NPF — progress and perspective

Mindaugas Rasiukevicius
The NetBSD Project
rmind@netbsd.org

AsiaBSDCon 2014, Japan
15 March 2014

Introduction

What is NPF?

NPF – is a **Net**BSD packet filter, which can do TCP/IP traffic filtering, stateful inspection and network address translation.

Introduction

Motivation: multi-core world and 3rd party extensions

Multi-core world:

- There was no SMP optimised packet filter in *BSD.
- The code base of other packet filters seemed unsatisfactory.
- NPF idea was partly a response to *nftables* developed under the Linux Netfilter project.

Users and vendors often need custom solutions.

- There was no packet filter in *BSD with an emphasis on modularity.
- Linux Netfilter provided the most convenient framework for custom extensions. GPL is an issue: there are known GPL-related legal disputes with vendors using Netfilter.

Features

Highlights

Hence, **NPF**:

- Written from scratch with a focus on performance, scalability and modularity.
- Supports stateful packet filtering and network address translation.
- Convenient support for extensions.
- Protocol independence in the NPF core engine.
- Support for "tables": storage designed for large IP sets and frequent updates.
- 2-clause BSD license: liberal and vendor-friendly.
- IPv6 support, extensions for normalisation, logging and more!

Design

Packet classification engine – BPF

NPF packet classification engine i.e. rule processing is based on byte-code instruction processing.

- NPF uses BPF byte-code with JIT compilation.
- This design allows us to have protocol independence, e.g. support for a new protocol can be added without any modifications to the kernel part.
- sljit[1] is used for JIT compilation. The compiler supports various architectures, is also used by the PCRE library and is reasonably tested and benchmarked.
- However, the original BPF instruction set is limited: it cannot perform complex operations, e.g. table lookup.

[1] http://sljit.sourceforge.net/

Design

BPF COP

BPF was extended with "coprocessor" support for offloading complex operations.

- BPF coprocessor honours the tradition of RISC-like instruction sets, but the debate whether BPF should grow some complex instructions (e.g. to handle IPv6 headers) is still on.
- Two new instructions in the misc category: BPF_COP and BPF_COPX. They call a **predetermined** function using an array index. The functions can only be set by the kernel.
- They can read the packet in a read-only manner, use the memstore and return a value. They cannot change the flow, so BPF byte-code does not become Turing-complete.

Design

BPF everywhere

- Additionally, NPF also supports pcap(3) – its syntax and capabilities. Virtually any filter pattern can be constructed. An example:

```
block out final pcap-filter "dst 10.1.1.252 and ip[2:2] > 576"
```

- By the way: the idea of unifying all packet classification engines under BPF is not new. It has been floating around for, at least, few decades...

Design

Rules

- There are **static rules** and **dynamic rules**. The former are loaded together with the configuration. The latter can be added/removed on the fly.
- A group is a rule which has sub-rules. Therefore, the rules in NPF can be **nested** (there is an artificial limit, though).
- In the kernel, the list of static rules is represented as an array with jump/skip marks. Therefore, rule inspection is a simple non-recursive iteration which, as a side note, is also cache-friendly.

Design

Ruleset

- Ruleset reload is performed as a single one step commit with a minimum performance impact on the packet processing.
- The ruleset is protected using *passive serialisation*[2]. Hence, the ruleset inspection is lockless.

[2] Similar concept to RCU, but patent-free

Design

Dynamic rules

Dynamic rules can be added/removed on the fly, without reloading the entire configuration. Some notes:

- Each rule gets a unique identifier which is returned on addition.
- Also, SHA1 hash is calculated on rule meta data and therefore rule can be removed given its definition/filter criteria.
- The rule can be reliably removed using the unique ID. This is the more efficient and recommended way.
- While rule inspection is lockless, rule addition or removal has significant overhead.

Design

Dynamic rules

Example:

```
$ npfctl rule "test-set" add block proto tcp from 192.168.0.6
OK 1
$ npfctl rule "test-set" list
block proto tcp from 192.168.0.6
$ npfctl rule "test-set" add block from 192.168.0.7
OK 2
$ npfctl rule "test-set" list
block proto tcp from 192.168.0.6
block from 192.168.0.7
$ npfctl rule "test-set" rem block from 192.168.0.7
$ npfctl rule "test-set" rem-id 1
$ npfctl rule "test-set" list
$
```

Design

Stateful inspection

NPF supports stateful filtering – costly, but demanded feature.

- It performs full tracking of TCP connections. This means not only tracking of source and destination IP addresses with port numbers, but also TCP state, sequence numbers and window sizes.
- Tracked connections are stored in a hash table with a red-black tree per bucket, protected by a read-write lock.
- The hash table distributes the locks and thus significantly reduces the lock contention.
- The tree prevents from DoS attacks exploiting hash collisions and O(n) behaviour.

Design

Stateful inspection

- In NPF, the state is uniquely identified by a 6-tuple.
- Bypassing the ruleset on other interfaces can have undesirable effects, e.g. a packet with a spoofed IP address might bypass ingress filtering.
- However, there are legitimate cases when bypassing on other interfaces is safe and can increase the performance.
- Therefore, **stateful-ends** keyword was added to perform the state lookup on other interfaces as well.

Design

Stateful inspection

The current performance of state lookup is "good enough", but not optimal.

