hBPF: A Library and Embedded DSL for BPF Assembly Generation

Andrew Duffy
agd@cs.stanford.edu

Matt Denton
mdenton@cs.stanford.edu

ABSTRACT

The contribution presented by the authors is two-fold. First, we present a set of Haskell data structures that can be linked in sequence and compiled to create BPF assembly. We also provide a high-level API on top of this framework to allow users to write filters that have simple chaining, including simple conjunctions and disjunctions. The final product takes the form of a Haskell embedded domain-specific language.

1. INTRODUCTION

Today's web services need to deal with large amounts of traffic, including floods of malicious or corrupted data. Much of this malicious traffic can be filtered by simple techniques, and because of the large volume of the data, is preferentially done statelessly.

The two most important qualities of a good packet filtering system are (1) speed, and (2) security (these two qualities are often at odds with each other). This is especially important as filtering code can become complex and security errors can become more likely. In addition, the parsing of network data is a common attack surface for malicious packets. And because filtering code must be fast (as every single packet must be parsed and filtered), it is often written in a more unsafe and error-prone language such as C, that neglects such high-level guarantees as bounds-checking of array accesses. Therefore, as much of the traffic can be malicious, it would be ideal to perform this filtering in userspace, out of the privileged kernel context.

Unfortunately, userspace packet filtering comes at a large speed cost, as every single packet (no matter how small) that the kernel collects must be copied across the kernel protection boundary to user-accessible memory. Since most of the benefit of packet filtering comes in high-throughput networking, this makes userland packet filtering mostly implausible for proprietors of modern Internet services.

While much development has recently occurred in making userland filtering possible [2], most filtering is still performed by a Linux and BSD kernel facility known as Berkeley Packet Filters, commonly shortened to *BPF*.

BPF is now more than 20 years old, but is still implemented in modern versions of the Linux and BSD kernels. `libpcap` is the most common way to use BPF, which exposes a very simple syntax that is compiled into BPF bytecode. However, the syntax is written more for ad-hoc analysis than for advanced filtering, and as such its capabilities are often too limited for anything resembling deep packet inspections.

CloudFlare, a major DDoS prevention service with such notable customers as Reddit, Meetup, and the NASDAQ, has demonstrated that they rely on BPF for deep-packet inspection to mitigate the effects of DNS flood attacks, and compile filters written directly in BPF assembly to accomplish this task. They released a toolkit, called bpftools available on GitHub, which is summarily a set of Python scripts that perform search-and-replace over raw blocks of BPF assembly code. It is shocking that after over 20 years of BPF being used by the kernel, an alternative with a better user experience for creating filters does not exist.

The aim for this project is to write a library in Haskell that allows users to write high-level filters with the full power of BPF assembly, allowing easy composability of filters, but with an even simpler syntax, and without constraining the user, giving them the full capabilities of BPF.

1.1 Network Programming and BPF

In traditional network programming, users write programs that utilize system calls provided by the kernel to form stream (TCP) or datagram (UDP) connections. The general sequence of operations are as follows:

1. User requests that the kernel allocate it networking resources (TCP/UDP port number, etc.)
2. The kernel's networking subsystem will find an available port, create a socket listening on that port and pass the socket descriptor back to the user.
3. User performs I/O on the provided socket.

In this form of networking, the user specifies which *connections* for which it will receive packets.

The Linux and BSD kernels extend this functionality, adding a method for user-programmable packet filtering. Here, a user writes a filter program in a specific bytecode format that is interpreted by the kernel. The name of this format is called Berkeley Packet Filters, or BPF for short [1].

BPF has an assembly language on top of it, and there is a *libpcap* DSL syntax that will be familiar to any who have used the Wireshark or `tcpdump` tools. However, for more advanced filter creation, there is little to compare to BPF, as in fact the authors could find no record in the literature or online of a language the compiles to BPF assembly or bytecode save for `libpcap` filters.

The contributions of the authors is two-fold. First, we present a set of Haskell data structures that can be linked in sequence and compiled to create BPF assembly. We also provide a set of high-level combinators on top of this framework to allow users to write filters that have a chaining operation, meant to be used as an embedded domain-specific language in Haskell source code.

