Value Analysis for eBPF

Simon

October 5, 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

```
checkPrerequisite :: ((Int, Instruction) -> Bool) -> LabeledProgram ->
    Maybe ()
checkPrerequisite p lprog =
  let res = any p lprog
   in (if res then Nothing else Just ())

-- Example use case for checking jump instructions.
checkJumpInstructions :: LabeledProgram -> Maybe ()
checkJumpInstructions lprog = checkPrerequisite checkInst lprog
 where
  checkInst (l, instr) = case instr of
    JCond _ _ _ off -> checkJump off l
    Jmp off -> checkJump off l
    -- Do not allow any calls to extern
    (Call _) -> True
    _ -> False
  checkJump off l =
    let
      res = all (\(i', _) -> i' == getLabel off l) lprog
     in
      not res

checkPrerequisites :: LabeledProgram -> Maybe ()
checkPrerequisites lprog = do
  -- Registers 1,2,10 are restricted by definition of the assignment.
     register 9, 11 is used for guards
  _ <- checkRegisterUse lprog [1, 2, 9, 10, 11]
  _ <- checkJumpInstructions lprog
  -- 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

```
1  sfiAlgorithm' :: LabeledProgram -> Int -> Maybe LabeledProgram
2  sfiAlgorithm' cur_prog curr =
3    if curr >= length cur_prog
4      then Just cur_prog
5      else
6        let (l, curr') = cur_prog !! curr
7         in case curr' of
8              i@(Store _ dst off _) -> do
9                -- In store we want to guard the destination
10               (newProg, newCurr) <- handleMemloc l dst off i
11               -- After adding the guard, we now replace Reg and off
     with reg 11
12               sfiAlgorithm' newProg (newCurr + 1)
13             i@(Load _ _ src off) -> do
14               -- In load we want to guard the source
15               (newProg, newCurr) <- handleMemloc l src off i
16               sfiAlgorithm' newProg (newCurr + 1)
17             _ -> sfiAlgorithm' cur_prog (curr + 1)
18     where
19     ...
20     handleMemloc :: Label -> Reg -> Maybe MemoryOffset -> Instruction
       -> Maybe (LabeledProgram, Int)
21     handleMemloc l reg off i =
22       -- First create the guard
23       let guard = getGuard l reg off i
24           -- Remove the original possibly offending statement
25           lprog'' = removeLabel l cur_prog
26           -- Increment all labels, and make sure our jumps are correct
27           fixedProg = newLabels (length guard) l lprog''
28           -- Add the guard to the program, and then sort
29           newProg = sortOn fst (fixedProg ++ guard)
30        in Just (newProg, l + length guard - 1)
```

