Verification Condition Generation for eBPF

Troels Korreman Nielsen (xck773)
Supervisor: Ken Friis Larsen

PCC and eBPF
Department of Computer Science
University of Copenhagen

November 22, 2024

Table of Contents

Table of Contents

What is eBPF?

eBPF allows Linux kernel extensions at runtime.

- Ensures high safety & performance.
- Statically verifies and JIT-compiles programs before running them.
- Used for network filtering, load balancing, monitoring, and security.

eBPF Instruction Set

eBPF programs are submitted in the eBPF ISA:

- "Virtual machine":
  - Eleven 64-bit registers.
  - Integer-addressed memory.
  - Ability to call external helpers.
- Heavily limited/restrictive:
  - Only 64-bit and 32-bit ALU operations.
  - No indirect jumps.
  - No routines*.
  - No stack memory (only a frame buffer).
  - Easy to validate.

```
          mov r1 52
          add r1 3
          div r1 3
          mov r2 r1
          mul r2 11
          jlt r1 r2 label
          mov r1 42
label:
          mov r0 r1
          exit
```

Example eBPF program.

Verifier

All programs must be validated by the verifier before execution.

The verifier serves two goals:

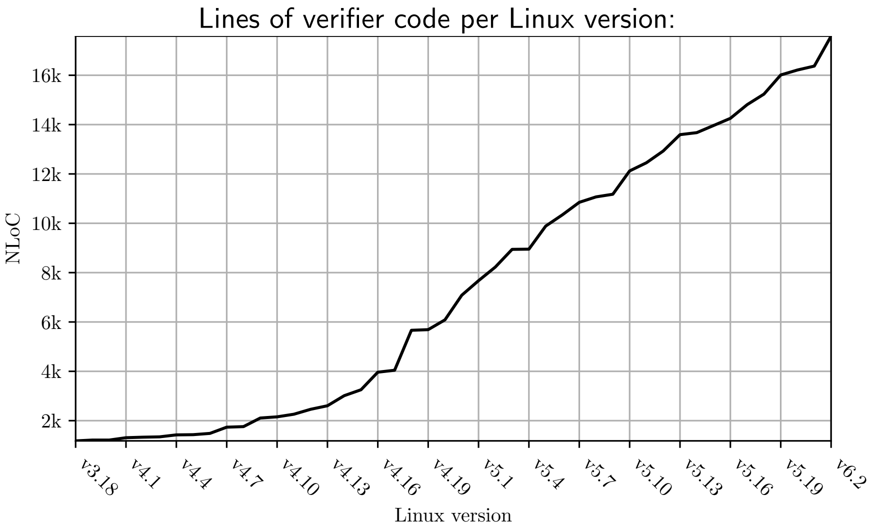
- Goal #1: reject all unsafe programs.
- Goal #2: accept many safe programs.

Static analysis of programs is hard. We have a conflict between ensuring soundness and improving completeness.

Safe programs must:

- Run to completion.
- Access memory correctly.
- Not leak secrets.
- Call helpers correctly.
- etc ...

Complexity


Lines of verifier code per Linux version:
NLoC
16k
14k
12k
10k
8k
6k
4k
2k
v3.18
v4.1
v4.4
v4.7
v4.10
v4.13
v4.16
v4.19
v5.1
v5.4
v5.7
v5.10
v5.13
v5.16
v5.19
v6.2
Linux version

Security

- Accepting unsafe programs means running unsanitized code in kernel space.
- eBPF is a popular attack surface for privilege escalation.
- Many kernel exploits have stemmed from eBPF verifier unsoundness.[1]

[1] Several dozens of eBPF verifier errors tracked on cve.mitre.org

Table of Contents

Proof-carrying code

- Alternative approach to program validation.
- The producer (user) supplies a formal safety proof along with each program.
- The consumer (Linux kernel) only needs to check the supplied proof.

PCC pushes non-trivial work to userspace.

Proofs about programs

Proof system:

- Create a logic that will only derive safe programs.
- We model statement effects through Hoare triples of the form $\{P\}\ S\ \{Q\}$.
  Meaning: If precondition P is true, then postcondition Q is true after executing S.

Verification condition generation:

- From program, generate conditions that imply safety if proven.
- Separates the proof from the program.
- We use weakest precondition calculus to compute preconditions from postconditions:

$$WP : \ \text{Program} \times \text{Formula} \to \text{Formula} \quad \textit{where} \quad \vdash \{WP(S, Q)\}\ S\ \{Q\}$$

Table of Contents

Verification condition generation (for eBPF)

Specified Hoare logic and VC-gen for:

- A subset of eBPF.
  - 64-bit arithmetic.
  - Conditional jumps.
  - Rudimentary memory.
- A subset of safety policies:
  - No division/modulo by zero.
  - Safe memory access.

Proof-of-concept implementation, written in Rust:

- Exports VCs to WhyML for solving.
- No proof extraction, no proof checking.

