Running Python in the Linux Kernel with bpf

Alex Gartrell

Software Engineer @ Facebook

Agenda

1. Who am I?
2. bpf Overview?
3. Python Virtual Machine
4. Transpiling from python to bpf bytecode
5. Woo, we made it!

Agenda

1. **Who am I?**
2. bpf Overview?
3. Python Virtual Machine
4. Transpiling from python to bpf bytecode
5. Woo, we made it!

Who am I?

Who am I?

- Alex Gartrell

Who am I?

- Alex Gartrell
- Software Engineer @ Facebook

Who am I?

- Alex Gartrell
- Software Engineer @ Facebook
- Work in Infrastructure for ~6 years
  - Caching Infrastructure
  - Content Distribution Network
  - Linux Kernel Networking Stack
  - Linux Containers
  - Network Block Devices

Who am I?

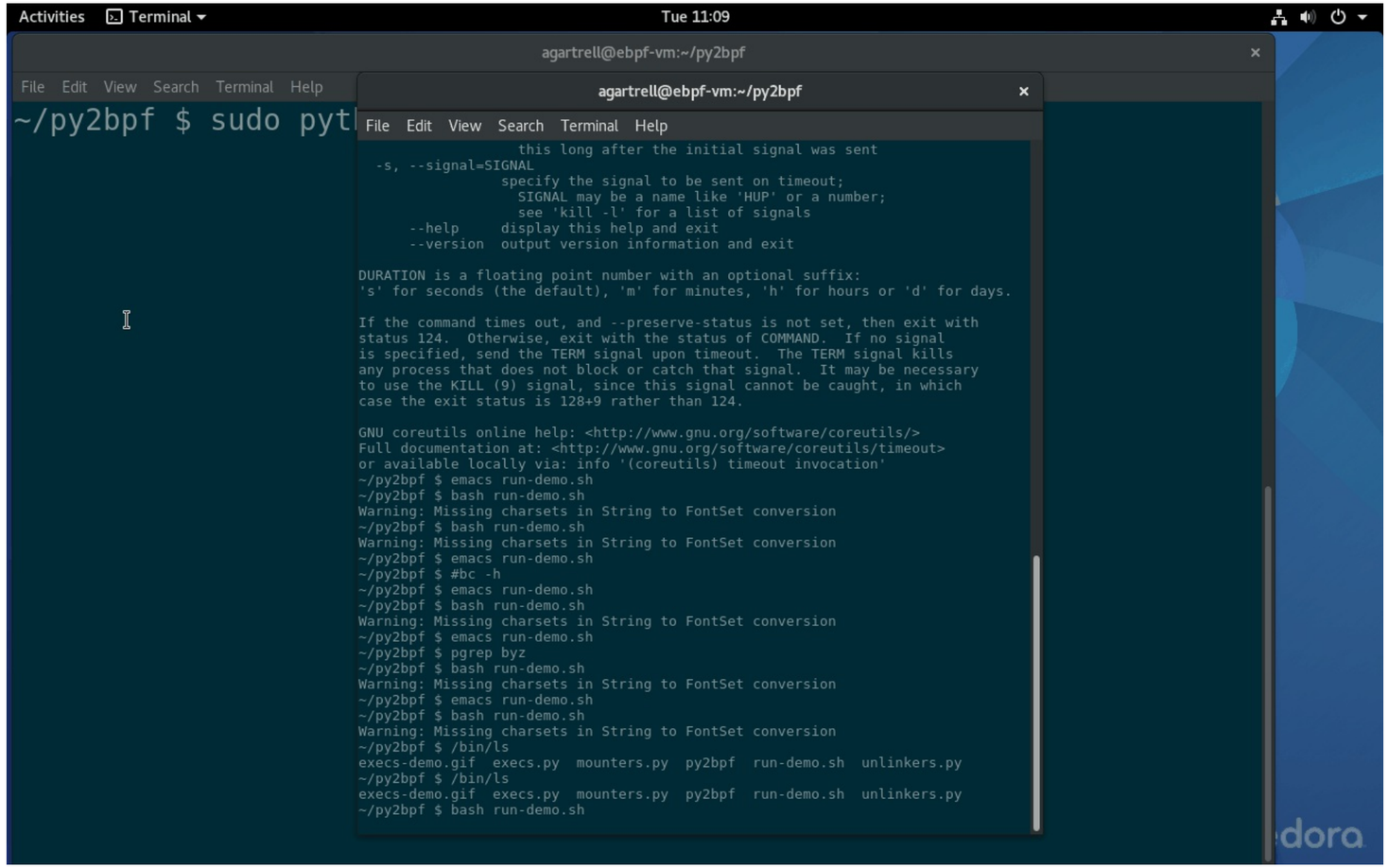
- Alex Gartrell
- Software Engineer @ Facebook
- Work in Infrastructure for ~6 years
  - Caching Infrastructure
  - Content Distribution Network
  - Linux Kernel Networking Stack
  - Linux Containers
  - Network Block Devices
- I am **not** a compilers expert

Agenda

1. Who am I?
2. **Overview**
3. Python Virtual Machine
4. Transpiling from python to bpf bytecode
5. Woo, we made it!

Let's watch programs start

Let's watch programs start



How are we doing that?

How are we doing that?

```
int sys_execve(filename, argv) {
   // Execute a program!
   ...
}
```

How are we doing that?

```
int sys_execve(filename, argv) {
   goto my_new_tracepoint;
   // Execute a program!
   ...
}
```

How are we doing that?

How are we doing that?

```
with ExecveProbe() as probe:
    for call in probe.queue:
        print(
            call.pid,
            call.comm.decode(),
            call.arg0.decode(),
            call.arg1.decode(),
            call.arg2.decode(),
            call.arg3.decode())
```

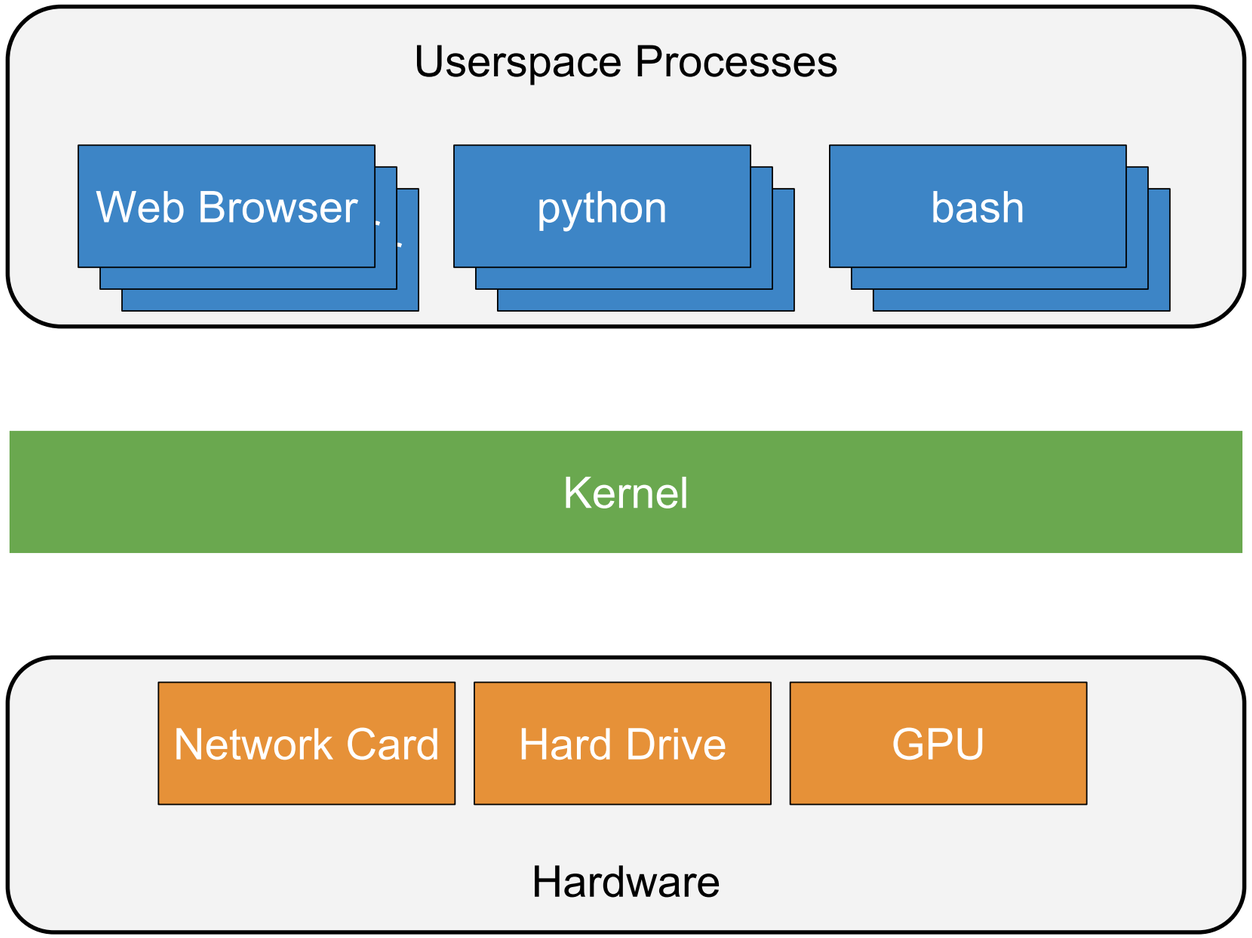
But what does the probe look like?

But what does the probe look like?