- State inspection involves 6-tuple lookup. Performing both the hash calculation and the tree iteration has a cost. Read-write locks suffer from *cache-line bouncing* effect.
- The current work is to replace hashed trees with more efficient data structure – a lockless and cache-aware B+ tree.[3]
- Very preliminary results indicate ∼**2x** faster state lookup with **linear** scalability!

[3] Masstree by Y. Mao, E. Kohler and R. Morris

Design

Network address translation (NAT)

NPF supports dynamic (stateful) and static (stateless) NAT.

- Inbound/source and outbound/destination NAT.
- Address-port translation (NAPT/masquerading) or just port translation (forwarding).
- Bi-directional NAT (a combination of inbound and outbound).
- Pretty much any variations can be defined using a single expressive form of syntax:

```
map = "map" interface
      ( "static" [ "algo" algorithm ] | "dynamic" )
      net-seg ( "->" | "<-" | "<->" ) net-seg
      [ "pass" filtopts ]
```

Design
NAT

NPF has also grown support for IPv6 Network Prefix Translation, as described in RFC 6296:

```
$net6_inner = fd01:203:405::/48
$net6_outer = 2001:db8:1::/48

map $ext_if static algo npt66 $net6_inner <-> $net6_outer
```

NPTv6 is a static NAT with a particular algorithm specified.

Design

Tables

Large IP sets can be stored in NPF tables for very efficient lookups. NPF tables are similar to the "ipset" module of Linux Netfilter.

- **Hash**: provides amortised $O(1)$ lookup time and lockless lookup. Obviously, it suffers from collisions and is not suitable for growing sets. Future work: lockless rehash.
- **Tree**: implemented using PATRICIA tree, therefore provides $O(k)$ lookup time and is more suitable for dynamic sets. However, protected with read-write lock. Future work: lockless prefix tree.
- **CDB**: constant database uses perfect hashing and thus guarantees $O(1)$ and lockless lookup. Ideal for sets which rarely change.

Design

Modularity

- NPF is modular, each component is abstracted and has its own strict interface.
- *Rule procedures* in NPF are a key interface to implement custom extensions. The syntax of npf.conf supports arbitrary procedures with their parameters, as supplied by the modules.
- An extension consists of two parts: a dynamic module (.so file) supplementing the npfctl(8) utility and a kernel module.
- Just ∼**160 lines of code** for a demo extension, which blocks an arbitrary percentage of traffic. No modifications required to the NPF core or npfctl(8)!

Testing

Running and debugging NPF in the userspace

- For testing, NPF uses NetBSD's RUMP (Runnable Userspace Meta Programs) framework – a kernel virtualisation and isolation technique, which enables running of the NetBSD kernel or parts of it in the userspace, like a regular program.
- For example, you can run NetBSD's TCP/IP stack as a regular program and pass other applications through it.[4]
- Makes debugging or profiling significantly easier due to availability of tools such as gdb(1).
- NPF regression tests are integrated into NetBSD's test suite and thus are part of periodic automated runs.

[4] https://github.com/anttikantee/buildrump.sh

- There are unit tests for every NPF subsystem. They are available within npftest(8) – a program containing both the tests and NPF kernel part running as a userspace program.
- npftest(8) can also read and process tcpdump pcap files with a passed npf.conf configuration. This enables analysis of a particular stream or connection in the userspace.
- The npfctl(8) utility has a 'debug' command which can print disassembled BPF byte-code and dump the configuration in the format sent to the kernel.
- Development, debugging and testing becomes much easier.

Testing

Testing and playing

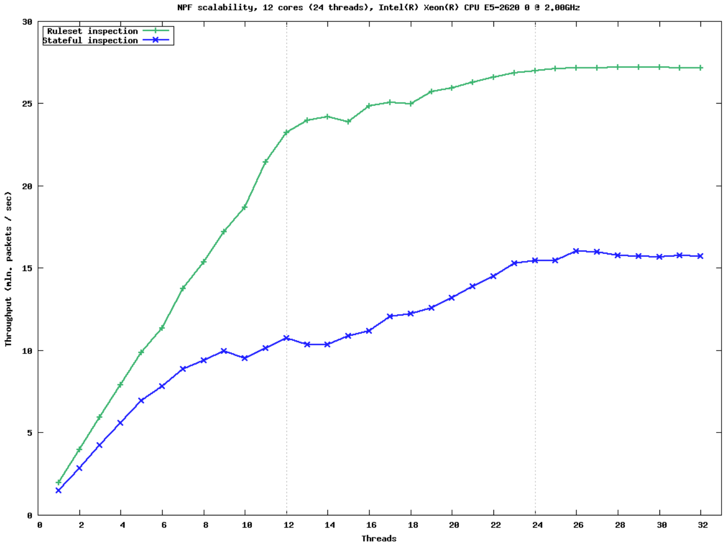
- Recently, NPF has gained support in rumprun project.[5]
- You can spawn RUMP kernels as regular programs and setup a network amongst them. For example, you can spawn a bunch of servers and test NAT.
- Can be done in a simple shell script (∼50 lines) and be spawned in a second!

[5] `https://github.com/rumpkernel/rumprun`

Scalability

So, can we demonstrate the scalability of NPF?

Scalability



Future directions

- Porting to FreeBSD and illumos is under consideration.
- High availability, load balancing.
- QoS: rate limiting, traffic shaping.
- More extensions.

Documentation

`http://www.netbsd.org/~rmind/npf/`

End

TH_FIN

The **Net**BSD Project

http://www.NetBSD.org/

2014