Table of Contents

WP-calculus

- Straightline instructions can be handled using a WP-calculus:

$$
\begin{array}{lcl}
WP(\texttt{assert}\ P,\ Q) & = & P\ \&\&\ Q \\
WP(\texttt{div}\ r\ v,\ Q) & = & v \neq 0 \land \forall v.\ v = alu(r, v) \implies Q[r \leftarrow v] \\
WP(\texttt{store}\ k\ m\ v,\ Q) & = & Q\ \land\ \exists p, s.\ \textbf{is\_buffer}(p, s)\ \land\ p \leq m < p + s - (k - 1) \\
\ldots & &
\end{array}
$$

WP-calculus

- Straightline instructions can be handled using a WP-calculus:

$$\begin{array}{lcl} WP(\texttt{assert}\ P,\ Q) & = & P\ \texttt{\&\&}\ Q \\ WP(\texttt{div}\ r\ v,\ Q) & = & v \neq 0 \wedge \forall v.\ v = alu(r, v) \implies Q[r \leftarrow v] \\ WP(\texttt{store}\ k\ m\ v,\ Q) & = & Q\ \wedge\ \exists p, s.\ \textbf{is\_buffer}(p, s)\ \wedge\ p \leq m < p + s - (k - 1) \\ \ldots & & \end{array}$$

- But control-flow instructions cannot:

$$WP(\texttt{jlt r0 r1 label}, ???) \quad = \quad ???$$

We need a different approach for these.

Table of Contents

Control graphs

- All eBPF jumps and branches are statically specified (direct).
- No routine calls.

$\Rightarrow$ Modules can be converted to a control-flow graph representation.

Bubble sort

```
    mov r4 0
    mov r8 r2
    sub r8 7
L0:
    mov r3 8
L1:
    mov r5 r3
    add r5 r1
    ldxdw r6 [r5 - 8]
    ldxdw r7 [r5]
    jlt r6 r7 L2
    stxdw [r5 - 8] r7
    stxdw [r5] r6
L2:
    add r3 8
    jlt r3 r8 L1
    add r4 8
    jlt r4 r2 L0
    mov r0 0
    exit
```

Control graph generation (split)

Split into blocks, separating:

- Before labels
- After jumps

```
    mov r4 0
    mov r8 r2
    sub r8 7
```

```
L0:
    mov r3 8
```

```
L1:
    mov r5 r3
    add r5 r1
    ldxdw r6 [r5 - 8]
    ldxdw r7 [r5]
    jlt r6 r7 L2
```

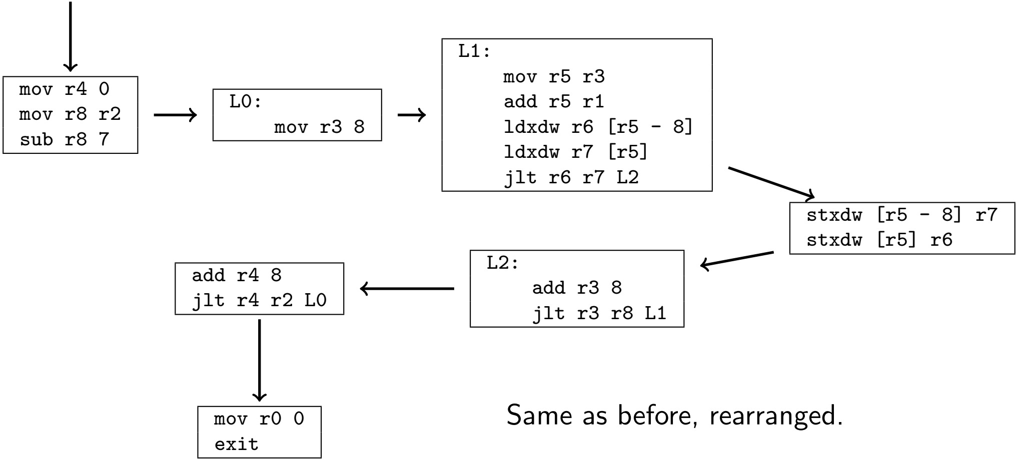
```
    stxdw [r5 - 8] r7
    stxdw [r5] r6
```

```
L2:
    add r3 8
    jlt r3 r8 L1
```

```
    add r4 8
    jlt r4 r2 L0
```

```
    mov r0 0
    exit
```

Control graph generation (rearrange)



Same as before, rearranged.

Control graph generation (blocks)

```
B0
mov r4 0
mov r8 r2
sub r8 7
jmp L0
```

```
L0
mov r3 8
jmp L1
```

```
L1
mov r5 r3
add r5 r1
ldxdw r6 [r5 - 8]
ldxdw r7 [r5]
jlt r6 r7 L2 B3
```

```
B3
stxdw [r5 - 8] r7
stxdw [r5] r6
jmp L2
```

