Comparing Security in eBPF and WebAssembly

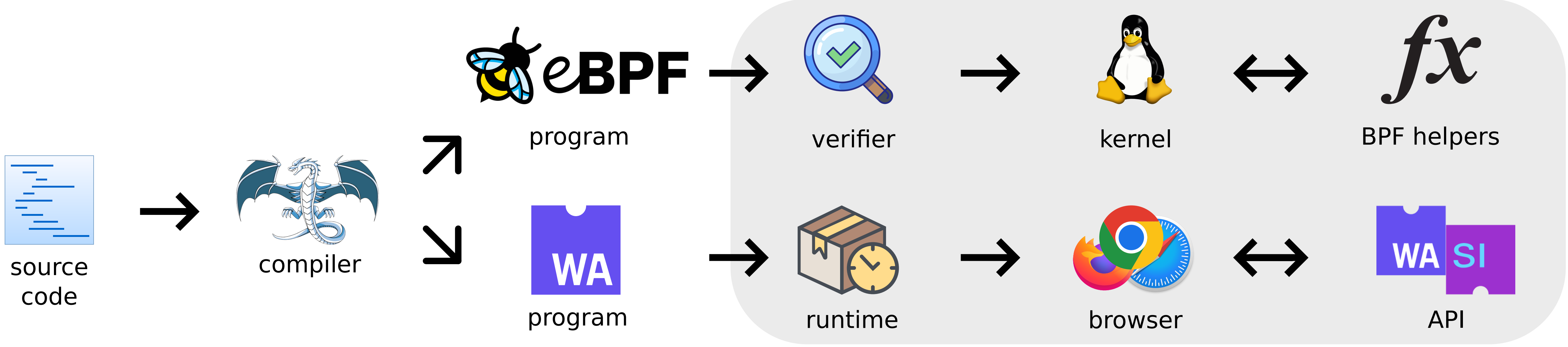
Jules DEJAEGHERE ▸ Bolaji GBADAMOSI ◆ Tobias PULLS ◆ Florentin ROCHET ▸

▸ University of Namur ◆ Karlstad University

1st Workshop on eBPF and Kernel Extensions

September 10, 2023, New York

Overview: lifecycle of eBPF and Wasm programs

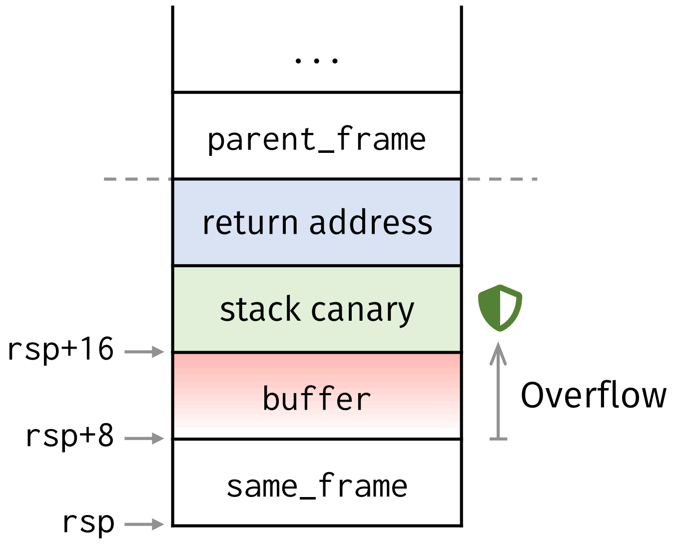


Possible lifecycle for eBPF and Wasm programs

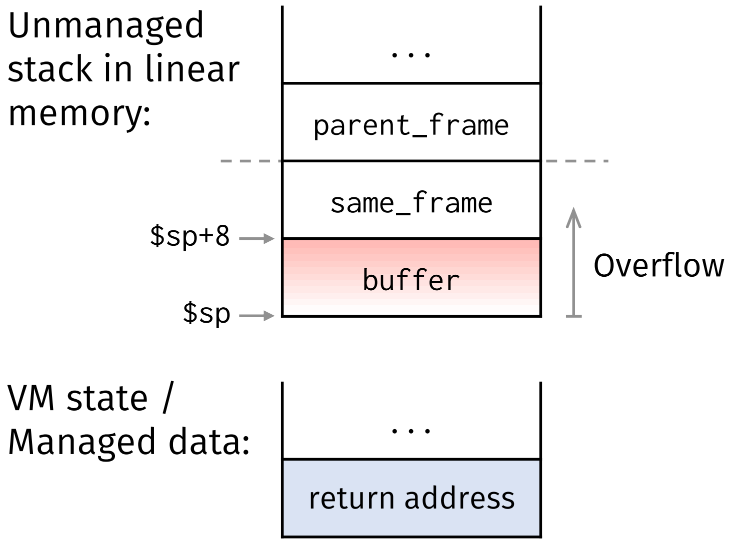
WebAssembly: selected key points

- Binary instruction format
- **Managed stack & linear memory**
- Bounded memory
- Web first but supports non-web embeddings
- **Checked indirect function calls**
- Default to no host access
- **1:1 mapping between binary ⇔ text format**

