Xtables2: Love for blobs

Jan Engelhardt <jengelh@inai.de>

Presented at NFWS 2010

2010-Oct-18
Current status

- ip_tables started with a packed serialized ruleset ("blob" – binary large object)
Current status

- `ip_tables` started with a packed serialized ruleset ("blob" – binary large object)
- `ip6_tables` is a copy-and-paste product of `ip_tables`. And so is `arp_tables`. And so is `ebtables`. Yuck!
Current status

- ip_tables started with a packed serialized ruleset ("blob" – binary large object)
- ip6_tables is a copy-and-paste product of ip_tables. And so is arp_tables. And so is ebtables. Yuck!
- Changes to ip_tables could still be mirrored to ip6_tables and arp_tables
- ebtables took its own incompatible path of development
Current status

- ip_tables started with a packed serialized ruleset ("blob" – binary large object)
- ip6_tables is a copy-and-paste product of ip_tables. And so is arp_tables. And so is ebtables. Yuck!
- Changes to ip_tables could still be mirrored to ip6_tables and arp_tables
- ebtables took its own incompatible path of development
- Combined with compat support, there are now seven formats to support in the kernel
- A big itch to scratch.
Current status

- ip_tables started with a packed serialized ruleset ("blob" – binary large object)
- ip6_tables is a copy-and-paste product of ip_tables. And so is arp_tables. And so is ebtables. Yuck!
- Changes to ip_tables could still be mirrored to ip6_tables and arp_tables
- ebtables took its own incompatible path of development
- Combined with compat support, there are now eight formats to support in the kernel
- Eight itches to scrub.
A protocol-independent format

- Rule tree without protocol-specific parts in it, to be used by and for all protocol handlers
- Translation from and to input formats on-the-fly, i.e. during `SO_SET_REPLACE/etc.`
A protocol-independent format

- Rule tree without protocol-specific parts in it, to be used by and for all protocol handlers
- Translation from and to input formats on-the-fly, i.e. during SO_SET_REPLACE/etc.
- Formats are just minimally different: serialized stream of struct ipt_entry vs. struct ip6t_entry
A protocol-independent format

- Rule tree without protocol-specific parts in it, to be used by and for all protocol handlers
- Translation from and to input formats on-the-fly, i.e. during SO_SET_REPLACE/etc.
- Formats are just minimally different: serialized stream of `struct ipt_entry` vs. `struct ip6t_entry`

⇒ Led to Xtables2
Developments

SSA/LL\textsuperscript{1,2} style:

- “proto1”: initial submission on 2009-Aug-04 for v2.6.31-rc (103 patches)
- busy dealing with cleanups: 46/103

\textsuperscript{1}Small scale allocations, or small scattered allocations, combined with linked lists

\textsuperscript{2}Has nothing to do with GCC’s SSA
Developments

SSA/LL\textsuperscript{1,2} style:

- “proto1”: initial submission on 2009-Aug-04 for v2.6.31-rc (103 patches)
- busy dealing with cleanups: 46/103
- “proto2”: partial set posted on 2010-Jun-04 for v2.6.35-rc (33 patches, and a nasty surprise)

\textsuperscript{1}Small scale allocations, or small scattered allocations, combined with linked lists
\textsuperscript{2}Has nothing to do with GCC’s SSA
Developments

SSA/LL\textsuperscript{1,2} style:

- “proto1”: initial submission on 2009-Aug-04 for v2.6.31-rc (103 patches)
- busy dealing with cleanups: 46/103
- “proto2”: partial set posted on 2010-Jun-04 for v2.6.35-rc (33 patches, and a nasty surprise)
- “proto3”: simple rebase for v2.6.36-rc for better comparison with the upcoming proto4

PCR style:

- “proto4”: xt2 using packed-chain rulesets, for v2.6.36-rc

\textsuperscript{1}Small scale allocations, or small scattered allocations, combined with linked lists
\textsuperscript{2}Has nothing to do with GCC’s SSA
**Chosen data layout**

- Linked lists allow for “easy manipulation” of the ruleset.
- Small-scale allocations (SSA) are more easily satisfiable.
Chosen data layout