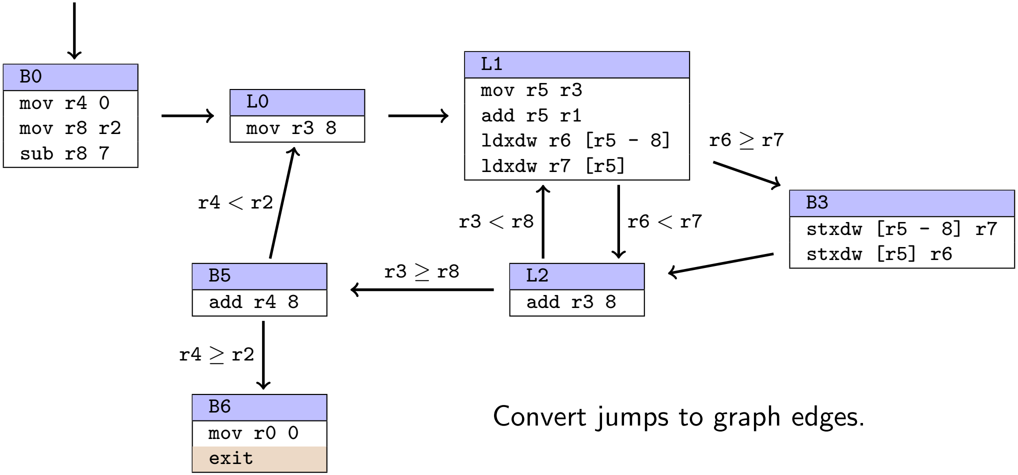
```
B5
add r4 8
jlt r4 r2 L0 B6
```

```
L2
add r3 8
jlt r3 r8 L1 B5
```

```
B6
mov r0 0
exit
```

- Generate extra labels.
- Add implicit jumps.
- Add implicit secondary branch targets.

Control graph generation (edge view)



Convert jumps to graph edges.

Control graph

- View the control graph as a dependency graph.
- The postcondition for a block is created from the conjunction of its jump targets.

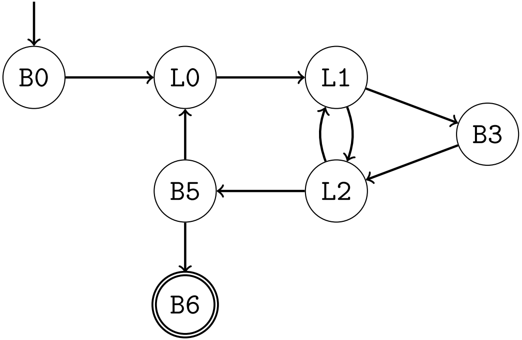
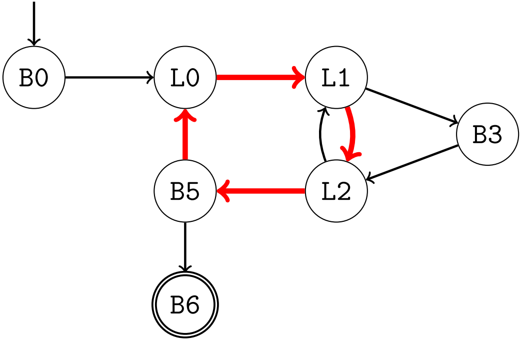


Table of Contents

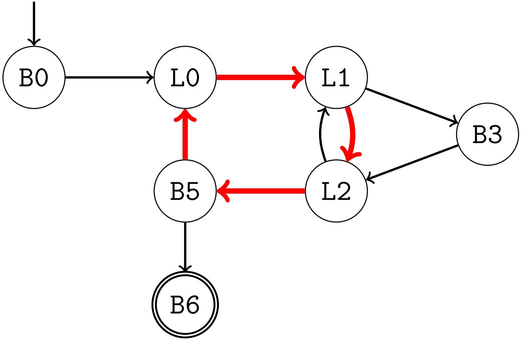
Dependency resolution (cycles)

- In cyclic control graphs, we must contend with dependency cycles.
- A block might indirectly depend on it's own postcondition.
- Prevents a finite definition of VCs.



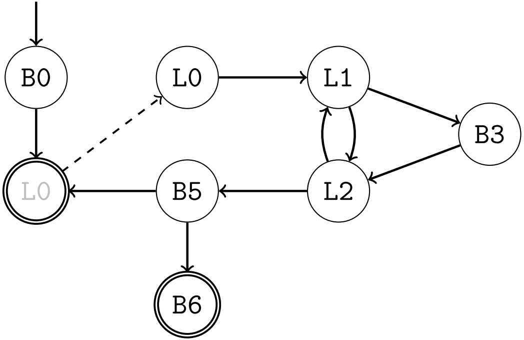
Dependency resolution (cycles)

Break up cycles by requiring user-annotated invariants.



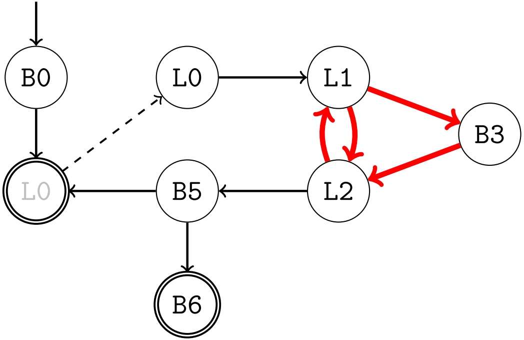
Dependency resolution (cycles)

Break up cycles by requiring user-annotated invariants.