```
31
32      newLabels :: Int -> Label -> LabeledProgram -> LabeledProgram
33      newLabels len l = map (updateLabel l len)
34      updateLabel :: Label -> Int -> (Int, Instruction) -> (Int,
        Instruction)
35      updateLabel l len (l', instr) =
36        -- n is the label the instr wants to jump to (in case of jump)
37        let (n, off) = case instr of
38              JCond _ _ _ code -> (getLabel code l', code)
39              Jmp code -> (getLabel code l', code)
40              _ -> (0, 0) -- dummy, won't be used
41              -- New target for jump instructions
42            newTarget
43              -- If we were jumping over the current label, then we now
        have to
44              -- take the guard into account. (Also irrelevant to use >=,
         as the
45              -- current l witll never be a jump anyhow.)
46              -- Also Also, in the case we are jumping from somewhere
        after, we
47              -- want to update it to instead jump to the guard. This is
        why n ==
48              -- l is included in the n <=.
49              | n <= l && l' >= l = off - (fromIntegral len - 1)
50              | n > l && l' <= l = off + (fromIntegral len - 1)
51              -- Otherwise, the guard changes nothing.
52              | otherwise = off
53            newInstr = case instr of
54              JCond cmp reg regimm _ ->
55                JCond cmp reg regimm (fromIntegral newTarget)
56              Jmp _ ->
57                Jmp (fromIntegral newTarget)
58              i -> i
59         in (if l' >= l then l' + len - 1 else l', newInstr)
```

Listing 4: SFI algorithm

The SFI implementation I used is the one mention in Section 1.2. After we have checked all prerequisites our goal is now to guard all memory accesses to the specified regions.

The general SFI algorithm can be seen in Listing 4. We recursively go through the labeled program and if we ever find a load or store operation (the only two statements involving memory access), then we handle them via `handleMemloc`. This function returns a new program, and the next instruction we have to check. The function follows the general structure:

- First, get the statements that replace the current load or store statement. These include the arithmetic in Section 1.2, and they also replace the current instruction to instead use our designated registers r11 and r9.
- Then we remove the current statement from the original program.
- We then update all labels in the program. Any label that would have come after the original statement has to be incremented by the length of

the guard we are inserting[1]. Now all the labels that are in the guard can not be found in the original program, and so we are able to insert the guard list of statements and sort the labeled program on the Labels to get the new program.

- We do not want our sfi algorithm to run on these new inserted statements, so we update what label should be checked next to be the original label plus the length of the guard minus 1. This gives us the label that is right after the original statement in the original program.

The `newLabels` function does the majority of the work here. It both has to update the current label of all the statement, but it also has to update all jump instructions to jump to the new locations. Importantly, we want every jump that jumped to the original load/store instruction to now jump to the start of our guard, such that we always guarantee that any load adheres to the SFI policy.

3 Discussion

3.1 Combining approaches

In Section 1 I presented the two main approaches I found viable for the task at hand. However, as I worked with the project I started to wonder whether they had to be separated at all. The naive approach would include 4 conditional jumps, but the total number of instructions would be smaller than the one seen in the address masking approach. With the large amount of conditional jumps it seems trivial that the address masking is better, but in the address masking we are still forced to have a single conditional jump, such that we can check the two separate regions. If instead we used a combination of the two, where we might initially check if the data is in the region of $[r1, r1 + r2)$ and then afterward force the value to be between $[r10, r10 + 512)$, then we could do this with two conditional jumps, but fewer instructions than in the full address masking approach. With this approach I would only need to claim a single register for the guard instructions and we would also not require that r2 is some power of 2. Looking back this approach might have been better, but it depends on how much more expensive the conditional branches are than the instructions I use to replace them. With specific knowledge on the branch protector one might have a better way of arguing which of the two approaches is preferred.

3.2 Optimisations

3.2.1 Value analysis

Previously in the SOS course we worked with value analysis, something which I initially thought I would use in this project. The idea would be to do an

[1]This guard length depends on whether or not we have an offset or not. Without an offset, we can save an instruction.

initial run of the value analysis on the program, and then we might be able to save guard instructions in locations where we know we only reach safe memory regions. However, with the current implementation of value analysis this is not possible. Although, we might be able to say that some memory load would be in the interval of $[0, 120]$, we would not know beforehand if that is in the regions of $[r1, r1 + r2)$ and $[r10, r10 + 512)$

For this to work the intervals would have to be in regards to the registers r1, r2 and r10, such that we could tell if any memory load is in the correct region, but this is not possible. For the value analysis we need an ordering of the elements for a vast majority of the computations. For instance, what would be the maximum of 120 and r10? Without information on r10 this is undecidable, and as such value analysis can not help us in this instance.

However there is one place the value analysis could help us. The value analysis could find memory access that are guaranteed to be outside the data regions. Lets say that we know $[r1, r1+r2)$ is some subset of $[0, 0xFFFF]$, and same for the r10 interval. Then if we find that some instruction would attempt to read from memory location $oxFFFFF$, we would know this is not allowed. To satisfy SFI policy, we could either let the program run and address mask the offending instruction instantly (i.e. force it to either the interval $[r1, r1 + r2)$ or $[r10, r10 + 512)$ without the conditional jump from Section 1.2), or we could simply state that we have violated some precondition and refuse to ever run the program.

3.3 Reducing repetition

One approach that could be implemented in reducing the number of repetition. If we know that we have just inserted a guard which checks that some memory region x satisfies our SFI policy, then if we attempt to write to x again, then we would not need to add another set of guard instructions to the second store instruction. While this seems easy in practise, there are some subtleties we have to keep in mind. If we have some statement z, then we would have to check all possible paths to z and determine that x satisfies the SFI policy already. That is, even though the previous instruction might have guaranteed that x satisfies the SFI policy, then we have to make sure there does not exist some jump to statement z for which x could violate the SFI policy. If there is a single case where this is possible, then we would somehow have to either guard this one path, or we would have to insert the general guard as in my implementation.

4 Conclusion

In this project, I have designed and implemented a Software Fault Isolation (SFI) policy for eBPF bytecode. The central goal was to enforce memory and control-flow safety by rewriting eBPF programs to ensure all memory accesses are confined to designated regions - a stack space and a data region - and that all jumps are local to code within the program.

My implementation, developed in Haskell, first performs a prerequisite check on the eBPF program. This static pass verifies that there is no use of reserved registers are not used, and that all control-flow transfers (jumps) are directed within the program's boundaries. Following this validation, the core of my approach involves traversing the program and replacing each memory load and store instruction with a sequence of guard instructions. I settled on an address masking technique, which, despite requiring 'r2' to be a power of two, was chosen over a more naive approach to reduce the overhead of conditional jumps. This technique forces every memory access into a valid region using bitwise operations, with a single conditional jump to select between the stack and data regions.

The discussion highlighted several alternative designs and potential optimizations. While static value analysis proved difficult to apply for proving safety due to the dynamic nature of the memory region registers, it could be used to detect guaranteed out-of-bounds accesses. Another significant optimization would be to eliminate redundant guards by analyzing data flow to see if a memory location has already been verified.

In conclusion, this work successfully demonstrates a practical method for enforcing SFI on eBPF programs. The implemented address masking provides a robust safety guarantee, but with specific constraints on program structure.

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.