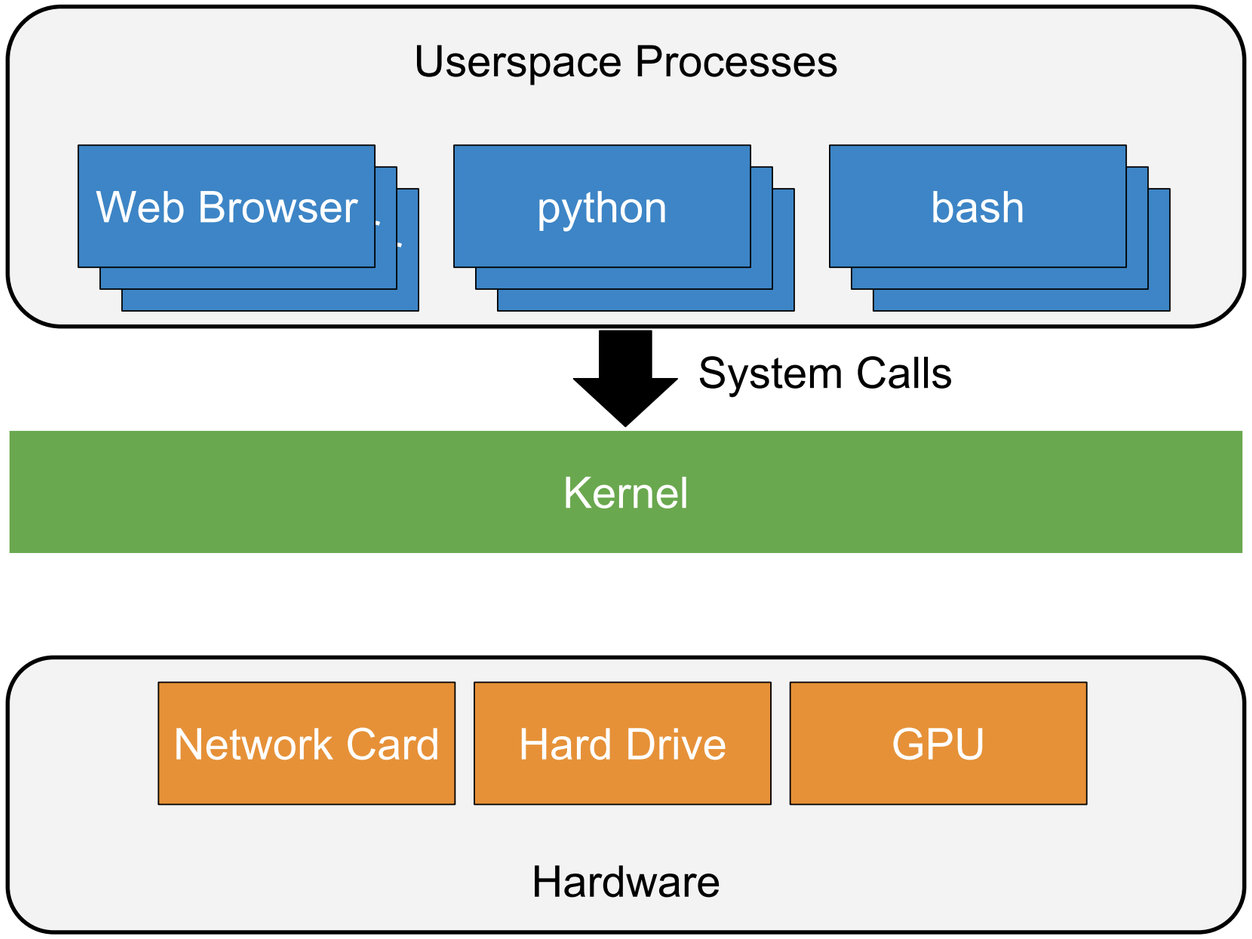
```
def fn(pt_regs):
    call = Call()
    call.pid = funcs.get_current_pid_tgid() & 0xffffffff
    funcs.get_current_comm(call.comm)

    arg = ctypes.c_int64()
    addrof_arg = funcs.addrof(arg)

    funcs.probe_read(addrof_arg, pt_regs.rsi)
    if arg != 0: # Read argv[0]
        funcs.probe_read(call.arg1, arg)
        funcs.probe_read(addrof_arg, pt_regs.rsi + 8)

    ...

    funcs.perf_event_output(pt_regs, q, 0, call)

    return 0
```

What is the kernel?

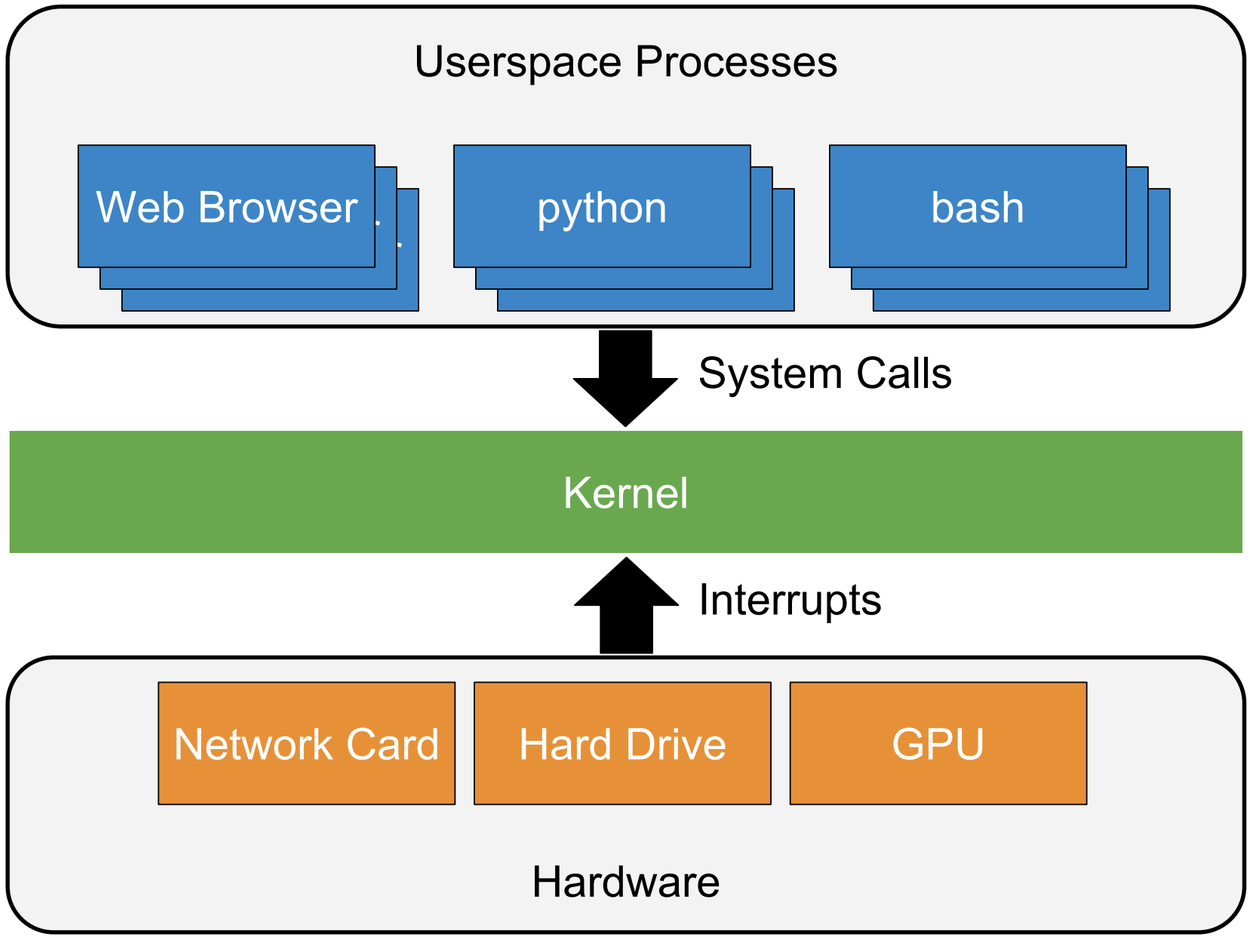
What is the kernel?



What is the kernel?



What is the kernel?



Constraints of the kernel environment

Constraints of the kernel environment

- Giant, complicated async app

Constraints of the kernel environment

- Giant, complicated async app
- Full access to everything

Constraints of the kernel environment

- Giant, complicated async app
- Full access to everything
- Bugs can:

Constraints of the kernel environment

- Giant, complicated async app
- Full access to everything
- Bugs can:
  - Cause a system crash

Constraints of the kernel environment

- Giant, complicated async app
- Full access to everything
- Bugs can:
  - Cause a system crash
  - Totally break security

So how can we probe the state of the kernel without breaking stuff?

History: The Berkeley Packet Filter

History: The Berkeley Packet Filter

- **Problem:** www.facebook.com isn't loading

History: The Berkeley Packet Filter

- **Problem:** www.facebook.com isn't loading
- Is it a network problem?

History: The Berkeley Packet Filter

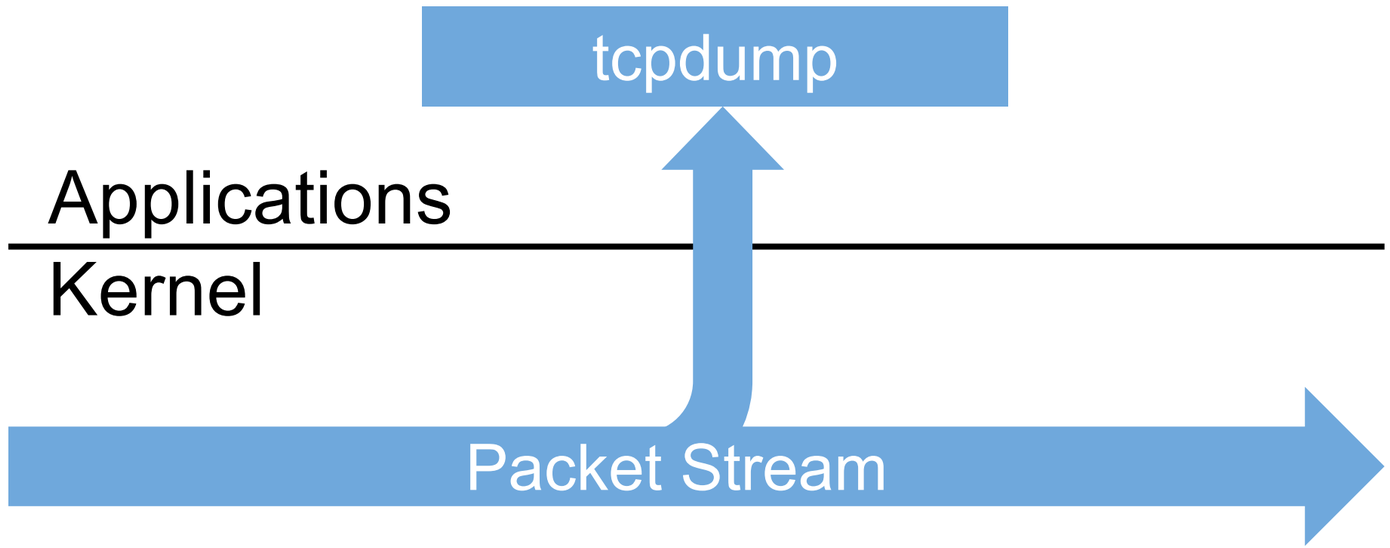
- **Problem:** www.facebook.com isn't loading
- Is it a network problem?
- Check by examining the network!