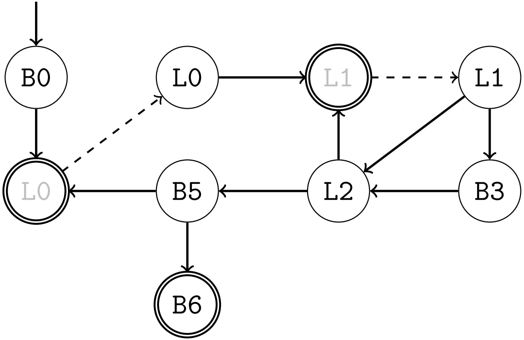
Dependency resolution (cycles)

Break up cycles by requiring user-annotated invariants.



Dependency resolution (cycles)

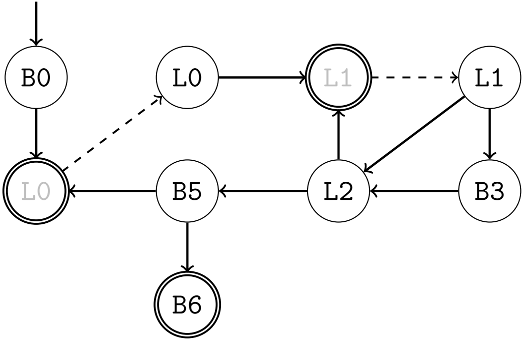
Break up cycles by requiring user-annotated invariants.



Dependency resolution (cycles)

A user-annotated block requirement:

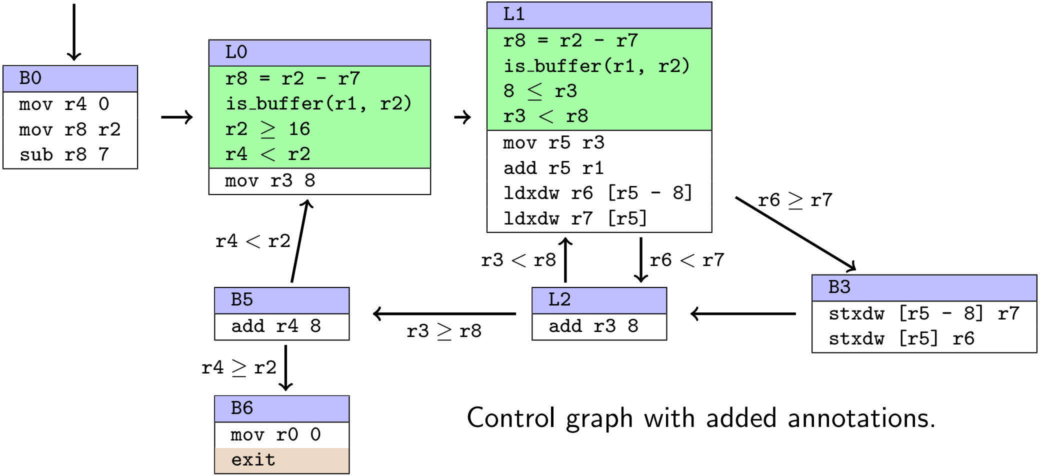
- Must be strong enough to validate its block.
- Must be weak enough to be validated by all dependants.



Annotations referred to by mapping:

$$\alpha : \text{ Label} \rightarrow \text{Formula}$$

Dependency resolution (cycles)

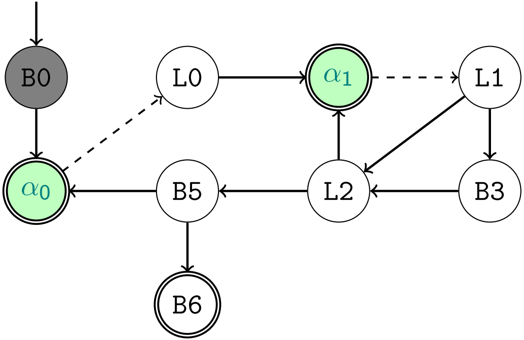


Control graph with added annotations.

Table of Contents

Dependency resolution (traversal)

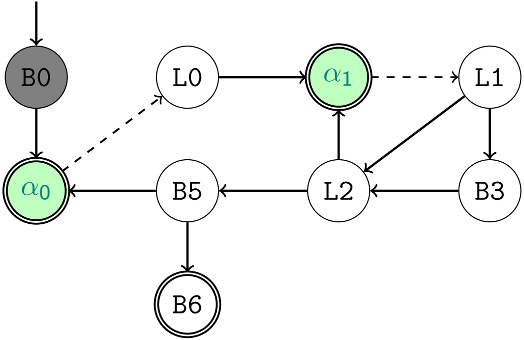
- Compute and cache in Γ the precondition for each block.
- Traverse blocks by reverse topological order.



Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

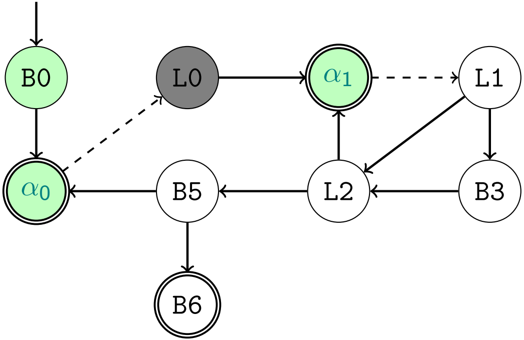
L	$\Gamma(L)$



Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
$\mathtt{B0}$	$WP(\mathtt{B0}, \alpha_0)$

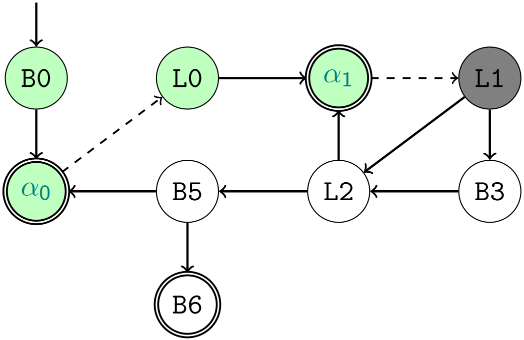


$G_1 : \ \Gamma(\mathtt{B0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
$\mathtt{B0}$	$WP(\mathtt{B0}, \alpha_0)$
$\mathtt{L0}$	$WP(\mathtt{L0}, \alpha_1)$

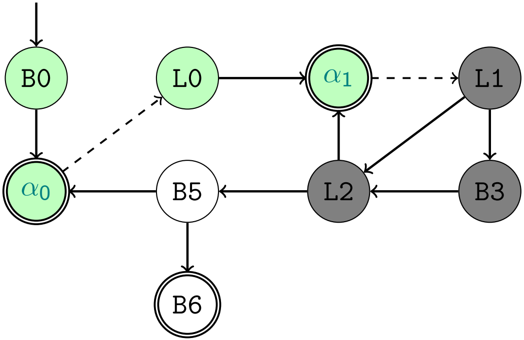


$$G_1 : \Gamma(\mathtt{B0})$$
$$G_2 : \alpha_0 \implies \Gamma(\mathtt{L0})$$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
B0	$WP(\mathtt{B0}, \alpha_0)$
L0	$WP(\mathtt{L0}, \alpha_1)$



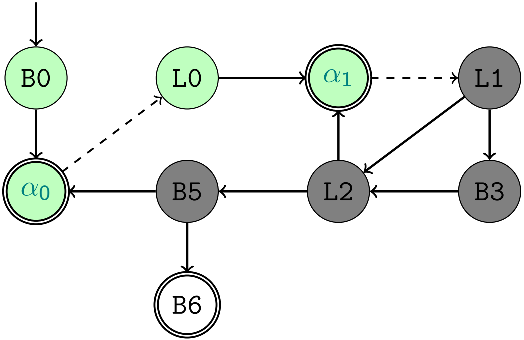
G_1 : $\Gamma(\mathtt{B0})$

G_2 : $\alpha_0 \implies \Gamma(\mathtt{L0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
$\texttt{B0}$	$WP(\texttt{B0}, \alpha_0)$
$\texttt{L0}$	$WP(\texttt{L0}, \alpha_1)$



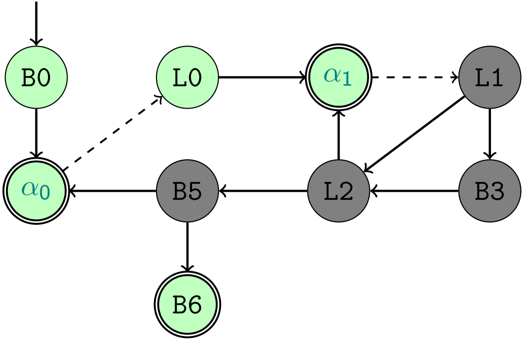
$G_1 : \Gamma(\texttt{B0})$

$G_2 : \alpha_0 \implies \Gamma(\texttt{L0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
$\mathtt{B0}$	$WP(\mathtt{B0}, \alpha_0)$
$\mathtt{L0}$	$WP(\mathtt{L0}, \alpha_1)$
$\mathtt{B6}$	$WP(\mathtt{B6}, \mathtt{true})$



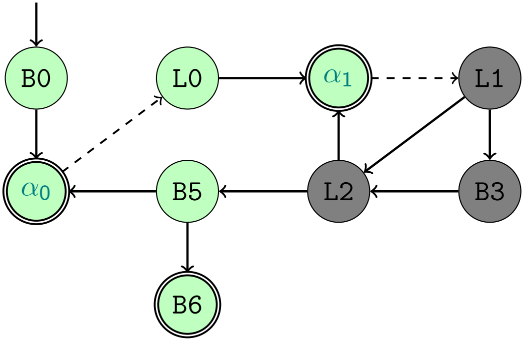
$G_1 : \ \Gamma(\mathtt{B0})$

$G_2 : \ \alpha_0 \implies \Gamma(\mathtt{L0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
B0	$WP(\texttt{B0}, \alpha_0)$
L0	$WP(\texttt{L0}, \alpha_1)$
B6	$WP(\texttt{B6}, \texttt{true})$
B5	$WP(\texttt{B5}, \alpha_0 \wedge \Gamma(\texttt{L0}))$



