LLVM BPF backend improvements

Open discussion

Goals

- The BPF Steering Committee (BSC) putting together SOW
- Share what is currently known
- Gather feedback from the community

LLVM and the verifier

- Translates code from C to LLVM IL to BPF byte code
- Performs a variety of optimizations
- Some optimizations break verification
  - Developers need to resort to inline-assembly (Tetragon as an example)
  - Correlated branches break Prevail Verifier

Code Coverage and BPF

- Code coverage tools is done via instrumentation
- Not yet supported in BPF
- Challenges exist as the BPF byte code is translated from byte code to machine code
- May require Kernel support (for code coverage)

Likely/Unlikely hints to the optimizer

- Hints may be present in the original C code
- This information is lost as BPF byte code is generated
- More advanced JIT compilers could leverage this
- Possibly extend this to even support PGO?

Possible solutions?

- Allow developers to provide explicit assertions
  - LLVM and verifier could then give better feedback
- Integrate the verifier into LLVM to fail compilation on unverifiable code

Prevail Verifier

- Verifier for BPF byte code built on Abstract Interpretation
- Abstract Interpretation is a field of math used in static analysis
- Operates over a control flow graph of BPF byte code
- Walks the graph in weak topographical order to produce assertions
- Performs analysis in polynomial time

Issues with Prevail and LLVM

- LLVM optimizer folds code paths to avoid repeating tests
- This results in correlated branches
- Two branches that are always either both taken or both not taken
- Breaks LTO based analysis
- This occurs frequently in the Cilium code base
- Work around involves marking pointers as volatile
  - Prevents LLVM from skipping the second test

Synthetic example of a correlated branch

```
r5 = 0                     // Flag set to false
r2 = *(u32 *)(r1 + 0)      // R2 points to data
r1 = *(u32 *)(r1 + 4)      // R1 points to data_end
r3 = r1                    // R3 points to data
r1 = r2                    // R1 points to data_end
r2 += 4                    // R2 points to data + 4
r2 > r3 goto +1            // Jump if (data + 4) > data_end
r5 = 1                     // Set flag to true
if r5 == 0 goto +1         // If flag is true, skip the next instruction
r0 = *(u32 *)(r1 + 0)      // Dereference data
```

Solutions

- Provide finer grained control over LLVM optimizer?
- Break through in Abstract Interpretation to solve this?
- Suggestions?