- Linked lists allow for “easy manipulation” of the ruleset.
- Small-scale allocations (SSA) are more easily satisfiable.
- Prototype: Translators work nicely, and with a bit of macro magic, eliminated 40% of LOC from the \{ip,ip6,arp\} combo.
Ruleset

- Just a simple ruleset that would be large enough so that wall time is visible
Ruleset

- Just a simple ruleset that would be large enough so that wall time is visible

Just struct ip6t_entry, but lots of them

-A $chain -s ::1 -d ::1

- no extensions, just struct ip6t_entry × 1000 rules × 100 chains reachable from INPUT (OUTPUT is left empty)
Ruleset

- Just a simple ruleset that would be large enough so that wall time is visible

Just struct `ip6t_entry`, but lots of them

```
-A $chain -s ::1 -d ::1
```

- no extensions, just `struct ip6t_entry × 1000 rules × 100 chains` reachable from INPUT (OUTPUT is left empty)
- 100,202 rules (100,000 base rules + 100 calls + 100 implicit invisible RETURNs converted from Xt1 + 2 implicit Xt1 RETURNs from base chains)
- ≈20 MB in packed form
Comparison with real rulesets

Jesper has down-to-earth rulesets:
67,892 visible rules in 18,329 chains: rule density distribution
> summary(data)

Min. 1st Qu. Median Mean 3rd Qu. Max.
1.000 1.000 2.000 3.745 4.000 119.000

Packed size is 16,866,200 bytes

Design: fanned tree, only
≈ 53 rules executed per packet

Low rule density sounds like management overhead – need to keep that in mind for later
Comparison with real rulesets

- Jesper has down-to-earth rulesets:

  67,892 visible rules in 18,329 chains: rule density distribution

  > summary(data)
  
  Min. 1st Qu.  Median    Mean  3rd Qu.  Max.  
  1.000  1.000  2.000    3.745  4.000  119.000

- Packed size is 16,866,200 bytes

- Design: fanned tree, only \(\approx 53\) rules executed per packet
Comparison with real rule sets

- Jesper has down-to-earth rulesets:

  **67,892 visible rules in 18,329 chains: rule density distribution**

  ```
  > summary(data)
  Min. 1st Qu. Median  Mean 3rd Qu. Max.
  1.000 1.000 2.000 3.745 4.000 119.000
  ```

- Packed size is 16,866,200 bytes
- Design: fanned tree, only \( \approx 53 \) rules executed per packet
- Low rule density sounds like management overhead – need to keep that in mind for later
Comparison with real rule sets

- Jesper has down-to-earth rule sets:

  662,160 visible rules in 151,426 chains: rule density distribution

  ```r
  > summary(data)
  Min. 1st Qu.  Median   Mean 3rd Qu.    Max.
   1.00    1.00    4.00    4.477   4.00   144.000
  ```

- Packed size is 156,258,112 bytes
- Design: fanned tree, only \approx 77 rules executed per packet
- Low rule density sounds like management overhead – need to keep that in mind for later
Test procedure

100,000 × struct ip6t_entry
-A mychain$i -s ::1 -d ::1

- Earlier tests with ping6 -f were flawed.

Testing proto2

ping6 -fqc 500 -i .001 localhost
Test procedure

100,000\times \text{struct ip6t\_entry}

-A mychain\$i -s ::1 -d ::1

- Earlier tests with ping6 -f were flawed.

Testing proto2

ping6 -fqc 500 -i .001 localhost

- Without rules, this gives 500 ms total execution time: packet handling is quick, ping is just waiting for the intervals to expire.
- -i .001 made sure that (with rules) no packets were reported dropped
- With rules, this goes up: once it starts going above 500 ms, we know packet processing takes longer than the 1 ms interval.
So-gathered statistics showed an execution time expansion of 4.30× (xt1: 3500 ms → proto2: 15000 msec)

“Linked lists no good?”
Results

- So-gathered statistics showed an execution time expansion of $4.30 \times$
  (xt1: 3500 ms $\rightarrow$ proto2: 15000 msec)
- “Linked lists no good?”
- Using ping this way was flawed... ping handles packets asynchronously when using $-f$
- Let’s reset.
Test procedure