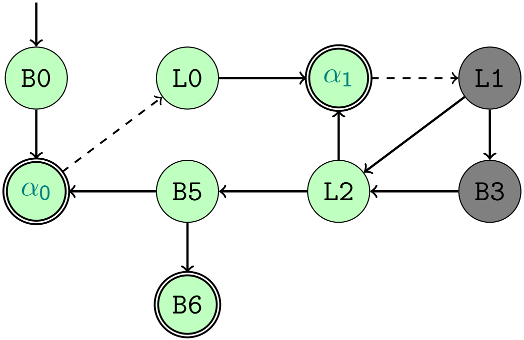
$G_1 : \; \Gamma(\texttt{B0})$

$G_2 : \; \alpha_0 \implies \Gamma(\texttt{L0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
B0	$WP(\texttt{B0}, \alpha_0)$
L0	$WP(\texttt{L0}, \alpha_1)$
B6	$WP(\texttt{B6}, \texttt{true})$
B5	$WP(\texttt{B5}, \alpha_0 \land \Gamma(\texttt{L0}))$
L2	$WP(\texttt{L2}, \alpha_1 \land \Gamma(\texttt{B5}))$



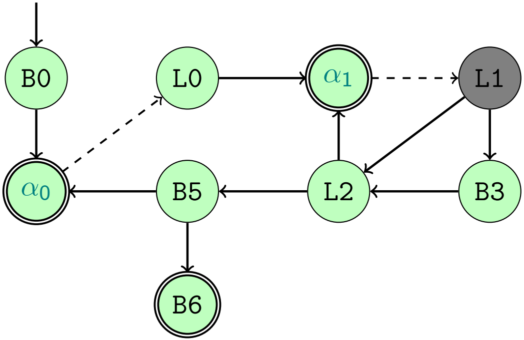
G_1 : $\Gamma(\texttt{B0})$

G_2 : $\alpha_0 \implies \Gamma(\texttt{L0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
B0	$WP(\texttt{B0}, \alpha_0)$
L0	$WP(\texttt{L0}, \alpha_1)$
B6	$WP(\texttt{B6}, \texttt{true})$
B5	$WP(\texttt{B5}, \alpha_0 \wedge \Gamma(\texttt{L0}))$
L2	$WP(\texttt{L2}, \alpha_1 \wedge \Gamma(\texttt{B5}))$
B3	$WP(\texttt{B3}, \Gamma(\texttt{L2}))$



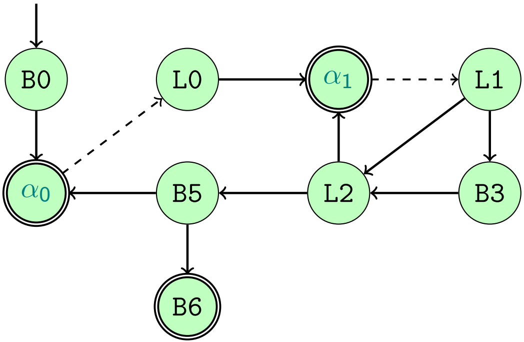
$G_1 : \Gamma(\texttt{B0})$

$G_2 : \alpha_0 \implies \Gamma(\texttt{L0})$

Dependency resolution (traversal)

(Ignore edge conditions for simplicity.)

L	$\Gamma(L)$
$\texttt{B0}$	$WP(\texttt{B0}, \alpha_0)$
$\texttt{L0}$	$WP(\texttt{L0}, \alpha_1)$
$\texttt{B6}$	$WP(\texttt{B6}, \texttt{true})$
$\texttt{B5}$	$WP(\texttt{B5}, \alpha_0 \wedge \Gamma(\texttt{L0}))$
$\texttt{L2}$	$WP(\texttt{L2}, \alpha_1 \wedge \Gamma(\texttt{B5}))$
$\texttt{B3}$	$WP(\texttt{B3}, \Gamma(\texttt{L2}))$
$\texttt{L1}$	$WP(\texttt{L1}, \Gamma(\texttt{L2}) \wedge \Gamma(\texttt{B3}))$



$$G_1 : \Gamma(\texttt{B0})$$
$$G_2 : \alpha_0 \implies \Gamma(\texttt{L0})$$
$$G_3 : \alpha_1 \implies \Gamma(\texttt{L1})$$

Table of Contents

In practice (input)

```
;# requires is_buffer(r1, r2)
;# requires r2 >= 16
    mov r4 0
    mov r8 r2
    sub r8 7
outer_loop:
;# req r8 = sub(r2, 7)
;# req is_buffer(r1, r2)
;# req r2 >= 16
;# req r4 < r2
    mov r3 8
inner_loop:
;# req r8 = sub(r2, 7)
;# req is_buffer(r1, r2)
;# req 8 <= r3
;# req r3 < r8
    mov r5 r3
    add r5 r1
    ldxdw r6 [r5 - 8]
    ldxdw r7 [r5]
    jlt r6 r7 skipped
    stxdw [r5 - 8] r7
    stxdw [r5] r6
skipped:
    add r3 8
    jlt r3 r8 inner_loop
    add r4 8
    jlt r4 r2 outer_loop
    mov r0 0
    exit
```

