Secure networking

Conference Paper · August 2006

CITATIONS
0

READS
38

2 authors:

Andreas Bogk
HERE Technologies
6 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Hannes Mehnert
University of Cambridge
10 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Secure networking

Andreas Bogk <bogk@andreas.org>, Hannes Mehnert <hannes@mehnert.org>

August 2, 2006

1 Motivation

The security industry is in a paradox situation: many security appliances and analysis tools, be it IDS systems, virus scanners, firewalls or others, suffer from the same weaknesses as the systems they try to protect. What makes them vulnerable is the vast amount of structured data they need to understand to do their job, and the bugs that invariably manifest in parsers for complex protocols if written in unsafe programming languages.

Since we noticed a lack of a decent secure framework for handling network packets, we have designed and implemented major parts of a TCP/IP stack in the high level programming language "Dylan", focusing on security, performance and code reuse.

Dylan is a high level language that provides a number of features to detect and prevent data reference failures, one of the most common sources of vulnerabilities in C software. Bounds checks for array accesses are inserted where needed by the compiler. Also a garbage collector is used, avoiding the need to care about manual memory management, and preventing bugs from early frees or double frees. Dylan is strongly typed, so bypassing the type system by doing casts and pointer arithmetic is not possible.

Even though it is as easy to use as common scripting languages, Dylan programs are compiled to machine code. It bridges the world between dynamic and static typing by doing optimistic type inferencing: bindings can be type annotated, and types of expressions can be computed at compile time. This often eliminates type checks or function dispatch in the code.

The high level language features found in Dylan, like object orientation, macros (metaprogramming), multiple inheritance, first-class functions and multiple dispatch, help writing code in a highly abstract and compact style. The goal of good code is to only ever write down each piece of information once, to increase readability and maintainability, and to avoid the sorts of bugs that keep creeping into code written in a cut and paste manner.

In the following paper, we will show our design approach. In section 2 we introduce our approach for for parsing and assembling protocol frames. In section 3 we describe a generic filter language for frames, section 4 presents the packet flow graph mechanism we implemented. Finally section 5 describes a protocol stacking mechanism, and in section 6 we give a conclusion and outlook.

2 Packetizer

2.1 Motivation

The implementation of a network protocol stack involves parsing and assembly of a great variety of network packets. Even though some communication protocol suites, such as

the OSI stack, use ASN.1 for formal definitions, and ASN.1 compilers for reading and writing the appropriate binary representation, many popular protocols such as TCP/IP are defined in an ad-hoc way, and require manually written code for parsing and generating the protocol data units.

This manual process of implementing network stacks is both tedious and error-prone. Our goal was thus to come up with a formal description language for ad-hoc protocols, which then allows the automatic generation of protocol parsers and generators from concise protocol specifications.

We found inspiration from two sources. For the syntax and semantics of protocol definitions, as well as proof that an attempt to implement such a description is feasible and results in elegant code, we must name the network analysis tool "scapy". The idea to implement a description of the structure of octet sequences (which we call those **frames**) using the macro facilities of a high level language, forming what is known as a domain specific language (DSL), is due to the Genera (Symbolics Lisp Machine OS) "defstorage" macro.

2.2 Implementation approach

We implemented our protocol description DSL as a set of macros in the "Dylan" language. During compilation, the definition macro statement is expanded into a set of class and function definitions, which implement interfaces for parsing and generating packets for the desired protocol.

There are a number of classes that are defined for every protocol definition. One is an abstract class, representing any possible instance of a frame of the defined protocol. The others are subclasses of that class, which manage the low-level and high level representations of said frame.

The low-level representation simply associates a to-be-parsed byte vector with the appropriate type information. Information for a specific field of a frame can be extracted using getter functions on that class, which then calculate the field offset in the frame, and call the correct parser to convert the byte representation into a high level object. Additionally, this information is cached in the low-level object, to avoid repeated parsing of the same information.

The high level representation stores high level values for every field in the frame. This is used for generation of frames from code in higher layers. There is code which is able to assemble this high level representation into a byte vector, for subsequent transmission to the outside world.

For the semantics of the translation, we looked at common protocol definitions and tried to isolate the different aspects that make up a structure definition. Then, we mapped those aspects to a class structure. Finally, we wrote a macro system that builds appropriate subclasses from a protocol definition in the form of a macro call.