History: The Berkeley Packet Filter

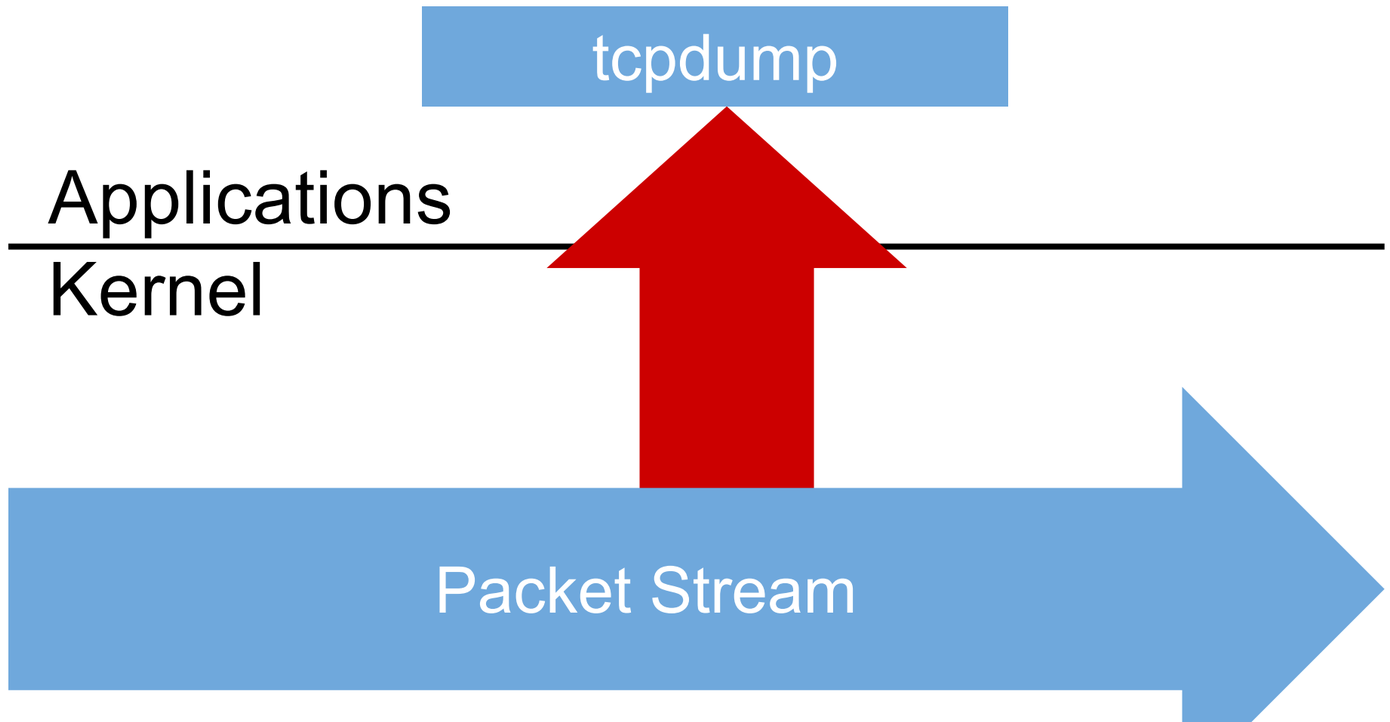
- **Problem:** www.facebook.com isn't loading
- Is it a network problem?
- Check by examining the network!

```
 $ sudo tcpdump -t -niany -c 10 port 80
IP 10.232.0.24.http > 192.168.124.179.52178: \
    Flags [S.], ... options [mss 1460], length 0
IP 192.168.124.179.52178 > 10.232.0.24.http: \
    Flags [.], ack 1, win 29200, length 0
IP 192.168.124.179.52178 > 10.232.0.24.http: \
     Flags [P.], seq 1:81, ack 1, win 29200, \
     length 80: HTTP: GET / HTTP/1.1
```

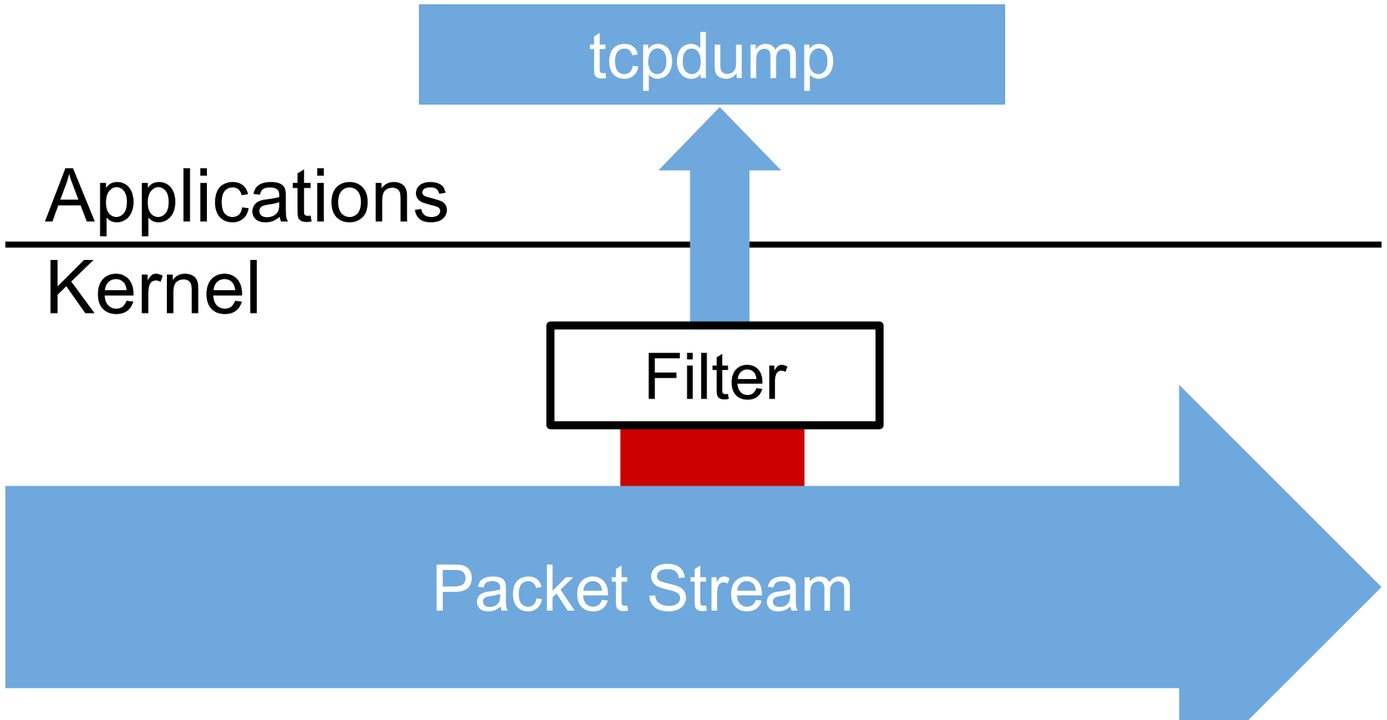
Getting packets from the kernel



It gets expensive fast



Answer: filter in the kernel



Building packet filters is hard

Building packet filters is hard

- Infinite network protocols to match on
  - TCP, UDP, DNS, ICMP, HTTP, ...

Building packet filters is hard

- Infinite network protocols to match on
  - TCP, UDP, DNS, ICMP, HTTP, ...
- Need to dynamically match on range of bytes (or bits!)

```
# Match tcp FIN packets
tcpdump 'tcp[13] & 1 != 0'

# Match packets with low (< 10 hop) time to live
tcpdump 'ip[8] < 10'
```

One option: run arbitary code

One option: run arbitary code

- Fast - perfectly optimized machine code

One option: run arbitary code

- Fast - perfectly optimized machine code
- Flexible - do anything you want

One option: run arbitary code

- Fast - perfectly optimized machine code
- Flexible - do anything you want
- Dangerous - A buggy filter crashes the system

One option: run arbitary code

- Fast - perfectly optimized machine code
- Flexible - do anything you want
- Dangerous - A buggy filter crashes the system
- Dangerous - Evil users are "above the law"

The Berkeley Packet Filter

The Berkeley Packet Filter

- Provide a minimal set of instructions

The Berkeley Packet Filter

- Provide a minimal set of instructions
- Statically verify that the code is valid

The Berkeley Packet Filter

- Provide a minimal set of instructions
- Statically verify that the code is valid
- (Optionally) translate it to native machine code

The Berkeley Packet Filter Operations

http://www.tcpdump.org/papers/bpf-usenix93.pdf

The Berkeley Packet Filter Operations

- Data access
  - Load/store data

http://www.tcpdump.org/papers/bpf-usenix93.pdf

The Berkeley Packet Filter Operations

- Data access
  - Load/store data
- Jumps
  - Unconditionally, >=, >, ==

http://www.tcpdump.org/papers/bpf-usenix93.pdf

The Berkeley Packet Filter Operations

- Data access
  - Load/store data
- Jumps
  - Unconditionally, >=, >, ==
- Math
  - +, -, *, /, &, |, <<, >>

http://www.tcpdump.org/papers/bpf-usenix93.pdf

The Berkeley Packet Filter Operations

- Data access
  - Load/store data
- Jumps
  - Unconditionally, >=, >, ==
- Math
  - +, -, *, /, &, |, <<, >>
- Return

http://www.tcpdump.org/papers/bpf-usenix93.pdf

Scope creep

Scope creep

- Turns out an in-kernel VM is useful

Scope creep

- Turns out an in-kernel VM is useful
- In 2014, Alexei begins making code changes to extend/generalize bpf
  - New bytecode that looks more like x86_64
  - General purpose bpf system call
  - Shared datastructures with applications (maps, arrays, etc.)

https://lwn.net/Articles/603983/

Extending bpf

Extending bpf

- Local memory

Extending bpf

- Local memory
- More registers
  - From 2 to 10

Extending bpf

- Local memory
- More registers
  - From 2 to 10
- Function calls
  - Built-in kernel helpers like getpid

Extending bpf

- Local memory
- More registers
  - From 2 to 10
- Function calls
  - Built-in kernel helpers like getpid
- Shared datastructures
  - Maps/Arrays that can be accessed from applications

Where can we use it today?

