Contributing to bpftrace

A practical guide to bpftrace internals

Dale Hamel

08/12/19 11:57:44 PM UTC

Contents

This document is also available in epub and pdf format if you prefer.

bpftrace

What drew me to bpftrace was an early post [1] about it by Brendan Gregg after it first got pulled into the iovisor org, from Alistair Robinson's [2] initial repository.

It was a pretty cool project! I had been playing around with bcc for a few months, trying to use it to trace network issues we were experiencing in our Kubernetes clusters, and see if eBPF could be used to help pin down other problems as well, including tracing the containers inside our Kubernetes pods.

I saw right away that a lot of the bcc scripts had been re-implemented in bpftrace scripts - cool! It was amazing to me to see that large amalgams of python and inline C code could be simplified down to just a few lines (or in some cases, getting most of the functionality from one single line!) of bpftrace.

I saw from Brendan's post that [1] bpftrace's syntax was easily translatable from dtrace, they were nearly on par with one another. I had heard of dtrace before, from some debugging my colleague Burke Libbey had done in our development environments on Darwin (OS X). It baffled me that such a powerful tool existed bundled into ever one of our laptops, but that there was no such analogous tool for Linux.

I had been following kernel tracing since 2013, as I had been trying (unsuccessfully) to implement Systemtap support into a side-project of mine, and find other frameworks to work with kprobes and uprobes in production. The overhead systemtap probes was a bit scary though, and it required loaded a new Kernel module, which wouldn't work on Chromium OS derivatives for security reasons, so wasn't viable for production use.

eBPF promised an intriguing new frontier - a way to access the kernel's uprobe and kprobe API, **built-in** to the linux kernel, and on top of the existing BPF infrastructure already employed for enabling tcpdump from kernel space. Unlike Systemtap, eBPF would work without issue on Chromium OS derivatives, and so was more viable for production usage.

Contributions

This is the best type of problem to solve - one you have yourself, as you get to act as your own QA / customer, and benefit directly from the new feature. To add to this, you can benefit from expert code reviews you wouldn't otherwise have access to, and learn to collaborate with people all over the world.

Gaining Context for first Contribution

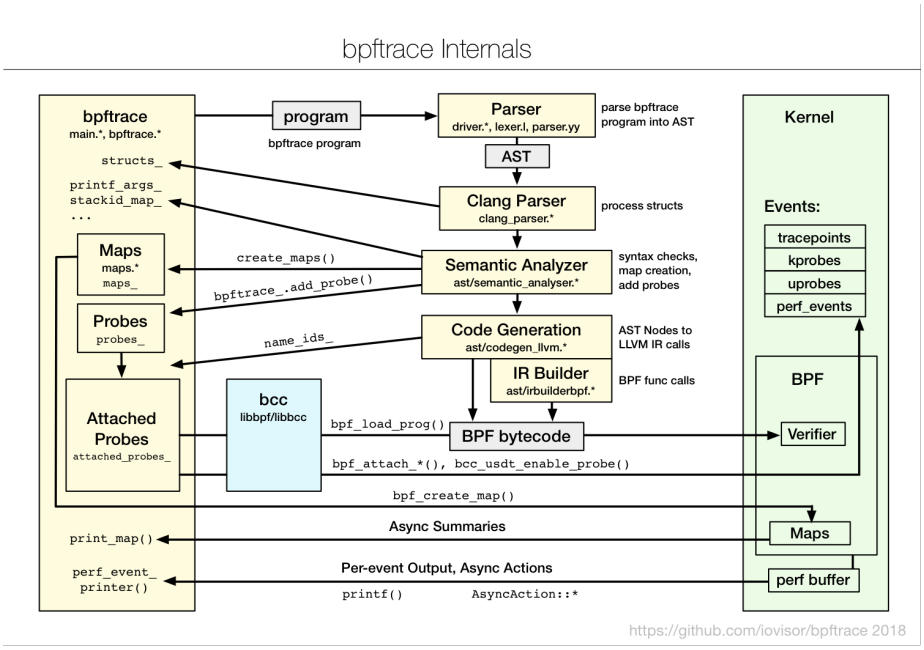
The first problem I encountered was that while a number of the other bcc scripts had been ported to bpftrace, there were none of the tcp tools available.