2.2.1 Frames and Fields

The first thing we noticed is that the structure of frames can be described recursively. Frames are made up of **fields**, which are identified by a name, e.g. the source-address field of an Ethernet frame. These fields correspond to certain subsequences of the frame, which in turn can be viewed as frames again. These subsequences don't have to fall on byte boundaries, it's thus possible to implement bit fields.

Some frames have no further structure, we call those **leaf frames**. An example for a leaf frame is a MAC address, it is simply defined as a byte vector of length six, as well

as some code to handle the common print representation of a MAC address. Frames that have fields are called **container frames**, an Ethernet frame is an instance of a container frame, as would be an IP option. The latter is also an example for a frame that is inside the field of another frame (the ip-options field of an IP frame). Fields of a frame can either have a single frame as their value (source-address in Ethernet frame), or they can represent multiple instances of the same frame (there are multiple IP option frames in the IP option field of an IP frame).

The class structure we derived from that consists of an abstract class <`frame`> with the two abstract subclasses <`leaf-frame`> and <`container-frame`>. Unsurprisingly, a MAC address is represented as a class <`mac-address`>, which derives from <`leaf-frame`>, whereas an Ethernet frame is represented using <`ethernet-frame`>, a subclass of <`container-frame`>. In fact, there will be multiple subclasses for container frames for different purposes, which will be explained later.

Fields in a container frame are represented using a class hierarchy rooted at the abstract superclass <`field`>. These associate a field name in a given container frame with the type of the frame in that field. Since there could be either a single frame in a field, or a list of frames of the same type, we have the two abstract superclasses <`single-field`> and <`repeated-field`>. Furthermore, sometimes the type of the frame in a field varies, depending on some protocol header values. For these cases, there are <`variably-typed-field`>s.

On frames, there are two **generic functions** defined: `parse-frame` accepts a byte vector and a frame type, and creates an instance of that frame type, providing structured access to the underlying data; `assemble-frame` provides the opposite functionality of turning a high level frame representation into a byte vector again. Additionally, it is possible to directly create frame instances using `make`, or using the `read-frame` generic function, which parses the human readable representation of certain leaf frames such as MAC or IP addresses.

2.2.2 Frame Translation

Looking at frames, we found that these come in two variants. Some just represent themselves using a subclass of <`frame`>, such as again a MAC address, we call those **untranslated frames**. Some other fields have representations as high level objects that already exist in the language, namely integer values and strings. There are many possible low-level representations for these two types, an integer could be represented as an unsigned two byte value in litte endian byte order, or a four byte twos complement big endian integer; a string could be zero-terminated or preceded by a character count. Obviously, it is desirable to specify the encoding of integers and strings only once, and have the framework take care of calling the necessary conversion functions at the appropriate time. For this purpose, we have introduced **translated frames**, which provide the mechanism for doing so. In theory, both leaf frames and container frames could be translated and untranslated, our current implementation doesn't allow translated container frames yet, though.

To support translated frames, we introduced two more abstract subclasses of <`frame`>, called <`translated-frame`> and <`untranslated-frame`>. This distinction is orthogonal to the distinction between leaf and container frames, concrete subclasses of frame are required to inherit exactly one from both of the translated/untranslated and leaf/container frame superclass pairs. Additionally, we had to introduce the `assemble-frame-as` generic function: since translated frames have a high level representation that is no longer a subclass of <`frame`>, we need to explicitly pass the type of the desired low-level representation. This corresponds to saying e.g. "turn this integer into a byte vector, using a

four byte two's complement signed representation in big endian".

Generic functions in Dylan provide the polymorphism. For every combination of types of the objects passed to a generic function, the appropriate **method** is selected. Implementation of the framework thus requires to write methods for all the subclasses of frames outlined above. Leaf frames turn out to be pretty straightforward to implement, the challenge is writing the parser and assembler functions for container frames. Given that a container frame consists of a list of fields, which in turn have again frames as their values, it is possible to write a straightforward recursive implementation of both `parse-frame` and `assemble-frame`.