We accomplish these tasks through the use of exclusively pure Haskell code, and take advantage of Haskell's clean syntax to make the language as expressive as possible. Specifically, we rely on Haskell's easy function composition and infix operator definition to attempt to make the language as user-friendly as possible.

2. BPF

Before getting into hBPF, we first need to talk about the Berkeley packet filters themselves. This section discusses some of the architectural details of the BPF subsystem as it is used by Linux. Some of these details overlap with BSD as well, but hBPF has only been tested for building and running filters on modern Linux kernels.

2.1 ARCHITECTURE

BPF was originally introduced in 1993 in McCane et al. [1], but has not stopped evolving since then. The original form, which is still supported on the modern Linux kernel, is a register-based in-kernel virtual machine, with an instruction set of 28 instructions and 10 different addressing modes that vary by the instruction class.

BPF is implemented in Linux as a kernel module that filters packets directly in place; i.e. it performs no copy into user or kernel buffers, instead filtering directly in the memory it is placed by the NIC. Thus, no copy operations need to be performed to filter a packet, and no extra memory is consumed. As a corollary, this means all the time consumed by the filter subsystem is solely the time spent interpreting the bytecode for each filter. With a naive interpreter, this can lead to degraded performance once the number of processes submitting filters gets large, as each filter must run sequentially for every packet. There are a few mechanisms that limit this, and ensure termination as well as make some speed guarantees

1. *Maximum Program Size*: Programs are limited to 4096 instructions. This is good for speed, but can limit the functionality of filters that wish to do complex pruning in the kernel, as writers quickly find they will run out of space for instructions.
2. *No Loops*: Relative jumps must have positive offsets, which along with (1) ensures that filters will terminate.
3. *JIT compilation*: On newer versions of the Linux kernel, BPF filters undergo JIT compilation to direct native instruction execution. This reduces the latency associated with interpreting bytecode for every filter over every incoming packet.

Its machine design is register-based, which is more constrained than a stack-based design, but allows a large increase in speed and security. The machine only allows the user's BPF program to access the memory belonging to the packet, by constraining its addressing modes.

Legacy BPF programs can contain memory accesses to one of 16 "memory slots", register loads and stores, ALU instructions performed on the accumulator register, and branching instructions. It also allows for labels similar to other assembly formats.

3. COMPARISON TO TCPDUMP

BPF assembly is not typically written directly by the user. A common way of writing BPF filters is through the user-level utility, `tcpdump`.

The main advantage of `tcpdump` is its easy human-readable syntax; for example, the command

```
tcpdump -i -en0 "ip and udp and dst port 53"
```

clearly filters packets that use the IP and UDP formats, with port 53 as the destination port. As demonstrated above, tcpdump also understands commonly-used protocols and can automatically generate BPF code for those formats. BPF assembly, on the other hand, was designed specifically to be protocol independent, and as such, has no protocol specific support (without extensions). Comparing the tcpdump syntax above to the BPF assembly in the previous section, we can see that not only is the tcpdump syntax shorter, but makes it easy to tell which protocols are being filtered.

However, tcpdump has a number of limitations:

1. *Not Composable*: The libpcap syntax is used mainly by the tcpdump and Wireshark tools, and is meant for ad-hoc filtering and analysis. As such, there is no mechanism for code reuse, and filters for doing all but the simplest analysis begin to look very bloated. hBPF solves this issue by restructuring filter generation as an embedded DSL, with the full power of Haskell for the user to use as much of or as little of as they wish.
2. *No Use of BPF Extensions*: BPF has a set of extensions that vary by the kernel, but these include such things as random number generation and getting the full length of the packet, as well as dropping only part of a packet, or classifying packets based on the return

value of your filter. None of these are supported by the libpcap compiler, and so anyone who wishes to use these features will need to write their filters directly in BPF assembly. While hBPF does not directly support kernel extensions, we make it exceedingly easy to extend the BPF representation to include additions for kernel-specific extensions by structuring assembly generation as a set of high-level Haskell data structures. In theory, any user would be able to add this functionality as part of the hBPF assembly representation.

4. HBPF LIBRARY DESIGN

In this section we review the design of our library, including our DSL for BPF assembly, our method of combining separate filters, and the creation of high-level, protocol specific filters using this DSL.