This was disappointing, as network tracing was one of the main use cases I had for eBPF. From Brendan Gregg's 2017 Velocity talk [3] , which I actually had the pleasure of attending in person, I learned that eBPF was able to provide a lot of the value of tcpdump, without instrumenting `send` or `receive`, but my being much more targetted on specific types of interesting events (or tracepoints).

If I was really going to use bpftrace, it would need to have at least the same utility for debugging network issues as the existing bcc tools did. So, I went ahead and filed an issue [4] for this, then looked to what would be needed in order to solve this myself. As a rule, I try to only submit an issue if I have some idea for how **I** might implement it, or some intention of being the one to own it and see it fixed.

I read through bpftrace's internals development docs, which gave an overview of how bpftrace uses LLVM to generate eBPF. Basically, it revealed how bpftrace uses the LLVM framework for code generation, that ultimately become compiled eBPF instructions that can be loaded into the Linux kernel. Cool!

There were even some examples that walked through how other functionality was added to bpftrace, so I could use these as a practical reference point and a "reverse-engineering" manual, to see how I could use the same concepts to implement this for myself.


bpftrace Internals
bpftrace
main.*, bpftrace.*
structs_
printf_args_
stackid_map_
...
Maps
maps.*
maps_
Probes
probes_
Attached Probes
attached_probes_
program
bpftrace program
Parser
driver.*, lexer.l, parser.yy
parse bpftrace program into AST
AST
Clang Parser
clang_parser.*
process structs
create_maps()
bpftrace_.add_probe()
Semantic Analyzer
ast/semantic_analyser.*
syntax checks, map creation, add probes
name_ids_
Code Generation
ast/codegen_llvm.*
AST Nodes to LLVM IR calls
IR Builder
ast/irbuilderbpf.*
BPF func calls
bcc
libbpf/libbcc
bpf_load_prog()
BPF bytecode
bpf_attach_*(), bcc_usdt_enable_probe()
bpf_create_map()
Async Summaries
print_map()
perf_event_ printer()
Per-event Output, Async Actions
printf()
AsyncAction::*
Kernel
Events:
tracepoints
kprobes
uprobes
perf_events
BPF
Verifier
Maps
perf buffer
https://github.com/iovisor/bpftrace 2018

[5]

Adding ntop to bpftrace

In order to debug network traffic and implement tcp tracing scripts for bpftrace, the main thing I knew was missing was the ability to print IP addresses. I decided to tackle the existing issue for implementing ntop [6]. So I set to work on this, which was my first introduction to bpftrace, and got to work on [7].

Lexer and Parser

bpftrace uses yacc to formally describe its syntax and the symbols of the language, which is used to parse the bpftrace AST.

To add a new builtin function, we rely on the `call` type of expression:

```
expr : INT             { $$ = new ast::Integer($1); }
     | STRING          { $$ = new ast::String($1); }
     | BUILTIN         { $$ = new ast::Builtin($1); }
     | ternary         { $$ = $1; }
     | map             { $$ = $1; }
     | var             { $$ = $1; }
     | call            { $$ = $1; }
```

Which we see will be the treatment for when an expression uses paretheses, accepting a variable size argument list:

```
call : ident "(" ")"       { $$ = new ast::Call($1); }
     | ident "(" vargs ")" { $$ = new ast::Call($1, $3); }
     ;
```

When the parser encounters a function call then, it passes the identity of the function (the identifier that preceded the parentheses) as the first argument, then the argument list (the third parser token) as the second argument to a new `Call` AST node. As for the variable argument list, we can see it is just an AST node for an `ExpressionList`:

```
vargs : vargs "," expr { $$ = $1; $1->push_back($3); }
      | expr           { $$ = new ast::ExpressionList;
↪  $$->push_back($1); }
      ;
```

So all this to say, as long as the caller follows these syntatic conventions, a new Call node should be created:

```
class Call : public Expression {
public:
  explicit Call(std::string &func) : func(func), vargs(nullptr) {
↪  }
  Call(std::string &func, ExpressionList *vargs) : func(func),
↪  vargs(vargs) { }
  std::string func;
  ExpressionList *vargs;

  void accept(Visitor &v) override;
};
```

This shows us that initializer lists are used to set the `func` and `vargs` members of this class, so that these values can be accessed by the semantic analyser to decide how to handle this.

