Value Analysis for eBPF

Simon

September 22, 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 goal of this project is to implement a value analysis for a subset of the eBPF language. This analysis aims to compute an approximation of the values of registers and memory cells at each program point. The analysis is based on the principles of abstract interpretation, specifically using an interval based abstract domain. This report documents the design choices and implementation details of the analysis. The project builds on the already implemented eBPF-tools[1]. eBPF-tools is able to generate control-flow graphs from the micro-eBPF language. These control-flow graphs are the foundation of the implementation provided within this project.

1.1 Abstract domain

Throughout this project, I have used the infinite intervals to represent the possible values of both registers and memory locations. That is to say, that an abstract value is an interval $[l, u]$ where $l, u \in \{-\infty, \infty\}$. These infinite intervals require the use of widening, followed by narrowing, to produce usable results.

In the micro-eBPF language we have 11 registers (r0-r10) and a memory section Mem, which contains 512 different cells. Throughout the implementation both registers and memory cells were treated in the same manner. That is to say, that a register/memory cell could take on any value in the interval mentioned previously, or they could be some unreachable state denoted by $\perp$. Also, the interval $\{-\infty, \infty\}$ itself is the representation of $\top$, indicating that the register/memory cell can take on any value.

1.2 Modelling program states

The modelling of program states is best described by an example. Say we have some arbitrary node, n, within the control flow graph. Now given some transition t from $n \rightarrow n'$ through some expression exp, then the state at n', namely s', is given by:

$$s' = (\llbracket exp \rrbracket s) \sqcup_{Itv} t' \tag{1}$$

Where t' is any other transition, which also ends in n'. That is to say, that the new state s' at node n' is given by the union of all the states that came before and end in the current node. For this to work properly, we first initialize all states to the neutral element of the interval union, which is $\perp$.

1.3 Abstract interpretation of expressions

In general my implementation follows the abstract interpretation of expressions seen in Miné[2]. However, Miné does not provide a definition for the binary operators. In my case I was not able to get the time to implement the binary operators. As such, if we ever see a binary operator, then we immediately just return $\top$.

1.3.1 Memory lookups

For a memory lookup `Mem[ri]`, the abstract value of the register `ri` determines the possible memory addresses to be read. Let $[l, u]$ denote the abstract interval of `ri`. The set of possible addresses is $\{l, l+1, \ldots, u\}$, meaning the result of the memory lookup is the union of the abstract values at these addresses:

$$\text{Let } R(s) = \bigsqcup_{Itv_{i \in [l,u]}} [\![\texttt{r[i]}]\!]s \tag{2}$$

$$[\![\texttt{Mem[ri]}]\!]s = \bigsqcup_{Itv_{j \in R(s)}} \texttt{Mem[j]} \tag{3}$$

This means, that there could be cases where we are writing to multiple memory cells at the same time, if we are using a register, which at the current state is not a constant value.

1.4 Iteration strategy

In this project the iteration strategy used is the work-set algorithm. This method was used to avoid re-evaluating parts of the program, that were not affected by the previous state. For more details on the specific implementation see Section 2.4.

The work-set algorithm was implemented with widening and narrowing. Widening (∇) is defined as in Mine[2], i.e. we have that:

$$\begin{aligned}[l_1, u_1]\nabla[l_2, u_2] = [&\text{if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \\ &\text{if } u_2 > u_1 \text{ then } \infty \text{ else } u_1]\end{aligned} \tag{4}$$

And Narrowing (Δ) is defined as:

$$\begin{aligned}[l_1, u_1]\Delta[l_2, u_2] = [&\text{if } l_1 = -\infty \text{ then } l_2 \text{ else } l_1, \\ &\text{if } u_1 = \infty \text{ then } u_2 \text{ else } u_1]\end{aligned} \tag{5}$$

To not overdo this, I included a counter for each node in the control flow graph. If we reach the node x times, then we perform one widening step. Afterwards, we perform some y amount of narrowing steps, but to avoid infinitely looping, we limit y to some finite number.

