What we talk about when we talk about pcap expressions

Nik Sultana
University of Pennsylvania

ABSTRACT

Pcap expressions are a popular domain-specific language for packet filters, used extensively to capture network traffic during network monitoring or testing. The language is simple but made nontrivial by implicit subformulas and other features that can lead to unexpected and unwanted results.

This paper formalises the semantics of pcap expressions by expanding them to disambiguate their meaning, and describes an SMT-based approach to check the equivalence of pcap expressions. This approach can be used to obtain increased confidence in the semantics of a pcap expression, by checking it against other expressions with which it is expected to be equivalent. These semantics are implemented into an open-source practical tool to help users of pcap expressions.

CCS CONCEPTS

• **Networks → Network properties**; *Network protocols*; • **Theory of computation → Semantics and reasoning**;

KEYWORDS

domain-specific languages, formal semantics, packet filtering

ACM Reference Format:
Nik Sultana. 2019. What we talk about when we talk about pcap expressions. In *Real World Domain Specific Languages (RWDSL), February 17, 2019, Washington D. C., DC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The BSD Packet Filter [19] (BPF) is the most successful host-based system for capturing network packets. It is virtually ubiquitous through the portable `libpcap` library, on which the `tcpdump` command-line program is based [1]. It builds on the work of Mogul et al. [20] to provide flexible yet performant packet capture: flexibility is provided by accepting *packet capture expressions* from the user, and performance is improved by interpreting these expressions as packet filters in the OS kernel to reduce overhead from userspace-kernel transitions.

The packet-filtering expressions accepted by `libpcap` form a domain-specific language (DSL) which we refer to as *pcap expressions*, distinguishing them from the *pcap file* format that is used to store captured packets.

Pcap expressions are used extensively to capture network traffic when testing new systems and diagnosing network problems. Once captured, packets can be immediately analysed "online", for network-intrusion detection [27] for example. Alternatively, packets captured from a network link could be stored in a pcap file for offline analysis.

RWDSL, February 17, 2019, Washington D. C., DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$0
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper we take a close look at the pcap expression language. The language is simple but made nontrivial by implicit subformulas and other features that can lead to unexpected and unwanted results. This paper provides an approach for giving formal semantics to the pcap expression language. These semantics are then implemented in an equivalence-checking tool for pcap expressions. This tool, called Caper, is intended to assist pcap users, and is made freely available as open-source software.[1]

The conciseness and intuitiveness of pcap expressions has made the language very popular. For example, the following expression describes a filter for traffic that is directed at an unspecified Web server – the filter matches packets whose destination port is that used by HTTP:

```
dst port http                                   (1)
```

Pcap expressions can contain clauses that are implicit, and these might deviate from the user's intended filter. For example, the expression shown above does not specify whether the packets are carried over a IPv4 or IPv6 network, nor which transport layer protocol is used. By inspecting the machine code to which the filter is translated [19] we find its behaviour to be equivalent to this pcap expression:

```
(ip6 or ip) and (sctp or tcp or udp) and
             (dst port http)
```

Two things worth pointing out:

- The two pcap expressions are *equivalent*, meaning that they match (and reject) the same packets. The machine code for the two filters is *almost* identical, and differ in one small detail: the second filter explicitly rejects fragmented IPv6 packets [10], while the first filter simply does not accept them. The original expression (1) results in an *identical* filter to this expression:

  ```
  (ip6 or ip) and (dst port http)
  ```

- The filter produced from expression (1) *overapproximates* what the user intended since HTTP traffic is usually transported by TCP. Capturing UDP or SCTP traffic is unnecessary. The user might more accurately express their filter as follows:

  ```
  tcp dst port http                             (2)
  ```

  This expression results in a shorter filter program executed in the kernel, which – depending on the workload – might improve performance since the filter program is executed on every single packet.

Such implicit subformulas help make pcap expressions more succinct, which is part of the language's appeal. Over time practitioners get accustomed to the contribution of these implicit subformulas, which is mostly intuitive and easily adjusted.

[1] https://www.nik.network/caper

Using the approach described in this paper, the original expression (1) results in an equivalent fully-expanded form:[2]

```
(ether proto \ip && ip proto \sctp && sctp dst port 80) ||
(ether proto \ip6 && ip6 proto \sctp && sctp dst port 80) ||
(ether proto \ip && ip proto \udp && udp dst port 80) ||
(ether proto \ip6 && ip6 proto \udp && udp dst port 80) ||
(ether proto \ip && ip proto \tcp && tcp dst port 80) ||
(ether proto \ip6 && ip6 proto \tcp && tcp dst port 80)
```

One of the objectives of this paper is to implement a source-to-source transformation of pcap expressions to make them maximally explicit in detail. This is achieved through an *expansion semantics* which recursively expands the meaning of an expression. This paper also describes quirks of the pcap expression language that influence its semantics.

An interesting semantic feature of pcap expressions is that tautologies are not necessarily equivalent since they might imply a different packet structure. For example `tcp[0] = tcp[0]` matches TCP packets whose first byte is equal to itself, but its meaning can differ from that of other tautologies, such as `ip[0] = ip[0]`. The first expression matches all TCP packets, while the second matches all IPv4 packets. Neither expression is equivalent to "`ip or not ip`" or "`tcp or not tcp`", which describe the weakest pcap expression. In pcap expressions, the simplification of expressions and pruning of tautologies must be done with extra care.

To better model the behaviour of real implementations the semantics presented in this paper include counter-intuitive features which violate the "rule of least surprise" [22].[3] One such example involves the "`vlan`" term which is used to describe filters over IEEE 802.1Q (VLAN) virtual networks [14]. VLANs are widely used to organise large Ethernet networks, such as corporate and university networks. A frame in a VLAN has an extra header indicating the identity of the VLAN to which it belongs, since several virtual networks may be overlaid on the same physical Ethernet. Moreover, VLANs may be nested – in which case a frame would have several tags, one for each level of the nesting. The "`vlan`" term in pcap is problematic because once it is encountered then the rest of the expression is interpreted in the context of the VLAN tag. This deviates from the expected behaviour of disjunctions. For example, it is *not* the case that "`vlan or ip`" iff "`ip or vlan`". In the first expression the test for "`ip`" is done at an offset in the packet that accounts for a VLAN tag – even if "`vlan`" was *not* matched in the packet. In other words, "`ip or vlan`" is subsumed by both "`ip`" and "`vlan`" as expected; but "`vlan or ip`" is only subsumed by "`vlan`" since non-VLAN IPv4 packets cannot be matched by the resulting filter. The tool described in this paper emits warnings when such expressions are encountered.

