Value Analysis for eBPF

Simon

October 1, 2025

Abstract

This report presents a value analysis for a subset of the eBPF language, called micro-eBPF. The analysis is based on the theory of abstract interpretation, which uses intervals to approximate the possible values of registers and memory cells at each program state.

Contents

1 Introduction

The SFI policy has two main stipulates. First, any memory access must lie in either the interval of $[r1, r1 + r2)$ or $[r10, r10 + 512]$. Second, all branches must be to the code region where the eBPF code is loaded.

The approach for solving the two varies significantly. The later is something which can be checked statically before program execution. With a single pass over the program, we could check every jump instruction and guarantee that it jumps to somewhere else in the program. For this I decided that the Call instruction is an instruction which violates the SFI policy, as we technically would execute code outside the current programs code.

However, the former is not something which we can conclude at by a simple check of the program, as we do not know the initial values of the registers. For this we could use a Inlined Reference Monitor Rewriter as mentioned in [1]. However, as our memory region depends on the registers r1, r2 and r10, then it seems a logical addition to the SFI policy, that any program which updates the aforementioned registers violate the SFI policy. Otherwise, any program could simply overwrite the registers and they would have arbitrary access to memory. This is another prerequisite (in the same manner as the jump instruction), and we can check this by a single pass over the program.

For this project I theorized about two primary approaches, which are presented in the following sections.

1.1 The naive approach

```
1 // Example instruction
2 mem(r1 + 12) := r2
3
4 // Rewritten program
5 r11 := r1 + 12
6 r9 := r10 + 511
7 if (r11 <= r10) goto next
8 if (r11 > r9) goto next
9 mem(r11) := r2
10 goto end
11 next:
12   r9 := r1 + r2
13   if (r11 <= r1) goto error
14   if (r11 > r9) goto error
15   mem(r11) := r2
16 end:
17   Exit
```

Listing 1: Example program

The naive approach to the SFI enforcement of data regions would be to add conditional jumps whenever we have a memory load/store. In Listing 1a program following the syntax defined in [1] is presented on how this would work.

This SFI enforcements work on conditional jumps, 4 in total to be exact. First, we check if we are in the memory region of $[r10, r10 + 512)$, if we are inside this region, then we fall through and can update the mem(r11) region. However, if we are not within the region, then we check for $[r1, r1 + r2)$ to see if we are in this memory region. Further, we also impose two new requirements on registers. As we are using the registers r9 and r11 for computations, then we would overwrite any data the user has in those registers. As such, when checking the prerequisites we would also have to check for the use of these two registers.

However, as this approach uses more conditional jumps than I would like I decided to move on from it and instead use the next approach.

1.2 Address masking

Address masking as presented in [1] would require that we impose restrictions on r1, r2 and r10. Namely we would have to impose that r2 is some power of two, and we have to impose that r1 fits in the upper bits of r2, i.e given $r2 - 1 = 0xFFF$, then r1 would have to be $r1 = ...000$, where ... indicate that we can have any values come before, and we would need the same requirement for r10 but in regards to $t11$. If this held, then we would be able to use $r2 - 1$ as the address mask, and use a binary OR to then compute the correct region.

```
1 // Example instruction
2 mem(r1 + 12) := r2
3
4 // Rewritten program
5 r2 := r2 - 1
```

```
 6 r9 := r1 + 12
 7 r11 := r9
 8 r9 := r9 - r10
 9 r9 := r9 & 511
10 r9 := r9 + r10
11 if (r9 = r11) goto succ
12 r11 := r11 - r1
13 r11 := r11 & r2
14 r11 := r11 + r1
15 mem(r11) := r2
16 goto end
17 succ:
18   mem(r9) := r2
19 end:
20   Exit
```

Listing 2: Example program

Although this might have been a more performant solution than the one I have applied, I found that this was too imposing on the data region. As such, I have decided instead only to impose that r2 should be a power of 2. Together with this I require that r9 and r11 are not used as in the naive approach. This allows us to rewrite the previous instruction as seen in Listing 2.

References

[1] Gang Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3):137–198, 2017.