4.1 BPF Assembly DSL

The initial layer of hBPF is the DSL for the BPF assembly. We encode the BPF assembly in a parsed state, using a hierarchy of algebraic data types. For example, here is the ADT the defines the entire instruction set:

```
data Instr = IInstr BPFInstr | ILabel Label

data BPFInstr =
  -- Load and store instructions
    ILoad NamedRegister LoadType LoadAddress
  | IStore NamedRegister StoreAddress
  -- Arithmetic instructions
  | ILeftShift ArithAddress
  | IRightShift ArithAddress
  | IAnd ArithAddress
  | IOr ArithAddress
  | IMod ArithAddress
  | IMult ArithAddress
  | IDiv ArithAddress
  | IAdd ArithAddress
  | ISub ArithAddress
  -- Conditional/Unconditional Jump instructions
  | IJmp UnconditionalJmpAddress
  | IJAbove UnconditionalJmpAddress
  | IJEq ConditionalJmpAddress
  | IJNotEq ConditionalJmpAddress
  | IJGreater ConditionalJmpAddress
  | IJGreaterEq ConditionalJmpAddress
  -- Misc
  | ISwapAX
  | ISwapXA
  | IRet ReturnAddress
```

There are also a set of sum types that define the supported addressing modes for each instruction class, that have been left out for brevity but can be seen in the GitHub repository for hBPF.

Besides the obvious advantage of easy manipulation when storing the BPF in a parsed state, there is another advantage to this DSL. If the Haskell programmer tries to encode an "illegal instruction," such as trying to return the X register or Jmp to the A register (neither of which are allowed in BPF), it cannot be encoded in our algebraic data types. Thus, if the Haskell program successfully type checks, it is guaranteed to be a syntactically correct BPF program. This may not seem significant, but it helps greatly when developing more complicated filters to get feedback from GHCi instead of the BPF assembler failing silently. Indeed, after much testing, the authors found that no error messages are produced by the `bpf_asm` tool provided by the Linux kernel, and seeing as error messages are incredibly helpful for development, this ends up being a huge improvement over the current state of affairs.

The basic unit of BPF programs in our library is the `Filter`:

```
data Filter = Filter {
    toInstr :: [Instr],
    numLabels :: Int
}
```

`Filter` encapsulates a basic BPF packet filter, with two main restrictions:

1. The filter's return value should not be returned, but instead stored into `M[15]`
2. The filter cannot have meaningful names for labels, as all labels are rewritten with fresh names in a post-processing step called `compile`.

The restrictions on the filter are to provide composability of the filters, as will become clear in the next subsection.

4.2 Combinators

We provide two default combinators for combining the results of separate filters: `&&` and ||, which override their counterparts in the Prelude and should be self-explanatory

```
(||) :: Filter -> Filter -> Filter
(||) = combineRetValues (orI (ArithAddrRegister X))

(&&) :: Filter -> Filter -> Filter
(&&) = combineRetValues (andI (ArithAddrRegister X))
```

The `combineRetValues` function concatenates the following BPF instructions:

1. The instructions of the first filter
2. Two instructions that store `M[15]` into `M[14]`
3. The instructions of the second filter, with the labels fixed so that none of the labels overlap with the labels of the first filters
4. Four instructions that load `M[15]` into A, load `M[14]` into X, combine the two using the "combining" instruction (`orI` for || and `andI` for `&&`), and finally store the result in `M[15]`

Thus, the combinators take Filters that return their results in `M[15]`, and combines them into a Filter that again returns its result in `M[15]`.

So, the first requirement on filters is so that a `ret` statement in the first filter will not prevent the second filter from running. In addition, if the two filters being combined use the same label name, then the BPF program will be rejected as malformed. So, the second requirement is to simplify the fixing of the labels. Label fixing can be done without this requirement, but at the cost of code complexity and poorer performance of our library.

Because the combinator must temporarily store the return value of the first filter whilst running the second, it must store it in the scratch register `M[14]`. This puts a third requirement on the second filter; it cannot use the register `M[14]`. This choice was made to avoid having to write complex register allocation code, but if we were to extend this for production use a less restrictive approach would have to be taken, as disallowing the use of a memory slot when they are so limited could reduce the ability to write fully-functional BPF programs.