Method outline. Pcap expressions describe predicates over packets, but deriving their semantics involves intermediate, discrete steps to map high-level pcap expressions into byte-level predicates over packets. The semantics presented in this paper is organised into these steps, which we implement in Caper: 1) expansion of pcap expressions to make explicit all implicit information; 2) resolution of various names (such as protocols, ports, and fields); 3) picking out side-effecting predicates and emitting a warning to the user; 4) mapping to a bit-vector formula that models a predicate over packets.

The contributions of this paper consist of:

(1) The first semantics of pcap expressions. The semantics uses a novel staging into two phases: first expanding pcap expressions to disambiguate their meaning, then mapping fully-expanded expressions into a Satisfiability Modulo Theories [21] (SMT) formalisation as predicates over packets.
(2) Caper: a practical tool that implements these semantics, warns users of potentially unwanted behaviour, and encodes subsumption- and equivalence-checking of pcap expressions in SMTLIBv2 format [17].

2 EXAMPLE

This section gives an example of using pcap expressions to describe a non-trivial filter for network traffic. Despite its simplicity the pcap expression language holds up to real-world use. This example by P. J. Malloy [18] describes a *signature* for suspicious traffic related to the Heartbleed [8] OpenSSL vulnerability, using which a server's memory could be read remotely without authorisation.

```
tcp src port 443 and
(tcp[((tcp[12] & 0xF0) >> 4) * 4] = 0x18) and
(tcp[((tcp[12] & 0xF0) >> 4) * 4 + 1] = 0x03) and          (3)
(tcp[((tcp[12] & 0xF0) >> 4) * 4 + 2] < 0x04) and
((ip[2:2] - 4 * (ip[0] & 0x0F)) -
  4 * ((tcp[12] & 0xF0) >> 4) > 69)
```

This expression captures anomalously large replies to SSL's Heartbeat feature. Such replies might indicate an exploitation of the Heartbleed vulnerability, but might also be a false-positive if the size of the reply was justified. One can add other filters and analysis to make the detection more accurate.

The expression conjoins together various sub-predicates. The meaning of the first line should be clear from the description in the previous section: it matches TCP-carried packets originating from port 443 of an unspecified host. Port 443 is that which usually listens for SSL connections.

The rest of the expression involves reading specific bytes of the packet, computing other byte offsets using familiar operators such as "&", ">>", and "*", and nesting expressions to check the values of bytes at those offsets. For instance the second line uses the result of a calculation over the 13th byte in a TCP segment (`tcp[12]`) to check a different part of the TCP payload.[4]

In the remainder of this section we interpret the second line of the expression to examine a non-trivial example of pcap usage, before elaborating the syntax and semantics of the language in the next section. Later in the paper we use the example from this section to evaluate our equivalence-checking tool for pcap expressions. There may be different ways to write an expression; for instance the first

[2]The brackets are necessary, otherwise `libpcap` interprets it as an expression that only accepts packets to TCP port 80 on IPv6. This is an artefact of `libpcap`'s parser (see footnote 6) whose documentation states that "Alternation and concatenation have equal precedence and associate left to right". In other words, both connectives have the same precedence, and are left-associative. This deviates from the conventional precedence of && over `||`.

[3]This rule relates to user interfaces, of which domain-specific languages are arguably an instance.

[4]Byte offsets in pcap expressions are zero-indexed.

line of the above example could be written as:

```
ip[9] = 0x6 && (ip[6:2] & 0x1fff) = 0 &&
ip[4 * (ip[0] & 0xf) : 2] = 0x1bb
```
(4)

Let S_1 be the innermost compound expression of the second line of expression (3): "`tcp[12] & 0xF0`", which masks the first 4 bits of the 13th byte of the TCP header. This corresponds to the "Data Offset" part of the TCP header [16], which indicates where the TCP payload begins – the TCP header is variable-sized, and one must calculate the offset of the payload as done in this expression. Let S_2 be "`(`S_1` >> 4) * 4`" which provides the byte offset within the packet where the TCP payload begins. Finally "`tcp[`S_2`] = 0x18`" checks that the first byte in the TCP payload consists of the value 24. At this location in the TCP payload we find the 8-bit type indicator of the SSL record [11], and the value 24 (or `0x18`) indicates that the TCP segment is carrying an SSL Heartbeat reply [23].

The rest of the expression checks the version of SSL being used and whether the size of the TCP segment exceeds 69 bytes. Thus this expression matches packets that might be involved in Heartbleed-enabled exfiltration.

3 PCAP EXPRESSIONS

This section presents a formalisation of the pcap language, based on information obtained from the `pcap-filter(7)` man page, manual experimentation using `tcpdump`, and reading of the target machine language to which pcap expressions are translated to disambiguate the contents of the documentation where needed.

We begin with some intuition about pcap expressions, before defining formal syntax and semantics. Pcap expressions describe predicates over packets, or "packet filters". They consist of Boolean connectives and two types of atoms.

One type of atoms is the *matcher*. It specifies a constraint on the structure of the packet, including fields. Expression (1) consists entirely of a matcher atom. As we saw earlier, one of the peculiarities of pcap expressions is that matcher "atoms" might not be fully-specified, and thus need to be expanded to give the expression its full meaning. We describe such an expansion semantics in §3.2.1.

The other type of atom consists of relations over the contents of the packet. Expression (4) consists entirely of such atoms conjoined together.

The Boolean connectives ("`and`", "`or`" and "`not`") behave as those of classical logic.[5] Negating an expression results in the inversion of what the derived packet filter accepts and rejects. For instance, this filter rejects packets that the expression (2) accepts, and accepts everything that (2) rejects:

```
not (tcp dst port http)
```

And the following expression filters for packets headed both *to* and *from* a Web server:

```
(tcp dst port http) or (tcp src port http)
```

In the same spirit of brevity by which pcap expressions may contain implicit clauses, a more compact syntax is provided for the above:

```
tcp dst or src port http
```

[5]The concrete syntax of pcap expressions also allows these to be written as "`&&`", "`||`" and "`!`".