Where can we use it today?

- Software Tracepoints

Where can we use it today?

- Software Tracepoints
- Socket filters

Where can we use it today?

- Software Tracepoints
- Socket filters
- Traffic Control

Where can we use it today?

- Software Tracepoints
- Socket filters
- Traffic Control
- In the network card with XDP

bpf Toolchain

bpf Toolchain

- Primary interface is bcc

bpf Toolchain

- Primary interface is bcc
- C function is compiled to bpf using new clang backend

bpf Toolchain

- Primary interface is bcc
- C function is compiled to bpf using new clang backend
- Python script loads and inserts compiled C function

bpf Toolchain

- Primary interface is bcc
- C function is compiled to bpf using new clang backend
- Python script loads and inserts compiled C function
- ctypes enables datastructure sharing

simple_tc.py example

```
# Copyright (c) PLUMgrid, Inc.
# Licensed under the Apache License, Version 2.0 ...

...

text = """
int hello(struct __sk_buff *skb) {
  return 1;
}
"""

b = BPF(text=text, debug=0)
fn = b.load_func("hello", BPF.SCHED_CLS)
IPRoute().tc("add-filter", "bpf", fd=fn.fd, ...)
```

https://github.com/iovisor/bcc/blob/master/examples/networking/simple_tc.py

Why not just translate python bytecode to bpf bytecode? — Me (naïve)

Agenda

1. Who am I?
2. Overview
3. **Python Virtual Machine**
4. Transpiling from python to bpf bytecode
5. Woo, we made it!

Python bytecode overview

Python bytecode overview

- Single magical, infinite stack of python objects

Python bytecode overview

- Single magical, infinite stack of python objects
- Each python bytecode operates on the magical stack

Python bytecode overview

- Single magical, infinite stack of python objects
- Each python bytecode operates on the magical stack
- No real concept of the physical machine

Wait, python bytecode?

Wait, python bytecode?

- python functions are compiled into bytecode, which is executed on the fly

Wait, python bytecode?

- python functions are compiled into bytecode, which is executed on the fly
- You can see this bytecode with the dis module

Wait, python bytecode?

- python functions are compiled into bytecode, which is executed on the fly
- You can see this bytecode with the dis module

```
>>> def func(a, b):
...     return a.x + max(b)
...
>>> dis.dis(func)
  2           0 LOAD_FAST                0 (a)
              2 LOAD_ATTR                0 (x)
              4 LOAD_GLOBAL              1 (max)
              6 LOAD_FAST                1 (b)
              8 CALL_FUNCTION            1
             10 BINARY_ADD
             12 RETURN_VALUE
```

Stack-based virtual machine??

Stack-based virtual machine??

- All operations occur with the stack

Stack-based virtual machine??

- All operations occur with the stack
- Take the expression: x + y * z

Stack-based virtual machine??

- All operations occur with the stack
- Take the expression: x + y * z
- In a stack-based representation:

```
push x
push y
push z
multiply
add
return
```

Stack-Based Example

x + y * z

```
push x
push y
push z
multiply
add
return
```

Stack-Based Example

x + y * z

```
push x    - stack.append(x)
push y    - stack.append(y)
push z    - stack.append(z)
multiply
add
return
```

Stack-Based Example

x + y * z

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add
return
```

Stack-Based Example

x + y * z

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return
```

Stack-Based Example

x + y * z

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return     - return stack.pop()
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x    - stack.append(x)
push y    - stack.append(y)
push z    - stack.append(z)
multiply  - stack.append(stack.pop() * stack.pop())
add       - stack.append(stack.pop() + stack.pop())
return    - return stack.pop()
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return     - return stack.pop()
```

```
stack = [2]
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x    - stack.append(x)
push y    - stack.append(y)
push z    - stack.append(z)
multiply  - stack.append(stack.pop() * stack.pop())
add       - stack.append(stack.pop() + stack.pop())
return    - return stack.pop()
```

```
stack = [2, 3]
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return     - return stack.pop()
```

```
stack = [2, 3, 5]
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return     - return stack.pop()
```

```
stack = [2, 15]
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return     - return stack.pop()
```

```
stack = [17]
```

Stack-Based Example

x + y * z (x=2, y=3, z=5)

```
push x     - stack.append(x)
push y     - stack.append(y)
push z     - stack.append(z)
multiply   - stack.append(stack.pop() * stack.pop())
add        - stack.append(stack.pop() + stack.pop())
return     - return stack.pop()
```

```
stack = []
```

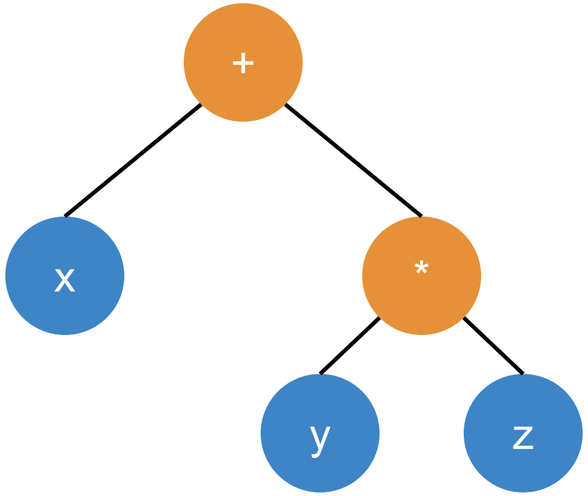
Generating stack-based bytecode

Generating stack-based bytecode

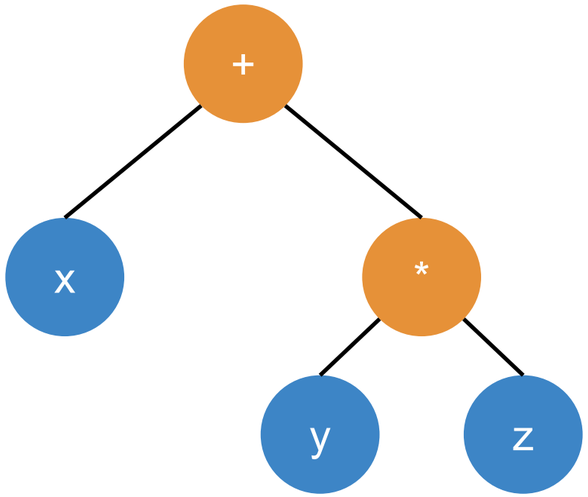
x + y * z

Generating stack-based bytecode

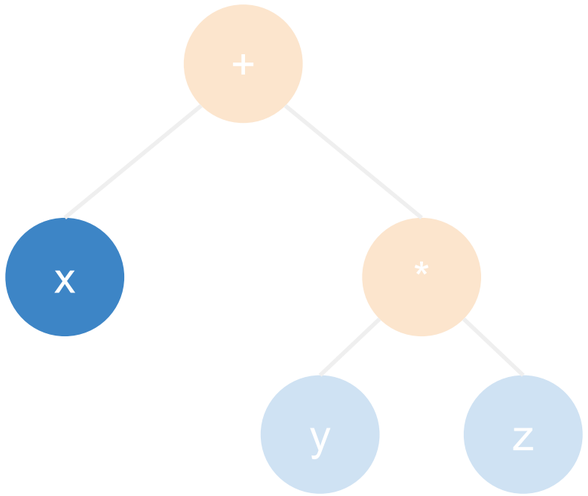
x + y * z



Generating stack-based bytecode

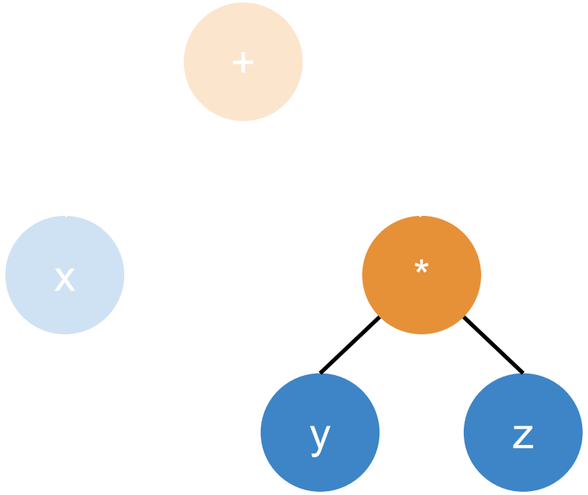


Generating stack-based bytecode



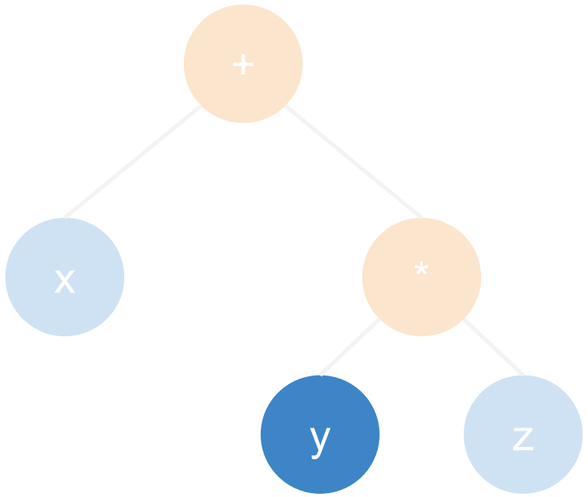