4.3 A Little Higher-Level

We do not expect users to form instructions directly, instead providing a set of "smart constructors" that mimic the instructions as they'd be written in assembly:

```
-- * Smart constructors for instructions
ld :: LoadAddress -> [Instr]
ldh :: LoadAddress -> [Instr]
ldx :: LoadAddress -> [Instr]
ldxb :: LoadAddress -> [Instr]
st :: StoreAddress -> [Instr]
stx :: StoreAddress -> [Instr]
andI :: ArithAddress -> [Instr]
sub :: ArithAddress -> [Instr]
jmp :: UnconditionalJmpAddress -> [Instr]
label :: Label -> [Instr]
```

There are many more than this, but just these few are presented for brevity.

There are two notable features of the above scheme, (1) the syntax is meant to resemble that of BPF assembly as much as possible so that those familiar with the assembly language can easily be productive writing in a typed environment, and (2) all smart constructors return type `[Instr]`. The latter was done to allow for use of the existing `Monoid` instance on lists for increased simplicity of syntax. For example, a snippet that checks if a packet is an IPv4 packet would look like the following:

```
let isIPv4 = ldh (LoadAddrByteOff 0xb)
          <> jneq 0x800 "failure"
          <> ret #1
          <> label "failure"
          <> ret #0
```

4.4 Instances and GHC Extensions

To ensure type-safety and static guarantees on produced filters, we needed to enhance the type system to distinguish between what would be otherwise identical types for numbers and strings.

The hBPF language defines `newtype`s for some important primitives to ensure that numbers and strings do not take on the same types:

```
-- | Literal: type for numeric literals
-- | i.e. "immediate values"
newtype Literal = Literal Int
    deriving (Eq, Show, Num)


-- | Offset: byte offsets as used in
-- |instructions like 'ld' and 'jeq'
newtype Offset = Offset Int
    deriving (Eq, Show, Num)

-- | Label type
newtype Label = Label String
    deriving (Eq, Show, IsString)
```

Note that all three `newtype`s derive some non-standard instances, thanks to use of `-XGeneralizedNewtypeDeriving`.

Both `Literal` and `Offset` derive instances of `Num`, which means that you'd be able to use them anywhere a `Num` instance is expected, and similarly you can use numeric literals when writing code and they will automatically be converted to `Literal` or `Offset` based on the context in which they're placed. We saw use of this in the previous example for `isIPv4`, where `0x800` was used as a `Literal` without explicitly needing to wrap it with the `Literal` constructor.

Another extension we make use of is `-XOverloadedStrings` for labels. Note that `Label` derives and instance of `IsString`, also making it possible to use string literals in positions where a `Label` is expected.

We generalize compilation of instructions into the `StringRep` type class, defining a `toString` method that yields the `String` representation of the instruction. `BPFInstr` has an instance of `StringRep`, and `[Instr]` simply maps `toString` over the elements of the list and concatenates them together to form the full assembly code.

We provide a `compile` method that performs `toString` after adding an "epilogue", which is responsible for adding a pair of instructions at the end of the filter which will load the return value from `M[15]` into `A` and then return the value in the `A` register.

4.5 Writing and Distributing New Filters

Currently hBPF is an implementation of the BPF assembly format as Haskell data structures, and all creation of filters is left to developers. This includes creating libraries for "snippets" (as we like to call them) of instructions that perform meaningful work on a packet with a specific protocol.

For example, if someone wished to use hBPF in creating an HTTP inspecting filter, they'd likely want to support ways to

1. The the request URL from an incoming HTTP Request
2. Check if a certain header is contained in the request
3. Determine the size of the HTTP payload
4. Get the value of a specific cookie in a request header

A snippet developer will need some knowledge of BPF assembly to create these initial functions, but they have the help of the Haskell type system to guide them in what parameters to use for the smart constructors, as opposed to "flying blind" by writing and compiling pure BPF assembly.