2 Implementation

2.1 Definitions

```
data Bound = NegInf | Val Integer | PosInf deriving (Eq, Ord)
data Interval = Interval Bound Bound deriving (Eq, Ord)

// Bottom M
data BottomM a = Bottom | Value a
  deriving (Eq, Ord)

// IntervalM
type IntervalM = BottomM Interval

type Mem = Array Int IntervalM

data State = State
  { registers :: Array Int IntervalM
  , memory :: Mem
  }
  deriving (Eq)
```

Listing 1: Bound data type

As mentioned I make use of the infinite interval. This is of course not something machines can correctly represent, and as such I use the Bound data type, whose definition can be seen in Listing 1. Bound can thus take up 3 different ”values”. Either it is NegInf($-\infty$), some value or PosInf(∞). Note, that because we define it in the order we have, then NegInf will be seen as smaller than any Val and PosInf will be seen as larger than any Val. This allows us to simply use the default min and max functions on the Bound type.

Along with Bound there is also the definition of the Interval type, which contains two Bound values, the first indicating the lower bound and the second indicating the upper bound.

This now allows me to represent all the interval values except for $\perp$. For $\perp$ instead of simply adding the value to the Interval type, I decided to make use of BottomM. BottomM is defined exactly the same way as the Maybe monad usually used in Haskell. The only difference between the two is that BottomM makes use of Bottom instead of Nothing, and Value x instead of Just x.

This choice was made as the large majority of operations on both intervals and bounds return Bottom, if any of the inputs are Bottom. This is the exact same functionality from the Maybe monad, and saves a lot of repetitive code checking the specific cases.

At last, we have the definition of the State. The state simply contains two arrays. One which maps the registers to some IntervalM and then the same for each of the memory cells. A state is then created for each of the nodes in the control flow graph when initializing the setup.

2.2 Operator implementation

The definitions from Section 2.1 allows me to define functions as the type signature:

```
binaryBound :: Bound -> Bound -> BottomM Bound
unaryBound :: Bound -> BottomM Bound
```

```

// Interval operators
type IntervalM = BottomM Interval
binaryInterval ::  Interval -> Interval -> IntervalM
unaryInterval ::  Interval -> Interval -> IntervalM

// Interval comparisons
binaryInterval :: Interval -> Interval ->
                  BottomM (Interval, Interval)
unaryInterval ::  Interval ->
                  BottomM (Interval, Interval)
```

Examples

```
divBound :: Bound -> Bound -> BottomM Bound
divBound (Val x) (Val y)
  | y == 0 = Bottom
  | otherwise = Value $ Val (x `div` y)
divBound _ PosInf = Value $ Val 0
divBound _ NegInf = Value $ Val 0
divBound NegInf (Val y)
  | y > 0 = Value NegInf
  | y < 0 = Value PosInf
  | y == 0 = Bottom
divBound PosInf (Val y)
  | y > 0 = Value PosInf
  | y < 0 = Value NegInf
  | y == 0 = Bottom
divBound _ _ = Bottom

divInterval :: Interval -> Interval -> IntervalM
divInterval ab@(Interval a b) cd@(Interval c d)
  | Val 1 <= c = do
      ac <- divBound a c
      ad <- divBound a d
      bc <- divBound b c
      bd <- divBound b d
      Value $ Interval (min ac ad) (max bc bd)
  | d <= Val (-1) = do
      bc <- divBound b c
      bd <- divBound b d
      ac <- divBound a c
      ad <- divBound a d
      Value $ Interval (min bc bd) (max ac ad)
  | otherwise =
      let first = do
            t1 <- intersectInterval cd (Interval (Val 1) PosInf)
            divInterval ab t1
          second = do
            t2 <- intersectInterval
                    cd
                    (Interval NegInf (Val (-1)))
            divInterval ab t2
       in unionIntervalM first second
```
Listing 2: Full division implementation

Division: As an example of division of intervals are provided in Listing 2. divBound follows our general intuition, where the main difference is that division by 0 is not undefined, but rather just returns bottom.

The implementation of divInterval is more complex and requires special handling. It first makes sure that c is greater than or equal to 1. That way we are not dividing by 0, which would result in bottom, which we do not want for the entire interval. If the check for c fails, then we instead attempt to evaluate if d is less than or equal to -1. This would be the case, if the entire interval is negative. If both of these evaluations fails, then we know that we do not have an entirely positive nor entirely negative interval, and as such we go to the recursive call.

In the recursive call, we simply find the intersection between all possible positive numbers and our divisor, and attempt to divide ab with it and then the same for the negative values. At last, we just take the union between the two intervals.