```
push x
```

Generating stack-based bytecode



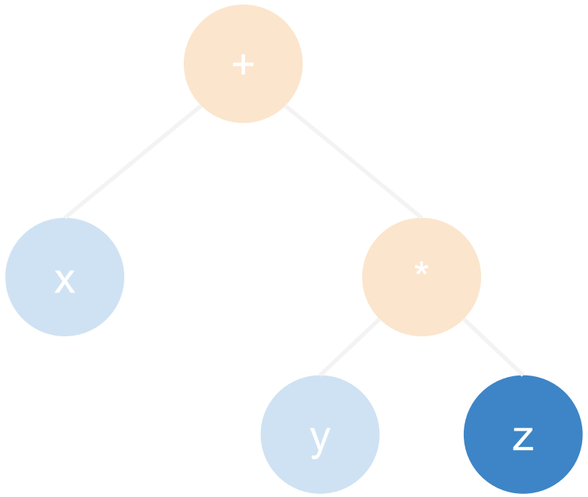

```
push x
```

Generating stack-based bytecode



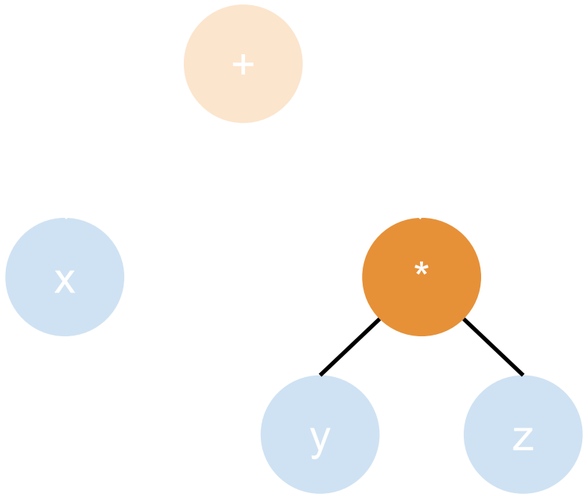

```
push x
push y
```

Generating stack-based bytecode



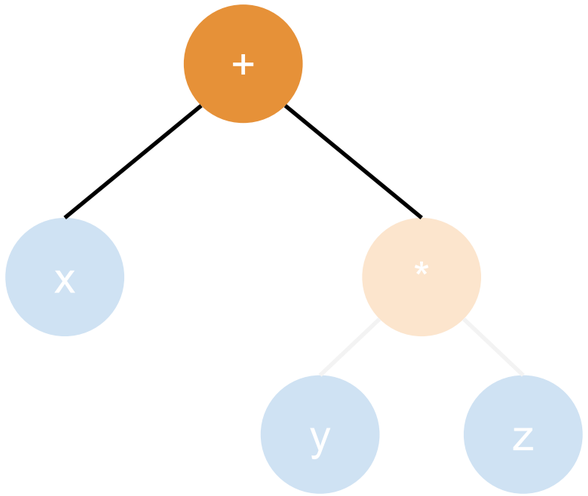

```
push x
push y
push z
```

Generating stack-based bytecode



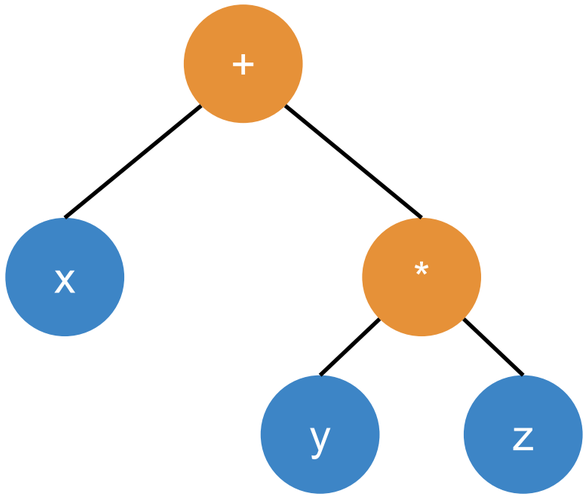

```
push x
push y
push z
multiply
```

Generating stack-based bytecode




```
push x
push y
push z
multiply
add
```

Generating stack-based bytecode




```
push x
push y
push z
multiply
add
return
```

The python virtual machine (in python!)

```
def run(instructions):



    while True:
        pass
```

Program Counter

```
def run(instructions):
    pc = 0



    while True:
        ins = instructions[pc]
        pc += 1
```

The Stack - Write Ops

```
def run(instructions):
    pc = 0
    stack = []


    while True:
        ins = instructions[pc]
        pc += 1
        if ins.opcode == LOAD_CONST:
            stack.append(ins.argval)
        ...
```

The Stack - Read Ops

```
def run(instructions):
    pc = 0
    stack = []


    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == RETURN_VALUE:
            return stack.pop()
        ...
```

The Stack - Read/Write Ops

```
def run(instructions):
    pc = 0
    stack = []


    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == BINARY_MULTIPLY:
            arg1 = stack.pop()
            arg2 = stack.pop()
            stack.append(arg1 * arg2)
        ...
```

Control Flow - Absolute

```
def run(instructions):
    pc = 0
    stack = []


    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == JUMP_ABSOLUTE:
            pc = ins.argval
        ...
```

Control Flow - Conditional

```
def run(instructions):
    pc = 0
    stack = []


    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == POP_JUMP_IF_TRUE:
            if stack.pop():
                pc = ins.argval
        ...
```

Fast Vars - Reading

```
def run(instructions):
    pc = 0
    stack = []
    vars = {}

    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == LOAD_FAST:
            stack.append(vars[ins.argval])
        ...
```

Fast Vars - Writing

```
def run(instructions):
    pc = 0
    stack = []
    vars = {}

    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == STORE_FAST:
            var[ins.argval] = stack.pop()
        ...
```

Global Vars

```
def run(instructions, globals):
    pc = 0
    stack = []
    vars = {}

    while True:
        ins = instructions[pc]
        pc += 1
        ...
        elif ins.opcode == LOAD_GLOBAL:
            stack.append(globals[ins.argval])
        elif ins.opcode == STORE_GLOBAL:
             globals[ins.argval] = stack.pop()
        ...
```

Many more python opcodes

POP_TOP	ROT_TWO	ROT_THREE	DUP_TOP
DUP_TOP_TWO	NOP	UNARY_POSITIVE	UNARY_NEGATIVE
UNARY_NOT	UNARY_INVERT	BINARY_MATRIX_MULTIPLY	INPLACE_MATRIX_MULTIPLY
BINARY_POWER	BINARY_MULTIPLY	BINARY_MODULO	BINARY_ADD
BINARY_SUBTRACT	BINARY_SUBSCR	BINARY_FLOOR_DIVIDE	BINARY_TRUE_DIVIDE
INPLACE_FLOOR_DIVIDE	INPLACE_TRUE_DIVIDE	GET_AITER	GET_ANEXT
BEFORE_ASYNC_WITH	INPLACE_ADD	INPLACE_SUBTRACT	INPLACE_MULTIPLY
INPLACE_MODULO	STORE_SUBSCR	DELETE_SUBSCR	BINARY_LSHIFT
BINARY_RSHIFT	BINARY_AND	BINARY_XOR	BINARY_OR
INPLACE_POWER	GET_ITER	GET_YIELD_FROM_ITER	PRINT_EXPR
LOAD_BUILD_CLASS	YIELD_FROM	GET_AWAITABLE	INPLACE_LSHIFT
INPLACE_RSHIFT	INPLACE_AND	INPLACE_XOR	INPLACE_OR
BREAK_LOOP	WITH_CLEANUP_START	WITH_CLEANUP_FINISH	RETURN_VALUE
IMPORT_STAR	SETUP_ANNOTATIONS	YIELD_VALUE	POP_BLOCK
END_FINALLY	POP_EXCEPT	STORE_NAME	DELETE_NAME
UNPACK_SEQUENCE	FOR_ITER	UNPACK_EX	STORE_ATTR
DELETE_ATTR	STORE_GLOBAL	DELETE_GLOBAL	LOAD_CONST
LOAD_NAME	BUILD_TUPLE	BUILD_LIST	BUILD_SET
BUILD_MAP	LOAD_ATTR	COMPARE_OP	IMPORT_NAME
IMPORT_FROM	JUMP_FORWARD	JUMP_IF_FALSE_OR_POP	JUMP_IF_TRUE_OR_POP
JUMP_ABSOLUTE	POP_JUMP_IF_FALSE	POP_JUMP_IF_TRUE	LOAD_GLOBAL
CONTINUE_LOOP	SETUP_LOOP	SETUP_EXCEPT	SETUP_FINALLY
LOAD_FAST	STORE_FAST	DELETE_FAST	STORE_ANNOTATION
RAISE_VARARGS	CALL_FUNCTION	MAKE_FUNCTION	BUILD_SLICE
LOAD_CLOSURE	LOAD_DEREF	STORE_DEREF	DELETE_DEREF
CALL_FUNCTION_KW	CALL_FUNCTION_EX	SETUP_WITH	LIST_APPEND
SET_ADD	MAP_ADD	LOAD_CLASSDEREF	EXTENDED_ARG
BUILD_LIST_UNPACK	BUILD_MAP_UNPACK	BUILD_MAP_UNPACK_WITH_CALL	BUILD_TUPLE_UNPACK
BUILD_SET_UNPACK	SETUP_ASYNC_WITH	FORMAT_VALUE	BUILD_CONST_KEY_MAP
	BUILD_STRING	BUILD_TUPLE_UNPACK_WITH_CALL	

https://docs.python.org/3.6/library/dis.html

Agenda

1. Who am I?
2. Overview
3. Python Virtual Machine
4. **Transpiling from python to bpf bytecode**
5. Woo, we made it!

bpf bytecode overview

bpf bytecode overview

- Small stack is just array of bytes

bpf bytecode overview

- Small stack is just array of bytes
- Most opcodes operate on registers only

bpf bytecode overview

- Small stack is just array of bytes
- Most opcodes operate on registers only
- Need to store/load from/to memory

bpf bytecode overview

- Small stack is just array of bytes
- Most opcodes operate on registers only
- Need to store/load from/to memory
- Everything has a fixed size and type
  - 32-bit int, 8-bit character, etc.

bpf bytecode overview

- Small stack is just array of bytes
- Most opcodes operate on registers only
- Need to store/load from/to memory
- Everything has a fixed size and type
  - 32-bit int, 8-bit character, etc.
- No backward or absolute jumping allowed

Essential Steps

Essential Steps

1. Freeze python-isms in place with constant folding

Essential Steps

1. Freeze python-isms in place with constant folding

2. Normalize stack-based VM to variable-based VM

Essential Steps

1. Freeze python-isms in place with constant folding

2. Normalize stack-based VM to variable-based VM

3. Deduce type and allocate space for each variable

Essential Steps

1. Freeze python-isms in place with constant folding

2. Normalize stack-based VM to variable-based VM

3. Deduce type and allocate space for each variable

4. Translate bytecodes

Constant folding

Constant folding

- Freeze all globals in place

Constant folding

- Freeze all globals in place
- Perform as many operations as we can
  - Attribute loads
  - Arithmetic
  - Function calls

Constant folding

