Kneecap: model-based generation of network traffic[*]

Nik Sultana and Richard Mortier

Computer Lab, Cambridge University

Abstract

Packet generation is an important activity for network administration and security. Tools for packet generation work through template instantiation and are used in an imperative programming style. We describe a new design for a *declarative* packet generator that affords users rich expressiveness to describe the packets they wish to generate. This relies on a domain-specific language for describing packets and constraints over them. This is translated into bitvector constraints that are dispatched to an SMT solver. The resulting bitvector solutions are then concatenated and composed into the different layers of the network protocol stack, and can be sent over the network interface. In this paper we describe a library implementation of this approach, and evaluate its extensibility and scalability.

1 Introduction

Packet generation is an essential part of network testing and active monitoring and measurement during configuration, debugging, fuzz-testing, and penetration testing. For example it is used to test how a router reacts to packets that have spoofed IP addresses; or whether a firewall would forward, or itself fall victim to, abnormal packets in the style of ping-of-death;[1] or whether a receiver could be put into an error state by changing flags in packets sent to them, in the style of christmas-tree packets; or to fingerprint the devices on a network [6].

Current research in networking emphasises the importance of flexibility in the configuration and tooling for networks, to cope with the increasing diversity, sizes and throughput of networks [4]. Current tooling for packet generation relies on instantiating packet templates, using a GUI or through an API. Only limited constraints over packets are possible.

In this paper we describe a packet generation tool that offers better flexibility than the state of the art, by providing a declarative interface for packet generation. Our design is based on the observation that traffic generation can be reduced to solving bitvector constraints that are within the grasp of SMT solvers. Because of our reliance on solvers, and the latency this incurs, we anticipate that this method is mostly suitable for offline packet generation. Generated packet traces could then be played over the network at high speeds.

Suppose that we wished to filter IPv4 broadcast or multicast packets that were encapsulated in Ethernet frames having 0 as the first byte of the source address. This could be expressed using the widely-used *pcap expression* domain-specific language [7] as follows:[2]

```
ether[0] = 0 and ip[16] >= 224
```

The expression specifies a filter for Ethernet frames whose first byte is 0, and that encapsulate IPv4 packets whose 17th byte must be at least 224. Interpreting this pcap expression presupposes knowledge of the packet formats concerned—to recognise when an Ethernet frame contains an IP packet, for instance. Furthermore, we implicitly assume that the expression will

[*] Supported by the EPSRC project "NaaS: Networks as a Service" (EP/K034723/1 and EP/K031724/1).

[1] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0128

[2] Based on an example from the tcpdump website: http://www.tcpdump.org/tcpdump_man.html

only be tested on 'well-formed' network traffic—since Ethernet frames with incorrect checksums are usually silently dropped by the network interface.

By making such knowledge and assumptions explicit, we obtain a form of expression that can be used to *generate* packets. For the Ethernet part of a query, one would formulate:

$$\begin{aligned}
\texttt{frame} &= \overbracket{\texttt{src\_add}}^{48} \cdot \overbracket{\texttt{dst\_add}}^{48} \cdot \overbracket{\texttt{ethtype}}^{16} \cdot \overbracket{\texttt{pload}}^{P} \cdot \overbracket{\texttt{fcs}}^{32} \qquad (*) \\
\texttt{src\_add} &= \boxed{\texttt{00}\ \boxed{\texttt{* * * * *}}} \\
\texttt{ethtype} &= \boxed{\texttt{08 00}} \\
\texttt{fcs} &= \mathbf{crc32}(\texttt{src\_add}, \texttt{dst\_add}, \texttt{ethtype}, \texttt{pload})
\end{aligned}$$

In this notation, symbols like `frame` and `src_add` stand for constants—fixed but possibly unknown values. These values consist of bitstrings, the size of which is fixed. For example, `src_add` is 48 bits wide. It is not currently clear what the width of `pload` is, therefore its length value is represented by the variable P, which we italicise to indicate that it is a variable. **crc32** is a special function that computes the frame's checksum.

