Linux Plumbers Conference 2022

Contribution ID: **293** Type: **not specified**

OPENED Tool for Managing eBPF Heterogeneity

Wednesday, 14 September 2022 12:30 (30 minutes)

Case for OPENED for eBPF NF Development

The recent past has been the emergence of eBPF in building high performance networking usecases such as load balancing, K8s CNI, DDoS protection, traffic shaping etc. However, unlike traditional software datapath technologies, eBPF code development exhibits enormous heterogenity in terms of choice of kernel hook points, data sharing mechanisms as well as kernel loading tools. Today, these decisions are made at code development time; however, to be truly effective such decision must be made hollistically using information about other eBPF programs running on the server.

We argue that the developer of an network function (NF) (consisting of multiple eBPF functions) has no idea of the other NFs that will be chained together at run time to create the datapath. Hence, decisions taken at the development stage are bound to be suboptimal. A solution for this problem can be taking eBPF specific decisions (such as hook point) at run-time. Unfortunately the process of altering design choices at run time is non-trivial due to two properties of the eBPF runtime. First, porting code written for one hook point to another requires modification in terms of input data structures and available bpf_helper functions. Second, deciding the optimal and most efficient combination eBPF specific decisions (e.g., data structures) requires exploring a large number of design choices.
For example, porting and reusing existing functionalities, say GUE encap/decap processing from Meta's Katran code base, in a new program would require isolating the GUE specific functionalities and their associated control and data dependencies, and modifying them for use in the new program. This process requires complete understanding of the program, is time consuming and typically tends to be error prone.

The porting task is further complicated in eBPF due to its heterogenity that prevents code written for one hook point from generally being able to run at a different hook point. For example, consider an observability program parsing packet headers and updating counters, that is written in XDP chained with a TC program that also parses headers and performs QoS enforcement. To avoid duplication of parsing, the developer might want to move the observability program to the TC hook point, chain it with the QoS enforcement TC program and share parsed packet headers between both modules. However, without appropriate transformations, XDP code cannot be run at TC.

This difficulty of porting also leads to large eBPF projects working in (strong) siloes and not reusing similar functionalities available in other production grade open source projects. A documented example of such behavior is the decision of the Cloudflare team to develop their own load balancer code, Unimog, instead of reusing Meta's Katran load balancer. A consequence of this difficulty to port eBPF code is that a typical eBPF solution is built as a monolith consisting of a number of tightly coupled eBPF programs.

Clearly such siloed and monolithic developer community does not augur well for both a) wider eBPF adoption as new developers will either have to rewrite readily available modules or reuse the entire code base, a choice that will likely introduce unnecessary bloat and overheads to their solution. As well as for b), future innovation as developers will waste effort in adding implementations of similar functionalities (in same language!) in their siloed codebases, resulting in replication of effort instead of combining forces innovating on the newer paradigm changing design options that eBPF introduces.

While there have been efforts like BTF CORE to streamline deployment of eBPF code across different kernel versions, very little effort exists in making eBPF code reusable amongst codebases. In particular, recent efforts such as Walmart's L3AF requires rewriting code to use tail calls and is further limited to programs of the same type. We believe that demonstrating the feasibility of a general approach {\to transforming eBPF NF code built with certain (development time) eBPF design choices to run time optimal choices based on actual datapath requirements} is a key first step towards breaking developer silos and fostering concerted innovation in eBPF based datapath technology.

This motivates us to create tooling that enables 1) automated *extraction* of specific eBPF code pieces from different projects, 2) hook point specific *transformation* that facilitates running code written for one hook point against a target hook point and 3)*composition* of multiple programs to create the necessary pipeline. We envision a world, where NF developers can pick and choose functionalities from different projects and compose them together to build flexible and high performance network datapaths. In this paper, we describe OPENED, a tool that supports extracting specific code functionalities from a given project, transforming them for running at the desired target hook point and composing them together to build flexible packet processing pipelines.

Workflow

Our tool has a three stage workflow corresponding to three major tasks for consuming third party code in one's project, viz., a) Extraction, b) Transformation and c) Composition. Each of the stages, in turn, consists of a multi-step user-in-the-loop workflows to inform and guide the tool in making appropriate decisions. The input to the system consists of a yaml specification describing the required information for all three stages.