(Filter $\mathcal{F}$)
$\mathcal{F} \;::=\; \mathcal{A} \mid \mathcal{R} \mid \mathcal{F}_1 \text{ and } \mathcal{F}_2 \mid \mathcal{F}_1 \text{ or } \mathcal{F}_2 \mid \text{not } \mathcal{F}'$
(Relation $\mathcal{R}$)
$\mathcal{R} \;::=\; \mathcal{E}_1 > \mathcal{E}_2 \mid \mathcal{E}_1 < \mathcal{E}_2 \mid \mathcal{E}_1 = \mathcal{E}_2$
(Expression $\mathcal{E}$)
$\mathcal{E} \;::=\; \mathcal{L} \mid \mathcal{P}[\mathcal{E}' : \mathcal{W}] \mid \mathcal{E}_1 \# \mathcal{E}_2 \mid \text{len}$
where:
(Width in bytes $\mathcal{W}$)
$\mathcal{W} = \{1, 2, 4\}$
(Operator #)
$\# = \{+, -, \times, \div, \&, |, >>, <<\}$
(Literal $\mathcal{L}$)
$\mathcal{L} \;::=\; \mathcal{L}_{\text{string}} \mid \mathcal{L}_{\text{dec}} \mid \mathcal{L}_{\text{hex}}$
(Protocol $\mathcal{P}$)
$\mathcal{P} \;::=\; \text{ether} \mid \text{vlan} \mid \text{mpls} \mid \text{ip} \mid \text{ip6} \mid \text{arp} \mid \text{rarp} \mid \text{icmp} \mid \text{tcp} \mid \text{udp}$
(Type of entity $\mathcal{T}$)
$\mathcal{T} \;::=\; \text{host} \mid \text{net} \mid \text{port} \mid \text{portrange} \mid \text{proto}$
(Direction $\mathcal{D}$)
$\mathcal{D} \;::=\; \text{src} \mid \text{dst} \mid \text{src\_or\_dst} \mid \text{src\_and\_dst}$
(Matcher $\mathcal{M}$)
$\mathcal{M} \;::=\; \mathcal{P}^? \, \mathcal{D}^? \, \mathcal{T}^?$
(Atom $\mathcal{A}$)
$\mathcal{A} \;::=\; \mathcal{M} \, \mathcal{V}^?$
(Value $\mathcal{V}$)
$\mathcal{V} \;::=\; \mathcal{V}_{\text{string}} \mid \mathcal{V}_{\text{dec}} \mid \mathcal{V}_{\text{address}} \mid \mathcal{V}_{\text{network}}$

Figure 1: Syntax of the pcap language.

3.1 Syntax

This section formalises the syntax of pcap expressions, relative to which the semantics are given in the next section.

The abstract syntax is given in Fig. 1, which covers a substantial part of the language. It is simplified slightly from the syntax described in the `pcap-filter(7)` man page, to exclude some features for clarity (e.g., syntactic sugaring such as $\mathcal{E}_1 \geq \mathcal{E}_2$) and conciseness. Caper supports the different notations for expressing the addresses of networks – i.e., as network prefixes, or using netmasks – but these are elided in Fig. 1.

Fig. 1 shows the language's various syntactic classes. The notation $\cdot^?$ indicates an optional component to a term. For example, an atom $\mathcal{A}$ is formed from a matcher $\mathcal{M}$ and an optional value $\mathcal{V}^?$. If the value is present, then it strengthens the matcher (to match that specific value), otherwise the matcher matches any suitable value. For example, matcher "ip $\frown\frown$" matches any IPv4 address. The notation $\frown$ indicates the absence of an optional component: in this example the $\mathcal{D}$ and $\mathcal{T}$ components. For example, the matcher "ip src_or_dst host" has the exact meaning as the previous example but includes all components.

We turn to the domain-specific elements of the language next. The top-level syntactic categories (filters, relations, expressions) are intuitive and mostly general. Within expressions $\mathcal{E}$, len denotes the size of the whole packet. $\mathcal{P}[\mathcal{E}' : \mathcal{W}]$ is used to identify a segment of $\mathcal{W}$ bytes in the packet, beginning at an offset of $\mathcal{E}'$ bytes from the header of protocol $\mathcal{P}$. The interpretation of such an expression is conditional upon whether such a header is in its expected location in the packet. Thus tcp[0 : 4] matches the first four bytes of a TCP segment; but this is only meaningful if the packet contains a TCP segment.

The categories of literals $\mathcal{L}$ and values $\mathcal{V}$ are abstracted in Fig. 1; their opaque description is sufficient for our purposes. The two categories might appear similar – for example both of them contain strings of characters, and decimal numerals – but serve different purposes in the syntax: literals may only appear in expressions, and values may only appear in matched atoms, and this affects their interpretation. For example, strings in expressions are interpreted as the names of fields in a protocol header, while strings in a value are interpreted to be the name associated with a host, network, or port.

Finally, the components of matchers may include a protocol $\mathcal{P}$, direction $\mathcal{D}$, and type $\mathcal{T}$. "Type" here refers to which values are applicable to a matcher: for example, matchers having type host may only be applied to host names (strings) or network addresses; and port may only be applied to well-known port names (also strings) or port numbers (integers between 0 and 65535).

The pcap language does not have a type system, but there is a notion of well-formedness of expressions: for example the matcher "ip $\frown\frown$" may only be applied to an IPv4 address value. It would not be acceptable to apply it to an IPv6 address: the address field of IPv4 packets is 32-bits wide, while that of an IPv6 packet is 128-bits wide. The well-formedness relation is not included here since it is straightforward, but it is included as a check on parsed pcap expressions in Caper.

3.2 Semantics

The semantics of the pcap language is given in two stages: an *expansion* semantics (§3.2.1) that rewrites expressions to disambiguate them, and a bit-vector semantics (§3.2.2) that maps the expanded expression into bit-vector logic to characterise a predicate over packets. Then in §4 we describe how the bit-vector semantics is used for equivalence-checking between pcap expressions, by using off-the-shelf SMT solvers.

3.2.1 Expansion Semantics. This section describes a function $\leadsto \subseteq \mathcal{F} \times \mathcal{F}$ that rewrites pcap filter expressions into "long form", making all implicit details explicit.

