Kneecap: model-based generation and analysis of network traffic

Nik Sultana  
Computer Lab, Cambridge University  
nik.sultana@cl.cam.ac.uk

Richard Mortier  
Computer Lab, Cambridge University  
richard.mortier@cl.cam.ac.uk

ABSTRACT

Current research in networking emphasises the importance of flexibility in the tooling and configuration of networks. Traffic generation and analysis is an essential part of network configuration and debugging, but the tooling is not satisfactorily flexible in what it allows users to express.

We describe a new design for network traffic tools, one that affords users more expressiveness to describe the packets to generate or analyse. This design involves translating users’ expressions into constraint-satisfaction problems, that are dispatched to general reasoning tools for solution.

The description language subsumes the expressiveness of pcap filter expressions. Using this design we can use the same description to test network traffic (by verifying a fully-specified constraint-satisfaction problem) and to generate traffic (by solving an under-specified CSP).

We describe a library implementation of this approach, and evaluate its extensibility and scalability.

Categories and Subject Descriptors

C.2.3 [Network Operations]: Network monitoring;  
D.2.5 [Testing and Debugging]: Testing tools—data generators

General Terms

Experimentation, Measurement, Theory

Keywords

traffic generation, packet filtering, SMT

1. INTRODUCTION

The tools for administering and debugging networks should flexibly support new protocols, combinations of existing protocols, and allow users to express their queries easily. Current tools for packet generation mainly work through template instantiation, and only allow users to express limited constraints over packets. We describe how to build traffic analysis and generation tools that offer better flexibility than the state of the art. Our design is based on the observation that both traffic generation and analysis can be reduced to solving similar problems, and that these problems are within the grasp of generic reasoning engines used in logic and software verification.

This idea is inspired from problems that one frequently encounters in maths: Solve

\[ n^2 - n - 90 = 0 \] (1)

A similar notation is used to specify filters for network traffic, using the now-standard pcap expression language. Consider this example of a pcap expression, taken from the tcpdump website:

\[ \text{ether}[0] & 1 = 0 \text{ and ip}[16] \geq 224 \]

Given a stream of network traffic, this expression can be used “To print IP broadcast or multicast packets that were not sent via Ethernet broadcast or multicast.” The expression specifies that the first byte of the Ethernet frame must be 0, and the 17th byte of the IP header must be at least 224. Interpreting this pcap expression presupposes knowledge of the packet formats concerned, and implicitly assumes that the expression will only be tested on ‘well-formed’ network traffic—since Ethernet frames with incorrect checksums are usually silently dropped by the network interface, for example.

By making such knowledge and assumptions explicit, we obtain a form of the pcap expression that can be used for generating traffic. One would start formulating this as follows for the Ethernet part:

\[
\begin{align*}
\text{frame} &= \text{src_add} \cdot \text{dst_add} \cdot \text{ethtype} \cdot \text{pload} \cdot \text{fcs} \\
\text{src_add} &= 00 \quad * * * * \\
\text{ethtype} &= 08 \quad 00 \\
\text{fcs} &= \text{crc32}(\text{src_add}, \text{dst_add}, \text{ethtype}, \text{pload})
\end{align*}
\]

In this notation, symbols like frame and src_add stand for constants—fixed but possibly unknown values. These values consists of bitstrings, the size of which is fixed. src_add is 48 bits wide; it is not currently clear what the width of pload is. The equations given earlier constrain the values of src_add, ethtype, and fcs,
which in turn constrain frame. Literals are shown in grey boxes—containing bytes in hexadecimal notation, in this example. Wildcards are indicated using standard notation, and by breaking out of the grey box back into a white background.

Formulating and solving such problems is sometimes difficult. Packet formats sometimes contain dependencies within packets, and even across different sorts of encapsulated packets. This will be dealt with in later sections. One of the most satisfactory realizations is that the abstraction afforded by packet encapsulation can be exploited to break up constraint problems, and solve them separately. For instance, in the problem above, we would first solve the IP-related constraints, then use the IP packet to constrain the solution of the Ethernet frame.

Finally, the above formulation can be used for testing as well as generation. For example, if we wanted to test a solution to equation 1, say \( n = 9 \), then we would assert the equation and the solution together to the solver, and it would tell us whether they are consistent.

\[
(n = 9) \land (n^2 - n - 90 = 0)
\]

Similarly, network traffic can be tested by decomposing it into its fields, and asserting them together with the equation and its reserved field must be set to 0).

\[
sr_{c\_add} = \text{00 5E B8 0A 10 F3}
dst_{\_add} = \text{18 B2 67 AC F0 33}
ethtype = \text{08 00}
\]

2. BACKGROUND AND RELATED WORK

Traffic analysis and 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\(^3\) and Scapy\(^4\), to packet generators on specialised or reconfigurable hardware [3, 1], which could be seeded by a template provided as a PCAP file.