Stage 1: Extraction

For the first stage of extracting code, the specification provides an array of network functions of source code, in the form [URI:kernel_ebpf_code_repository, URI:file_path:line number] of function definitions. For example, the "xdpdecap" function in Katran will be specified as: [github.com/facebookincubator/katran/blob/main/katran, github.com/facebookincubator/katran/blob/main/katran/decap/bpf/decap_kern.c:223]. Given this input, our current prototype computes the Minimal Compilable Unit (MCU), i.e. the minimal set of source artefacts e.g. source files, configruations, data sources, build files etc. in third party code base which when taken together will successfully compile, and be able to load and execute in the kernel at the same kernel hook point. The automated extraction of MCU involves identifying both control and data dependencies (in the form of eBPF map updates and look ups) amongst functions. Our prototype extends Codequery tool which provides a sqlite db with querying capabilities on top of CTAGS and Cscope indices of the entire codebase. We extend codequery to determine the function call graph of our extraction target and the functions called by them recursively. We stop exploration of function call graph once the called functions are defined in standard system libraries.

The output of the tool is two JSON arrays corresponding to the list of all functions along with their location (file, start and end line numbers) inside code_repository and the definitions of various maps which are utilized in the code. Our tool also generates two types of warning results for which it needs user-in-the-loop intervention a) specific instances of global maps for which definitions was not found in the source code inside code_repository (maps which are instantiated in user code), b) list of functions for which multiple declarations with same call signature are found. For warnings of first type, the user is expected to ensure that map instantiation (inside user code) is also done for the ported instance. For the second type of warnings the user needs to keep the right function call details and remove the details of duplicates. A simple program then copies all the selected files and map declarations to create a new source file for MCU, which is then compiled and loaded at the original hookpoint to complete the extraction stage.

Stage 2: Transformation

For the hook point transformation, the input yaml specifies the target hook point for the function extracted earlier. Hook point transformation is implemented using source code transformation tools viz. coccinelle and TXL that allow developers to express matching patterns/rules in source code and their corresponding code level transformations. Our choice of using source code transformation tools, as opposed to byte code level transformation is motivated by the need for developers to maintain/debug source code repositories over longer time periods. Source code transformation tools seem to be sufficient for most of the use cases we have encountered so far. For instance, for XDP to TC transformation, we need to replace XDP decisions such as XDP_PASS and XDP_DROP with corresponding TC actions such as TC_ACT_OK(/PIPE) and TC_ACT_SHOT. Similarly we need rules to replace byte offsets such as ethernet header protocol (ethhdr->h_proto) value with corresponding skb struct field (skb->protocol) accesses. Similarly, we require rules that can transform bpf helper functions across hook points. Based on our experiments with large open source code repositories, we find that the combination of coccinelle and TXL is sufficient for our transformation rules. We would also point out, that not all pairs of hook point transformations are feasible, starting from source code, for instance due to the unavailability of corresponding helper functions, e.g. bpf_msg_push_data at XDP layer. In this case, the tool will raise an error that the transformation is not feasible due to missing helper functions or lack of available kernel state (e.g. connection tracking state at XDP). To enable universal hook point transformation, there is a need for a domain specific language(DSL) where the developer expresses packet processing operations at a high level, that are then compiled down to eBPF hook point specific programs. We leave the design of such a DSL to future work. At the end of the transformation stage, we verify that the transformed code is semantically equivalent to the original code by running and verifying program output against function specific unit test cases extracted from the third party codebase.

Stage 3: Composition

In the composition stage, the user-in-the-loop input is the order in which the (multiple) eBPF programs for a given interface at different hookpoints. To this end, transformed eBPF programs are chained together using hook point specific mechanisms such as libxdp (for multiple XDP progs) or TC multi-prog (for TC), or using generic tail calls for hook points that do not provide specific mechanisms for program chaining.

Status

Our current prototype is able to transform XDP programs to TC compatible programs and we have validated results on a variety of opensource cobebases viz. xdp tutorial, Mizar, suricata xdp filter and Meta's Katran. The prototype is written in 425 LoC of C++ code. We currently have seven rules for transforming the TC compatible programs. For the largest program, Katran, our tool took ~500ms. One of the many instances of the user-in-the-loop intervention that we observed while running our tool on Katran was: during the extraction phase, our tool identified many functions defined in multiple files and required the user to determine which to keep (e.g., process_packet).

I agree to abide by the anti-harassment policy

Yes

Primary authors: Prof. BENSON, Theophilus (Brown University); Dr KODESWARAN, Palanivel (IBM Research); Dr SEN, Sayandeep (IBM Research)

Presenters: Prof. BENSON, Theophilus (Brown University); Dr KODESWARAN, Palanivel (IBM Research); Dr SEN, Sayandeep (IBM Research)

Session Classification: eBPF & Networking

Track Classification: eBPF & Networking Track