For an example of how this works, consider the pcap filter expression "`src foo`" where "foo" is a host name. Using the notation from §3.1, this corresponds to an *atom* ($\mathcal{A}$) "$\frown$ src $\frown$ foo".
This expression is rewritten in the following steps:

$$\begin{array}{rl} & \frown \text{src} \frown \texttt{foo} \\ \text{(i)} & \frown \text{src host } \texttt{foo} \\ \text{(ii)} & \left(\begin{array}{l} \text{rarp src host } \texttt{foo} \text{ or} \\ \text{arp src host } \texttt{foo} \text{ or} \\ \text{ip src host } \texttt{foo} \end{array}\right) \\ \text{(iii)} & \left(\begin{array}{l} \text{ether} \frown \text{proto rarp and rarp src host } \texttt{foo} \text{ or} \\ \text{arp src host } \texttt{foo} \text{ or} \\ \text{ip src host } \texttt{foo} \end{array}\right) \end{array}$$

In step (i) the implicit type ($\mathcal{T}$ in Fig .1) host is added, in line with the convention expressed in the `pcap-filter(7)` man page. In step (ii) we fill in the details about which protocols can apply to "`host src foo`", and in step (iii) we specify which protocols can carry the protocols specified in step (ii).

The information about the rewriting in these steps was extracted from the pcap-filter documentation. It was also checked with the implementation,[6] particularly when the documentation was unclear: then the meaning of pcap expressions was reconstructed from the machine code to which they were compiled. This led to the realisation that the meaning of `tcp[tcpflags]`, which extracts the TCP flags from a TCP header, deviates from expectation since it excludes two of the flags. Where such deviations were discovered, Caper was changed to match the behaviour of `libpcap`.

The function "$\leadsto$" expands the expression fully. Starting with "`src foo`" we end up with:[7]

```
(ether proto \rarp && rarp src host foo) ||
(ether proto \arp && arp src host foo) ||        (5)
(ether proto \ip && ip src host foo)
```

Once an expression is fully-expanded, then "$\leadsto$" behaves as the identity function.

Note that expressions are not necessarily fully-specified when "$\leadsto$" terminates: for example, "ether $\frown$ proto rarp" in step (iii) is in its final form. This is because fully-specifying atoms might not be sensible: in this example, the direction ($\mathcal{D}$ in Fig .1) does not apply to the value "rarp" since the latter is a protocol which, unlike a host or a port, cannot in itself be the source or destination of packets.

The set of protocols might evolve over time: for example at present there is a push to transition to using IPv6 more on the public Internet, instead of IPv4. It appears that such changes will have little effect on "$\leadsto$" beyond the updating of the mapping of protocol dependencies. This mapping will be described further as we look at the expansion function in more detail.

In Caper, "$\leadsto$" is defined as the following composition of functions, each carrying out a transformation of the overall expression until a fixpoint is reached.

$$\leadsto_{\text{Simp}} \circ \leadsto_{\text{Subs}} \circ \leadsto_{\text{Dedup}} \circ \leadsto_{\text{ImplicitX}} \circ \leadsto_{\text{AtomX}} \circ \leadsto_{\text{DisjX}}$$

The functions DisjX, AtomX and ImplicitX carry out expansion, while Dedup, Subs and Simp carry out simplification of pcap expressions. The first three functions are described in more detail below; the last three functions are mostly straightforward, and described more briefly.

Disjunction expansion: $\leadsto_{\text{DisjX}}$. For convenience, in pcap one can write "`host Alpha or Bravo or Charlie`", where Bravo and Charlie are names, to abbreviate "`host Alpha or host Bravo or host Charlie`". Disjunction expansion simply propagates an atom's matcher components to adjacent disjunct atoms that have undefined matchers.[8] For conciseness we only show the key transformation made by this function; Caper defines it for all syntax phrases of $\mathcal{F}$ (Fig. 1):

$$\widehat{p}\ \widehat{d}\ \widehat{t}\ v_1 \text{ or } \frown\frown\frown\ v_2 \quad \leadsto_{\text{DisjX}} \quad \widehat{p}\ \widehat{d}\ \widehat{t}\ v_1 \text{ or } \widehat{p}\ \widehat{d}\ \widehat{t}\ v_2$$

[6]`libpcap version 1.8.1 -- Apple version 67.60.2`
[7]Note that in (5) we backslash-escape \arp, but do not escape it in step (iii): this is because the former uses actual pcap syntax, whereas the latter uses the simpler grammar from Fig .1
[8]This expansion also seems to apply to *conjunctions* in `libpcap`, but this tends to produce impossibly strong predicates, e.g., "`src port 50 and 51`" is interpreted to mean "`src port 50 and src port 51`".

$$
\begin{array}{rcll}
p \frown\frown\frown & \rightsquigarrow_{\text{AtomX}} & \frown\frown \text{ proto } p & \blacktriangleleft \\
\wideparen{p}\ \wideparen{d} \frown v & \rightsquigarrow_{\text{AtomX}} & \wideparen{p}\ \wideparen{d} \text{ host } v & \text{(a)} \\
 & & \text{if } p \notin \{\text{vlan}, \text{mpls}\} & \\
 & & \quad \text{and } v \text{ addressable to } p & \\
 & & \quad \text{or } \wideparen{p} = \frown & \\
\wideparen{p} \frown t\ v & \rightsquigarrow_{\text{AtomX}} & \wideparen{p} \text{ src\_or\_dst } t\ v & \text{(b)} \\
 & & \text{if } p \notin \{\text{vlan}, \text{mpls}\} & \\
\wideparen{p} \text{ src\_or\_dst } \wideparen{t}\ v & \rightsquigarrow_{\text{AtomX}} & \wideparen{p} \text{ src } \wideparen{t}\ v \text{ or} & \blacktriangleleft \\
 & & \wideparen{p} \text{ dst } \wideparen{t}\ v & \\
\wideparen{p} \text{ src\_and\_dst } \wideparen{t}\ v & \rightsquigarrow_{\text{AtomX}} & \wideparen{p} \text{ src } \wideparen{t}\ v \text{ and} & \blacktriangleleft \\
 & & \wideparen{p} \text{ dst } \wideparen{t}\ v & \\
\frown \wideparen{d} \text{ host } v & \rightsquigarrow_{\text{AtomX}} & \text{ether } \wideparen{d} \text{ host } v & \\
 & & \text{if } v \text{ is an Ethernet address} & \\
\frown \wideparen{d} \text{ host } v & \rightsquigarrow_{\text{AtomX}} & \text{ip } \wideparen{d} \text{ host } v \text{ or} & \text{(c)} \\
 & & \text{arp } \wideparen{d} \text{ host } v \text{ or} & \\
 & & \text{rarp } \wideparen{d} \text{ host } v & \\
 & & \text{if } v \text{ is an IPv4 address} & \\
 & & \quad \text{or a host name} & \\
\frown \wideparen{d} \text{ port } v & \rightsquigarrow_{\text{AtomX}} & \text{tcp } \wideparen{d} \text{ port } v \text{ or} & \text{(d)} \\
 & & \text{udp } \wideparen{d} \text{ port } v \text{ or} & \\
 & & \text{sctp } \wideparen{d} \text{ port } v & \\
 & & \text{if } v \text{ is a port name} & \\
 & & \quad \text{or number} &
\end{array}
$$

Figure 2: Some of the transformations effected during atom expansion. The left-hand expression of $\rightsquigarrow_{\text{AtomX}}$ is rewritten into the right-hand expression, possibly subject to a condition (if ...) being satisfied.