This straightforward implementation already adds some interesting design problems. Most leaf frames know about their size, a MAC address is always six bytes long. We can thus parse an Ethernet frame by calling `parse-frame` for a MAC address at byte vector offset 0 to get the destination address field value, and then again at offset 6 to get at the source address field value. We could even go as far as noting that the destination address is always at offset 6, we're thus able to just skip parsing the destination address if all that we care about is the source address. However, some frames don't have a fixed size, and some fields don't have start offset that's equal to the next byte after the end offset for the preceding field. The payload of an IP frame is a good example for both: the length of the payload obviously varies, and the start offset of the payload is computed using the value of the header length field of the IP frame.

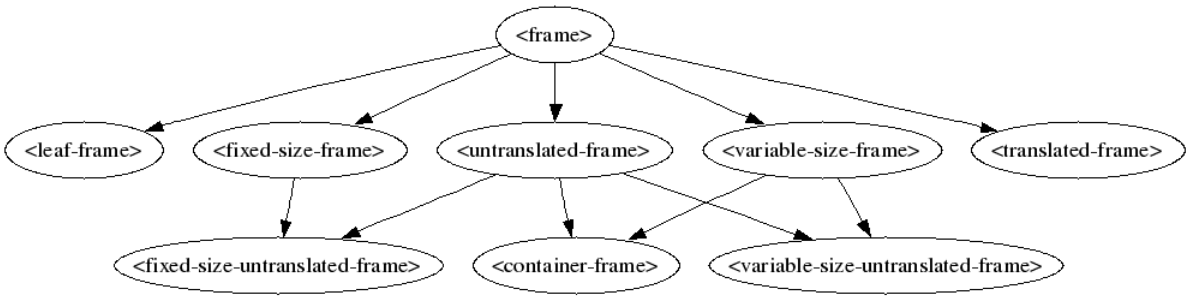

Figure 1: Frame class hierarchy

2.2.3 Fixed and Variable sized Frames

We model the distinction between fixed and variable size frames using a further set of mutually exclusive abstract superclasses: <`fixed-size-frame`> and <`variable-size-frame`>. Whether or not a field has a fixed offset is not represented using a type, though. The reason is that this sometimes requires computation to discover. We know that the payload in an IP packet has a dynamic start offset, since we specify a function computing it from a header value in our protocol definition, but for the case of a field following a variable size field, we don't know without walking over the list of preceding fields. Thus, we remember in every <`field`> instance the fixed offset if known, and compute that at program startup time. For efficiency reasons, a part of that computation is written in a way that allows the compiler to evaluate the computation expression at compile time. For common cases, like the offset of the source address in an Ethernet frame, the offset is already known to be 6 at compilation.

	fixed size	variable size
translated	2bit-unsigned-integer	null-terminated string
untranslated	ipv4-address	raw-frame

Table 1: Leaf frame examples

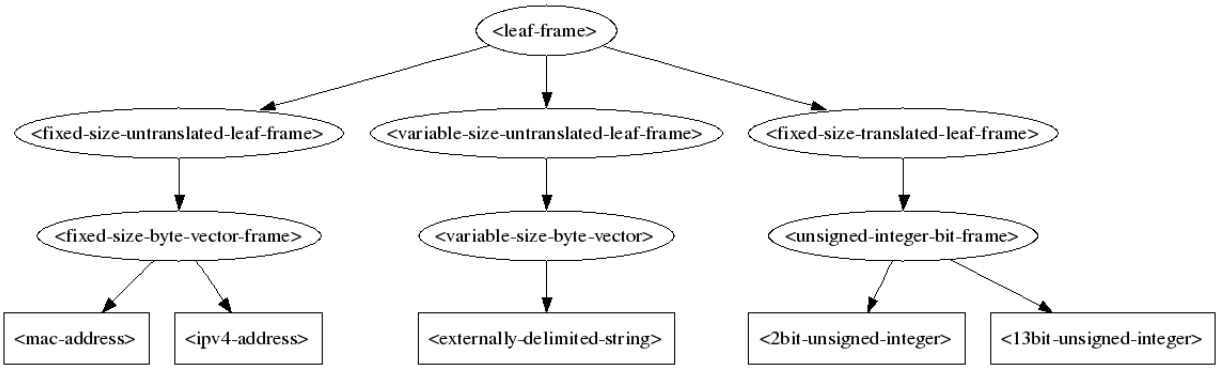

Figure 2: Leaf frame class hierarchy
The class hierarchy of <`leaf-frame`> is shown. Some leaf frames share a common algorithm for parsing and assembling, for those we introduced common superclasses. A <`fixed-size-byte-vector-frame`> always contains a byte vector of a compile-time defined length (an <`ipv4-address`> always has four bytes). An <`unsigned-integer-bit-frame`> has a fixed size and translates to a limited(<`integer`>, min: 0, max: $2^{(specified\ size)} - 1$).