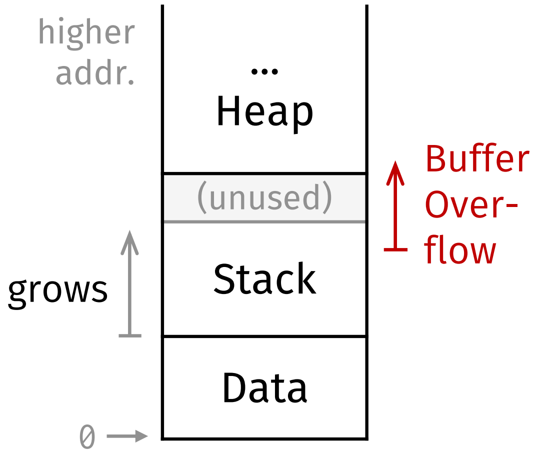
WebAssembly: managed stack & linear memory



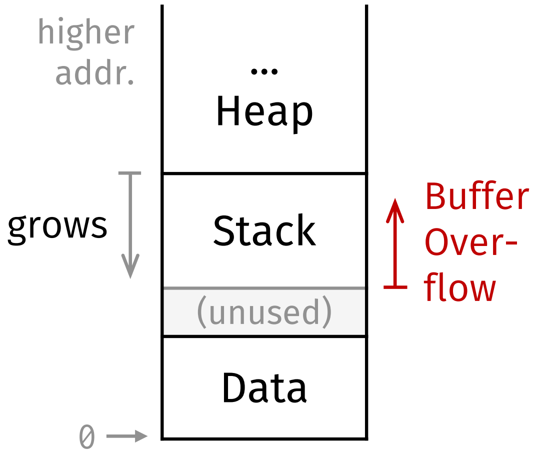
Stack layout on x86-64 with canaries and reordering



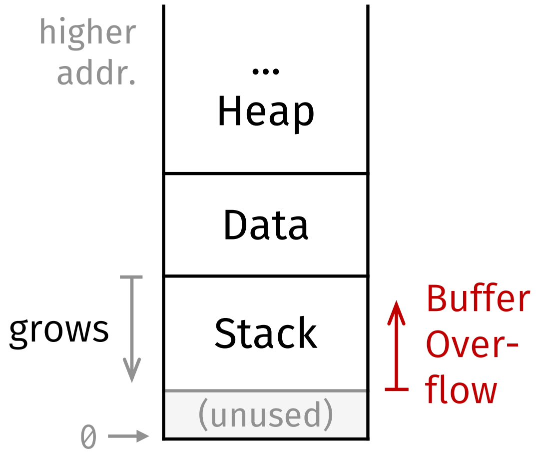
Linear memory and VM state in WebAssembly



`emcc 1.39.7` (*fastcomp* backend, deprecated)



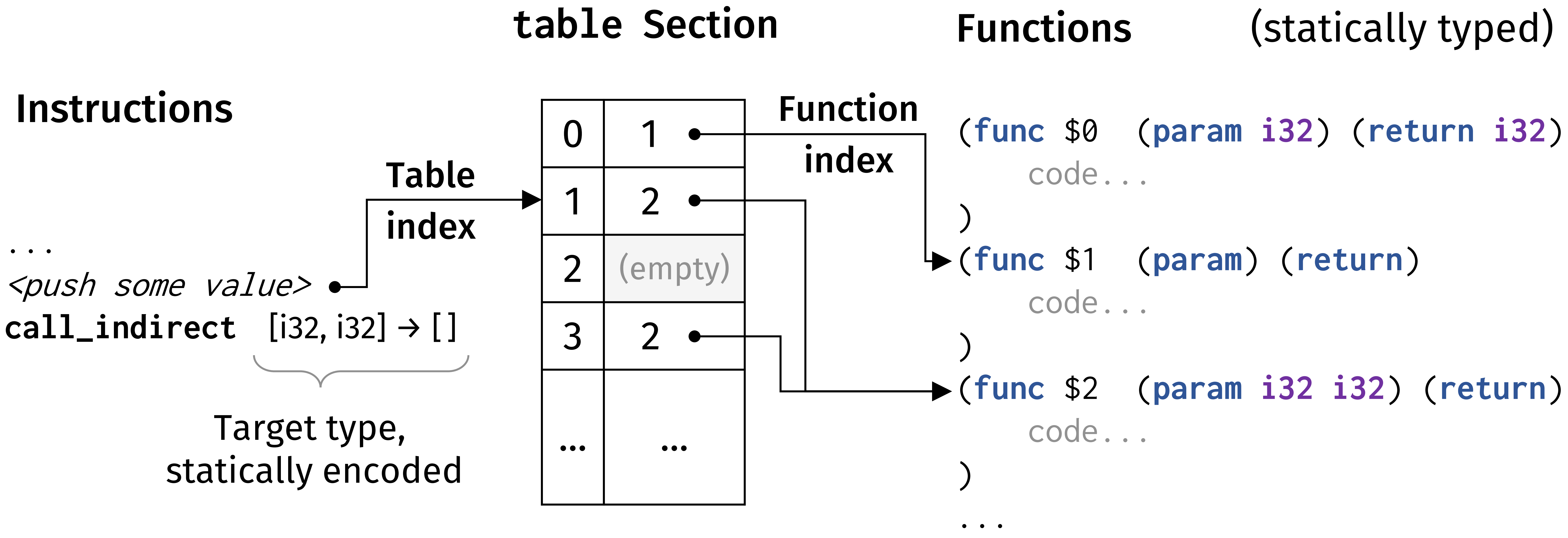
`emcc 1.39.7` (*upstream* backend), `clang 9` (WASI)