Semantic Analyser

As shown above, the parser doesn't actually have any idea if a function is actually implemented. That is up to the semantic analyser to decide, and also perform validations to ensure that the semantics of the parsed AST make sense.

When the AST node is visited, during semantic analyses, the type of the node being a `Call` node will result in the dynamic method lookup feature of C++ to call:

```
void SemanticAnalyser::visit(Call &call)
```

This function will ensure that the vargs are visited first, if provided:

```
  if (call.vargs) {
    for (Expression *expr : *call.vargs) {
      expr->accept(*this);
    }
  }
```

Then the rest of the function is basically a big `switch` on `call.func`. In order

to handle a new function, we just have to add to the if ladder a case for it:

```
  else if (call.func == "ntop") {
    check_nargs(call, 2);
    check_arg(call, Type::integer, 0);
    check_arg(call, Type::integer, 1);
    // 3 x 64 bit words are needed, hence 24 bytes
    //  0 - To store address family, but stay word-aligned
    //  1 - To store ipv4 or first half of ipv6
    //  2 - Reserved for future use, to store remainder of ipv6
    call.type = SizedType(Type::inet, 24);
  }
```

This describes what the function signature ought to be, and asserts as much by examining the parsed AST node.

It also sets `call.type`, which can be thought of like the "return type" for this builtin function. We can see that a new type, `Type::inet`, is added, that indicates what is returned from this function.

While this PR only implemented 4 byte IPv4 addresses, it specifies specifies that the arguments it takes will be in 3 x 64 bit array, so that it knows how to access both IPv4 and, in the future, IPv6 addresses.

Code Generation

Now that the function has been described, I had to implement the code to that would actually get compiled to eBPF. This is a sort of meta-programming, where you call LLVM functions to build LLVM IR. The level of this programming is very much like assembly, but genericized. LLVM optimizes this IR (intermediate representation), to output the resulting eBPF probes that con be loaded into the kernel.

```
  else if (call.func == "ntop")
  {
    // store uint64_t[2] with: [0]: (uint64_t)address_family,
    ↪  [1]: (uint64_t) inet_address
    // To support ipv6, [2] should be the remaining bytes of the
    ↪  address
    AllocaInst *buf = b_.CreateAllocaBPF(call.type, "inet");
    b_.CreateMemSet(buf, b_.getInt8(0), call.type.size, 1);
    Value *af_offset = b_.CreateGEP(buf, b_.getInt64(0));
    Value *inet_offset = b_.CreateGEP(buf, {b_.getInt64(0),
↪   b_.getInt64(8)});
    call.vargs->at(0)->accept(*this);
    b_.CreateStore(expr_, af_offset);
```

```
    call.vargs->at(1)->accept(*this);
    b_.CreateStore(expr_, inet_offset);
    expr_ = buf;
  }
```

We declare the allocation with `CreateAllocaBPF` noting the `inet` type:

```
    // store uint64_t[2] with: [0]: (uint64_t)address_family,
    ↪  [1]: (uint64_t) inet_address
    // To support ipv6, [2] should be the remaining bytes of the
    ↪  address
    AllocaInst *buf = b_.CreateAllocaBPF(call.type, "inet");
```

Then we actually inject a `memset` instruction, to actually allocate an array of the specified size of a 3 x 64 bit array (24 bytes):

```
    // store uint64_t[2] with: [0]: (uint64_t)address_family,
    ↪  [1]: (uint64_t) inet_address
    // To support ipv6, [2] should be the remaining bytes of the
    ↪  address
    AllocaInst *buf = b_.CreateAllocaBPF(call.type, "inet");
```

Next, we want to be able to reference specific values within this array as we described in the signature of the semantic analyser:

```
    Value *af_offset = b_.CreateGEP(buf, b_.getInt64(0));
    Value *inet_offset = b_.CreateGEP(buf, {b_.getInt64(0),
↪   b_.getInt64(8)});
```

This command, `CreateGEP` is like asking to get the pointer within the above allocated array buffer. We can see that since the `af_offset` is 0, it is the first byte. To get the IPv4 address, we get the value in the second byte, by offsetting the first 8 bytes we already read, and get this `inet_offset`.