We follow the structure of Ethernet frames;[3] `frame` denotes the bitstring that comprises the whole frame, formed from the concatenation of the following substrings: `src_add` (source address), `dst_add` (destination address), `ethtype` (type of protocol encapsulated in the payload), `pload` (the payload bitstring) and `fcs` (frame check sequence—i.e., checksum value).

The equations given above constrain the value of `frame`, and of its substrings `src_add`, `ethtype`, and `fcs`. Literals are shown in grey boxes—containing bytes in hexadecimal notation, in this example. An ethtype of 0800 indicates that an Ethernet frame encapsulates an IPv4 packet. Wildcards are indicated using asterixes, and by breaking out of the grey box back into a white background to indicate that wildcards are interpreted by the meta-language rather than the object-language. Thus, the source address constraint above specifies that it must start with a 0 byte, followed by any byte values.

The example given above is simple, but formulating and solving such problems can be difficult. Packet formats sometimes contain dependencies within packets, and even across different sorts of encapsulated packets. Generating such bitvector constraints is ripe for automation.

We noticed that packet *encapsulation*—that is, carrying packets of one protocol inside another, such as IPv4 over Ethernet—can be exploited to break up constraint problems into smaller problems that can be solved separately. For instance, in the problem described above on IP broadcast or multicast, we would first solve the IP-related constraints to produce a candidate IP packet bitstring, let's equate it with the constant `ip_soln`. We then use this to constrain the solution of the Ethernet frame in equation $(*)$ by asserting `pload` = `ip_soln`. This gradual, inside-out generation of a frame is more tractable than attempting to immediately generate a bitstring for the entire Ethernet frame. (Standard frames can be around 1500 bytes long, or 12Kbits.) Later we describe an example of a stack of encapsulations that is six protocols deep.

Contributions. We describe the first application of SMT solvers to the problem of network traffic generation, and produce a more declarative interface to packet generation. For improved performance we exploit the layer-based abstractions used in network protocols to reduce the complexity of constraint-solving. We implement this method as a .NET library, and example F# code of its use can be seen in Figure 1. We evaluate our system using a non-trivial packet generation example, and make both code and data available online.[4]

[3] https://en.wikipedia.org/wiki/Ethernet_frame

[4] http://www.cl.cam.ac.uk/~ns441/kneecap/

```
 1  use eth = new ethernet(184u)
 2  use ip = new ipv4(180u)
 3
 4  eth.constrain <@ ethernet.source_address =
 5                     ethernet.mac_address "[1-5,10]:34:56:78:90:*"
 6                   && ethernet.ethertype = ethernet.ethertype_ipv4 @>
 7  <== ip.constrain <@ ipv4.version = 4 &&
 8                      ipv4.source_address =
 9                        ipv4.ipv4_address "10.10.10.[55-60]" &&
10                      ipv4.source_address = ipv4.destination_address &&
11                      ipv4.internet_header_length = 5 &&
12                      ipv4.total_length = 170 &&
13                      ipv4.TTL = 5 &&
14                      ipv4.protocol = ipv4.protocol_ip_in_ip
15                    @>
```

Figure 1: Using our library (§5), this F# code instantiates the Ethernet template on line 1 using the standard `new` operator, to create frames sized at most 184 bytes. It also instantiates the IPv4 template on line 2, to create packets of at most 180 bytes. The Ethernet template is constrained further on lines 4-6, and IP instances on lines 7-14. The IP instance is encapsulated in the Ethernet instance using the `<==` operator on line 7. The `constrain` method is exposed by each protocol in our library. This method is passed an expression written in our DSL. These expressions are enclosed in `<@..@>` to use F#'s introspection feature, which implicitly type-checks the expressions. The expressions are then translated into bitvector constraints by our system as described in §4.1, which also manages the exchange for solutions with the back-end SMT solver following Figure 3.