2.2.4 Variable size fields

A further complication arises from the different ways that protocol specifications use for communicating the size of a field if it is not fixed. There are two cases that can be distinguished: a field can be externally delimited, in which case the parser for this field is told how many bytes to consume, or it can be internally delimited, in which case the parser knows when to stop from the information it reads and passes the number of consumed bytes back to the caller. To handle both, our parsing framework passes information both ways during recursive calls.

2.2.5 Lazy Parsing and Container Frame Representations

Unfortunately, the naive recursive implementation of parsing and assembling isn't terribly efficient. It is absolutely not necessary to parse the destination address of an ethernet frame if one is only interested in the source address. Generally speaking, it is completely sufficient to calculate start and end offset of a field, and parse just that. Also, assembling a frame by assembling all the fields and then concatenating the generated byte vectors comes with the overhead of allocating a lot of small vectors, and copying the contents of the small vectors into a big vector. It would be much more efficient to calculate the total length of the frame, allocate a single vector, and assemble right into that.

To make this work, some more changes to the representation are needed. For every class representing a container frame, we actually generate two subclasses, which inherit from the

new abstract superclasses <unparsed-container-frame> and <decoded-container-frame>. An <ethernet-frame> is now abstract, and comes with the two concrete subclasses <unparsed-ethernet-frame> and <decoded-ethernet-frame>. Decoded frames have a slot for every field of the frame, in which the high level representation for the frame corresponding to the field is stored. Decoded frames can be incomplete, some fields may have the value #f for false, or not known. Unparsed frames simply associate a byte vector with a decoded frame, which is used as a cache. For every field, getter functions are generated. These functions check in the cache whether the value has already been computed. Otherwise, start and end offset are computed and parse-frame is invoked on the corresponding byte vector subsequence. parse-frame on a <container-frame> is now a constant time operation, since it does nothing but creating an instance of <unparsed-container-frame> and storing a reference to the underlying byte vector.

Real work is done when accessing the fields, but lazily: the algorithm keeps track of start and end offsets for fields using a <frame-field>. If computation of an offset requires parsing of another field, such as the header-length in an IP frame to compute the start of the payload, this is done recursively until a field with a fixed offset is encountered.

Assembly also benefits from the improved offset calculation framework. The length of the frame can be calculated in advance, and then an appropriately sized vector can be allocated, and passed to the actual assembly functions.

2.2.6 Frame Inheritance

Sometimes it might be desirable to form a further inheritance structure for container frames. For instance, we introduced a subclass of <container-frame> called <header-frame>. A **header frame** is a frame that ends in a payload field, which is typical for layering protocols. This is used in higher level code to traverse all layered protocols, such as in the typical case of a DNS frame in a UDP frame in an IP frame in an Ethernet frame.

The other application of frame inheritance is for cases in which a number of different frames share structure, and also belong to the same abstract type. An example is again IP options. There are different kinds of IP options, but they all start with the same type identifier field. When implementing that, we introduce a frame <ip-option> with an ip-option-type field. Actual instances of IP options, such as a <record-route-ip-option>, inherit from this frame. This prepends the list of fields with the fields defined in the parent frame, and makes the Dylan high level representation of that frame a proper subclass of the parent frame type.

The repeated field ip-options in an IP frame now just specifies the abstract parent frame <ip-option> as the frame type of its elements. A method of parse-frame specialized on that type then does the actual dispatch from the value in the ip-option-type to the concrete type of the IP option frame being parsed. parse-frame is then called recursively with the computed type.