To actually read these values, we visit the code generation for the first arg, at which point we know that the the global `expr_` has been set by the accept call to the type of the AST node for the first argument.

```
    call.vargs->at(0)->accept(*this);
    b_.CreateStore(expr_, af_offset);
```

Since we validated that this must be of `Type::integer` above in the semantic analyzer, we can see that this is the function that will get called and how it sets `expr_` to what we want to store:

```
void CodegenLLVM::visit(Integer &integer)
{
  expr_ = b_.getInt64(integer.n);
```

```
}
```

Then, this call to `CreateStore` says to copy the data from the address of the first argument, to the location we pointed to for `af_offset` above. This is actually storing the value in our buffer:

```
    b_.CreateStore(expr_, af_offset);
```

We repeat this process to also get the IPv4 address:

```
    call.vargs->at(1)->accept(*this);
    b_.CreateStore(expr_, inet_offset);
```

Then, following the same convention, we set `expr_` to be the 3 x 64 bit buffer that we allocated, in a sense, "returning" it in the way that the `Type::integer` was "returned".

Some parts of bpftrace also need to be made aware that rather than "returning" a single, 64 bit value, it is returning an "array" of these, and so should be treated like other "arrays", such as `usym` and `string` types in bpftrace:

```
    for (Expression *expr : *map.vargs) {
      expr->accept(*this);
      Value *offset_val = b_.CreateGEP(key, {b_.getInt64(0),
↪   b_.getInt64(offset)});
      if (expr->type.type == Type::string || expr->type.type ==
      ↪  Type::usym ||
        expr->type.type == Type::inet)
        b_.CREATE_MEMCPY(offset_val, expr_, expr->type.size, 1);
      else
        b_.CreateStore(expr_, offset_val);
```

```
    for (Expression *expr : *map.vargs) {
      expr->accept(*this);
      Value *offset_val = b_.CreateGEP(key, {b_.getInt64(0),
↪   b_.getInt64(offset)});
      if (expr->type.type == Type::string || expr->type.type ==
      ↪  Type::usym ||
        expr->type.type == Type::inet)
        b_.CREATE_MEMCPY(offset_val, expr_, expr->type.size, 1);
      else
        b_.CreateStore(expr_, offset_val);
```

Calling inet_ntop

Now that all the work has been done to prepare the necessary bytes by allocating some space in a predictable schema for the arguments, the actually call to `inet_ntop` can be made!

A new function is implemented, that accepts this 3 x 64 bit array we have described in the semantic analyzer, and implemented in LLVM code generation:
‘

```
std::string BPFtrace::resolve_inet(int af, uint64_t inet)
{

  // FIXME ipv6 is a 128 bit type as an array, how to pass as
  ↪  argument?
  if(af != AF_INET)
  {
    std::cerr << "ntop() currently only supports AF_INET (IPv4);
    ↪  IPv6 will be supported in the future." << std::endl;
    return std::string("");
  }
  char addr_cstr[INET_ADDRSTRLEN];
  inet_ntop(af, &inet, addr_cstr, INET_ADDRSTRLEN);
  std::string addrstr(addr_cstr);
  return addrstr;
}
```

This function will be called each time a new IPv4 address needs to be parsed. It indicates that IPv6 addresses are not yet supported, but if the `AF_INET` type is as expected, it will parse the address using the library function `inet_ntop`, then return the string representation of this value.

That concludes the work that was done to get support IPv4 network addresses! As this was my main (and probably the main) use-case, IPv6 support could come later.

tcp tools enabled by ntop support

I was able to use this functionality to port over IPv4 versions of the missing tcp tools! Exactly what I had wanted to do. Using basically the same approach, I was able to implement simple versions of:

- tcpaccept.bt: a bpftrace version of tcpaccept.py, which traces when a a new tcp connection has been accepted, great for checking what is connecting to a server.

- tcpconnect.bt: a bpftrace version of tcpconnect.py, which traces outgoing tcp connections, to see what a server is connection to for outbound connections.
- tcpdrop.bt: a bpftrace version of tcpdrop.py, which traces when tcp packets are dropped by the linux kernel for some reason or another, useful for checking if the network is behaving reliably.
- tcpretrans.bt: a bpftrace version of tcpretrans.py, which traces when tcp packets need to be retransmitted by the linux kernel, useful for checking if the network is behaving reliably.