We provide background and describe related work in the next section. In §3 we describe our model for packets and constraints, and in §4 our overall architecture and translation into bitvector formulas. We describe and evaluate our implementation in §5.

2 Background and Related Work

Traffic generation. A packet format describes a scheme or template that all packets in that protocol must instantiate. All the packet generation tools that we are aware of are based on instantiating templates, and their interface allows you to pick a packet type (say, DNS) and specify the values of its fields. These tools range from software-only tools such as iperf, netsniff-ng[5], hping[6], and Scapy[7] to packet generators on specialised hardware [2, 1], which could be seeded by a template provided as a pcap file. Note that "pcap" can refer both to an expression language, as in the previous section, or to a file format for a sequence of packets.[8] The contents of this file might comprise traffic that arrived on a network interface, or it might have been generated by a tool. In §5 we describe our such tool, the output of which can be processed by standard network software, and sent over existing networks.

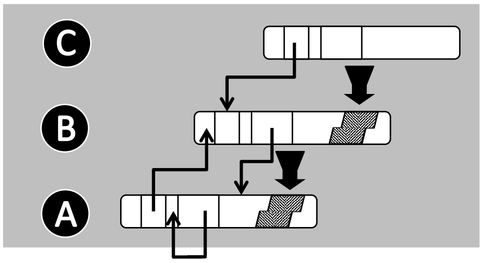
[5] http://netsniff-ng.org/

[6] http://www.hping.org/

[7] http://www.secdev.org/projects/scapy/

[8] The pcap file format is described at: https://wiki.wireshark.org/Development/LibpcapFileFormat

Figure 2: Templates relating to packet formats A, B, and C are instantiated and encapsulated. For example, A could be Ethernet, B could be IPv4, and C could be ICMP. Thin arrows indicate constraints between fields (in the same packet, or across packets). Thick arrows indicate *encapsulation*: one packet forms the *payload* of the packet in the layer below.



Often, additional constraints need to be satisfied in order for the traffic to be well-formed. For example, the checksum and length fields must contain the correct values.

Most existing tools implicitly model the packet as a scheme. Our system's main advantage is that it can handle arbitrary logical constraints between fields and values, and it allows a fully declarative style of expression (in terms of equations and inequations, and using standard arithmetic and Boolean operators). All other systems follow an imperative style of packet generation, where for-loops are used to test and set field values of individual packets, to encode inter-layer constraints.

Bitvector formulas. Packet formats are described at the bit and byte level. Fixed-width bitvectors are among the theories supported by SMT solvers. The solutions found by SMT solvers can almost be sent directly onto the network, save for any reordering of bytes that the packet format might require.

Bitvector expressions consist of bits, strings of bits, their concatenation, and arithmetical and logical operations over them—such as addition, shifting, etc. Bitvector formulas consist of atoms asserting equality or inequality between bitvector expressions, and the usual logical formation rules over them (such as negation). The precise details are not important here, and a more detailed explanation is given by Kroening and Strichman [5].

Using existing packet-generation tools, we can generate packets by instantiating templates using directives such as $\texttt{src\_port} = 22$ and $17 < \texttt{src\_port} \leq 22$. Using the model-based approach described in this paper we can express more complex constraints relative to different fields across protocol layers, such as

$$(p.\texttt{src\_port} = 22) \ \vee \ (p.\texttt{src\_port} < (2 \times q.\texttt{dst\_port}))$$

where p and q are transport protocols at different layers. Using the design described in the next section, a user could express arbitrary constraints over a stack of encapsulated protocols.

3 Packets and constraints

In this section we describe the information communicated by a packet specification, which we will use to generate instances of that packet. Then in the next section we describe how we use an SMT solver to generate the packet.

A packet specification communicates three bits of information: (i) the packet formats involved, (ii) how the instances of different packet formats should be encapsulated, (iii) constraints within packets, or across encapsulated packets. Figure 2 illustrates these different sorts of constraints.