2.2.7 Assembly Support

Assembling a frame sometimes requires further computation after the frame has been assembled into a byte vector, because things like offsets of fields might not be known yet when a fields representing this offset needs to be emitted. For this purpose, fields support a fixup function. All defined fixup functions are called after the first stage of packet assembly is done. Furthermore, a global fixup function is definable for every frame, which is used for purposes like CRC computation.

Some fields never vary, and are always filled in with the same constant. For them, a default value can be specified. The user doesn't need to provide a value for this field when constructing a high level object, and the default value is subsequently used during assembly. If the user specifies a value for such a field, his value overrides the default value.

2.2.8 Definition Macro

Dylan macros work by doing pattern matching on the abstract syntax tree. This can be seen as adding additional rules to the grammar of the language. A Dylan macro call thus typically looks like any other Dylan statement. Actual Dylan code can be written anywhere a macro definition allows that, and we frequently use this feature in our protocol definitions.

For definition of container frames, we chose the form of a definition macro. A call of this macro begins with the keywords `define protocol`, followed by the name of the defined container frame (without angle brackets), then the parent frame in parentheses. After that, an optional summary and a semicolon-separated list of field definitions follows. The definition macro is terminated by an `end` keyword.

The summary field starts with a `summary` keyword, followed by a format string, and a comma-separated list of functions to produce the format arguments when called with the frame to be printed. Since all field definitions produce getter functions, which are first class objects, one can simply write field names here.

Field definition statements start with either `field`, `repeated field` or `variably-typed-field`, depending on whether a single, repeated or variably typed field is defined. Following that is the name of the field. Single fields then have a static frame type definition, which consists of `::` followed by the frame type of that field. Repeated fields also have a type definition with the same syntax, but a slightly different meaning: the type specifies the superclass of each element in the list. Variably-typed fields don't have a static type definition. Next for all fields is an optional default value, using `=` followed by the value.

All fields might have additional information, which are written down as pairs of keywords and Dylan expressions.

Variably-typed fields must have a keyword `type-function:`, followed by a Dylan epression, in which the variable `frame` is bound to the frame for which the type value is computed. The expression is supposed to return the computed type.

Repeated field statements must contain one of the following keywords: `count:` or `reached-end?:` to distuingish between the different kinds of repeated fields, and to specify a condition to detect the end of a repeated field. In the subsequent expression of the `count:` keyword the variable `frame` is also bound to the frame being parsed. The expression is supposed to return the number of frames in the repeated field. `reached-end?:` expects a method that takes one argument and returns a boolean. This method gets called each time a frame of the repeated field is parsed, and returns true on the final element.

All field definitions support keywords to override the default computation of start offset and field length. The default case assumes that the first field starts at offset zero, fields follow one another directly without padding, and that the length is either statically known for the frame type in the field or requires actual parsing of that frame to determine. This behavior can be changed using the `start-offset:`, `end-offset:` and `length:` keywords, which are again followed by a Dylan expression binding the variable `frame` and returning the appropriate value. If those values are statically known, the equivalent keywords `static-start-offset:`, `static-end-offset:` and `static-length:` are used, followed by the appropriate value. Since any of the three values can be computed from the other

two, providing all three variants is a convenience feature, only a maximum of two of them have to be actually specified.

All field definitions also support the keyword `fixup:`, which is evaluated when assembling is done. The succeeding Dylan expression also gets a binding to the frame via the variable `frame`. The result of the expression is then filled in into the appropriate field.

Listing 1: Specification of an ethernet frame

```
1  define protocol ethernet-frame (header-frame)
2    summary "ETH %= -> %=/%s",
3      source-address, destination-address, compose(summary, payload);
4    field destination-address :: <mac-address>;
5    field source-address :: <mac-address>;
6    field type-code :: <2byte-big-endian-unsigned-integer>;
7    variably-typed-field payload,
8      type-function:
9        select (frame.type-code)
10         #x800 => <ipv4-frame>;
11         #x806 => <arp-frame>;
12         otherwise => <raw-frame>;
13       end;
14 end;
```

An example usage of the domain-specific language is the definition of an ethernet frame. It starts with a summary section. Then it contains of four fields, `destination-address` and `source-address`, which have a static type, `<mac-address>`. `type-code` is a `<2byte-big-endian-unsigned-integer>`, and finally `payload`, a variably typed field, which has a type-function dispatching on the `type-code` field.