In this case, there is never the possibility of dividing by zero, but using the bottomM monad, we don't have to check for this case explicitly.

```
unionIntervalM :: IntervalM -> IntervalM -> IntervalM
unionIntervalM m1 m2 =
  case (m1, m2) of
    (Bottom, Value i2) -> Value i2
    (Value i1, Bottom) -> Value i1
    (Value i1, Value i2) -> unionInterval i1 i2
    (_, _) -> Bottom
```

Listing 3: Union interval implementation

Union Interval: The union interval is the one operator for which Bottom is the neutral element. This means, that if one of the inputs is Bottom, then we should not just return Bottom ourselves. As such, the implementation of union can be seen in Listing 3, where if only one input is bottom, then we simply return the other input.

2.3 Interpretation

Using eBPF-tools, we are given a control flow graph, which is represented as:

```
Set (Label, Trans, Label)
```

This means, that we are given a set of tuples, where the first element is the source of the transformation, followed by the transformation and at last the node we end in.

```
handleTrans :: State -> Trans -> State
handleTrans state Unconditional = state
handleTrans state (NonCF inst) = handleNonCF state inst
handleTrans state (Assert jmp (Reg lhs)) regimm = ...
```

Listing 4: The handleTrans functnion

The handling of Trans is separated into a separate function, `handleTrans`, which can be seen in Listing 4. The function works by getting an initial State(state), and then taking the transformation into account, before returning a new updated State. For example, for the unconditional jump, we simply return the state we got as input, as we gain no new information from the transformation. In the following sections, I will go into more details with an example of the NonCF(non-Control flow) and the Assert (conditional jump).

2.3.1 Non-Control flow

```
handleNonCF :: State -> Instruction -> State
handleNonCF state inst =
  case inst of
    Binary _ alu reg regimm -> handleBinary state (alu, reg, regimm)
    Unary _ alu reg -> handleUnary state (alu, reg)
    Store _ (Reg dst) offset regimm ->
      storeMemory dst offset regimm
    Load _ (Reg dst) (Reg src) offset ->
      loadMemory dst offset (R (Reg src))
    LoadImm (Reg dst) imm ->
      storeMemory dst Nothing (Imm imm)
    _ -> state
 where
  loadMemory :: Int -> Maybe MemoryOffset -> RegImm -> State
  loadMemory dst off (R (Reg src)) =
    let intv = registers state Array.! src
     in case getBounds intv off of
          Nothing -> state
          Just (loSrc, hiSrc) ->
            -- We calculate the new interval as the union between all possible memory locations
            let newInterval = Prelude.foldl unionIntervalM Bottom $ [memory state Array.! i | i <- [loSrc .. hiSrc]]
             in state{registers = registers state // [(dst, newInterval)]}
  loadMemory dst _ (Imm imm) =
    -- offset not used when loading immediate value
    let newInterval = fromInteger $ fromIntegral imm
     in state{registers = registers state // [(dst, newInterval)]}
  ...
  getBounds :: IntervalM -> Maybe MemoryOffset -> Maybe (Int, Int)
  getBounds (Value (Interval lo' hi')) off' =
    -- Make sure we have some memory we can index
    let off = maybe 0 fromIntegral off'
     in let lo = case max lo' (Val (-off)) of
              -- Lower bound has to be the maximum of -off and our item. We then add off after
              Val x' -> x' + off
              _ -> 0
         in let hi = case min hi' (Val (511 - off)) of
                  -- Upper bound has to be the maximum of 511-off and our item. We then add off after
                  Val x' -> x' + off
                  _ -> 511
             in Just (fromInteger lo, fromInteger hi)
```

```
41  getBounds Bottom _ = Nothing
```

Listing 5: handleNonCF

When we are given a non-control flow action, then we gain information about the input state. The Binary and Unary calculations just make use of the interval calculations directly that we have seen from Section 2.2.

I have decided to include the implementation of the loadMemory in Listing 5. This function is called, when we have an eBPF instruction such as:

```
ldxb r5, [r0]
```

Intuitively this instruction wants us to get the value x in r0, then read memory location x and save the result in register r5. However, as mentioned in Section 1.3.1, x is not a single value in our analysis and rather it is an interval. The way we handle this is by first gathering the interval value of r0 on line 16. Then, we find the valid boundary, i.e., we only want to take the value of the interval, which lies within our memory region. If we currently assume that r0 is $\top$, then we only want to look in the memory of [0, 511], as anything else would be out of bounds. The bounds function then also takes into account if we want to offset the value, in which case we simply subtract the value from the interval.

Once we have the valid memory interval, then on line 21 we calculate the union of all these memory cells, which gives us a new single interval for the possible values r5 could take on after the instruction.