Testing proto3 with revised command

```
ping6 -Ac 500 ::1
```

Observing ping's RTT statistics rather than execution time

Additionally, I sampled the CPU cycle counter around `xt2_do_table` and the `ematch` loop in `xt2_do_actions` ⇒ much more consistent results
Test procedure

Testing proto3 with revised command

```
ping6 -Ac 500 ::1
```

- Observing ping’s RTT statistics rather than execution time
Test procedure

Testing proto3 with revised command

```
ping6 -Ac 500 ::1
```

- Observing ping’s RTT statistics rather than execution time
- Additionally, I sampled the CPU cycle counter around `xt2_do_table` and the ematch loop in `xt2_do_actions`

⇒ much more consistent results
Results

- Expansion factor: $2.80 \times (xt1: 40.477 \text{ ms} \rightarrow \text{proto3: 113.424 ms})$
- Increase expected (being a pessimist), but this much still blew everything

---

Results

- Expansion factor: $2.80 \times (\text{xt1: } 40.477 \text{ ms } \rightarrow \text{proto3: } 113.424 \text{ ms})$
- Increase expected (being a pessimist), but this much still blew everything
- Speculation: lots of D-cache misses\(^3\) due to the objects being “spread out” in memory

---

Results

- Expansion factor: $2.80 \times (\text{xt1: 40.477 ms} \rightarrow \text{proto3: 113.424 ms})$
- Increase expected (being a pessimist), but this much still blew everything
- Speculation: lots of D-cache misses due to the objects being “spread out” in memory
- Use of `kmem_cache` pools for objects of constant size (table, chain and rule list heads) showed no improvement

---

Results

- Expansion factor: $2.80 \times \text{(xt1: 40.477 ms $\rightarrow$ proto3: 113.424 ms)}$
- Increase expected (being a pessimist), but this much still blew everything
- Speculation: lots of D-cache misses\(^3\) due to the objects being “spread out” in memory
- Use of `kmem_cache` pools for objects of constant size (table, chain and rule list heads) showed no improvement
- And then there was memory...

Memory usage

Previously, with a blob:

- 1 vmalloc’d object of $\approx 20$ MB
Memory usage

Previously, with a blob:

- 1 vmalloc’d object of ≈20 MB

Now, split allocations...?

- SL*B has to housekeep an 1,002,111 extra kmalloc’d objects now
  - 1× struct xt2_table
  - 100× struct xt2_chains
  - 100,201× struct xt2_rules
  - 100,201× struct xt2_entry_match for “ipv6”
  - 100,201× struct ip6t_ip6 for “ipv6”
  - 200,402× struct xt2_entry_match for “quota”
  - 200,402× struct xt_quota for “quota”
  - 200,402× struct xt_quota_priv for “quota”
  - 100,201× struct xt2_entry_target for implicit CONTINUE

This is of course the other end of the two extremes.

Memory usage increase of 2.7× (i586).

[proc/slabinfo]:
≈900,000× size-32
≈100,000× size-192
48 MB, plus some housekeeping, for a total of

Layman’s observation:

Free: ip6tables-restore bigrules; free

Used free:

-/+ buffers/cache: 34056 1002172
-/+ buffers/cache: 86392 949836

⇒ Small scattered allocations are a no-go.
Memory usage

Previously, with a blob:

- 1 vmalloc’d object of \( \approx 20 \) MB

Now, split allocations...?

- SL*B has to housekeep an 1,002,111 extra kmalloc’d objects now
- Memory usage increase of \( 2.7 \times \) (i586).
  
  /proc/slabinfo:
  
  - \( \approx 900,000 \times \) size-32
  - \( \approx 100,000 \times \) size-192
  
  48 MB, plus some housekeeping, for a total of \( \approx 53 \) MB

Layman’s observation

```
# free; ip6tables-restore bigrules; free

used free
-/+ buffers/cache: 34056 1002172
-/+ buffers/cache: 86392 949836
```
Memory usage

Previously, with a blob:

- 1 vmalloc’d object of \(\approx 20 \text{ MB}\)

Now, split allocations...?