```
>>> def filter(skb):
...     return socket.htons(ETH_P_IP)
...
```

Constant folding

```
>>> def filter(skb):
...     return socket.htons(ETH_P_IP)
...
```

```
>>> dis.dis(filter)
  2           0 LOAD_GLOBAL              0 (socket)
              2 LOAD_ATTR                1 (htons)
              4 LOAD_GLOBAL              2 (ETH_P_IP)
              6 CALL_FUNCTION            1
              8 RETURN_VALUE
```

Constant folding

```
LOAD_GLOBAL('socket')
LOAD_ATTR('htons')
LOAD_GLOBAL('ETH_P_IP')
CALL_FUNCTION(1)
RETURN_VALUE
```

Constant folding - pin globals

```
    LOAD_CONST(socket) # socket module
    LOAD_ATTR('htons')
    LOAD_CONST(0x0800) # ETH_P_IP
    CALL_FUNCTION(1)
    RETURN_VALUE
```

Constant folding - fold constants

```
LOAD_CONST(socket.htons) # socket.htons
LOAD_CONST(0x0800) # ETH_P_IP
CALL_FUNCTION(1)
RETURN_VALUE
```

Constant folding - fold constants

```
LOAD_CONST(0x0008) # socket.htons(ETH_P_IP)
RETURN_VALUE
```

Eliminating the stack

Eliminating the stack

- Trace every execution path

Eliminating the stack

- Trace every execution path
- Identify source instructions for each operation

Eliminating the stack

- Trace every execution path
- Identify source instructions for each operation
- Normalize all destinations into "variables"

Eliminating the stack

```
>>> def func(a, b):
...     return a * (b if b > 0 else 7)
...
```

Eliminating the stack

```
>>> def func(a, b):
...     return a * (b if b > 0 else 7)
...
```

```
>>> dis.dis(func)
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 LOAD_CONST               1 (0)
              6 COMPARE_OP               4 (>)
              8 POP_JUMP_IF_FALSE       14
             10 LOAD_FAST                1 (b)
             12 JUMP_FORWARD             2 (to 16)
        >>   14 LOAD_CONST               2 (7)
        >>   16 BINARY_MULTIPLY
             18 RETURN_VALUE
```

Tracing execution paths

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')
  2:   LOAD_CONST(0)
  3:   COMPARE_OP(>)
  4:   POP_JUMP_IF_FALSE(14)

  5:   LOAD_FAST('b')
  6:   JUMP_FORWARD(16)

  7: LABEL(14)
  8:   LOAD_CONST(7)

  9: LABEL(16)
 10:   BINARY_MULTIPLY()
 11:   RETURN_VALUE
`
```

Tracing execution paths - b <= 0

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')
  2:   LOAD_CONST(0)
  3:   COMPARE_OP(>)
  4:   POP_JUMP_IF_FALSE(14)

  5:   LOAD_FAST('b')
  6:   JUMP_FORWARD(16)

  7: LABEL(14)
  8:   LOAD_CONST(7)

  9: LABEL(16)
 10:   BINARY_MULTIPLY()
 11:   RETURN_VALUE
```

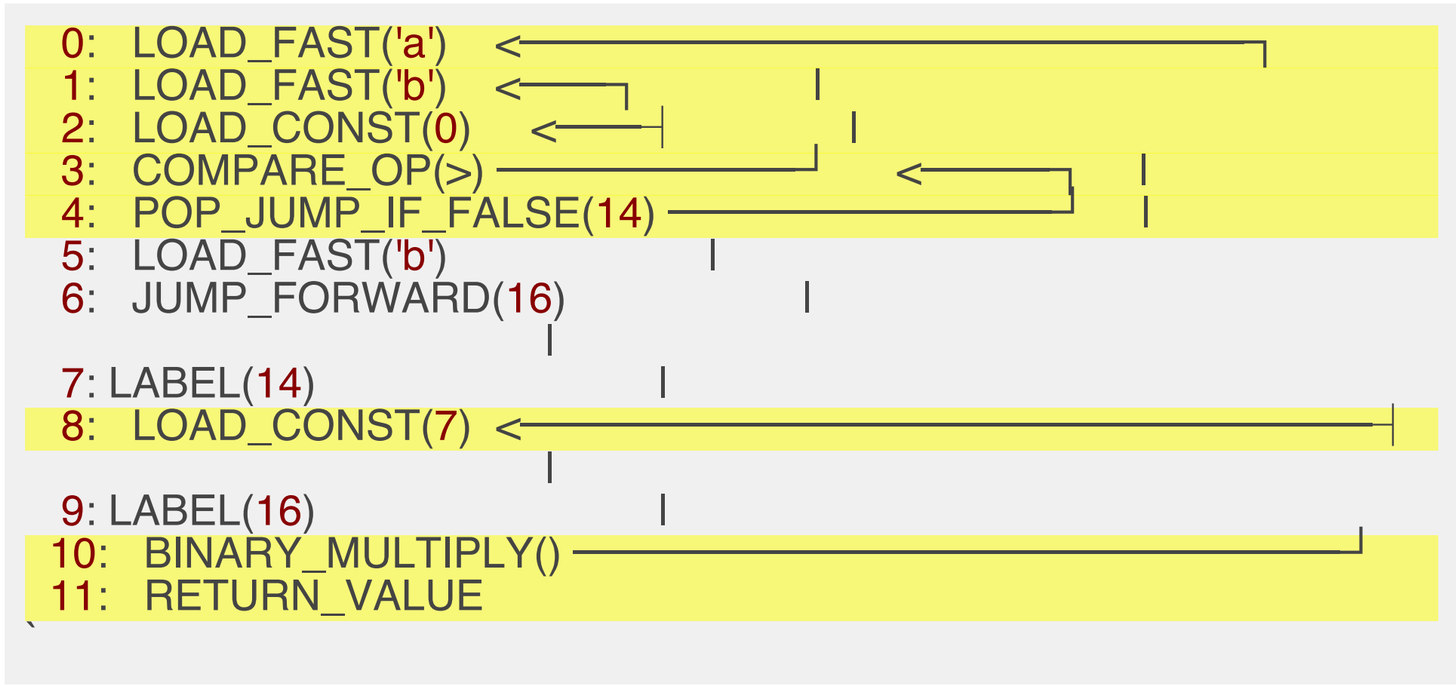
Tracing execution paths - b <= 0

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')    <----,
  2:   LOAD_CONST(0)     <----|
  3:   COMPARE_OP(>) ---------'
  4:   POP_JUMP_IF_FALSE(14)
  5:   LOAD_FAST('b')
  6:   JUMP_FORWARD(16)

  7: LABEL(14)
  8:   LOAD_CONST(7)

  9: LABEL(16)
 10:   BINARY_MULTIPLY()
 11:   RETURN_VALUE
```