In practice (output)

```
use mach.int.UInt64
use int.Int
use int.ComputerDivision
predicate is_buffer (p: uint64) (s: uint64)

goal inner_loop: forall r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 : uint64 . (((((r8 = (r2 - 7)) && is_buffer r1 r2) &&
     (8 <= r3)) && (r3 < r8)) -> (forall v6 : uint64 . ((v6 = r3) -> (forall v5 : uint64 . ((v5 = (v6 + r1)) ->
     ((exists p3 : uint64 . (exists s3 : uint64 . (is_buffer p3 s3 /\ ((p3 <= (v5 + -8)) /\ ((v5 + -8) < ((p3 +
     s3) - 7)))))) /\ (forall v4 : uint64 . ((exists p2 : uint64 . (exists s2 : uint64 . (is_buffer p2 s2 /\ ((p2
      <= (v5 + 0)) /\ ((v5 + 0) < ((p2 + s2) - 7)))))) /\ (forall v3 : uint64 . (((v4 < v3) && (forall v2 :
     uint64 . ((v2 = (r3 + 8)) -> (((v2 < r8) && ((((r8 = (r2 - 7)) && is_buffer r1 r2) && (8 <= v2)) && (v2 < r8
     ))) \/ (not ((v2 < r8)) && (forall v1 : uint64 . ((v1 = (r4 + 8)) -> (((v1 < r2) && ((((r8 = (r2 - 7)) &&
     is_buffer r1 r2) && (r2 >= 16)) && (v1 < r2))) \/ (not ((v1 < r2)) && true)))))))))) \/ (not ((v4 < v3)) &&
     ((exists p1 : uint64 . (exists s1 : uint64 . (is_buffer p1 s1 /\ ((p1 <= (v5 + -8)) /\ ((v5 + -8) < ((p1 +
     s1) - 7)))))) /\ ((exists p0 : uint64 . (exists s0 : uint64 . (is_buffer p0 s0 /\ ((p0 <= (v5 + 0)) /\ ((v5
     + 0) < ((p0 + s0) - 7)))))) /\ (forall v2 : uint64 . ((v2 = (r3 + 8)) -> (((v2 < r8) && ((((r8 = (r2 - 7))
     && is_buffer r1 r2) && (8 <= v2)) && (v2 < r8))) \/ (not ((v2 < r8)) && (forall v1 : uint64 . ((v1 = (r4 +
     8)) -> (((v1 < r2) && ((((r8 = (r2 - 7)) && is_buffer r1 r2) && (r2 >= 16)) && (v1 < r2))) \/ (not ((v1 < r2
     )) && true)))))))))))))))))))))

goal outer_loop: forall r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 : uint64 . (((((r8 = (r2 - 7)) && is_buffer r1 r2) && (
     r2 >= 16)) && (r4 < r2)) -> (forall v7 : uint64 . ((v7 = 8) -> ((((r8 = (r2 - 7)) && is_buffer r1 r2) && (8
     <= v7)) && (v7 < r8)))))

goal entry: forall r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 : uint64 . (((true /\ is_buffer r1 r2) /\ (r2 >= 16)) -> (
     forall v10 : uint64 . ((v10 = 0) -> (forall v9 : uint64 . ((v9 = r2) -> (forall v8 : uint64 . ((v8 = (v9 -
     7)) -> ((((v8 = (r2 - 7)) && is_buffer r1 r2) && (r2 >= 16)) && (v10 < r2)))))))))
```

Final step

In a proper PCC architecture:

1. Pass to solver.
2. Extract proof if successful solve.
3. Pass proof along with program to Linux subsystem.

In these experiments:

1. Pass to Why3.
2. Why3 tells us whether conditions can be proven or not.

✓ Why3 accepts the bubble sort.

Comparison to verifier

VC-gen can validate things the verifier can't:

- Relationships between registers (and memory cells, theoretically).
- Non-contiguous value ranges.
- Dynamically sized buffers.
- . . . and more.

Table of Contents

Termination challenges

- If PCC for eBPF is to be viable, we need total correctness.

Termination challenges

- If PCC for eBPF is to be viable, we need total correctness.
- Proof rules for termination in structured languages:

$$\frac{\{C \wedge P \wedge v = X\}\ s\ \{P \wedge v \prec X\}}{\{P\}\ \textbf{while}\ C\ \textbf{invariant}\ P\ \textbf{variant}\ v\ \textbf{do}\ s\ \textbf{done}\ \{P \wedge \neg C\}}$$

Where $(\prec)$ is a well-founded relation.

