Value Analysis for eBPF

Simon

October 3, 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.

This approach includes an initial statement to every program. This is the statement $r2 := r2 - 1$. As a prerequisite we have that r2 must be a power of 2, and as such this instruction makes it such that r2 now contains a mask. For example, if we had: $0x1000$, then after this statement we would have $0xFFF$.

Next, as in the naive approach, we compute the memory location into a designated register, r9 in this case. Then we copy the value into register r11, before forcing r9 to be in the interval of [r10 + 512). We force this as by:

First, subtract r10 from r9. For this operation we have two cases that are important for us:

- If r9 was already in the aforementioned interval, then subtracting r10 from r9 makes it such that r9 is instead in the interval [0, 512), the following AND operation would then have no affect on r9, and we then add back r10, which leaves us with the original value.
- If instead r9 is outside the initial interval, then this operation produces some arbitrary value x, for which the next AND instruction forces to be in the interval [0, 512). Then we add back r10, leaving us with some different value x', which is in the interval [r10, r10 + 512).

Note for this masking to work, we can see that 512 is already a power of 2 as before, and we subtract 1 (for the same reason as with r2) to get the address mask.

Now we guarantee that r9 has a value in the interval [r10, r10 + 512). However, we do not know if this is because the original value was already within this region or if we forced it within the region. As we saved the original value in r11, we must then do a conditional jump between the two registers, and if we see that the value was not changed, then the read already satisfied SFI policy, and we execute the memory operation with r9. In the case that r9 was changed, then we instead do the same computation for r11, and force the memory read to be in the interval of [r1, r1+r2). This time we do not check if we kept the original value, as we just want it to be within the allowed regions.

2 Implementation

2.1 Prerequisite

```
1 checkPrerequisite :: ((Int, Instruction) -> Bool) -> LabeledProgram ->
      Maybe ()
2 checkPrerequisite p lprog =
3   let res = any p lprog
4    in (if res then Nothing else Just ())
5
6 -- Example use case for checking jump instructions.
7 checkJumpInstructions :: LabeledProgram -> Maybe ()
8 checkJumpInstructions lprog = checkPrerequisite checkInst lprog
9  where
10   checkInst (l, instr) = case instr of
11     JCond _ _ _ off -> checkJump off l
12     Jmp off -> checkJump off l
13     -- Do not allow any calls to extern
14     (Call _) -> True
15     _ -> False
16   checkJump off l =
17     let
18       res = all (\(i', _) -> i' == getLabel off l) lprog
19      in
20       not res
21
22 checkPrerequisites :: LabeledProgram -> Maybe ()
23 checkPrerequisites lprog = do
24   -- Registers 1,2,10 are restricted by definition of the assignment.
       register 9, 11 is used for guards
25   _ <- checkRegisterUse lprog [1, 2, 9, 10, 11]
26   _ <- checkJumpInstructions lprog
27   -- Now that we have a program that is nice, we can just return
28   return ()
```

Listing 3: Cheking prerequisites

The first item of the implementation is checking the prerequisites mentioned in Section 1. For this I defined the helper function `checkPrerequisite`, which can be seen in Listing 3. The function simply checks if the predicate is true for any of the statements in the program. If it is, then we return `Nothing`, otherwise we return `Just ()`.

An example of the use case is the `checkJumpInstructions`. Within this function, we define the predicate `checkInst`. If we have a jump condition, then we have to check if there exists some other label in the labeled program, which starts with the location we are attempting to jump to (the `getLabel` takes the offset and the current label, and calculates the label we would jump to). If the jump is illegal, then res would be False on line 18, so we negate res and return.

With this, we can use the functions with the do notation of the Maybe monad as seen in `checkPrerequisites` in Listing 3. If any of the prerequisites return `Nothing`, then the entire function simply returns `Nothing`.

2.2 SFI Policy Algorithm

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.