2.2.9 Security

One of the initial reasons to start working on frame representations is the perceived lack of security in handling network packets in existing applications. There is a myriad of ways to create malformed packets, and a naive implementation of a parser is prone to data reference failures while attempting to dissect such a broken packet. Unfortunately, in practice many data reference failures lead to vulnerabilities, manifesting as data leaks, denial of service problems or even remote code execution. The best common practice of manually checking packet validity has proven to be insufficient: many IP analysis tools are regularly plagued by newly discovered vulnerabilities, and even well-tested and widely used IP stacks have their occasional bug.

Preventing such vulnerabilities was one of the design goals of our implementation. Fortunately, data reference failures are easily detected by using bounds checking. When bogus packet data leads to an offset calculation that points outside of the packet being parsed, subsequent access to the array holding the packet data will trigger an out-of-bounds exception, and the packet can be safely discarded. The easy design decision to use bounds checked arrays for holding packets already eliminates all remote code execution vulnerabilities, denial of service from crashing applications, and data leak problems except for intra-packet data leaks.

But we can do better than that. Since we have an offset calculation framework, we can determine the start and end offset for a given field in a frame. When parsing a field

in a frame, a subsequence is created, which is a bounds-checked array slice aliasing into the original sequence. That way, a parser for a frame is unable to access bytes outside of its field, and a number of intra-packet data leak problems are automatically gone.

A further gain of the offset calculation framework is that it is possible to detect situations where the end offset of a field doesn't match the start offset of the following field. Two situations can arise: there might be a gap between the fields, or the fields could overlap. Both situations are interesting from a security point of view: gaps represent data hiding opportunities, overlaps indicate a malformed packet that try to trick some protocol parser into parsing the same bytes using different interpretations, or even an attempt to exploit an overrun situation or a data leak problem. Since we use the information we have on start and end offsets to construct bounds-checked subsequences, we already discover the case of a repeated field overlapping with a subsequent field, and reject the frame accordingly. We're working on discovering and rejecting all types of overlaps, and an infrastructure to universally represent gaps in protocols.

3 Filter Language

A useful feature when handling network packets of any kind is the ability to filter them according to some criteria. For many use cases, it is also desirable to have a description language for building such filters available. Since we know about the structure of frames from our definitions, it was easy to write code that generically checks for the presence of a certain frame type, or which checks whether a field has a certain value.

We then specified a language for filters using a token and LALR parser grammar. A parser is generated from that using the "monday" Dylan parser generator tool.

Our filter language knows two kinds of basic rules:

- presence of a frame (by protocol-class name, "tcp" for example)
- value of a field in a frame ("tcp.destination-port = 80")

Rules can be combined using logical operators:

- & logical and
- | logical or
- ~ logical negation

Rules are surrounded by parenthesis when used with operators. An example would be "(udp.destination-port = 53) | (udp.source-port = 53)" for filtering all packets from or to udp port 53.

We defined a generic function `matches?` which accepts a rule and a frame instance as arguments. It checks whether the frame matches the rule, and returns `#t` in that case. `matches?` on operators delegates `matches?` to the operands and combines the result with the appropriate logic operation.

`matches?` behaves special when the frame being passed is an instance of <`header-frame`>: If the rules don't match on the frame itself, the rules are recursively checked on the payload.

4 Flow graph

Structured processing of packets requires more infrastructure than mere being able to generate and understand them. Depending on the application, packets need to be bridged,

routed, printed, filtered, or queued. Generally speaking, there are different stages of processing packets, and packets flow on their way through the stack through these stages. Following the ideas found in the "click modular router" framework, we decided to model packet flow as a graph through nodes, in which nodes do the processing, and packets flow along the edges.

Every node in our flow graph inherits from the abstract superclass <`node`>, and provides zero or more inputs and outputs, which are general instances of the classes <`input`> and <`output`>. Every output, unless being left unconnected, is connected to an input using the `connect` function.

Connections between inputs and outputs can either be of push or pull type. The difference lies in the control flow of code execution. In a push type connection, a packet being sent along the connection triggers a function call on the receiving node. Code execution in the sending node is suspended until the receiving function returns. In a pull type connection, control flow is reversed: the receiving node calls a function on the sending node, which blocks until the sender has a packet available.