The notation $\wideparen{p}$ indicates a protocol (drawn from $\mathcal{P}$, here ranged over by p) or an empty component $\frown$.

Atom expansion: $\rightsquigarrow_{\text{AtomX}}$. The next step involves filling the empty components of atoms as much as possible, as we saw happen in steps (i) and (ii) in the example at the start of §3.2.1. Fig. 2 shows some of the rules that are implemented in $\rightsquigarrow_{\text{AtomX}}$.

Clause (a) in Fig. 2 contributed to step (i) of the earlier example, and clause (c) contributed to step (ii).

The transformations described in both clauses are conditional: in both cases, if p is given then it must not be vlan or mpls. Clause (a) has the additional condition that value v must be interpreted to be an address to protocol p, or $\wideparen{p}$ is empty in which case the transformation can go through – by default v is then expected by `libpcap` to be a host on an IPv4 network.

The conditions of clause (b) are simpler since the matchers for vlan and mpls are less expressive than those for other protocols. Using pcap expressions we can query packets for whether they belong to a VLAN or MPLS network, and *which* such network they belong to, by examinig the VLAN or MPLS tag. Thus VLAN matchers can either be of form "vlan $\frown\frown\frown$" or "vlan $\frown\frown$ v".

Clauses marked with ◀ serve to transform expressions into a canonical form, to enable other transformations.

Clauses (c) and (d) are derived from the following sentence in the `libpcap` documentation:

> "If there is no proto qualifier, all protocols consistent with the type are assumed. E.g., 'src foo' means '(ip or arp or rarp) src foo' (except the latter is not legal syntax), 'net bar' means '(ip or arp or rarp) net bar' and 'port 53' means '(tcp or udp) port 53'."
> – from `pcap-filter(7)` man page.

This is related to the protocol dependency mapping mentioned earlier. Protocols and their dependencies can vary over time, and Caper can be adjusted fairly easily to keep up with changes since the set of protocols and their dependencies are encoded prominently in $\rightsquigarrow_{\text{AtomX}}$.

Indeed some protocol evolution appears to have already taken place since the `libpcap` documentation was written. The documentation quoted above does not fully agree with the `libpcap` implementation, to which 'port 53' means '(tcp or udp or sctp) port 53'. As mentioned before, the source-to-source pcap expression transformation in Caper seek to follow the implicit transformation done by `libpcap` rather than rely entirely on `libpcap`'s documentation.

Implicit subformula expansion: $\rightsquigarrow_{\text{ImplicitX}}$. The last expansion makes explicit the cross-layer protocol dependencies of pcap expressions. This expansion contributed to step (iii) of the earlier example, as well as the remaining steps to reach the fully expanded expression (5).

The function $\rightsquigarrow_{\text{ImplicitX}}$ starts by gathering the protocols mentioned in atom and relation terms ($\mathcal{A}$ and $\mathcal{R}$ from Fig. 1), and iteratively conjoins all the possible protocol dependencies, proceeding down the layers of the network protocol stack. In so doing, it is possible that redundant information is added to pcap filters; this is simplified away during later transformations.

The function $\rightsquigarrow_{\text{ImplicitX}}$ is supported by two functions. The first is a function that extracts the protocols mentioned in an atom or relation. For example, for atom (1) this function returns $\emptyset$, for atom (2) it returns {tcp}, for each relation in expression (4) it returns {ip}, and for the more contrived expression `tcp[5] = ip[0]` it returns {ip, tcp}.

The second function maps protocols to lower-layer protocols that might encapsulate them. For example in step (iii) above we see the result of "rarp" being mapped to "ether".

A pcap expression may override the default dependency mapping, but the result might not be sensible. For example, `libpcap` accepts the expression:

```
ip proto \icmp6
```
(6)

This filters for very unusual traffic: IPv4 packets carrying ICMPv6 [7] datagrams. In line with `libpcap`'s behaviour, Caper was altered to accept such expressions but it emits a warning for the user's benefit, in case a different expression was intended.[9]

Deduplication $\rightsquigarrow_{\text{Dedup}}$*, Subsumption* $\rightsquigarrow_{\text{Subs}}$*, and Simplification* $\rightsquigarrow_{\text{Simp}}$. These transformations are mostly straightforward, but sensitive to the packet-level interpretation of matchers.

[9]Despite producing a filter for expression (6), `libpcap` does not produce a filter for "`ip && icmp6`", which suggests that (6) might be a corner case. Caper replicates this behaviour.

Simplification involves small changes such as eliminating double-negations, and finding contradictions which might make subformulas redundant. Contradictions are not only purely logical – such as "`ip && not ip`" – but are also domain-sensitive: e.g., "`ip && ip6`" and "`tcp && udp`" are both unsatisfiable. Caper detects such contradictions by first mapping each protocol to an integer associated with the protocol's network layer – for example $\{\mathsf{ip} \mapsto 3, \mathsf{ip6} \mapsto 3, \mathsf{udp} \mapsto 4, \mathsf{tcp} \mapsto 4\}$. Then conjunctions are checked for multiple occurrences of the same integer. This transformation is applied after the expression has been simplified, to avoid the mistake of treating "`tcp and tcp`" as a contradiction. As before, `vlan` and `mpls` matchers are treated specially: they cannot contribute to a contradiction since at every occurrence they specify a fresh encapsulation of the packet constrained by the expression that follows.

Deduplication removes repeated subformulas, but because of domain-sensitivity it must heed the subformula's contents. For example, "`tcp and tcp`" is simplified to "`tcp`" but "`vlan and vlan`" is not changed since the second occurrence of `vlan` is not redundant: it refers to VLAN-encapsulated packet that is itself VLAN-encapsulated.[10]

3.2.2 Bit-Vector Semantics. Starting with a fully-expanded pcap filter from the previous stage, we give it semantics by mapping the filter into a formula in bit-vector logic [2]. In particular, we target the theory of bit-vectors and arrays which was standardised for implementation in various SMT solvers [17]. We can then give the formula to a solver to check pcap filters for equivalence: two filters are equivalent when their SMT-encoded formulas are logically equivalent.