The store memory just works in the opposite direction. Here we load the immediate value or the value in a register, and then we offset when writing to the interval of possible memory locations.

2.4 Work-set algorithm

```
workSetAlgorithm :: CFG -> Map Label State -> CFG -> Map Label Int ->
    Map Label State
workSetAlgorithm graph states worklist counters
  | Set.null worklist = states
  | otherwise =
      let (l, instr, n) = Set.elemAt 0 worklist
          w' = Set.deleteAt 0 worklist
          current_state = getPreviousState l
          evalState = handleTrans current_state instr
          oldState = states Map.! n
          newState = unionState evalState oldState
        in if oldState == newState
            -- We don't add anything else
            then workSetAlgorithm graph states w' counters
            else
              workSet' w' n newState oldState
 where
  getSuccessors n' = Set.filter (\(l', _, _) -> l' == n') graph
  getPreviousState l = states Map.! l
  workSet' w' n newState oldState =
    -- Check the counter to figure out if we widen or not
    let total_count = counters Map.! n
```

```
22          newWorkset = Set.union w' successors
23          newStates = Map.insert n newState states
24          successors = getSuccessors n
25          res
26            -- Regular work set algorithm, if we have reached the node
       less than 4 times
27            | (total_count < wideningCount) =
28                let newCount = Map.insert n (total_count + 1) counters
29                 in workSetAlgorithm graph newStates newWorkset newCount
30            -- Widening if we have been here 4 times.
31            | (total_count == wideningCount) =
32                let newStates' = Map.insert n (widenState oldState
       newState) states
33                    newCount = Map.insert n (total_count + 1) counters
34                 in workSetAlgorithm graph newStates' newWorkset newCount
35            -- Narrow state until we have reached the node 10 times
36            | (wideningCount < total_count) && (total_count <
       narrowingCount) =
37                let newStates' = Map.insert n (narrowingState oldState
       newState) states
38                    newCount = Map.insert n (total_count + 1) counters
39                 in workSetAlgorithm graph newStates' newWorkset newCount
40            | otherwise = states
41       in res
```

Listing 6: Workset algorithm

This leads us to the work-set algorithm itself. The workSetAlgorithm function implements just this. The function takes in the full control-flow graph(graph), states (which maps a node(Label) to some State), the workset(a new CFG which contains the tuples (Label, Instr, Label) that remain in the work-set) and counters (a Map from node(Label) to counters(Int).

First, on line 3 we check the base case. If the current workset is empty, then we just return the states.

Otherwise, we remove the first element from the workset, get the state of the label for which we came from, and then evaluate the instruction with handleTrans from before. handleTrans then gives us a new evaluated state given, which in Equation 1(From Section 1), this would correspond to ($[\![exp]\!]s$). We then have to take the union between this new evalState and the previous state from the current node. This way, if a previous transition(Label, Instr, Label) has already reached this node, then we correctly take the union between the two.

Line 11 then checks if this transition lead to a state different than the previous one. If we have the same state, then we add nothing to the work-set, and simply call our workSetAlgorithm again just with the currently checked transition.

If instead we have a changed state, then we go to the workSet' function. The function works by:

1. Get the total number of times we have reached the current Node(total_count).
2. Get all transitions, which start with the current node n(successors).

3. Create the new work-set with all transitions that come after the current one(newWorkset).
4. Create the new states, where we simply substitute the current nodes state with the newState(newStates).
5. At last, we check how many times we have reached the current node. And do one of the following
    - If we have reached the node less than wideningCount times, then we just call the worksetAlgorithm again with the new values, and increment the counter of the current node
    - If we have reached the current node wideningCount amount of times, then we do a single windening step, following the definition from Equation 4 before the recursive call.
    - If we have reached the current node more than wideningCount times, but less than the narrowingCount times, then we do narrowing following the definition of Equation 5 before the recursive call.
    - If we have reached the current node more than narrowingCount times, then we decide that we probably will not get a better estimate, and simply return the states stopping the recursive calls.