3.1 Objectives

Our goal is to generate network traffic by exploiting the expressiveness and automation enabled by current SMT technology. More specifically, we aim to provide a language that allows:

1. encapsulation of any packets inside payload-carrying packets;[9]
2. constraints on all packet fields;
3. constraints relating different fields in a packet;
4. constraints relating fields in different packets (which are related by encapsulation);
5. extension to support new packet formats (for instance, to support a modified form of TCP in a datacentre).

We later relaxed requirement 2 since constraints over "computational" fields (such as a checksum) do not make much practical sense, as described in §4.1.1. Consequently we also relaxed requirements 3 and 4 for such fields.

3.2 Describing constraints on packets

Conceptually, we aim to combine the expressive language that can be processed by SMT solvers, with a generic description of protocol interfaces. Constraints on packets consist of bitvector formulas over a signature that is extended to interpret protocol-related fields and their values. We now elaborate how these concepts are extracted from protocol specifications, and how they are made available in our language.

A *packet format* is the defining description of a packet, specifying its fields, their widths, their dependencies on other fields, how bytes are ordered, and so forth. This information is typically extracted from RFCs.[10] Packet formats are usually simple, to facilitate parsing at high speeds by network elements or end-hosts. Despite being simple, packet formats are highly diverse. It is difficult to have a complete yet constrained language for describing packet formats, and research on this is ongoing [8]. As a result, people often use general-purpose languages to describe protocols. We too took this approach, where we used F# to describe protocol formats in our library.

From a packet format we extract what we refer to as the *packet interface*. This specifies the vocabulary to refer to parts of the packet—the names of fields—or its values—legal contents of a field. The interface largely consists of a presentation of the packet format to the user of our domain-specific language. It specifies precisely what symbols can be used in constraints, and what they mean to the packet's description. There are three types of symbols:

- *fields* are names of fields, such as 'ethertype' on line 6 in Figure 1;
- *constants* are more meaningful symbols for numeric values, such as 'ethertype_ipv4' for `08 00`;
- *protocol-specific functions* interpret values supplied by the user, into bitvectors. For instance, they parse notations such as "192.168.1.1" and "192.168.1.0/30". The meaning of these notations is local to a single packet format.

Note that the packet format specifies how fields are concatenated to form a packet, while the packet interface is not concerned with this. The interface describes the *protocol-specific meaning* of fields, constants and protocol-specific functions.

[9] Ethernet and IPv4 are examples of payload-carrying protocols, whereas ARP does not carry a payload.

[10] http://www.ietf.org/rfc.html

For example, the header formats for Ethernet, IPv4, and IPv6 have fields for a 'source address', but they differ in the width and representation of what they each mean by 'source address'. The packet interface allows us to resolve precisely what we mean when we speak of a 'source address' in the context of a particular protocol.

The atoms making up the packet interface are combined through logical connectives and arithmetical operators to form constraints over packets. Unlike the symbols in a packet interface, the meaning of logical connectives and arithmetical operators is the same across all packets.

Using the vocabulary described above, we can form constraints over packets' fields, and thus over packets. Unconstrained fields can, by default, be assigned any value by the solver—unless we implicitly constrain the model to generate only well-formed traffic, as described in §4.1.2.

If the constraints are too strong, then it may be that there is no solution to constraints—this did arise during testing, when we mistakenly constrained a packet's field to be equal to two distinct values.

4 Architecture

We now turn to how the specification described in the previous section is translated into bitvector constraints and passed to a back-end solver. Recall that a specification may reference other packet specifications when one packet encapsulates another.

Figure 3 outlines our architecture, which serves to translate high-level packet constraints into bitvector constraints for each protocol layer. The constraints of successive layers are solved, and the solutions are used to constrain the generation of lower-layer packets. Thus, we generate a packet gradually by taking advantage of how protocols are layered to form a stack, rather than generate a full outer-level packet at one go.