Our formalisation is centered on a model M that represents a packet, which is modelled as a function mapping an index to a byte: $M \in \left(\mathbb{N} \to \mathbb{B}^8\right)$. To improve the model's fidelity we bound its size from above, thus capturing the observation that packets are finite. We pick a number $\mathsf{mtu} \in \mathbb{N}$ to be the *maximum transmission unit* [15] – the size of the largest possible packet that may cross the intended network interface. We also bound the size of M from below by picking a value $\mathsf{min} \in \mathbb{N}$ to capture the observation that packets in practice have a minimum size. In Caper we set $\mathsf{min} = 64$ and $\mathsf{mtu} = 1514$, consistent with the widely-used Ethernet [12] standard.

As a result of these bounds, M ranges over a finite set of finite functions mapping indices to bytes, modelling the full set of possible packets. The SMT solver surveys this set as it attempts to find an M that refutes semantical properties of pcap expressions – for instance, that two formulas are equivalent.

The notation $M \models \phi$ shall be used to mean that the model M satisfies the SMT formula ϕ. This is a judgement decided by a SMT-solver, which we use to deduce properties of pcap filter expressions.

We now turn to the mapping from pcap filters (§3.1) to bit-vector logic. Let $\ulcorner\cdot\urcorner$ denote this mapping, overloading the symbol to map from the syntactic categories from Fig. 1 to corresponding terms in bit-vector logic.

[10]One could argue that, for the sake of consistency, "`tcp and tcp`" should also denote a TCP-encapsulated TCP segment, but this is not the interpretation used in `libpcap`. For more flexible encapsulation syntax see Kneecap [25].

The mapping is mostly straightforward: logical connectives and arithmetical operators are mapped to their analogues in the target logic. Let $f \in \mathcal{F}$ be a pcap filter expression, and $\ulcorner f \urcorner$ be its SMT-encoded counterpart. This is part of the definition of $\ulcorner\cdot\urcorner$ for filters, which proceeds in the obvious way:

$$\begin{aligned}
\ulcorner f_1 \text{ and } f_2 \urcorner &=_{\text{def}} \ulcorner f_1 \urcorner \wedge \ulcorner f_2 \urcorner \\
\ulcorner f_1 \text{ or } f_2 \urcorner &=_{\text{def}} \ulcorner f_1 \urcorner \vee \ulcorner f_2 \urcorner \\
\ulcorner \text{not } f \urcorner &=_{\text{def}} \neg \ulcorner f \urcorner
\end{aligned}$$

The most interesting mappings are those for atoms ($\mathcal{A}$) and expressions ($\mathcal{E}$), and we will focus on these next.

In our semantics, pcap expressions are ultimately turned into constraints on M. This cannot be seen at the level of filters, shown above, but can be seen explicitly in the selection of atom semantics shown in Fig. 3. In that mapping, the parameter i represents the increase in the offset from the start of M as a side-effect of the expression being translated; this captures the behaviour of `vlan` and `mpls` that was described in §1.

Expressions ($\mathcal{E}$) are translated into bit-vector expressions in a mostly straightforward manner. For expressions we extend the translation to thread a parameter S that accummulates conjunctive constraints on the dependencies of an expression – for example, an expression mentioning the protocol `ip` requires that protocol to be involved in the packet being filtered. The two clauses below are a sample showing a simple case – addition – which involves straightforward recursion through the structure of the expression, and the most complex type of expression to translate: packet-array projection. This is complex since it combines a protocol-specific offset (ipOffset in the case of `ip`), adds a constraint to S, and nests a user-provided expression e to project a value from M.

$$\begin{aligned}
\ulcorner e_1 + e_2 \urcorner \triangleleft i, S &=_{\text{def}} \ulcorner e_1 \urcorner \triangleleft i, S + \ulcorner e_2 \urcorner \triangleleft i, S \\
\ulcorner \mathsf{ip}[e] \urcorner \triangleleft i, S &=_{\text{def}} M\left(i + \boxed{\text{ipOffset}} + \ulcorner e \urcorner\right) \triangleleft i, (S \cup \mathsf{ip})
\end{aligned}$$

Bit-vector dimensioning. For clarity of presentation the above description did not include a specification of the widths of bit-vector expressions. Well-formed bit-vector expressions in the target theory must have a definite width. Caper's mapping into bit-vector formulas includes this width when it is explicit in the source expression. For example, $\mathcal{P}[\mathcal{E} : 2]$ is 16-bits wide (cf $\mathcal{W}$ in Fig. 1). When the width is not immediately known then Caper allows the width to be undefined – for instance, the width of "17" in the expression "$\mathcal{P}[\mathcal{E} : 2] + 17$". During a later pass it then propagates known widths to calculate the undefined widths; thus in the previous example it encodes "17" as 0x0011 rather than 0x11. When bit-vector widths are already known, they may be *widened* further as needed to accommodate calculations over wider subexpressions. For example, in the expression "$\mathcal{P}[\mathcal{E} : 2] + \mathcal{P}[\mathcal{E} : 4]$" we widen the left subexpression to 32-bits from 16-bits in order to carry out the addition.

Name resolution. Recall that the Web server is *unspecified* in filter (2), but should we wish to strengthen the filter for packets to or from specific hosts then this can easily be accommodated. For example:

```
tcp dst port http and ip host www.cis.upenn.edu
```

$$\begin{aligned}
\ulcorner\text{ether} \frown \text{proto arp}\urcorner \triangleleft\ i &=_{\text{def}} \left(M\left(i + \text{ethertype}\right) \cdot M\left(i + \text{ethertype} + 1\right)\right) = \text{0x0806} \triangleleft\ i \\
\ulcorner\text{vlan} \frown\frown \widehat{v}\urcorner \triangleleft\ i &=_{\text{def}} \left(\begin{array}{l} M(i + \text{ethertype}) \cdot M(i + \text{ethertype} + 1) = \text{0x8100} \vee \\ M(i + \text{ethertype}) \cdot M(i + \text{ethertype} + 1) = \text{0x88a8} \vee \\ M(i + \text{ethertype}) \cdot M(i + \text{ethertype} + 1) = \text{0x9100} \end{array}\right) \triangleleft\ i + 4 \\
\ulcorner\text{ip} \frown \text{proto tcp}\urcorner \triangleleft\ i &=_{\text{def}} M\left(i + \text{protocol}\right) = \text{0x06} \triangleleft\ i
\end{aligned}$$

Figure 3: Bit-vector semantics of some atom clauses ($\mathcal{A}$). Each clause specifies a constraint on a segment of the packet, based on the description of standardised packet formats. The "·" bit-vector operator concatenates an m-bit vector and an n-bit vector to yield an $(m+n)$-bit vector. The numeric constants ethertype and protocol are offsets of fields in Ethernet and IPv4 packet formats respectively. Most clauses preserve the index offset i; `vlan` is an example of an expression that implicitly increments the value of i.