A fundamental node for doing meaningful work is the <`ethernet-interface`>. It abstracts away the details of sending and receiving raw frames from physical Ethernet interfaces. Packets being received on the interface are parsed as <`ethernet-frame`>s and passed on to the output. Frames passed to the input of this nodes are assembled into a byte vector, and then transmitted on the interface.

Another useful node is the <`summary-printer`>. It has a single input only, packets received on that input are printed to a stream, using the `summary` function automatically generated from the protocol definition.

From these two nodes, a basic sniffer application can be constructed. When run, a single line summary is printed for every packet received on the interface.

The source code for constructing and running the graph looks like this:

Listing 2: Simple Sniffer

```
15 let interface = make(<ethernet-interface>, name: ''eth0'');
16 connect(interface, make(<summary-printer>, stream: *standard-output*));
17 toplevel(interface)
```

In the first line an <`ethernet-interface`> instance is created, which binds in this case to the physical interface with the name 'èth0", the first network card on Linux. In the next line the interface is connected to to a freshly created instance of <`summary-printer`>, printing to `*standard-output*`. Afterwards `toplevel(interface)` is called, which is the main loop reading packets from the OS, and passing it down the output.

Here is a slightly more complex example, a sniffer with filter:

Listing 3: Filtering Sniffer

```
18 let interface = make(<ethernet-interface>, name: ''eth0'');
19 let frame-filter = make(<frame-filter>, frame-filter: ''dns'');
20 connect(interface, frame-filter);
21 connect(frame-filter, make(<verbose-printer>, stream: *standard-output*));
22 toplevel(interface)
```

This graph includes a <`frame-filter`>, which is also a single-push input and single-push output node. It accepts a filter expression on creation, which uses the filter language we described earlier. When receiving a frame on the input, it checks whether the frame matches the filter rule, and drops the packet on no match, or pushes it on the output on match. In this example, only dns packets are printed. Also, a <`verbose-printer`> is

used, which prints a detailed view of the frame by printing all the fields recursively.

Other useful nodes we implemented include:

<pcap-reader>	read packets from a PCAP file
<pcap-writer>	write packets to a PCAP files
<decapsulator>	strip header from a <header-frame>
<demultiplexer>	keeps a list of outputs and associated filters. Packets on input are passed on to the outputs with matching filters.
<queue>	provides push inputs and a pull output for combining push and pull nodes. Frames are stored in a FIFO buffer.
<fan-in>	provides multiple push inputs and one push output Frames received on one of the inputs is pushed to the output.
<fan-out>	provides one push input and multiple push outputs. Each frame received on the input is pushed to all connected outputs.

5 Protocol Stacking

A flow graph works fine for applications like packet forwarding or analysis. For implementation of a protocol stack, a further abstraction for handling stacking of protocols is desirable, though. One wants to be able to easily specify that a certain protocol handler instance is run on top of another one. Such a protocol handler encapsulates not only the packet flow, but also the state and configuration required to actually implement a certain protocol.

Our protocol handlers are implemented as subclasses of the abstract class <layer>. A typical configuration of a network stack could have an Ethernet layer on the bottom, an IP layer on top of that, followed by a UDP layer, and finally a DNS layer. At any time, an additional Ethernet layer for another interface could be plugged below the IP layer, or another high level protocol handler above the UDP layer.

The design of our layering model is still to be considered work in progress. What we have come up so far already provides some good abstraction, however, implementation of real world protocols still revealed cases where abstraction boundaries are violated. The following presentation should be considered a snapshot of the design process.

In our current design, connections between layers are handled using a data structure which for lack of a better name we called <socket>s. A layer that wants to stack itself on top of another layer requests a socket from the layer below, specifying the necessary information for multiplexing and demultiplexing the packets received and sent. An IP layer would for instance ask for the payload of Ethernet frames with a **type-code** of **0x800**, a DNS layer would request a socket from the UDP layer for **destionation-port** of 53, and so forth.