2.4.1 Calling the worksetAlgorithm

```
let graph = cfg prog
    labels = Set.toList $ allLabels graph
    initial_states = Map.fromList [(l, bottomState) | l <- labels]
    initial_counters = Map.fromList [(l, 0) | l <- labels]
    final_states = workSetAlgorithm graph initial_states (Set.singleton
      $ Set.elemAt 0 graph) initial_counters
    (_, exit) = Map.findMax final_states
    output = printf (printState exit)
```

Listing 7: Calling the workset algorithm

The workSetAlgorithm can then be called as shown in Listing 7. First, we generate the empty states for each of the nodes, then we initialize all counters(that keep track of the number of times we have visited a node) to zero, before running the function. The function then gives us back a new Map of labels to States, and we simply return the largest Label as our final State, as this is when we exit the program. However, the final_states variable contain the final states for all of the nodes within the graph.

3 Validation

Validation of the implementation includes both unit tests[1] and the default programs provided by eBPF-tools, where the output was simply checked by hand.

[1]These unit tests were created with the help of LLMs, where I then looked over to make sure the test results made sense.

Some programs with load and store instructions were also created and checked by hand to make sure that the implementation gave expected results.

4 Future work

Although the implementation seems to generally work, there are some areas, which I simply have not had the time to improve upon. These parts are referenced here as what could be worked on further.

4.1 Optimizations

By no means is this implementation optimal. The general goal of the project is to keep it as easy to understand as possible, and this has a large effect on the overall optimizations. There are multiple cases of this, but I will just mention what I think would lead to the best performance gain:

- Making it such that we don't store the state of every single node. The current implementation stores the state for every node in the CFG. This is highly inefficient. If a node is visited only once, we could pass its state directly to the next node, eliminating the need to store the state of the incoming node.. We would most likely have to do one pass through of the CFG to check for such nodes, but afterwards we would save a lot of the memory usage that probably occur on larger programs.

- Narrowing and Widening are currently implemented by checking every single register and memory cell to see if the cell/register has changed since the last state. This is highly inefficient, and it should be possible to reduce the number of checked elements. For example for an instruction which sends the value to a register, then we would only have to check that single register. The reason this was not implemented is because of instructions such as:

  ```
  stb [r0], 10
  ```

  Here we are writing to the memory location by the value of r0. r0 is seen as an interval, so in this case we would have to handle the interval as the possible memory cells that have to be updated instead of a single index. For the sake of time, this was not implemented

4.2 Unhandled actions

There are two main areas where I have taken a shortcut and not handled the instructions. Those two are explained here.

Binary operators: The binary operators are not evaluated, instead I simply return the top interval whenever we reach a binary operator. Binary operators would of course give us some further information about the current interval, but due to time constraints these were not implemented.

Conditional jumps: In the provided snippet is how we handle the conditional jumps.

```
case jmp of
  Jeq -> equalInterval i1 i2
  Jne -> notEqualInterval i1 i2
  Jlt -> lessThanInterval i1 i2
  Jle -> lessThanEqualInterval i1 i2
  Jgt -> greaterThanInterval i1 i2
  Jge -> greaterThanEqualInterval i1 i2
  Jslt -> lessThanInterval i1 i2
  Jsle -> lessThanEqualInterval i1 i2
  Jsgt -> greaterThanInterval i1 i2
  Jsge -> greaterThanEqualInterval i1 i2
  _ -> Value (i1, i2) -- For other jumps, we don't refine
```

Here we can see that signed and unsigned usage is not taken into account, and the Jset instruction is also not handled. The Jset instruction is simply not implemented, and we just return our input and with the signed and unsigned one could argue, that we should be more restrictive on the interval. If we ever reach an unsigned instruction, then we would know that the register could never be negative, instead of just using the default interval calculations.

5 Conclusion

In this report I have detailed the implementation of a static value analysis for a subset of the eBPF language, based on the principles of abstract interpretation. The goal was to approximate the possible values of registers and memory at each program point using an infinite interval domain.

The analysis was built upon the control-flow graphs generated by eBPF-tools. The core of the implementation is a work-set algorithm designed to propagate abstract states through the program. The algorithm incorporates a strategy of widening to an infinite interval after a fixed number of iterations, followed by a series of narrowing steps to refine the approximation. The implementation successfully models the abstract semantics for a range of eBPF instructions.

The implementation was validated through a combination of unit tests for the abstract domain operators and by manually verifying the output on some example eBPF programs. These tests indicate that the implementation produces valid output.

Despite this, several areas for future work have been mentioned, including optimizations and handling of more instructions.

In conclusion, this project has successfully produced a functional, though simplified, value analysis for eBPF.

References

[1] Ken Friis Larsen. eBPF Tools, 2025. Accessed: 2025-09-21.

[2] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages*, 4(3-4):120–372, 2017.