During the translation process, `libpcap` converts the human-readable name to a network address. We can also provide a network address instead of a name:

```
tcp dst port http and ip host 158.130.69.163
```

An address does not require further resolution: its numeric value is matched by the filter directly.

Different kinds of names can occur in pcap expressions, and we look at several examples next. All names are ultimately resolved into unsigned integers, but the kind of resolution used depends on what kind of name is being resolved.

Host names such as `www.cis.upenn.edu` are resolved by calling `getaddrinfo()` on POSIX systems. A host name might resolve to several addresses, and on several networks. In such cases a formula is turned into a disjunction with one disjunct for each address. Similarly port names can be resolved by calling `getservbyname()` on POSIX systems, which in turn queries the system's mapping from port names to numbers.

Other kinds of name occurrences usually do not have a system-queryable resolution API, but their resolution is usually standardised. For example, the expression "`icmp[icmptype] != icmp-echo`" contains three types of names: a protocol (`icmp`), field (`icmptype`) and constant (`icmp-echo`). The field and constant are mapped using a simple table that was populated using information extracted from the documentation of pcap filters. The protocol is resolved into an offset in the packet. This offset adds to the offsets of lower-layer protocols that are carrying the ICMP packet.

4 USING THE SEMANTICS

We can use the semantics to answer the following questions:

(1) *What does a given pcap expression mean?* We can answer this question by using an implementation of the expansion semantics described in §3.2.1 to write out the expression in full, to reveal hidden meaning in the original expression as we saw in example expression (1).

(2) *Are two given pcap expressions related by subsumption – does one expression capture the packets captured by the other expression?* This is answered by an implementation of the bit-vector semantics described in §3.2.2, and using a decision procedure for the target theory.

We say "f_2 subsumes f_1" if f_2 accepts all the packets that f_1 accepts. We encode the problem as follows for an SMT solver:

$$M \models \neg\,(\ulcorner f_1 \urcorner \longrightarrow \ulcorner f_2 \urcorner)$$

If the solver is unable to find a witnessing M then we conclude that the non-negated statement holds for all models.

(3) *Are two given pcap expressions equivalent – do they capture the same set of packets?* This too is answered using the bit-vector semantics. It involves proving that the two expressions subsume each other, which corresponds to the following statement:

$$\models \ulcorner f_1 \urcorner \longleftrightarrow \ulcorner f_2 \urcorner$$

5 EVALUATION

All the pcap expressions shown in this paper have been put through Caper, a new tool that implements a parser for the full pcap language and the semantics described in §3.2.1 to expand them into equivalent pcap expressions. Some pairs of expressions were also checked for equivalence, by using the semantics described in §3.2.2 to generate SMT problems for which we used Z3 [9] version 4.5.1 as the back-end SMT solver.

The remainder of this section describes equivalences that were checked using this approach. If two expressions are determined equivalent by the tool then this is a strong result: the SMT solver proves a theorem stating that, for all packets π, one expression accepts π iff the other expression accepts π. SMT solvers can emit a proof for such a theorem [4], specialised for bit-vector reasoning [13] on which we rely heavily in this work. Emitted proofs are usually not directly usable by humans, but can be imported into proof assistants [6] for checking or reuse.

We also used this approach to *disprove* equivalence, such as that between "`vlan || ip`" and "`ip || vlan`", but this outcome is arguably less useful in practice than the verification of equivalence since we cannot yet use counter-models to help the user understand *why* two expressions are not equivalent.

The simplest example verified is the equivalence between "`ip && tcp src port 443`" and a manually-rewritten expression (4). This is a simple but non-obvious rewrite that casts a matcher ($\mathcal{M}$) into a conjunction of relations ($\mathcal{R}$); moreover this rewrite does not directly correspond to the bit-vector formula mapping (§3.2.2) which casts formulas in terms of packets, not in terms of the `ip` layer.

A more complex example involved expanding the expression "`tcp`" and proving it to be equivalent to the following manually-rearranged expression using the distributivity laws for propositional logic:

```
(ether proto \ip || ether proto \ip6) &&
(ether proto \ip || ip6 proto \tcp) &&
(ip proto \tcp || ether proto \ip6) &&
(ip proto \tcp || ip6 proto \tcp)
```

The most complex example verified is the equivalence between expression (3), which encodes a signature for possible Heartbleed-enabled exfiltration, and the manually-crafted equivalent:

```
ether proto \ip && ip[9] = 0x6 && (ip[6:2] & 0x1fff) = 0 &&
ip[4*(ip[0] & 0xf):2] = 0x1bb && ip[4*(ip[0] & 0xf) +
  (((ip[4*(ip[0] & 0xf)+12]) &  0xf0)>>4)*4] = 0x18 &&
ip[4*(ip[0] & 0xf) + (((ip[4*(ip[0] & 0xf)+12]) &
  0xf0)>>4)*4+1] = 0x03 &&
ip[4*(ip[0] & 0xf) + (((ip[4*(ip[0] & 0xf)+12]) &
  0xf0)>>4)*4+2] = 0x04 &&
(ip[2:2]-4*(ip[0] & 0x0f))-4*((ip[4*(ip[0] &
  0xf)+12] & 0xf0)>>4) > 0x45
```

The above expression seems contrived when compared to the original but it tests various syntactic features of expressions to exercise the semantics and their implementation in Caper when checking a real-world pcap filter.

The last equation takes noticeably longer to verify than the first two. Across 5 runs, the average (arithmetical mean) wall-clock time, in seconds, of running Z3 to prove the equations was: 0.04s, 0.01s, and 3.99s for each equation respectively. This experiment was carried out on a 2016 MacBook Pro with 2.9 GHz Intel Core i5 and 16GB RAM. These timing results are not expected to be robust or deeply meaningful, but were produced to measure the relative complexity of proving the different equations using a state-of-the-art solver as the expressions being equated become more complex. Future work could survey a collection of real-world pcap expressions and assess whether, for pairs of equivalent expressions, the verification time is feasible in a real-world setting.