- SL*B has to housekeep an 1,002,111 extra kmalloc’d objects now
- Memory usage increase of \(2.7 \times (i586)\).

/proc/slabinfo:

- \(\approx 900,000 \times \text{size-32}\)
- \(\approx 100,000 \times \text{size-192}\)
- 48 MB, plus some housekeeping, for a total of \(\approx 53 \text{ MB}\)

Layman’s observation

```bash
# free; ip6tables-restore bigrules; free

 used free
-/+ buffers/cache: 34056 1002172
-/+ buffers/cache: 86392 949836
```

⇒ Small scattered allocations are a no-go.
Section TOC
Love for blobs

- Evaluation of rules: we want no scattered allocs
- Housekeeping: we want few allocs
Ideas for fixing Packed rulesets

Love for blobs

- Evaluation of rules: we want no scattered allocs
- Housekeeping: we want few allocs
- Original iptables design decision pays off (Harald was right all along!)
  - packed ruleset allows for streaming reads
  - ipfw and pf use linked lists

Jan Engelhardt (NFWS2010)
Love for blobs

- Evaluation of rules: we want no scattered allocs
- Housekeeping: we want few allocs
- Original iptables design decision pays off (Harald was right all along!)
  - packed ruleset allows for streaming reads
  - ipfw and pf use linked lists
- Let’s try concentrating on packed rulesets again (kernel side only)
Love for blobs

- Evaluation of rules: we want no scattered allocs
- Housekeeping: we want few allocs
- Original iptables design decision pays off (Harald was right all along!)
  - packed ruleset allows for streaming reads
  - ipfw and pf use linked lists
- Let’s try concentrating on packed rulesets again (kernel side only)

Need to find ways to make working with them easier
**Love for blobs**

- Evaluation of rules: we want no scattered allocs
- Housekeeping: we want few allocs
- Original iptables design decision pays off (Harald was right all along!)
  - packed ruleset allows for streaming reads
  - ipfw and pf use linked lists `<°}))))>`
- Let’s try concentrating on packed rulesets again (kernel side only)

Need to find ways to make working with them easier

- A good API is half the job
- Algorithms to keep the time cost of updating rulesets in-place low
About APIs

- Opaque macros/functions gone too opaque

**IP6T_MATCH_ITERATE**

```c
struct compat_ip6t_entry *e = ...;
ret = COMPAT_IP6T_MATCH_ITERATE(e, compat_find_calc_match, name, &e->ipv6, e->comefrom, &off, &j);
```

**xt_ematch_foreach**

```c
struct compat_ip6t_entry *e = ...;
struct xt_entry_match *ematch;
xt_ematch_foreach(ematch, e) {
    ret = compat_find_calc_match(ematch, name, &e->ipv6, e->comefrom, &off);
    if (ret != 0)
        break;
    ++j;
}
```
Ideas for fixing API guide

- Implementation is also much friendlier to long-term maintainers
- `xt_ematch_foreach` is KISS and may save function call overhead

### IP6T_MATCH_ITERATE

```c
#define XT_MATCH_ITERATE(type, e, fn, args...) \
({ \
    unsigned int i; \ 
    int ret; \ 
    struct xt_entry_match *m; \ 
    for (i = sizeof(type); i < e->target_offset; i += m->u.match_size) { \ 
        m = e + i; \ 
        ret = fn(m, ## args); \ 
        if (ret != 0) \ 
            break; \ 
    } \ 
    ret; \ 
})
```

### xt_ematch_foreach

```c
#define xt_ematch_foreach(pos, entry) \
for (pos = entry->elems; \ 
    pos < entry + entry->target_offset; \ 
    pos = pos + pos->u.match_size)
```
Blobs for ¥100: Single rules

Xt1 blob rules refer to chains (when jumping) by their absolute offset in the blob (i.e. bytes from the start of the blob). Insertion or deletion of a chain/rule in a blob shifts the offset of all subsequent chains. Requires updating the chain offsets of all jumping rules. With $k$ rules already loaded, that is $O(k)$. Adding $n$ rules leads to $O(n^2)$ behavior – ouch.

