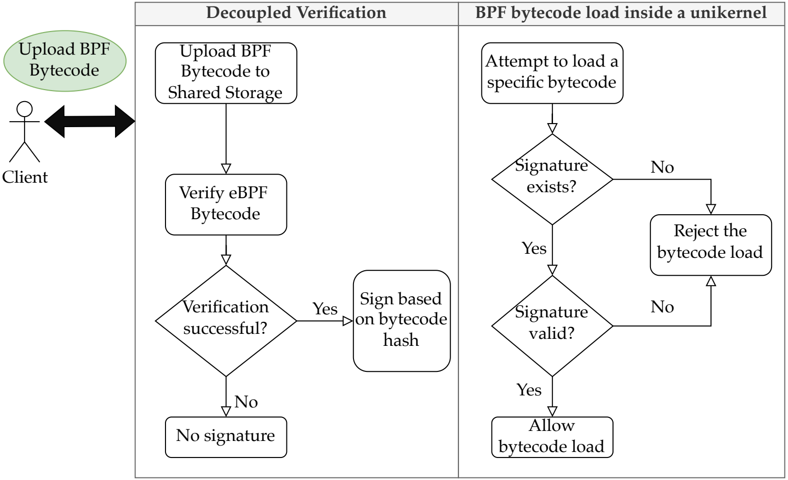
#	Helper Name	Description
1	`bpf_print_text`	Print a message up to a specified size
2	`bpf_print_integer`	Prints a 64-bit integer
3	`bpf_print_hashtable`	Prints a table content

Challenge III
BPF Verification

Decoupled Verification [2]

- Reduced complexity
  - Unikernel is kept minimal and optimized for its task
- Flexibility
  - More powerful system can run the verification for resource-intensive analysis
- Scalability
  - In multi-node environment, single centralized verifier can service multiple unikernels reducing duplication

Challenge III
Verification Design

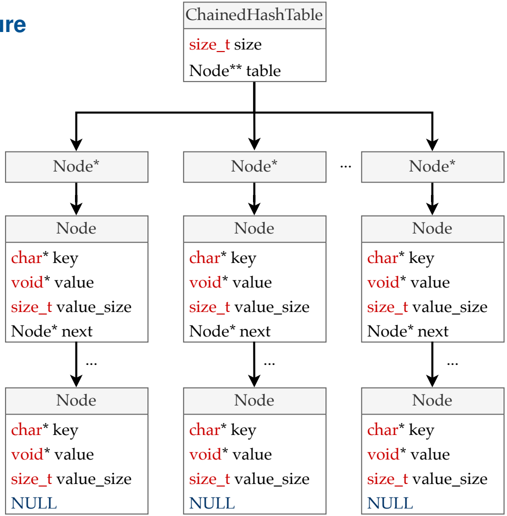


Outline

Implementation Details
External projects

- Unikernels: Unikraft [7]
  - easy to configure, optimized initialization time; modular library approach
- BPF library: uBPF [9]
  - there is already a porting unikraft library done by V. Hendrychová [6]
- BPF Verification: PREVAIL [10]
  - strict, robust and already used is operating systems like Windows
- Data processing algorithm: MapReduce [3]
  - fundamental data processing execution model at the core of the majority of industry-wide solutions
- Virtual Emulator: QEMU [4]
  - popular, with extensive documentation and appropriate to simulate different VM scenarios
- Shared File System: 9pfs [5]
  - lightweight, suitable for minimalist environments
- Cryptography: OpenSSL [8]

Implementation Details
Communication Data Structure



Implementation Details
BPF Program Context

struct Context
uint64_t data_address
uint64_t data_size
uint64_t data_structure_address

Pointer cast to `uint64_t`

```
char* line_ptr = (char*)malloc(DYNAMIC_SIZE * sizeof(char));
context->data_address = (uint64_t)(uintptr_t) line_ptr;
```

`uint64_t` cast to a pointer

```
char* line_ptr = (char*)(uintptr_t) context->data_address;
```

Implementation Details
PREVAIL Custom Platform

eBPF Helper Prototype in PREVAIL

```
static const struct EbpfHelperPrototype bpf_print_text_proto = {
    .name = "print_text",
    .return_type = EBPF_RETURN_TYPE_INTEGER_OR_NO_RETURN_IF_SUCCEED,
    .argument_type = {
            EBPF_ARGUMENT_TYPE_PTR_TO_READABLE_MEM,
            EBPF_ARGUMENT_TYPE_CONST_SIZE
    },
};
```