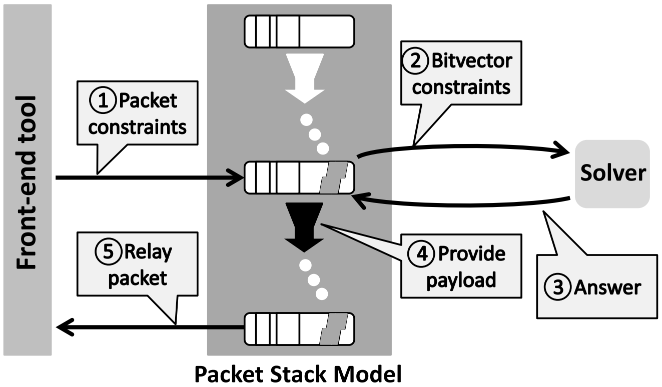


Figure 3: Flow of information in the system. (1) Constraints are received from a front-end tool. (2) Packet constraints are translated by instances of packet templates, into bitvector constraints. Constraints are processed from the inner-most payload, proceeding outwards one layer at a time. (3) The solver returns an answer consisting of a packet, otherwise the generation of all encapsulating layers fails too. If the packet is encapsulated inside another packet, then (4) the answer is returned to the encapsulating packet as a payload (constraint), and we now attempt to solve the constraints for the encapsulating packet. Otherwise, the packet must be the lowest-level packet, therefore (5) it is returned to the front-end caller.

4.1 Translation to bitvector formulas

The translation of user-supplied specifications, such as that in Figure 1, is syntax directed and covers the classes of syntax described in §3.2: theory (arithmetical and logical) constants (such as conjunction, `&&`), protocol-specific constants (such as `ethernet.ethertype_ipv4`), and protocol-specific functions (e.g., `ethernet.mac_address "[1-5,10]:34:56:78:90:*"`). Let $T_P(t)$ be our translation function, where P is a parameter indicating the default protocol whose constraints we are processing, and t is the term being translated. For example, translating `ethernet.ethertype_ipv4` involves evaluating $T_{\text{ethernet}}(\text{ethertype\_ipv4})$. Constants such as `ethertype_ipv4` are translated to distinguished bitvector constants. Theory constants (such as the addition operator) are translated to bitvector equivalents. Protocol-specific functions are first applied to their arguments to produce an arithmetical expression in our constraint language, which is then translated recursively into bitvector constraints.

For example, lines 4-6 of Figure 1 are translated into the following bitvector problem when written in SMT-LIB syntax:

```
(let ((a!1 (concat (concat (concat (concat range0 #x34) #x56) #x78) #x90)))
(let ((a!2 (= src_mac (concat a!1 wild1)))))
  (and (= ethertype #x0800)
       a!2
       (or (= range0 #x0a)
           (= range0 #x01)
           (= range0 #x02)
           (= range0 #x03)
           (= range0 #x04)
           (= range0 #x05))
       )))
```

The symbol `src_mac` is a distinguished constant that is mapped from `ethernet.source_address`. `range0` and `wild1` are fresh constants generated by the `ethernet.mac_address` protocol-specific function: `range0` is constrained to the values in $\{1, 2, 3, 4, 5, 10\}$ (following "[1-5,10]") whereas `wild1` is unconstrained since it is translated from the (wildcard) asterisk character. The constant hex value `x0800` is the number mapped from `ethernet.ethertype_ipv4`, indicating that an Ethernet frame is carrying an IPv4 packet. This mapping is defined by an IEEE standard.[11]

Packet constraints can make reference to fields in other layers. We found it useful to limit the kinds of references that can be made, as follows: a packet's constraints may only refer to fields in encapsulated packets, not vice versa. This fits our model, drawn in Figure 3, in which solutions flow *downwards* in the packet stack: a solution at one layer becomes part of a solution at lower network layers. For example, in Figure 2, A's constraints may refer to the fields of B and C, but C's constraints may not refer to any other layer's fields.