`clang 9` (WASI with `stack-first`), `rustc 1.41` (WASI)

Illustrations from Lehmann, D. et al. [1]

WebAssembly: checked indirect function calls



Indirect function calls via the table section

Illustration from Lehmann, D. et al. [1]

WebAssembly: binary ⇔ text

```rust
#[no_mangle]
pub extern "C" fn add(left: i32, right: i32) -> i32 {
    left + right
}
```

```
$ rustc lib.rs --target wasm32-wasi --crate-type cdylib -C opt-level=3
$ wasm2wat lib.wasm
```

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (func $add (type 0) (param i32 i32) (result i32)
    local.get 1
    local.get 0
    i32.add)
  (table (;0;) 1 1 funcref)
  (memory (;0;) 16)
  (global $__stack_pointer (mut i32) (i32.const 1048576))
  (global (;1;) i32 (i32.const 1048576))
  (global (;2;) i32 (i32.const 1048576))
  (export "memory" (memory 0))
  (export "add" (func $add))
  (export "__data_end" (global 1))
  (export "__heap_base" (global 2)))
```

Rust function compiled to a WebAssembly module in textual format

Comparing eBPF and WebAssembly

vs

Threat model

The verifier acts as the gatekeeper to ensure kernel safety

Untrusted code can run without compromising the host

Memory safety

- Few limitations on what programmers can write
- Verifier ensures safety
- No proof, then no execution

- Limited set of constructs
- Grammatically correct, then execution allowed
- Runtime checks

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
SEC("xdp")
int buffer(void *ctx) {
  int a[3];
  int i;
  for (i = 0; i < 100; i++) {
    bpf_printk("%d ", a[i]);
  }
  return 0;
}
char LICENSE[] SEC("license") = '
```

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int a[3];
  int i;
  for(i = 0; i < 100; i++) {
    printf("%d ", a[i]);
  }
    return 0;
}
```

Code will not run

Code will run

Control flow integrity

- CFI enforced by the verifier
- Flagging of programs violating CFI
- Verifier ensures termination

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
char _license[] SEC("license") =
int a;
SEC("socket")
int prog(void *ctx){
  while (1) {
    a++;
  }
  return 0;
}
```

Code will not run

- CFI achieved via semantics
- Jump only to the beginning of valid constructs
- Indirect function calls prevent call redirection

```c
int a;
int main() {
  while (1) {
    a++;
  }
  return 0;
}
```

Code will run (forever)

API access

- Many helper functions available by default
- Each program type can only call a subset of the helper functions
- Access to helper functions is restricted if unprivileged BPF is enabled

- Default to no host access
- API implementation is provided by the host
- Standardized: e.g. WebAssembly System Interface

Side-channels

- Constant blinding to avoid code as constant and JIT spraying
- Retpoline when tail calls cannot be converted to direct calls
- Impossible path verification

- Out of scope for the language, in scope for the runtime
- Bound checking when accessing function table (e.g. `call_indirect`)
- No bound verification for linear memory by default (relying on page fault), can be enabled in some settings

Conclusion

eBPF

- Checks ahead of the execution
- Does not execute if policy violation is found
- Code is trusted but the code is not trustworthy
- Access to many kernel-provided helpers, by default

WA

- Checks at runtime
- Traps when policy violation occurs
- Code is untrusted
- No access to host resources, unless explicitly granted

Takeaways

What are the performance impacts of eBPF and WebAssembly?

Is one approach more efficient than the other?

What can we learn from both technologies?

How could we measure and captures the differences?

Comparing Security in eBPF and WebAssembly

Jules Dejaeghere [▸] Bolaji Gbadamosi [◆] Tobias Pulls [◆] Florentin Rochet [▸]

[▸] University of Namur [◆] Karlstad University

1st Workshop on eBPF and Kernel Extensions

September 10, 2023, New York

This research was partially funded by the CyberExcellence project of the Public Service of Wallonia (SPW Recherche), convention No. 2110186, and Red Hat Research in the Security and safety of Linux systems in a BPF-powered hybrid user space/kernel world project.

References

[1] Lehmann, D., Kinder, J. and Pradel, M. 2020. Everything old is new again: Binary security of WebAssembly. *29th USENIX security symposium (USENIX security 20)* (Aug. 2020), 217–234.

This presentation has been designed using images from Freepik - Flaticon.com.

Meltdown and Spectre icons are from Meltdown and Spectre website.