Termination challenges

- If PCC for eBPF is to be viable, we need total correctness.
- Proof rules for termination in structured languages:

$$\frac{\{C \land P \land v = X\}\ s\ \{P \land v \prec X\}}{\{P\}\ \textbf{while}\ C\ \textbf{invariant}\ P\ \textbf{variant}\ v\ \textbf{do}\ s\ \textbf{done}\ \{P \land \neg C\}}$$

  Where $(\prec)$ is a well-founded relation.
- eBPF is unstructured, so above method doesn't apply.

Termination challenges

- If PCC for eBPF is to be viable, we need total correctness.
- Proof rules for termination in structured languages:

$$\frac{\{C \land P \land v = X\}\ s\ \{P \land v \prec X\}}{\{P\}\ \textbf{while}\ C\ \textbf{invariant}\ P\ \textbf{variant}\ v\ \textbf{do}\ s\ \textbf{done}\ \{P \land \neg C\}}$$

  Where $(\prec)$ is a well-founded relation.
- eBPF is unstructured, so above method doesn't apply.
- Idea: Limit scope to "semi-structured" control graphs!

Semi-structured control graphs

- "Semi-structured" control graph: partial ordering $\left(\overrightarrow{<}\right)$ of nodes in graph G.

Semi-structured control graphs

- "Semi-structured" control graph: partial ordering $\left(\overrightarrow{<}\right)$ of nodes in graph G.
- Straight path: Acyclic path from entry to node.

Semi-structured control graphs

- "Semi-structured" control graph: partial ordering $\left(\overrightarrow{<}\right)$ of nodes in graph G.
- Straight path: Acyclic path from entry to node.
- Relations:
  - $A \overrightarrow{<} B$ if all straight paths to B include A.
  - $A \overrightarrow{\not<} B$ if no straight paths to B include A.

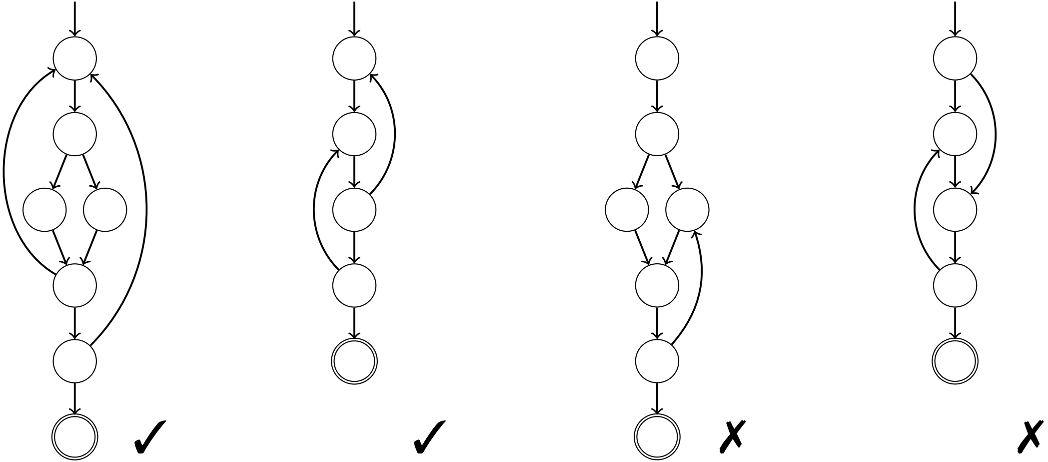
Semi-structured control graphs

- "Semi-structured" control graph: partial ordering $(\overset{\rightarrow}{<})$ of nodes in graph G.
- Straight path: Acyclic path from entry to node.
- Relations:
  - $A \overset{\rightarrow}{<} B$ if all straight paths to B include A.
  - $A \overset{\rightarrow}{\not<} B$ if no straight paths to B include A.
- Valid graph: for all edges $A \rightarrow B \in G$, either $B \overset{\rightarrow}{<} A$ or $B \overset{\rightarrow}{\not<} A$.

Semi-structured control graphs

- "Semi-structured" control graph: partial ordering $\left(\overrightarrow{<}\right)$ of nodes in graph G.
- Straight path: Acyclic path from entry to node.
- Relations:
  - $A \overrightarrow{<} B$ if all straight paths to B include A.
  - $A \overrightarrow{\nless} B$ if no straight paths to B include A.
- Valid graph: for all edges $A \to B \in G$, either $B \overrightarrow{<} A$ or $B \overrightarrow{\nless} A$.
- Ordering is chronological:
  Edges jump either forwards or backwards.

Examples



Termination proofs

Any backwards jump targets a guaranteed previously visited node. We must then show:

1. Decreasing variants for all backwards jumps.
2. No sequence of backwards jumps can indefinitely increase a variant.

Not within scope of this presentation.
Nor fully within the scope of this project, for that matter.

Table of Contents

Conclusion

In this project, I have:

- Identified safety policies for eBPF.
- Demonstrated VC-gen for an eBPF subset.
- Shown viability of VC-gen for branching & looping assembly.
- Shown PCC validating programs that Linux verifier cannot.

Future work:

- Proper exploration of termination proofs.
- Expansion of eBPF subset.
- Secret leak prevention.
- Proof-of-concept full framework.