These all hooked into the same places their bcc counterparts did, and some were limited by the current functionality of bpftrace at the time, as the header parsing was limited in what could be accessed via dynamic tracing, and many tcp tracepoints weren't widely available yet.

Many of these scripts didn't adhere to best practices for style, and others have graciously improved on them and kept them up-to-date with the other bpftrace scripts, ensuring they adhere to bpftrace idioms.

// FIXME attribution to these fine folks

Improvements ntop

The glaring gap of my contribution was that it didn't handle IPv6 addresses, because at the time there was no way to access this data, as the struct parsing of bpftrace was limited in determining the offsets of IPv6 fields.

This is not an issue when using later kernel versions, which can use the IP tracepoint API to avoid this problem altogether. Support was then added by Matheus Marchini [8] to refactor this to handle IPv6 addresses [9] and it is instructive to examine these changes.

I did not actually do this work, but I followed it and felt impelled to describe the improvements Matheus made, as I also reviewed his pull request.

Semantic analyser

In the semantic analyser, Matheus updated the implementation:

```
else if (call.func == "ntop") {
  if (!check_varargs(call, 1, 2))
    return;

  auto arg = call.vargs->at(0);
  if (call.vargs->size() == 2) {
    arg = call.vargs->at(1);
    check_arg(call, Type::integer, 0);
  }

  if (arg->type.type != Type::integer && arg->type.type !=
  ↪  Type::array)
    err_ << call.func << "() expects an integer or array
    ↪  argument, got " << arg->type.type << std::endl;

  // Kind of:
  //
```

```
    // struct {
    //   int af_type;
    //   union {
    //     char[4] inet4;
    //     char[16] inet6;
    //   }
    // }
    int buffer_size = 8;
    if (arg->type.type == Type::array) {
      if (arg->type.elem_type != Type::integer ||
       ↪  arg->type.pointee_size != 1 || !(arg->type.size == 4 ||
       ↪  arg->type.size == 16)) {
        err_ << call.func << "() invalid array" << std::endl;
      }
      if (arg->type.size == 16)
        buffer_size = 20;
    }
    call.type = SizedType(Type::inet, buffer_size);
    call.type.is_internal = true;
  }
```

Note that it better describes the storage of the argument as a struct, and how the bytes are intended to be used to describe the union of the addresses.

He also allowed for different argument types to be accepted, and for the buffer size to be used to be specified dynamically.

This allows for the signature of ntop to be more complicated, with default and variable number arguments.

Code generation

The new implementation of the code generation must correspondingly handle the new cases, for both IPv4 and IPv6 addresses.

```
  else if (call.func == "ntop")
  {
    // struct {
    //   int af_type;
    //   union {
    //     char[4] inet4;
    //     char[16] inet6;
    //   }
    // }
    AllocaInst *buf = b_.CreateAllocaBPF(call.type, "inet");
```

```
    Value *af_offset = b_.CreateGEP(buf, b_.getInt64(0));
    Value *af_type;

    auto inet = call.vargs->at(0);
    if (call.vargs->size() == 1) {
      if (inet->type.type == Type::integer || inet->type.size ==
      ↪  4) {
        af_type = b_.getInt32(AF_INET);
      } else {
        af_type = b_.getInt32(AF_INET6);
      }
    } else {
      inet = call.vargs->at(1);
      call.vargs->at(0)->accept(*this);
      af_type = b_.CreateIntCast(expr_, b_.getInt32Ty(), true);
    }
    b_.CreateStore(af_type, af_offset);

    Value *inet_offset = b_.CreateGEP(buf, {b_.getInt64(0),
↪   b_.getInt64(4)});

    inet->accept(*this);
    if (inet->type.type == Type::array) {
      b_.CreateProbeRead(reinterpret_cast<AllocaInst
      ↪  *>(inet_offset), inet->type.size, expr_);
    } else {
      b_.CreateStore(b_.CreateIntCast(expr_, b_.getInt32Ty(),
      ↪  false), inet_offset);
    }

    expr_ = buf;
  }
```

He updated the integer type to support 128 bit types, as would be needed for IPv6, meaning that if a 128 bit integer could be read (such as by a tracepoint) it could hold a literal IPv6 value!

```
    switch (stype.size)
    {
      case 16:
        ty = getInt128Ty();
        break;
```

This was a trick I had discovered too, but hadn't committed as I couldn't find a good case for supporting 128 bit ints generically in bpftrace. // FIXME link

issue.