Userspace iptables(8) still submits entire tables, but translation process does currently add one rule at a time to xt2 however. Important to keep in mind for future fine-grained modifications initiated from userspace.
Blobs for ¥100: Single rules

- Xt1 blob rules refer to chains (when jumping) by their absolute offset in the blob (i.e. bytes from the start of the blob)
Blobs for ¥100: Single rules

- Xt1 blob rules refer to chains (when jumping) by their absolute offset in the blob (i.e. bytes from the start of the blob)
- Insertion or deletion of a chain/rule in a blob shifts the offset of all subsequent chains
- Requires updating the chain offsets of all jumping rules
- With $k$ rules already loaded, that is $O(k)$
Blobs for ¥100: Single rules

- Xt1 blob rules refer to chains (when jumping) by their absolute offset in the blob (i.e. bytes from the start of the blob)
- Insertion or deletion of a chain/rule in a blob shifts the offset of all subsequent chains
- Requires updating the chain offsets of all jumping rules
- With \( k \) rules already loaded, that is \( \mathcal{O}(k) \)
- Adding \( n \) rules leads to \( \mathcal{O}(n^2) \) behavior – ouch
- Userspace iptables(8) still submits entire tables, but translation process does currently add one rule at a time to xt2 however
- Important to keep in mind for future fine-grained modifications initiated from userspace
Blobs for ¥200: Bulk operations

Insertion of rules can be batched; reservation of enough bytes at once:

\[
\text{new} = \text{malloc} (\text{cur\_size} + x);
\]

\[
\text{memcpy} (\text{new}, \text{cur\_ruleset}, \text{ins\_offset});
\]

\[
\text{memcpy} (\text{new} + \text{ins\_offset} + x, \text{cur\_ruleset} + \text{ins\_offset}, \text{cur\_size} - \text{ins\_offset});
\]

Process is similar for bulk deletion

Largest contiguous block is the set of rules of a chain

Therefore, with \(c\) chains, a bulk update would only be

\[
O (c \cdot n)
\]

Still suboptimal: Consider low rule density from earlier:

\(n^c \rightarrow 1\)

\[
\lim_{c \rightarrow n} O (c \cdot n) = O (n^2)
\]
Blobs for ¥200: Bulk operations

- Insertion of rules can be batched; reservation of enough bytes at once:

```
Multi-rule reservation also in \( O(k) \)

new = malloc(cur_size + x);
memcpy(new, cur_ruleset, ins_offset);
memcpy(new + ins_offset + x, cur_ruleset + ins_offset, cur_size -
ins_offset);
```

- Process is similar for bulk deletion
Blobs for ¥200: Bulk operations

- Insertion of rules can be batched; reservation of enough bytes at once:

```c
new = malloc(cur_size + x);
memcpy(new, cur_ruleset, ins_offset);
memcpy(new + ins_offset + x, cur_ruleset + ins_offset, cur_size - ins_offset);
```

Multi-rule reservation also in $O(k)$

- Process is similar for bulk deletion
- Largest contiguous block is the set of rules of a chain
- Therefore, with $c$ chains, a bulk update would only be $O(c \cdot n)$
Blobs for ¥200: Bulk operations

- Insertion of rules can be batched; reservation of enough bytes at once:
  
  
  
  Multi-rule reservation also in $O(k)$

  ```c
  new = malloc(cur_size + x);
  memcpy(new, cur_ruleset, ins_offset);
  memcpy(new + ins_offset + x, cur_ruleset + ins_offset, cur_size - ins_offset);
  ```

- Process is similar for bulk deletion
- Largest contiguous block is the set of rules of a chain
- Therefore, with $c$ chains, a bulk update would only be $O(c \cdot n)$
- Still suboptimal: Consider low rule density from earlier:
  \[
  \frac{n}{c} \to 1 \implies \lim_{c\to n} O(c \cdot n) = O(n^2)
  \]
Can we get rid of the costly updates?
Ideas for fixing

Update cost considerations

Blobs for ¥500: Indirect addressing

- Can we get rid of the costly updates?

