Enabling eBPF on Embedded Devices Through Decoupled Verification

Milo Craun, Adam Oswald, Dan Williams - Virginia Tech
BPF is Useful

Typical BPF Use Cases

- Networking
- Increased observability over the system
- Increased security
- Improve performance of systems
- Safely extend kernel
BPF is Useful on Embedded

- BPF is used for networking → Embedded systems used at the edge for sensor networks
- BPF is used for observability → Observability over embedded systems matters
- BPF is used to change kernel policy → Safely and dynamically change embedded behavior

New use cases will come in the future
Main Challenges to BPF on Embedded
Main Challenges to BPF on Embedded

Write BPF Code
libbpf, BCC
Main Challenges to BPF on Embedded

- Write BPF Code
  *libbpf, BCC*

- Compile to BPF bytecode
Main Challenges to BPF on Embedded

Write BPF Code
libbpf, BCC

→

Compile to BPF bytecode

→

Load Program
Main Challenges to BPF on Embedded
Main Challenges to BPF on Embedded
Main Challenges to BPF on Embedded
Main Challenges to BPF on Embedded

EXPENSIVE!
Main Challenges to BPF on Embedded

EXPENSIVE!
Main Challenges to BPF on Embedded
Main Challenges to BPF on Embedded

EXPENSIVE!
Verification is Expensive

- Verification grows with number of paths
- Occurs at load time, every time
- 20 to 70 times slower
- Weaker processor verifies slower
- Close to 10 seconds for 2048 paths
- BPF programs will not become simpler
Key Insight

BPF program verification need not happen at load time

- Only verify programs one time
- Can verify programs at any time
- Can spend as much CPU time as needed

We achieve this by decoupling verification
Decoupled Flowchart

1. Write BPF Code
   libbpf, BCC

2. Compile to BPF bytecode

3. Verify and JIT compile BPF Program

4. Attach BPF program

5. Program ready to use

Embedded System
Talk Roadmap

- Motivate BPF on embedded
  - Challenges
  - Opportunities
- Walkthrough our decoupling design
  - Key points and design goals
- Discuss how decoupling goes beyond embedded
  - Verification as a service
  - Expanding the BPF Ecosystem
Design Goals

- Ensure Linux kernel compatibility
- Ensure verifier trust
- Enable pre-verified programs to be loaded
Design Goals

- Ensure Linux kernel compatibility
- Ensure verifier trust
- Enable pre-verified programs to be loaded
Ensuring Kernel Compatibility

- Use the in-place Linux kernel verifier and JIT
- Ensure that the program would have been verified on embedded device
- Keeps all the safety properties of the current verifier
- Produce what would have been created if the embedded kernel verified the program
Ensuring Trust

- A user cannot trust the pre-verified code on its own
- Ensure trust by signing the pre-verified blob
- If the user trusts the verifier then they can load
Enabling Loading

- Helper functions and maps need to be relocated
- Emit metadata that specifies helper id and where the relocation needs to be done
- Maps are still in progress
- Resolve these on the embedded kernel
Decoupling is Deeper than Embedded

Decoupling opens up new opportunities in BPF world

- Increasing BPF program complexity
- Verification as a service
- Expanding the ecosystem
Raising BPF Limits

- BPF program complexity is limited by the verifier
- Techniques to bypass this are clever, but clumsy
  - Separating program into smaller pieces to verify
- Decoupling allows us to increase these limits substantially
- Greatly increases the possible complexity of BPF programs
Verify Once Run Everywhere

- BPF wants to be Compile Once Run Everywhere
  - Allows for compatibility between similar kernel versions
  - Aids distribution of BPF programs
- End goal is to make it easy to load BPF programs
- With decoupled verification BPF can be Verify Once Run Everywhere
  - Extremely easy to load BPF programs
  - Computationally cheap
Expanding Ecosystem

- Current verifier and JIT are ad-hoc
- Decoupling better enables the use of alternative verifiers and JITs
- Alternative verifiers may be better than Linux verifier
  - Better time complexity
  - Ensure different properties
- Allow experimental/architecture specific JIT
  - Take more advantage of the specific hardware being run on
Distributed Verification

BPF Bytecode → Verification Stage → JIT Unit → Output Stage

- Input Stage
- Verification Stage
- JIT Stage
- Output Stage
Takeaways

- BPF is useful on embedded devices
- BPF program verification does not need to be at load time
- Decoupling verification from load time allows
  - BPF programs on embedded
  - Expanded BPF ecosystem and program complexity
  - New types of services to support BPF