The challenge of PITA of compiling for verified targets

Jose E. Marchesi

November 13, 2023
This is an unsolvable problem
Motivation

We need to make compiled BPF practical.

In GCC:

1. Achieve parity with clang/llvm.
2. Compile and run all kernel BPF selftests.
3. Compile and run all **actually existing** BPF programs.
4. Compile and run as many **potentially existing** BPF programs as possible.
Verified targets

--- Linux verifier
foo.c ---> foo.o ---> Windows PREVAIL ---> run
       ---> Others...
                |     |                                 |
                |     +------------------------------- reject

• Verification vs. sandboxing.
• Assembly programming vs. higher-level languages.
• Impact of optimizing compilers.
• Practical diagnosis.
• Multiple verifiers, in multiple versions.
• Verifiers themselves need to be bounded.
• Require metadata in form of debug info.
Toolchain Challenges

- Dealing with architectural peculiarities
- Generating verifiable code (big deal)
- Making compiling verifiable code **practical** and **tolerable**
What can lead to unverifiable code

• Unverifiable source constructs
  → error.

• Optimization-driven transformation
  → if not avoidable, error.
Approach 0: do nothing

- Compiler behavior is influenced by the backend.
- But optimizations are handled as usual.
- Good enough for DTrace’s BPF support routines.
- Probably not good enough for the kernel BPF selftests.
- **For sure** not good enough for actually existing BPF.
- Strategy currently used in bpf-unknown-none-gcc :P
Approach 1: disable all optimizations

- Impact on performance.
- Impact on program size, which is limited.
- Partial solution: unverifiable source constructs remain.
- Actually existing BPF requires -O2 or higher.
- Potentially existing BPF will benefit from opts.
Approach 2: disable some optimizations

• Disable optimizations that lead to unverifiable code.
• Should be automatic to be practical.
• verifier → constraints → IR contract
• Process:
  1. Try pass.
  2. Check constraints in resulting IR.
  3. If constraints not respected, discard pass effects.
• Bad granularity: a pass may perform many transformations.
• Can GCC discard pass effects after the pass is run?
Approach 3: target counterpasses/antipasses

- Target adds anti-passes that undo some transformations performed by optimization passes.
- Better granularity.
- Pretty neutral to rest of the compiler.
- Strategy currently used by clang/LLVM.
- Fragile: forks.
- Maintenance hell.
Approach 4: target driven pass tailoring

- BPF backend disables particular transformations by hooking in passes.
- Strategy currently used by clang/LLVM.
- Other compiler maintainers are reluctant.
- Legal transformations become “illegal”.

Approach 5: generic pass tailoring

• We already have -Osmall, -Ofast.
• Let’s add -Overifiable (or -Opredictable).
• Passes adapt to the “verifiable” criteria.
• Tradeoffs.
• Not restricted to any particular backend.
• May be useful for “normal” targets too.
Approach 6: language level support

- You-must-know pragmas.

```c
#pragma loop must bound 0..64
for (i = 0; i < x; ++i)
{
    ...
}
```

Fail at compile-time if the compiler cannot guarantee with 100% certainty, all compilation stages considered, that the bounds of the loop are indeed between 0 and 100.

- Optimize-or-fail pragmas: always_inline, musttail, etc.
Approach 7: assembler support

- Put the kernel verifier in the assembler.
  - Maintenability concerns.
  - Licensing concerns.
- Invoking the kernel verifier (syscall) from the assembler.
  - Portability concerns.
  - Cross-assembling concerns.
  - Interface concerns: parsing verifier output.
- Assembly time static analyzer based on ginsns and cfg.
- DWARF to associate errors with source constructs.
Discussion

• Solution is likely a combination of approaches and techniques.
• Must be pre-meditated and consensuated.
• Shouldn’t be BPF centric.
• Coordination between toolchains
  • BPF standardization process.
  • bpf@vger mailing list.
  • GCC BPF wiki: https://gcc.gnu.org/wiki/BPFBackEnd
  • Clang/llvm issues: https://reviews.llvm.org
  • BPF office hours.
A few extra discussion items

• Inclusion of libc headers in BPF programs, like stdint.h.
• Assembler: register names as symbols in certain contexts.
• Optimization-generated funcalls, like strcmp.
• Keeping kernel selftests building with GCC: CI.
Thanks