6 RELATED WORK

This paper intersects with two closely-related bodies of work: practical advice on using pcap, and pcap-related work based on formal methods.

Practical language clarifications. Begel et al. outline the development of BPF [3] and Snellman [24] gives more examples of the pcap language's rough edges. This paper seeks to create a faithful model of the pcap language to support existing users through tooling.

Pcap-related formal methods approaches. Jitk [26] seeks to develop correct in-kernel interpreters, that can be used for packet filtering among other uses. It targets the interpretation of the BPF machine language, to which pcap expressions are compiled, whereas this paper focuses on the semantics of the pcap expressions themselves. Another related work is Kneecap [25], which devises a pcap-like language and uses it for model-based packet generation. The Kneecap language has cleaner semantics than pcap expressions since it allows for arbitrary nesting of protocols and avoids some of the shortcomings of pcap expressions, but the latter have widespread usage. In this paper we sought to be consistent with the widely-used language.

7 CONCLUSION

Formalisation is a means to understanding a language, and this paper sought to model an existing and widely-used language, warts and all. But formalisation is not sufficiently useful by itself: it is more useful if it enables formalisation-based implementations of tools.

This paper sought to use formalisation to achieve practical benefits by developing a tool around the two-stage organisation of semantics: (i) an expansion semantics to aid the understanding of terse pcap expressions before they are turned into a filter, and (ii) bit-vector semantics to compare expressions for equivalence using off-the-shelf solvers.

One direction for future work involves validating our semantics against an actual instance of `libpcap` [1], inspired from the approach used by Bishop et al. for model-based testing of real-world implementations [5]. The semantics described in this paper were extracted from the language's documentation and implementation in `libpcap`, and were tested against Caper to discover inconsistencies between the `libpcap` and Caper; but more rigorous testing can help tighten the consistency of the semantics with the `libpcap`.

Our formalisation consists solely of specification – no theorems or proofs were given. Another direction for future work involves formalising a translation of pcap expression to the BPF machine language [19] and proving this to be behaviour-preserving.

ACKNOWLEDGMENTS

I am indebted to Richard Mortier, Bjoern Zeeb, Peter Sewell and Joachim Breitner for discussions on pcap and BPF, and the anonymous reviewers for their feedback. This work was partly supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts HR0011-17-C-0047 and HR0011-16-C-0056.

REFERENCES

[1] [n. d.]. tcpdump and libpcap. https://www.tcpdump.org/.
[2] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. 1998. A Decision Procedure for Bit-vector Arithmetic. In *Proceedings of the 35th Annual Design Automation Conference (DAC '98)*. ACM, New York, NY, USA, 522–527. https://doi.org/10.1145/277044.277186
[3] Andrew Begel, Steven McCanne, and Susan L. Graham. 1999. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. *SIGCOMM Comput. Commun. Rev.* 29, 4 (Aug. 1999), 123–134. https://doi.org/10.1145/316194.316214
[4] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. 2011. A Nelson-Oppen based Proof System using Theory Specific Proof Systems. In *First International Workshop on Proof eXchange for Theorem Proving-PxTP 2011*.
[5] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2006. Engineering with Logic: HOL Specification and Symbolic-evaluation Testing for TCP Implementations. *SIGPLAN Not.* 41, 1 (Jan. 2006), 55–66. https://doi.org/10.1145/1111320.1111043
[6] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. 2011. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.
[7] Alex Conta, Stephen Deering, and Mukesh Gupta. 2006. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443. https://doi.org/10.17487/rfc4443

[8] National Vulnerability Database. [n. d.]. CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.
[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
[10] Stephen E. Deering and Robert M. Hinden. 2017. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200. https://doi.org/10.17487/rfc8200
[11] Tim Dierks and Eric Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. https://doi.org/10.17487/rfc5246
[12] Ethernet Working Group. [n. d.]. IEEE 802.3. http://www.ieee802.org/3/.
[13] Liana Hadarean, Clark Barrett, Andrew Reynolds, Cesare Tinelli, and Morgan Deters. 2015. Fine Grained SMT Proofs for the Theory ofÂăFixed-Width Bit-Vectors. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 340–355.
[14] IEEE 2014. *802.1Q-2014: Bridges and Bridged Networks*. IEEE, http://www.ieee802.org/1/pages/802.1Q.html.
[15] University of Southern California Information Sciences Institute. 1981. Internet Protocol. RFC 791. https://doi.org/10.17487/rfc791
[16] University of Southern California Information Sciences Institute. 1981. Transmission Control Protocol. RFC 793. https://doi.org/10.17487/rfc793
[17] The SMT-LIB Initiative. [n. d.]. SMT-LIB: The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu/.
[18] P.J. Malloy. [n. d.]. How to Detect a Prior Heartbleed Exploit. https://bit.ly/2T5fMu7.
[19] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1267303.1267305
[20] J. Mogul, R. Rashid, and M. Accetta. 1987. The Packer Filter: An Efficient Mechanism for User-level Network Code. *SIGOPS Oper. Syst. Rev.* 21, 5 (Nov. 1987), 39–51. https://doi.org/10.1145/37499.37505
[21] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 6 (Nov. 2006), 937–977. https://doi.org/10.1145/1217856.1217859
[22] Eric Steven Raymond. [n. d.]. Applying the Rule of Least Surprise. http://www.catb.org/esr/writings/taoup/html/ch11s01.html.
[23] Robin Seggelmann, Michael Tuexen, and Michael Glenn Williams. 2012. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520. https://doi.org/10.17487/rfc6520
[24] Juho Snellman. [n. d.]. What's wrong with pcap filters? https://www.snellman.net/blog/archive/2015-05-18-whats-wrong-with-pcap-filters/.
[25] Nik Sultana and Richard Mortier. 2016. Kneecap: Model-based Generation of Network Traffic. In *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories affiliated with the International Joint Conference on Automated Reasoning, SMT@IJCAR 2016, Coimbra, Portugal, July 1-2, 2016. (CEUR Workshop Proceedings)*, Tim King and Ruzica Piskac (Eds.), Vol. 1617. CEUR-WS.org, 4–14. http://ceur-ws.org/Vol-1617/paper1.pdf
[26] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 33–47. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi
[27] Rafal Wojtczuk. [n. d.]. libnids. http://libnids.sourceforge.net/.