Yes, in two stages. Number one:

**Indirect chain lookup**

```c
next_rule = tbl->blob +
    tbl->chain_offset[rule->chain_index]
```

- (cf. Xtl: `next_rule = tbl->blob + rule->jump_offset`)

On rule insertion/deletion, only `chain_offset` needs to be adjusted, for $O(c)$. Still has other costs: chain head deletion is $O(k)$ (can be mitigated by lazy deletion).
Blobs for ¥500: Indirect addressing

- Can we get rid of the costly updates?

Yes, in two stages. Number one:

**Indirect chain lookup**

\[
\text{next\_rule} = \text{tbl}\rightarrow\text{blob} + \\
\text{tbl}\rightarrow\text{chain\_offset}[\text{rule}\rightarrow\text{chain\_index}]
\]

- (cf. Xt1: next\_rule = tbl\rightarrow\text{blob} + rule\rightarrow\text{jump\_offset})
- On rule insertion/deletion, only chain\_offset needs to be adjusted, for \( \mathcal{O}(c) \).
Blobs for ¥500: Indirect addressing

- Can we get rid of the costly updates?

Yes, in two stages. Number one:

**Indirect chain lookup**

\[
\text{next\_rule} = \text{tbl}\rightarrow\text{blob} + \\
\text{tbl}\rightarrow\text{chain\_offset}[\text{rule}\rightarrow\text{chain\_index}]
\]

- (cf. Xt1: next_rule = tbl->blob + rule->jump_offset)
- On rule insertion/deletion, only chain_offset needs to be adjusted, for \(O(c)\).
- Still has other costs: chain head deletion is \(O(k)\) (can be mitigated by lazy deletion).
Blobs for ¥1,000: Decoupled chains

- Prediction/Assumption: Since jumps can go across the entire blob, D-cache won’t help anyway
- Loosen up on strict packing, just a little
Blobs for ¥1,000: Decoupled chains

- Prediction/Assumption: Since jumps can go across the entire blob, D-cache won’t help anyway
- Loosen up on strict packing, just a little
- Let largest contiguous entity be the chain rather than table
Blobs for ¥1,000: Decoupled chains

- Prediction/Assumption: Since jumps can go across the entire blob, D-cache won’t help anyway
- Loosen up on strict packing, just a little
- Let largest contiguous entity be the chain rather than table
- Combined with indirect chain lookup, no chain offset updates needed at all.
xt2 sample chain head

```c
struct xt2_chain {
    char name[XT_EXTENSION_MAXNAMELEN];
    void *rule_blob;
};
```

Jump action

```c
struct xt2_packed_etarget *target;
next_rule = target->r_jump->rule_blob;
```

- `&some_xt2_chain` always remains the same over its lifetime – no more updates of rules required
Results

- 100k rules like before, measuring RTT again

Testing RTT for proto4

ping6 -Ac 500 ::1
Results

- 100k rules like before, measuring RTT again

Testing RTT for proto4

```
ping6 -Ac 500 ::1
```

- Observed expansion: $1.83 \times (\text{xt1: 40.477 ms} \rightarrow \text{proto4: 74.135 ms})$
- Splendid! Packed-chain rulesets work.
Results

- 100k rules like before, measuring RTT again

Testing RTT for proto4

ping6 -Ac 500 ::1

- Observed expansion: $1.83 \times (xt1: 40.477 \text{ ms} \rightarrow \text{proto4: 74.135 ms})$
- Splendid! Packed-chain rulesets work.
- But what’s with the remaining 83%?
Rule counters in Xtables2

- xt2 rules carry absolutely nothing per default
- Per-rule counters are temporarily implemented by using two xt_quota ematches in upcounting mode
Rule counters in Xtables2

- xt2 rules carry absolutely nothing per default
- Per-rule counters are temporarily implemented by using two xt_quota ematches in upcounting mode
  - The “ipv6” match with -s ::1 -d ::1 runs in 200–300 cycles
  - One “quota” ematch takes prohibitely costly 4500 cycles
Rule counters in Xtables2