PREVAIL Context Descriptor

```
constexpr ebpf_context_descriptor_t bpf_prog_descr = {
    .size = 256,
    .data = 0,
    .end = 255,
    .meta = -1
};
```

Outline

Evaluation

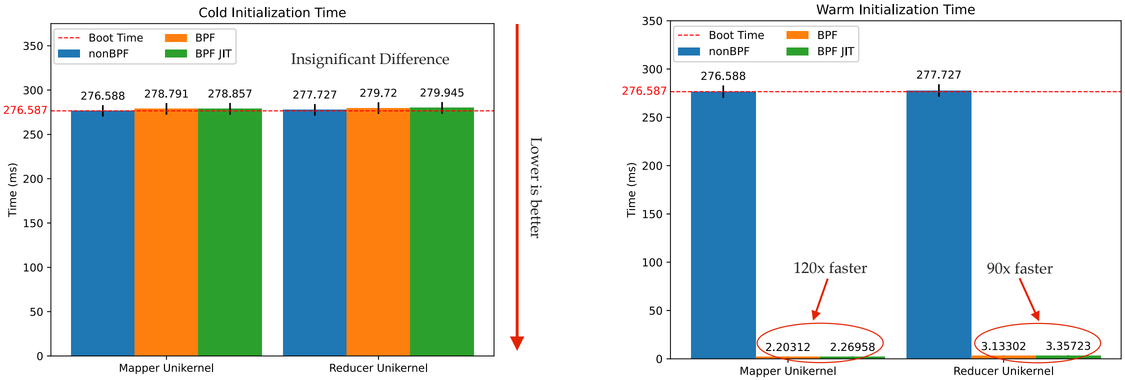
Qualitative Analysis

- Development Complexity
- Debugging
- Learning Curve
- Code Maintenance

Quantitative Evaluation

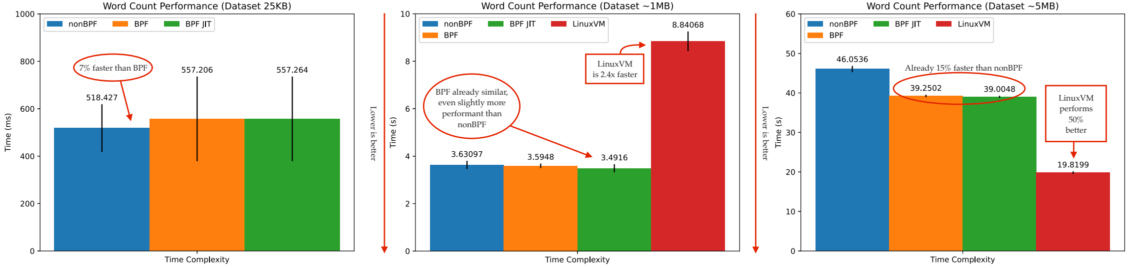
- Initialization Time
- Performance
- Memory Footprint
- CPU-Intensive BPF Functions
- Flexibility