Since an array can also be passed, we determine if the array holds 4 bytes (IPv4) or 16 bytes (IPv6). We also use 4 bytes to store AF_INET. This is why we either have a buffer size of 4 bytes (AF_INET) + 4 bytes (IPv4) = 8 bytes for IPv4 addresses, or 4 bytes (AF_INET) + 16 bytes (IPv6) = 20 bytes for IPv6 addresses.

We accordingly make the `af_inet` determination based on the buffer size and number of arguments passed:

```
    auto inet = call.vargs->at(0);
    if (call.vargs->size() == 1) {
      if (inet->type.type == Type::integer || inet->type.size ==
      ↪  4) {
        af_type = b_.getInt32(AF_INET);
      } else {
        af_type = b_.getInt32(AF_INET6);
      }
    } else {
      inet = call.vargs->at(1);
      call.vargs->at(0)->accept(*this);
      af_type = b_.CreateIntCast(expr_, b_.getInt32Ty(), true);
    }
    b_.CreateStore(af_type, af_offset);
```

Since the address is always after the first 4 bytes, we can just read the rest of the bytes handling both the literal integer, and array cases:

```
    Value *inet_offset = b_.CreateGEP(buf, {b_.getInt64(0),
↪   b_.getInt64(4)});

    inet->accept(*this);
    if (inet->type.type == Type::array) {
      b_.CreateProbeRead(reinterpret_cast<AllocaInst
      ↪  *>(inet_offset), inet->type.size, expr_);
    } else {
      b_.CreateStore(b_.CreateIntCast(expr_, b_.getInt32Ty(),
      ↪  false), inet_offset);
    }
```

Calling inet_ntop

Now that the codegen is implemented, we have to update our helper to support calling the `inet_ntop` for both `AF_INET` and `AF_INET6` address types:

```
std::string BPFtrace::resolve_inet(int af, uint8_t* inet)
{
  std::string addrstr;
  switch (af) {
    case AF_INET:
      addrstr = resolve_inetv4(inet);
      break;
    case AF_INET6:
      addrstr = resolve_inetv6(inet);
      break;
    default:
    std::cerr << "ntop() got unsupported AF type: " << af <<
    ↪  std::endl;
    addrstr = std::string("");
  }
```

Which then calls either the IPv4 or IPv6 static / private versions accordingly:

```
static std::string resolve_inetv4(uint8_t* inet) {
  char addr_cstr[INET_ADDRSTRLEN];
  inet_ntop(AF_INET, inet, addr_cstr, INET_ADDRSTRLEN);
  return std::string(addr_cstr);
}


static std::string resolve_inetv6(uint8_t* inet) {
  char addr_cstr[INET6_ADDRSTRLEN];
  inet_ntop(AF_INET6, inet, addr_cstr, INET6_ADDRSTRLEN);
  return std::string(addr_cstr);
}
```

[1] B. Gregg, "bpftrace (DTrace 2.0) for Linux 2018." [Online]. Available: http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html

[2] A. Robertson, "Alastair Robertson's Homepage." [Online]. Available: https://ajor.co.uk/projects/

[3] B. Gregg, "Brendan Gregg 2017 Velocity talk." [Online]. Available: https://www.slideshare.net/brendangregg/velocity-2017-performance-analysis-superpowers-with-linux-ebpf

[4] D. Hamel, "Bpftrace Network Tracing Limitations Issue." [Online]. Available: https://github.com/iovisor/bpftrace/issues/245

[5] "bpftrace internals_development.md doc." [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/docs/internals_development.md

[6] B. Gregg, "Bpftrace add inet_ntop() Issue." [Online]. Available: https:

//github.com/iovisor/bpftrace/issues/30

[7] D. Hamel, “Initial inet_ntop implementation.” [Online]. Available: https://github.com/iovisor/bpftrace/pull/269

[8] M. Marchini, “Matheus Marchini’s homepage.” [Online]. Available: https://mmarchini.me/

[9] M. Marchini, “ntop add support for arrays and IPv6.” [Online]. Available: https://github.com/iovisor/bpftrace/pull/566