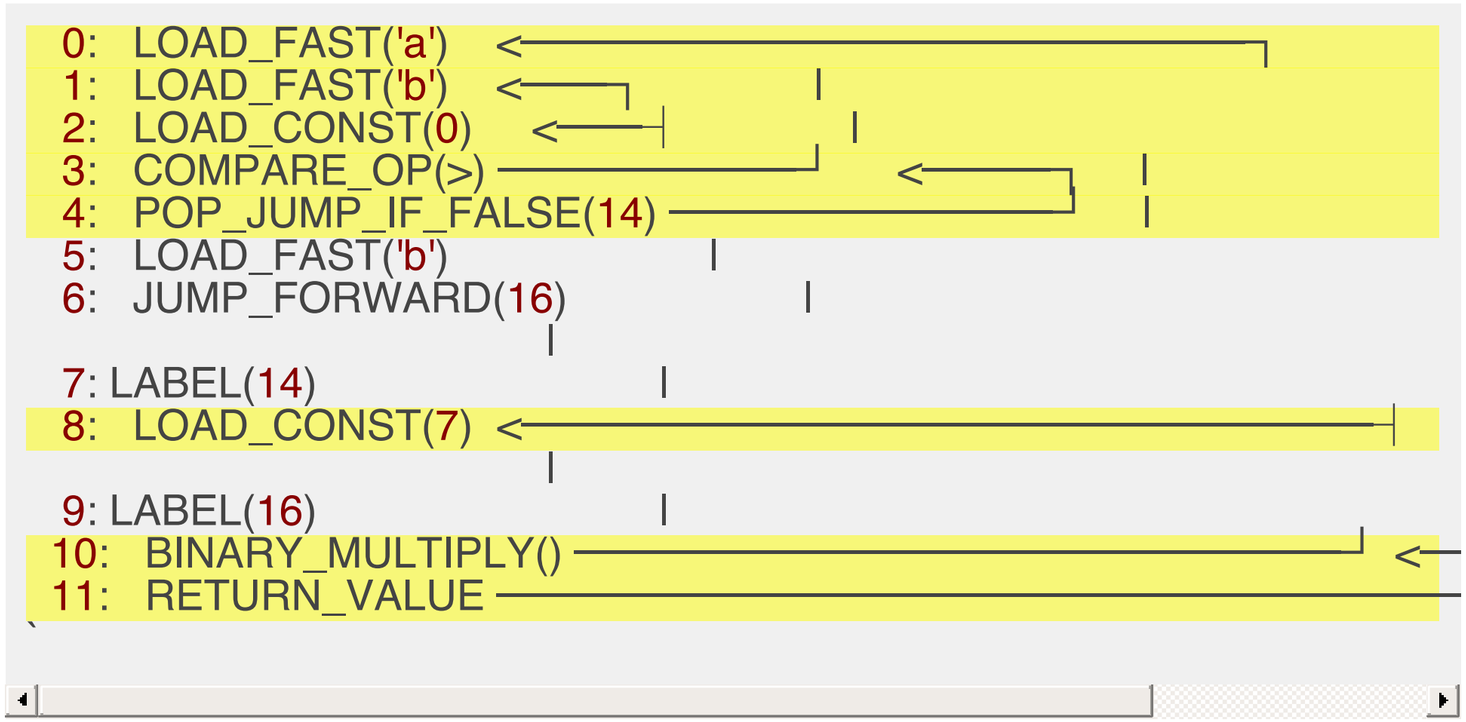
Tracing execution paths - b <= 0

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')    <----,
  2:   LOAD_CONST(0)     <----|
  3:   COMPARE_OP(>)     -----|----,
  4:   POP_JUMP_IF_FALSE(14) -|----|----,
  5:   LOAD_FAST('b')         |    |    |
  6:   JUMP_FORWARD(16)       |    |    |
                              |    |    |
  7: LABEL(14)                |    |    |
  8:   LOAD_CONST(7)          |    |    |
                              |    |    |
  9: LABEL(16)                |    |    |
 10:   BINARY_MULTIPLY()      |    |    |
 11:   RETURN_VALUE           |    |    |
`                             |    |    |
```

Tracing execution paths - b <= 0



Tracing execution paths - b <= 0



Tracing execution paths - b > 0

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')
  2:   LOAD_CONST(0)
  3:   COMPARE_OP(>)
  4:   POP_JUMP_IF_FALSE(14)
  5:   LOAD_FAST('b')
  6:   JUMP_FORWARD(16)

  7: LABEL(14)
  8:   LOAD_CONST(7)

  9: LABEL(16)
 10:   BINARY_MULTIPLY()
 11:   RETURN_VALUE
```

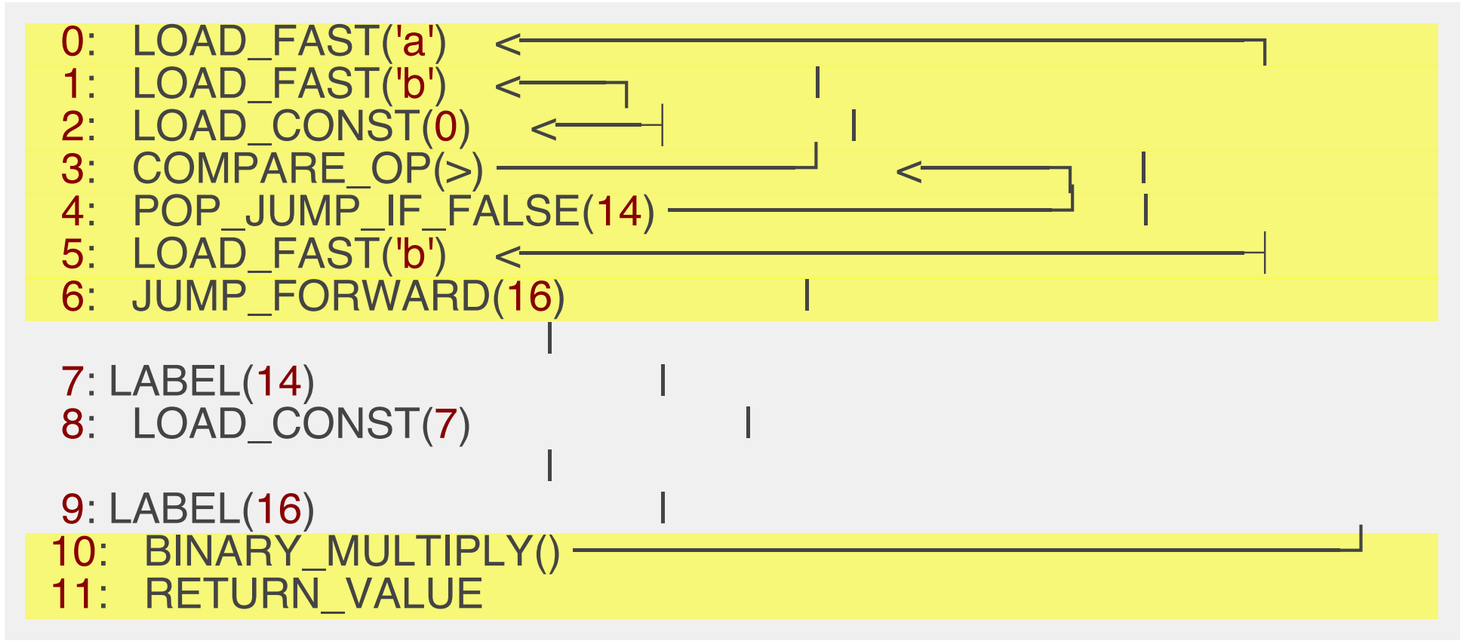
Tracing execution paths - b > 0

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')    <┐
  2:   LOAD_CONST(0)     <┤
  3:   COMPARE_OP(>) ─────┘
  4:   POP_JUMP_IF_FALSE(14)
  5:   LOAD_FAST('b')
  6:   JUMP_FORWARD(16)

  7: LABEL(14)
  8:   LOAD_CONST(7)

  9: LABEL(16)
 10:   BINARY_MULTIPLY()
 11:   RETURN_VALUE
`
```

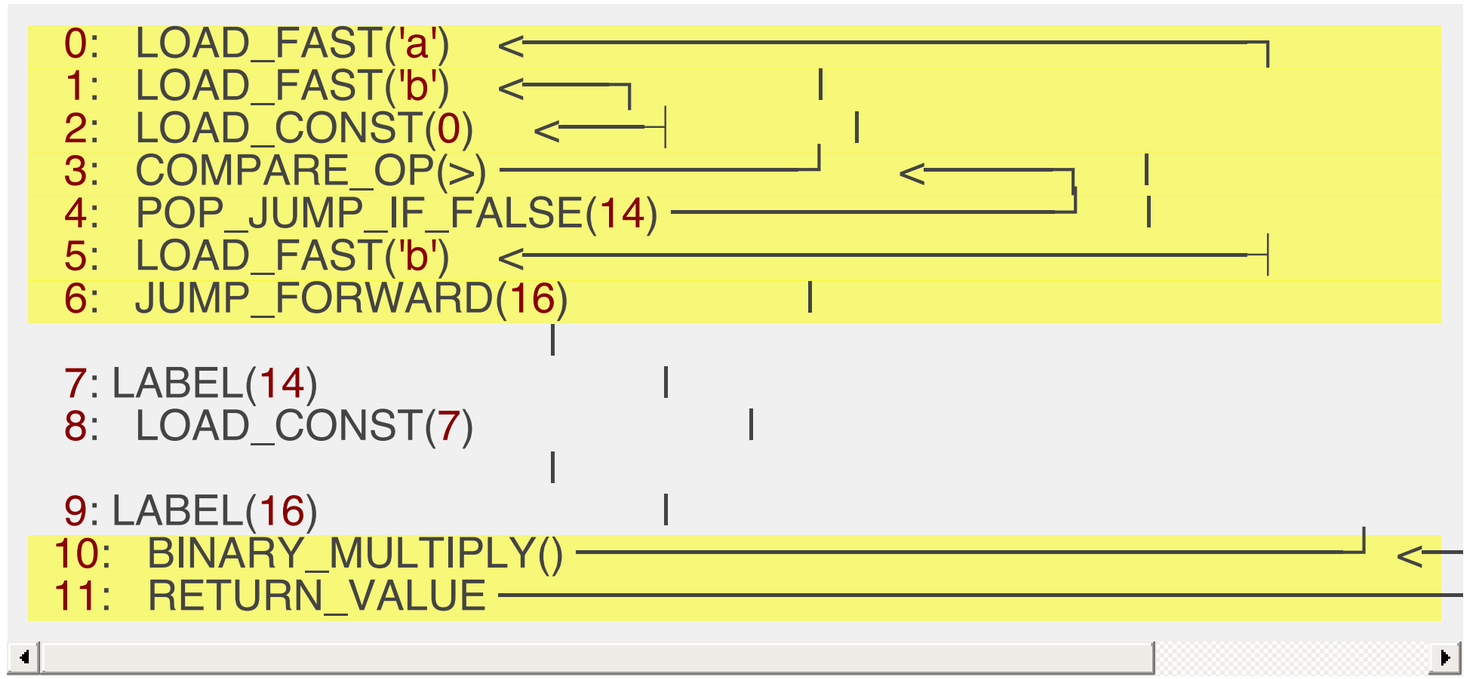
Tracing execution paths - b > 0

```
  0:   LOAD_FAST('a')
  1:   LOAD_FAST('b')    <----┐
  2:   LOAD_CONST(0)     <----|
  3:   COMPARE_OP(>)     ---------┘      <----┐
  4:   POP_JUMP_IF_FALSE(14) ---------------------┘
  5:   LOAD_FAST('b')
  6:   JUMP_FORWARD(16)

  7: LABEL(14)
  8:   LOAD_CONST(7)

  9: LABEL(16)
 10:   BINARY_MULTIPLY()
 11:   RETURN_VALUE
`
```

Tracing execution paths - b > 0



Tracing execution paths - b > 0



Normalizing to variables

Normalizing to variables

- Most of these were uninteresting
  - 3 always reads 1 and 2
  - 4 always reads 3
  - etc.

Normalizing to variables

- Most of these were uninteresting
  - 3 always reads 1 and 2
  - 4 always reads 3
  - etc.

- 10 is interesting
  - If b <= 0, it reads 0 and 8
  - If b > 0, it reads 0 and 5

Normalizing to variables

- Most of these were uninteresting
  - 3 always reads 1 and 2
  - 4 always reads 3
  - etc.

- 10 is interesting
  - If b <= 0, it reads 0 and 8
  - If b > 0, it reads 0 and 5

- So we know that 5 and 8 output to the same variable

Now, with variables

```
  var1 = LOAD_FAST('a')
  var2 = LOAD_FAST('b')
  var3 = LOAD_CONST(0)
  var4 = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6 = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6 = LOAD_CONST(7)