Often, additional constraints need to be satisfied in order for the traffic to be well-formed. For example, the checksum field must contain the right checksum value for that packet, and a length field must specify the right value.

Most existing tools implicitly model the packet as a scheme, which could be written as a formula, as described in the previous section. This could capture dependencies between fields, and between encapsulated packets too. Our system’s main advantage is that it can handle arbitrary logical constraints between fields and values, and has a more declarative style of expression (in terms of equations and inequations, and using standard arithmetic and Boolean operators).

Another line of research builds probabilistic models of network traffic, and then ‘runs the model forward’ to generate traffic using that model [8]. This can generate traffic patterns that are consistent with the observations on which the model was based. They use models that are qualitatively different than the models used here, but there might be potential for combination of the two approaches. The models used here are discrete models based on the vocabulary provided by a packet format—its fields, their widths, and the range of values they accept—and constraints that relate fields in this vocabulary (for instance, the EtherIP RFC\(^5\) states that valid packet instances must have its version set to 3, and its reserved field must be set to 0).

Constraint satisfaction.

Automated reasoning tools, such as those used for solving constraint satisfaction problems, have improved dramatically during the past few decades, and found application in industry. Boolean Satisfiability (SAT) solvers are heavily used in hardware verification [2]. During the last two decades, much research has been done on so-called Satisfiability Modulo Theories (SMT). Whereas in SAT you can only test for the consistency of sets of propositional formulas, SMT formulas have a much broader universe of discourse, ranging across integer and real arithmetic, bitvectors and arrays.\(^6\) These solvers have been applied to a variety of problems, ranging from configuration management [6] to real-world software checking [4]. They have not yet been applied to generating network traffic.

Bitvector formulas.

Packet formats are described at the bit and byte level. Fortunately, fixed-width bitvectors are among the theories supported by SMT solvers. This means that 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 (negation, etc). The precise details are not important here. A more detailed

\(^3\)http://netsniff-ng.org/

\(^4\)http://www.secdev.org/projects/scapy/


\(^6\)http://smtlib.org/
Figure 1: Using our library (§6), this F# code instantiates the Ethernet template on line 1, 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 4-5, and IP instances on lines 6-13. The IP instance is encapsulated in the Ethernet instance using the $$\Leftarrow$$ operator on line 6.

explanation is given by Kroening [7].

Using existing tools, we can generate packets by instantiating templates using directives such as $$\text{src} \text{port} = 22 \text{ and } 17 < \text{src} \text{port} \leq 22$$. Using the model-based approach we can express much more complex constraints, relative to different fields across protocol layers, such as $(p.\text{src} \text{port} = 22) \text{ || } (p.\text{src} \text{port} < (2 \times q.\text{dst} \text{port}))$

where $p$ and $q$ are transport protocols at different layers. Using the design described in the next section, the user can express arbitrary constraints over a stack of encapsulated protocols. The constraints will be transparently translated into bitvector formulas, which are dispatched to a solver.

3. PACKETS AND CONSTRAINTS

3.1 Usage model

We want to retain the existing model, where specifying traffic analysers and generators starts by instantiating blocks representing packet templates. Then one specifies constraints over instantiated templates, and combines them into a stack of protocols. (These steps are done explicitly or implicitly, depending on the system being used.) This results in a structure as shown in Fig. 3.1. In this section we describe the conceptually important parts of this structure. Then in the next section we describe the interface with an SMT solver.

3.2 Objectives

Our design is intended to make available the expressiveness afforded by SMT solvers to generate and check network traffic. Our objects are to allow:

- encapsulation of any packets inside payload-carrying packets;
- constraints on all fields of packets;
- constraints relating different fields in a packet;
- constraints relating fields in different packets (which are related by encapsulation);
- extension to support new packet formats.

3.3 Levels of description

Our objectives identify multiple levels of expressions that need to be supported:

1. **Packet template**: this is the most general description of a packet, describing its fields, their widths, their internal and external dependencies, how bytes are ordered, and so forth. This information is typically extracted from RFCs.

Figure 2: Templates relating to packet formats A, B, and C are instantiated. 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.
2. Packet interface: this specifies the vocabulary to refer to parts of the packet or its values. It largely consists of a presentation of the ‘Packet template’, seen above, to the user, but it plays an important role: 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 5 in Fig. 1;
- **constants** are more meaningful symbols for numeric values, such as ‘ethertype.ipv4’ for 08 00;
- **interpretations** are packet-specific functions that 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, and should be handled locally within that format’s module.