Other than the careful handling of names, the translation to bitvector formulas is straightforward, since the target (bitvector constraint) language can directly interpret the operators used in our source language—such as addition, shift, negation, etc. Our implementation includes a straightforward analysis to compute any necessary extensions that are needed to bitvectors. For instance, in order to compare two values they must be of the same bit width; this is a matter of zero-extending the smaller one to be the same size as the other. The user is spared having to specify mundane details.

[11] The official list of values can be obtained from: http://standards-oui.ieee.org/ethertype/eth.txt

4.1.1 Constraints involving computations

We came across two computations that need to be frequently executed. The first is checksum computation, and the second is byte-order transformation. We decided to exclude such computations from constraints.

In principle, checksums could be computed by the solver. We encoded the checksum algorithms used by IPv4 and by Ethernet. The first is simple, and its data is small, consisting only of the IP header. In contrast, the CRC32 algorithm to compute the Ethernet checksum is much more complex, and its data consists of the entire Ethernet frame. We found that the CRC32 computation to be impossibly slow when done using the solver, even for relatively small frames. We therefore decided that this computation should occur outside the solver, after the solver has produced values for other fields. This means that users will not be able to specify constraints over checksum values, but it seems very unlikely they would need to.

Depending on the *endianness* of the host's architecture, it can happen that the multi-byte values generated by the solver need to be reversed before writing them to a pcap file or sending them over the network. Modifying the byte order *could* be done using the solver, but since it is not a search problem, it seems best to do this outside the solver to avoid incurring overhead.

4.1.2 Modes

We identified four basic modes of operation for a packet generation tool. Each mode relates to different intended uses for the generated traffic, and involves strengthening the constraints supplied by the user through additional constraints.

1. In **manual** mode the user can specify any constraint. They may also specify checksum values since the checksum algorithm is not run in this mode. Byte-reordering is done however, otherwise the packets' contents would not be interpreted correctly by the receiving end. In this mode, the user may deliberately generate invalid instantiations, for example, by using an IPv4 template and setting the version field to the value 3.
2. **Checksum-supported** mode is like manual mode, except that the checksum is computed for the solution obtained from the solver. This follows the description in §4.1.1.
3. **Locally well-formed** mode specifies that, in addition to having a valid checksum, for a packet to be well-formed its fields need to have values from a valid range for values. For instance, in IPv4 the 'version' must be 4, the 'Internet header length' field's value must not be less than 5, and the 'total length' field must contain the correct value.
4. **Fully well-formed** mode additionally involves asserting inter-layer consistency. For example, encapsulating IPv4 in Ethernet will restrict the latter's range of ethertype values to precisely a single one: that for IPv4.

In manual mode the system does not provide any support. A packet generator operating in this mode could be used by another tool that might make its own processing. In other modes, the generator adds progressively stronger background constraints to those supplied by the user.

5 Implementation and evaluation

We implemented the method described in the paper in a library called Kneecap. This was implemented in F# and uses Z3 as the backend solver [3]. The library contains the specifications for Ethernet, ARP, IPv4, and EtherIP, all of which can be nested in arbitrary order. The deepest nesting we tested contained six layers, as described in §5.2.

Our implementation has the following limitations: currently only fixed-sized packets may be generated—generating variable-sized packets is future work; and the system only works in **manual mode**, described in §4.1.2.

The library contains supporting functions to generate a number of packets, and write them to a pcap file, which can then be played out on a network interface using a tool like tcpreplay[12] or examined using a tool like Wireshark.[13]

5.1 Evaluation

Encapsulation. We experimented with stacking different packets; for ease of expression we will refer to a given configuration of a stack of packets as a *stacket*. Our most complex stacket consisted of the following packets, in order: Ethernet, IPv4, IPv4, EtherIP, Ethernet, ARP. Each of these were assigned constraints. Those for the first two layers are exactly those shown in the snippet in Figure 1, and those for the remaining layers are shown in §5.2. We then generated a pcap file for distinct sequences of this stacket, and checked that the files were readable by Wireshark.