LABEL(16)
  var7 = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Observed the shared variable!

```
  var1 = LOAD_FAST('a')
  var2 = LOAD_FAST('b')
  var3 = LOAD_CONST(0)
  var4 = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6 = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6 = LOAD_CONST(7)

LABEL(16)
  var7 = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Deducing types

Deducing types

- Each variable needs to become "real"
  - i.e. a ctype with sizeof

Deducing types

- Each variable needs to become "real"
  - i.e. a ctype with sizeof
- We know the input argument types ahead of time
  - In this example, ints

Deducing types

- Each variable needs to become "real"
  - i.e. a ctype with sizeof
- We know the input argument types ahead of time
  - In this example, ints
- Other type clues
  - Type of constant
  - Operation type (i.e. compare, multiply)
  - We created the var (and knew it then)
  - Type of attribute

Deducing types

```
  var1 = LOAD_FAST('a')
  var2 = LOAD_FAST('b')
  var3 = LOAD_CONST(0)
  var4 = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6 = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6 = LOAD_CONST(7)

LABEL(16)
  var7 = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Deducing types from constants

```
  var1 = LOAD_FAST('a')
  var2 = LOAD_FAST('b')
  var3<int> = LOAD_CONST(0)
  var4 = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6 = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6<int> = LOAD_CONST(7)

LABEL(16)
  var7 = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Deducing types from known arguments

```
  var1<int> = LOAD_FAST('a')
  var2<int> = LOAD_FAST('b')
  var3<int> = LOAD_CONST(0)
  var4 = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6<int> = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6<int> = LOAD_CONST(7)

LABEL(16)
  var7 = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Deducing types from operations

```
  var1<int> = LOAD_FAST('a')
  var2<int> = LOAD_FAST('b')
  var3<int> = LOAD_CONST(0)
  var4<bool> = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6<int> = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6<int> = LOAD_CONST(7)

LABEL(16)
  var7<int> = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Deducing types

```
  var1<int> = LOAD_FAST('a')
  var2<int> = LOAD_FAST('b')
  var3<int> = LOAD_CONST(0)
  var4<bool> = COMPARE_OP(>, var1, var2)
  POP_JUMP_IF_FALSE(14, var4)

  var6<int> = LOAD_FAST('b')
  JUMP_FORWARD(16)

LABEL(14)
  var6<int> = LOAD_CONST(7)

LABEL(16)
  var7<int> = BINARY_MULTIPLY(va1, var6)
  RETURN_VALUE(var7)
```

Dealing with primitive types

Dealing with primitive types

```
x = 1
foo(x) # (Logically) makes a copy

x = Obj()
foo(x) # Passes a reference
```

Dealing with primitive types

```
x = 1
foo(x) # (Logically) makes a copy

x = Obj()
foo(x) # Passes a reference
```

- Both can be references in python, because ints are immutable

Dealing with primitive types

```
x = 1
foo(x) # (Logically) makes a copy

x = Obj()
foo(x) # Passes a reference
```

- Both can be references in python, because ints are immutable
- Not how machine code works

Dealing with primitive types

```
x = 1
foo(x) # (Logically) makes a copy

x = Obj()
foo(x) # Passes a reference
```

- Both can be references in python, because ints are immutable
- Not how machine code works
- Some backflips to do what you'd expect, semantically
  - Not in scope for this presentation

Assigning vars to memory

Assigning vars to memory

- Variables can't just magically "exist"

Assigning vars to memory

- Variables can't just magically "exist"
- Need to live somewhere

Assigning vars to memory

- Variables can't just magically "exist"
- Need to live somewhere
  - Pointed to by arguments

Assigning vars to memory

- Variables can't just magically "exist"
- Need to live somewhere
  - Pointed to by arguments
  - Stored on the stack

Assigning vars to memory

```
>>> def func(a):
...     c = a
...     return c
```

Assigning vars to memory

```
>>> def func(a):
...     c = a
...     return c
```

```
>>> dis.dis(func)
  2           0 LOAD_FAST                0 (a)
              2 STORE_FAST               1 (c)

  3           4 LOAD_FAST                1 (c)
              6 RETURN_VALUE
```

After folding and type deduction

```
var1<int> = LOAD_FAST('a')
STORE_FAST('c', var1<int>)
var2<int> = LOAD_FAST('c', var1<int>)
RETURN_VALUE(var2)
```

Convert argument fast vars

```
var1<int> = LOAD_FAST('a')
STORE_FAST('c', var1<int>)
var2<int> = LOAD_FAST('c', var1<int>)
RETURN_VALUE(var2)
```

```
var1<int> = arg_var_0<int> # register 1
STORE_FAST('c', var1<int>)
var2<int> = LOAD_FAST('c', var1<int>)
RETURN_VALUE(var2)
```

Convert other fast vars (locals)

```
var1<int> = arg_var_0<int>
STORE_FAST('c', var1<int>)
var2<int> = LOAD_FAST('c', var1<int>)
RETURN_VALUE(var2)
```

```
var1<int> = arg_var_0<int>
fast_var_c<int> = var1
var2<int> = fast_var_c
RETURN_VALUE(var2)
```

Allocate space for local vars

```
var1<int> = arg_var_0<int>
fast_var_c<int> = var1
var2<int> = fast_var_c
RETURN_VALUE(var2)
```

```
var1<int> = arg_var_0<int>
stack_var_c<int, off=0> = var1
var2<int> = stack_var_c<int, off=0>
RETURN_VALUE(var2)
```

Allocate space for remaining vars

```
var1<int> = arg_var_0<int>
stack_var_c<int, off=0> = var1
var2<int> = stack_var_c<int, off=0>
RETURN_VALUE(var2)
```

```
stack_var1<int, off=8> = arg_var_0<int>
stack_var_c<int, off=0> = stack_var1<int, off=8>
stack_var2<int, off=16> = stack_var_c<int, off=0>
RETURN_VALUE(stack_var2<int, off=16>)
```

Template compilation

Template compilation

- At this point, representation is "close enough"

Template compilation

- At this point, representation is "close enough"
- We can simply take each op code and translate it

Translating RETURN_VALUE

Pseudo-python

```
RETURN_VALUE(stack_var1<int, off=0>)
```

Translating RETURN_VALUE

Pseudo-python

```
RETURN_VALUE(stack_var1<int, off=0>)
```

bpf

```
# Return value is in register 0
LOAD64(src=stack, off=0, dst=R0)
RET()
```

Translating BINARY_MULTIPLY

Pseudo-python

```
stack_var3<int, off=16> = BINARY_MULTIPLY(
    stack_var1<int, off=0>, stack_var2<int, off=8>)
```

Translating BINARY_MULTIPLY

Pseudo-python

```
stack_var3<int, off=16> = BINARY_MULTIPLY(
    stack_var1<int, off=0>, stack_var2<int, off=8>)
```

bpf

```
LOAD64(src=stack, off=0, dst=R1)
LOAD64(src=stack, off=8, dst=R2)
MULTIPLY(src=R2, dst=R1)
STORE64(src=R1, dst=stack, off=16)
```

Translating COMPARE_OP(==)

```
stack_var3<bool, off=16> = COMPARE_OP(
    '==', stack_var1<int, off=0>,
    stack_var2<int, off=8>)
```

Translating COMPARE_OP(==)

```
stack_var3<bool, off=16> = COMPARE_OP(
    '==', stack_var1<int, off=0>,
    stack_var2<int, off=8>)
```

```
LOAD64(src=stack, off=0, dst=R1)
LOAD64(src=stack, off=8, dst=R2)
JUMP_IF_EQUAL_TO(src=R1, dst=R2, off='tmp_true')

STORE64(imm=FALSE, dst=R1)
JUMP(off='tmp_done')

LABEL('tmp_true')
STORE64(imm=TRUE, dst=R1)

LABEL('tmp_done')
STORE64(src=R1, dst=stack, off=16)
```

Dealing with shared datastructures

Dealing with shared datastructures

- Some datastructures can be shared

Dealing with shared datastructures

- Some datastructures can be shared
  - Hash Map

Dealing with shared datastructures

- Some datastructures can be shared
  - Hash Map
  - Queue

Dealing with shared datastructures

- These are incredibly useful

```
# Create a map from protocol to packet count
my_map = create_map(
    ctypes.c_int8, ctypes.c_int32, 256)

def socket_filter(skb):
    my_map[skb.protocol] += 1 # Access in kernel
    return 0

start_filter(socket_filter)

print(my_map[ETH_P_IP]) # Access in userspace
```

Accessing shared datastructures

Accessing shared datastructures

- In application
  - Referenced by file descriptor
  - Accessed with bpf(...) syscall

Accessing shared datastructures

- In application
  - Referenced by file descriptor
  - Accessed with bpf(...) syscall

- In kernel
  - Referenced by pointer
  - Accessed with built-in functions

Accessing shared datastructures

- In application
  - Referenced by file descriptor
  - Accessed with bpf(...) syscall
- In kernel
  - Referenced by pointer
  - Accessed with built-in functions
- Solution: create a special datastructure class

Accessing shared datastructures

- In application
  - Referenced by file descriptor
  - Accessed with bpf(...) syscall
- In kernel
  - Referenced by pointer
  - Accessed with built-in functions
- Solution: create a special datastructure class
  - Provides __getitem__, __setitem__ for application access

Accessing shared datastructures

- In application
  - Referenced by file descriptor
  - Accessed with bpf(...) syscall
- In kernel
  - Referenced by pointer
  - Accessed with built-in functions
- Solution: create a special datastructure class
  - Provides __getitem__, __setitem__ for application access
  - Looks like ctypes.c_int to variable allocator

Accessing shared datastructures

- In application
  - Referenced by file descriptor
  - Accessed with bpf(...) syscall

- In kernel
  - Referenced by pointer
  - Accessed with built-in functions

- Solution: create a special datastructure class
  - Provides __getitem__, __setitem__ for application access
  - Looks like ctypes.c_int to variable allocator
  - Supports opcode translation for BINARY_SUBSCR, et. al.

Agenda

1. Who am I?
2. Overview
3. Python Virtual Machine
4. Transpiling from python to bpf bytecode
5. **Woo, we made it!**

More things to do

More things to do

- Variable lifetime analysis
  - Keep things in register to avoid stores/loads
  - Reuse stack space

More things to do

- Variable lifetime analysis
  - Keep things in register to avoid stores/loads
  - Reuse stack space

- Higher level abstractions to simplify use
  - Currently tracing requires low-level knowledge of C

Thank You