Of course, one doesn't want a layer to know about the details of the layer below. A protocol handler should be written in a way that's independent of the transport layer it is running on. To isolate a protocol handler from transport details, we added adapters, which encapsulate the specific knowledge of running a certain protocol on top of another one. A typical example is the <ip-over-ethernet-adapter>, which not only knows that IP frames in an Ethernet frame have a **type-code** of **0x800**, but also handles the ARP address resolution protocol required to successfully deliver the packet. An <ip-layer> now just maintains a list of adapters to talk to the lower layers.

Our <ethernet-layer> consists of a number of flow graph nodes. An <ethernet-interface>

is used as the source and sink of actual packets being received and sent. The output of that is connected to a <demultiplexer> node. Every demultiplexer output is then connected via a <decapsulator>, which strips the Ethernet header, to the socket that handles the connection to the layer above. In the other direction, an input maintained by the socket is connected to an <encapsulator>, which adds an Ethernet header to a packet. The encapsulator output is connected to a <fan-in>, which in turn is connected via a <queue> to the <ethernet-interface>.

When a socket is created on an <ethernet-layer>, the user has to specify a value for the **type-code** field, and optionally a MAC address which is then used instead of the default MAC address. From the provided information, a filter is generated and added to the demultiplexer for the receiving side, and an encapsulator for filling in the Ethernet header of frames being sent is created as well.

On the receiving side, the layer that requested a socket from below just connects the output of the socket to an input of its own graph. The sending side is more complicated however, since the socket might represent a point-to-multipoint connection, and additional out-of-band information about the actual destination address of a packet needs to be passed on too. We can't use a simple flow graph connection here, since we can't pass on out-of-band data. Thus, we introduced a generic function `send`, which accepts a frame, a socket and a destination address as arguments.

The `send` API is also used for adapters. If the IP layer wishes to send a packet, it would look up the appropriate adapter in its forwarding table, and call `send` with the packet to send, and the next-hop IP address. In case of an <ip-over-ethernet-adapter>, the adapter then looks up the MAC address of the destination using ARP, and passes on the packet to the Ethernet layer by calling `send` on the socket, passing the packet and destination MAC address.

6 Conclusion

We presented and implemented several modules to solve different problems of writing a TCP/IP stack: A framework for parsing and assembling protocols, a filter language, a flow graph, and a work-in-progress stacking mechanism.

We have reached our design goal of security in terms of security against vulnerabilities that result from data reference failures. In other words, we're confident that our code will never have any vulnerability that leads to code execution. As described, we're working on detecting certain malformed packets and data hiding opportunities for extended security requirements.

To achieve a high performance, we have chosen a compiled programming language, and worked to make sure all parsing is done lazily. Future work on performance issues will include profiling to isolate hot spots, and usage of procedural macros to move offset computations to compile time.

Also, our design decision to put all demultiplexing for a layer into a single place leads to a further optimization opportunity. The filter rules can be translated into state machines, which can then be unified for all filters applicable on a certain layer. Also, there is the opportunity to offload parts of the filtering into hardware, if applicable, without having to touch any high level code.

We have mostly succeeded in our goal of compact code representation without any needless information duplication. Improvements are still possible in the packet definition: the relationship between for instance the `header-length` and the start offset of the payload in an IP header definition is annotated twice: once for computing the start offset from

the header length for parsing purposes, and once for the fixup function that fills in the correct field value during assembly. Automatically deriving both equations from a single definition would be desirable. Also, the relationship between type codes and payload types is sometimes still scattered around the code.

For a full implementation of TCP/IP, some parts are still missing, such as a good abstraction for stateful protocols.

We intend to continue working on our framework, in order to make it a secure, universal tool for all kinds of networking requirements.

7 References

H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software", in Proceedings of OOPSLA 1995, (Austin, Texas), ACM, October 1995

P. Biondi,"Scapy, a powerful interactive packet manipulation program", http://www.secdev.org/projects/scapy/

E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek,"The Click modular router", in ACM Transactions on Computer Systems 18(3), pages 263-297, August 2000

W. Richard Stevens, "TCP/IP Illustrated, Volume 1 The Protocols", Addison-Wesley, 1994

Dylan, http://www.opendylan.org

Source code,
http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/libraries/packetizer
http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/libraries/flow
http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/libraries/network-flow
http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/libraries/sniffer
http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/libraries/layer