Fields, constants and interpretations have packet-specific meaning. For example, 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 template interface allows us to resolve precisely what we mean when we speak of a ‘source address’ in the context of a particular protocol.

3. Constraint and expression formation: this concerns symbols that have the same meaning across all packets. This includes logical connectives, such as the symbols for ‘and’ and ‘not’; arithmetical operators, such as ‘+’; and binary operators, such as the symbols for XOR and left shift. Irrespective of whether we speak of Ethernet or IPv4, addition always has the same meaning.

Conceptually, we aim to combine the expressive language that can be processed by SMT solvers, with a generic description of protocol interfaces. Interfaces consists of general templates, which users can flexibly constrain. We describe how this is achieved in the next section, and how packet descriptions can be flexibly encapsulated and stacked.

4. ARCHITECTURE

This section describes how the protocol templates described in the previous section interact with one another (when one encapsulates the other), and with the back-end solvers.

Fig. 4 outlines our architecture. Rather than generating a packet at one go, we take advantage of the abstraction provided by the layering of protocols, as reflected in the encapsulation of their packets. This layering is reflecting in the model built by composing together the templates for each packet, in encapsulation order. Each layered template is responsible for translating the constraints on its packets to a form that can be processed by a back-end solver. This is described next.

4.1 Translation to bitvector formulas

If a packet constraint is purely intra-layer (it does not make reference to fields in other layers), then each field is translated to a unique name in the resulting bitvector formula, so the solver will be able to distinguish between fields. The solver will produce solutions for all fields within the given constraints.

Packet constraints can make reference to fields in other layers. We found it useful to limit the kinds of references that can be made: a packet’s constraints may only refer to fields in encapsulated packets, not vice
versa. This fits our model, drawn in Fig. 4, in which information flows downwards in the packet stack: a solution at one layer becomes part of a solution at each lower layer. For example, in Fig. 3.1, 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 language can directly interpret the operators used in the 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 bitwidth; 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.

4.1.1 Constraints involving computations

We came across two frequently-occurring deterministic 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 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

There appear to be four basic modes of operation that are most useful to most users. Each mode relates to different intentions for the traffic being generated, 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—the checksum algorithm is not run in this mode, but byte-reordering is done. In this mode, the user may deliberately generate invalid instantiations, for example, by using an IPv4 template and setting the version field to 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 range for values. For instance, in IPv4 the version must be 4, the Internet header length must not be less than 5, and the total length 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. This mode could be used by another tool using our system, if the tool was designed to make its own processing. In other modes, the system adds progressively stronger background constraints to those supplied by the user.

5. ANALYSING PACKETS

Analysing packets is a special case of generating them: we provide the solver with concrete values for all fields, together with the constraints translated from the model, and the model checks whether the concrete values are consistent with the model. If the two are consistent, then the packet passes the filter. Otherwise, the solver could point out one or all inconsistencies between the model and the packet.

6. IMPLEMENTATION AND EVALUATION

Kneecap was implemented in F# as a library, using Z3 as the backend solver [5]. 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 later.

It 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 or examined using a tool like Wireshark.
6.1 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. We then generated a PCAP files for distinct sequences of this stacket, and checked that they were readable by Wireshark.

Performance.

We evaluated Kneecap in a Windows 8.1 VM with 8GB RAM. We generated 1000 64-byte unique Ethernet frames for four times, and found that the average time was 18.99ms. We then generated 1000 unique 584-byte Ethernet frames for four times, and saw that the average time was 609.1ms. We then generated 1000 unique stacklets (with the configuration described earlier), and plotted the time it took to generate each successive packet. The result is shown in Fig.6.1. It appears that timing artefacts distorted the results; we tried regularly forcing a garbage collection but this had no noticeable effect. We plan to carry out a larger range of tests outside a VM.

7. CONCLUSION

We described the first application of SMT solvers to the problem of network traffic generation and analysis. Our work benefitted from the maturity of SMT technology, the complemantarity between bitvector formulas and packet descriptions, and from the designed modularity of layers of encapsulation. This enabled us to break down the problem of generating packets into generating a succession of smaller payloads.

The challenges we faced were largely due to quirks and heterogeneity between packet formats. Currently we only generate traffic for fixed size packets. We made early experiments with variable-size traffic, and have not yet resolved how to deal with optional header fields.

Our approach provides two main benefits over existing work. First, it allows using the same language to both specify and check traffic. Checking is simply a special case of generating, when the constraints are fully specified. Second, it allows asserting arbitrary constraints on packets, thus offering unprecedented flexibility.

8. ACKNOWLEDGMENTS

Funded by the EPSRC project “NaaS: Networks as a Service” (EP/K034723/1).

9. REFERENCES