Evaluation
Initialization Time

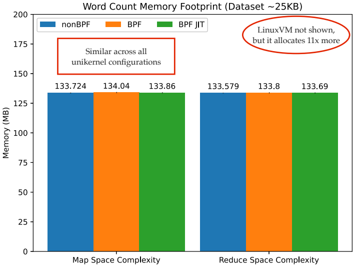


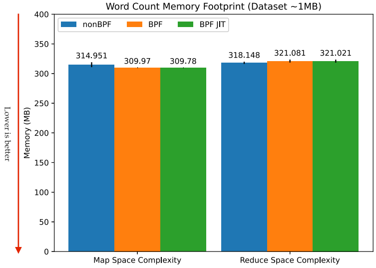
Unique Word Count

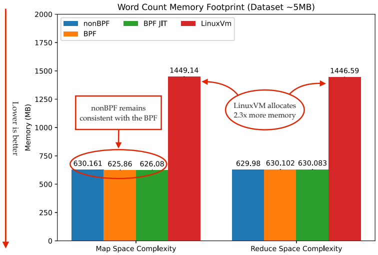
Evaluation
Performance



Evaluation
Memory Footprint





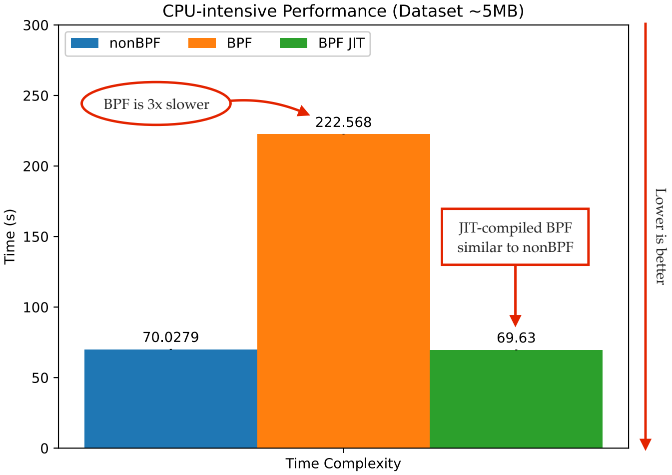


Evaluation
CPU-Intensive BPF Functions

- Mapper: Fibonacci number calculation ($N = 50000$)
- Reducer: π calculation using the Leibniz Series ($I = 1000000000$, $S = 100000000$)

	Average across 10 runs
Mapper Unikernel	133.877MB
Reducer Unikernel	133.855MB

Memory Footprint



Evaluation
Flexibility: Search Engine

BPF Mapper Function

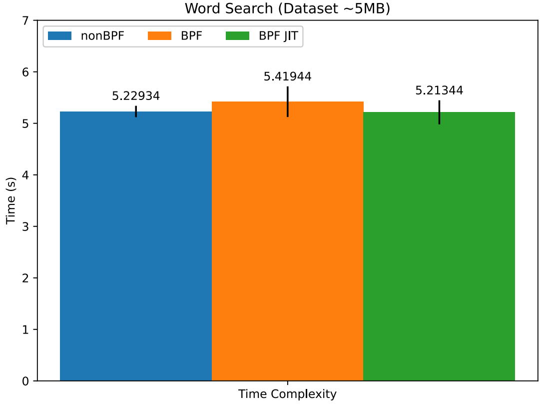
```
for (i : data_size) {
    int token_start = i;
    int match_idx = bpf_data_get_next_index_map_word(data,
            data_size, ds_address, MAP_SEARCH_KEY, i);

    if (match_idx = data_size) {  // no match
        break;
    }

    int value = bpf_hashtable_get_map_key_int(
            ds_address, MAP_DOC_ID_KEY);

    bpf_hashtable_insert_map_key_int(
            ds_address, MAP_DOC_ID_KEY, ++value);

    i = match_idx;
}
```



nonBPF again slightly slower

Evaluation
Flexibility: Inverted Index

Transformed Data

$$word_1 -> [(doc\_id_1, occurrences), ...(doc\_id_n, occurrences)]$$

$$word_2 -> [(doc\_id_2, occurrences), ...(doc\_id_k, occurrences)]$$

$$...$$

$$word_m -> [...]$$

Outline

Conclusion

Serverless data processing framework built on unikernels with the ability to dynamically inject code

- Better initialization time
- Faster performance with bigger datasets
- Efficient memory footprint
- Capable of CPU-intensive tasks
- Lagging behind Linux due to missing unikernel internal optimizations
- Flexible Helpers API

Bibliography I

[1] Mihnea Firoiu Alexander Jung. *Unikernel Concepts*. URL: https://unikraft.org/docs/concepts.

[2] Milo Craun, Adam Oswald, and Dan Williams. "Enabling eBPF on Embedded Systems Through Decoupled Verification". In: *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, 63–69. ISBN: 9798400702938. DOI: 10.1145/3609021.3609299. URL: https://doi.org/10.1145/3609021.3609299.

[3] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[4] The QEMU Project Developers. *QEMU Docs*. URL: https://www.qemu.org/docs/master/.

[5] Plan 9 Foundation. *Plan 9 from Bell Labs*. URL: https://9p.io/plan9/.

[6] Vanda Hendrychová. *ubpf for Unikraft*. Accessed: 2024-06-15. 2023. URL: https://github.com/vandah/uk_ubpf.

Bibliography II

[7] Simon Kuenzer et al. “Unikraft: fast, specialized unikernels the easy way”. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys ’21. Online Event, United Kingdom: Association for Computing Machinery, 2021, 376–394. ISBN: 9781450383349. DOI: 10.1145/3447786.3456248. URL: https://doi.org/10.1145/3447786.3456248.

[8] 2024 OpenSSL. *OpenSSL*. URL: https://www.openssl.org/.

[9] Iovisor Team. *uBPF: Userspace eBPF VM*. Accessed: 2024-10-15. 2024. URL: https://github.com/charlespwd/project-title.

[10] PREVAIL Team. *PREVAIL - A new eBPF verifier*. Accessed: 2024-11-15. 2024. URL: https://github.com/vbpf/ebpf-verifier.