Performance. We anticipate that our packet-generation method will be used *offline*, and therefore its rate of generation is not a likely concern, but we sought to measure this nonetheless. We evaluated Kneecap in a Windows 8.1 VM with 8GB RAM running on a 2GHz Core i7 MacBook Pro with 16GB RAM and SSD secondary storage. We generated 1000 64-byte unique Ethernet frames four times, and found that the average time was 18.99ms. We then generated 1000 unique 584-byte Ethernet frames four times, and saw that the average time was 609.1ms. Carrying out a larger range of tests outside a VM is future work.

5.2 Specifying arbitrarily-encapsulated packets

In §5.1 we used a specification involving a stack of six protocols. This stack is specified in this section. The stack builds on the "ip" instance defined in line 2 of Figure 1 (and further constrained between lines 7 and 15 of the same figure). The extended specification is shown below.

```
ip +==
  [(new ipv4(150u)).constrain
    <@ ipv4.version = 4 &&
       ipv4.source_address = ipv4.ipv4_address "192.168.4.[55-60]" &&
       ipv4.destination_address =
         ipv4.ipv4_address "194.100.[1-254].[10-20]" &&
       ipv4.internet_header_length = 5 &&
       ipv4.total_length = 150 &&
       ipv4.TTL = 7 &&
       ipv4.protocol = ipv4.protocol_etherip
     @>;

   (new etherip(100u)).constrain <@ etherip.version = 3 @>;

   (new ethernet(80u)).constrain
```

[12] https://github.com/appneta/tcpreplay

[13] https://www.wireshark.org/

```
 <@ ethernet.source_address =
      ethernet.mac_address "00:11:22:33:44:55" &&
    ethernet.destination_address =
      ethernet.mac_address "13:24:35:46:57:68" &&
    ethernet.ethertype = ethernet.ethertype_arp @>;

(new arp<ethernet, ipv4>(eth, ip)).constrain
 <@ arp<ethernet, ipv4>.HTYPE = arp<ethernet, ipv4>.HTYPE_ethernet &&
    arp<ethernet, ipv4>.PTYPE = arp<ethernet, ipv4>.PTYPE_ipv4 &&
    arp<ethernet, ipv4>.HLEN = 6 &&
    arp<ethernet, ipv4>.PLEN = 4 &&
    arp<ethernet, ipv4>.OPER = arp<ethernet, ipv4>.OPER_Reply
  @>]
```

In the above code, we encapsulate IPv4, EtherIP, Ethernet, and ARP packets, on top of the IPv4 instance from Figure 1. The `+==` operator takes a list of templates on the right, and encapsulates them in reverse order.

Acknowledgements. We thank Markulf Kohlweiss, Carles Gomez Montenegro, Robert Soulé, Daniel Thomas, Christoph Wintersteiger, and the anonymous reviewers for helpful feedback.

References

[1] G. Antichi, M. Shahbaz, et al. OSNT: Open Source Network Tester. *Network, IEEE*, 28(5):6–12, 2014.

[2] R. Bolla, R. Bruschi, et al. A High Performance IP Traffic Generation Tool Based on the Intel IXP2400 Network Processor. In *Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements*, pp. 127–142. Springer, 2006.

[3] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

[4] N. Feamster, J. Rexford, et al. The road to SDN. *Queue*, 11(12):20, 2013.

[5] D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view.* Springer, 2008.

[6] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[7] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference*, pp. 2–2. USENIX Association, 1993.

[8] R. Sommer, M. Vallentin, et al. HILTI: An Abstract Execution Environment for Deep, Stateful Network Traffic Analysis. In *Internet Measurement Conference*, IMC ’14, pp. 461–474. ACM, 2014.