For example, this is a simple filter that checks the TCP (or UDP) port number:

```
isPort :: Int -> Filter
isPort port = newFilter $
     ldx (LoadAddrLiteral 0)
  <> stx (StoreAddrIndexedRegister M15)
  <> ldxb (LoadAddrNibble ipPacketLengthField)
    -- skip IP header
  <> ldh (LoadAddrOffFromX sourcePortOffset)
  <> jneq (ConditionalJmpAddrLiteralTrue
           port "label1")

  <> ldh (LoadAddrOffFromX destPortOffset)
  <> jneq (ConditionalJmpAddrLiteralTrue
           port "label1")
  <> ldx (LoadAddrLiteral 65535)
  <> stx (StoreAddrIndexedRegister M15)
  <> label "label1"
```

While this may not seem any better than writing in BPF assembly directly, keep in mind that (1) using hBPF provides a typed interface above BPF assembly, which is safer, and (2) once someone writes a filter for a specific protocol, they can then publish that as a module on Hackage to allow for easy distribution. Currently there is no story around distribution of BPF filters, aside from posting the raw assembly on a site such as Pastebin, emailing or uploading the filter to GitHub. Using the existing Haskell infrastructure for package management could lead to more people adopting and using this method of deep-packet inspecting filters. Thus, we see a future where corporations that need deep packet filtering for availability guarantees such as CloudFlare can create new type BPF filters as well as use existing protocol-specific filters developed by the larger Haskell community.

5. IMPROVEMENTS AND FUTURE WORK

While hBPF is a big improvement over the current state of affairs when it comes to advanced packet filtering, there is still a tremendous amount of work to be done before it can serve as a complete replacement. Some of the improvements envisioned by the authors are outlined below.

5.1 Language Improvements

The language, while better and safer than writing BPF filters directly, still gets close enough to the metal that it might make some turn away. While this version of the language is targeted at those who need to write deep packet filtering and have no way to do so except in raw BPF assembly, we see a future where users can write in an even higher-level of abstraction. Some ways to improve the level of abstraction is to export a higher-level interface, possibly by creating a more deeply embedded language over the assembly generation mechanism. For example, it is imaginable that people would want to write code along the lines of *bind variable X to the word at packet location 12, then use X in several other places.* While we did not include the use of variable bindings in this version of the language, they are a powerful abstraction tool that allow us to forget about loads and stores, at the expensive of more complexity in the hBPF backend.

5.2 Optimizations

There are a number of opportunities for optimization given that the library has access to the entire AST, including constant folding and other arithmetic reductions, as well as more advanced string-matching mechanisms that actually increase the cardinality of the set of programs that can be written by merging code paths together.

5.2.1 Arithmetic Optimizations

Since we have the entire parse tree available to us at compilation time, we can reduce certain arithmetic expressions by noting which memory slots are guaranteed to hold specific numbers, and then inlining those numbers where that memory slot was otherwise used, for example. The benefit of this is relatively small, though, as people rarely seem to write complicated arithmetic expressions in their assembly code.

5.2.2 Prefix Tree Merging for String Matching

One common operation that needs to be performed in deep packet inspection involves checking for the presence of one or several of a set of strings. For example, in its bpftools, CloudFlare examines incoming DNS packets, dropping any that are doing a lookup on a specific set of hostnames. In BPF, there is no string matching instruction, this must be done by sequentially loading bytes into registers, and checking them for equality to bytes at specific indexes in the string. Disjunctions and conjunctions must be performed sequentially, meaning that the number of instructions generated is proportional to the number of bytes taken by every single string, which can quickly approach the 4096 instruction limit as your number of match strings increases.

To mitigate the effects of this and allow for a larger set of search strings, it would be useful to provide a

`match :: [String] -> Filter` combinator, which takes a set of strings and forms a prefix-tree of them. Now, any redundancy in the strings can be used to compress the number of output instructions, which allows for an increase in the number of strings we can include in the matching set. This we foresee as potentially the most important optimization that hBPF does not currently contain, but we are looking to add this feature in a future release.

6. ACKNOWLEDGEMENTS

We'd like to thank David Mazières, Bryan O'Sullivan, David Terei, and Riad S. Wahby for their knowledgeable instruction of Haskell, particularly David Terei for discussing with us routes for the final project and putting BPF on our radar.

CloudFlare's BPFTools can be found **at this link**, and the Linux kernel's documentation of BPF that we referred to for our implementation of the BPF assembly representation and compiler can be found **here**.

7. REFERENCES

[1] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[2] L. Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.