- xt2 rules carry absolutely nothing per default
- Per-rule counters are temporarily implemented by using two `xt_quota` ematches in upcounting mode
  - The “ipv6” match with `-s ::1 -d ::1` runs in 200–300 cycles
  - One “quota” ematch takes prohibitely costly 4500 cycles
  - (In)significance of raw cycle counts
  - Does not tell whether PCR might still incur a bottleneck
  - Main function of `xt_quota` is only 19 LOC, but `xt_ipv6`'s is 79 LOC.
Equal-power comparison

- A INPUT -s ::1 -d ::1 -m quota --grow -m quota --grow

- Driving xt1 with xt_quota counters yields an RTT of 77.373 ms.
Equal-power comparison

Just as costly

```bash
-A INPUT -s ::1 -d ::1 -m quota --grow -m quota --grow
```

- Driving xt1 with xt_quota counters yields an RTT of 77.373 ms.
- Xtables2 PCR (74.135 ms) is absolutely on par
- xt_quota is the one and only bottleneck
xt_quota analysis

- Using the simplest possible counter implementation instead of full-featured xt_quota, proto4 execution time drops to 44.254 ms.
xt_quota analysis

- Using the simplest possible counter implementation instead of full-featured xt_quota, proto4 execution time drops to 44.254 ms.
- Adding a kmalloc for a private data structure to this simple impl. and time jumps to 50.733 ms (= +15%).
- D-cache misses – again!? 
Future

Roadmap:
- Continue using packed rulesets for packet processing

Deemed solvable:
- Optimize extensions to contain fewer far-away accesses

Deemed infeasbly solvable:
- ebtables
Questions

- I know you have some!
Questions

- K7: AMD K7 Athlon 1.66GHz (manuf. 2003) 256K cache 2.6.36
- i7: Intel Core i7 920 4-core 2.67GHz (2009) 8MB 2.6.33
- VM: VirtualBox machine 1-core on i7 2.6.36

<table>
<thead>
<tr>
<th>Driver</th>
<th>RTT K7</th>
<th>RTT i7</th>
<th>RTT VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>xt1 +2s</td>
<td>40.447</td>
<td>2.83</td>
<td>3.08</td>
</tr>
<tr>
<td>xt1 +1Q</td>
<td>58.882</td>
<td>5.18</td>
<td>11.47</td>
</tr>
<tr>
<td>xt1 +2Q</td>
<td>77.373</td>
<td>11.50</td>
<td>21.00</td>
</tr>
<tr>
<td>xt2-proto3 +2Q</td>
<td>113.424</td>
<td>n/a</td>
<td>24.47</td>
</tr>
<tr>
<td>xt2-proto4 +2Q</td>
<td>74.135</td>
<td>n/a</td>
<td>21.79</td>
</tr>
<tr>
<td>xt2-proto4 +2s</td>
<td>44.254</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

s: simple local counters
Q: xt_quota-based counters
Questions

- K7: AMD K7 Athlon 1.66GHz (manuf. 2003) 256K cache 2.6.36
- i7: Intel Core i7 920 4-core 2.67GHz (2009) 8MB 2.6.33
- VM: VirtualBox machine 1-core on i7 2.6.36

<table>
<thead>
<tr>
<th>Driver</th>
<th>RTT K7</th>
<th>RTT i7</th>
<th>RTT VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>xt1 +2s</td>
<td>40.447</td>
<td>2.83</td>
<td>3.08</td>
</tr>
<tr>
<td>xt1 +1Q</td>
<td>58.882</td>
<td>5.18</td>
<td>11.47</td>
</tr>
<tr>
<td>xt1 +2Q</td>
<td>77.373</td>
<td>11.50</td>
<td>21.00</td>
</tr>
<tr>
<td>xt2,proto3 +2Q</td>
<td>113.424</td>
<td>n/a</td>
<td>24.47</td>
</tr>
<tr>
<td>xt2,proto4 +2Q</td>
<td>74.135</td>
<td>n/a</td>
<td>21.79</td>
</tr>
<tr>
<td>xt2,proto4 +2s</td>
<td>44.254</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>nft +2s</td>
<td>57.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- s: simple local counters
- Q: xt_quota-based counters